1.3.2 Daten-Konstruktoren

Sind Datentypen gegeben, so kann man hieraus neue Datentypen erzeugen. Man orientiert sich hierbei an den mathematischen Operatoren auf Mengen. Solche Operatoren sind:

- Bildung von Unterbereichen (Intervalle, Projektion, "range"),
- kartesisches Produkt ("record"),
- kartesisches Produkt mit sich (Vektoren, Matrizen, "array"),
- disjunkte Vereinigung ("varianter record"),
- Folgen-Bildung (freies Monoid, "string"-Bildung),
- Menge der Teilmengen (Potenzmenge, "set of"),
- Menge von Abbildungen ("function").

Grundsätzliches Vorgehen:

Zu jedem Konstruktor K, der auf k Datentypen T_i wirkt,

$$K\,(T_1, T_2, ..., T_k)$$

wird angegeben:

- Wie sieht die Wertemenge aus (bezogen auf die Wertemengen der Datentypen T_i)?
- Wie kann man auf die einzelnen Komponenten T_i zugreifen?
- Welche Operationen der Datentypen T_i werden übernommen und in welcher Form?
- Werden neue Operationen hierdurch eingeführt?

Ein Beispiel, das Sie bereits kennen:

Der Konstruktor array wirkt auf zwei Datentypen:

array (T_1, T_2)

in Ada geschrieben als

ARRAY (T_1) OF T_2

Wenn T_1 die Wertemenge D mit n Elementen und T_2 die Wertemenge M besitzen, so ist M^D die zu array (T_1, T_2) gehörende Wertemenge (bis auf Isomorphie).

Für ein beliebiges Element $a \in D$ greift man auf den zu a gehörenden Funktionswert in M zu mittels

[a] (in Ada: (a)).

Ein Element aus array (T_1, T_2) ist somit eine als Tabelle dargestellte Abbildung von D nach M.

Ein Beispiel (Fortsetzung)

Oft ist die Wertemenge des Indexdatentyps T_1 eine Menge von Zahlen, z.B. die Menge $\{1, 2, ..., n\}$. In diesem Fall gehört zu

array (T_1, T_2)

die Wertemenge M^n (also ein Vektor über M). Man greift auf die einzelnen Komponenten zu mittels

[i] (in Ada: (i))

für eine Zahl $1 \leq i \leq n$.

Außer dieser Zugriffsoperation werden durch den Konstruktor array keine neuen Operationen eingeführt.

1.3.2.1 Unterbereiche (vgl. 1.1.2.20)

Unterbereich oder Intervall $[a .. b] = \{x \in M \mid a \leq x \leq b\}$.
Speziell: Intervallbildung auf diskreten Mengen:
Wenn $M = \{..., m_1, m_2, m_3, ..., m_q, ...\}$ eine abzählbare Menge ist mit $m_1 < m_2 < m_3 < ... < m_n$, dann ist ein Intervall eine Teilmenge $[m_i, m_k] = \{m_i, m_{i+1}, m_{i+2}, ..., m_k\} \subseteq M$ mit $1 \leq i \leq k \leq n$.

Definition eines Datentyps "intervall" als Unterbereich des Datentyps T:

type intervall = T $[m_i .. m_k]$;

Alle Operationen von T gelten auch für "intervall", sofern der Unterbereich hierbei (und bei Zwischenrechnungen) nicht verlassen wird (letzteres kann man auch anders festlegen; z.B. beachtet Ada die Zwischenrechnungen nicht).

type natural = integer [0 .. flex];

Mit Hilfe des Symbols "flex" (=flexibel) lässt man die jeweilige Grenze offen. Dies kann man auch nach unten verwenden:

type kleiner80 = natural [flex..79];

type integer2 = integer [flex..flex];
Dies definiert einen Datentyp, der genau wie die ganzen Zahlen aufgebaut ist, aber zunächst nicht "kompatibel" ist. D.h. für var X: integer; Y: integer2; ... ist die Zuweisung X:=Y verboten.
(integer könnte für die Maßeinheit "Meter" und integer2 für "Fuß" stehen; man darf nicht einfach ohne Anpassungen zwischen verschiedenen Maßsystemen umschalten.)

Die Unterbereichsbildung in Ada:

Unterbereiche werden in Ada als "subtype" eines anderen Datentyps bezeichnet. Die allgemeine Form lautet:

subtype <Name> is <Datentyp> [<constraint>]

Die Beschränkung <constraint> gibt den Bereich ("range") von unterer bis oberer Grenze an:
<constraint> ::= range <Ausdruck> .. <Ausdruck>

Alle Operationen des Typs <Datentyp> sind auch für den Unterbereichsdatentyp gültig.

Die Beschränkung darf fehlen; dann hat der Unterdatentyp die gleiche Wertemenge wie der <Datentyp>.

Basistyp

In der Definition

subtype U is T <constraint>

ist U der Unterdatentyp von T und T der Oberdatentyp von U. Der Datentyp, von dem U letztlich abgeleitet wurde, heißt Basisdatentyp zu U. Es sei T ein elementarer Datentyp. Im Falle

subtype U is T <constraint>
subtype V is U <constraint>
subtype W is V <constraint>

ist W Unterdatentyp von V, von U und von T. U ist Obertyp von V und von W. Der Basisdatentyp von U, V und W ist T.

Operationen

"subtype" bezieht sich ausschließlich auf die Wertebereiche. Alle Operationen des Obertyps werden vom Unterdatentyp übernommen.

In Ada wird jede Variable eines Unterdatentyps stets auch als Variable des Basistyps aufgefasst, so dass man also rechnen kann, als ob man im Obertyp wäre; erst bei der Zuweisung des Ergebnisses wird geprüft, ob das Ergebnis die Beschränkung erfüllt.

Beispiele

subtype Monatstage is integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

Wenn man nun zu einem Tag vom Typ Monatstage den Tag der folgenden Woche (plus 7) berechnen will, so könnte man schreiben:

```
X, Z: Monatstage;   .....
Z := X + 7;
if Z > 31 then Z := Z-31 end if;
```

Dies führt natürlich zu einem Fehler, falls X mindestens den Wert 25 besaß, da dann Z den Unterbereich verlassen würde. Dagegen führt `if X > 24 then Z := X + 7 - 31; end if;` nicht zu einem Fehler, auch wenn zwischenzeitlich der Bereich verlassen wurde, da Ada den Ausdruck X+7-31 im Obertyp integer berechnet.

Zuweisungen an Variablen der Unterdatentypen sind immer erlaubt, wenn der zuzuweisende Wert im Unterbereich liegt:

subtype Monatstage is integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

X, Y, Z: Monatstage; A: MinMonatstage; H, J: integer; .....
H := 50; X := 30; Z := 20;

Erlaubt sind dann (beachte: (H*Z) mod 30 +1 ist ein Ausdruck in den ganzen Zahlen, der dort berechnet wird, erst das Ergebnis wird der Variablen Y zugewiesen):

A := Z; Y := H-20; Y := (H*Z) mod 30 +1; J := X; A := X-Z;

Verboten sind dann (d.h., zu Bereichsüberschreitungen führen):

A := X; Z := H; A := Z*Z;

Wichtige vordefinierte Unterdatentypen in Ada sind:

subtype natural is integer range 0..integer'Last;

subtype positive is integer range 1..integer'Last;

Unterbereiche kann man auch für den Typ float einführen, sofern die Grenzen Ausdrücke sind, deren Ergebnisse reellwertig sind:

subtype einheitsintervall is float range 0.0 .. 1.0;

1.3.2.2 Records (Datensätze)

Oft muss man die Elemente verschiedener Mengen zu einem Paar, einem Tripel oder einem n-Tupel zusammenfassen. Gegeben seien die Mengen M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören mögen. Dann kann man nach folgendem Schema hieraus den Datentyp T konstruieren, dessen Wertemenge die Menge $M = M_1 \times M_2 \times ... \times M_n$ ist (dies ist zugleich die Ada-Darstellung):

type T is record S_1: T_1; S_2: T_2; ...; S_n: T_n; end record;

Hierbei sind S_1, S_2, ..., S_n Namen, die so genannten "Selektoren", mit deren Hilfe man auf die einzelnen Komponenten des Datentyps T zugreifen kann. Man sagt, man "selektiert" die i-te Komponente, indem man S_i durch einen Punkt getrennt hinter den Namen der Variable vom Typ T hängt (*dot-notation*).

Beispiele

```
type Monatsname is (Januar, Februar, März, April, Mai, Juni,
        Juli, August, September, Oktober, November, Dezember);

type Datum is record
        Jahr: integer;
        Monat: Monatsname;
        Tag: 1..31;
end record;

heute, ende: Datum;

heute.Jahr := 2002; heute.Monat := November; heute.Tag := 28;

if heute.Jahr = 2002 then
        ende.Jahr := 2003; ende.Monat := Juli; ende.Tag := 24;
else ende := heute; end if;
```

Beispiele (Fortsetzung)

Komplexe Zahlen:

```
type komplex is record
        Realteil: float;
        Imaginärteil: float;
end record;
```

Rationale Zahlen:

```
type rational is record
        Zähler: integer;
        Nenner: positive;
end record;
P, R: rational; gleich, eins, wurzel2: Boolean; ...
gleich := P.Zähler * R.Nenner = R.Zähler * P.Nenner;
eins := P.Zähler = P.Nenner;
wurzel2 := P.Zähler * P.Zähler = 2 * P.Nenner * P.Nenner;
```


Beispiele (Fortsetzung)

Records spielen im täglichen Leben eine zentrale Rolle, weil sie die programmiersprachliche Beschreibung von Formularen sind. Wer ein Formular ausfüllt, füllt einen record-Datentyp.

```
type hotelformular is record
        Ankunftstag, Abreisetag: Datum;
        Name: array (1..30) of character;
        Geburtsjahr: 1880..2002;
        Wohnort: array (1..30) of character;
        PLZ: array (1..5) of '0'..'9';
        Strasse: array (1..30) of character;
        Hausnummer: positive;
        allein: Boolean;
        männlich: Boolean;
        Raucherzimmer: Boolean;
end record;
```

1.3.2.3 Felder (vgl. 1.1.2.21)

Für Elemente der Wertemengen M^r benötigt man r Variablen der gleichen Art, auf die mit einem Index zugegriffen werden kann. Diese fasst man unter einem Namen zu einem "Feld" (oder Vektor) zusammen. Man muss den Indexdatentyp und den Wertedatentyp angeben: array <Indexdatentyp> of <Datentyp>.

Hierbei darf der <Datentyp> der Komponenten erneut eine Felddeklaration sein:

array <Datentyp des 1.Index> of
array <Datentyp des 2.Index> of <Datentyp>

Dies kürzt man meist ab, z.B. durch

array <Datentyp des 1.Index, Datentyp des 2.Index> of <Datentyp>

und nennt dies ein zwei-dimensionales Feld. Wiederholt man dies, so lassen sich n-dimensionale Felder deklarieren, n ≥ 1.

Beispiel aus 1.1.2.21: (In Ada ist die Darstellung etwas anders!)

type Wochentage = (Mo, Di, Mi, Do, Fr, Sa, So)

Hierzu bilden wir den Unterbereich:

type Arbeitstage = Wochentage [Mo .. Fr]

und dann folgendes Feld aus fünf Elementen

type Arbeitsbeginn = array Arbeitstage of [0 .. 23]

Variablen dieses Typs werden deklariert durch

var X: Arbeitsbeginn;

Auf ihre Komponenten wird mittels X[J] zugegriffen. Dabei durchläuft J den Wertebereich des Datentyps Arbeitstage, d.h. J∈ {Mo, Di, Mi, Do, Fr}.

Ein weiteres Beispiel "skalarprodukt1" finden Sie unter 1.1.2.21.

Unendliche Indexbereiche:

In der Programmierung sind Wertebereiche der Form M^{∞} ungebräuchlich, auch wenn sie in der Mathematik eine bedeutende Rolle spielen. Will man solche Wertebereiche jedoch verwenden, um eine Folge von Werten darzustellen, so sollte man die Folgenbildung (siehe später) benutzen.

Man könnte aber auch einen Datentyp

array [1..flex] of ...

einführen, wobei das Symbol "flex" (=flexibel) bedeutet, dass die obere Grenze offen bleibt und sich während der Berechnungen ständig ändern kann.

Diese Möglichkeit gibt es in manchen Sprachen, z.B. in Algol 68 und in gewisser Form auch Ada.

Felder in Ada:

Grundsätzliche Form zur Einführung eines Feld-Datentyps:

type <name> is array (<Indexdatentyp>) of <Datentyp>

Beispiele:

type Hvektor is array (integer range 1..100) of float;

Da durch 1..100 bereits der Datentyp integer festgelegt ist, darf man in Ada hierfür auch kürzer schreiben:

type Hvektor is array (1..100) of float;

type Hmatrix is array (1..50) of Hvektor;

type Bundesligatabelle is array (1..18) of fussballverein;

type codierung is array (Character) of Character;

Die iterierte array-Bildung kürzt man ebenfalls ab, indem alle Indexbereiche in eine Definition geschrieben werden:

Statt
type Hvektor is array (1..100) of float;
type Hmatrix is array (1..50) of Hvektor;
type Hvolume is array (1..80) of Hmatrix;

schreibt man also kurz:
type Hvolume is array (1..100,1..50,1..80) of float;

Die Zahl der hierbei verwendeten Indexbereiche heißt die *Dimension* des Feldes. Hvolume ist also ein 3-dimensionales Feld.

Statt Konstanten dürfen in den Indexbereichen auch Ausdrücke verwendet werden, sofern jeder Ausdruck in dem Augenblick, in dem die Deklaration erreicht wird, auch tatsächlich ausgerechnet werden kann.

Beispiele (wobei f und g Funktionen vom Ergebnistyp integer sein sollen):
type Hvektor is array (1..N, x..I*J) of Boolean;
type ausschnitt is array (f(unten)..g(oben)) of integer;
type sonstiges is array ((oben-unten) div 2..f(g(unten*oben))) of float;

Ein Feld heißt *statisch,* wenn zur Übersetzungszeit (also unabhängig von Eingabewerten) alle Feldgrenzen bekannt sind. Wird mindestens eine Feldgrenze erst zur Laufzeit des Programms berechnet, so heißt das Feld *dynamisch.*

1.3.2.3.a: Unspezifizierte Feldgrenzen sind zulässig. Man trägt dann nur den Datentyp des Index ein und fügt ein

range <> ("<>" sprich "box")

hinzu. Solche unspezifizierten array-Deklarationen *muss* man bei ihrer Verwendung spezifizieren, z.B. bei der Deklaration von Variablen und beim konkreten Parameteraufruf.

Beispiele:

```
type text is array (natural range <>) of Character;
type raster is array (integer range <>, integer range <>) of Boolean;
type ganzzahlvektor is array (integer range <>) of integer;
```

Bei der Deklaration von Variablen gibt man dann die konkreten Grenzen statisch oder dynamisch vor, z.B.:
X: ganzzahlvektor (-10..10) oder Y: ganzzahlvektor (I..J)

In Ada sind weitere Operationen mit arrays verbunden (in Ada werden sie als Attribute bezeichnet):

Die Konstante "Range(i)" gibt zu jeder Feld-Variablen den Indexdatentyp ihrer i-ten Dimension an. Analog bezeichnet "Length(i)" die Anzahl der Elemente des Indexdatentyps in der i-ten Dimension. Mittels First(i) und Last(i) erhält man das erste bzw. letzte Element des Indexdatentyps der i-ten Dimension. Hat man nur eine Dimension, so darf man "(1)" weglassen.

Beispiel:

```
type Codes is (character range 'A'..'Z', 1..32) of
                                    character range 'B'..'z';
W: Codes;
```

Dann gilt:

```
W'Range(2) = 1..32,     W'Length(1) = 26,
W'First(1) = 'A' ,      W'Last(2) = 32
```

Weiterhin darf man Feldvariablen, die *vom gleichen Datentyp* sind, einander zuweisen.

type vektor is array (1..100) of character;

X, Y: vektor;

....

X := Y;

Bedeutung dieser Zuweisung: Der gesamte Inhalt von Y wird komponentenweise in die Variable X kopiert.

1.3.2.3.b: In Ada gibt es eine klare restriktive Regel, wann zwei Variablen A und B vom gleichen Datentyp sind: Sie müssen den gleichen "benannten" Datentyp haben, d.h., sie besitzen entweder den gleichen vordefinierten Datentyp (Boolean, character, integer, natural, positive, float usw.) oder es gibt eine Typ- oder Untertyp-Definition mit einem Namen und beide Variablen sind mit diesem Namen deklariert worden.

"Benannt" bedeutet, dass dem Datentyp ein Name per Datentypdefinition zugewiesen ist. In Ada kann man nämlich Feld-Variablen auch unbenannt deklarieren:

X: array (1..100) of character;

Solch eine Deklaration wird stets als neue, noch nicht vorhandene Feld-Datentypdefinition aufgefasst (s.u.).

Gleichen Datentyp haben z.B.:

```
type zehnziffern is array(1..10) of 0..9;
A, B: integer; K, L: positive;
X, Y: zehnziffern;
```

Dagegen haben *nicht* den gleichen Datentyp in Ada:

```
D: zehnziffern; E: array (1..10) of 0..9;
F, G: array(1..10) of 0..9;
H, I: 0..50;
```

denn Ada fasst H, I: 0..50 auf als die Deklarationsfolge

```
H: 0..50;
I: 0..50;
```

und diese beiden Datentypen sind verschieden, da sie keinen vom Benutzer vergebenen Namen besitzen.

Weiteres Beispiel:

```
type Codes is (character range 'A'..'Z', 1..32) of
                    character range 'B'..'z';
V, W: Codes;  R, S: character range W'Range(1);
begin
for I in W'Range(1) loop
    for K in W'Range(2) loop
        W(I,K) := val(pos(I) + K);  end loop;  end loop;
V := W;      -- dies bewirkt, dass das ganze Feld W nach V kopiert wird
R := W('C',17);   -- es wird geprüft, ob das Ergebnis im Unterdatentyp liegt
S := W(R,32);     -- dies wird daher zu einem Fehlerabbruch führen
...
end;
```

1.3.2.3.c: Realisierung eines n-dimensionalen arrays durch ein eindimensionales array:

Wir wissen: Der Speicher eines gängigen Computers (mit 32-Bit-Wörtern als Speicherzellen) ist vom Datentyp

array $(0..2^m-1)$ of $0..2^{32}-1$;

wobei m heute eine Zahl in der Größenordnung 26 bis 30 (Hauptspeicher) bzw. 35 bis 45 (Hintergrundspeicher) ist.

Ein Compiler muss daher letztlich jedes n-dimensionale Feld auf ein eindimensionales Feld abbilden. Wie lautet diese Abbildung?

Malen Sie es sich auf und durchlaufen Sie ein 2-dimensionales Feld zeilenweise. So erhalten Sie die übliche Einbettung:

Wenn das n-dimensionale Feld ($n \geq 2$) von der Form

array $(u_1..o_1, u_2..o_2, ..., u_n..o_n)$ of ...

und das eindimensionale Feld von der Form

array (unten..oben) of ...

und alle Indextypen $(u_j..o_j)$ und (unten..oben) ganzzahlige Intervalle sind, dann kann man den Index $(i_1, i_2, ..., i_n)$ abbilden auf

$$f(i_1, i_2, ..., i_n) = \text{unten} + \sum_{k=1}^{n} d_k \cdot (i_k - u_k) \quad \text{mit} \quad d_k = \prod_{j=k+1}^{n} (o_j - u_j + 1)$$

Nebenbedingung: oben-unten+1 = d_0. (beachte: $d_n = 1$)

f heißt Speicherabbildungsfunktion.

Hinweis 1: Die oben angegebene Funktion f speichert das mehrdimensionale Feld "zeilenweise".

Will man spaltenweise speichern (FORTRAN macht das so), dann muss man die Funktion leicht modifizieren. (Wie?)

Hinweis 2: Die Funktion f lässt sich auch schreiben als:

$$f(i_1, i_2, ..., i_n) = \text{unten} - \sum_{k=1}^{n} d_k \cdot u_k + \sum_{k=1}^{n} d_k \cdot i_k = u_{reduz} + \sum_{k=1}^{n} d_k \cdot i_k$$

mit der "reduzierten Anfangsadresse" $u_{reduz} = \text{unten} - \sum_{k=1}^{n} d_k \cdot u_k$

Ein Compiler speichert daher u_{reduz} und die d_k-Werte sowie eventuell alle o_j und u_j, um die Bereichsüberschreitung zu testen.

Beispiel 1.3.2.3.d: Intervallschachtelung (oder binäre Suche)

Ein Feld A: array (1..n) of integer sei gegeben. Das Feld sei sortiert, d.h.: $A(i) \leq A(i+1)$ für $i = 1, 2, ..., n-1$.

Aufgabe: Man schreibe einen Algorithmus, der zu einer Zahl s in möglichst kurzer Zeit feststellt, ob s im Feld A liegt oder nicht. Im Falle, dass s im Feld A enthalten ist, soll ein Index m mit $A(m) = s$ ausgegeben werden, anderenfalls sei $m = 0$.

Geht man das Feld von links nach rechts durch, so dauert es bis zu n Schritte, um das Ergebnis zu ermitteln.

Ein schnelleres Verfahren ist die Intervallschachtelung: Teste, ob s genau in der Mitte A(mitte) von A liegt, falls nein und es ist $A(\text{mitte}) < s$, suche rechts von der Mitte weiter, sonst links.

Programm 1 (in Ada): wir setzen hier nmax = 100_000.

```
procedure SEARCH1 is
m, links, rechts, s: Integer; gefunden: Boolean;
A: array (1..100_000) of integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl s ein"
     links:=1; rechts := n; gefunden := false;
     while (links <= rechts) and (not gefunden) loop
          m := (rechts+links) / 2;
          if A(m) = s then gefunden := true;
          elsif A(m) < s then links := m+1;
                         else rechts := m-1; end if;
     end loop;
     if not gefunden then m := 0; end if;
     ...;                    -- "drucke das Ergebnis aus"
end SEARCH1;
```


Wie lange dauert es, bis dieser Algorithmus spätestens endet?

Hierzu zählen wir die Wertzuweisungen und Bedingungen, die im ungünstigsten Fall ausgerechnet werden müssen.
In diesem Sinne dauert die Durchführung der 3 Anweisungen

```
links:=1; rechts := n; gefunden := false;
```

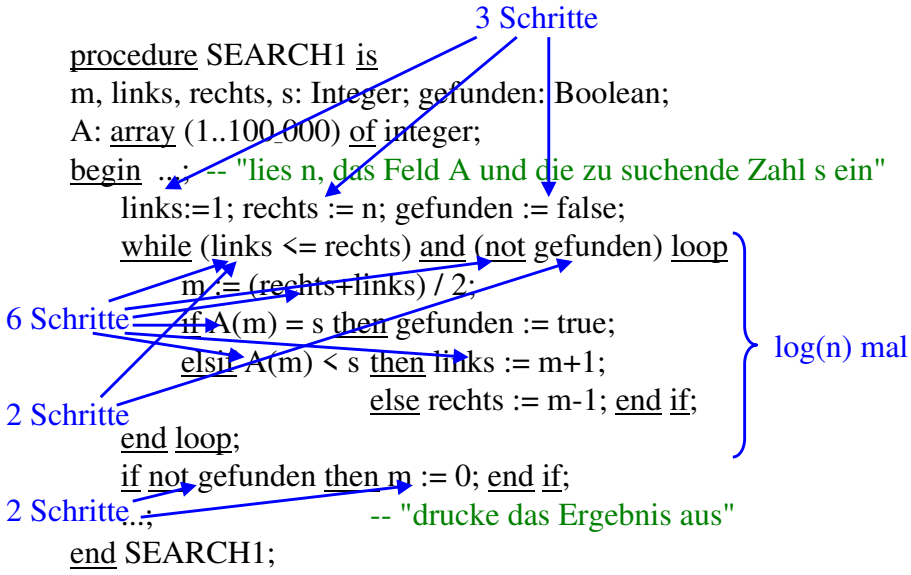
genau 3 Schritte.
In der Schleife werden maximal 6 Schritte benötigt, nämlich je einer für die Bedingungen

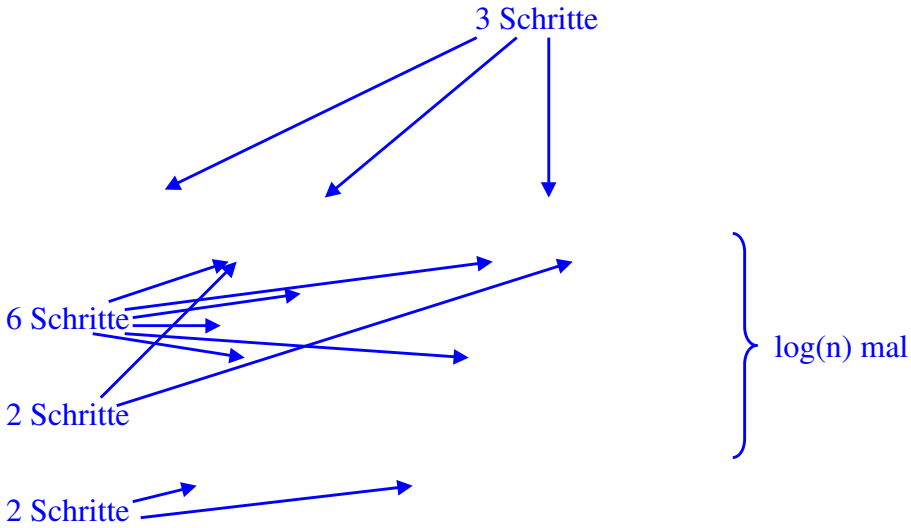
(links <= rechts), (not gefunden), A(m) = s und A(m) < s

und je einer für zum Beispiel die Wertzuweisungen

```
m := (rechts+links) / 2;  und  links := m+1;
```

Wie oft wird die Schleife durchlaufen? Das Intervall von links bis rechts halbiert sich mindestens in jedem Schritt, folglich muss nach log(n) Schleifendurchläufen Schluss sein.





Gesamt: 6·log(n) + 7 Schritte ∈ O(log(n))

Der ungünstigste oder schlechteste Fall (der "worst case") kann auch tatsächlich eintreten, wenn nämlich das gesuchte Element s nicht im Feld A enthalten ist. Wir sagen: Die *uniforme worst case Zeitkomplexität t(n)* des Programms SEARCH1 lautet

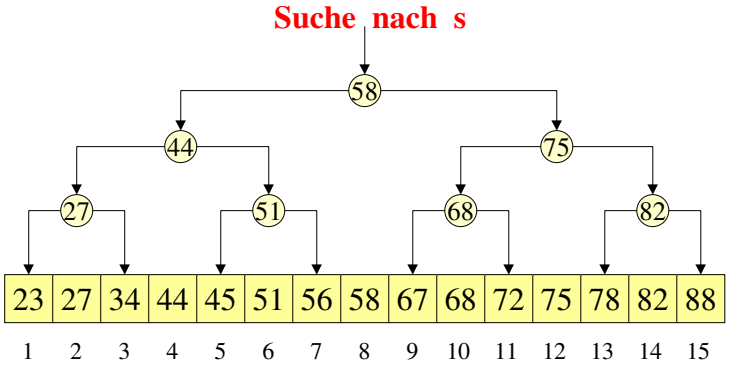
$t(n) = 6 \cdot \log(n) + 7.$

Beachten Sie: n ist hier die Anzahl der Elemente im Feld A. "Uniform" weil wir annehmen, alle Wertzuweisungen und Bedingungen würden die gleiche Zeit kosten.

Was ist der beste Fall? In diesem Fall wird s im ersten Durchgang der while-Schleife gefunden, d.h. nach 13 Schritten ist der Teil, der das Element s finden soll, beendet.

Mit wie vielen Schritten muss man im Mittel rechnen? Hierzu nehmen wir an, dass sich das gesuchte Element s tatsächlich im Feld A befindet (sonst kann man nur die obige worst case Abschätzung verwenden).

Wir skizzieren die Verhältnisse, wobei wir hier $n=15=2^4-1$ wählen:



In $2^3 = 8$ Fällen braucht man 4 Schleifendurchläufe,
in $2^2 = 4$ Fällen braucht man 3 Schleifendurchläufe,
in $2^1 = 2$ Fällen braucht man 2 Schleifendurchläufe,
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Allgemein gilt also, wenn $n = 2^k - 1$ ist:

In 2^{k-1} Fällen braucht man k Schleifendurchläufe,
in 2^{k-2} Fällen braucht man k-1 Schleifendurchläufe,
in 2^{k-3} Fällen braucht man k-2 Schleifendurchläufe,
....
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Daher braucht man im Mittel:

$\frac{1}{n}\,(k\cdot 2^{k-1} + (k-1)\cdot 2^{k-2} + (k-2)\cdot 2^{k-3} + \ldots + 2\cdot 2^1 + 1)$ Durchläufe.

Berechne also die Summe $\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j$

$$\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j = \frac{1}{2}\sum_{j=1}^{k} (j-1)\cdot 2^j + \frac{1}{2}\sum_{j=1}^{k} 2^j$$

$$= \sum_{j=1}^{k} (j-1)\cdot 2^{j-1} + \frac{1}{2}\,(2^{k+1} - 2)$$

$$= \sum_{j=0}^{k-1} j\cdot 2^j + (2^k - 1) = \sum_{j=1}^{k} j\cdot 2^j - k\cdot 2^k + (2^k - 1), \quad \text{d.h.:}$$

$\frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j = k\cdot 2^k - (2^k - 1)$. Folglich erhalten wir

$$\frac{1}{n}\,(k\cdot 2^{k-1} + (k-1)\cdot 2^{k-2} + \ldots + 2\cdot 2^1 + 1) = \frac{k\cdot 2^k - (2^k - 1)}{2^k - 1} \approx k - 1$$

Somit beträgt die *average case Zeitkomplexität* der Intervallschachtelung ziemlich genau $6\cdot\log(n) + 1$ Schritte, also nur einen Schleifendurchlauf weniger als im schlechtesten Fall.

Erkenntnis: Im Mittel spart man nur eine konstante Zahl an Operationen gegenüber dem schlechtesten Fall. Folglich lohnt sich zum Beispiel die Abfrage " A(m)=s " nicht; könnte man sie weglassen, so würde man log(n) viele Schritte sparen. Dies führt auf folgende bessere Version des Algorithmus für die Suche mittels Intervallschachtelung:

Man entscheide erst ganz am Ende, ob A(m) = s gewesen ist; hierzu muss man im Falle A(m) < s in dem rechten Teil des Feldes weitersuchen (links:=m+1), anderenfalls im linken Teil einschließlich des gerade betrachteten Feldes m (rechts:=m).

Programm 2 (in Ada): im Mittel besser als SEARCH1

```
procedure SEARCH2 is
m, links, rechts, s: Integer; gefunden: Boolean;
A: array (1..100_000) of integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl s ein"
     links:=1; rechts := n;
     while (links < rechts) loop
          m := (rechts+links) / 2;
          if A(m) < s then links := m+1;
                          else rechts := m; end if;
     end loop;
     gefunden := A(m) = s;
     if not gefunden then m := 0; end if;
     ...;                        -- "drucke das Ergebnis aus"
end SEARCH2;
```

Weisen Sie nun nach, dass für diese Version SEARCH2 gilt:

Die *uniforme worst case Zeitkomplexität* beträgt $5 + 4 \cdot \log(n)$; die *uniforme average case Zeitkomplexität* besitzt genau den gleichen Wert.

Hierbei ist n die Zahl der Elemente im array.

Diverse Suchverfahren auf Feldern betrachten wir im Teil 3 der Grundvorlesung(im SS 03).

1.3.2.4 Vereinigung

Gegeben seien die Mengen M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören. Dann kann man nach folgendem Schema hieraus den Datentyp T konstruieren, dessen Wertemenge die disjunkte Vereinigung dieser Mengen

$$M = M_1 \cup M_2 \cup ... \cup M_n$$

ist (dies ist zugleich die Ada-Darstellung):

```
type T is record
    case index is
        when 1 => S1: T1;
        when 2 => S2: T2;   ...
        when n => Sn: Tn;
    end case;
end record;
```

Statt des "index", der hier aus der Menge {1, 2, ..., n} ist, kann auch ein anderer Name und ein anderer Datentyp gewählt werden. Dieses auswählende Element heißt "Diskriminator".

Die Menge ist eine disjunkte Vereinigung, weil durch den Diskrimator "index" genau eine Menge ausgewählt wird, die Mengen also als verschieden angesehen werden, auch wenn verschiedene M_i gleiche Elemente enthalten sollten.

Natürlich können im record noch weitere Komponenten enthalten sein, so dass man bei der disjunkten Vereinigung in der Programmierung von einem "varianten Anteil" ("variant part") spricht. An folgendem Beispiel wird dies klar.

Bei Fahrzeugen kann man an verschiedenen Dingen interessiert sein: Wir nehmen an, dass für alle Fahrzeuge die Länge, die Breite, die Höhe und die Fahrzeugnummer vorliegen müssen, bei Bussen die Zahl der Sitzplätze, bei Lastkraftwagen die Größe der Ladefläche in qm und bei einem PKW die Zahl der Airbags. Dies führt zu folgendem Datentyp:

```
type mass is delta 0.001 range 0.0 .. 50.0;
type art is (PKW, LKW, Bus);
type fahrzeug is record
    länge, breite, höhe: mass;
    nummer: positive;
    case art is
        when Bus => sitzplätze: 8 .. 150;
        when LKW => ladefläche: positive;
        when PKW => airbagzahl: 0..10;
end record;
```