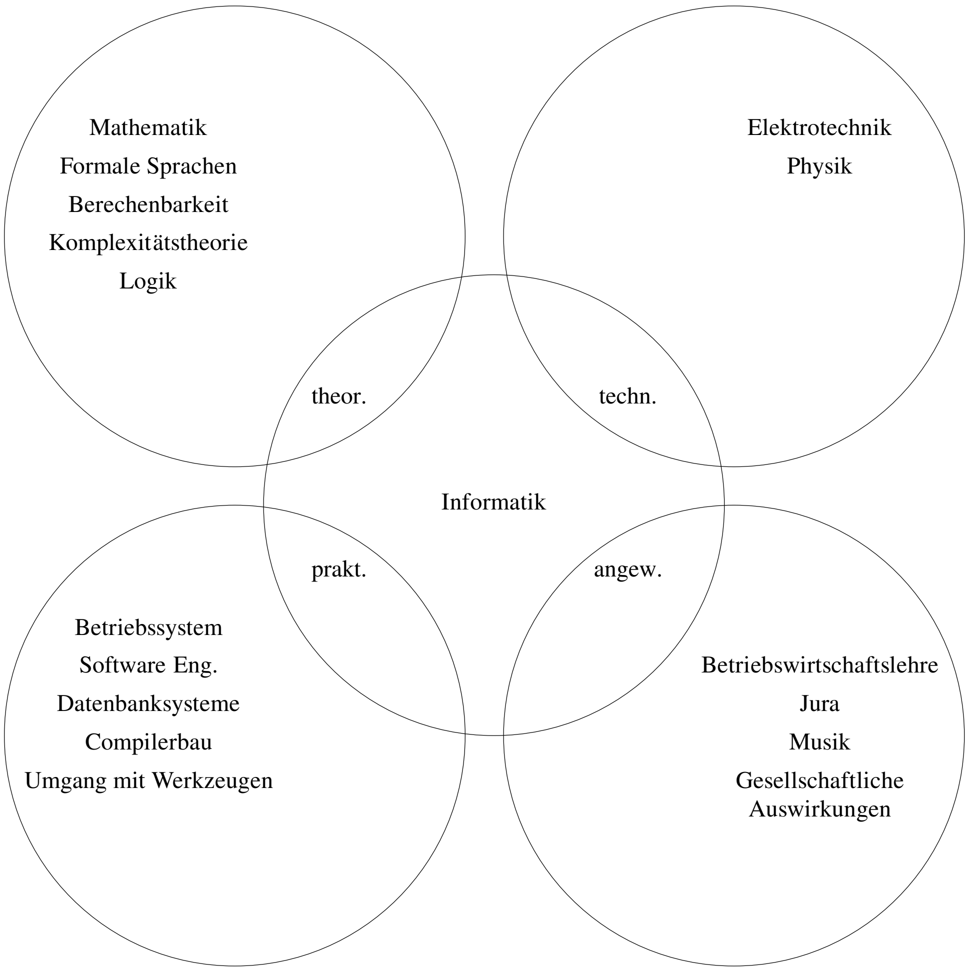
1. Einführung

1.1. Informatik

Wissenschaft von der EDV
Konzepte, unabhängig von Technologie
Formalismus
Interdisziplinärer Charakter



1.2. Algorithmus, Programm, Prozeß

Eine endlich lange Vorschrift, bestehend aus Einzelanweisungen, heißt *Algorithmus* .

Telefonieren:	Hörer abnehmen Geld einwerfen wählen sprechen auflegen
Kochrezept:	Man nehme ...
Bedienungsanleitung:	Mit Zange Z die Schraube S auf Platte P festziehen.

Durchführender kennt Einzelanweisungen; deterministisch, nicht zufällig.

Endliche Vorschrift heißt **nicht** endliche Laufzeit.

aber: Vorschrift muß endlich sein.

hier: Elementaranweisungen müssen vom Computer verstanden werden.

Programm	$\rightarrow$	Maschinenprogramm
`IF a > b THEN`	Compiler	`011011101110110111`

Ein für den Compiler formulierter Algorithmus heißt *Programm* .

Ein Programm in Ausführung heißt *Prozeß* .

1.3. Anweisungen, Ablaufprotokoll

Elementare Anweisungen

Beispiel: teile x durch 2
erhöhe y um 1

strukturierte Anweisungen

enthalten Kontrollstruktur, Bedingung, Teilanweisungen.

Beispiele:

```
WENN es tutet
   DANN waehle
   SONST lege auf

SOLANGE besetzt ist WIEDERHOLE
                       auflegen
                       waehlen
```

Beispiel für einen Algorithmus in umgangssprachlicher Form

```
1 lies x
2 setze z auf 0
  SOLANGE x ≠ 1 WIEDERHOLE
3   WENN x gerade
        DANN halbiere x
        SONST verdreifache x und erhoehe um 1
4   erhoehe z um 1
5 drucke z
```

Ablaufprotokoll (*trace*)

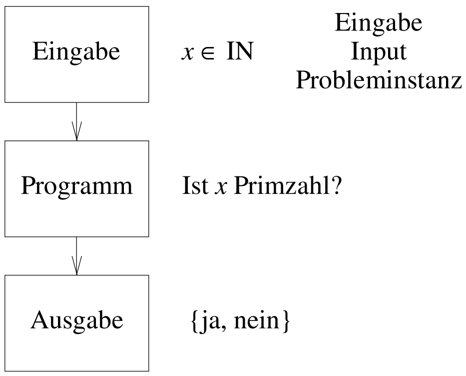
Zeile	x	z
	undef.	undef.
1	3	undef.
2	3	0
3	10	0
4	10	1
3	5	1
4	5	2
3	16	2
4	16	3
3	8	3
4	8	4
3	4	4
4	4	5
3	2	5
4	2	6
3	1	6
4	1	7
5	Ausgabe	7

Programm iteriert Collatz-Funktion

$f : \mathrm{IN} \to \mathrm{IN}$

$f(x)$ = Anzahl der Iterationen, um x auf 1 zu transformieren

Typisch: Programm löst Problem



Terminierung, Korrektheit und Effizienz sind nicht algorithmisch zu bestimmen. Dafür ist jeweils eine neue Idee erforderlich.

```
(***********************************************************************************)
(*      Datum:  18.10.93  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  collatz.m2                                                       *)
(*              berechnet collatz-Funktion, d.h.                                 *)
(*              Anzahl der Iterationen der Funktion g:N->N                       *)
(*              bis die Eingabe auf 1 transformiert ist                          *)
(*              mit g(x) = x/2 falls x gerade, 3*x+1 sonst                       *)
(***********************************************************************************)

MODULE collatz;

FROM InOut IMPORT ReadInt,WriteInt,WriteString,WriteLn; (* importiere I/O-Routinen      *)

VAR x,z : INTEGER;                                      (* deklariere 2 Integer-Zahlen  *)

BEGIN

        WriteString("Bitte Zahl eingeben: ");           (* schreibe einen String        *)
        ReadInt(x);                                     (* lies Zahl ein und weise x zu *)

        z:=0;                                           (* setze z auf 0                *)

        WHILE x <> 1 DO                                 (* solange x ungleich 1 ist     *)
                IF NOT ODD(x)                           (* falls x nicht ungerade       *)
                        THEN x := x DIV 2;              (* teile durch 2                *)
                        ELSE x := 3*x + 1               (* nimm mit 3 mal und addiere 1 *)
                END;                                    (* Ende der Fallunterscheidung  *)
                INC(z)                                  (* erhoehe z um 1               *)
        END;                                            (* Ende der While-Schleife      *)

        WriteInt(z,6);                                  (* schreibe z auf 6 Stellen     *)
        WriteString(" Iterationen wurden benoetigt.");  (* schreibe einen String        *)
        WriteLn()                                       (* Zeilenvorschub               *)

END collatz.
```

2. Modula

Ein Modula-2-Programm ist ein Modul.
Ein Modul besteht aus:

- dem Wort `MODULE`, gefolgt vom Modulnamen und `";"`
- der Importliste
- den Deklarationen
- dem Wort `BEGIN`
- einer Folge von Anweisungen
- dem Wort `END`, gefolgt vom Modulnamen und `"."`

Die *Syntax* einer Sprache legt fest, welche Satzformen (= Programme) zulässig sind.

Die *Semantik* einer Sprache legt fest, welche Bedeutung die zulässigen Satzformen haben.

Syntax: `z := 0;`   Semantik: Setze *z* auf 0

Eine Methode zur Formulierung der Syntax ist der *Erweiterte Backus Naur Formalismus* (EBNF).

2.1. EBNF

Menge von Regeln, bestehend aus

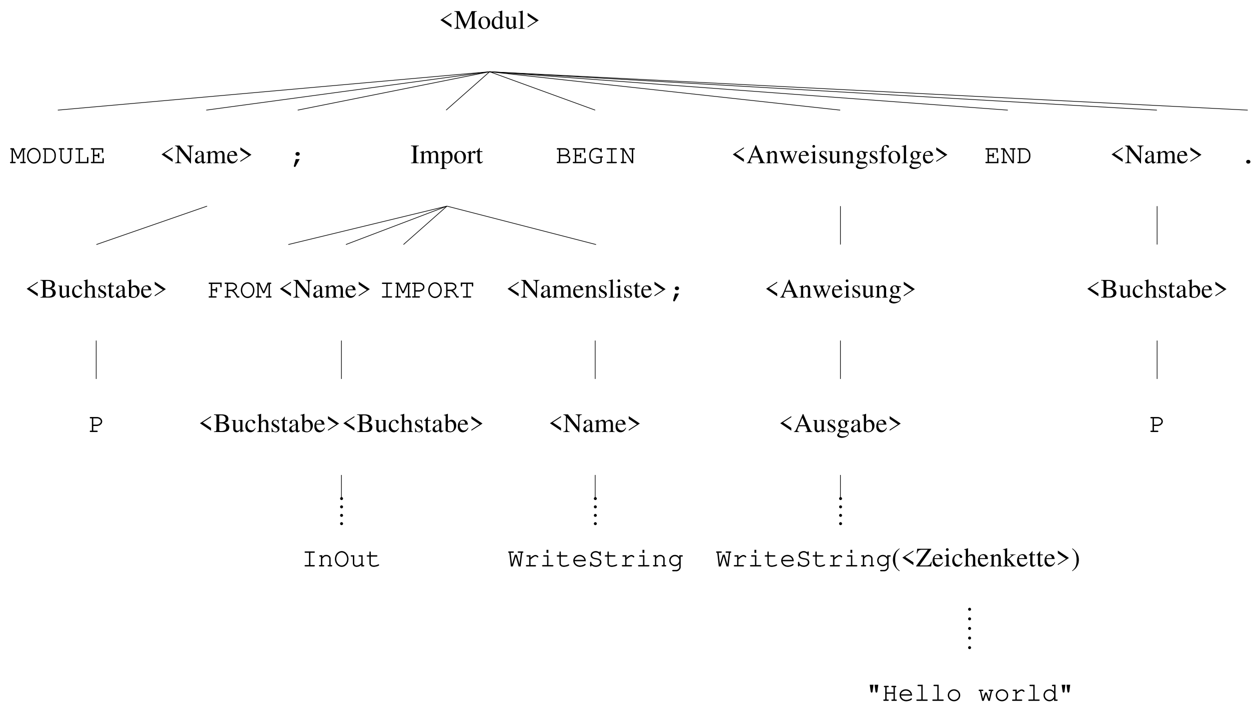
Terminalsymbolen	z.B. `BEGIN END ; a 3 7`
Nichtterminalsymbolen	z.B. `<Name> <Ziffer>`
Metasymbole	`::= [] {} |`
Startsymbol	`<Modul>`

Syntaktisch korrekt gelten alle Folgen von Terminals, die aus dem Startsymbol abgeleitet werden können und einige verbal formulierte Zusatzbedingungen erfüllen.

Ein Nonterminal links von `::=` kann ersetzt werden durch den Ausdruck rechts von `::=`.

Ein Ausdruck in `[]` kann gesetzt oder weggelassen werden.
Ein Ausdruck in `{}` kann beliebig oft, auch gar nicht, gesetzt werden.
Ausdrücke getrennt durch `|` können alternativ gesetzt werden.

```
<Modul>                  ::= MODULE <Name>;
                             {<Import>}
                             {<Deklaration>}
                             BEGIN
                             [<Anweisungsfolge>]
                             END <Name>.
<Name>                   ::= <Buchstabe> {<Buchstabe> | <Ziffer>}
<Buchstabe>              ::= A | B | C | D | E | F | G | H | I |
                             J | K | L | M | N | O | P | Q | R |
                             S | T | U | V | W | X | Y | Z |
                             a | b | c | d | e | f | g | h | i |
                             j | k | l | m | n | o | p | q | r |
                             s | t | u | v | w | x | y | z
<Ziffer>                 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Beispiele für Namen:         i
                             E605
                             karl
                             KarlderGrosse
<Import>                 ::= FROM <Name> IMPORT <Namensliste>;
<Namensliste>            ::= <Name>{, <Name>}
<Anweisungsfolge>        ::= <Anweisung>{; <Anweisung>}
<Anweisung>              ::= <Ausgabe> | <Eingabe> | <Zuweisung>
<Ausgabe>                ::= WriteLn [()] |
                             WriteString (<Zeichenkette>) |
                             WriteInt (<Ausdruck>, <Ziffernanzahl>)
<Eingabe>                ::= ReadInt (<Name>)
<Zuweisung>              ::= <Variablenname> := <Ausdruck>
<Zeichenkette>           ::= "{<Zeichen>}" | '{<Zeichen>}'
Beispiele:                   "Aber hallo!"
                             'Gestatten: 007!'
<Ausdruck>     z.B.          (3 * y + 1) - 12
                             13 DIV 3
                             17 MOD 3
<Deklaration>            ::= VAR {<Variablendeklaration>;}
<Variablendeklaration>   ::= <Namensliste> : <Typ>
<Typ>                    ::= INTEGER | CARDINAL | REAL | BOOLEAN | CHAR
```



Eine konkrete Ableitung läßt sich als Baum darstellen; das Startsymbol bildet die Wurzel.

Ausdrücke rechts von `::=` bilden die Söhne des Ausdrucks links von `::=`.
Blätter mit Terminalsymbolen bilden das Programm.

Zusatzbedingungen

Die Namen hinter `MODULE` und vor `"."` müssen übereinstimmen.

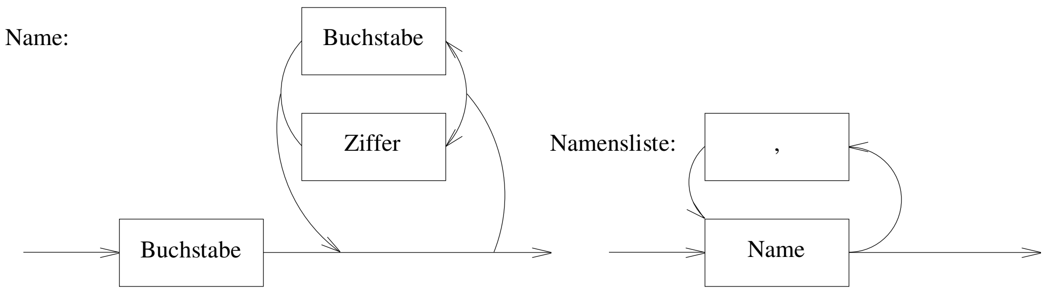
Als Namen dürfen keine reservierten Wörter verwendet werden, wie z.B.

```
BEGIN   INTEGER
END     VAR
FROM
```

Alle Ein-/Ausgabeprozeduren müssen importiert werden. Jede Variable muß durch Angabe ihres Typs und des Namens deklariert werden.
Kommentare können beliebig eingestreut werden:

```
(* jetzt geht's los *)
```

Statt EBNF auch Syntaxdiagramm möglich:



2.2. Kontrollstrukturen

```
(*************************************************************************************)
(*      Datum:  19.10.93  14:15 Uhr                                                  *)
(*      Autor:  Oliver Vornberger                                                    *)
(*      Modul:  if.m2                                                                *)
(*              demonstriert if-statements                                           *)
(*                                                                                   *)
(*              <Vergleichsoperator>::=  <  | <= | =  | >= | > | # | <>              *)
(*                                                                                   *)
(*              <logischer Operator>::= AND | OR | NOT                               *)
(*                                                                                   *)
(*              <if-Anweisung>      ::= IF <Bedingung>                               *)
(*                                      THEN <Anweisungsfolge>                       *)
(*                                      { ELSIF <Bedingung> THEN <Anweisungsfolge> } *)
(*                                      [ ELSE <Anweisungsfolge> ]                   *)
(*                                      END                                          *)
(*                                                                                   *)
(*************************************************************************************)

MODULE if;
FROM InOut IMPORT ReadInt, WriteInt, WriteString;       (* importiere I/O-Routinen      *)
VAR n, monat, x, y : INTEGER;                           (* deklariere 4 Integer-Zahlen  *)
BEGIN

        IF x < 0 THEN x := -x END;              (* setze x auf seinen Absolutbetrag     *)

        IF x > 0                                (* setze y auf den Absolutbetrag von x  *)
                THEN y := x
                ELSE y := -x
        END;

        IF (monat=12) OR (monat<=2)                              (*  12,1,2 = Winter    *)
        THEN    WriteString("Winter")
        ELSIF monat >= 9                                         (* 9,10,11 = Herbst    *)
        THEN    WriteString("Herbst")
        ELSIF monat >= 6                                         (*   6,7,8 = Sommer    *)
        THEN    WriteString("Sommer")
        ELSE    WriteString("Fruehling")                         (*   3,4,5 = Fruehling *)
        END;

END if.
```

```
(********************************************************************************)
(*      Datum:  19.10.93  14:15 Uhr                                             *)
(*      Autor:  Oliver Vornberger                                               *)
(*      Modul:  schleife.m2                                                     *)
(*              demonstriert  for-, while- und repeat-Schleife                  *)
(*                                                                              *)
(*              <for-Anweisung>     ::= FOR <Name> := <Ausdruck> TO <Ausdruck>  *)
(*                                      [ BY <Konstantenausdruck> ]             *)
(*                                      DO <Anweisungsfolge> END                *)
(*                                                                              *)
(*              <while-Anweisung>   ::= WHILE  <Bedingung> DO <Anweisungsfolge> END  *)
(*                                                                              *)
(*              <repeat-Anweisung>  ::= REPEAT <Anweisungsfolge> UNTIL <Bedingung>  *)
(*                                                                              *)
(*                                                                              *)
(********************************************************************************)

MODULE schleife;
FROM InOut IMPORT ReadInt,WriteInt,WriteLn,WriteString; (* importiere I/O-Routinen      *)
VAR i, x, y, summe, monat : INTEGER;                    (* deklariere 5 Integer-Zahlen  *)
BEGIN

        FOR i := 1 TO 100 DO                            (* schreibe die ersten 100      *)
                WriteInt(i*i,6);                        (* Quadratzahlen, jede Zahl     *)
                WriteLn                                 (* auf eine Zeile               *)
        END;


        WHILE x > 0 DO                        (* jeweils zuerst die Bedingung auswerten *)
                x := x - 1;                 (* fortfahren, falls die Bedingung zutrifft *)
                y := y * 2
        END;


        REPEAT
                x := x - 1;
                y := y * 2                     (* jeweils zuletzt die Bedingung auswerten *)
        UNTIL x <= 0;                          (* abbrechen, falls die Bedingung zutrifft *)


        WriteString("Bitte Zahlen eingeben. 0 als Abbruch: ");
        summe := 0;
        ReadInt(x);
        WHILE x <> 0 DO                 (* solange die eingelesene Zahl ungleich 0 ist  *)
                summe := summe + x;                           (* wird die Summe erhoeht *)
                ReadInt(x)                          (* und die naechste Zahl eingelesen *)
        END;
        WriteString("Die Summe betraegt ");
        WriteInt(summe, 6);
        WriteLn;

        REPEAT
                WriteString(" Bitte Monatszahl: ");
                ReadInt(monat)                                     (* solange einlesen *)
        UNTIL (1<=monat) AND (monat<=12);             (* bis Zahl im richtigen Bereich *)

END schleife.
```

```
(****************************************************************************************)
(*      Datum:  25.10.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  fakultaet.m2                                                            *)
(*                                                                                      *)
(*                                                                                      *)
(*              Sei     n!      :=      1               fuer n = 0                      *)
(*                                      1*2*3* ... * n  fuer n > 0                      *)
(*                                                                                      *)
(*                                                                                      *)
(****************************************************************************************)

MODULE fakultaet;
FROM InOut IMPORT WriteLn, WriteInt, WriteString;       (* importiere I/O-Routinen      *)
VAR i, n, fakultaet : INTEGER;                          (* deklariere 3 Integer-Zahlen  *)
BEGIN

        fakultaet := 1;                                 (* berechne n!                  *)
        FOR i:=1 TO n DO                                (* mit for-Schleife             *)
                fakultaet := fakultaet * i
        END;



        fakultaet := 1;                                 (* berechne n!                  *)
        i := 1;                                         (* mit while-Schleife           *)
        WHILE i<=n DO
                fakultaet := fakultaet * i;
                i := i+1
        END;



        fakultaet := 1;                                 (* berechne n!                  *)
        i := 1;                                         (* mit repeat-Schleife          *)
        REPEAT
                fakultaet := fakultaet * i;
                i := i+1
        UNTIL i > n;



        FOR n := 0 TO 10 DO                             (* fuer 0,1,2,3,4,5,6,7,8,9,10  *)
                fakultaet := 1;                         (* berechne n!                  *)
                FOR i := 1 TO n DO                      (* mit for-Schleife             *)
                        fakultaet := fakultaet * i
                END;
                WriteInt(n,6);                          (* gib n und n! aus             *)
                WriteString("! = ");
                WriteInt(fakultaet,9);
                WriteLn
        END

END fakultaet.
```

```
(**********************************************************************************)
(*      Datum:  19.10.93  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  ggt.m2                                                           *)
(*                                                                               *)
(*              ggt(x,y)  :=    groesster gemeinsamer Teiler von x und y         *)
(*                                                                               *)
(*                              x               falls x = y                      *)
(*              ggt(x,y)  =     ggt(x-y, y)     falls x > y                      *)
(*                              ggt(x, y-x)     falls y > x                      *)
(*                                                                               *)
(*                              denn: wegen x=t*f1 und y=t*f2 folgt (x-y) = t*(f1-f2)   *)
(*                                                                               *)
(*              Beispiel:       315     60                                       *)
(*                              255     60                                       *)
(*                              195     60                                       *)
(*                              135     60                                       *)
(*                               75     60                                       *)
(*                               15     60                                       *)
(*                               15     45                                       *)
(*                               15     30                                       *)
(*                               15     15                                       *)
(*                                                                               *)
(*              ggt(x,y)  =     x               falls y = 0                      *)
(*                              ggt(y, x mod y) sonst                            *)
(*                                                                               *)
(**********************************************************************************)

MODULE ggt;

VAR x, y, z, teiler : INTEGER;                      (* deklariere 4 Integer-Zahlen  *)

BEGIN


        (* Drei Methoden zur Bestimmung des ggt. Es werden x und y jeweils eingelesen:  *)

        teiler := x;                                          (* Initialisierung      *)
        WHILE (x MOD teiler <> 0) OR (y MOD teiler <> 0) DO   (* ineffiziente Methode *)
                teiler := teiler - 1;                         (* mit while-Schleife   *)
        END;                                                  (* Ergebnis in teiler   *)



        WHILE x<>y DO                               (* Methode nach Euklid:         *)
                IF x>y  THEN x := x-y               (* subtrahiere jeweils          *)
                        ELSE y := y-x               (* die kleinere Zahl von        *)
                END                                 (* der groesseren Zahl          *)
        END;                                        (* Ergebnis in x                *)



        WHILE y<>0 DO                               (* verbesserter Euklid:         *)
                z := x MOD y;                       (* ersetze mehrfaches Abziehen  *)
                x := y;                             (* durch einmalige Restbildung  *)
                y := z
        END;                                        (* Ergebnis in x                *)

END ggt.
```

```
(****************************************************************************)
(*      Datum:  24.10.95  14:15 Uhr                                          *)
(*      Autor:  Oliver Vornberger                                            *)
(*      Modul:  case.m2                                                      *)
(*              demonstriert case-statements                                 *)
(*                                                                           *)
(*              <case-Anweisung>    ::= CASE <Ausdruck> OF <Fall> {| <Fall> } *)
(*                                      [ ELSE <Anweisungsfolge> ]           *)
(*              <Fall>              ::= [ <Fallmarkenliste> : <Anweisungsfolge> ] *)
(*              <Fallmarkenliste>   ::= <Fallmarken> {, <Fallmarken> }       *)
(*              <Fallmarken>        ::= <Konstantenausdruck> [..Konstantenausdruck] *)
(*                                                                           *)
(****************************************************************************)

MODULE case;
FROM InOut IMPORT WriteString;                   (* importiere I/O-Routinen      *)
VAR n, monat : INTEGER;                          (* deklariere 2 Integer-Zahlen  *)
BEGIN

        CASE monat OF
        12,1..2 : WriteString("Winter") |
        9..11   : WriteString("Herbst") |
        6..8    : WriteString("Sommer") |
        3..5    : WriteString("Fruehling")
        ELSE    WriteString("undefiniert")
        END;

        WriteString("Die Zahl endet auf ");
        CASE n MOD 10 OF
        0:      WriteString("null")     |
        1:      WriteString("eins")     |
        2:      WriteString("zwei")     |
        3:      WriteString("drei")     |
        4:      WriteString("vier")     |
        5:      WriteString("fuenf")    |
        6:      WriteString("sechs")    |
        7:      WriteString("sieben")   |
        8:      WriteString("acht")     |
        9:      WriteString("neun")
        END;

END case.
```

2.3. Elementare Datentypen

Der Datentyp legt fest:

- Wertebereich
- Operationen.

Die Implementierung verlangt:

- Codierung.

elementar = von der Programmiersprache vorgegeben.

INTEGER

Wertebereich: ganze Zahlen von `MIN(INTEGER)` bis `MAX(INTEGER)`.

Codierung der positiven Zahlen in Dualzahldarstellung:

Sei

$$x = \sum_{i=0}^{n-1} d_i \cdot 2^i$$

Algorithmus dezimal → dual:

```
WHILE x <> 0 DO
   IF ODD (x) THEN Write ("1")
              ELSE Write ("0")
   END;
   x:= x DIV 2
END;
```

Obacht: Bits werden rückwärts generiert!

Codierung der ganzen Zahlen im 2-er Komplement:

d_3	d_2	d_1	d_0	x
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0
1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

Beispiel zur Berechnung des 2-er Komplements einer negativen Zahl:

Gegeben $-x$	`-4`
Finde d_i zu x	`0100`
Invertiere Bits	`1011`
Addiere 1	`1100`

Vorteil: Nur ein Addierwerk!

```
  0011    3
+ 1011   -5
----------
= 1110   -2
```

Subtraktion mittels Negierung auf Addition zurückführen. Obacht: Überlauf beachten!

```
  0111    7
+ 0001    1
--------------------
= 1000   -8   falsch
```

Trick: Vorzeichenbits verdoppeln, müssen nach der Verknüpfung identisch sein:

```
  00111             7          00011         3
+ 00001             1          11011        -5
--------------------          ---------------
  01000                        11110        -2
 Ergebnis undefiniert          Ergebnis ok!
```

Operationen

`+`	:	`INTEGER` × `INTEGER`	→	`INTEGER`	Addition
`-`	:	`INTEGER` × `INTEGER`	→	`INTEGER`	Subtraktion
`*`	:	`INTEGER` × `INTEGER`	→	`INTEGER`	Multiplikation
`DIV`	:	`INTEGER` × `INTEGER`	→	`INTEGER`	ganzzahlige Division
`MOD`	:	`INTEGER` × `INTEGER`	→	`INTEGER`	Modulo = Rest der Division
`-`	:	`INTEGER`	→	`INTEGER`	Vorzeichenumkehr
`ABS`	:	`INTEGER`	→	`INTEGER`	Betrag
`ODD`	:	`INTEGER`	→	`BOOLEAN`	Test auf ungerade

Konstantenbezeichner

123   +123   -123

CARDINAL

Wertebereich: ganze Zahlen von 0 bis `MAX(CARDINAL)`.

Bei n Bits : 0 bis $2^n - 1$
$n = 16$: 0 bis 65535

Operationen wie INTEGER

Auf negative Werte achten!

falsch:

```
i := n;
WHILE i >= 0 DO
   <action>
   i := i - 1
END;
```

richtig:

```
i := n + 1;
WHILE i > 0 DO
    i := i - 1;
    <action>
END;
```

INTEGER und CARDINAL nicht mischen

```
VAR i: INTEGER ;
VAR c: CARDINAL ;
i := c;
```

nicht erlaubt

```
i := i + INTEGER(c);
c := CARDINAL(i) + c;
```

erlaubt mit Hilfe von Typübertragungsfunktion

REAL

$$0,d_{-1}\,d_{-2}\,d_{-3}\,\cdots\,d_{-k} \text{ bedeutet } \sum_{i=-k}^{-1} d_i \cdot 2^i$$

Wertebereich: alle Zahlen x darstellbar als

$$x = \mathit{Mantisse} \cdot \mathit{Basis}^{\mathit{Exponent}}$$
$$0.25 \cdot 2^3 = 2$$

Normalisiert, wenn $\frac{1}{\mathit{Basis}} \leq |\mathit{Mantisse}| < 1$, d.h. für Basis = 2 $\Rightarrow \frac{1}{2} \leq |\mathit{Mantisse}| < 1$

Dualzahldarstellung für Mantisse:

½	`0.10`
¼	`0.01`
½ + ¼	`0.11`
⅛	`0.001`

Algorithmus dezimal → dual (sei `r < 1` gegeben)

```
WHILE TRUE DO
   r := r * 2.0;
   IF r >= 1.0 THEN Write ("1"); r := r - 1.0
               ELSE Write ("0")
   END
END
```

⇒ normalisierte Mantisse beginnt mit 0.1 ...

Mögliche Codierung bei $n = 32$ Bits:

1	Bit für Vorzeichen der Mantisse
23	Bits für Mantisse (⇒ 7 signifikante Dezimalstellen)
8	Bits für Exponenten im 2-er Komplement

Pos. Wertebereich von	$0.1 * 2^{10000000}$	bis $0.11111111111111111111111 * 2^{01111111}$
d.h. von	$\frac{1}{2} * 2^{-128}$	bis $\approx 1*2^{127}$
d.h. von	10^{-39} $(2^t = 10^{0.3t})$	bis 10^{38}
Neg. Wertebereich von	-10^{38}	bis -10^{-39}

Spezielle Darstellung für die Null.

Codierung bei $n = 64$ Bits

1	Bit	für Vorzeichen der Mantisse
52	Bit	für Mantisse (⇒ 16 signifikante Dezimalstellen)
11	Bit	für Exponent ⇒ Exponent von -1024 bis +1023 ⇒ Wertebereich von 10^{-308} bis 10^{308}

Operationen

`+`	:	`REAL` × `REAL`	→	`REAL`	Addition
`-`	:	`REAL` × `REAL`	→	`REAL`	Subtraktion
`*`	:	`REAL` × `REAL`	→	`REAL`	Multiplikation
`/`	:	`REAL` × `REAL`	→	`REAL`	Division
`ABS`	:	`REAL`	→	`REAL`	Betrag
`-`	:	`REAL`	→	`REAL`	Vorzeichenumkehr

Multiplikation: (Exponenten addieren, Mantissen multiplizieren)

$$
\begin{array}{lclclclcl}
2 & * & 30 & = & & & & & \\
0.2\cdot 10^1 & * & 0.3\cdot 10^2 & = & 0.2*0.3 & * & 10^1 * 10^2 & & \\
 & & & = & 0.06 & * & 10^3 & & \\
 & & & = & 0.6 & * & 10^2 & = & 60
\end{array}
$$

Addition: (Exponenten angleichen, Mantissen addieren)

$$
\begin{array}{rlcl}
2 & 0.2 \cdot 10^1 & = & 0.02 \cdot 10^2 \\
+\ 30 & 0.3 \cdot 10^2 & = & 0.30 \cdot 10^2 \\
\hline
32 & & & 0.32 \cdot 10^2
\end{array}
$$

Problem:

$$
\begin{array}{rlcl}
1000 & 0.1 \cdot 10^4 & = & 0.1000000 \cdot 10^4 \\
+ \quad 0.001 & 0.1 \cdot 10^{-2} & = & 0.0000001 \cdot 10^4 \\
\hline
= 1000.001 & & & 0.1000001 \cdot 10^4 \\
 & \text{bei 6 signifikanten Dezimalstellen} & = & 0.1 \cdot 10^4 = 1000
\end{array}
$$

Rundungsfehler

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$$

$$0.333 + 0.333 + 0.333 = 0.999$$

Konstantenbezeichner

$2.$
2.0
-2.538
$2.5E2 = 2.5 * 10^2 = 250$
$2.5E{-2} = 2.5 * 10^{-2} = 0.025$

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1$$

```
summe := 0.0; summand := 1.0;
WHILE summand > 0.0000001 DO
   summand := summand / 2.0;
   summe := summe + summand;
END;
WriteFloat (summe, 3, 15);
```

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots$$

```
pi = 4.0;
nenner := 1.0;
FOR i := 1 TO 1000 DO
   nenner := nenner + 2.0;
   summand := 4.0 / nenner;

   IF ODD(i) THEN summand := -summand; END;
   pi := pi + summand
END;
```

Typumwandlung

```
FLOAT : INTEGER → REAL
TRUNC : REAL    → INTEGER

   VAR durchschnitt, gesamt : REAL;
       anzahl               : INTEGER;
   BEGIN
      durchschnitt := gesamt / FLOAT(anzahl);
      WriteString ("Mindestens eine Person wiegt mehr als ");
      WriteInt (TRUNC(durchschnitt), 3);
      WriteString (" Kilo")
   END;
```

BOOLEAN

Wertebereich: `TRUE` = wahr
`FALSE` = falsch
Konstantenbezeichner ↑

Mögliche Codierung in einem Byte:

```
FALSE  =  0
TRUE   =  1
```

Operationen

AND	:	BOOLEAN	×	BOOLEAN	→	BOOLEAN	logisches Und
OR	:	BOOLEAN	×	BOOLEAN	→	BOOLEAN	logisches Oder
NOT	:	BOOLEAN			→	BOOLEAN	Negation

P	Q	P AND Q	P OR Q	NOT Q
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	
TRUE	TRUE	TRUE	TRUE	

Ausdrücke werden von links nach rechts ausgewertet:

```
WHILE (t > 0) AND (n DIV t <> b) DO
   t := t - 1
END;
IF (3 <= x ) AND ( x <= 10) ...
```

De Morgan'sche Regeln:

```
(NOT p) AND (NOT q) = NOT (p OR q)
(NOT p) OR  (NOT q) = NOT (p AND q)
```

Typischer Einsatz

```
VAR trace : BOOLEAN;
   :
IF ... THEN trace := TRUE
       ELSE trace := FALSE
END;
   :
IF trace THEN Write ...          END;


VAR fertig : BOOLEAN;
   :
fertig := FALSE;
REPEAT
   :
   IF ... THEN fertig := TRUE END;
   :
UNTIL fertig;
```

CHAR

Wertebereich: alle Zeichen eines Zeichensatzes,
z.B. ASCII-Zeichensatz (American Standard Code for Information Interchange)

```
VAR c : CHAR;
   :
c := "A";
c := "a"; c := 'a';
c := "7";
c := "%"
```

`VAR c : CHAR;` Typischerweise codiert als Zahl zwischen 0 und 255

`c := "A";` ‘‘A’’ codiert als $0100001_2 = 65_{10}$

Die sogenannten Kontrollzeichen bewirken gewisse Aktionen bei Ausgabegeräten, z.B. Piep, Wagenrücklauf, ...

```
REPEAT
   WriteString ("Fertig?");
   Read(c);
UNTIL (c = 'J') OR (c = 'j') OR (c = 'N') OR (c = 'n');
```

Vergleichsoperationen

```
<, <=, =, >=, >, <>, # : CHAR × CHAR → BOOLEAN
```

verlangen Ordnung auf dem Zeichensatz!

Es gilt `0 < 1 <...< 9 <...< A < B <...< Z <...< a < b <...< z < ...`

```
Read(c);
IF((("A" <= c) AND (c <= "Z")) OR
   (("a" <= c) AND (c <= "z")))
   THEN WriteString ("Das Zeichen ist ein Buchstabe")
END;
```

```
(******************************************************************************)
(*      Datum:  01.11.93  14:15 Uhr                                           *)
(*      Autor:  Oliver Vornberger                                             *)
(*      Modul:  char.m2                                                       *)
(*              demonstriert Typumwandlung bei Character-Variablen            *)
(*                                                                            *)
(*              ORD     : CHAR    -->   CARDINAL                              *)
(*              ORD(c)  = Nummer von Zeichen c                                *)
(*                                                                            *)
(*              CHR     : INTEGER / CARDINAL    -->     CHAR                  *)
(*              CHR(i)  = Zeichen mit Nummer i                                *)
(*                                                                            *)
(******************************************************************************)

MODULE char;
FROM InOut IMPORT Read,Write,WriteInt,WriteString,WriteLn; (* importiere I/O-Routinen   *)
VAR i, wert, ziffernwert : INTEGER;                      (* deklariere 3 Integer-Zahlen  *)
VAR c : CHAR;                                            (* deklariere Character-Variable*)
BEGIN

        WriteString(" ASCII-Tabelle: ");
        WriteLn; WriteString("         ");

        FOR i:=32 TO 127 DO                              (* erstelle eine ASCII-Tabelle  *)
                WriteInt(i,7);
                Write(' ');
                Write(CHR(i));
                IF (i MOD 10)=0 THEN WriteLn END;
        END;
        WriteLn;

        WriteString("Bitte eine Ziffernfolge eingeben: ");
        wert := 0;
        Read(c);
        WHILE ('0' <= c) AND (c <= '9') DO               (* lies eine Zeichenfolge ein   *)
                ziffernwert := ORD(c) - ORD('0');        (* und interpretiere sie als    *)
                wert := 10 * wert + ziffernwert;         (* Integer-Zahl                 *)
                Read(c)
        END;

        WriteString("Der Wert der eingelesenen Zahl lautet ");
        WriteInt(wert,6);
        WriteLn

END char.
```

Ausgabe des Programms char.m2

```
ASCII-Tabelle:
              32       33 !     34 "     35 #     36 $     37 %     38 &     39 '     40 (
     41 )     42 *     43 +     44 ,     45 -     46 .     47 /     48 0     49 1     50 2
     51 3     52 4     53 5     54 6     55 7     56 8     57 9     58 :     59 ;     60 <
     61 =     62 >     63 ?     64 @     65 A     66 B     67 C     68 D     69 E     70 F
     71 G     72 H     73 I     74 J     75 K     76 L     77 M     78 N     79 O     80 P
     81 Q     82 R     83 S     84 T     85 U     86 V     87 W     88 X     89 Y     90 Z
     91 [     92 \     93 ]     94 ^     95 _     96 `     97 a     98 b     99 c    100 d
    101 e    102 f    103 g    104 h    105 i    106 j    107 k    108 l    109 m    110 n
    111 o    112 p    113 q    114 r    115 s    116 t    117 u    118 v    119 w    120 x
    121 y    122 z    123 {    124 |    125 }    126 ~    127
Bitte eine Ziffernfolge eingeben: Der Wert der eingelesenen Zahl lautet  12345
```

2.4. Konstanten

```
(****************************************************************************************)
(*      Datum:  01.11.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  konstante.m2                                                            *)
(*              demonstriert Benutzung von Konstanten                                   *)
(*                                                                                      *)
(****************************************************************************************)

MODULE konstante;

FROM MathLib IMPORT sin;                                    (* importiere Sinusfunktion     *)
FROM InOut IMPORT Read, Write,WriteInt,WriteString,WriteLn; (* importiere I/O-Routinen  *)
FROM RealInOut IMPORT WriteFloat;                           (* Ausgabe fuer Gleitkommazahl  *)

CONST MAXGRAD   = 360;                                      (* Integer-Konstante            *)
CONST PI        = 3.141592654;                              (* Real-Konstante               *)
CONST BITTE     = "Bitte eine Zeichenfolge: ";              (* String-Konstante             *)
CONST BLANK     = ' ';                                      (* Character-Konstante          *)
CONST ENDOFLINE = 12C;                                      (* Character-Konstante, oktal   *)
                                                            (* entspricht CHR(10)           *)

VAR grad        : INTEGER;                                  (* deklariere Integer-Variable  *)
VAR r           : REAL;                                     (* deklariere Real-Variable     *)
VAR c           : CHAR;                                     (* deklariere Charakter-Variable*)

BEGIN

        FOR grad:=0 TO MAXGRAD DO
                r := sin(FLOAT(grad) * PI/180.0);           (* bestimme Sinus von Winkel    *)
                WriteInt(grad,6);
                WriteFloat(r,6,9);
                WriteLn;
        END;

        WriteString(BITTE);
        REPEAT                                              (* kopiere Eingabe ohne Blanks  *)
                Read(c);
                IF c # BLANK THEN Write(c) END;
        UNTIL c = ENDOFLINE;
        WriteLn

END konstante.
```

2.5. Typen

```
(****************************************************************************************)
(*      Datum:  02.11.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  typen.m2                                                                *)
(*              Umgang mit selbstdefinierten Typen                                      *)
(*                                                                                      *)
(*              Der neue Typ uebernimmt Wertebereich, Konstantenbezeichner und          *)
(*              Operationen vom alten Typ und ist mit ihm kompatibel.                   *)
(*              Die nachfolgend definierten Typen sind nur dann kompatibel, wenn sie    *)
(*              durch Gleichsetzung eingefuehrt wurden.                                 *)
(*                                                                                      *)
(*              TYPE T1 = ...                                                           *)
(*              TYPE T2 = ...   T2 ist nicht kompatibel zu T1                           *)
(*              TYPE T3 = T1;   T3 ist kompatibel zu T1                                 *)
(****************************************************************************************)

MODULE typen;

FROM    InOut   IMPORT ReadString,WriteString,WriteLn;  (* Importiere I/O-Routinen      *)

TYPE    Nummer  = INTEGER;                              (* Typ durch Umbenennung        *)
                                                        (* Unterbereichstypen:          *)
        Monat   = [1..12];                              (* Basistyp CARDINAL            *)
        Ziffer  = ['0'..'9'];                           (* Basistyp CHAR                *)
        Index   = INTEGER[0..99];                       (* Basistyp INTEGER             *)
        Apostel = [1..12];                              (* Basistyp CARDINAL            *)

VAR     x       : INTEGER;
        k       : CARDINAL;
        n       : Nummer;                               (* x+n erlaubt                  *)
        m       : Monat;                                (* k+m erlaubt                  *)
        z       : Ziffer;                               (* ORD(z) erlaubt               *)
        i       : Index;                                (* x+i erlaubt                  *)
        a       : Apostel;                              (* nicht erlaubt ist a+m        *)
                                                        (* nur bei TYPE Apostel = Monat *)
TYPE    Farbe   = (rot, gelb, gruen);                   (* Aufzaehlungstyp              *)
        Wochentag = (montag, dienstag, mittwoch, donnerstag, freitag, samstag, sonntag);
        Werktag = [montag..freitag];                    (* Unterbereich von Wochentag   *)

VAR     f,g     : Farbe;                                (* Variable vom Typ Farbe       *)
        tag     : Wochentag;                            (* Variable vom Typ Wochentag   *)
        zustand : (ja, nein, weissnicht);               (* Anonyme Typdefinition        *)
BEGIN
        g := rot;
        f := g;
        IF f < g THEN (* ... *) END;

        CASE f OF                                       (* es gilt:                     *)
        rot:    WriteString("Rot")  |                   (* ORD(rot)   = 0               *)
        gelb:   WriteString("Gelb") |                   (* ORD(gelb)  = 1               *)
        gruen:  WriteString("Gruen")                    (* ORD(gruen) = 2               *)
        END;

        tag := montag;
        WHILE tag < sonntag DO INC(tag) END;            (* setze tag auf Nachfolger     *)
        FOR tag := montag TO freitag DO (* ... *) END;  (* durchlaufe alle Tage         *)
END typen.
```

2.6. Felder

```
(********************************************************************************)
(*      Datum:  02.11.93  14:15 Uhr                                             *)
(*      Autor:  Oliver Vornberger                                               *)
(*      Modul:  array.m2                                                        *)
(*              Umgang mit arrays                                               *)
(*                                                                              *)
(********************************************************************************)

MODULE array;

CONST   UNTEN   = 0;
        OBEN    = 99;

TYPE    Vektor  = ARRAY [UNTEN..OBEN] OF REAL;          (* eindimensionaler Vektor      *)
        Matrix  = ARRAY [1..8],[1..8] OF INTEGER;       (* zwei-dimensionale Matrix     *)

        Figur   = (Weiss, Leer, Schwarz);
        Brett   = ARRAY ['a'..'h'],[1..8] OF Figur;

VAR     v,w     : Vektor;                               (* zwei Vektoren                *)
        m       : Matrix;                               (* eine Matrix                  *)
        b       : Brett;                                (* ein Brett                    *)

        i,j     : INTEGER;
        sum     : REAL;
        c       : CHAR;

BEGIN

        sum := 0.0;
        FOR i := UNTEN TO OBEN DO                       (* Summiere ueber alle Elemente *)
                sum := sum + v[i]
        END;

        w :=v;                                          (* komponentenweises Kopieren   *)

        FOR i := 1 TO 8 DO
                FOR j := 1 TO 8 DO                      (* setze alle Matrixelemente    *)
                        m[i,j] := 0;                    (* auf 0                        *)
                END
        END;

        FOR c := 'a' TO 'h' DO
                FOR j := 1 TO 8 DO                      (* besetze alle Brettfelder     *)
                        b[c,j] := Leer                  (* mit der leeren Figur         *)
                END
        END;


END array.
```

3. Einsatz von Feldern

3.1. Feld von Daten

```
(*****************************************************************************************)
(*      Datum:  02.11.93  14:15 Uhr                                                      *)
(*      Autor:  Oliver Vornberger                                                        *)
(*      Modul:  matrix.m2                                                                *)
(*              Multiplikation zweier NxN Matrizen                                       *)
(*              (ohne Ein- und Ausgabe)                                                  *)
(*                                                                                       *)
(*              c       :=      Summe          a    *    b                               *)
(*               ij          k=0 bis N-1        ik        kj                             *)
(*                                                                                       *)
(*****************************************************************************************)

MODULE matrix;

CONST N         = 10;                                 (* Elemente pro Dimension      *)

TYPE Matrix     = ARRAY [0..N-1], [0..N-1] OF REAL;   (* Typ fuer Matrix             *)

VAR a,b,c       : Matrix;                             (* 3 Matrizen                  *)
VAR i,j,k       : INTEGER;                            (* Laufindizes                 *)
VAR summe       : REAL;                               (* Zwischensumme               *)

BEGIN

        FOR i := 0 TO N-1 DO
                FOR j := 0 TO N-1 DO

                        summe := 0.0;
                        FOR k := 0 TO N-1 DO
                                summe := summe + a[i,k] * b[k,j]
                        END;
                        c[i,j] := summe

                END
        END

END matrix.

(* Obacht:   TYPE Feld = ARRAY[0..N-1],[0..N-1] OF REAL ist nicht kompatibel mit Matrix *)
(* wohl aber TYPE Feld = Matrix                                                         *)
```

3.2. Feld von Zeichen

```
(****************************************************************************************)
(*      Datum:  02.11.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  string.m2                                                               *)
(*              Vergleich von Zeichenketten                                             *)
(*                                                                                      *)
(****************************************************************************************)

MODULE string;

FROM    InOut   IMPORT ReadString,WriteString,WriteLn;  (* Importiere I/O-Routinen      *)

CONST   N = 50;                                         (* Groesse der Strings          *)
        EOS = CHR(0);                                   (* End of String                *)

TYPE    String  = ARRAY [0..N-1] OF CHAR;               (* Typ fuer Zeichenkette        *)

VAR     s,t     : String;                               (* 2 Strings                    *)
        i       : INTEGER;                              (* Laufindex                    *)

BEGIN

        WriteString("Bitte eine Zeichenkette: ");       (* Es werden 2 Strings          *)
        ReadString(s);                                  (* eingelesen                   *)

        WriteString("Bitte eine Zeichenkette: ");
        ReadString(t);

        i := 0;                                         (* bestimme die Position der    *)
        WHILE (i < N) AND (s[i] = t[i])                 (* ersten Ungleichheit          *)
                AND (s[i] <> EOS) AND (t[i] <> EOS) DO  (* oder eines Stringendes       *)
                i := i + 1
        END;

        WriteString(s);

        IF (i = N) OR (s[i] = t[i])     THEN WriteString(" ist gleich ")
        ELSIF s[i] < t[i]               THEN WriteString(" ist kleiner als ")
                                        ELSE WriteString(" ist groesser als ")
        END;

        WriteString(t);
        WriteLn;

END string.
```

3.3. Feld von Wahrheitswerten

```
(*****************************************************************************************)
(*      Datum:  08.11.93  14:15 Uhr                                                      *)
(*      Autor:  Oliver Vornberger                                                        *)
(*      Modul:  sieb.m2                                                                  *)
(*              Sieb des Eratosthenes zur Ermittlung von Primzahlen                      *)
(*              Idee: Streiche jeweils alle Vielfachen von Primzahlen                    *)
(*                                                                                       *)
(*****************************************************************************************)

MODULE sieb;

FROM    InOut   IMPORT WriteInt, WriteString, WriteLn;  (* I/O—Routinen               *)

CONST   N       = 1000;                                 (* Intervallobergrenze        *)

VAR     sieb    : ARRAY [2..N] OF BOOLEAN;              (* sieb[i]=TRUE,falls i Primzahl*)
        i, j    : INTEGER;

BEGIN


        WriteString("Alle Primzahlen von 2 bis ");
        WriteInt(N,6);
        WriteLn;

        FOR i := 2 TO N DO sieb[i] := TRUE END;         (* alle sind zunaechst Primzahl *)
        FOR i := 2 TO N DO

                IF sieb[i] THEN                         (* falls i als Primzahl erkannt *)

                        WriteInt(i,6);                  (* gib sie aus und            *)
                        j := i + i;
                        WHILE j <= N DO
                                sieb[j] := FALSE;       (* streiche Vielfaches von i    *)
                                j := j + i
                        END

                END
        END;
        WriteLn;

END sieb.
```

3.4. Feld von Indizes

```
(*********************************************************************************)
(*      Datum:  08.11.93  14:15 Uhr                                             *)
(*      Autor:  Oliver Vornberger                                               *)
(*      Modul:  arrayabzaehlreim.m2                                             *)
(*              N kinder stehen im Kreis                                        *)
(*              jedes K-te wird abgeschlagen                                    *)
(*              (Realisierung durch array mit Nachfolger)                       *)
(*                                                                              *)
(*********************************************************************************)

MODULE  arrayabzaehlreim;

FROM    InOut   IMPORT WriteCard, WriteString, WriteLn;

CONST   N = 20;                                      (* Anzahl der Kinder           *)
        K = 4;                                       (* jedes k-te abschlagen       *)

VAR     i,index : CARDINAL;
        next    : ARRAY[0..N-1] OF CARDINAL;         (* next[i] = Nachfolger von i   *)

BEGIN
        FOR i:=0 TO N-1 DO                           (* initiale Aufstellung        *)
                next[i] := (i+1) MOD N;
        END;
                                                     (* index zeigt auf das Kind vor *)
        index := N-1;                                (* dem ausgezeichneten Kind     *)
        WHILE next[index] <> index DO                (* solange abschlagen, bis je-  *)
                FOR i:=1 TO K-1 DO                   (* mand sein eigener Nachfolger *)
                        index := next[index];
                END;
                WriteString("Ausgeschieden ist ");   (* Kind hint. index scheidet aus*)
                WriteCard(next[index],6);
                WriteLn;
                next[index] := next[next[index]];    (* index erh. neuen Nachfolger  *)
        END;

        WriteString("Uebrig geblieben ist ");
        WriteCard(index,3);
        WriteLn;

END arrayabzaehlreim.
```

3.5. Feld von Zuständen

Ein *endlicher Automat* A ist ein 5-Tupel $A = (S, \Sigma, \delta, s_0, F)$ mit

S	=	endliche Zustandsmenge
Σ	=	endliches Eingabealphabet
$\delta: S \times \Sigma \rightarrow S$	=	Überführungsfunktion
$s_0 \in S$	=	Anfangs- oder Startzustand
$F \subseteq S$	=	Endzustände

Ein Wort $w \in \Sigma^*$ wird akzeptiert, falls

$\delta^* (s_0, w) \in F$

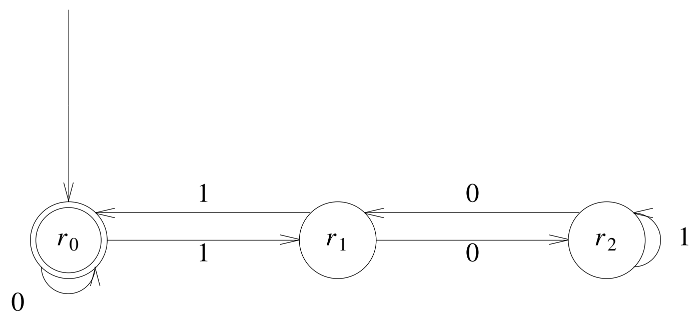
$\delta^* (s_0, x_0 \, x_1 \, x_2 \, \cdots \, x_k) = \delta(\, \cdots \, \delta(\delta(\delta(s_0, x_0), x_1), x_2) \, \cdots \, x_k)$

Beispiel:

A soll durch 3 teilbare Dualzahlen erkennen.

$S = \{r_0, r_1, r_2\}$
$\Sigma = \{\texttt{"0"}, \texttt{"1"}\}$
Startzustand ist r_0
$F = \{r_0\}$

Die Wirkungsweise der Überführungsfunktion δ kann durch den folgenden Zustandsüberführungsgraphen beschrieben werden. In diesem Graphen führt eine mit x beschriftete Kante von Knoten a zu Knoten b, falls gilt: $\delta(a, x) = b$.



```
(******************************************************************************)
(*      Datum:  08.11.93  14:15 Uhr                                           *)
(*      Autor:  Oliver Vornberger                                             *)
(*      Modul:  automat.m2                                                    *)
(*              Endlicher Automat mit  delta : Zustand x Eingabe --> Zustand  *)
(*              liest Dualzahl ein und merkt sich den Rest bei Teilung durch 3 *)
(*                                 1                        0                 *)
(*                               <----                    <----               *)
(*              RestNull                      RestEins               RestZwei *)
(*              ---0--->         ---->                    ---->   ---1---->   *)
(*                                 1                        0                 *)
(******************************************************************************)

MODULE automat;

FROM    InOut   IMPORT Read, WriteString, WriteLn;      (* I/O-Routinen                *)

CONST   EOL     = 12C;                                  (* End-of-Line-Eingabe         *)

TYPE    Eingabe = ['0'..'1'];

        Zustand = (RestNull, RestEins, RestZwei);       (* der bisher gelesene String  *)
                                                        (* liefert bei Division durch 3 *)
                                                        (* den Rest 0, Rest 1, Rest 2  *)

        Tabelle = ARRAY Zustand, Eingabe OF Zustand;    (* zweidimensionale Tabelle    *)
                                                        (* Zustand X Eingabe           *)
VAR     z       : Zustand;
        delta   : Tabelle;
        c       : CHAR;

BEGIN

        delta[RestNull,'0'] := RestNull;                (* Initialisierung der         *)
        delta[RestNull,'1'] := RestEins;                (* Zustandsueberfuehrungs-     *)
        delta[RestEins,'0'] := RestZwei;                (* funktion delta des Automaten *)
        delta[RestEins,'1'] := RestNull;
        delta[RestZwei,'0'] := RestEins;
        delta[RestZwei,'1'] := RestZwei;

        WriteString("Bitte eine Dualzahl eingeben: ");

        z := RestNull;                                  (* Anfangszustand              *)
        Read(c);                                        (* c ist das erste Zeichen     *)
        WHILE c # EOL DO                                (* solange kein Zeilenende     *)
                z := delta[z,c];                        (* bestimme Folgezustand       *)
                Read(c);                                (* c ist das naechste Zeichen  *)
        END;

        IF z=RestNull   THEN WriteString("Die Zahl ist durch 3 teilbar")
                        ELSE WriteString("Die Zahl ist nicht durch 3 teilbar")
        END;
        WriteLn;

END automat.
```

3.6. Lineare und binäre Suche

```
(****************************************************************************************)
(*      Datum:  9.11.93  14:15 Uhr                                                       *)
(*      Autor:  Oliver Vornberger                                                        *)
(*      Modul:  suche.m2                                                                 *)
(*              lineare Suche im array                                                   *)
(*              binaere Suche im geordneten array                                        *)
(*                                                                                       *)
(****************************************************************************************)

MODULE suche;

FROM    InOut   IMPORT WriteInt, WriteString, WriteLn;  (* I/O-Routinen                  *)

CONST   N       = 100;

TYPE    Vektor  = ARRAY [0..N-1] OF INTEGER;

VAR     folge                   : Vektor;
        i, x                    : INTEGER;
        min, index              : INTEGER;
        links, rechts, mitte    : INTEGER;
        gefunden                : BOOLEAN;

BEGIN
        i := 0;                                         (* lineare Suche                 *)
        WHILE (i<N) AND (folge[i] # x) DO INC(i) END;   (* laufe bis zum Ende oder       *)
                                                        (* bis Element gefunden          *)
        IF i=N  THEN WriteString("Nicht gefunden !")
                ELSE WriteString("Gefunden an Position "); WriteInt(i,6)
        END;



        min := folge[0];                                (* finde das Minimum             *)
        FOR i := 1 TO N-1 DO
                IF folge[i] < min THEN index := i; min := folge[i] END
        END;


                                                        (* binaere Suche                 *)
        links    := 0;                                  (* Startwert fuer linken Rand    *)
        rechts   := N-1;                                (* Startwert fuer rechten Rand   *)
        gefunden := FALSE;                              (* zunaechst nichts gefunden     *)
        WHILE (links <= rechts) AND NOT gefunden DO
                mitte := (links+rechts) DIV 2;                    (* Suchadresse         *)
                IF      folge[mitte] = x THEN gefunden := TRUE    (* richtig gesprungen *)
                ELSIF   folge[mitte] < x THEN links    := mitte+1 (* zu kurz gesprungen *)
                ELSE (* folge[mitte] > x *)   rechts   := mitte-1 (* zu weit gesprungen *)
                END
        END;
        IF gefunden THEN WriteString("Gefunden an Position "); WriteInt(mitte,6)
                    ELSE WriteString("Nicht gefunden !")
        END;

END suche.
```

Analyse der Laufzeit der linearen Suche

Beh.: Die Anzahl der Vergleiche beträgt im ungünstigsten Fall n und im Durchschnitt $\frac{n}{2}$.

Analyse der Laufzeit der binären Suche

Beh.: In einem Schleifendurchlauf schrumpft ein Intervall der Länge $2^k - 1$ auf ein Intervall der Länge $2^{k-1} - 1$.

Beweis:

Sei Länge vom alten Intervall =

$$
\begin{array}{llcl}
 & 2^k - 1 & = & \text{rechts} - \text{links} + 1 \\
\Rightarrow & 2^k & = & \text{rechts} - \text{links} + 2 \\
\Rightarrow & 2^{k-1} & = & \frac{\text{rechts} - \text{links}}{2} + 1 \\
\Rightarrow & 2^{k-1} - 1 & = & \frac{\text{rechts} - \text{links}}{2} = \frac{\text{rechts} + \text{rechts} - \text{links} - \text{rechts}}{2} \\
 & & = & \text{rechts} - (\frac{\text{links} + \text{rechts}}{2} + 1) + 1 \\
 & & = & \text{neue Intervallänge im THEN-Zweig}
\end{array}
$$

Korollar: $2^k - 1$ Zahlen verursachen höchstens k Schleifendurchläufe, da nach k Halbierungen die Intervall-Länge 1 beträgt.

Beispiel: $2^5 - 1$, $2^4 - 1$, $2^3 - 1$, $2^2 - 1$, $2^1 - 1$
31 15 7 3 1

Anders formuliert:
n Zahlen verursachen höchstens $\log_2 n$ Schleifendurchläufe.

4. Prozeduren

4.1. Sichtbarkeit, Lebensdauer, Parameter

Sichtbarkeit statisch, compilergeregelt

Die formalen Parameter und gegebenenfalls weitere lokale Variablen sind nur im Rumpf der Prozedur sichtbar (und in darin deklarierten Prozeduren).

Lebensdauer dynamisch, vom Laufzeitsystem geregelt

Sie ‘‘leben’’ nur während der Abarbeitung der Prozedur.

```
MODULE haupt;
VAR i, j : INTEGER;

PROCEDURE P;
VAR i, k : CHAR;
BEGIN
   (* bekannt ist j als INTEGER und i, k als CHAR *)
END P;

BEGIN
   (* bekannt ist i, j als INTEGER *)
   i := 5; j := 7;
   P;
END haupt.
```

Es gibt zwei Arten von formalen Parametern:
Wert-Parameter (*call by value*) und Variablen-Parameter (*call by reference*).

Wert-Parameter:

Bei Aufruf der Prozedur werden die formalen Parameter als lokale Variablen betrachtet und mit dem Wert der aktuellen Parameter initialisiert.
Nebeneffekt: Die aktuellen Parameter sind nach Ablauf der Prozedur unverändert.

Variablen-Parameter:

Bei Aufruf der Prozedur werden die aktuellen Parameter (es müssen Variablen sein) mit den Namen der formalen Parameter angesprochen.
Nebeneffekt: Die Werte der aktuellen Parameter können sich verändern! Beabsichtigt!

```
PROCEDURE halbiere (VAR  x : INTEGER);
BEGIN                    ↑ Variablen-Parameter
 x := x DIV 2;
END halbiere;
```

Aufruf: `halbiere(a);`

```
(*************************************************************************************)
(*      Datum:  09.11.93  14:15 Uhr                                                 *)
(*      Autor:  Oliver Vornberger                                                   *)
(*      Modul:  prozedur.m2                                                         *)
(*              Typen der aktuellen Parameter muessen mit den Typen der             *)
(*              formalen Parameter uebereinstimmen                                  *)
(*************************************************************************************)

MODULE prozedur;

FROM InOut IMPORT ReadInt,WriteInt,WriteString,WriteLn; (* I/O-Routinen              *)

PROCEDURE bitte;                                        (* Prozedur ohne Parameter    *)
BEGIN
        WriteString("Bitte eine Zahl: ")
END bitte;

PROCEDURE vorschub(n : INTEGER);                        (* n ist Value-Parameter      *)
VAR i : INTEGER;                                        (* lokale Variable            *)
BEGIN
        FOR i:=1 TO n DO WriteLn END
END vorschub;

PROCEDURE hoch(a, b : INTEGER) : INTEGER;               (* Fun.prozedur,liefert INTEGER *)
VAR i, x : INTEGER;
BEGIN
        x := 1;
        FOR i:=1 TO b DO x := x*a END;
        RETURN x                                        (* Rueckgabe des Funktionswerts *)
END hoch;

PROCEDURE halbiere(VAR n : INTEGER);                    (* n ist Variablen-Parameter  *)
BEGIN
        n := n DIV 2
END halbiere;

PROCEDURE bittezahl (VAR x : INTEGER);                  (* Rueckgabeparameter muessen *)
BEGIN                                                   (* Variablen-Parameter sein   *)
        WriteString("Bitte Zahl: "); ReadInt(x)
END bittezahl;

PROCEDURE tausche (VAR x,y : INTEGER);                  (* vertausche den Inhalt von  *)
VAR hilf : INTEGER;                                     (* zwei Speicherplaetzen      *)
BEGIN
        hilf := x;
        x := y;
        y := hilf
END tausche;

VAR     i, j : INTEGER;                         (* lokale Variable des Hauptprogramms  *)
        a    : ARRAY[0..99] OF INTEGER;
BEGIN
        bitte;                                  (* Aufruf der Prozedur bitte           *)
        vorschub(3*i+9);                        (* aktueller Parameter ist 3i+9        *)
        i := hoch(3,4);                         (* i erhaelt Ergebnis von "hoch"       *)
        WriteInt(hoch(3,4),5);                  (* Schreibe 3 hoch 4 auf 5 Stellen     *)
        halbiere(i);                            (* i wird halbiert                     *)
        bittezahl(i);                           (* i wird eingelesen                   *)
        tausche(a[i],a[i+1]);                   (* tausche i-ten und i+1-ten Eintrag   *)
END prozedur.
```

```
(****************************************************************************************)
(*      Datum:  15.11.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  arrayparameter.m2                                                       *)
(*              Uebergabe von arrays                                                    *)
(****************************************************************************************)

MODULE arrayparameter;

FROM InOut IMPORT ReadInt,WriteInt,WriteString,WriteLn;

CONST   UNTEN   = 20;
        OBEN    = 28;

TYPE    Vektor  = ARRAY [UNTEN .. OBEN] OF REAL;        (* Real-Vektor                  *)


PROCEDURE summe (VAR a : Vektor; VAR ergebnis : REAL);  (* summiere die Feldelemente    *)
VAR     i : INTEGER;
BEGIN
        ergebnis := 0.0;
        FOR i := UNTEN TO OBEN DO
                ergebnis := ergebnis + a[i];
        END;
END summe;


PROCEDURE summiere (a : ARRAY OF REAL; VAR ergebnis : REAL);    (* summiere             *)
VAR i : CARDINAL;                                               (* die Feldelemente     *)
BEGIN
        ergebnis := 0.0;
        FOR i := 0 TO HIGH(a) DO
                ergebnis := ergebnis + a[i];
        END
END summiere;
(* Beispiel fuer die Entsprechung der Array-Indizes:                                    *)
(* formal:       0       1       2       3       4       5       6       7       8      *)
(* wirklich:    20      21      22      23      24      25      26      27      28      *)


VAR     v : Vektor;
        w : ARRAY[20..28] OF REAL;
        r : REAL;
BEGIN
        summe(v,r);                                     (* r = Summe aller v[i]         *)

                                                        (* summe(w,r) nicht moeglich,   *)
                                                        (* da w nicht vom Typ Vektor    *)

        summiere(v,r);                                  (* r = Summe aller v[i]         *)
        summiere(w,r);                                  (* r = summe aller w[i]         *)

END arrayparameter.
```

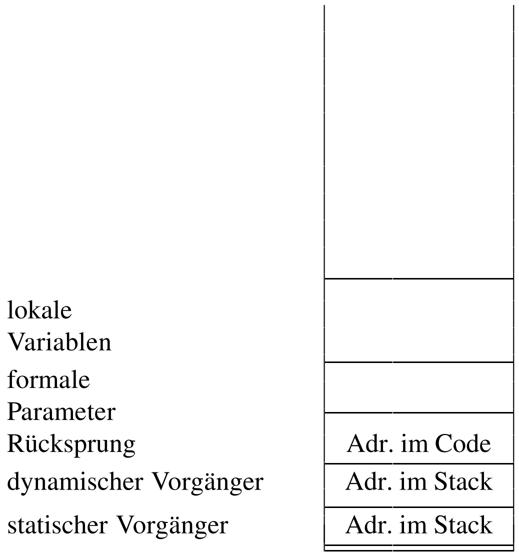
```
(****************************************************************************)
(*      Datum:  9.11.93  14:15 Uhr                                          *)
(*      Autor:  Oliver Vornberger                                           *)
(*      Modul:  variablesarray.m2                                           *)
(*              dynamisches Fuellen eines arrays                            *)
(****************************************************************************)

MODULE variablesarray;

FROM RealInOut IMPORT ReadReal, WriteFloat, Done;          (* Done liefert TRUE, falls  *)
FROM InOut IMPORT ReadInt, WriteInt, WriteString, WriteLn; (* Einlesen erfolgreich war  *)

CONST   max = 100;

TYPE Vector = ARRAY [0..max-1] OF REAL;


PROCEDURE insert(VAR a:Vector; VAR n:CARDINAL; x:REAL): BOOLEAN; (* fuegt x ins array a *)
VAR index : CARDINAL;                                   (* ein und liefert TRUE, falls  *)
BEGIN                                                   (* array voll, sonst FALSE      *)
        index := n;
        WHILE (index>0) AND (a[index-1] > x) DO
                a[index] := a[index-1];
                DEC(index);
        END;
        a[index] :=x;
        INC(n);
        IF n = max THEN RETURN TRUE ELSE RETURN FALSE END;
END insert;


PROCEDURE print (VAR a : ARRAY OF REAL; n : CARDINAL);  (* gibt die ersten n Elemente    *)
VAR i : CARDINAL;                                       (* eines Real-array a aus       *)
BEGIN
        WriteString("Der Inhalt lautet: ");
        WriteLn;
        FOR i:=0 TO n-1 DO WriteFloat(a[i],3,3); END;
        WriteLn;
END print;

VAR     a : Vector;                                     (* REAL - Vektor                *)
        n : CARDINAL;                                   (* momentane Anzahl             *)
        r : REAL;                                       (* eine REAL-Zahl               *)
        arrayvoll : BOOLEAN;                            (* TRUE, falls array voll ist   *)
BEGIN
        n := 0; arrayvoll := FALSE;
        WriteString("Bitte mehrere Real-Zahlen: (Abbruch durch ^D): "); WriteLn;
        ReadReal(r);
        WHILE Done AND NOT arrayvoll DO
                arrayvoll := insert(a,n,r);
                ReadReal(r);
        END;

        print(a,n);

END variablesarray.
```

4.2. Laufzeitkeller

Bei Aufruf einer Prozedur wird eine neue Schachtel auf den **Laufzeitkeller** (*Stack*) gelegt mit Speicherplatz für

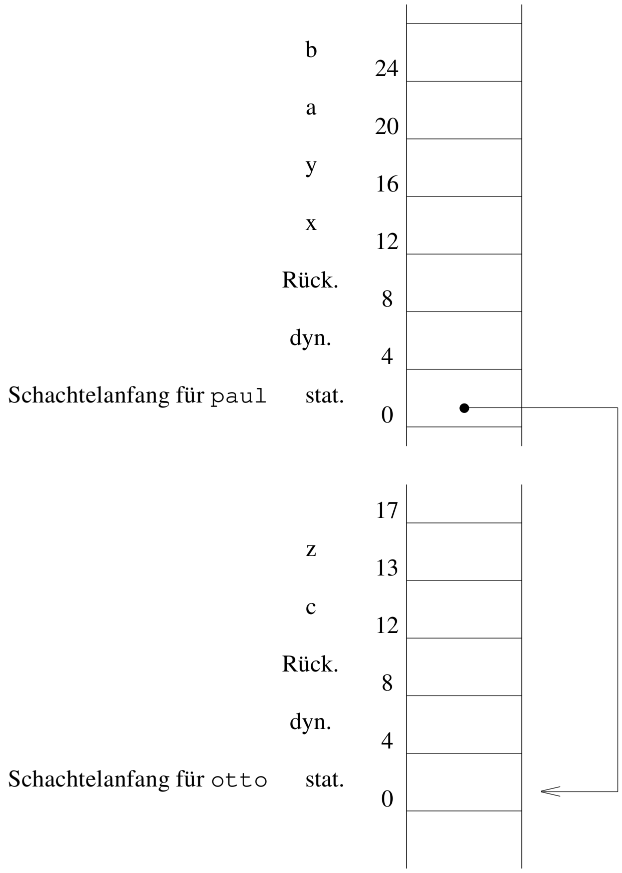
- **statischer Vorgänger**
  (= Schachtelanfang der umschließenden Prozedur)
  zur Referenz von globalen Variablen
- **dynamischer Vorgänger**
  (= Schachtelanfang der aufrufenden Prozedur)
  zum Abräumen der Schachtel
- **Rücksprungadresse**
  (= Adresse im Code-Segment)
- **formale Parameter**, initialisiert
  mit den Werten bzw. Adressen der
  aktuellen Parameter
- **lokale Variablen**

Schachtelanfang →



Beispiel

```
PROCEDURE otto(c : CHAR);
VAR z : INTEGER;
   PROCEDURE paul(x, y : REAL);
   VAR a, b : INTEGER;
   BEGIN
      b := z;
```



Welcher Maschinenbefehl muß für

`b := z;`

erzeugt werden?

Sei `bottom` die Adresse im Stack, wo die aktuelle Schachtel beginnt.

Der Compiler generiert für die Adresse von *b*:

`adresse1 := bottom + 24;`

Der Compiler generiert für die Adresse von *z*:

`adresse2 := stack[bottom] + 13;`

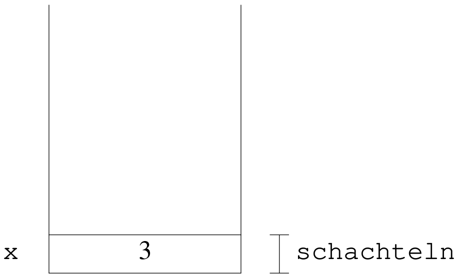
Daraus folgt der Befehl

`stack[adresse1] := stack[adresse2];`

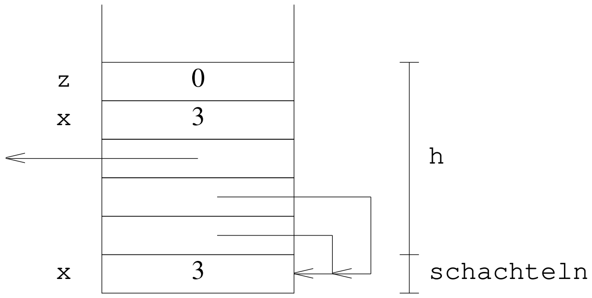
```
(****************************************************************************************)
(*      Datum:  15.11.93  14:15 Uhr                                                    *)
(*      Autor:  Oliver Vornberger                                                      *)
(*      Modul:  schachteln.m2                                                          *)
(*              demonstriert ineinandergeschachtelte Prozeduren                        *)
(*                                                                                     *)
(****************************************************************************************)

MODULE schachteln;

FROM InOut IMPORT ReadInt, WriteInt, WriteString, WriteLn;

PROCEDURE inc(VAR a : INTEGER);                       (* inkrementiert                *)
BEGIN
        a := a + 1;                                   (* Schnappschuss Nr. 4          *)
END inc;

PROCEDURE h(x : INTEGER) : INTEGER;                   (* berechnet Collatz-Funktion   *)
                                                      (* d.h. Anzahl der Transformat. *)

VAR z : INTEGER;                                      (* lokale Variable fuer Proz. h *)

        PROCEDURE f(x : INTEGER) : INTEGER;           (* berechnet eine Transformation*)
        BEGIN   (* von f  *)                          (* Schnappschuss Nr. 3          *)
                inc(z);
                IF ODD(x)
                        THEN RETURN 3*x+1
                        ELSE RETURN x DIV 2
                END;
        END f;


BEGIN (* von h *)
                z := 0;                               (* Schnappschuss Nr. 2          *)
                WHILE x <> 1 DO
                        x := f(x);                    (* transformiere einmal         *)
                END;
                RETURN z;
END h;

VAR x : INTEGER;

BEGIN (* Hauptprogramm *)
        REPEAT
                WriteString("Bitte Zahl (1=Abbruch): ");
                ReadInt(x);                           (* Schnappschuss Nr. 1          *)
                WriteString("Collatz(");
                WriteInt(x,3); WriteString(") = ");
                WriteInt(h(x),6);
                WriteLn;
        UNTIL x=1;

END schachteln.
```

Schnappschüsse zu Modul `schachteln.m2`**:**

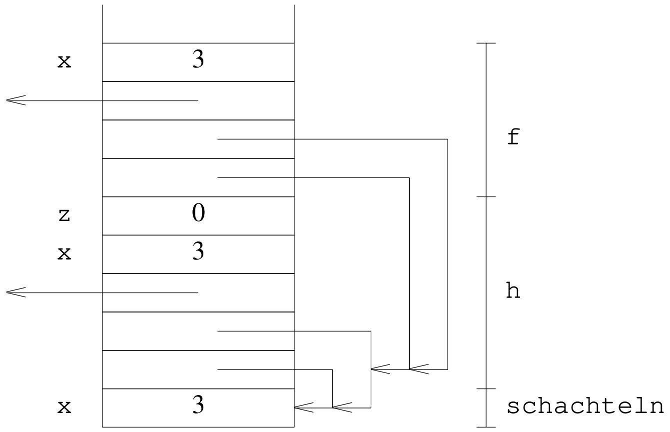
Nr. 1



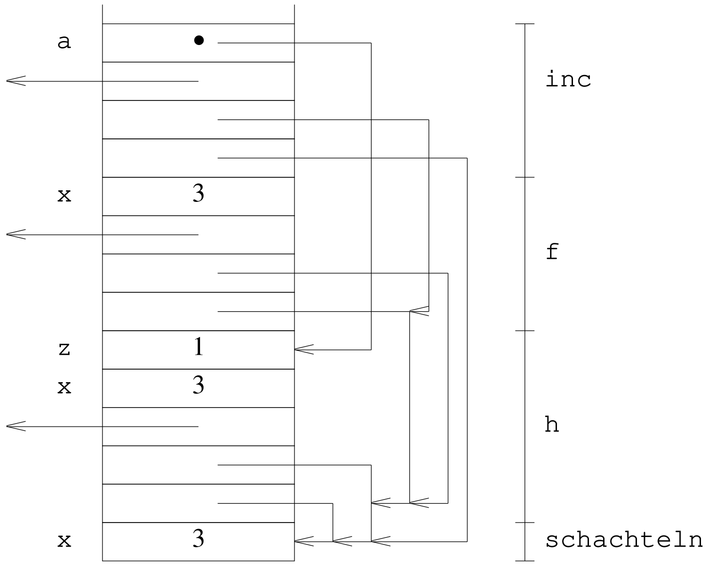
Nr. 2



Nr. 3



Nr. 4



5. Rekursion

5.1. Fakultät, Potenzieren, Fibonacci, GGT

```
(****************************************************************************************)
(*      Datum:  16.11.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  rekursion.m2                                                            *)
(*                                                                                      *)
(****************************************************************************************)

MODULE rekursion;

PROCEDURE fakultaet(n:CARDINAL):CARDINAL;       (*              /                     *)
BEGIN                                           (*              | 1         falls n=0 *)
        IF n=0                                  (*      n! :=  <                      *)
                THEN RETURN 1                   (*              | n*(n-1)!  sonst     *)
                ELSE RETURN n*fakultaet(n-1)    (*              \                     *)
        END
END fakultaet;


PROCEDURE zweihoch(n:CARDINAL):CARDINAL;        (*              /                     *)
BEGIN                                           (*       n      |  1        falls n=0 *)
        IF n=0                                  (*      2  :=  <      n-1             *)
                THEN RETURN 1                   (*              |  2*2      sonst     *)
                ELSE RETURN 2*zweihoch(n-1)     (*              \                     *)
        END
END zweihoch;


PROCEDURE fib(n:CARDINAL):CARDINAL;             (* Jedes Kaninchenpaar bekommt vom    *)
BEGIN                                           (* 2. Jahr an ein Paar als Nachwuchs  *)
        IF n<=1                                 (*                                    *)
                THEN RETURN 1                   (* Jahr         0  1  2  3  4  5  6  7 *)
                ELSE RETURN fib(n-1)+fib(n-2)   (* # Paare      1  1  2  3  5  8 13 21 *)
        END                                     (*                                    *)
END fib;


PROCEDURE ggt(x,y:CARDINAL):CARDINAL;           (*              /                     *)
BEGIN                                           (*              |  x        falls y=0 *)
        IF y=0                                  (* ggt(x,y):=  <                      *)
                THEN RETURN x                   (*              |  ggt(y,x mod y) sonst *)
                ELSE RETURN ggt(y, x MOD y)     (*              \                     *)
        END
END ggt;


BEGIN END rekursion.
```

5.2. Türme von Hanoi

```
(*******************************************************************************)
(*      Datum:  13.11.95  14:15 Uhr                                            *)
(*      Autor:  Oliver Vornberger                                              *)
(*      Modul:  hanoi.m2                                                       *)
(*             x                                                               *)
(*            xxx                                                              *)
(*           xxxxx                                                             *)
(*          xxxxxxx                                                            *)
(*             A                          B                        C           *)
(*                                                                             *)
(*      n Scheiben verschiedener Groesse sollen von Ort A zu Ort C gebracht werden.     *)
(*      Jede Scheibe muss einzeln transportiert werden.                        *)
(*      Es darf nie eine groessere auf einer kleineren zu liegen kommen.       *)
(*      Es darf ein Hilfsort B verwendet werden.                               *)
(*      Idee:   Um Scheiben 1,2,...n von start nach ziel zu bringen:           *)
(*              Bringe Scheiben 1,2,...n-1 von start nach zwischen,            *)
(*              Bringe Scheibe  n von start nach ziel,                         *)
(*              Bringe Scheiben 1,2,...,n-1 von zwischen nach ziel.            *)
(*******************************************************************************)
MODULE hanoi;
FROM InOut IMPORT WriteLn, Write, WriteInt,WriteString;
PROCEDURE verlege (n:INTEGER;start,zwischen,ziel:CHAR); (* druckt die Verlegebefehle,   *)
                                                        (* um n Scheiben von start      *)
                                                        (* unter Verwendung von zwischen*)
BEGIN                                                   (* nach ziel zu legen           *)
        IF n > 1 THEN
                        verlege (n-1, start, ziel, zwischen) ;
                        WriteString('Scheibe '); WriteInt(n,4);
                        WriteString(' von ');    Write(start);
                        WriteString(' nach ');   Write(ziel); WriteLn;
                        verlege (n-1, zwischen, start, ziel)
                 ELSE
                        WriteString('Scheibe    1 von '); Write(start);
                        WriteString(' nach '); Write(ziel); WriteLn
        END;
END verlege;
BEGIN   verlege(10,'A','B','C');                (* Aufruf fuer 10 Scheiben von A nach C *)
END hanoi.
```

Sei $f(n)$ die Anzahl der generierten Verlegebefehle bei Aufruf von
`verlege (n, start, zwischen, ziel).`

Offenbar:
$$f(1) = 1$$
$$f(n) = f(n-1) + 1 + f(n-1)$$

d.h.

n	$f(n)$
1	1
2	3
3	7
4	15

Verdacht: $f(n) = 2^n - 1$

Beweis durch Induktion

$f(1) = 1 = 2^1 - 1$

Sei bis $n-1$ bewiesen

$$f(n) \underset{\substack{\uparrow \\ \text{Rek.-Gl.}}}{=} 2 \cdot f(n-1) + 1 \underset{\substack{\uparrow \\ \text{Ind.-Annahme}}}{=} 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$$

6. Komplexität und Verifikation

nicht: absolute CPU-Zeit angeben
nicht: Anzahl der Maschinenbefehle zählen
sondern: Wachstum der Laufzeit in ''Schritten'' in Abhängigkeit
von der Größe der Eingabe
= Länge ihrer Kodierung
= Anzahl der Daten beschränkter Größe oder Länge der Darstellung einer Zahl

6.1. O-Notation

Seien $f, g : \mathrm{IN} \rightarrow \mathrm{IN}$
f ist höchstens von der Größenordnung g
in Zeichen: $f \in O(g)$
falls $n_0, c \in \mathrm{IN}$ existieren mit

$$\forall\, n \geq n_0 : \quad f(n) \leq c \cdot g(n).$$

Statt $f \in O(g)$ sagt man auch $f = O(g)$
$\Rightarrow$ Wegen des konstanten Faktors c ist die exakte Festlegung eines Schrittes nicht erforderlich.

Beispiel

Sei $f(n) = 31n + 12n^2$.

Dann ist $f = O(n^2)$
Natürlich gilt auch: $f = O(n^7)$

Die wesentlichen Klassen

$\log n$	n	$n \cdot \log n$	n^2	n^3	n^4	...	2^n	3^n	$n!$
↑	↑	↑	↑	↑			↑		↑
binäre Suche	lineare Suche	schlaues Sortieren	dummes Sortieren	Matrix-Multi-plikation			alle Teil-mengen		alle Permu-tationen

Annahme:
1 Schritt dauert 1 Mikrosekunde = 0.000001 sec

	10	20	30	40	50	60
n	0.00001	0.00002	0.00003	0.00004	0.00005	0.00006
n^2	0.0001	0.0004	0.0009	0.0016	0.0025	0.0036
n^3	0.001	0.008	0.027	0.064	0.125	0.216
2^n	0.001	1.0	18 min	13 Tage	36 J	366 Jh
3^n	0.059	58 min	6.5 J	3855 Jh	10^8 Jh	10^{13} Jh
$n!$	3.62	77 Jh	10^{16} Jh	10^{32} Jh	10^{48} Jh	10^{66} Jh

Wächst n um 1, so wächst 2^n um den Faktor 2,
wächst n um 10, so wächst 2^n um den Faktor $2^{10} = 1024$.

D.h.: Ein 1000-mal schnellerer Computer kann eine um 10 Daten größere Eingabe in derselben Zeit rechnen (für ein Problem der Komplexität $O(2^n)$)!

Analoge Aussagen sind möglich bzgl. Speicherplatz
nicht: Anzahl der Bytes
sondern: Wachstum des Platzbedarfs in Speicherplätzen in Abhängigkeit von der Größe der Eingabe.

Aussagen über *Laufzeit* und *Platzbedarf* beziehen sich auf

- *best case* günstigster Fall
- *worst case* ungünstigster Fall
- *average case* im Mittel

Analysen von `FOR`-Schleifen:

Beispiel: Minimumsuche in einem array

```
min := a[0];
FOR i := 1 TO n - 1 DO
   IF a[i] < min THEN min := a[i] END
END;
```

Laufzeit $O(n)$ für best case
worst case
average case.

Beispiel:

```
FOR i := 1 TO k DO
   FOR j := 1 TO k DO
      brett[i, j] := 0
   END
END
```

k^2 Schritte für k^2 Daten
$\Rightarrow O(n)$ – Algorithmus

```
FOR i := 1 TO k DO
   FOR j := 1 TO k DO
      IF a[i] = a[j] ...
   END
END
```

k^2 Schritte für k Daten
$\Rightarrow O(n^2)$ – Algorithmus

Analyse von `WHILE`-Schleifen:

Beispiel: Lineare Suche im array

```
i := 0;
WHILE (i < n) AND (a[i] <> x) DO
   INC (i)
END;
```

Laufzeit:	best case	1 Schritt	$\Rightarrow O(1)$
	worst case	n Schritte	$\Rightarrow O(n)$
	average case	$\frac{n}{2}$ Schritte	$\Rightarrow O(n)$

Annahme für den average case:
Es liegt Permutation der Zahlen von 1 bis n vor.
Dann ist die mittlere Anzahl $= \frac{1}{n} \sum_{i=1}^{n} i \approx \frac{n}{2}$.

Beispiel:

Suche 0 in einem array, bestehend aus Nullen und Einsen.

Laufzeit:	best case	1 Schritt	$\Rightarrow O(1)$
	worst case	n Schritte	$\Rightarrow O(n)$
	average case	$\sum_{i=1}^{n} i \cdot \frac{1}{2^i} \le 2$	$\Rightarrow O(1)$

Obacht:

Alle Laufzeitangaben beziehen sich jeweils auf einen konkreten Algorithmus A (für ein Problem P) = *obere Schranke* für Problem P.

Eine Aussage darüber, wieviel Schritte jeder Algorithmus für ein Problem P mindestens durchlaufen muß, nennt man *untere Schranke* für Problem P.

Beispiel: naives Pattern-Matching

```
VAR s : ARRAY [0..N - 1] OF CHAR;      (* String *)
VAR p : ARRAY [0..M - 1] OF CHAR;      (* Pattern *)
```

Frage: Taucht Pattern p im String s auf?

```
i := -1;            (* Index im String *)
REPEAT
   i := i + 1;      (* naechster Versuch  *)
   j := 0;          (* Index im Pattern *)
   WHILE (j < M) AND s[i + j] = p[j] DO
      j := j + 1;
   END;
UNTIL (j = M) OR (i = N - M);
         ↑              ↑
       Erfolg       Mißerfolg
```

Laufzeit

best case: $O(1)$

worst case: $(N - M + 1) \cdot M$ Vergleiche für $s = AAA...AB$
$p = \quad A...AB$

Sei n die Summe der Buchstaben in Pattern p und String s. Sei $M = x \cdot n$ und $N = (1 - x) \cdot n$ für $0 < x < 1$.

Gesucht wird x, welches $((1 - x) \cdot n - x \cdot n + 1) \cdot x \cdot n = (n^2 + n) \cdot x - 2n^2 \cdot x^2$ maximiert.

Bilde 1. Ableitung und setze auf 0 : $0 = n^2 + n - 4n^2 \cdot x$

$\Rightarrow$ Maximum liegt bei $\frac{n^2 + n}{4n^2} \approx \frac{1}{4} n$

Also können $(\frac{3}{4} n - \frac{1}{4} n + 1) \cdot \frac{1}{4} n = \frac{1}{8} n^2 + \frac{1}{4} n$ Vergleiche entstehen.

$\Rightarrow$ Die Laufzeit im worst case beträgt $O(n^2)$.

Beispiel für Analyse eines rekursiven Programms

Die Laufzeit von *fib* (n) beträgt:

$$f(n) = \begin{cases} 1, & \text{falls } n \le 1 \\ f(n-1) + f(n-2) + 1 & \text{sonst} \end{cases}$$

Offenbar gilt: $f \in O(\alpha^n)$ für ein $\alpha < 2$ (denn $f(n) = f(n-1) + f(n-1)$ würde zu $O(2^n)$ führen).

Gesucht ist also ein α, so daß für große n gilt:

$$\alpha^n = \alpha^{n-1} + \alpha^{n-2} + 1$$

Teile durch α^{n-2}:

$\Rightarrow$ $\alpha^2 = \alpha + 1 + \frac{1}{\alpha^{n-2}}$. Für $n \to \infty$ und $\alpha > 1$ geht $\frac{1}{\alpha^{n-2}} \to 0$.

$\Rightarrow$ $\alpha^2 = \alpha + 1 \Rightarrow \alpha = \frac{1}{2} + \sqrt{\frac{1}{4} + 1} = 1.61803$ (gemäß Formel $-\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$)

$\Rightarrow$ Das rekursive Fibonacci-Programm hat die Laufzeit $O(1.62^n)$, wobei n der Wert des Inputs ist, nicht seine Länge!

für $n = 20$ ist $1.62^n \approx 15.000$
$2^n \approx 1.000.000$.

6.2. Korrektheit und Terminierung

Durch *Testen* kann nachgewiesen werden, daß sich ein Programm für **endlich viele** Eingaben korrekt verhält. Durch eine *Verifikation* kann nachgewiesen werden, daß sich ein Programm für **alle** Eingaben korrekt verhält.

Bei der *Zusicherungsmethode* sind zwischen den Statements sogenannte Zusicherungen eingestreut, die eine Aussage darstellen über die momentane Beziehung zwischen den Variablen.

z.B. $\{i > 0 \wedge z = i^2\}$

Aus einer Zusicherung und der folgenden Anweisung läßt sich dann eine weitere Zusicherung ableiten:

```
i := i - 1;
```

$\Rightarrow \{i \ge 0 \wedge z = (i+1)^2\}$

Bei Schleifen wird die Zusicherung P, die vor Eintritt und vor Austritt gilt, die *Schleifeninvariante* genannt.

```
{P}
WHILE Q DO
   {P ∧ Q}
    :
    :
    :
   {P}
END;
{P ∧ ¬ Q}
```

Beginnend mit einer ersten, offensichtlich richtigen Zusicherung, läßt sich als letzte Zusicherung eine Aussage über das berechnete Ergebnis ableiten = *partielle Korrektheit* .

Zusammen mit dem Nachweis der *Terminierung* ergibt sich die *totale Korrektheit* .

Beispiel für Zusicherungsmethode

```
VAR n, x, y, z : CARDINAL;
ReadCard(n);
```
$\{n \geq 0\}$
```
x := 0; y := 1; z := 1;
```
$\{z = (x + 1)^2 \wedge y = 2x + 1 \wedge x^2 \leq n\} \quad (=P)$
```
WHILE z <= n DO
```
$(=Q)$

$\{z = (x + 1)^2 \wedge y = 2x + 1 \wedge x^2 \leq n\} \wedge \{z \leq n\} \Rightarrow \{(x+1)^2 \leq n\}$
```
    x := x + 1;
```
$\{z = x^2 \wedge y = 2x - 1 \wedge x^2 \leq n\}$
```
    y := y + 2;
```
$\{z = x^2 \wedge y = 2x + 1 \wedge x^2 \leq n\}$
```
    z := z + y;
```
$\{z = x^2 + 2x + 1 \wedge y = 2x + 1 \wedge x^2 \leq n\}$

$\{z = (x + 1)^2 \wedge y = 2x + 1 \wedge x^2 \leq n\} \quad (= P)$
```
END;
```
$\{z = (x + 1)^2 \wedge y = 2x + 1 \wedge x^2 \leq n\} \wedge \{z > n\} \quad (= P \wedge \neg Q)$

$\{x^2 \leq n < (x + 1)^2\}$

$\{x = \lfloor \sqrt{n} \rfloor \}$
```
WriteCard (x, 6);
```
$\{Ausgabe : \lfloor \sqrt{n} \rfloor \}$

Terminierung

Da y immer positiv ist und z immer um y erhöht wird, muß irgendwann gelten $z > n$

$\Rightarrow$ Abbruch gesichert

$\Rightarrow$ totale Korrektheit.

Laufzeit

In jedem Schleifendurchlauf wird x um eins erhöht.

Da x bei 0 beginnt und bei $\lfloor \sqrt{n} \rfloor$ endet, wird der Schleifenrumpf $\sqrt{n}$ mal durchlaufen. Der Schleifenrumpf selbst hat eine konstante Anzahl von Schritten.

$\Rightarrow$ Laufzeit $O(n^{1/2})$, wobei n der Wert der Eingabezahl ist!

Weiteres Beispiel für die Zusicherungsmethode

```
ReadCard (n);
{n ≥ 0}
x := 2; y := n; z := 1;
{z · x^y = 2^n}
WHILE y > 0 DO
   {z · x^y = 2^n ∧ y > 0}
   IF ODD(y) THEN
      {z · x^y = 2^n ∧ y > 0 ∧ y ungerade}
      z := z * x;
      {z/x · x^y = 2^n ∧ y > 0 ∧ y ungerade}
      y := y - 1
      {z/x · x^(y+1) = 2^n ∧ y ≥ 0}
      {z · x^y = 2^n ∧ y ≥ 0}
   ELSE
      {z · x^y = 2^n ∧ y > 0 ∧ y gerade}
      x := x * x;
      {z · (x^½)^y = 2^n ∧ y > 0 ∧ y gerade}
      y := y DIV 2;
      {z · (x^½)^(2y) = 2^n ∧ y ≥ 0}
      {z · x^y = 2^n ∧ y ≥ 0}
   END
   {z · x^y = 2^n ∧ y ≥ 0}
END;
{z · x^y = 2^n ∧ y = 0} ⇒ {z = 2^n}
WriteCard(z, 6);
{Ausgabe: 2 hoch n}
```

Terminierung

In jedem Schleifendurchlauf wird y kleiner $\Rightarrow$ irgendwann einmal Null $\Rightarrow$ Abbruch.

Laufzeit

Die Dualzahldarstellung von y schrumpft spätestens nach 2 Schleifendurchläufen um eine Stelle

$\Rightarrow$ $O(\log n)$ Durchläufe, wobei n der Wert der Eingabezahl ist, d.h. $O(n)$, wenn n die Länge der Eingabe ist.

Hinweis

Der naive Ansatz

```
FOR i := 1 TO n DO  x := 2 * x END;
```

hat Laufzeit $O(n)$, wenn n der Wert der Eingabezahl ist, d.h. $O(2^n)$, wenn n die Länge der Eingabe ist.

6.3. Halteproblem

Beh.: Es gibt kein Modula-Programm, welches entscheidet, ob ein gegebenes Programm, angesetzt auf einen gegebenen Input, anhält.

Beweis durch Widerspruch

Annahme: Es gibt eine

```
PROCEDURE haltetest (s, t : String) : BOOLEAN;

(* liefert TRUE, falls das durch den String s dargestellte
Modula-Programm bei dem durch String t dargestellten Eingabedaten
anhaelt, liefert FALSE, sonst *)
```

Dann läßt sich folgendes Modula-Programm konstruieren:

```
PROCEDURE quer (s : String);
BEGIN
   IF haltetest (s,s) THEN REPEAT UNTIL FALSE;
   END
END quer;
```

Sei `q` der String, der die Prozedur `quer` zum Inhalt hat.
Was passiert bei Aufruf von `quer(q)`?

1. Fall:	Hält an	⇒ `haltetest(q,q) = FALSE`
		⇒ `quer` angesetzt auf `q` hält nicht!
2. Fall:	Hält nicht	⇒ `haltetest(q,q) = TRUE`
		⇒ `quer` angesetzt auf `q` hält an!

⇒ Also kann es die Prozedur `haltetest` nicht geben!

7. Sortieren

Motivation für Sortieren:

1.) Häufiges Suchen
1mal sortieren, dann jeweils $\log n$ Aufwand.

2.) Tritt ein Element in zwei Listen L_1, L_2 auf?
Sortiere $L_1 \cdot L_2$, dann nach Doppelten suchen!

7.1. Selection Sort

''Hole jeweils das kleinste Element nach vorne''

```
VAR a       : ARRAY [0..n - 1] OF REAL;
    x       : REAL;
    i, j, k : INTEGER;
   :
FOR i := 0 TO n - 1 DO
   (* Berechne k := Index des kleinsten
      Elementes von a_i, a_{i+1}, . . . , a_{n-1}
      Vertausche a_i mit a_k *)
   k := i;     (* vorl. Index des kleinsten *)
   x := a[i];  (* vorl. Wert des kleinsten *)
   FOR j := i + 1 TO n - 1 DO
      IF a[j] < x THEN
                         k := j;
                         x := a[j]
                      END
   END;
   a[k] := a[i];
   a[i] := x;
END;
```

Beispiel

4 9 3 2 5
2 9 3 4 5
2 3 9 4 5
2 3 4 9 5
2 3 4 5 9

Analyse für *Selection Sort*

Worst case und *average case* :
2 ineinander geschachtelte `FOR`-Schleifen

$$n-1+n-2+n-3+\cdots+1$$
$$=\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Platzbedarf: $O(n)$
zusätzlich zu den Daten: $O(1)$

Der Algorithmus wird nicht schneller, wenn die Zahlen bereits sortiert sind!

7.2. Bubblesort

Sortieren mit *Bubblesort* = Blasen steigen auf
''Vertausche jeweils unsortierte Nachbarn''

```
4 9 3 2 5
  3 9
    2 9
4 3 2 5 9
3 4
  2 4
3 2 4 5 9
   :
   :

VAR getauscht : BOOLEAN;
   :
   :
REPEAT
   getauscht := FALSE;
   FOR i := 0 TO n - 2 DO
      IF a[i] > a [i + 1] THEN
                              tausche (a[i], a[i + 1]);
                              getauscht := TRUE
      END
   END;
UNTIL NOT getauscht;
```

Analyse von *Bubblesort*

Best case: $O(n)$
Worst case: $O(n^2)$

Die kleinste Zahl wandert in der `FOR`-Schleife jeweils um eine Position nach links.
Wenn sie zu Beginn ganz rechts steht, so sind $n - 1$ Phasen nötig.

average case: $O(n^2)$

Weitere Verbesserungen möglich (*Shaker Sort*), aber es bleibt bei $O(n^2)$
Grund: Austauschpositionen liegen zu nahe beieinander.

7.3. Mergesort

Idee (rekursiv formuliert):

```
Sortiere die vordere Hälfte der Folge;
Sortiere die hintere Hälfte der Folge;
Mische die beiden sortierten Folgen zu einer sortierten Folge
```

```
(****************************************************************************************)
(*      Datum:  29.11.93  14:15 Uhr                                                    *)
(*      Autor:  Oliver Vornberger                                                      *)
(*      Modul:  merge.m2                                                               *)
(*                                                                                     *)
(*              mischt eine sortierte  Folge der Laenge N                              *)
(*              und eine sortierte  Folge der Laenge M                                 *)
(*              zu einer sortierten Folge der Laenge N+M                               *)
(*                                                                                     *)
(****************************************************************************************)

MODULE merge;

CONST   N= 20;
        M= 30;

VAR     a: ARRAY[0..N-1]   OF REAL;
        b: ARRAY[0..M-1]   OF REAL;
        c: ARRAY[0..N+M-1] OF REAL;


VAR i,j,k : CARDINAL;

BEGIN
        i:=0; j:=0; k:=0;

        WHILE (i<N) AND (j<M) DO       (* solange mischen, bis ein array leer wird     *)
                IF a[i] < b[j]
                        THEN  c[k] := a[i]; INC(i);
                        ELSE  c[k] := b[j]; INC(j);
                END;
                INC(k)
        END;

        IF (i=N)                                     (* nichtleeres array kopieren    *)

                THEN WHILE j<M DO c[k] := b[j]; INC(j); INC(k) END
                ELSE WHILE i<N DO c[k] := a[i]; INC(i); INC(k) END
        END;


END merge.

(****************************************************************************************)
(* Laufzeit:    O(n)  fuer n=N+M Daten,                                                *)
(*              denn es werden N+M Elemente generiert                                  *)
(* Aber:        O(n) zusaetzlicher Platz                                               *)
(****************************************************************************************)
```

Analyse von *Mergesort* (und ähnlich gelagerte Rekursionen)

$$f(n) \le \begin{cases} c_1, & \text{für } n = 1 \\ 2 \cdot f(\frac{n}{2}) + c_2 \cdot n & \text{sonst} \end{cases}$$

Beh.: $f \in O(n \cdot \log n)$

Zeige: $f(n) \le (c_1 + c_2) \cdot n \cdot \log n + c_1$

Verankerung:

$n = 1 \Rightarrow \quad f(1) \le c_1$ nach Rekursion

$f(1) \le (c_1 + c_2) \cdot 1 \cdot \log 1 + c_1$

Induktionsschluß

Sei bis $n - 1$ bewiesen

$$\begin{aligned} f(n) \quad & \le 2 \cdot f(\frac{n}{2}) + c_2 \cdot n \\ & \uparrow \\ & \text{Rek.} \\ & \le 2 \cdot [(c_1 + c_2) \cdot \frac{n}{2} \cdot \log \frac{n}{2} + c_1] + c_2 \cdot n \\ & \uparrow \\ & \text{Ind.-Annahme} \\ & = 2 \cdot [(c_1 + c_2) \cdot \frac{n}{2} \cdot (\log n - 1) + c_1] + c_2 \cdot n \\ & = (c_1 + c_2)\, n \cdot \log n - (c_1 + c_2) \cdot n + 2c_1 + c_2 \cdot n \\ & = [(c_1 + c_2)\, n \cdot \log n + c_1] + [c_1 - c_1 \cdot n] \\ & \le (c_1 + c_2)\, n \cdot \log n + c_1 \end{aligned}$$

Aber: *Mergesort* benötigt $O(n)$ zusätzlichen Platz!

Iterative Version von *Mergesort* (für $n = 2^k$)

```
l := 1;         (* Laenge der Teilfolgen *)
k := n;         (* Anzahl der Teilfolgen *)
WHILE k > 1 DO
   {alle Teilfolgen der Länge l sind sortiert}
   {Sie beginnen bei l · i für i = 0, 1, · · · , k − 1}
   Mische je zwei benachbarte Teilfolgen der Laenge l
   zu einer Teilfolge der Laenge 2 * l
   l := 2 * l; k := k DIV 2;
   {alle Teilfolgen der Länge l sind sortiert}
   {Sie beginnen bei l · i für i = 0, 1, · · · , k − 1}
END;
```

Es ergeben sich $\log n$ Phasen mit jeweils linearem Aufwand.

$\Rightarrow O(n \cdot \log n)$

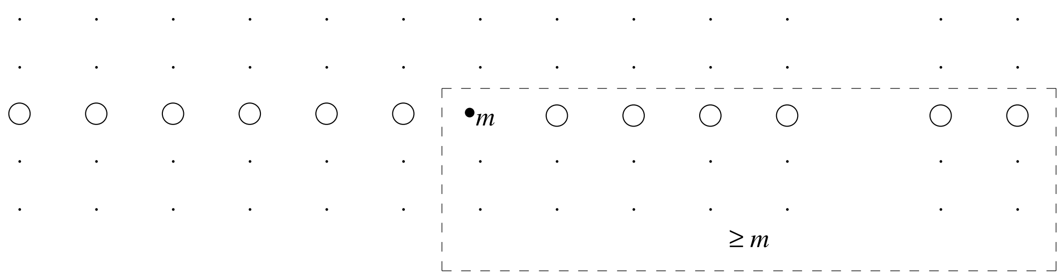
7.4. Quicksort

```
(****************************************************************************************)
(*      Datum:  29.11.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  quicksort.m2                                                            *)
(*              rekursives Sortieren                                                    *)
(*              Bilde eine kleinere und eine groessere Haelfte und sortiere diese       *)
(*                                                                                      *)
(****************************************************************************************)

MODULE quicksort;

CONST   N =     1000;
VAR     a :     ARRAY [0..N-1] OF REAL;
        i :     INTEGER;

PROCEDURE quicksort (unten,oben : INTEGER);             (* sortiert das array a         *)
VAR i,j : INTEGER;                                      (* in den Grenzen unten .. oben *)
    w,x : REAL;
BEGIN
        i := unten;
        j := oben;
        x := a[(unten+oben) DIV 2];

        REPEAT                                  (* Bilde Partition anhand des Element x *)
                WHILE a[i] < x DO i:=i+1 END;
                WHILE a[j] > x DO j:=j-1 END;
                (* x fungiert als Bremse *)
                IF i<=j THEN
                        w    := a[i];
                        a[i] := a[j];
                        a[j] := w;
                        i    := i+1;
                        j    := j-1
                END;
        UNTIL i > j;

        (* es ist eine kleinere und eine groessere Haelfte enstanden *)
        IF unten < j THEN quicksort(unten,j) END;
        IF i < oben  THEN quicksort(i,oben)  END;

END quicksort;

BEGIN

END quicksort.

(****************************************************************************************)
(*      Worst case:   O(n*n),      wenn Partitionen entarten: 1 Teil und n-1 Teile      *)
(*      best case:    O(n*log n),  wenn Partitionen immer gleich gross sind             *)
(*      average case: O(n*log n),  nur um den Faktor 1.4 schlechter als best case !     *)
(****************************************************************************************)
```

7.5. Bestimmung des Medians

```
PROCEDURE select (S : Menge; k : CARDINAL) : REAL;
(* liefert aus der Menge S das k - t kleinste Element. *)
(* Die Menge S habe n Elemente.                        *)
BEGIN
   IF |S| < 50 THEN RETURN k-t kleinstes per Hand
   ELSE
      sei n := |S|
      zerlege S in Gruppen zu je 5 Elementen S_1,S_2, ··· S_{n/5}
      Bestimme in jeder Gruppe S_i den Median m_i
      Bestimme den Median der Mediane
      m := select (⋃_i m_i, n/10)
      Bilde
         A := {x | x < m}
         B := {x | x = m}
         C := {x | x > m}

      IF    |A|         >= k THEN RETURN select (A, k)
      ELSIF |A|+|B|     >= k THEN RETURN m
      ELSIF |A|+|B|+|C| >= k THEN RETURN select (C, k-|A|-|B|)
      END
   END
END select;
```

Beh.: $|A| \le \frac{3}{4}n$



d.h. mind. $\frac{1}{4}$ von S ist $\ge m$

$\Rightarrow$ höchstens $\frac{3}{4}$ von S ist $< m$

$\Rightarrow |A| \le \frac{3}{4}\, n$, analog $|C| \le \frac{3}{4}\, n$.

Sei $f(n)$ die Anzahl der Schritte, die die Prozedur `select` benötigt für eine Menge S der Kardinalität n.
Also:

$$f(n) \leq \begin{cases} c, & \text{für } n < 50 \\ \underbrace{c \cdot n}_{\substack{\text{Mediane} \\ \text{der 5er Gruppe}}} + \underbrace{f(\frac{n}{5})}_{\substack{\text{Median} \\ \text{der Mediane}}} + \underbrace{f(\frac{3}{4}n)}_{\substack{\text{Aufruf mit} \\ A \text{ oder } C}} & \text{sonst} \end{cases}$$

Beh.: $f \in O(n)$
Zeige: $f(n) \leq 20 \cdot c \cdot n$

Beweis durch Induktion:

$f(n) \leq c \leq 20 \cdot c \cdot n$ für $n < 50$
Sei bis $n - 1$ bewiesen

$$\begin{aligned} f(n) &\underset{\text{Rek.gl.}}{\leq} c \cdot n + f(\tfrac{n}{5}) + f(\tfrac{3}{4}n) \\ &\underset{\text{Ind.-Annahme}}{\leq} c \cdot n + 20 \cdot c \cdot \frac{n}{5} + 20 \cdot c \cdot \frac{3}{4} \cdot n \\ &= 1 \cdot c \cdot n + 4 \cdot c \cdot n + 15 \cdot c \cdot n \\ &= 20 \cdot c \cdot n \end{aligned}$$

q.e.d.

7.6. Heapsort

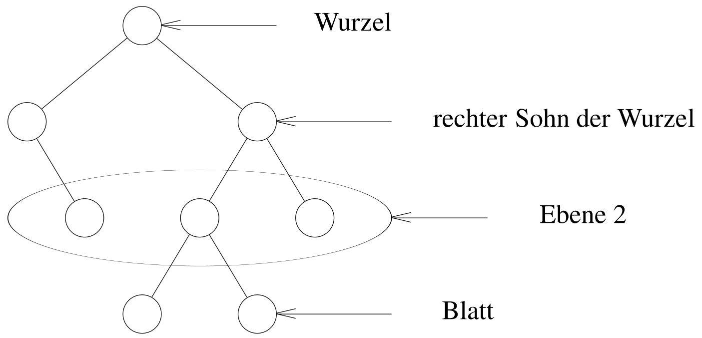
Idee für *Heapsort*: Verwendet wird ein Heap = Datenstruktur, die das Entfernen des Minimums unterstützt (siehe unten).

Gegeben $a_1, ..., a_n$.
Baue einen Heap mit den Marken $a_1, ..., a_n$.

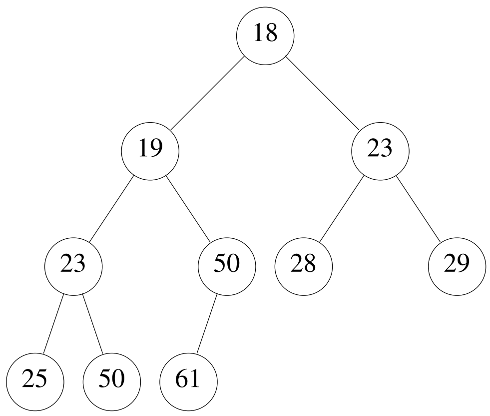
```
REPEAT
   entferne Wurzel (=Minimum)
   reorganisiere Heap
UNTIL Heap ist leer
```

Baum und Heap

Def.: Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Dieser heißt dann *Vater* des linken bzw. rechten Teilbaums. Ein Knoten ohne Vater heißt *Wurzel*. Die Knoten, die x zum Vater haben, sind seine *Söhne*. Knoten ohne Söhne heißen *Blätter*.
Ebene 0 = Wurzel. Ebene $i + 1$ = Söhne von Ebene i.



Def.: Ein *Heap* ist ein binärer Baum, in dem die Ebenen $0, 1, \cdots i-1$ vollständig sind; Ebene i ist von links beginnend bis zum sogenannten letzten Knoten vollständig. Die Knoten sind markiert. Die Marke eines Knotens ist kleiner oder gleich der Marke seiner Söhne.



Offenbar steht das kleinste Element eines Heaps in der Wurzel.

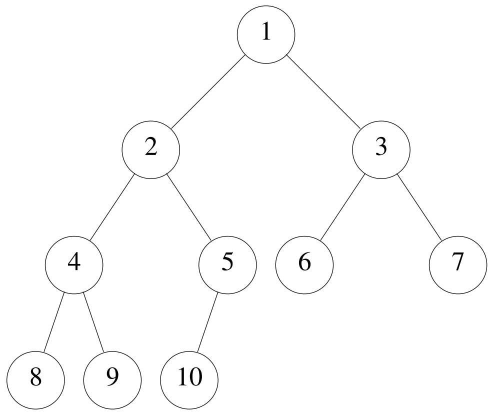
Idee für Wurzelentfernen

Entferne ''letzten'' Knoten im Heap und schreibe seinen Schlüssel in die Wurzel.

Vertausche so lange Knoten mit kleinerem Sohn bis Heapbeziehung eintritt.

Problem: Wie implementiert man einen Heap?

Lösung: Die Knoten werden wie folgt numeriert:
Wurzel erhält Nr. 1,
linker Sohn von Knoten i erhält die Nr. $2 \cdot i$
rechter Sohn von Knoten i erhält die Nr. $2 \cdot i + 1$



Die Marke von Knoten i sei `a[i]` für

```
VAR a : ARRAY [1..N] OF REAL;
```

Vorteil: Die Vater/Sohn-Beziehung ist allein aus dem Knotenindex zu berechnen.
$2i$ bzw. $2i + 1$ heißen die Söhne von i
i *DIV* 2 heißt der Vater von i.

```
(***************************************************************************************)
(*      Datum:  07.12.93  14:15 Uhr                                                   *)
(*      Autor:  Oliver Vornberger                                                     *)
(*      Modul:  heapsort.m2                                                           *)
(*              Sortieren durch Austauschen mit Hilfe eines Heaps                     *)
(*                                                                                    *)
(***************************************************************************************)

MODULE heapsort;

CONST   N       = 100;                                      (* maximale Heapgroesse *)
TYPE    Element = REAL;                                     (* Typ fuer Heapeintrag *)
        Heap    = ARRAY [1..N] OF Element;                  (* Typ fuer Heap        *)
VAR     a       : Heap;                                     (* Instanz fuer Heap    *)
        n       : CARDINAL;                                 (* aktuelle Heapgroesse *)

PROCEDURE sift(L,R : CARDINAL);      (* betrachte array-Komponenten L,L+1,...,R als Heap *)
                                       (* wobei ggf. die Wurzel L angepasst werden muss *)
VAR i,j : CARDINAL;
      x : Element;
BEGIN
        i := L; x := a[L];
        j := 2*L;                                                      (* linker Sohn *)
        IF (j<R) AND (a[j+1] < a[j]) THEN j:=j+1 END;        (* j ist der kleinere Sohn *)

        WHILE (j<=R) AND (a[j]<x) DO                (* Vater tauscht mit kleinerem Sohn *)
                a[i] := a[j];
                i := j;
                j := j*2;
                IF (j<R) AND (a[j+1] < a[j])
                THEN j := j+1 END;     (* j ist der kleinere Sohn, Vater hat den Wert x *)
        END;
        a[i] := x;
END sift;


PROCEDURE heapsort;                                  (* sortiert n Elemente im array a *)
VAR L,R : CARDINAL;
      x : Element;
BEGIN
        FOR L := n DIV 2 TO 1 BY -1 DO sift(L,n) END;                  (* Heap aufbauen *)

        FOR R := n TO 2 BY -1 DO                 (* n mal das kleinste Element entfernen *)
                x    := a[1];                                (* kleinstes Element holen *)
                a[1] := a[R];                              (* letztes Element nach vorn *)
                a[R] := x;                                     (* kleinstes nach hinten *)
                sift(1,R-1);                                (* Heap korrigieren bis R-1 *)
        END
END heapsort;

BEGIN   (*                                 _1_________R_________________________n_ *)
        (* array a mit n Elementen fuellen |x| | | | | | | | | | | | | | | | | | | *)
        heapsort; (*                                   |--> absteig. sortiert <--| *)
        (* array a ist nun sortiert *)
END heapsort.
```

Aufwand für die Konstruktion eines Heaps

Sei $n - 1 = 2^h - 1$ die Anzahl der Elemente, z.B. $15 = 2^4 - 1$.

Ebene	Sickertiefe	Anzahl
$h - 1$	1	$\frac{n}{2}$
$h - 2$	2	$\frac{n}{4}$
$h - 3$	3	$\frac{n}{8}$
$\vdots$	$\vdots$	
0	h	$\frac{n}{2^h} = 1$

Anzahl der Schritte $\sum_{i=1}^{h} c \cdot i \cdot \frac{n}{2^i} \quad = \quad c \cdot n \cdot \sum_{i=1}^{h} \frac{i}{2^i}$

$\Rightarrow$ Aufwand $O(n)$, denn: $\quad \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots < 2$

Aufwand für einmaliges Minimumentfernen $O(\log n)$

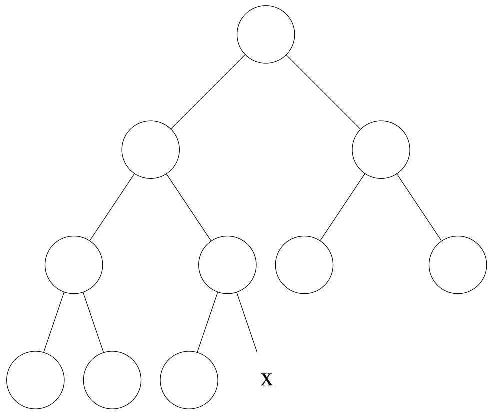
$\Rightarrow$ **Gesamtaufwand** $O(n) + O(n \cdot \log n) = O(n \cdot \log n)$
für best, average und worst case.

Weitere Einsatzmöglichkeit des Heaps

Verwende eine dynamisch sich ändernde Menge von Schlüsseln mit den Operationen

- `initheap` legt leeren Heap an
- `get_min` liefert das momentan Kleinste
- `del_min` entfernt das momentan Kleinste
- `insert(x)` fügt x hinzu
- `heapleer` testet, ob Heap leer ist

Idee für Einfügen: (schlecht: von oben nach unten)
besser: Füge neues Blatt mit Schlüssel x an.

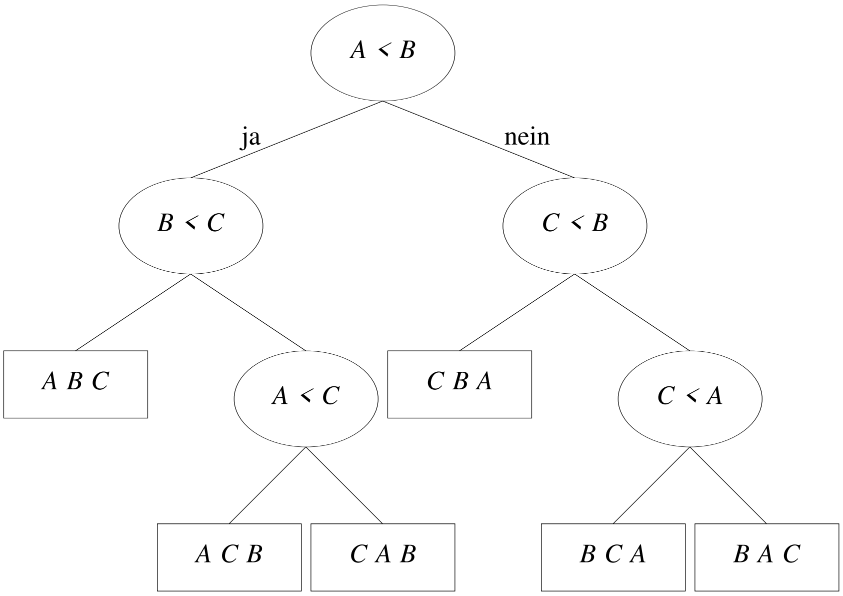



```
i := neuer Index;
WHILE a[i] < a [vater[i]] DO
   tausche (a[i], a[vater[i]]);
   i := vater[i]
END;
```

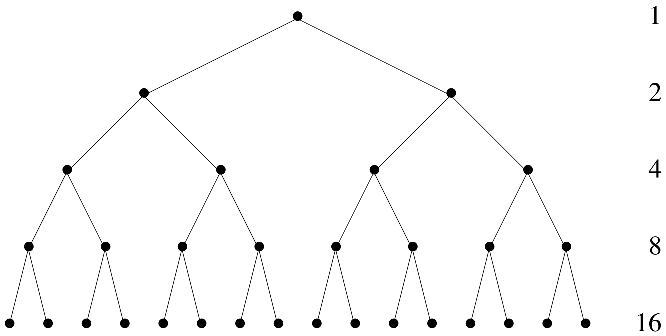
Aufwand $O(\log n)$

7.7. Untere Schranke für Sortieren durch Vergleichen

Entscheidungsbaum zur Sortierung von 3 Elementen:
gegeben A, B, C



Der Entscheidungsbaum zur Sortierung von n Elementen hat $n!$ Blätter.



Höhe = $\log k$ bei k Blättern

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdots \frac{n}{2} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$\begin{aligned}
\Rightarrow \log n! \quad &\geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \frac{n}{2}(\log n - 1) \\
&= \frac{n}{2} \cdot \log n - \frac{n}{2} = \frac{n}{4} \cdot \log n + \frac{n}{4} \cdot \log n - \frac{n}{2} \\
&\geq \frac{n}{4} \cdot \log n \quad \text{für } n \geq 4
\end{aligned}$$

$\Rightarrow$ Ein binärer Baum mit $n!$ Blättern hat mindestens die Höhe $\frac{n}{4} \cdot \log n$.

$\Rightarrow$ Jeder Sortieralgorithmus, der auf Vergleichen beruht, hat als Laufzeit mindestens $O(n \cdot \log n)$. Dies ist eine untere Schranke.

7.8. Bucket Sort

```
(*****************************************************************************)
(*      Datum:  20.12.93  14:15 Uhr                                          *)
(*      Autor:  Oliver Vornberger                                            *)
(*      Modul:  bucketsort.m2                                                *)
(*              Sortieren durch Verteilen auf Faecher                        *)
(*                                                                           *)
(*****************************************************************************)

MODULE bucketsort;

FROM InOut IMPORT WriteString, WriteLn, Read, Write;

CONST   EOL =   12C;    (* Zeilenende *)

VAR     c   :   CHAR;
        i,j :   CARDINAL;
        b   :   ARRAY ['a' .. 'z'] OF CARDINAL;

BEGIN

        FOR c:='a' TO 'z' DO b[c] := 0 END;             (* Buckets initialisieren      *)

        WriteString("Bitte String eingeben: ");
        Read(c);
        WHILE c <> EOL DO

                INC(b[c]);                     (* das fuer c zustaendige Bucket erhoehen *)
                Read(c);                       (* naechstes Zeichen lesen                *)
        END;

        WriteString("String sortiert:       ");

        FOR c:='a' TO 'z' DO
                FOR j:=1 TO b[c] DO Write(c) END;              (* alle Buckets auslesen *)
        END;
        WriteLn;

END bucketsort.

(*****************************************************************************)
(* Sei A die Menge der moeglichen Schluessel                                 *)
(* Laufzeit, um n Schluessel zu sortieren:  O(|A|) + O(n)                    *)
(* Fuer A = {a,b,c,...,z} und n > 26 ==>    Laufzeit O(n)                    *)
(*****************************************************************************)
```

7.9. Radix Sort

Idee: Sortieren von Strings über einem endlichen Alphabet durch mehrmaliges Anwenden von Bucket Sort.

Es soll eine Liste von Wörtern mit insgesamt n Buchstaben in $O(n)$ sortiert werden. Pro Buchstabe des Alphabets existiert ein Bucket.

Zunächst wird angenommen, daß alle Wörter die Länge N haben.

Es werden sukzessive Listen $W_N, W_{N-1}, ..., W_0$ gebildet.
W_j enthält alle Wörter, sortiert nach Positionen $j+1, ..., N$.
Die unsortierte Ausgangsliste ist W_N, das Ergebnis der Sortiervorgänge ist W_0.

```
FOR j := N TO 1 BY - 1 DO
   verteile die Wörter aus W_j
   gemäß j-tem Buchstaben
   auf die Buckets;
   sammele Buckets auf nach W_{j-1}
END;
```

Beispiel

Zu sortieren seien die Strings

```
hefe   bach   gaga   cafe   geha
```

Es entstehen nacheinander folgende Listen (unterstrichen ist jeweils der Buchstabe, der zum Verteilen benutzt wird):

W_4:	$\texttt{hef}\underline{\texttt{e}}$	$\texttt{bac}\underline{\texttt{h}}$	$\texttt{gag}\underline{\texttt{a}}$	$\texttt{caf}\underline{\texttt{e}}$	$\texttt{geh}\underline{\texttt{a}}$
W_3:	$\texttt{ga}\underline{\texttt{g}}\texttt{a}$	$\texttt{ge}\underline{\texttt{h}}\texttt{a}$	$\texttt{he}\underline{\texttt{f}}\texttt{e}$	$\texttt{ca}\underline{\texttt{f}}\texttt{e}$	$\texttt{ba}\underline{\texttt{c}}\texttt{h}$
W_2:	$\texttt{b}\underline{\texttt{a}}\texttt{ch}$	$\texttt{h}\underline{\texttt{e}}\texttt{fe}$	$\texttt{c}\underline{\texttt{a}}\texttt{fe}$	$\texttt{g}\underline{\texttt{a}}\texttt{ga}$	$\texttt{g}\underline{\texttt{e}}\texttt{ha}$
W_1:	$\underline{\texttt{b}}\texttt{ach}$	$\underline{\texttt{c}}\texttt{afe}$	$\underline{\texttt{g}}\texttt{aga}$	$\underline{\texttt{h}}\texttt{efe}$	$\underline{\texttt{g}}\texttt{eha}$
W_0:	bach	cafe	gaga	geha	hefe

Die zugehörigen Buckets lauten:

	$W_4 \rightarrow W_3$	$W_3 \rightarrow W_2$	$W_2 \rightarrow W_1$	$W_1 \rightarrow W_0$
a	gaga geha		bach cafe gaga	
b				bach
c		bach		cafe
d				
e	hefe cafe		hefe geha	
f		hefe cafe		
g		gaga		gaga geha
h	bach	geha		hefe

Beim Aufsammeln werden die Buckets in alphabetischer Reihenfolge durchlaufen und ihre Inhalte konkateniert.

Um beim Einsammeln der Buckets nur die gefüllten anzufassen, ist es hilfreich, zu Beginn

```
VAR nichtleer : ARRAY [1..N] OF Liste;
```

zu berechnen. `nichtleer`[j] enthält die Buchstaben, welche in den zu sortierenden Wörtern an j-ter Position vorkommen. Verteile hierzu zunächst Positionen auf Buchstaben:

`pos` [x] = {j | Buchstabe x kommt an j-ter Position vor}

```
a | 2 2 4 2 4
b | 1
c | 3 1
d |
e | 2 4 4 2
f | 3 3
g | 1 3 1
h | 1 4 3
```

Nach dem Einsammeln der Buckets werden die Buchstaben auf Positionen verteilt:

```
1 | b c g g h
2 | a a a e e
3 | c f f g h
4 | a a e e h
```

Obacht: Bei der Konstruktion von `nichtleer` müssen doppelte Einträge vermieden werden (möglich, da sie hintereinander stehen würden).

Bei Vermeidung der Doppelten hat `nichtleer`[j] folgende Gestalt:

```
1 | b c g h
2 | a e
3 | c f g h
4 | a e h
```

Der Aufwand zur Konstruktion von `nichtleer` beträgt $O(n)$. Jedes Wort wird bzgl. jedes Buchstabens einmal verteilt und einmal aufgesammelt, jeweils in konstanter Zeit.

Bei Wörtern **unterschiedlicher** Länge bilde zunächst

`Laenge`[j] := {w | Länge von Wort $w = j$}

Der Aufwand hierfür beträgt $O(n)$.

Verteile im j-ten Durchlauf zunächst alle Wörter aus `Laenge`[j];
z.B. `fad` wird bzgl. des 3. Buchstabens verteilt, bevor W_3 verteilt wird.

Der Aufwand zum Verteilen und Aufsammeln der Listen $W_N,...,W_0$ beträgt $O(n)$, da jedes Zeichen einmal zum Verteilen benutzt wird und einmal beim Aufsammeln unter Vermeidung der nichtleeren Buckets zu einem Schritt beiträgt.

7.10. Externes Sortieren

Schlecht: Iteratives *Mergesort* mit 3 Bändern.

Besser: Gegeben eine Folge von Schlüsseln.

Ein *Lauf* ist eine monoton nicht fallende Teilfolge.

$$\underbrace{3 \quad 7 \quad 8}_{\text{Lauf}} \quad \underbrace{2 \quad 9}_{\text{Lauf}} \quad \underbrace{4}_{\text{Lauf}} \quad \underbrace{1 \quad 5 \quad 6}_{\text{Lauf}}$$

Gegeben 3 Magnetbänder, mit initial 0, n, 0 Läufen. Je 2 Bänder mischen ihre Läufe zusammen auf das dritte:

Band 1:	0	0	8	3	0	2	1	0
Band 2:	21	13	5	0	3	1	0	1
Band 3:	0	8	0	5	2	0	1	0
								sortiert

(In jeder Spalte ist die momentane Anzahl der Läufe vermerkt.)

Allgemeine Regel:

Sei $fib(i)$ die i-te Fibonacci-Zahl.

Es habe Band	A	$fib(i)$	Läufe
	B	$fib(i-1)$	Läufe
	C	0	Läufe

dann mische $fib(i-1)$ Läufe von A und B zusammen auf Band C.

8. Mehr Modula

8.1. Files

```
(****************************************************************************************)
(*      Datum:  20.12.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  files.m2                                                                *)
(*              demonstriert Umgang mit Dateien                                         *)
(*              liest Integers aus einer Datei und schreibt sie in eine andere Datei    *)
(*                                                                                      *)
(****************************************************************************************)

MODULE files;
FROM InOut IMPORT WriteString, WriteLn;
FROM Files IMPORT FILE, OpenRead, OpenWrite, Close;
FROM FtdIO IMPORT FreadInt, FwriteInt;

IMPORT Files;                                         (* um Files.Done zu kennen      *)
IMPORT FtdIO;                                         (* um FtdIO.Done zu kennen      *)

VAR     f,g : FILE;
        i   : INTEGER;
BEGIN

        OpenRead(f,"otto.dat");                       (* Datei zum Lesen oeffnen      *)
        IF NOT Files.Done THEN                        (* falls nicht erfolgreich:     *)
                WriteString("Fehler bei OpenRead");   (* Fehlermeldung ausgeben       *)
                WriteLn;
                RETURN                                (* Programm beenden             *)
        END;

        OpenWrite(g,"paul.dat");                      (* Datei zum Schreiben oeffnen  *)
        IF NOT Files.Done THEN                        (* falls nicht erfolgreich:     *)
                WriteString("Fehler bei OpenWrite");  (* Fehlermeldung ausgeben       *)
                WriteLn;
                RETURN                                (* Programm beenden             *)
        END;

        FreadInt(f,i);                                (* Integer lesen aus File f     *)
        WHILE FtdIO.Done DO                           (* solange erfolgreich          *)
                FwriteInt(g,i,6);                     (* Integer schreiben ins File g *)
                IF FtdIO.Done THEN FreadInt(f,i) END; (* Integer lesen aus File f     *)
        END;

        Close(f);
        IF NOT Files.Done THEN                        (* Dateien schliessen           *)
                WriteString("Fehler bei Close f");
                RETURN
        END;

        Close(g);
        IF NOT Files.Done THEN                        (* Dateien schliessen           *)
                WriteString("Fehler bei Close g");
                RETURN
        END;

END files.
```

8.2. Records

```
(****************************************************************************************)
(*      Datum:  21.12.93  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  records.m2                                                              *)
(*              demonstriert Umgang mit records                                         *)
(****************************************************************************************)

MODULE records;

FROM InOut IMPORT WriteCard, Write, WriteString, WriteLn;

CONST N     =   200;
TYPE Datum  =   RECORD
                        tag   : [1..31];
                        monat : [1..12];
                        jahr  : [1900..1999];
                END;

TYPE String =   ARRAY [0..20] OF CHAR;

TYPE Person =   RECORD
                        vorname    :    String;
                        nachname   :    String;
                        geschlecht :    BOOLEAN;        (* true = weiblich              *)
                        geburtstag :    Datum;
                END;
VAR     d1,d2       :   Datum;
        p           :   Person;
        hoersaal    :   ARRAY[0..N-1] OF Person;
        i           :   INTEGER;
BEGIN
        d1 := d2;                               (* kopiere komplettes Record           *)

        d1.tag   := d2.tag;                     (* kopiere die Komponente tag          *)
        d1.monat := d2.monat;                   (* kopiere die Komponente monat        *)
        d1.jahr  := d2.jahr;                    (* kopiere die Komponente jahr         *)

        WITH d1 DO                              (* alle Komponenten bezogen auf d1     *)
                tag   := 14;
                monat := 12;
                jahr  := 1993;
        END;

        p.vorname   := 'Erika';
        hoersaal[1].vorname := "Willi";
        hoersaal[1]  := p;

        FOR i:=0 TO N-1 DO                          (* gib fuer alle Frauen im Hoersaal *)
                WITH hoersaal[i] DO                    (* ihren Namen und ihr Alter aus *)
                        IF geschlecht THEN
                                WriteString (vorname);                  Write(' ');
                                WriteString (nachname);                 Write(' ');
                                WriteCard   (1993-geburtstag.jahr,2); WriteLn;
                        END
                END
        END;
END records.
```

8.3. Zeiger

```
(****************************************************************************************)
(*      Datum:  21.12.93  14:15 Uhr                                                    *)
(*      Autor:  Oliver Vornberger                                                      *)
(*      Modul:  pointer.m2                                                             *)
(*              Ein Zeiger verweist auf eine Variable bestimmten Typs.                 *)
(*              Die Variable ist anonym, d.h. nicht ueber einen eigenen                *)
(*              Namen zu erreichen und ihr Speicherplatz wird dynamisch                *)
(*              angefordert.                                                           *)
(*                                                                                     *)
(****************************************************************************************)

MODULE pointer;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;       (* erforderlich fuer NEW und DISPOSE    *)
FROM InOut   IMPORT WriteCard, WriteString;

TYPE Person =   RECORD
                        vorname, nachname : ARRAY[0..20] OF CHAR;
                        alter      : CARDINAL;
                        geschlecht : BOOLEAN;
                END;

TYPE Zeiger =   POINTER TO Person;

VAR     z1, z2 : Zeiger;

BEGIN

        NEW(z1);  NEW(z2);     (* Auf dem Heap werden 2 Bereiche vom Typ Person angelegt *)
                                              (* und z1 bzw. z2 zeigen jeweils dorthin *)


        z1^.vorname := "Willi";                        (* in dem Bereich, auf den z1 zeigt *)
                                                   (* wird die Komponente Vorname belegt *)


        z2^ := z1^;       (* Das, worauf z1 zeigt, wird dorthin kopiert, wohin z2 zeigt *)


        z2 := z1;                     (* Zeiger z2 zeigt dorthin, wohin Zeiger z1 zeigt *)
                           (* nun ist der Bereich, auf den z2 gezeigt hat, unerreichbar *)


        WriteString(z2^.vorname);                  (* von dem, worauf z2 zeigt, wird die *)
                                                       (* Komponente vorname ausgegeben *)


        z2 := NIL;                                                 (* z2 zeigt auf nichts *)

        DISPOSE(z1);                  (* der Platz, auf den z1 zeigt, wird freigegeben *)
                                       (* nun zeigen z1 und z2 auf etwas undefiniertes *)

END pointer.
```

```
(****************************************************************************)
(*      Datum:  5.12.95  14:15 Uhr                                          *)
(*      Autor:  Oliver Vornberger                                           *)
(*      Modul:  zeigerabzaehlreim.m2                                        *)
(*              N kinder stehen im Kreis                                    *)
(*              jedes K-te wird abgeschlagen                                *)
(*              (Realisierung durch Pointer)                                *)
(*                                                                          *)
(****************************************************************************)

MODULE  zeigerabzaehlreim;

FROM    InOut   IMPORT WriteCard, WriteString, WriteLn;
FROM    Storage IMPORT ALLOCATE, DEALLOCATE;

CONST   N = 20;                                     (* Anzahl der Kinder     *)
        K = 4;                                      (* jedes k-te abschlagen *)

TYPE    Kindzeiger = POINTER TO Kind;

        Kind       = RECORD
                        nr :  CARDINAL;
                        next : Kindzeiger
                     END;

VAR     erster, index, hilf : Kindzeiger;
        i                   : CARDINAL;

BEGIN
        NEW(erster);                            (* besorge Speicherplatz         *)
        erster^.nr := 0;                        (* vergib erste Nummer           *)
        index      := erster;                   (* verweise auf letztes Kind     *)
        FOR i:=1 TO N-1  DO                     (* haenge N-1 mal an             *)
                NEW (index^.next);              (* besorge Speicherplatz         *)
                index := index^.next;           (* setzte Verweis weiter         *)
                index^.nr       := i;           (* trage laufende Nummer ein     *)
        END;
        index^.next := erster;                  (* schliesse den Kreis           *)

        WHILE index <> index^.next DO           (* solange abschlagen, bis jemand *)
                                                (* sein eigener Nachfolger ist   *)
                FOR i:=1 TO K-1 DO              (* K-1 mal weitergehen           *)
                        index := index^.next;
                END;
                hilf := index^.next;            (* zeige auf abzuschlagendes Kind *)
                WriteString("Ausgeschieden ist ");  (* Gib aus:                  *)
                WriteCard(hilf^.nr,6);          (* Nr. des Kindes                *)
                index^.next := hilf^.next;      (* trage neuen Nachfolger ein    *)
                DISPOSE (hilf);                 (* gib Platz frei                *)
                WriteLn;
        END;
        WriteString("Uebrig geblieben ist ");
        WriteCard(index^.nr,3);
        WriteLn;

END zeigerabzaehlreim.
```

9. Abstrakte Datentypen

Def.: Ein *abstrakter Datentyp* ADT ist eine Datenstruktur zusammen mit darauf definierten Operationen.

Modula unterstützt den Umgang mit ADTs durch die Bereitstellung von Modulen.

Ein *Modul* besteht aus einem

- **DEFINITION MODULE** <name>
  hier werden dem Anwender die verfügbaren Konstanten, Typen, Variablen und Prozedurköpfe bekannt gemacht.
- **IMPLEMENTATION MODULE** <name>
  hier sind die im *definition module* angekündigten Prozeduren und ggf. noch weitere Prozeduren implementiert.

Im Hauptprogramm (oder einem anderen Modul) können die vom Modul *exportierten* Operationen, Konstanten, Typen und Variablen verwendet werden, sofern sie dort *importiert* wurden.

```
MODULE haupt;
FROM <name> IMPORT ...
```

Beispiel für das Übersetzen und Binden

error.d	enthält	Definition-Modul
error.m2	enthält	Implementation-Modul
liste.d	enthält	Definition-Modul
liste.m2	enthält	Implementation-Modul
listetest.m2	enthält	Aufrufe der Listenoperationen
m2c error.d	erzeugt	error.sy
m2c -c error.m2	benötigt	error.sy
	erzeugt	error.o
m2c liste.d	erzeugt	liste.sy
m2c -c liste.m2	benötigt	liste.sy error.sy
	erzeugt	liste.o
m2c -c listetest.m2	benötigt	liste.sy
	erzeugt	listetest.o
m2c error.o liste.o listetest.o	erzeugt	a.out

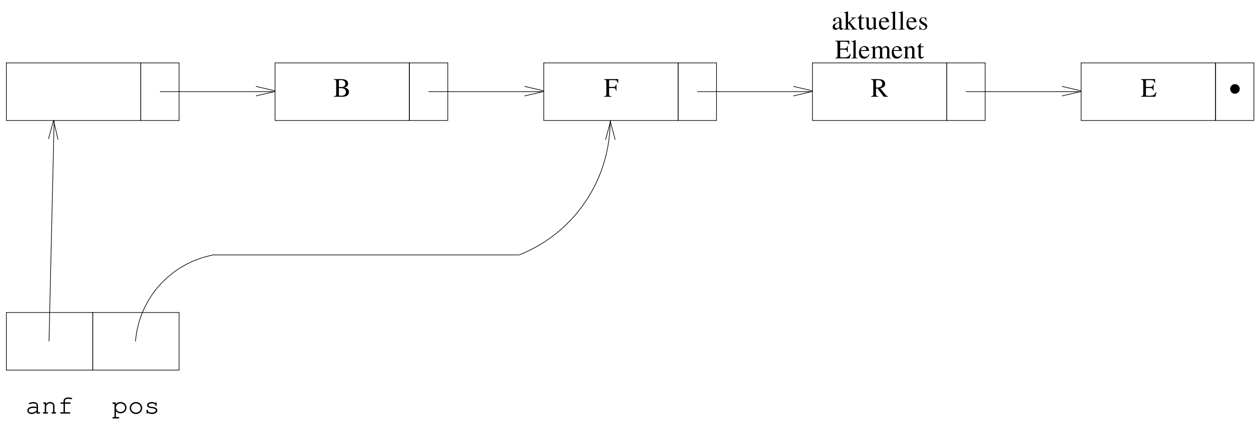
9.1. Liste

Def.: Eine *Liste* ist eine (ggf. leere) Folge von Elementen eines bestimmten Typs *T* zusammen mit einem sogenannten (ggf. undefinierten) *aktuellen* Element.

Schnittstelle des `ADT liste`:

`createlist`	:		$\rightarrow$	Liste	legt leere Liste an
`emptylist`	:	Liste	$\rightarrow$	Boolean	liefert true, falls Liste leer
`endpos`	:	Liste	$\rightarrow$	Boolean	liefert true, wenn Liste abgearbeitet
`advance`	:	Liste	$\rightarrow$	Liste	der Nachfolger des akt. wird zum aktuellen
`reset`	:	Liste	$\rightarrow$	Liste	das erste Listenelement wird zum aktuellen
`elem`	:	Liste	$\rightarrow$	Element	liefert das aktuelle Element
`insert`	:	Liste $\times$ T	$\rightarrow$	Liste	fügt vor das aktuelle Element ein Element ein; das neu eingefügte wird zum aktuellen
`delete`	:	Liste	$\rightarrow$	Liste	löscht das aktuelle Element; der Nachfolger wird zum aktuellen

Implementation einer Liste mit Zeigern



`anf` zeigt auf das erste Record (leerer Inhalt),
`pos` zeigt auf das Record **vor** dem Record mit dem aktuellen Element.

```
(******************************************************************************)
(*                                                                            *)
(*      Datum:  11.01.94  14:15 Uhr                                           *)
(*      Autor:  Oliver Vornberger                                             *)
(*      Modul:  error.d                                                       *)
(*              Routine zur Fehlerbehandlung                                  *)
(*                                                                            *)
(******************************************************************************)

DEFINITION MODULE error;

PROCEDURE error (e: ARRAY OF CHAR);     (* druckt die Fehlermeldung e und haelt an      *)

END error.
```

```
(******************************************************************************)
(*                                                                            *)
(*      Datum:  11.01.94  14:15 Uhr                                           *)
(*      Autor:  Oliver Vornberger                                             *)
(*      Modul:  error.m2                                                      *)
(*              Routine zur Fehlerbehandlung                                  *)
(*                                                                            *)
(******************************************************************************)

IMPLEMENTATION MODULE error;

FROM InOut IMPORT WriteString, WriteLn;

PROCEDURE error (e: ARRAY OF CHAR);     (* druckt die Fehlermeldung e und haelt an      *)
BEGIN
        WriteString(e);
        WriteLn;
        HALT
END error;

END error.
```

```
(****************************************************************************************)
(*      Datum:  10.01.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  liste.d                                                                 *)
(*              fuehrt den Typ Liste ein und die darauf definierten Operationen         *)
(*                                                                                      *)
(****************************************************************************************)

DEFINITION MODULE liste;

TYPE Listenelement = INTEGER;                           (* Typ fuer Listeninhalte *)

TYPE Liste;                                (* Typ in opaker (= undurchsichtiger) Form *)

PROCEDURE createlist (VAR l : Liste);                        (* legt leere Liste an *)

PROCEDURE emptylist  (l : Liste) : BOOLEAN;         (* liefert TRUE, wenn Liste leer *)

PROCEDURE endpos     (l : Liste) : BOOLEAN;     (* liefert TRUE, wenn Liste am Ende *)

PROCEDURE advance    (l : Liste);                            (* rueckt in Liste vor *)

PROCEDURE reset      (l : Liste);                           (* rueckt an den Anfang *)

PROCEDURE elem       (l : Liste) : Listenelement;        (* liefert aktuelles Element *)

PROCEDURE insert     (l : Liste; x:Listenelement);               (* fuegt Element ein *)

PROCEDURE delete     (l : Liste);                        (* entfernt aktuelles Element *)

END liste.
```

```
(****************************************************************************************)
(*      Datum:  10.01.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  liste.m2                                                                *)
(*              implementiert den Typ Liste und die darauf definierten Operationen      *)
(****************************************************************************************)

IMPLEMENTATION MODULE liste;

FROM    Storage  IMPORT ALLOCATE, DEALLOCATE;                    (* fuer NEW und DISPOSE *)
FROM    error    IMPORT error;

TYPE Listenzeiger  = POINTER TO Listeneintrag;

     Listeneintrag = RECORD
                        inhalt : Listenelement;
                        next   : Listenzeiger;
                     END;

TYPE Liste         = POINTER TO
                     RECORD
                        anf:    Listenzeiger;                       (* zeigt auf Anfang *)
                        pos:    Listenzeiger;            (* zeigt vor akt. Listenelement *)
                     END;



PROCEDURE createlist (VAR l : Liste);                            (* legt leere Liste an *)
BEGIN
        NEW(l); NEW (l^.anf);
        l^.pos       := l^.anf;
        l^.anf^.next := NIL;
END createlist;


PROCEDURE emptylist (l : Liste) : BOOLEAN;                 (* liefert TRUE, wenn Liste *)
BEGIN                                                                      (* leer ist *)
        RETURN (l^.anf^.next = NIL);
END emptylist;


PROCEDURE endpos (l : Liste) : BOOLEAN;                   (* liefert TRUE, wenn Liste   *)
BEGIN                                                               (* abgearbeitet ist *)
        RETURN l^.pos^.next = NIL;
END endpos;


PROCEDURE advance (l : Liste);                                  (* rueckt in Liste vor *)
BEGIN
        IF NOT endpos(l) THEN l^.pos := l^.pos^.next;
                         ELSE error('in advance: am Ende')
        END;
END advance;
```

```
PROCEDURE reset (l : Liste);                             (* rueckt an den Anfang *)
BEGIN
        l^.pos := l^.anf;
END reset;


PROCEDURE elem (l : Liste) : Listenelement;         (* liefert aktuelles Listenelement *)
BEGIN
        IF NOT endpos(l)
                THEN RETURN l^.pos^.next^.inhalt;
                ELSE error('in elem: Kein aktuelles Listenelement')
        END;
END elem;


PROCEDURE insert (l : Liste; x:Listenelement);      (* fuegt Listenelement ein. Das neu *)
VAR hilf : Listenzeiger;                                  (* eingefuegte kommt vor das  *)
BEGIN                                                     (* aktuelle und wird aktuell  *)
        NEW(hilf);
        IF hilf=NIL THEN error("in insert: kein Platz mehr da")
                    ELSE hilf^.inhalt := x;
                         hilf^.next   := l^.pos^.next;
                         l^.pos^.next := hilf
        END
END insert;


PROCEDURE delete (l : Liste);                       (* entfernt aktuelles Listenelement *)
VAR hilf : Listenzeiger;
BEGIN
        IF NOT endpos(l)
                THEN
                        hilf := l^.pos^.next;
                        l^.pos^.next := hilf^.next;
                        DISPOSE (hilf);
                ELSE error('in delete: Kein aktuelles Listenelement')
        END;
END delete;

END liste.
```

9.2. Keller

Def.: Ein *Keller* ist eine (ggf. leere) Folge von Elementen eines bestimmten Typs *T* zusammen mit einem sogenannten (ggf. undefinierten) *Top*-Element.

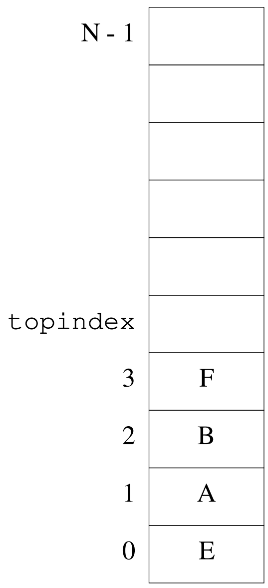
Schnittstelle des `ADT keller`:

`createstack`	:		$\rightarrow$	Keller	legt leeren Keller an
`push`	:	Keller $\times$ *T*	$\rightarrow$	Keller	legt Element auf Keller
`pop`	:	Keller	$\rightarrow$	Keller	entfernt oberstes Element
`top`	:	Keller	$\rightarrow$	Element	liefert oberstes Element
`emptystack`	:	Keller	$\rightarrow$	Boolean	liefert `TRUE`, falls Keller leer ist

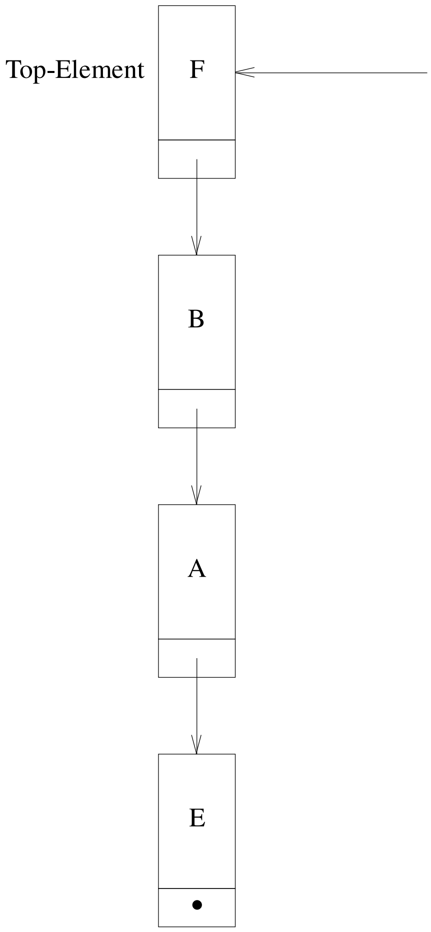
Semantik der Kelleroperationen:

```
A1)  emptystack (createstack)   =  TRUE
A2)  emptystack (push(k,x))     =  FALSE
A3)  pop (push(k,x))            =  k
A4)  top (push(k,x))            =  x
```

Implementation eines Kellers mit einem Array



Implementation eines Kellers mit Zeigern



```
(*********************************************************************************)
(*      Datum:  11.01.94  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  keller.d                                                         *)
(*              Schnittstelle zum Zugriff auf einen Keller                       *)
(*              ueber einem vorgegebenen Kellerelementtyp                        *)
(*                                                                               *)
(*********************************************************************************)

DEFINITION MODULE keller;


TYPE Kellerelement = CHAR;

TYPE Keller;                                          (* Details sind im           *)
                                                      (* Implementation—Modul      *)
                                                      (* versteckt                 *)

PROCEDURE createstack (VAR k: Keller);                (* legt leeren Keller an     *)

PROCEDURE push        (VAR k: Keller; x: Kellerelement);(* fuegt x in Keller ein   *)

PROCEDURE top         (k: Keller) : Kellerelement;    (* liefert oberst.Kellerelement *)

PROCEDURE pop         (VAR k: Keller);                (* entf. oberstes Kellerelement *)

PROCEDURE emptystack  (k: Keller) : BOOLEAN;          (* testet, ob Keller leer ist   *)

END keller.
```

```
(*****************************************************************************************)
(*      Datum:  11.01.94  14:15 Uhr                                                      *)
(*      Autor:  Oliver Vornberger                                                        *)
(*      Modul:  zeigerkeller.m2                                                          *)
(*              Implementierung eines Kellers mit Hilfe von Zeigern                      *)
(*                                                                                       *)
(*****************************************************************************************)

IMPLEMENTATION  MODULE keller;

FROM error      IMPORT error;
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;
FROM InOut      IMPORT WriteString, WriteLn;

TYPE Keller        = POINTER TO Kellereintrag;            (* Zeiger auf Kellereintraege *)

     Kellereintrag = RECORD                                            (* Kellereintrag *)
                        inhalt : Kellerelement;
                        next   : Keller;
                     END;

PROCEDURE createstack (VAR k: Keller);                (* legt leeren Keller an        *)
BEGIN   k := NIL;       END createstack;

PROCEDURE emptystack (k: Keller) : BOOLEAN;           (* testet, ob Keller leer ist   *)
BEGIN   RETURN (k=NIL); END emptystack;

PROCEDURE push (VAR k: Keller; x: Kellerelement);     (* fuegt x in Keller ein        *)
VAR hilf : Keller;
BEGIN
        NEW(hilf);
        IF hilf=NIL THEN error("in push: kein Platz mehr da") ELSE
                hilf^.inhalt := x;
                hilf^.next := k;
                k := hilf;
        END
END push;

PROCEDURE top (k : Keller) : Kellerelement;     (* liefert oberstes Kellerelement      *)
BEGIN
        IF emptystack(k) THEN error("in top: Keller leer")
                         ELSE RETURN k^.inhalt;
        END
END top;

PROCEDURE pop (VAR k: Keller);                  (* entfernt oberstes Kellerelement     *)
VAR hilf : Keller;
BEGIN
        IF emptystack(k) THEN error("in pop: Keller leer")
                         ELSE hilf := k;
                              k := k^.next;
                              DISPOSE (hilf)
        END
END pop;

BEGIN
        WriteString("Eingebunden ist die Pointerversion des Kellers"); WriteLn;
END keller.
```

```
(*******************************************************************************)
(*      Datum:  11.01.94  14:15 Uhr                                            *)
(*      Autor:  Oliver Vornberger                                              *)
(*      Modul:  arraykeller.m2                                                 *)
(*              Implementierung eines Kellers mit Hilfe eines Arrays           *)
(*              und Index auf die erste freie Array—Komponente                 *)
(*******************************************************************************)

IMPLEMENTATION  MODULE keller;

FROM error      IMPORT error;
FROM Storage    IMPORT ALLOCATE;
FROM InOut      IMPORT WriteString, WriteLn;

CONST N= 1000;                                (* maximale Anzahl von Kellerelementen *)
TYPE Keller =   POINTER TO                             (* Zeiger auf array und index *)
                RECORD
                        inhalt :   ARRAY [0..N—1] OF Kellerelement;
                        topindex:  CARDINAL;
                END;

PROCEDURE createstack (VAR k: Keller);                 (* legt leeren Keller an        *)
BEGIN
        NEW(k);
        k^.topindex := 0;
END createstack;

PROCEDURE emptystack (k: Keller) : BOOLEAN;            (* testet, ob Keller leer ist   *)
BEGIN
        RETURN (k^.topindex=0);
END emptystack;

PROCEDURE push (VAR k: Keller; x: Kellerelement);      (* fuegt x in Keller ein        *)
BEGIN
        IF k^.topindex=N THEN error("in push: Keller ist voll")
                         ELSE k^.inhalt[k^.topindex] := x;
                              INC (k^.topindex);
        END
END push;

PROCEDURE top (k : Keller) : Kellerelement;          (* liefert oberstes Kellerelement *)
BEGIN
        IF emptystack(k) THEN error("in top: Keller ist leer")
                         ELSE RETURN k^.inhalt[k^.topindex—1];
        END
END top;

PROCEDURE pop (VAR k: Keller);                      (* entfernt oberstes Kellerelement *)
BEGIN
        IF emptystack(k) THEN error("in pop: Keller ist leer")
                         ELSE DEC (k^.topindex);
        END
END pop;

BEGIN
        WriteString("Eingebunden ist die array—Version des Kellers"); WriteLn;
END keller.
```

```
(*********************************************************************************)
(*      Datum:  17.01.94  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  klammer.m2                                                       *)
(*              liest eine Folge von runden und eckigen Klammern ein             *)
(*              und ueberprueft auf korrekte Klammerung;                         *)
(*              unterhalb der ersten fehlerhaften Klammer wird das Zeichen ^ gesetzt    *)
(*********************************************************************************)

MODULE klammer;

FROM keller     IMPORT Keller, createstack, push, pop, top;
                (* keller.d muss "TYPE Kellerelement = CHAR;" enthalten *)

FROM InOut      IMPORT EOL, Read, Write, WriteString, WriteLn;

VAR k: Keller;
    c: CHAR;
    fehler : BOOLEAN;

BEGIN

        createstack(k);
        fehler := FALSE;

        WriteString("Bitte Klammerausdruck eingeben: ");
        WriteLn;

        push(k,EOL);

        REPEAT
                Read(c);
                CASE c OF

                "(":            push(k,")"); Write(' ');|
                "[":            push(k,"]"); Write(' ');|
                ")","]",EOL:    IF top(k)=c
                                        THEN Write(' '); pop(k);
                                        ELSE Write('^'); fehler := TRUE ;
                                END; |
                ELSE            Write('^'); fehler := TRUE;

                END;


        UNTIL fehler OR (c=EOL);

        IF fehler
                THEN WriteString("fehlerhaft")
                ELSE WriteString("korrekt geklammert")
        END;
        WriteLn

END klammer.
```

```
(*****************************************************************************)
(*      Datum:  17.01.94  14:15 Uhr                                          *)
(*      Autor:  Oliver Vornberger                                            *)
(*      Modul:  postfix.m2                                                   *)
(*              liest einen Ausdruck in Infix-Notation ein und gibt ihn in Postfix aus  *)
(*              Es findet keine Fehlerueberpruefung statt, d. h.             *)
(*              der eingegebene Ausdruck muss ein korrekter Infix-Ausdruck sein !      *)
(*              Der Infix-Ausdruck enthaelt die binaeren Operatoren +, -, *, /         *)
(*              die Klammern (,) und als Operanden die Variablen a,b,c, ..., x,y,z     *)
(*****************************************************************************)

MODULE postfix;

FROM InOut      IMPORT EOL, Read, Write, WriteString, WriteLn;
FROM keller     IMPORT Keller, createstack, push, pop, top, emptystack;
                (* keller.d muss "TYPE Kellerelement = CHAR;" enthalten *)
VAR k: Keller;
    c: CHAR;
BEGIN
        createstack(k);
        WriteString("Bitte Infix-Ausdruck: ");
        REPEAT
                Read(c);
                CASE c OF

                "(":            push(k,c) |            (* "(" auf den Keller legen     *)

                ")":            WHILE top(k) # "(" DO  (* Keller bis vor 1."(" ausgeben*)
                                        Write(top(k));
                                        pop(k)
                                END;
                                pop(k)  |              (* "(" vom Keller entfernen     *)

                "a" .."z":      Write(c) |             (* Operanden sofort ausgeben    *)

                "+","-":        WHILE (NOT emptystack(k)) AND (top(k) # "(")
                                DO
                                        Write(top(k));  (* Operatoren vom Keller ausgeb.*)
                                        pop(k)
                                END;
                                push(k,c) |            (* Operator auf den Keller legen*)

                "*","/":        IF (NOT emptystack(k))
                                AND ((top(k)="*") OR (top(k)="/"))
                                THEN
                                        Write(top(k));  (* "*" oder "/" vom Keller ausg.*)
                                        pop(k)
                                END;
                                push(k,c) |            (* Operator auf den Keller legen*)

                EOL:            WHILE NOT emptystack(k) DO      (* Keller ganz ausgeben *)
                                        Write(top(k));
                                        pop(k);
                                END
                END (*CASE *)
        UNTIL (c=EOL);
        WriteLn;

END postfix.
```

```
(**********************************************************************************)
(*      Datum:  19.12.95  14:15 Uhr                                               *)
(*      Autor:  Oliver Vornberger                                                 *)
(*      Modul:  element.d                                                         *)
(*              fuehrt den Typ Element in opaker Form ein                         *)
(*              und die darauf definierten Prozeduren                             *)
(**********************************************************************************)

DEFINITION MODULE element;

TYPE Element;                                (* Details sind im Implementation—     *)
                                             (* Modul versteckt                     *)


PROCEDURE gleich(x,y:Element):BOOLEAN;       (* liefert TRUE, falls x = y           *)

PROCEDURE kleiner(x,y:Element): BOOLEAN;     (* liefert TRUE, falls x < y           *)

PROCEDURE createelement(VAR x:Element);      (* legt ein (undefiniertes) Element an *)

PROCEDURE removeelement(x: Element);         (* entfernt ein Element                *)

PROCEDURE readelement (VAR x:Element):BOOLEAN; (* versucht ein Element zu lesen     *)
                                             (* Liefert True, wenn erfolgreich      *)

PROCEDURE copyelement(x,y: Element);         (* kopiert y nach x                    *)

PROCEDURE writeelement (x:Element);          (* gibt ein Element aus                *)

PROCEDURE istoperator(x:Element) : BOOLEAN;  (* liefert TRUE,falls x Operator ist   *)
                                             (* fuer Postfixausdruck erforderlich   *)

PROCEDURE ord(x:Element): CARDINAL;          (* liefert die Ordnungszahl            *)
                                             (* fuer Hashorganisation erforderlich  *)
END element.
```

```
(******************************************************************************************)
(*      Datum:  18.12.95  14:15 Uhr                                                        *)
(*      Autor:  Oliver Vornberger                                                          *)
(*      Modul:  charelement.m2                                                             *)
(*              implementiert den Typ Element als CHAR                                     *)
(*              und die darauf definierten Prozeduren                                      *)
(******************************************************************************************)

IMPLEMENTATION  MODULE element;

FROM InOut      IMPORT EOL, Read, Write, WriteString, WriteLn;
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;

TYPE Element =  POINTER TO CHAR;

PROCEDURE gleich(x,y:Element):BOOLEAN;          (* liefert TRUE, falls x und y          *)
BEGIN RETURN x^ = y^ END gleich;                (* gleichen Inhalt haben                *)

PROCEDURE kleiner(x,y:Element): BOOLEAN;        (* liefert TRUE, falls Inhalt von x     *)
BEGIN RETURN x^ < y^ END kleiner;               (* kleiner Inhalt von y                 *)

PROCEDURE createelement(VAR x:Element);         (* legt ein (undefiniertes) Element an  *)
BEGIN NEW(x) END createelement;

PROCEDURE removeelement(x:Element);             (* loescht das Element                  *)
BEGIN DISPOSE(x) END removeelement;

PROCEDURE copyelement(x,y: Element);            (* kopiert Inhalt von y nach x          *)
BEGIN x^ := y^ END copyelement;

PROCEDURE readelement (VAR x:Element):BOOLEAN;  (* versucht ein Element zu lesen        *)
VAR c : CHAR;                                   (* falls erfolgreich, wird der gelesene *)
BEGIN                                           (* Wert in das vorhandene Element       *)
        Read(c);                                (* kopiert und TRUE zurueck geliefert   *)
        IF c  = EOL THEN RETURN FALSE
                    ELSE x^ := c; RETURN TRUE
        END;
END readelement;

PROCEDURE writeelement (x:Element);                     (* gibt ein Element aus         *)
BEGIN
        Write(x^);
END writeelement;

PROCEDURE istoperator(x:Element) : BOOLEAN;             (* liefert TRUE,falls x Operator*)
BEGIN                                                   (* fuer Postfixausdruck erford. *)
        RETURN (x^="+") OR (x^="-") OR (x^="*") OR (x^="/")
END istoperator;

PROCEDURE ord(x:Element): CARDINAL;                     (* liefert die Ordnungszahl     *)
BEGIN                                                   (* fuer Hashorganisation erford.*)
        RETURN ORD(x^)
END ord;

BEGIN
        WriteString("Eingebunden ist die CHAR-Version des Typs Element"); WriteLn;
END element.
```

```
(*************************************************************************************)
(*      Datum:  19.12.95  14:15 Uhr                                                 *)
(*      Autor:  Oliver Vornberger                                                   *)
(*      Modul:  reverse.m2                                                          *)
(*              liest eine Folge von Elementen                                      *)
(*              und gibt sie in umgekehrter Reihenfolge wieder aus                  *)
(*                                                                                  *)
(*              fuer das MODULE reverse wird ein Keller fuer Elemente benoetigt     *)
(*              das File keller.d muss daher beginnen:                              *)
(*                                                                                  *)
(*              DEFINITION MODULE keller;                                           *)
(*              FROM element IMPORT Element;                                        *)
(*              TYPE Kellerelement = Element;                                       *)
(*              ...                                                                 *)
(*              END keller.                                                         *)
(*                                                                                  *)
(*************************************************************************************)

MODULE reverse;

FROM element    IMPORT Element, createelement, readelement, writeelement, removeelement;
FROM keller     IMPORT Keller, createstack, push, pop, top, emptystack;
FROM InOut      IMPORT WriteString, WriteLn;

VAR k: Keller;
    x: Element;

BEGIN
        createstack(k);

        WriteString("Bitte Elemente eingeben: ");

        createelement(x);
        WHILE readelement(x) DO
                push(k,x);
                createelement(x);
        END;
        removeelement(x);               (* das unnoetig erzeugte Element loeschen       *)

        WriteString("Umgekehrte Reihenfolge:  " );

        WHILE NOT emptystack(k) DO
                writeelement  (top(k));
                removeelement(top(k));
                pop(k);
        END;
        WriteLn;
END reverse.
```

```
# makefile zum Uebersetzen des Programms 'reverse'
# Aufruf mit 'make —f reverse.mak' oder
# Aufruf mit 'make' falls Inhalt in Datei 'makefile' kopiert wurde

# Genereller Aufbau im makefile:
#
# F0:           F1, F2, F3, ...
# <Tabulator>   Aktion zum Erzeugen von F0,
#               wird aufgerufen, falls eine von F1, F2, F3, ... juenger ist als F0
#               oder eine ihrer Voraussetzungen juenger ist als F0
#               oder falls F0 nicht existiert


reverse:        charelement.o error.o zeigerkeller.o reverse.o
                m2c charelement.o error.o zeigerkeller.o reverse.o
                mv a.out reverse

element.sy:     element.d
                m2c element.d

error.sy:       error.d
                m2c error.d

keller.sy:      element.sy keller.d
                m2c keller.d

charelement.o:  element.sy charelement.m2
                m2c —c charelement.m2

error.o:        error.sy error.m2
                m2c —c error.m2

zeigerkeller.o: error.sy keller.sy zeigerkeller.m2
                m2c —c zeigerkeller.m2

reverse.o:      element.sy keller.sy reverse.m2
                m2c —c reverse.m2

# 'make clean' loescht alle ueberfluessigen Dateien
clean:
                rm —f element.sy charelement.r charelement.o \
                error.sy error.r error.o keller.sy \
                zeigerkeller.r zeigerkeller.o reverse.o reverse.r reverse

# Bemerkung:
# Der Quelltext eines Definition—Moduls mit Namen <name>
# muss gespeichert werden in einem File mit Namen <name>.d
#
# Der Quelltext eines Implementation—Moduls
# kann in einem File mit beliebigem Namen gespeichert werden.
```

9.3. Schlange

Def.: Eine *Schlange* ist eine (ggf. leere) Folge von Elementen eines bestimmten Typs T zusammen mit einem sogenannten (ggf. undefinierten) *Front*-Element.

Schnittstelle des `ADT schlange`:

`createqueue`	:		$\rightarrow$	Schlange	legt leere Schlange an
`enq`	:	Schlange $\times T$	$\rightarrow$	Schlange	fügt Element hinten ein
`deq`	:	Schlange	$\rightarrow$	Schlange	entfernt vorderstes Element
`front`	:	Schlange	$\rightarrow$	Element	liefert vorderstes Element
`emptyqueue`	:	Schlange	$\rightarrow$	Boolean	liefert `TRUE`, falls Schlange leer ist

Implementation einer Schlange mit einem Array

					E	B	A	F	B	R							
0					front					last							N-1

```
(******************************************************************************************)
(*      Datum:  18.01.94  14:15 Uhr                                                      *)
(*      Autor:  Oliver Vornberger                                                        *)
(*      Modul:  schlange.d                                                               *)
(*              definiert den Typ Schlange fuer Schlangenelemente eines beliebigen Typs *)
(*                                                                                       *)
(******************************************************************************************)

DEFINITION MODULE schlange;

FROM element IMPORT Element;

TYPE Schlangenelement = Element;

TYPE Schlange;                                          (* Details sind im              *)
                                                        (* Implementation—Modul         *)
                                                        (* versteckt                    *)

PROCEDURE createqueue(VAR s: Schlange);                 (* legt leere Schlange an       *)

PROCEDURE enq        (s: Schlange; x:Schlangenelement);      (* fuegt x hinten in s ein *)

PROCEDURE deq        (s: Schlange);             (* entfernt vorderstes Schlangenelement *)

PROCEDURE front      (s: Schlange) : Schlangenelement; (* liefert vorderst. Schl.element*)

PROCEDURE emptyqueue (s: Schlange) : BOOLEAN;           (* testet, ob Schlange leer ist *)

END schlange.
```

```
(*******************************************************************************)
(*      Datum:  18.01.94  14:15 Uhr                                            *)
(*      Autor:  Oliver Vornberger                                              *)
(*      Modul:  arrayschlange.m2                                               *)
(*              implementiert eine Schlange mit Hilfe eines Arrays             *)
(*               ________________________________________                      *)
(*              |       xxxxxxxxxxxx                     |                     *)
(*               ________________________________________                      *)
(*               ^    ^          ^                      ^                      *)
(*               0   front      last                   N-1                     *)
(*                                                                             *)
(*******************************************************************************)

IMPLEMENTATION  MODULE schlange;

FROM error      IMPORT error;
FROM Storage    IMPORT ALLOCATE;
FROM InOut      IMPORT WriteString, WriteLn;

CONST N = 100;                            (* max. Anzahl von Schlangenelementen   *)

TYPE Schlange = POINTER TO                (* Zeiger auf array und index           *)
        RECORD
                inhalt :   ARRAY [0..N-1] OF Schlangenelement;
                frontindex:  CARDINAL;    (* zeigt auf vorderstes Schlangenelement*)
                lastindex:   CARDINAL;    (* zeigt auf letztes Schlangenelement   *)
        END;


PROCEDURE createqueue (VAR s: Schlange);              (* legt leere Schlange an      *)
BEGIN
        NEW(s);
        s^.frontindex := 0;
        s^.lastindex  := N-1;
END createqueue;


PROCEDURE emptyqueue (s: Schlange) : BOOLEAN;         (* testet, ob Schlange leer ist *)
BEGIN
        RETURN (s^.lastindex+1) MOD N = s^.frontindex;
END emptyqueue;


PROCEDURE fullqueue (s: Schlange) : BOOLEAN;          (* testet, ob Schlange voll ist *)
BEGIN
        RETURN (s^.lastindex+2) MOD N = s^.frontindex;
END fullqueue;
```

```
PROCEDURE enq (s: Schlange; x: Schlangenelement);       (* fuegt x hinten in s ein      *)
BEGIN
        IF fullqueue(s)
        THEN    error("in enq: Schlange ist voll")
        ELSE
                s^.lastindex := (s^.lastindex + 1) MOD N;
                s^.inhalt[s^.lastindex] := x;
        END
END enq;


PROCEDURE deq (s: Schlange);                    (* entfernt vorderstes Schlangenelement *)
BEGIN
        IF emptyqueue(s)
                THEN error("in deq: Schlange ist leer")
                ELSE s^.frontindex := (s^.frontindex + 1) MOD N;
        END;
END deq;


PROCEDURE front (s: Schlange) : Schlangenelement;  (* liefert oberstes Schlangenelement *)
BEGIN
        IF emptyqueue(s)
                THEN error("in front: Schlange ist leer")
                ELSE RETURN s^.inhalt[s^.frontindex];
        END
END front;



BEGIN
        WriteString("Eingebunden ist die array—Version der Schlange "); WriteLn;
END schlange.
```

```
(****************************************************************************************)
(*      Datum:  08.01.96  14:15 Uhr                                                    *)
(*      Autor:  Oliver Vornberger                                                      *)
(*      Modul:  schlangetest.m2                                                        *)
(*              reiht eine Folge von Elementen in eine Schlange ein                    *)
(*              und gibt bei Eingabe von Return jeweils das vorderste aus              *)
(*                                                                                     *)
(****************************************************************************************)

MODULE schlangetest;

FROM element    IMPORT Element, createelement, readelement, writeelement, removeelement;
FROM schlange   IMPORT Schlange, createqueue, enq, deq, front, emptyqueue;
FROM InOut      IMPORT WriteString, WriteLn;

VAR     s:      Schlange;
        x:      Element;

BEGIN

        createqueue(s);

        WriteString("Bitte Elemente einreihen bzw. mit RETURN anfordern: ");

        createelement(x);

        REPEAT
                IF readelement(x)
                THEN
                        enq(s,x);
                        createelement(x);
                ELSE
                        IF NOT emptyqueue(s)
                        THEN
                                writeelement(front(s));
                                WriteLn;
                                removeelement(front(s));
                                deq(s)
                        END
                END
        UNTIL emptyqueue(s);

        removeelement(x);

END schlangetest.
```

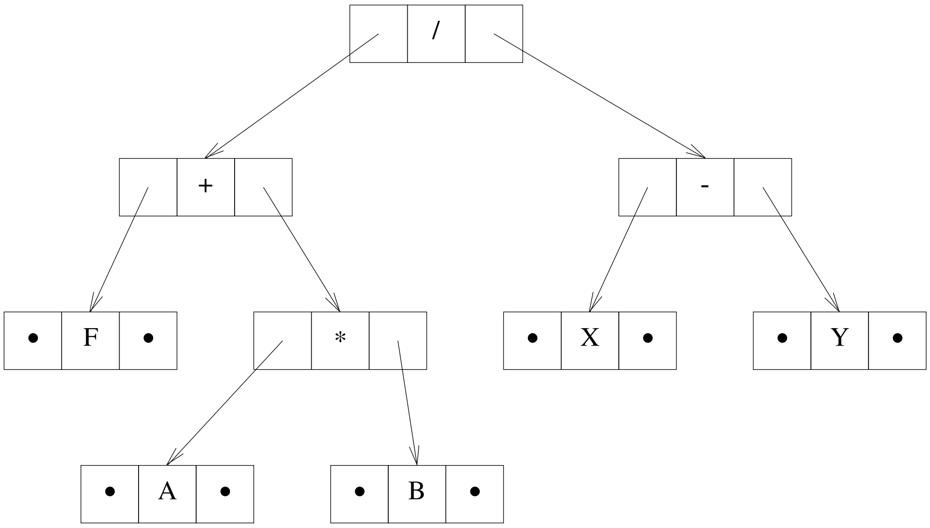
9.4. Baum

Def.: Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Die Elemente in einem Knoten sind vom Typ *T*.

Schnittstelle des `ADT baum`:

`createtree`	:		$\rightarrow$	Baum	legt leeren Baum an
`emptytree`	:	Baum	$\rightarrow$	Boolean	liefert `TRUE`, falls Baum leer ist
`left`	:	Baum	$\rightarrow$	Baum	liefert linken Teilbaum
`right`	:	Baum	$\rightarrow$	Baum	liefert rechten Teilbaum
`value`	:	Baum	$\rightarrow$	T	liefert Wurzelelement
`createnode`	:		$\rightarrow$	Baum	legt Baum mit Knoten an
`setvalue`	:	Baum $\times$ T	$\rightarrow$	Baum	legt Element in Wurzel ab
`setleft`	:	Baum $\times$ Baum	$\rightarrow$	Baum	hängt Baum als linken Sohn ein
`setright`	:	Baum $\times$ Baum	$\rightarrow$	Baum	hängt Baum als rechten Sohn ein

Implementation eines Baumes mit Zeigern



Traversierungen dieses Baumes

Präfix:	`/ + F * A B − X Y`
Infix:	`F + A * B / X − Y`
Postfix:	`F A B * + X Y − /`
Infix mit Klammern:	`(F + A * B) / (X − Y)`

```
(***********************************************************************************)
(*                                                                                 *)
(*      Datum:  24.01.94  14:15 Uhr                                                *)
(*      Autor:  Oliver Vornberger                                                  *)
(*      Modul:  baum.d                                                             *)
(*              fuehrt den Typ Baum ein und die darauf definierten Operationen     *)
(*                                                                                 *)
(***********************************************************************************)

DEFINITION MODULE baum;

FROM element IMPORT Element;                          (* Element-Typ                *)

TYPE Baumelement = Element;

TYPE Baum;                                            (* Details sind im            *)
                                                      (* Implementation-Modul       *)
                                                      (* versteckt                  *)


PROCEDURE createtree (VAR b: Baum);                   (* legt leeren Baum an        *)

PROCEDURE emptytree  (b: Baum) : BOOLEAN;             (* liefert TRUE,falls Baum leer *)

PROCEDURE left       (b: Baum) : Baum;                (* liefert linken Sohn von b  *)

PROCEDURE right      (b: Baum) : Baum;                (* liefert rechten Sohn von b *)

PROCEDURE value      (b: Baum) : Baumelement;         (* liefert Inhalt von Wurzel  *)

PROCEDURE createnode (VAR b: Baum);                   (* legt Knoten an             *)

PROCEDURE setvalue   (b: Baum; x: Baumelement);       (* setzt x in die Wurzel von b *)

PROCEDURE setleft    (b: Baum; sohn: Baum);           (* haengt sohn links ein      *)

PROCEDURE setright   (b: Baum; sohn: Baum);           (* haengt sohn rechts ein     *)

END baum.
```

```
(****************************************************************************************)
(*      Datum:  09.01.95  14:15 Uhr                                                    *)
(*      Autor:  Oliver Vornberger                                                      *)
(*      Modul:  baum.m2                                                                *)
(*              Implementierung eines binaeren Baumes mit Zeigern                      *)
(****************************************************************************************)

IMPLEMENTATION  MODULE baum;

FROM error      IMPORT error;                     (* zur Fehlerbehandlung        *)
FROM Storage    IMPORT ALLOCATE;                  (* fuer NEW                    *)

TYPE Baum   =   POINTER TO Knoten;                (* Zeiger auf Knoten           *)

     Knoten =   RECORD
                        inhalt: Baumelement;      (* Inhalt                      *)
                        links:  Baum;             (* linker Teilbaum             *)
                        rechts: Baum;             (* rechter Teilbaum            *)
                END;

PROCEDURE createtree    (VAR b:Baum);             (* legt leeren Baum an         *)
BEGIN
        b:=NIL;
END createtree;


PROCEDURE emptytree     (b:Baum) : BOOLEAN;       (* liefert TRUE,falls Baum leer *)
BEGIN
        RETURN b = NIL
END emptytree;


PROCEDURE left          (b:Baum) : Baum;          (* liefert linken  Sohn von b   *)
BEGIN
        IF emptytree(b)
                THEN error("in left: leerer Baum")
                ELSE RETURN b^.links
        END
END left;


PROCEDURE right         (b:Baum) : Baum;          (* liefert rechten Sohn von b   *)
BEGIN
        IF emptytree(b)
                THEN error("in right: leerer Baum")
                ELSE RETURN b^.rechts
        END
END right;


PROCEDURE value         (b:Baum) : Baumelement;   (* liefert Inhalt von Knoten    *)
BEGIN
        IF emptytree(b)
                THEN error("in value: leerer Baum")
                ELSE RETURN b^.inhalt
        END
END value;
```

```
PROCEDURE createnode    (VAR b:Baum);                    (* legt neuen Knoten an          *)
BEGIN
        NEW(b);
        IF emptytree(b)
                THEN error("in createnode: kein Platz mehr da")
                ELSE b^.links := NIL;
                     b^.rechts:= NIL;
        END
END createnode;


PROCEDURE setvalue(b:Baum; x:Baumelement);               (* setzt x in die Wurzel von b  *)
BEGIN
        IF emptytree(b)
                THEN error("in setvalue: leerer Baum")
                ELSE b^.inhalt :=x;
        END
END setvalue;


PROCEDURE setleft (b:Baum; sohn:Baum);                   (* haengt sohn links ein         *)
BEGIN
        IF emptytree(b)
                THEN error("in setleft: leerer Baum")
                ELSE b^.links :=sohn
        END
END setleft;


PROCEDURE setright(b:Baum; sohn:Baum);                   (* haengt sohn rechts ein        *)
BEGIN
        IF emptytree(b)
                THEN error("in setright: leerer Baum")
                ELSE b^.rechts:=sohn
        END
END setright;


END baum.
```

Traversierungen

Eine Traversierung eines binären Baumes besteht aus dem systematischen Besuchen aller Knoten in einer bestimmten Reihenfolge.

```
(*****************************************************************************************)
(*      Datum:  24.01.94  14:15 Uhr                                                      *)
(*      Autor:  Oliver Vornberger                                                        *)
(*      Modul:  traversal.d                                                              *)
(*              inorder, preorder, postorder und klammerinorder Baumdurchlaeufe          *)
(*****************************************************************************************)

DEFINITION MODULE traversal;


FROM baum IMPORT Baum;

PROCEDURE inorder        (b: Baum);      (* gibt Elemente des Baumes b                  *)
                                         (* in inorder-Reihenfolge aus:                 *)
                                         (* linker Sohn - Knoten - rechter Sohn         *)

PROCEDURE preorder       (b: Baum);      (* gibt Elemente des Baumes b                  *)
                                         (* in preorder-Reihenfolge aus:                *)
                                         (* Knoten - linker Sohn - rechter Sohn         *)

PROCEDURE postorder      (b: Baum);      (* gibt Elemente des Baumes b                  *)
                                         (* in postorder-Reihenfolge aus:               *)
                                         (* linker Sohn - rechter Sohn - Knoten         *)

PROCEDURE klammerinorder (b: Baum);      (* gibt Elemente des Baumes b                  *)
                                         (* geklammert in inorder aus:                  *)
                                         (* "(" linker Sohn - Knoten - rechter Sohn ")" *)


END traversal.
```

```
(****************************************************************************************)
(*      Datum:  24.01.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  traversal.m2                                                            *)
(*              inorder, preorder und postorder Baumdurchlaeufe                         *)
(****************************************************************************************)

IMPLEMENTATION MODULE traversal;

FROM element    IMPORT writeelement;
FROM baum       IMPORT Baum, emptytree, value, left, right;
FROM InOut      IMPORT Write;

PROCEDURE inorder(b:Baum);                      (* druckt die Inhalte in inorder        *)
BEGIN
        IF NOT emptytree(b) THEN
                inorder(left(b));               (* linker Sohn                          *)
                writeelement(value(b));         (* Knoten b                             *)
                inorder(right(b));              (* rechter Sohn                         *)
        END;
END inorder;


PROCEDURE preorder(b:Baum);                     (* druckt die Inhalte in preorder       *)
BEGIN
        IF NOT emptytree(b) THEN
                writeelement(value(b));         (* Knoten b                             *)
                preorder(left(b));              (* linker Sohn                          *)
                preorder(right(b));             (* rechter Sohn                         *)
        END;
END preorder;


PROCEDURE postorder(b:Baum);                    (* druckt die Inhalte in Postorder      *)
BEGIN
        IF NOT emptytree(b) THEN
                postorder(left(b));             (* linker Sohn                          *)
                postorder(right(b));            (* rechter Sohn                         *)
                writeelement(value(b));         (* Knoten b                             *)
        END;
END postorder;


PROCEDURE klammerinorder(b:Baum);               (* druckt die Inhalte des Baumes mit    *)
                                                (* Klammern in Inorder-Reihenfolge aus  *)
BEGIN
        IF NOT emptytree(b) THEN
                IF NOT emptytree(left(b)) THEN Write("("); END;         (* "("          *)
                klammerinorder(left(b));                                (* linker Sohn  *)
                writeelement(value(b));                                 (* Knoten b     *)
                klammerinorder(right(b));                               (* rechter Sohn *)
                IF NOT emptytree(right(b)) THEN Write(")"); END;        (* ")"          *)
        END;
END klammerinorder;

END traversal.
```

```
(****************************************************************************************)
(*      Datum:  12.01.96  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  tiefsuche.d                                                             *)
(*              Prozedur fuer die Tiefensuche in einem Baum                             *)
(****************************************************************************************)

DEFINITION MODULE tiefsuche;

FROM baum IMPORT Baum;

PROCEDURE tiefensuche  (b : Baum);              (* druckt die Inhalte eines Baums in    *)
                                                (* Preorder-Reihenfolge aus             *)

END tiefsuche.
```

```
(*********************************************************************************)
(*      Datum:  12.01.96  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  tiefsuche.m2                                                     *)
(*              iterative Tiefensuche mit Hilfe eines Kellers                    *)
(*                                                                               *)
(*              fuer die Prozedur tiefensuche wird ein Keller fuer Baeume benoetigt     *)
(*              das File keller.d muss daher beginnen:                           *)
(*                                                                               *)
(*              DEFINITION MODULE keller;                                        *)
(*              FROM baum IMPORT Baum;                                           *)
(*              TYPE Kellerelement = Baum;                                       *)
(*              ...                                                              *)
(*              END keller.                                                      *)
(*                                                                               *)
(*********************************************************************************)

IMPLEMENTATION MODULE tiefsuche;

FROM element    IMPORT writeelement;
FROM baum       IMPORT Baum, emptytree, value, left, right;
FROM keller     IMPORT Keller, createstack, push, pop, top, emptystack;


PROCEDURE tiefensuche (b : Baum);              (* durchlaueft den Baum in Preorder-    *)
                                               (* Tiefensuche  mit Hilfe eines Kellers *)
VAR k : Keller;
BEGIN
        createstack(k);

        WHILE NOT emptytree(b) DO
                REPEAT
                        writeelement(value(b));
                        IF NOT emptytree(right(b))
                                THEN push(k,right(b))
                        END;
                        b := left(b)
                UNTIL emptytree(b);

                IF NOT emptystack(k) THEN
                        b := top(k);
                        pop(k)
                END
        END;

END tiefensuche;

END tiefsuche.
```

```
(****************************************************************************************)
(*      Datum:  24.01.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  breitsuche.d                                                            *)
(*              Prozedur fuer die Breitensuche in einem Baum                            *)
(****************************************************************************************)

DEFINITION MODULE breitsuche;

FROM baum IMPORT Baum;

PROCEDURE breitensuche (b : Baum);              (* druckt die Inhalte eines Baums       *)
                                                (* ebenenweise aus                      *)

END breitsuche.
```

```
(******************************************************************************)
(*      Datum:  24.01.94  14:15 Uhr                                          *)
(*      Autor:  Oliver Vornberger                                            *)
(*      Modul:  breitsuche.m2                                                *)
(*              iterative Breitensuche mit Hilfe einer Schlange              *)
(*              Der Baum wird ebenenweise ausgegeben mit einem Zeilenumbruch *)
(*              nach jeder Ebene.                                            *)
(*                                                                           *)
(*              fuer die Prozedur breitensuche wird eine Schlange fuer Baeume benoetigt *)
(*              das File schlange.d muss daher beginnen:                     *)
(*                                                                           *)
(*              DEFINITION MODULE schlange;                                  *)
(*              FROM baum IMPORT Baum;                                       *)
(*              TYPE Schlangenelement = Baum;                                *)
(*              ...                                                          *)
(*              END schlange.                                                *)
(******************************************************************************)

IMPLEMENTATION MODULE breitsuche;

FROM element    IMPORT writeelement;
FROM baum       IMPORT Baum, createtree, emptytree, value, left, right;
FROM schlange   IMPORT Schlange, createqueue, emptyqueue, enq, deq, front;
FROM InOut      IMPORT WriteLn;

PROCEDURE breitensuche (b : Baum);      (* druckt die Inhalte eines Baums          *)
                                        (* ebenenweise aus (mit Hilfe einer Schlange) *)
VAR leer : Baum;
VAR s    : Schlange;
BEGIN
        createtree(leer);               (* fungiert als Trennsymbol am Ende einer Ebene *)
        createqueue(s);                 (* enthaelt Baeume in Form von Knotenzeigern  *)

        IF NOT emptytree(b)
                THEN enq(s, b);
                     enq(s, leer)
        END;

        WHILE NOT emptyqueue(s) DO

                b := front(s); deq(s);

                IF NOT emptytree(b)

                THEN                            (* nichtleere Soehne hinten einhaengen *)
                        writeelement(value(b));
                        IF NOT emptytree(left (b)) THEN enq(s, left (b)) END;
                        IF NOT emptytree(right(b)) THEN enq(s, right(b)) END;

                ELSE                       (* leerer Baum bedeutet: Ebene abgearbeitet *)
                        WriteLn;
                        IF NOT emptyqueue(s) THEN enq(s, leer); END
                END;

        END;

END breitensuche;

END breitsuche.
```

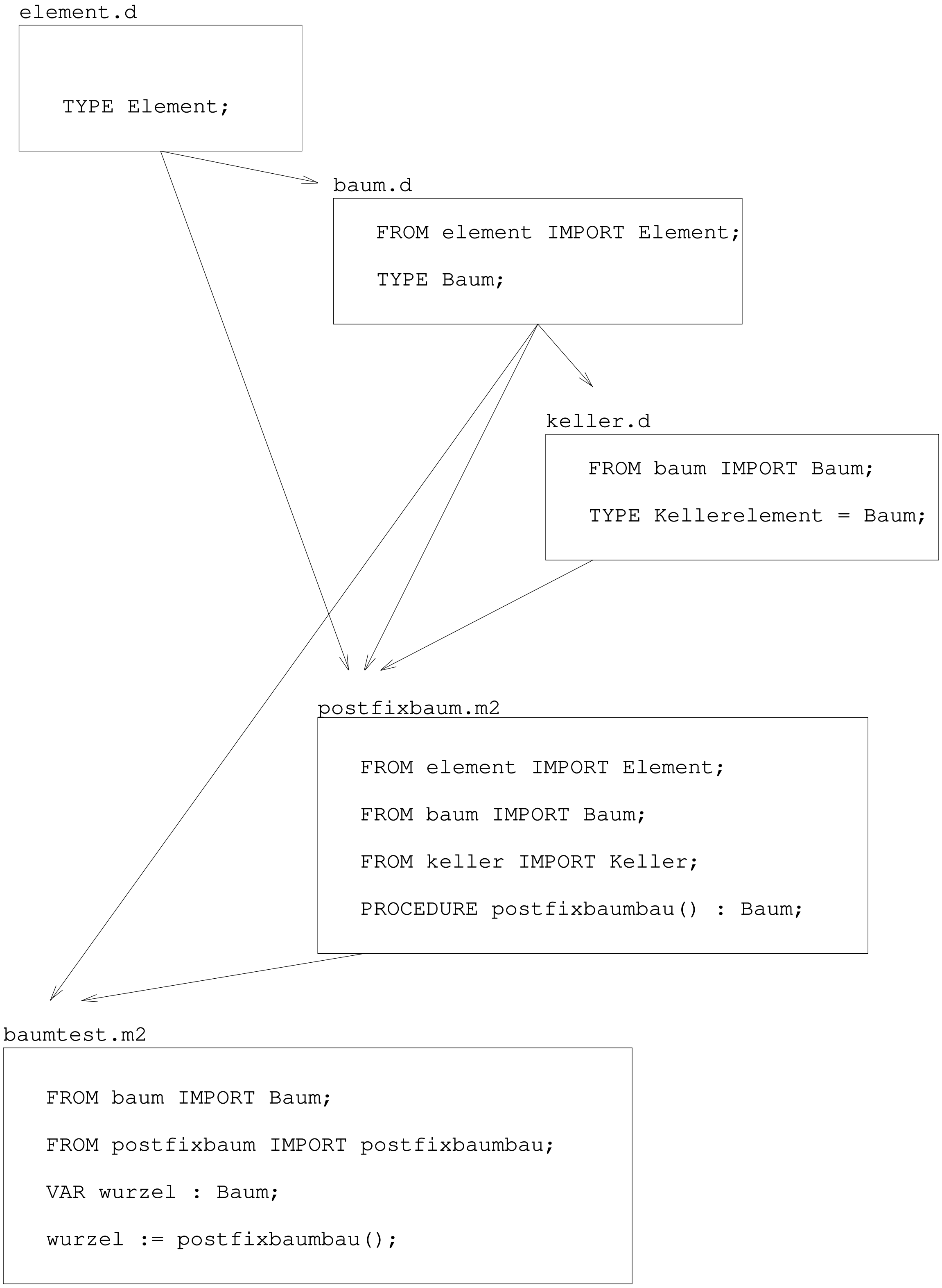
```
(****************************************************************************************)
(*      Datum:  25.01.94  14:15 Uhr                                                    *)
(*      Autor:  Oliver Vornberger                                                      *)
(*      Modul:  praefixbaum.d                                                          *)
(*              konstruiert aus einem Praefixstring einen binaeren Baum                *)
(*                                                                                     *)
(****************************************************************************************)

DEFINITION  MODULE praefixbaum;

FROM baum       IMPORT Baum;

PROCEDURE praefixbaumbau () : Baum;          (* liest einen Praefixausdruck und      *)
                                             (* konstruiert den zugehoerigen Baum    *)

END praefixbaum.
```

```
(*********************************************************************************)
(*      Datum:  15.01.96  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  praefixbaum.m2                                                   *)
(*              konstruiert aus einem Praefixstring einen binaeren Baum          *)
(*                                                                               *)
(*              die Eingabestrings werden als korrekt vorausgesetzt              *)
(*                                                                               *)
(*********************************************************************************)

IMPLEMENTATION MODULE praefixbaum;

FROM element    IMPORT Element, createelement, istoperator, readelement, removeelement;
FROM baum       IMPORT Baum, createtree, createnode, setvalue, setleft, setright;


VAR leer : Baum;


PROCEDURE praefixbaumbau() : Baum;      (* liest einen Praefixausdruck und konstruiert  *)
                                        (* den zugehoerigen Baum (rekursiv)             *)
VAR z : Baum;
VAR x : Element;
BEGIN

        createelement(x);
        IF readelement(x)
        THEN
                createnode(z);
                setvalue(z,x);
                IF istoperator(x)
                THEN
                        setleft (z,praefixbaumbau());
                        setright(z,praefixbaumbau());
                END;
                RETURN z;
        ELSE
                removeelement(x);
                RETURN leer
        END

END praefixbaumbau;


BEGIN

        createtree(leer);               (* als Rueckgabewert bei leerem Eingabestring   *)

END praefixbaum.
```

```
(*****************************************************************************************)
(*      Datum:  25.01.94  14:15 Uhr                                                       *)
(*      Autor:  Oliver Vornberger                                                         *)
(*      Modul:  postfixbaum.d                                                             *)
(*              konstruiert aus Postfixstring einen binaeren Baum                        *)
(*                                                                                        *)
(*****************************************************************************************)

DEFINITION  MODULE postfixbaum;

FROM baum       IMPORT Baum;


PROCEDURE postfixbaumbau () : Baum;              (* liest einen Postfixausdruck und      *)
                                                 (* konstruiert den zugehoerigen Baum    *)

END postfixbaum.
```

```
(****************************************************************************************)
(*      Datum:  15.01.96  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  postfixbaum.m2                                                          *)
(*              konstruiert aus einem Postfixstring einen binaeren Baum                 *)
(*              die Eingabestrings werden als korrekt vorausgesetzt                     *)
(*                                                                                      *)
(*              fuer die Prozedur postfixbaumbau wird ein Keller fuer Baeume benoetigt  *)
(*              das File keller.d ist daher zu aendern:                                 *)
(*                                                                                      *)
(*              DEFINITION MODULE keller;                                               *)
(*              FROM baum IMPORT Baum;                                                  *)
(*              TYPE Kellerelement = Baum;                                              *)
(*              ...                                                                     *)
(****************************************************************************************)

IMPLEMENTATION MODULE postfixbaum;

FROM element    IMPORT Element, createelement, istoperator, readelement, removeelement;
FROM baum       IMPORT Baum, createtree, createnode, setvalue, setleft, setright;
FROM keller     IMPORT Keller, createstack, push, pop, top, emptystack;


VAR leer : Baum;


PROCEDURE postfixbaumbau() : Baum;      (* liest einen Postfixausdruck und konstruiert   *)
                                        (* den zugehoerigen Baum (mit Keller fuer Baeume)*)
VAR z : Baum;
VAR k : Keller;
VAR x : Element;
BEGIN
        createstack(k);

        createelement(x);
        WHILE readelement(x) DO
                createnode(z);
                setvalue(z,x);
                IF istoperator(x)
                THEN
                        setright(z,top(k)); pop(k);
                        setleft (z,top(k)); pop(k);
                END;
                push(k,z);
                createelement(x);
        END;
        removeelement(x);

        IF emptystack(k) THEN RETURN leer
                         ELSE RETURN top(k);
        END;

END postfixbaumbau;


BEGIN
        createtree(leer);               (* als Rueckgabewert bei leerem Eingabestring   *)
END postfixbaum.
```

Abhängigkeiten für die Implementation der Prozedur `postfixbaumbau`



9.5. Suchbaum

```
(****************************************************************************************)
(*      Datum:  25.01.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  suchbaum.d                                                              *)
(*              definiert die fuer Suchbaeume geeigneten Operationen                    *)
(*                                                                                      *)
(*              Def.:   Ein binaerer Suchbaum ist ein binaerer Baum, bei dem            *)
(*                      alle Eintraege im linken Teilbaum eines Knotens x               *)
(*                      kleiner sind  als der Eintrag im Knoten x und bei dem           *)
(*                      alle Eintraege im rechten Teilbaum eines Knotens x              *)
(*                      groesser sind als der Eintrag im Knoten x                       *)
(*                                                                                      *)
(****************************************************************************************)

DEFINITION MODULE suchbaum;

FROM element IMPORT Element;                    (* Element-Typ                           *)

TYPE Suchbaum;                                  (* Suchbaum                              *)

PROCEDURE lookup (b:Suchbaum; x:Element;        (* sucht im Suchbaum b nach Element x;  *)
                  VAR treffer:Suchbaum          (* liefert in treffer Verweis auf Knoten*)
                 ): BOOLEAN;                    (* und als Ergebnis TRUE, falls gefunden*)


PROCEDURE insert (VAR b:Suchbaum; x:Element     (* fuegt in Suchbaum b Element x ein    *)
                 ):BOOLEAN;                     (* liefert TRUE, wenn erfolgreich, d.h. *)
                                                (* x war nicht schon im Suchbaum        *)


PROCEDURE delete (VAR b:Suchbaum; x:Element     (* versucht aus Suchbaum b Element x zu *)
                 ):BOOLEAN;                     (* loeschen; liefert TRUE, falls        *)
                                                (* erfolgreich, d.h. x wurde entfernt   *)

END suchbaum.
```

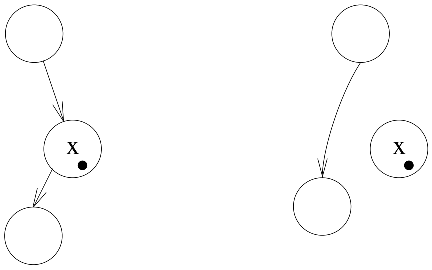
```
(****************************************************************************************)
(*      Datum:  25.01.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  suchbaum.m2                                                             *)
(*              Implementierung eines binaeren Suchbaumes mit Zeigern                   *)
(*                                                                                      *)
(****************************************************************************************)

IMPLEMENTATION  MODULE suchbaum;

FROM element    IMPORT Element, gleich, kleiner;
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;

TYPE Suchbaum = POINTER TO Knoten;                               (* Zeiger auf Knoten    *)

     Knoten   = RECORD
                        inhalt: Element;                         (* Inhalt               *)
                        links : Suchbaum;                        (* linker Teilbaum      *)
                        rechts: Suchbaum;                        (* rechter Teilbaum     *)
                END;


(****************************************************************************************)
(*      Der Aufwand aller Suchbaumoperationen ist proportional zur Anzahl der Knoten    *)
(*      auf dem Wege von der Wurzel bis zu einem Blatt.                                 *)
(*                                                                                      *)
(*      Best case:      Hat jeder Knoten 2 Soehne, so hat der Baum bei Hoehe h          *)
(*                         h                                                            *)
(*                      n=2 -1 Knoten. Die Anzahl der Weg-Knoten ist h = log(n).        *)
(*                                                                                      *)
(*      Worst case:     Werden die Elemente sortiert eingegeben, so entartet der Baum   *)
(*                      zur Liste, der Aufwand betraegt dann O(n).                      *)
(*                                                                                      *)
(*      Average case:   Fuer n zufaellige Schluessel betraegt der Aufwand O(log(n)),    *)
(*                      genauer: Die Wege sind um 39% laenger als im best case.         *)
(*                                                                                      *)
(****************************************************************************************)
```

```
PROCEDURE lookup (b:Suchbaum; x: Element;       (* sucht im Suchbaum b nach Element x;  *)
                  VAR treffer: Suchbaum         (* liefert in treffer Verweis auf Knoten*)
                 ): BOOLEAN;                    (* und als Ergebnis TRUE, falls gefunden*)
BEGIN
        IF    b=NIL                 THEN treffer := b; RETURN FALSE
        ELSIF gleich (x,b^.inhalt) THEN treffer := b; RETURN TRUE
        ELSIF kleiner(x,b^.inhalt) THEN RETURN lookup(b^.links, x,treffer)
                                    ELSE RETURN lookup(b^.rechts,x,treffer)
        END
END lookup;


PROCEDURE insert (VAR b:Suchbaum; x:Element     (* fuegt in Suchbaum b Element x ein    *)
                 ):BOOLEAN;                     (* liefert TRUE, wenn erfolgreich, d.h. *)
BEGIN                                           (* x war nicht schon im Suchbaum        *)
        IF b=NIL THEN
                NEW(b);
                b^.inhalt := x;
                b^.links  := NIL;
                b^.rechts := NIL;
                RETURN TRUE
        ELSIF gleich (x,b^.inhalt) THEN RETURN FALSE
        ELSIF kleiner(x,b^.inhalt) THEN RETURN insert(b^.links,x)
                                    ELSE RETURN insert(b^.rechts,x)
        END;
END insert;

PROCEDURE delete (VAR b:Suchbaum; x:Element):BOOLEAN; (* loescht Element x.Liefert TRUE *)
VAR q: Suchbaum;                                      (* falls geklappt, sonst FALSE    *)

PROCEDURE del (VAR r : Suchbaum);        (* laufe solange nach rechts, bis kein rechter  *)
                                         (* Sohn mehr da. Danach ueberschreibe Knoten q  *)
                                         (* mit dem Wert dieses Endknotens *)
BEGIN
        IF r^.rechts <> NIL THEN del(r^.rechts)
        ELSE
                q^.inhalt := r^.inhalt;
                q := r; (* um DISPOSE anwenden zu koennen *)
                r := r^.links;
        END
END del;

BEGIN
        IF b=NIL THEN RETURN FALSE                    (* Element ist nicht im Suchbaum *)
        ELSIF kleiner(x,b^.inhalt) THEN RETURN delete(b^.links,x)
        ELSIF kleiner(b^.inhalt,x) THEN RETURN delete(b^.rechts,x)
        ELSE                    (* Knoten b traegt Element x und muss geloescht werden *)
                q := b;                              (* um  DISPOSE anwenden zu koennen *)
                IF   b^.rechts=NIL THEN b := b^.links
                ELSIF b^.links=NIL THEN b := b^.rechts
                ELSE (* 2 Soehne *)  del(b^.links)
                END;
                DISPOSE(q); RETURN TRUE;
        END
END delete;

END suchbaum.
```

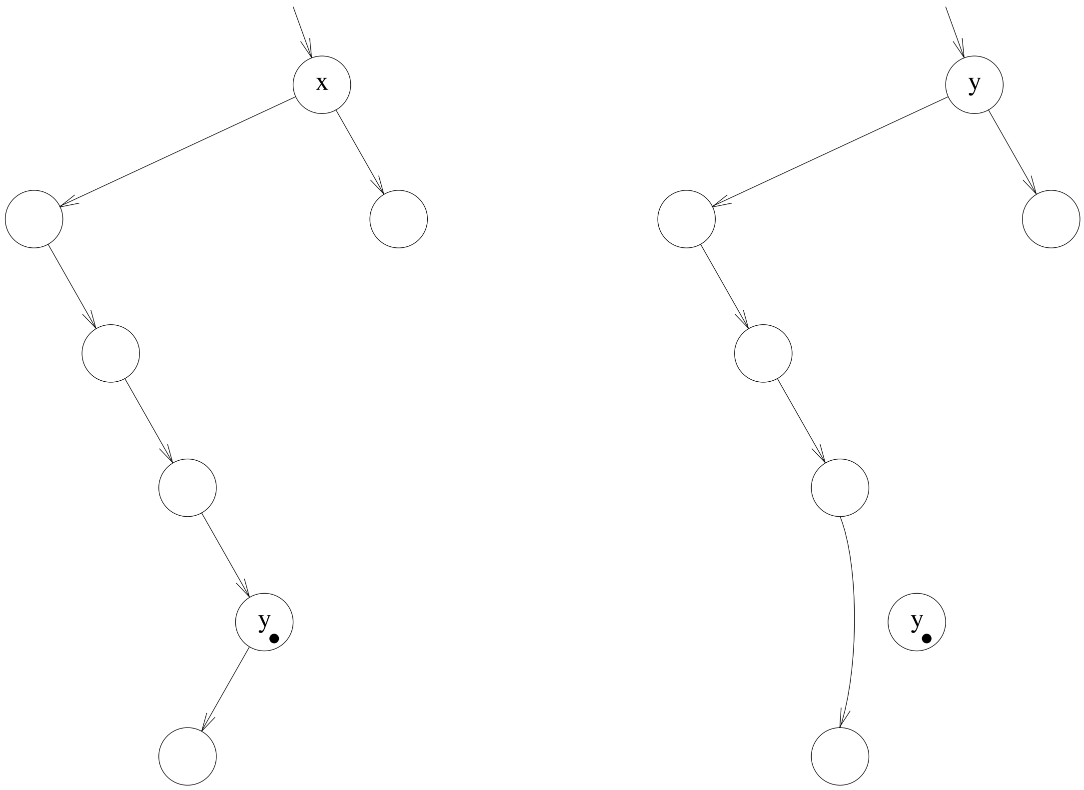
Sei x das Element in dem zu löschenden Knoten des Suchbaums.

Löschen eines Knotens ohne Söhne

Löschen eines Knotens mit einem Sohn



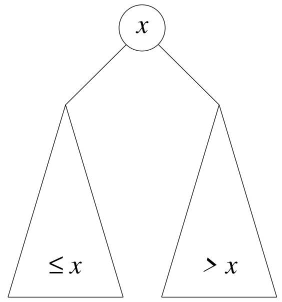
Löschen eines Knotens mit zwei Söhnen



```
(****************************************************************************************)
(*      Datum:  25.01.94  14:15 Uhr                                                      *)
(*      Autor:  Oliver Vornberger                                                        *)
(*      Modul:  baumtest.m2                                                              *)
(*              testet verschiedene Baumprozeduren                                       *)
(****************************************************************************************)
MODULE baumtest;

FROM element     IMPORT Element, createelement, readelement, writeelement;
FROM baum        IMPORT Baum, createtree, value;
FROM traversal   IMPORT preorder, inorder, postorder;
FROM tiefsuche   IMPORT tiefensuche;            FROM breitsuche  IMPORT breitensuche;
FROM postfixbaum IMPORT postfixbaumbau;         FROM praefixbaum IMPORT praefixbaumbau;
FROM suchbaum    IMPORT Suchbaum, insert, delete, lookup;
FROM InOut       IMPORT WriteString, WriteLn;

VAR     wurzel:            Baum;        (* createtree, value verlangen Baum             *)
        swurzel, streffer: Suchbaum;    (* insert, lookup, delete verlangen Suchbaum    *)
        x:                 Element;
BEGIN
        createtree(wurzel);             (* durch kreieren eines Baumes und anschliess.  *)
        swurzel := Suchbaum(wurzel);    (* Typkonvertierung entsteht ein Suchbaum       *)

        WriteString("Bitte Elemente fuer INSERT: ");
        createelement(x);
        WHILE readelement(x) DO
                writeelement(x);
                IF insert(swurzel,x)
                        THEN WriteString(" wurde eingefuegt"); createelement(x);
                        ELSE WriteString(" wurde abgelehnt");
                END; WriteLn;
        END;

        WriteString("Preorder:    "); preorder     (Baum(swurzel)); WriteLn;
        WriteString("Tiefensuche: "); tiefensuche  (Baum(swurzel)); WriteLn;
        WriteString("Breitensuche:"); breitensuche (Baum(swurzel)); WriteLn;

        WriteString("Bitte Elemente fuer LOOKUP: ");
        WHILE readelement(x) DO
                writeelement(x);
                IF lookup(swurzel,x,streffer)
                        THEN WriteString(" im Baum:");writeelement(value(Baum(streffer)));
                        ELSE WriteString(" wurde nicht gefunden");
                END; WriteLn;
        END;

        WriteString("Bitte Elemente fuer DELETE: ");
        WHILE readelement(x) DO
                writeelement(x);
                IF delete(swurzel,x)
                        THEN WriteString(" wurde entfernt  ");
                        ELSE WriteString(" nicht vorhanden ");
                END; WriteLn;
        END;

        WriteString("Bitte Postfixausdruck: "); wurzel := postfixbaumbau();
        WriteString("Inorder:     "); inorder (wurzel); WriteLn;
        WriteString("Bitte Praefixausdruck: "); wurzel := praefixbaumbau();
        WriteString("Postorder:     "); postorder (wurzel); WriteLn;
END baumtest.
```

Speichern von Mehrfachexemplaren in einem Suchbaum

1. Möglichkeit: Elemente doppelt halten



2. Möglichkeit: Zähler im Knoten mitführen

```
TYPE Knoten = RECORD
                inhalt: Baumelement;
                count: CARDINAL;
                links, rechts: Baum
              END;
```

Beim Einfügen: Zähler hochzählen, sofern Element schon vorhanden, sonst einfügen.

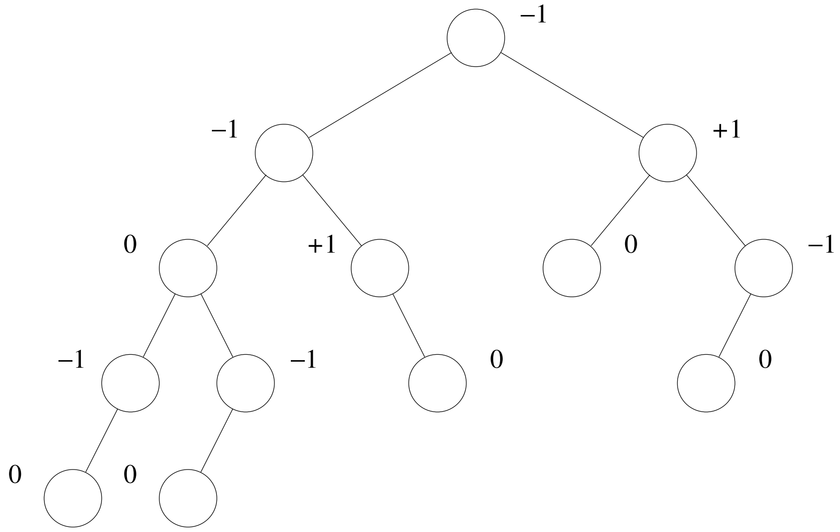
Beim Löschen: Zähler herunterzählen, sofern mehrfach da, sonst entfernen.

9.6. AVL-Baum

(benannt nach Adelson-Velskii und Landis, 1962)

Def.: Ein Knoten eines binären Baumes ist *ausgeglichen*, wenn sich die Höhen seiner beiden Söhne um höchstens 1 unterscheiden.

Def.: Ein binärer Suchbaum, in dem jeder Knoten ausgeglichen ist, heißt *AVL-Baum*.



Sei $bal(x)$ = Höhe des rechten Teilbaums von x minus Höhe des linken Teilbaums von x.

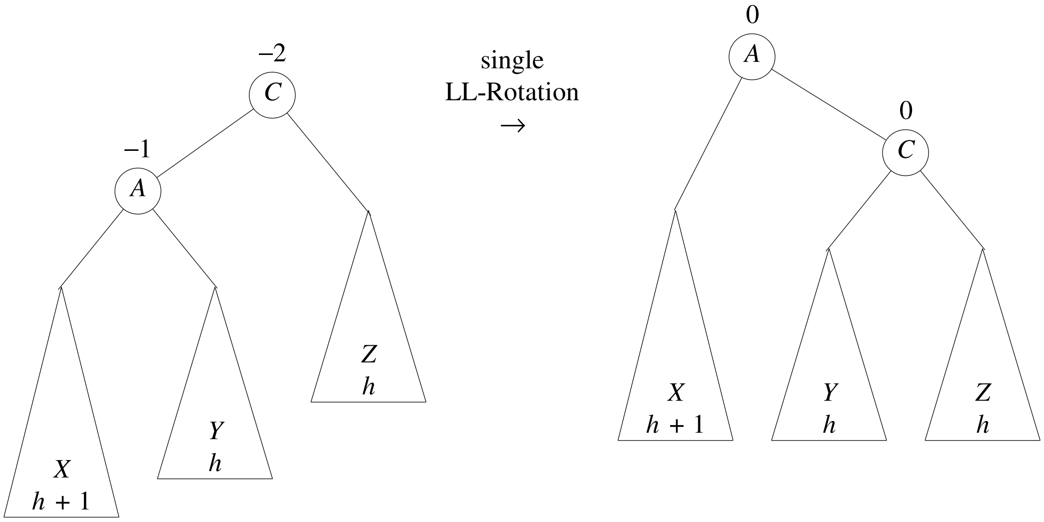
Def.: Ein Knoten x heißt *balanciert*, wenn $bal(x) \in \{-1, 0, 1\}$.

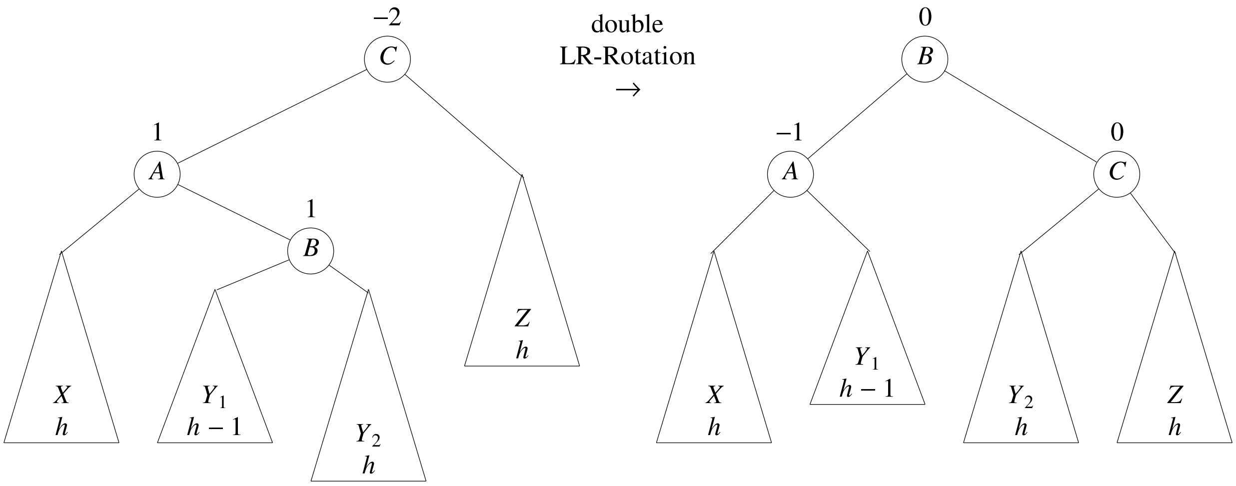
```
(****************************************************************************************)
(*      Datum:  31.01.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  avlbaum.d                                                               *)
(*              definiert den Typ AVL-Baum und die darauf definierten Operationen       *)
(*                                                                                      *)
(*              Ein AVL-Baum ist ein ausgeglichener binaerer Suchbaum                   *)
(*              In einem ausgeglichenen Baum ist jeder Knoten ausgeglichen              *)
(*              Ein Knoten ist ausgeglichen, wenn sich die Hoehen seiner Soehne         *)
(*              um hoechstens 1 unterscheiden                                           *)
(*                                                                                      *)
(****************************************************************************************)

DEFINITION MODULE avlbaum;

FROM element IMPORT Element;                    (* Element-Typ                          *)

TYPE AVLBaum;                                   (* Details sind im                      *)
                                                (* Implementation-Modul                 *)
                                                (* versteckt                            *)

PROCEDURE createavltree (VAR b:AVLBaum);        (* legt leeren AVL-Baum an              *)

PROCEDURE lookupavl (b:AVLBaum;                 (* sucht im AVL-Baum b nach dem         *)
                     x:Element;                 (* Element x. Liefert in treffer        *)
                     VAR treffer: AVLBaum       (* den Verweis auf den Knoten           *)
                     ): BOOLEAN;                (* liefert TRUE, falls erfolgreich      *)


PROCEDURE insertavl (VAR p:AVLBaum;             (* fuegt in AVL-Baum p                  *)
                     x:Element;                 (* das Element x ein und                *)
                     VAR hoeher:BOOLEAN         (* liefert in hoeher=TRUE, falls        *)
                                                (* Hoehe des Baums gewachsen ist        *)
                     ):BOOLEAN;                 (* liefert TRUE, falls erfolgreich      *)



PROCEDURE deleteavl (VAR b:AVLBaum;             (* Versucht im AVL-Baum b, das          *)
                     x:Element                  (* Element x zu loeschen                *)
                     ):BOOLEAN;                 (* liefert TRUE, falls erfolgreich      *)

END avlbaum.
```

```
(*********************************************************************************)
(*      Datum:  31.01.94  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  avlbaum.m2                                                       *)
(*              implementiert den Typ AVL—Baum und die darauf definierten Operationen   *)
(*                                                                               *)
(*********************************************************************************)

IMPLEMENTATION MODULE avlbaum;

FROM Storage    IMPORT ALLOCATE, DEALLOCATE;
FROM element    IMPORT Element, gleich, kleiner;        (* Element—Typ           *)
FROM error      IMPORT error;

TYPE AVLBaum =  POINTER TO AVLKnoten;

     AVLKnoten =RECORD
                        inhalt : Element;       (* Inhalt                        *)
                        links  : AVLBaum;       (* linker Teilbaum               *)
                        rechts : AVLBaum;       (* rechter Teilbaum              *)
                        bal    : [—1..+1]       (* rechte Hoehe minus linke Hoehe        *)
                END;


PROCEDURE createavltree (VAR b:AVLBaum);        (* legt leeren AVL—Baum an       *)
BEGIN
        b := NIL;
END createavltree;


PROCEDURE lookupavl (b:AVLBaum;                 (* sucht im AVL—Baum b nach dem  *)
                     x:Element;                 (* Element x. Liefert in treffer *)
                     VAR treffer: AVLBaum       (* den Verweis auf den Knoten    *)
                     ): BOOLEAN;                (* liefert TRUE, falls erfolgreich       *)
BEGIN
        IF    b=NIL                 THEN treffer := b; RETURN FALSE
        ELSIF gleich (x,b^.inhalt) THEN treffer := b; RETURN TRUE
        ELSIF kleiner(x,b^.inhalt) THEN RETURN lookupavl(b^.links, x,treffer)
                                    ELSE RETURN lookupavl(b^.rechts,x,treffer)
        END
END lookupavl;



PROCEDURE deleteavl (VAR b:AVLBaum;             (* Versucht im AVL—Baum b, das   *)
                     x:Element                  (* Element x zu loeschen         *)
                     ):BOOLEAN;                 (* liefert TRUE, falls erfolgreich       *)
BEGIN
(* siehe Wirth p. 231 *)
END deleteavl;
```

```
PROCEDURE insertavl (VAR p:AVLBaum;            (* fuegt in AVL-Baum p              *)
                     x:Element;                (* das Element x ein und            *)
                     VAR hoeher:BOOLEAN        (* liefert in hoeher=TRUE, falls    *)
                                               (* Hoehe des Baums gewachsen ist    *)
                     ):BOOLEAN;                (* liefert TRUE, falls erfolgreich  *)
VAR p1, p2 : AVLBaum;
BEGIN
        IF p=NIL THEN
                NEW(p); IF p=NIL THEN error("in insertavl: kein Platz da") END;
                p^.inhalt := x;
                p^.links  := NIL;
                p^.rechts := NIL;
                p^.bal    := 0;
                hoeher    := TRUE;
                RETURN TRUE
        ELSIF gleich(x,p^.inhalt) THEN RETURN FALSE
        ELSIF kleiner(x,p^.inhalt) THEN (* links einsteigen *)
                IF insertavl(p^.links, x, hoeher) THEN
                        (* Element wurde eingefuegt *)
                        IF hoeher THEN (* linker Teilbaum ist gewachsen *)
                            CASE p^.bal OF
                                1: p^.bal :=  0; hoeher := FALSE |
                                0: p^.bal := -1; hoeher := TRUE  |
                               -1: (* rebalancieren erforderlich *)
                                  p1 := p^.links;   (* Hilfszeiger fuer LL *)
                                  p2 := p1^.rechts; (* Hilfszeiger fuer LL + LR *)

                                  IF p1^.bal = -1 THEN (* single LL-Rotation *)
                                    p^.links   := p1^.rechts;
                                    p1^.rechts := p;
                                    p^.bal     := 0;
                                    p          := p1;

                                  ELSE (* double LR-Rotation *)
                                    p1^.rechts := p2^.links;
                                    p2^.links  := p1;
                                    p^.links   := p2^.rechts;
                                    p2^.rechts := p;
                                    IF p2^.bal=-1 THEN p^.bal := 1 ELSE p^.bal :=0 END;
                                    IF p2^.bal=+1 THEN p1^.bal:=-1 ELSE p1^.bal:=0 END;
                                    p          := p2;

                                  END;(* IF p1^.bal *)
                                  p^.bal := 0;
                                  hoeher := FALSE

                            END;(* case *)
                        END; (* IF hoeher *)
                        RETURN TRUE; (* es wurde eingefuegt, ggf. balanciert *)
                ELSE RETURN FALSE
                END (* IF insert *)
        ELSE (*rechts einsteigen, analog zu links einsteigen *)
        END (* p=NIL, gleich, kleiner, groesser *)
END insertavl;

END avlbaum.
```

Rotationen für AVL-Baum bei linksseitigem Übergewicht

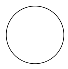




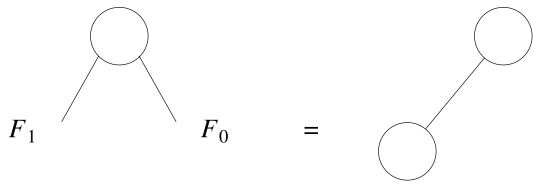
Ungünstige AVL-Bäume sind *Fibonacci* -Bäume

F_0: Höhe 0

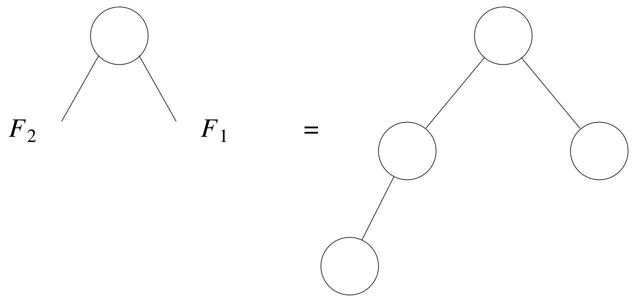
F_1: Höhe 1



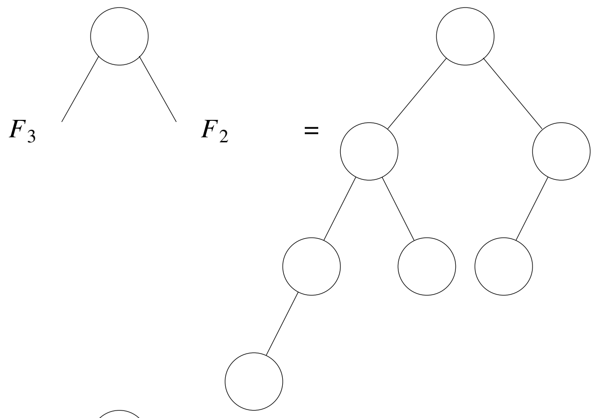
F_2: Höhe 2



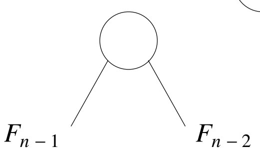
F_3: Höhe 3



F_4: Höhe 4



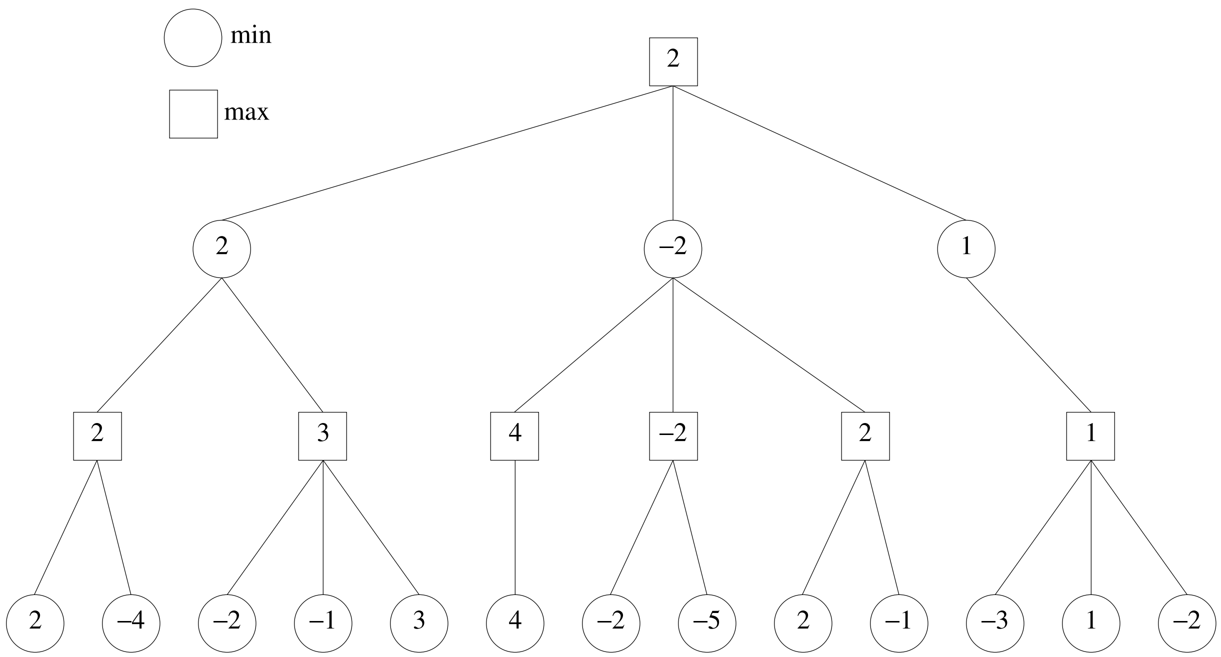
F_n: Höhe n



Satz: Die Höhe eines AVL-Baumes mit n Knoten ist $\leq 1.45 \cdot \log n$, d.h. höchstens 45% größer als erforderlich.

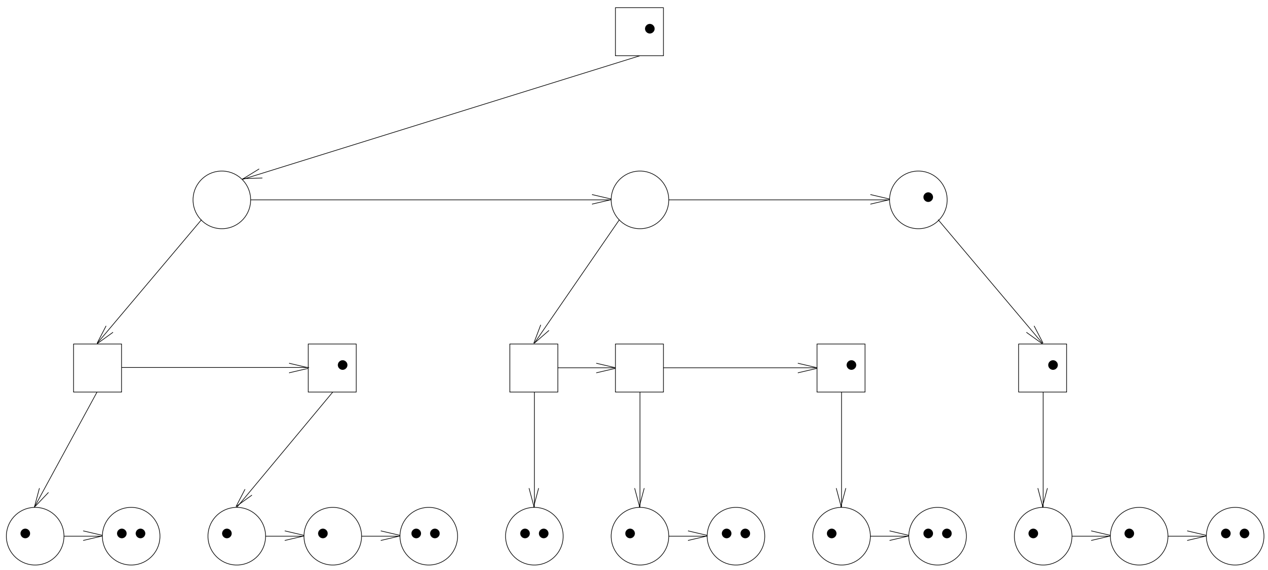
9.7. Spielbaum

Beispiel für einen Spielbaum



Implementation

(Jeder Knoten hat einen Verweis auf den ältesten Sohn und den nächstjüngeren Bruder.)



```
(****************************************************************************************)
(*      Datum:  31.01.94  14:15 Uhr                                                    *)
(*      Autor:  Oliver Vornberger                                                      *)
(*      Modul:  spielbaum.d                                                            *)
(*              definiert den Typ Spielbaum und die darauf definierten Operationen     *)
(*                                                                                     *)
(*              Ein Spielbaum ist ein Baum mit zwei Typen von Knoten:                  *)
(*              Minimum-Knoten und Maximum-Knoten.                                     *)
(*                                                                                     *)
(*              Die Knoten repraesentieren Spielstellungen                             *)
(*                                                                                     *)
(*              Der Wert eines Blattes wird bestimmt durch eine                        *)
(*              statische Stellungsbewertung                                           *)
(*                                                                                     *)
(*              Der Wert eines Minimum-Knotens ist das Minimum der Werte seiner Soehne *)
(*                                                                                     *)
(*              Der Wert eines Maximum-Knotens ist das Maximum der Werte seiner Soehne *)
(*                                                                                     *)
(*                                                               h-1                   *)
(*              Obacht: Bei Hoehe h und Verzweigungsgrad d gibt es d    Blaetter       *)
(*                                                                                     *)
(*              Z.B. 8 Halbzuege mit je 20 Alternativen => 25.000.000.000 Blaetter     *)
(****************************************************************************************)

DEFINITION  MODULE spielbaum;

TYPE Spielbaum;

PROCEDURE minmax(s: Spielbaum) : INTEGER;               (* bestimmt den Wert des       *)
                                                        (* Spielbaums s                *)

END spielbaum.
```

```
(********************************************************************************)
(*      Datum:  31.01.94  14:15 Uhr                                             *)
(*      Autor:  Oliver Vornberger                                               *)
(*      Modul:  spielbaum.m2                                                    *)
(*              implementiert den Typ Spielbaum und die darauf definierten Operationen  *)
(*                                                                              *)
(********************************************************************************)

IMPLEMENTATION MODULE spielbaum;

TYPE Stellung  = RECORD END;                        (* Kodierung der Stellung       *)

TYPE Spielbaum = POINTER TO Knoten;

     Knoten    = RECORD
                       typ             : (min,max);     (* Maximierung oder Minimierung *)
                       stellung        : Stellung;      (* Spielstellung                *)
                       erstersohn      : Spielbaum;     (* Verweis auf aeltesten Sohn   *)
                       naechsterbruder : Spielbaum;     (* Verweis auf juengeren Bruder *)
                 END;

PROCEDURE statisch(s:Stellung):INTEGER;   (* fuehrt statische Stellungsbewertung durch  *)
BEGIN
END statisch;


PROCEDURE minmax(s: Spielbaum) : INTEGER; (* wertet den Spielbaum s aus, indem rekursiv *)
                                          (* die Soehne ausgewertet werden. Es wird je  *)
                                          (* nach Typ,das Minimum oder Maximum bestimmt *)
VAR best, bruderwert: INTEGER;
VAR bruder      : Spielbaum;
BEGIN
        IF s^.erstersohn = NIL                      (* Falls keine Soehne existieren*)
        THEN
                RETURN statisch(s^.stellung)        (* liefere den statischen Wert  *)
        ELSE
                bruder := s^.erstersohn;

                IF s^.typ = min
                        THEN best := MAX(INTEGER)   (* setze vorlaeufigen Wert      *)
                        ELSE best := MIN(INTEGER)   (* setze vorlaeufigen Wert      *)
                END;

                WHILE bruder <> NIL DO              (* durchlaufe alle Soehne       *)
                        bruderwert := minmax(bruder);   (* bestimme Wert des Knotens    *)
                        IF ((s^.typ = min) AND (bruderwert < best)) OR
                           ((s^.typ = max) AND (bruderwert > best))
                           THEN best := bruderwert  (* verbessere bisherigen Wert   *)
                        END;
                        bruder := bruder^.naechsterbruder;
                END;

                RETURN best                         (* liefere Optimum zurueck      *)
        END

END minmax;

END spielbaum.
```

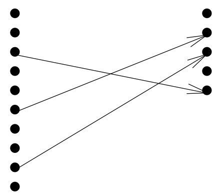
10. Hashing

Zum Abspeichern und Wiederfinden von Elementen wäre folgende Funktion hilfreich:

```
f: Element -> CARDINAL
```

Dann könnte Element x bei Adresse $f(x)$ gespeichert werden.

Problem: Anzahl der möglichen Elemente >> Anzahl der Adressen



Gegeben Adressen von 0 bis $N - 1$.

Sei `TYPE Element = CHAR;`
oder `TYPE Element = CARDINAL;`

dann wäre $f(x) =$ `ORD(x) MOD N` eine *Hashfunktion.*

Sei $x = x_{n-1}\, x_{n-2} \cdots x_1\, x_0$ ein String, dann wäre $f(x) = \left[\sum_{i=0}^{n-1} ORD(x_i) \right] MOD\ N$ eine Hashfunktion.

Gilt: $f(x) = f(y)$, so liegt eine Kollision vor.

Behandlung der ***Kollision***

10.1. Offenes Hashing

```
VAR h : ARRAY [0 .. N - 1] OF Liste;
```

Alle Elemente x mit $f(x) = y$ befinden sich in der Liste $h[y]$. Bei N Buckets und n Elementen enthält jede Liste im Mittel $\frac{n}{N}$ Elemente.

10.2. Geschlossenes Hashing

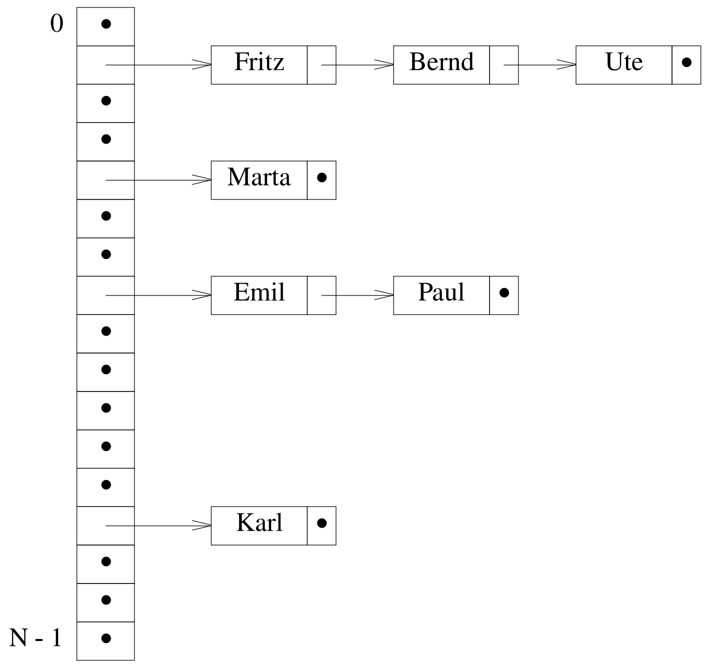
```
TYPE Eintrag = RECORD
                     zustand = (LEER, BELEGT, GELOESCHT);
                     inhalt  : Element
                   END;
VAR h : ARRAY [0 .. N - 1] OF Eintrag;
```

Falls $y = f(x)$ schon belegt ist, so suche für x einen Alternativplatz.

$y + 1, y + 2, y + 3, y + 4, ...$	*lineares Sondieren*
$y + 1, y + 4, y + 9, y + 16, ...$	*quadratisches Sondieren**
$y + f_2(x)$	*Double Hashing* (Schrittweite wird durch 2. Hashfunktion bestimmt)

* Nicht alle Elemente werden besucht, aber mindestens $\frac{N}{2}$.

Implementation des offenen Hashings



Implementation des geschlossenen Hashings

0	L	
	B	Fritz
	B	Bernd
	L	
	B	Marta
	B	Ute
	L	
	B	Emil
	G	
	L	
	L	
	G	
	L	
	B	Karl
	L	
	L	
N - 1	B	Paul

L = leer
B = belegt
G = gelöscht

```
(****************************************************************************************)
(*      Datum:  01.02.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  hashing.d                                                               *)
(*              definiert den Typ Hashtabelle und die darauf definierten Operationen    *)
(*                                                                                      *)
(****************************************************************************************)

DEFINITION MODULE hashing;

FROM element    IMPORT Element;

TYPE Hashtabelle;                            (* Details im Implementation-Modul      *)


PROCEDURE createhashtabelle(VAR h:Hashtabelle); (* legt leere Hashtabelle an          *)


PROCEDURE insert (h: Hashtabelle;            (* versucht, in die Hashtabelle h       *)
                  x: Element                 (* das Element x einzufuegen            *)
                 ): BOOLEAN;                 (* liefert TRUE, falls erfolgreich      *)


PROCEDURE lookup (h: Hashtabelle;            (* sucht in der Hashtabelle h           *)
                  x: Element                 (* das Element x                        *)
                 ): BOOLEAN;                 (* liefert TRUE, falls erfolgreich      *)


PROCEDURE delete (h: Hashtabelle;            (* versucht, aus der Hashtabelle h      *)
                  x: Element                 (* das Element x zu entfernen           *)
                 ): BOOLEAN;                 (* liefert TRUE, falls erfolgreich      *)


PROCEDURE zeigetabelle(h:Hashtabelle);       (* gibt Tabelle aus                     *)

END hashing.
```

```
(*********************************************************************************)
(*      Datum:  01.02.94  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  ofhashing.m2                                                     *)
(*              implementiert den Typ Hashtabelle und die darauf definierten Operationen*)
(*              als offenes Hashing mit einem array von Listen                   *)
(*********************************************************************************)

IMPLEMENTATION MODULE hashing;

FROM Storage    IMPORT ALLOCATE, DEALLOCATE;
FROM InOut      IMPORT WriteCard, WriteString, Write, WriteLn;
FROM element    IMPORT Element, gleich, ord, writeelement;
FROM liste      IMPORT Liste, createlist, reset, elem, advance, endpos;
IMPORT liste;   (* um liste.insert und liste.delete verwenden zu koennen *)

CONST N          = 25;

TYPE Hashtabelle = POINTER TO ARRAY[0..N-1] OF Liste;


PROCEDURE hash (x:Element                        (* berechnet aus dem Element x          *)
               ):CARDINAL;                       (* den Hashwert                         *)
BEGIN
        RETURN (ord(x) MOD N)
END hash;


PROCEDURE createhashtabelle(VAR h:Hashtabelle); (* legt leere Hashtabelle an            *)
VAR i : CARDINAL;
BEGIN
        NEW(h);
        FOR i:=0 TO N-1 DO createlist(h^[i]);
        END
END createhashtabelle;


PROCEDURE zeigetabelle (h:Hashtabelle);         (* gibt Tabelle aus                     *)
VAR i : CARDINAL;
BEGIN
        FOR i:=0 TO N-1 DO
                WriteCard(i,4); Write(' ');
                reset(h^[i]);
                WHILE NOT endpos(h^[i]) DO
                        writeelement(elem(h^[i])); Write(' ');
                        advance(h^[i]);
                END;
                WriteLn
        END;
END zeigetabelle;
```

```
PROCEDURE finde  (l:Liste;                      (* sucht in Liste l                  *)
                  x:Element                     (* nach dem Element x                *)
                  ) : BOOLEAN;                  (* liefert TRUE, falls erfolgreich   *)
                                                (* Seiteneffekt: Bei erfolgreicher Suche*)
                                                (* wird die Liste so verlassen, dass *)
                                                (* x das aktuelle Element ist        *)
BEGIN
        reset(l);
        WHILE NOT endpos(l) AND NOT gleich(x,elem(l)) DO advance(l) END;
        RETURN NOT endpos(l);
END finde;


PROCEDURE lookup (h: Hashtabelle;               (* sucht in der Hashtabelle h        *)
                  x: Element                    (* das Element x                     *)
                  ): BOOLEAN;                   (* liefert TRUE, falls erfolgreich   *)
VAR index : CARDINAL;
BEGIN
        index := hash(x);
        IF finde(h^[index],x) THEN RETURN TRUE ELSE RETURN FALSE END;
END lookup;


PROCEDURE insert (h:Hashtabelle;                (* versucht, in die Hashtabelle h    *)
                  x:Element                     (* das Element x einzufuegen         *)
                  ):BOOLEAN;                    (* liefert TRUE, falls erfolgreich   *)
VAR index : CARDINAL;
BEGIN
        index := hash(x);
        IF NOT finde(h^[index],x) THEN liste.insert(h^[index],x); RETURN TRUE
                                  ELSE RETURN FALSE
        END
END insert;


PROCEDURE delete (h: Hashtabelle;               (* versucht, aus der Hashtabelle h   *)
                  x:Element                     (* das Element x zu entfernen        *)
                  ):BOOLEAN;                    (* liefert TRUE, falls erfolgreich   *)
VAR index : CARDINAL;
BEGIN
        index := hash(x);
        IF finde(h^[index],x) THEN liste.delete(h^[index]); RETURN TRUE
                              ELSE RETURN FALSE
        END
END delete;


END hashing.
```

```
(******************************************************************************)
(*      Datum:  01.02.94  14:15 Uhr                                          *)
(*      Autor:  Oliver Vornberger                                             *)
(*      Modul:  gehashing.m2                                                  *)
(*              implementiert den Typ Hashtabelle und die darauf definierten Operationen*)
(*              als geschlossenes Hashing mit einem array von Elementen      *)
(******************************************************************************)

IMPLEMENTATION MODULE hashing;

FROM element    IMPORT Element, gleich, ord, writeelement;
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;
FROM InOut      IMPORT WriteCard, WriteString, WriteLn;

CONST N          = 8;

TYPE Hashtabelle = POINTER TO ARRAY[0..N-1] OF
                   RECORD
                        inhalt : Element;
                        zustand: (LEER, BELEGT, GELOESCHT)
                   END;


PROCEDURE hash (x:Element                          (* berechnet aus dem Element x          *)
               ):CARDINAL;                         (* den Hashwert                         *)
BEGIN
        RETURN (ord(x) MOD N)
END hash;


PROCEDURE createhashtabelle(VAR h:Hashtabelle); (* legt leere Hashtabelle an           *)
VAR i : CARDINAL;
BEGIN
        NEW(h);
        FOR i:=0 TO N-1 DO
                h^[i].zustand := LEER;
        END
END createhashtabelle;


PROCEDURE zeigetabelle (h:Hashtabelle);          (* gibt Tabelle aus *)
VAR i : CARDINAL;
BEGIN
        FOR i:=0 TO N-1 DO
                WriteCard(i,4);
                CASE h^[i].zustand OF
                        LEER:      WriteString(" L ") |
                        BELEGT:    WriteString(" B ") |
                        GELOESCHT: WriteString(" G ")
                END;
                IF h^[i].zustand = BELEGT THEN WriteCard(hash(h^[i].inhalt),4);
                                               WriteString(' ');
                                               writeelement(h^[i].inhalt);
                END;
                WriteLn;
        END;
END zeigetabelle;
```

```
(****************************************************************************************)
(*      Ein Loch in der Hashtabelle ist ein leeres Feld oder ein geloeschtes Feld       *)
(*      Der Lochindex ist der Index des zuerst gefundenen Loches                        *)
(****************************************************************************************)

PROCEDURE finde   (h: Hashtabelle;                (* sucht in der Hashtabelle h          *)
                   x: Element;                    (* das Element x. Falls gefunden:      *)
                   VAR index : CARDINAL           (* liefert in index die array-Position *)
                                                  (* des Elementes bzw. eines Loches     *)
                   ): BOOLEAN;                    (* und TRUE, sonst FALSE               *)

VAR     versuche     : CARDINAL;                  (* Anzahl der Versuche Sondierschritte *)
VAR     lochindex    : CARDINAL;                  (* Index eines geloeschten Elementes   *)
VAR     d            : CARDINAL;                  (* Sondierschrittweite                 *)
VAR     lochgefunden : BOOLEAN;                   (* TRUE, falls Loch gefunden           *)

BEGIN
        versuche := 0;
        lochgefunden := FALSE;
        d := 1;
        index := hash(x);

        WHILE NOT (((h^[index].zustand = BELEGT) AND (gleich(h^[index].inhalt,x)))
                      OR  (h^[index].zustand = LEER) OR (versuche=N)) DO

                        (* nur wegen anschliessendem insert *)
                        IF (NOT lochgefunden) AND (h^[index].zustand = GELOESCHT)
                               THEN lochgefunden := TRUE; lochindex := index END;

                        index    := (index + d ) MOD N;
                        d        := d + 2;
                        versuche := versuche +1;
        END;

        IF (h^[index].zustand=BELEGT) AND (gleich(h^[index].inhalt,x))
                THEN RETURN TRUE
                ELSE IF lochgefunden THEN index := lochindex END; RETURN FALSE
        END;

END finde;
```

```
PROCEDURE lookup (h: Hashtabelle;              (* sucht in der Hashtabelle h          *)
                  x: Element                   (* das Element x                       *)
                  ): BOOLEAN;                  (* liefert TRUE, falls erfolgreich     *)
VAR index : CARDINAL;                          (* dummy—Variable                      *)
BEGIN
        IF finde(h,x,index)
                THEN RETURN TRUE
                ELSE RETURN FALSE
        END;
END lookup;



PROCEDURE insert (h: Hashtabelle;              (* versucht, in die Hashtabelle h      *)
                  x: Element                   (* das Element x einzufuegen           *)
                  ): BOOLEAN;                  (* liefert TRUE, falls erfolgreich     *)
VAR index : CARDINAL;
BEGIN
        IF (NOT finde(h,x,index)) AND (NOT (h^[index].zustand = BELEGT))
        THEN
                h^[index].inhalt := x;
                h^[index].zustand:= BELEGT;
                RETURN TRUE
        ELSE
                RETURN FALSE
        END

END insert;



PROCEDURE delete (h: Hashtabelle;              (* versucht, aus der Hashtabelle h     *)
                  x: Element                   (* das Element x zu entfernen          *)
                  ): BOOLEAN;                  (* liefert TRUE, falls erfolgreich     *)
VAR index : CARDINAL;
BEGIN
        IF finde(h,x,index)
        THEN
                h^[index].zustand := GELOESCHT;
                RETURN TRUE
        ELSE
                RETURN FALSE
        END
END delete;


END hashing.
```

```
(****************************************************************************************)
(*      Datum:  01.02.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  hashingtest.m2                                                          *)
(*              reiht eine Folge von Elementen mit "insert" in eine Hashtabelle ein     *)
(*              sucht sie mit "lookup" und loescht sie mit "delete"                     *)
(****************************************************************************************)

MODULE hashingtest;

FROM element IMPORT Element, createelement, readelement, writeelement;
FROM hashing IMPORT Hashtabelle, createhashtabelle, zeigetabelle, insert, delete, lookup;
FROM InOut   IMPORT WriteString, WriteLn;

VAR     h     : Hashtabelle;
        x     : Element;
        index : CARDINAL;

BEGIN
        createhashtabelle(h);

        WriteString("Bitte Elemente fuer INSERT: ");
        createelement(x);
        WHILE readelement(x) DO
                writeelement(x);
                IF insert(h,x) THEN
                        WriteString(" wurde eingefuegt");
                        createelement(x);
                ELSE
                        WriteString(" wurde abgelehnt");
                END;
                WriteLn;
        END;

        zeigetabelle(h);

        WriteString("Bitte Elemente fuer LOOKUP: ");
        WHILE readelement(x) DO
                writeelement(x);
                IF lookup(h,x)
                        THEN WriteString(" wurde gefunden");
                        ELSE WriteString(" wurde nicht gefunden");
                END;
                WriteLn;
        END;

        WriteString("Bitte Elemente fuer DELETE: ");
        WHILE readelement(x) DO
                writeelement(x);
                IF delete(h,x)
                        THEN WriteString(" wurde entfernt  ");
                        ELSE WriteString(" nicht vorhanden ");
                END;
                WriteLn;
        END;

        zeigetabelle(h);

END hashingtest.
```

Perfekte Hashfunktion

keine Kollision

z.B. f:	{ braun,	rot,	blau,	violett,	türkis }	$\rightarrow$ IN
	5	3	4	7	6	
	2	0	1	4	3	

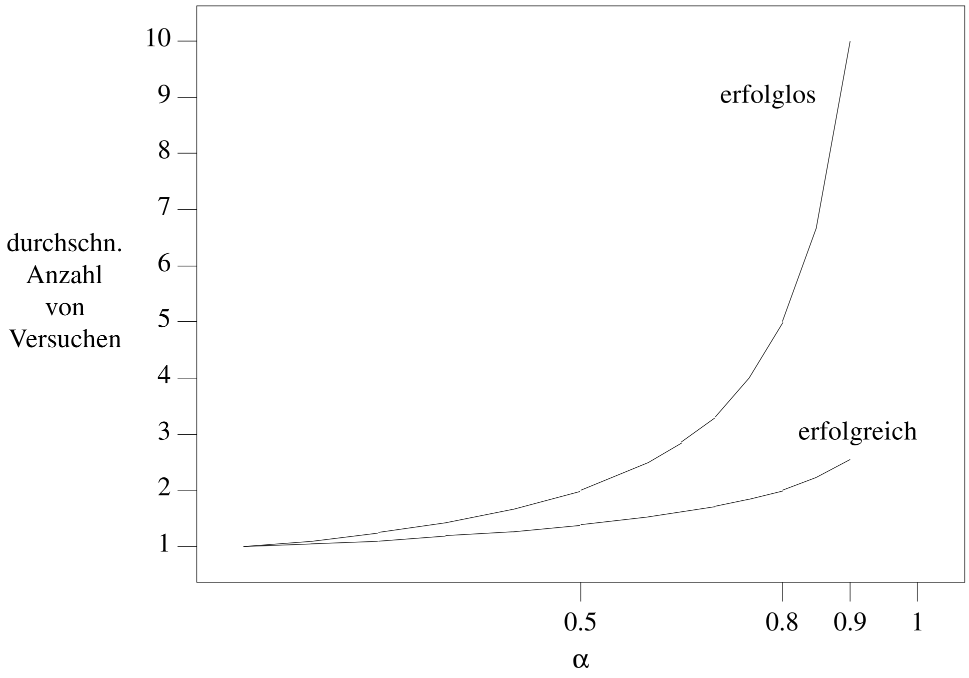
$f(w) = \text{Länge}(w) - 3 \in [0 .. 4]$

Laufzeit bei geschlossenem Hashing

Sei $\alpha \leq 1$ der Auslastungsfaktor. Dann ergibt sich für die Anzahl der Schritte mit Double-Hashing als Kollisionsstrategie bei

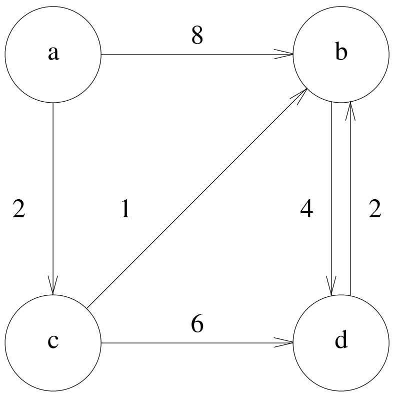
- erfolgloser Suche: $\approx \frac{1}{1-\alpha} = 5.0$, für $\alpha = 0.8$
- erfolgreicher Suche: $\approx -\frac{ln(1-\alpha)}{\alpha} = 2.01$, für $\alpha = 0.8$

d.h. in 2 Schritten wird von 800.000 Elementen aus einer 1.000.000 großen Tabelle das richtige gefunden.



11. Graphen

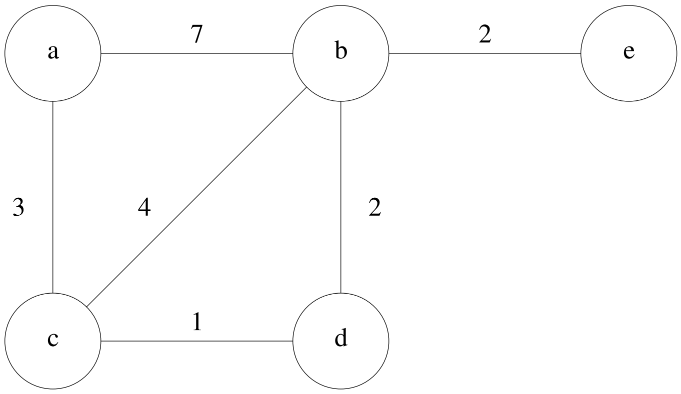
Ein *gerichteter Graph* $G = (V,\ E)$ besteht aus *Knotenmenge* V
und *Kantenmenge* $E \subseteq V \times V$



$V = \{a,\ b,\ c,\ d\}$
$E = \{(a,\ c),\ (a,\ b),\ (c,\ b),\ (c,\ d),\ (d,\ b)\}$

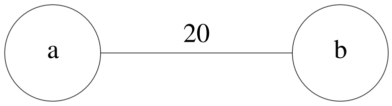
Kanten können gewichtet sein durch eine *Kostenfunktion* $c : E \rightarrow \mathbb{Z}$.

Ein *ungerichteter Graph* $G = (V,\ E)$ besteht aus *Knotenmenge* V
und *Kantenmenge* $E \subseteq P_2\ (V)$ = 2-elem. Teilmengen von V.

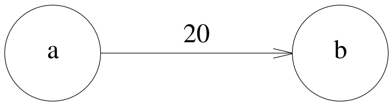


Mit Graphen können zwischen Objekten ($\hat{=}$ Knoten) binäre Beziehungen ($\hat{=}$ Kanten) modelliert werden.

1.) Orte mit Entfernungen



Länge
Kosten
Dauer

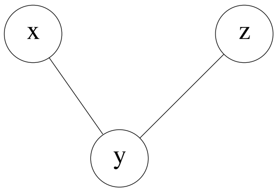


2.) Personen mit Beziehungen

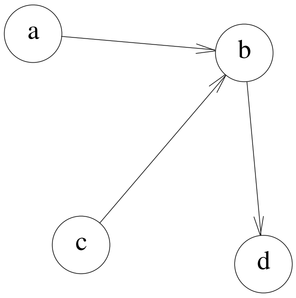
verheiratet mit

3.) Ereignisse mit Vorrang

a muß vor *b* geschehen



x ist zu *y* *adjazent*
x und *y* sind *Nachbarn*
x und *z* sind *unabhängig*
Der *Grad* von *y* ist 2



a ist *Vorgänger* von *b*
b ist *Nachfolger* von *a*
Eingangsgrad von *b* ist 2
Ausgangsgrad von *b* ist 1

Ein *Weg* ist eine Folge von adjazenten Knoten.
Ein *Kreis* ist ein Weg mit Anfangsknoten = Endknoten.

11.1. Implementation von Graphen

Es sei jedem Knoten eindeutig ein Index zugeordnet. Für den gerichteten Graphen auf Seite 127 ergibt sich:

Index	Knoten
0	a
1	b
2	c
3	d

Implementation durch Adjazenzmatrix

	0	1	2	3
0	0	1	1	0
1	1	0	0	1
2	0	1	0	1
3	0	1	0	0

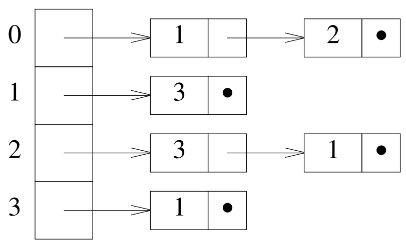
$$m[i, j] := \begin{cases} 1, & \text{falls } (i, j) \in E \\ 0 & \text{sonst} \end{cases}$$

	0	1	2	3
0	0	8	2	∞
1	∞	0	∞	4
2	∞	1	0	6
3	∞	2	∞	0

$$m[i, j] := \begin{cases} c(i, j), & \text{falls } (i, j) \in E \\ 0, & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Platzbedarf = $O(|V|^2)$.
Direkter Zugriff auf Kante (i, j) möglich.
Kein effizientes Verarbeiten der Nachbarn eines Knotens.
Sinnvoll bei dichtbesetzten Graphen.
Sinnvoll bei Algorithmen, die wahlfreien Zugriff auf eine Kante benötigen.

Implementation durch Adjazenzlisten



i-te Liste enthält *j*
falls $(i, j) \in E$

Platzbedarf = $O(|E|)$
Kein effizienter Zugriff auf Kante (x, y) möglich.
Sinnvoll bei dünnbesetzten Graphen.
Sinnvoll bei Algorithmen, die, gegeben ein Knoten x, dessen Nachbarn verarbeiten müssen.

```
(*********************************************************************************)
(*      Datum:  07.02.94  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  graph.d                                                          *)
(*              fuehrt den Typ Graph ein und die darauf definierten Operationen  *)
(*                                                                               *)
(*              Ein Graph besteht aus einer Menge von Knoten und Kanten          *)
(*              Die Knoten repraesentieren Elemente                              *)
(*              Die Kanten repraesentieren eine (ungerichtete oder gerichtete)   *)
(*              binaere Beziehung zwischen den Elementen                         *)
(*                                                                               *)
(*              Die n Knoten eines Graphen sind von 0 bis n-1 durchnumeriert     *)
(*              Durch knoten(i) erhaelt man das durch Knoten i repraesentierte Element  *)
(*              Durch index (x) erhaelt man den Index des Knotens fuer Element x *)
(*                                                                               *)
(*              Eine typische Schleife zum Bearbeiten aller Nachbarn von Knoten i *)
(*              im Graphen g lautet:                                             *)
(*                                                                               *)
(*              resetneighbors(g,i);                                             *)
(*              WHILE moreneighbors(g,i) DO                                      *)
(*                      j := nextneighbor(g,i);                                  *)
(*                      <bearbeite Knoten mit Index j>                           *)
(*              END;                                                             *)
(*********************************************************************************)
DEFINITION MODULE graph;

FROM element    IMPORT Element;

CONST maxknoten = 100;

TYPE Graph;                                  (* Details im Implementationmodul      *)

PROCEDURE creategraph(VAR g:Graph);          (* legt leeren Graphen an              *)

PROCEDURE number(g:Graph): CARDINAL;         (* liefert Anzahl der Knoten           *)

PROCEDURE knoten(g:Graph; i:CARDINAL): Element; (* liefert Element im Knoten mit Index i*)

PROCEDURE index(g:Graph; x: Element): CARDINAL; (* liefert Index vom Knoten zu Element x*)

PROCEDURE edge(g:Graph; i,j:CARDINAL): BOOLEAN; (* liefert TRUE, falls (i,j) existiert  *)

PROCEDURE resetneighbors(g:Graph;            (* bringt die Nachbarn von Knoten i    *)
                         i:CARDINAL);        (* in Ausgangsstellung                 *)

PROCEDURE moreneighbors(g:Graph;             (* liefert TRUE, falls es weitere      *)
                        i:CARDINAL): BOOLEAN;  (* Nachbarn des Knotens i gibt       *)

PROCEDURE nextneighbor(g:Graph;              (* liefert Index des naechsten         *)
                       i:CARDINAL): CARDINAL;  (* Nachbarn von Knoten i             *)

PROCEDURE readgraph(g:Graph);                (* liest einen Graphen ein             *)

PROCEDURE writegraph(g:Graph);               (* gibt einen Graphen aus              *)

END graph.
```

```
(******************************************************************************)
(*      Datum:  07.02.94  14:15 Uhr                                           *)
(*      Autor:  Oliver Vornberger                                             *)
(*      Modul:  graph.m2                                                      *)
(*              implementiert den Typ Graph und die darauf definierten Operationen     *)
(*              Verwendet wird ein array von Adjazenzlisten, d.h. im Graphen g   *)
(*              befinden sich die Indizes der Nachbarn von Knoten i         *)
(*              in der Liste g^.nachbarn[i]                                   *)
(*                                                                            *)
(*              Hierzu wird der ADT Liste ueber den CARDINALs benoetigt:      *)
(*                                                                            *)
(*              DEFINITION MODULE liste;                                      *)
(*              TYPE Listenelement = CARDINAL;                                *)
(*                                                                            *)
(******************************************************************************)
IMPLEMENTATION MODULE graph;

FROM error      IMPORT  error;
FROM element    IMPORT  Element, gleich, createelement, readelement, writeelement;
FROM liste      IMPORT  Liste, createlist, reset, endpos, elem, insert, advance;
FROM InOut      IMPORT  WriteString, WriteLn;
FROM Storage    IMPORT ALLOCATE;

TYPE Graph = POINTER TO
             RECORD
                anzahl   : CARDINAL;                         (* Anzahl der Knoten    *)
                inhalt   : ARRAY[0..maxknoten-1] OF Element;  (* Namen der Knoten     *)
                nachbarn : ARRAY[0..maxknoten-1] OF Liste;    (* Nachbarn eines Knoten*)
             END;


PROCEDURE rangeok(g: Graph; i:CARDINAL): BOOLEAN;  (* liefert TRUE, falls i ein fuer    *)
BEGIN                                               (* den Graphen g gueltiger Index ist *)
        RETURN (0<=i) AND (i < g^.anzahl)
END rangeok;


PROCEDURE creategraph(VAR g:Graph);                    (* legt leeren Graphen an       *)
VAR i : CARDINAL;
BEGIN
        NEW(g);
        FOR i:=0 TO maxknoten-1 DO
                createlist(g^.nachbarn[i])
        END;
        g^.anzahl := 0;
END creategraph;
```

```
PROCEDURE number(g:Graph): CARDINAL;                  (* liefert Anzahl der Knoten    *)
BEGIN
        RETURN g^.anzahl
END number;



PROCEDURE knoten(g:Graph;i:CARDINAL): Element;        (* liefert Knoten mit Index i   *)
BEGIN
        IF rangeok(g,i)
                THEN RETURN g^.inhalt[i]
                ELSE error("in element: ungueltiger Index")
        END
END knoten;



PROCEDURE index(g: Graph;                             (* sucht im Graphen g           *)
                x: Element                            (* zum Element x den Index      *)
                ) : CARDINAL;                         (* und liefert ihn ab           *)
VAR i : CARDINAL;
BEGIN
        WITH g^ DO
                i:=0;
                WHILE (i<anzahl) AND (NOT gleich(inhalt[i],x)) DO
                        i:=i+1
                END;
                IF i=anzahl
                        THEN error("in index: Element nicht vorhanden")
                        ELSE RETURN i
                END;
        END;
END index;


PROCEDURE edge(g:Graph; i,j:CARDINAL): BOOLEAN;              (* liefert TRUE, falls   *)
                                                             (* Kante (i,j) existiert *)
VAR gefunden : BOOLEAN;
BEGIN
        IF rangeok(g,i) AND rangeok(g,j)
            THEN
                reset(g^.nachbarn[i]);
                gefunden := FALSE;
                WHILE (NOT gefunden) AND (NOT endpos(g^.nachbarn[i])) DO
                        IF j = elem(g^.nachbarn[i])
                                THEN gefunden := TRUE
                                ELSE advance(g^.nachbarn[i])
                        END
                END;
                RETURN gefunden
            ELSE error("in edge: ungueltiger Index")
        END;
END edge;
```

```
PROCEDURE resetneighbors(g: Graph;              (* bringt die Nachbarn von Knoten i     *)
                         i: CARDINAL);          (* in Ausgangsstellung                  *)
BEGIN
        IF rangeok(g,i)
                THEN reset(g^.nachbarn[i])
                ELSE error("in resetneighbors: ungueltiger Index")
        END
END resetneighbors;



PROCEDURE moreneighbors (g: Graph;              (* liefert TRUE, falls es weitere       *)
                         i: CARDINAL): BOOLEAN; (* Nachbarn des Knotens i gibt          *)
BEGIN
        IF rangeok(g,i)
                THEN RETURN NOT endpos(g^.nachbarn[i])
                ELSE error("in moreneighbors: ungueltiger Index")
        END
END moreneighbors;



PROCEDURE nextneighbor  (g: Graph;              (* liefert Index des naechsten          *)
                        i: CARDINAL): CARDINAL; (* Nachbarn von Knoten i                *)
VAR j : CARDINAL;
BEGIN
        IF rangeok(g,i) THEN
                                IF NOT endpos(g^.nachbarn[i])
                                THEN
                                        j := elem(g^.nachbarn[i]);
                                        advance(g^.nachbarn[i]);
                                        RETURN j
                                ELSE error("in nextneighbor: keine weiteren Nachbarn")
                                END
                        ELSE error("in nextneighbor: ungueltiger Index")
        END
END nextneighbor;
```

```
PROCEDURE readgraph(g: Graph);                          (* liest einen Graphen ein      *)
VAR i, k : CARDINAL;
VAR x    : Element;
BEGIN
        WITH g^ DO
                anzahl := 0; WriteString("Bitte Knoten benennen: ");

                createelement(x);
                WHILE readelement(x) AND (anzahl < maxknoten) DO
                        inhalt[anzahl] := x;
                        anzahl := anzahl+1;
                        createelement(x);
                END;

                FOR i:= 0 TO g^.anzahl-1 DO
                        reset(nachbarn[i]);
                        WriteString("Bitte Nachbarn von ");
                        writeelement(inhalt[i]); WriteString(" : ");

                        WHILE readelement(x) DO
                                insert(nachbarn[i],index(g,x));
                                advance(nachbarn[i]);
                        END;
                END;
        END;
END readgraph;


PROCEDURE writegraph(g: Graph);                                 (* gibt einen Graphen aus *)
VAR i: CARDINAL;
BEGIN
        WriteString("Knoten Nachbarn"); WriteLn;
        WITH g^ DO
                FOR i:= 0 TO anzahl-1 DO
                        writeelement(inhalt[i]); WriteString("  : ");
                        reset(nachbarn[i]);
                        WHILE NOT endpos(nachbarn[i]) DO
                                writeelement(inhalt[elem(nachbarn[i])]);
                                advance(nachbarn[i]);
                        END;
                        WriteLn;
                END;
        END;
END writegraph;

END graph.
```

11.2. Graphalgorithmen

```
(****************************************************************************)
(*      Datum:  07.02.94  14:15 Uhr                                        *)
(*      Autor:  Oliver Vornberger                                          *)
(*      Modul:  graphalgo.d                                                *)
(*              Schnittstelle fuer einige Graphalgorithmen                 *)
(*                                                                          *)
(****************************************************************************)

DEFINITION MODULE graphalgo;

FROM graph IMPORT Graph;

PROCEDURE visitrekursiv (g:Graph);            (* durchlaeuft Graph g in Tiefensuche,  *)
                                              (* gibt die Knoten durchnumeriert aus   *)

PROCEDURE visititerativ (g:Graph);            (* durchlaeuft Graph g in Tiefensuche,  *)
                                              (* gibt die Knoten durchnumeriert aus   *)

PROCEDURE toposort  (g:Graph);                (* gibt die Knoten des Graphen g        *)
                                              (* in topologischer Reihenfolge aus,    *)
                                              (* soweit dies moeglich ist             *)
                                              (* topologisch sortiert heisst:         *)
                                              (* Knoten j kommt spaeter als Knoten i  *)
                                              (* wenn es Kante (i,j) gibt             *)

PROCEDURE hamilton(g:Graph);                  (* berechnet, sofern existiert, einen   *)
                                              (* Hamiltonkreis fuer G                 *)
                                              (* In einem Hamiltonkreis wird jeder    *)
                                              (* Knoten genau einmal besucht          *)

END graphalgo.
```

```
(*********************************************************************************)
(*      Datum:  30.01.96  14:15 Uhr                                              *)
(*      Autor:  Oliver Vornberger                                                *)
(*      Modul:  graphalgo.m2                                                     *)
(*              implementiert die Graphalgorithmen fuer                          *)
(*              Tiefensuche, topologisches, Sortieren, Hamilton-Kreis            *)
(*                                                                               *)
(*              benoetigt wird ein Keller ueber den CARDINALSs:                  *)
(*              DEFINITION MODULE keller;                                        *)
(*              TYPE Kellerelement = CARDINAL;                                   *)
(*                                                                               *)
(*              und eine Schlange ueber den CARDINALs:                           *)
(*              DEFINITION MODULE schlange;                                      *)
(*              TYPE Schlangenelement = CARDINAL;                                *)
(*                                                                               *)
(*********************************************************************************)

IMPLEMENTATION MODULE graphalgo;

FROM element    IMPORT Element, gleich, readelement, writeelement;
FROM graph      IMPORT Graph, maxknoten, number, knoten, edge,
                       resetneighbors, moreneighbors, nextneighbor;
FROM schlange   IMPORT Schlange, createqueue, front, emptyqueue, enq, deq;
FROM keller     IMPORT Keller, createstack, push, pop, top, emptystack;
FROM InOut      IMPORT ReadCard, Done, WriteString, WriteLn, WriteInt, WriteCard;
FROM Storage    IMPORT ALLOCATE;
```

```
PROCEDURE visitrekursiv (g:Graph);              (* durchlaeuft in Tiefensuche den Graph *)

VAR besucht:    ARRAY[0..maxknoten] OF BOOLEAN;
VAR k, id  :    CARDINAL;


PROCEDURE visit (g:Graph; k: CARDINAL);         (* besucht alle Knoten eines Graphen    *)
VAR x : CARDINAL;                               (* beginnend beim Knoten k              *)
BEGIN                                           (* rekursive Version                    *)
        id := id +1;                            (* naechste laufende Nummer             *)
        writeelement(knoten(g,k));
        WriteString(" bekommt Nr. ");
        WriteCard(id,4); WriteLn;
        besucht[k] := TRUE;                     (* markiere Knoten als besucht          *)

        resetneighbors(g,k);                    (* durchlaufe alle seine Nachbarn       *)
        WHILE moreneighbors(g,k) DO

                x := nextneighbor(g,k);

                IF NOT besucht[x]               (* falls Nachbar nicht schon besucht    *)
                        THEN visit(g,x)         (* dann besuche ihn                     *)
                END;
        END;
END visit;


BEGIN
        id := 0;                                (* Initialisierung:                     *)
        FOR k:=0 TO number(g)-1 DO              (* alle gelten als nicht besucht        *)
                besucht[k] := FALSE;
        END;

        FOR k:=0 TO number(g)-1 DO              (* fuer jede Zusammenhangskomponente:   *)
                IF NOT besucht[k]               (* besuche alle, die von k aus          *)
                        THEN visit(g,k)         (* erreichbar sind *)
                END
        END;

        WriteString("Rekursive Tiefensuche beendet "); WriteLn;

END visitrekursiv;
```

```
PROCEDURE visititerativ (g:Graph);              (* durchlaeuft in Tiefensuche den Graph *)

VAR gesichtet:  ARRAY[0..maxknoten] OF BOOLEAN;
VAR k, id  :    CARDINAL;
VAR s      :    Keller;                         (* stack ueber CARDINALs               *)


PROCEDURE visit (g:Graph; k: CARDINAL);         (* besucht alle Knoten eines Graphen    *)
VAR x : CARDINAL;                               (* beginnend beim Knoten k              *)
BEGIN                                           (* iterative Version                    *)
        push(s,k);                              (* Einstiegsknoten kommt auf Keller     *)
        gesichtet[k]:=TRUE;                     (* und gilt als gesichtet               *)
        REPEAT
                k := top(s); pop(s);            (* entferne Top—Element                 *)
                id := id +1;                    (* naechste laufende Nummer             *)
                writeelement(knoten(g,k));      (* wird an das Top—Element vergeben     *)
                WriteString(" bekommt Nr. ");
                WriteCard(id,4); WriteLn;

                resetneighbors(g,k);            (* durchlaufe alle seine Nachbarn       *)
                WHILE moreneighbors(g,k) DO

                    x := nextneighbor(g,k);


                    IF NOT gesichtet[x]         (* falls Nachbar noch nicht gesichtet   *)
                    THEN
                        gesichtet[x]:=TRUE;     (* markiere ihn als gesichtet           *)
                        push(s,x)               (* und speichere ihn im Keller          *)
                    END;
                END;
        UNTIL emptystack(s);
END visit;

BEGIN
        createstack(s);                         (* lege leeren Keller an                *)
        id := 0;                                (* Initialisierung:                     *)
        FOR k:=0 TO number(g)—1 DO              (* alle gelten als nicht gesichtet      *)
                gesichtet[k] := FALSE;
        END;

        FOR k:=0 TO number(g)—1 DO              (* fuer jede Zusammenhangskomponente:   *)
                IF NOT gesichtet[k]             (* besuche alle, die von k aus          *)
                        THEN visit(g,k)         (* erreichbar sind                      *)
                END
        END;

        WriteString("Iterative Tiefensuche beendet "); WriteLn;

END visititerativ;
```

```
PROCEDURE toposort (g:Graph);                    (* gibt die Knoten des Graphen g     *)
                                                 (* in topologischer Reihenfolge aus, *)
                                                 (* soweit dies moeglich ist          *)
VAR indegree : ARRAY[0..maxknoten-1] OF CARDINAL;  (* Eingangsgrad fuer jeden Knoten    *)
VAR i,j,id   : CARDINAL;
VAR s        : Schlange;
BEGIN

        FOR i:=0 TO number(g)-1 DO indegree[i] :=0; END;
        FOR i:=0 TO number(g)-1 DO
                resetneighbors(g,i);
                WHILE moreneighbors(g,i) DO
                        j := nextneighbor(g,i);
                        indegree[j] := indegree[j] + 1  (* nun enthaelt indegree[i]      *)
                END;                                    (* die Anzahl der Vorgaenger     *)
        END;                                            (* des Knotens mit Index i       *)

        createqueue(s);
        FOR i:=0 TO number(g)-1 DO
                IF indegree[i]=0 THEN enq(s,i) END;     (* nun enthaelt die Schlange     *)
        END;                                            (* alle Knoten ohne Vorgaenger   *)

        WriteString("Topologische Sortierung: ");
        WriteLn;
        id := 0;
        WHILE NOT emptyqueue(s) DO                      (* solange es Knoten ohne        *)
                                                        (* Vorgaenger gibt               *)
                i := front(s); deq(s);                  (* an das Frontelement           *)
                id := id +1;                            (* die naechste laufende Nummer  *)
                writeelement(knoten(g,i));              (* vergeben                      *)
                WriteString(" bekommt Nr. ");
                WriteCard(id,4); WriteLn;

                resetneighbors(g,i);
                WHILE moreneighbors(g,i) DO
                        j := nextneighbor(g,i);         (* Eingangsgrad der Nachbarn     *)
                        indegree[j] := indegree[j] - 1; (* verringern                    *)
                        IF indegree[j] = 0              (* Knoten ohne Vorgaenger in     *)
                                THEN enq(s,j);          (* die Schlange tun              *)
                        END;
                END
        END;

        IF id = number(g)  THEN WriteString("Graph ist azyklisch")
                           ELSE WriteString("Graph enthaelt Kreis");
        END;
        WriteLn;

END toposort;
```

```
PROCEDURE zeigetour(g: Graph; k:Keller);             (* Hilfsprozedur zum Anzeigen der *)
VAR k1 : Keller;                                      (* im Keller gespeicherten Tour   *)
BEGIN                                                 (* benoetigt einen Hilfskeller    *)
        createstack(k1);
        WHILE NOT emptystack(k) DO
                push(k1,top(k));
                pop(k)
        END;

        WriteString("Neue Tour: ");
        WHILE NOT emptystack(k1) DO
                push(k,top(k1));
                writeelement(knoten(g,top(k1)));
                pop(k1)
        END;
        WriteLn;
END zeigetour;
```

```
PROCEDURE hamilton (g:Graph);                   (* bestimmt Hamiltonkreis       *)
VAR anzahl, i, j : CARDINAL;
VAR k            : Keller;
VAR besucht      : ARRAY[0..maxknoten-1] OF BOOLEAN;
VAR gefunden     : BOOLEAN;
BEGIN
        createstack(k);                         (* lege Keller fuer Knoten an   *)
        push(k,0);                              (* Knoten 0 ist Anfang der Tour *)
        anzahl := 1;                            (* Anzahl der Knoten im Keller  *)
        FOR i:=1 TO number(g) DO besucht[i]:=FALSE;END; (* alle Knoten sind unbesucht   *)
        besucht[0] := TRUE;                     (* bis auf Knoten 0             *)
        resetneighbors(g,0);
        REPEAT
                i := top(k);
                IF (anzahl=number(g)) AND edge(g,i,0)   (* gib neue Tour aus            *)
                        THEN zeigetour(g,k)
                END;

                gefunden := FALSE;              (* suche unbesuchten Nachbarn   *)
                WHILE (NOT gefunden) AND moreneighbors(g,i) DO
                        j := nextneighbor(g,i);
                        IF NOT besucht[j]
                                THEN gefunden := TRUE
                        END
                END;

                IF gefunden                     (* es geht weiter               *)
                THEN                            (* markiere den neuen Nachfolger*)
                        besucht[j] := TRUE;     (* als besucht, lege ihn auf    *)
                        push(k,j);              (* den Stack und initialisiere  *)
                        resetneighbors(g,j);    (* seine Nachbarn               *)
                        anzahl := anzahl + 1;   (* erhoehe die Anzahl im Stack  *)
                ELSE                            (* es geht nicht weiter         *)
                        anzahl := anzahl-1;     (* mache Schritt rueckgaengig   *)
                        besucht[top(k)] := FALSE; (* Knoten gilt als unbesucht,   *)
                        pop(k);                 (* wird vom Keller genommen     *)
                END

        UNTIL emptystack(k);

END hamilton;


END graphalgo.
```

```
(****************************************************************************************)
(*      Datum:  08.02.94  14:15 Uhr                                                     *)
(*      Autor:  Oliver Vornberger                                                       *)
(*      Modul:  graphtest.m2                                                            *)
(*              ruft die Graphalgorithmen auf                                           *)
(****************************************************************************************)

MODULE graphtest;

FROM graph      IMPORT Graph, creategraph, readgraph, writegraph;
FROM graphalgo  IMPORT visitrekursiv, visititerativ, toposort, hamilton;

VAR g:  Graph;

BEGIN

        creategraph(g);
        readgraph(g);
        writegraph(g);
        visitrekursiv(g);
        visititerativ(g);
        toposort(g);
        hamilton(g);

END graphtest.
```