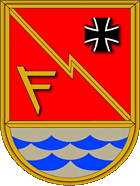
Schule Informationstechnik   
 der Bundeswehr



**Sprachausbildung Ada**



**„Dynamische Datenstrukturen“**

– Lernunterlage –

**Inhaltsverzeichnis**

[1 Dynamische Datenstrukturen 1](#_Toc528068698)

[1.1 Grundlagen dynamischer Datenstrukturen 1](#_Toc528068699)

[1.1.1 Verweise und Zielobjekte 1](#_Toc528068700)

[1.1.2 Deklaration von Zugriffstypen 3](#_Toc528068701)

[1.1.3 Operationen auf Zugriffsobjekte 4](#_Toc528068702)

[1.1.4 Zeiger auf unbeschränkte Reihungen oder Verbunde 10](#_Toc528068703)

[1.1.5 Freigabe von angefordertem Speicher 12](#_Toc528068704)

[1.2 Einsatzmöglichkeiten dynamischer Datenstrukturen 15](#_Toc528068705)

[1.3 Listen 19](#_Toc528068706)

[1.3.1 Speicherung verketteter Listen 19](#_Toc528068707)

[1.3.2 Unvollständige Typdefinition 20](#_Toc528068708)

[1.3.3 Operationen auf Listen 22](#_Toc528068709)

[1.3.4 Shallow Copy vs Deep Copy – Zwei Kopierverfahren 29](#_Toc528068710)

[1.3.5 Anwendungsbeispiel: Mengen 30](#_Toc528068711)

[1.4 Bäume 31](#_Toc528068712)

[1.4.1 Speicherung von Binärbäumen 33](#_Toc528068713)

[1.4.2 Operationen auf Binärbäume 34](#_Toc528068714)

[1.5 Graphen 41](#_Toc528068715)

[1.5.1 Speicherung von Graphen 42](#_Toc528068716)

[1.5.2 Operationen auf Graphen 44](#_Toc528068717)

[1.5.3 Autorouter 49](#_Toc528068718)

[1.6 Erweiterter Zugriff 57](#_Toc528068719)

[1.6.1 Zugriffe auf statisch deklarierte Objekte 57](#_Toc528068720)

[1.6.2 Globaler Zugriff auf lokale Objekte 59](#_Toc528068721)

[1.6.3 Zugriffe als formale Zugriffsparameter 59](#_Toc528068722)

[1.6.4 Zugriffe auf Unterprogramme 60](#_Toc528068723)

[Index I](#_Toc528068724)

# Dynamische Datenstrukturen

Die bisher vorgestellten Datentypen dienen der Deklaration von stati­schen Da­tenobjekten, die innerhalb eines Vereinbarungsteils deklariert und über ihren Bezeichner angesprochen werden. Sie werden als statisch bezeich­net, da sie während der ge­samten Lebensdauer einer Umgebung, zu der sie lokal sind, auf Grund einer Dekla­ration existieren und während des Programmablaufes nicht beliebig er­zeugt oder vernichtet werden können. Diese Re­striktion führt für Anwen­dungen, deren benötigte An­zahl an Datenobjekten zum Zeitpunkt der Programmie­rung unbekannt ist, zu Problemen.

dynamische

Datenobjekte

Aus diesem Grunde bietet Ada die Möglichkeit, Datenobjekte während des Programmab­laufes zu erzeugen (und – wie wir später noch sehen werden – zu vernichten). Datenobjekte dieser Art werden als dynamisch bezeichnet. Der Speicherplatz für sie wird erst während des Pro­grammablaufes auf Grund von Anweisungen, also in dem Maße, wie er benötigt wird, zur Verfügung gestellt.

## Grundlagen dynamischer Datenstrukturen

### Verweise und Zielobjekte



Verweis

Zeiger

Pointer

Dynamische Objekte (Zielobjekte) werden innerhalb eines Ada-Programmes nicht explizit de­klariert. Aus diesem Grunde kann auf sie auch nicht mit Hilfe eines Bezeichners direkt zugegriffen werden. Der Zugriff erfolgt über einen sogenannten Zugriffswert (Referenz) auf das entsprechende Zielobjekt. Den Zugriffswert kann man sich als Adresse der dynamischen Variablen im Arbeitsspeicher vorstellen, ein Typ eines Zugriffswerts ist aber ein privater Datentyp, die Operationen sind also beschränkt. Zur Aufnahme von Zugriffswerten dienen wiederum spezielle Kon­stanten oder Variablen, sogenannte Zugriffsobjekte (Zeiger, Verweis, engl. pointer).



Der Zeiger P enthält einen Zugriffswert auf eine dynamische Variable vom Typ INTEGER, die ihrerseits den Wert0 enthält.

Gleichheit von Zeigern

Zwei Zeiger sind genau dann gleich, wenn sie auf die gleiche dynami­sche Variable verweisen, also auf die gleiche „Adresse“ im Speicher zugreifen. Dabei ist der Wert der Zielobjekte, auf die die Zeiger verweisen unerheblich.

In folgendem Beispiel gilt: **P** = **Q**, **P** **R** und **Q** **R**:



Zeiger ohne Verweis

null

Zeiger müssen nicht grundsätzlich einen Verweis auf ein Zielobjekt enthalten. Alternativ ist auch der Wert null als Ausdruck dafür möglich, dass der Zeiger nicht auf ein Zielobjekt verweist. Der Wert null ist somit eine offene Konstante für Objekte von Zugriffstypen, die standardmäßig vordefiniert ist. Jeder Zeiger erhält null als Startwert, sofern er nicht explizit mit einem Startwert initialisiert wird. Damit haben Zeiger immer einen definierten Wert.



### Deklaration von Zugriffstypen



Es muss streng zwischen Zugriffsobjekten/ Zeigern und den Zielobjekten eines Zugriffstyps unterschie­den werden. Der Typ des Zielobjekts muss jedoch bei der Zugriffs­typ­deklaration ange­geben werden. Insbesondere können Zeiger auf beliebige, auch abstrakte Da­tentypen verweisen.



typ\_definition ::=

**type** identifier **is** access\_type\_definition

access\_type\_definition ::=

access\_to\_object\_definition

| access\_to\_subprogram\_definition

access\_to\_object\_definition ::=

**access** [general\_access\_modifier] subtype\_indication

general\_access\_modifier ::= all | constant

access\_to\_subprogram\_definition ::=

**access** [protected] procedure parameter\_profile

**|** **access** **[**protected**]** function parameter\_and\_result\_profile

In einem ersten Ansatz beschränken wir uns auf die grundlegende Definition von Zugriffstypen als Verweis auf dynamisch erzeugte Objekte.

access\_definition ::= **access** subtype\_mark ;

Man unterscheidet die Begriffe Zugriffstyp, Zieluntertyp und Zieltyp.

-- Zugriffstyp auf Zielobjekte vom Typ POSITIVE

type POSITIVE\_REF is access POSITIVE;

-- Zugriffstyp = POSITIVE\_REF

-- Zieluntertyp = POSITIVE

-- Zieltyp = INTEGER

-- Zugriffstyp auf Zielobjekte vom Typ FARBEN

type FARBEN\_T is (Rot, Gruen, Blau);

type FARBEN\_REF is access FARBEN\_T;

-- Zugriffstyp = FARBEN\_REF

-- Zieluntertyp = FARBEN\_T

-- Zieltyp = FARBEN\_T

Wir wollen unsere Betrachtung zunächst nur auf skalare und beschränkte Untertypen be­schränken. Die Untertypangabe besteht zunächst nur aus einem Untertypnamen.

subtype STRING\_20\_TA is STRING(1 .. 20);

type STRING\_20\_REF is access STRING\_20\_TA;

with Rational\_Zahlen;

...;

type RATIONALE\_ZAHL\_REF is access Rational\_Zahlen.RATIONALE\_ZAHL\_TR;

Zeiger

Auf der Basis eines Zugriffstypen (engl. access types**)** können nun Zugriffsobjekte (kurz Zeiger oder Pointer) deklariert werden.

Zeichenketten\_Ptr : STRING\_20\_REF;

Verweis\_auf\_Bruch\_Ptr : RATIONALE\_ZAHL\_REF;

Vorinitialisierung

Zugriffsobjekte sind Objekte, die bei der Deklaration automatisch mit einem Wert, und zwar den Wert null, vorinitialisiert werden, wenn nicht explizit anders besetzt:

Vorinitialisiert\_Ptr : STRING\_20\_REF; -- hat Wert null

Zugriffsobjekte unterscheiden sich hinsichtlich ihres Typs voneinander.

Folgende Zuweisung ist nicht erlaubt:



Zeichenketten\_Ptr := Verweis\_Auf\_Bruch\_Ptr; -- Syntaxfehler

### Operationen auf Zugriffsobjekte

Operationen auf Zugriffsobjekte

Auf Zugriffsobjekte (Zeiger) sind nur die folgenden Operationen vordefiniert:

* Zuweisung :=
* Vergleiche =, /=

Sie gehören damit zu der Typklasse der privaten Datentypen. Werden weitere Operationen benötigt, sind diese explizit zu erstellen.

Allokator

new



Zeiger können in Ada 83 nicht auf Variablen zeigen, die über eine statische Variablen­deklaration ge­schaffen wurden. In Ada 95 können Zeiger auch auf statische Objekte (aliased) verweisen. Die Erzeugung eines Zielobjekts wird in Ada 83 ausschließlich mit Hilfe eines sogenannten Allokators new während des Programm­ablaufes durchgeführt.

Untertypangabe

Qualifizierter Ausdruck

Der Allokator new wird in Ausdrücken eingesetzt und erhält als Parameter eine Untertypangabe oder einen qualifizierten Ausdruck. Bei beiden Versionen wird Speicher­platz für den angegebenen Zieluntertyp ange­fordert. Zusätzlich wird bei qualifizierten Ausdrücken der Ausdruck verwendet, um die dynamische Variable zu initialisieren.



type POSITIVE\_REF is access POSITIVE;

-- Allokator mit Untertypangabe: 1. Version

A\_Ptr : POSITIVE\_REF := new POSITIVE;

-- Allokator mit qualifiziertem Ausdruck: 2. Version

B\_Ptr : POSITIVE\_REF := new POSITIVE'(1);

Allokations-Schritte

Die Ausführung des Allokators new er­folgt in drei Schritten:



1. Erzeugung des (dynamischen) Zielobjekts
2. ggf. Initialisierung des Zielobjekts
3. der Allokator liefert die Adresse des Zielobjekts an das Zugriffsobjekt



Allokator ::=

**new** Untertypangabe | **new** qualifizierter Ausdruck

Der Allokator new ist dabei als eine Funktion zu sehen, die ein Zielobjekt erzeugt und einen Verweis darauf zurückliefert. Der Typ des Verweises muss aus dem Kontext bestimmt werden können. Ansonsten kann er eingesetzt werden wie ein Funktionsaufruf:

function NewWert (P : in POSITIVE) return ADRESSE;

A\_Ptr : ADRESSE := NewWert (5);

Kollektion

Bei der Definition eines Zugriffstyps wird für die Zielobjekte Speicherplatz reserviert, der als Kollektion oder Halde bezeich­net wird.



Einschränkungen bezüglich des Zieluntertyps gelten für die gesamte Kollektion, egal ob bei einer Objekterzeugung eine Untertyp­angabe des Zieltyps oder der Zieltyp selbst angegeben ist.

P1\_Ptr : POSITIVE\_REF := new INTEGER'(1000);

-- CONSTRAINT\_ERROR wegen Verletzung der Einschraenkung:

P2\_Ptr : POSITIVE\_REF := new INTEGER'(-1000);

-- Initialisierung des Objekts, jedoch keine zusaetzliche

-- Einschraen­kung. P3 kann weiterhin auf Objekte zeigen, die

-- im Wertebereich von POSITIVE liegen:

subtype KLEINE\_ZAHL\_T is POSITIVE range 1 .. 100;

P3\_Ptr : POSITIVE\_REF := new KLEINE\_ZAHL\_T'(100);

-- CONSTRAINT\_ERROR, jedoch nicht wegen der Bereichs-

-- einschraenkung der Kollektion, sondern bzgl. des

-- qualifizierten Ausdrucks:

P4\_Ptr : POSITIVE\_REF := new KLEINE\_ZAHL\_T'(1000);

-- Erzeugung ohne Initialisierung:

P5\_Ptr : POSITIVE\_REF := new KLEINE\_ZAHL\_T;

Werden abgeleitete Typen von Zugriffstypen gebildet, benutzen sie die­selbe Kollek­tion wie der Vatertyp.



Referenz

dereferenzieren

.all

Zeiger stellen also sogenannte Referenzen auf Objekte dar. Den Vorgang des Zu­griffs auf das Objekt selbst nennt man daher dereferenzieren. Zu diesem Zweck wird der Bezeichner eines Zugriffsobjekts mit dem Suffix .all versehen.



**P** bezeichnet eine statische Variable, die einen Verweis auf ein Zielobjekt enthält, während P.all das Zielobjekt bezeich­net.

Die Zuweisung

P.All := 10;

verändert z.B. den Wert des Zielobjekts auf 10.

Betrachten wir folgende Ausgangssituation:



Durch die Zuweisung

P.All := Q.All;

ergibt sich folgende Situation:



Der Wert des Zielobjekts von **Q** ist dem Zielobjekt von **P** zugewiesen worden. Geht man nun wieder von der Ausgangssituation aus, so ergibt sich als Resultat der Zu­weisung

P := Q;

folgende Situation:



In diesem Falle wurde der Verweis, den die Zeigervariable **Q** enthält, der Zeiger­variablen **P** als neuer Inhalt zugewiesen. Diese verweist nun auf dasselbe Zielobjekt wie **Q**.



Auf das Zielobjekt, auf das **P** ursprünglich verwie­sen hat, kann nach der Zuweisung nicht mehr zugegriffen werden, wenn auf dieses kein weiterer Verweis existiert.

Folgende Anweisungen erzeugen dynamisch 100 neue Zielobjekte. Da aber nur auf das letzte ein Verweis existiert, kann auf die ersten 99 nicht mehr zugegriffen werden.

type INTEGER\_REF is access INTEGER;

Int\_Ptr : INTEGER\_REF;

for I in 1 .. 100 loop

Int\_Ptr := new INTEGER;

end loop;

Dereferenzierung bei

zusammen­gesetzten Datentypen



Da der Zieltyp keinerlei Einschränkungen unter­liegt, sind insbesondere auch zusammengesetzte Datentypen möglich. In diesen Fällen ergeben sich Besonderheiten bei Zugriff auf das entspre­chende Zielobjekt.

type LOCHKARTE\_TA is array (1 .. 80) of CHARACTER;

type LOCHKARTE\_REF is access LOCHKARTE\_TA;

Zugriffsobjekt

[.all](Index)

Lochkarten\_Ptr : LOCHKARTE\_REF := new LOCHKARTE\_TA;



Zugriffsobjekt

[.all].Komponente

Lochkarten\_Ptr.All (3) := 'a'; -- entspricht der Schreibweise

Lochkarten\_Ptr(3) := 'a';

type DATUM\_TR is record

Tag : INTEGER range 1 .. 31;

Monat : INTEGER range 1 .. 12;

Jahr : INTEGER range 1900 .. 2100;

end record;

type DATUM\_REF is access DATUM\_TR;

Datum\_Ptr : DATUM\_REF;



Datum\_Ptr.All.Monat := 6; -- entspricht der Schreibweise

Datum\_Ptr.Monat := 6;

Implizite Dereferen­zierung

Bei Zugriffen auf Komponenten oder Bereiche von zusammengesetzten Datentypen kann der Suffix .all weggelassen werden, da aus dem Kontext zu erkennen ist, dass hier nicht der Wert des Zeigers, son­dern der des Zielobjekts gemeint ist.

Entsprechend sind für Zeigervariablen die Reihungsattribute definiert:

Lochkarten\_Ptr.All'Length = 80 -- entspricht der Schreibweise

Lochkarten\_Ptr'Length = 80



Werden zusammengesetzte Datentypen im Ganzen angesprochen, ist der Suffix .all jedoch immer anzugeben.

Start\_Ptr : DATUM\_REF := new DATUM\_TR;

Start\_Ptr := Datum\_Ptr; -- Zuweisung an ein Zugriffsobjekt

Start\_Ptr.All := Datum\_Ptr.All; -- Zuweisung an ein Zielobjekt

-- vom Typ DATUM



Ist ein Zugriffsobjekt als konstant definiert, kann der Zugriffswert nicht verän­dert werden und zeigt immer auf das gleiche Zielobjekt. Den­noch kann der Wert des Zielobjekts verändert werden.



Unbeugsamer\_Ptr : constant DATUM\_REF := new DATUM\_TR'(17, 1, 1994);

-- Zuweisung eines neuen Wertes an die dynamische Variable:

Unbeugsamer\_Ptr.All := (2,4,1996);

-- unzulaessiger Versuch, den konstanten Verweis zu aendern:

Unbeugsamer\_Ptr:= Start\_Ptr;

### Zeiger auf unbeschränkte Reihungen oder Verbunde

Zeiger auf unbeschränkte Reihungen

Zeiger können auch auf unbeschränkte Typen verweisen.

type STRING\_REF is access STRING; -- (1)

type STRING\_20\_REF is access STRING(1 .. 20); -- (2)

Mit diesen Definitionen werden zwei Kollektionen angelegt, die prinzi­piell Objekte des Datentyps STRINGaufnehmen können.

Die Kollektion (1) dient zur Speicherung beliebig langer Zeichenketten,  
Kollektion (2) nur zur Speicherung von Zeichenketten mit der ange­gebe­nen Indexeinschränkung.

Z1\_Ptr : STRING\_REF;

Z2\_Ptr : STRING\_20\_REF;

Z1\_Ptr := Z2\_Ptr; -- ist nicht erlaubt (Typkonflikt)

Z1\_Ptr.All := Z2\_Ptr.All; -- dagegen nur, wenn die

Z2\_Ptr.All := Z1\_Ptr.All; -- Indexeinschraenkungen

-- uebereinstimmen und kein Zeiger den Wert

-- null hat, ggf. wird die Ausnahme

-- CONSTRAINT\_ERROR ausgeloest.

Bei der Allokation muss aus dem Kontext eindeutig ersichtlich sein, welche Indexeinschränkung das erzeugte Objekt hat. Diese Indexeinschränkung muss den Einschränkungen des Zielobjekts genügen. Ist der Zieluntertyp ein unbeschränkter Typ, ist bei der Allokation eine Indexeinschränkung anzugeben.

Z1\_Ptr := new STRING; -- nicht erlaubt

Z1\_Ptr := new STRING'( others => ' ' ); -- nicht erlaubt



Z1\_Ptr := new STRING(1 .. 10); -- erlaubt

Z1\_Ptr := new STRING(1 .. 10)'( others => ' ' ); -- nicht erlaubt

Z1\_Ptr := new STRING'(1 .. 10 => ' ' ); -- erlaubt

Z1\_Ptr := new STRING'("Hallo"); -- erlaubt

Z2\_Ptr := new STRING; -- nicht erlaubt

Z2\_Ptr := new STRING'(others => ' '); -- nicht erlaubt

Z2\_Ptr := new STRING'("Hallo"); -- erlaubt, aber CONSTRAINT\_ERROR

Z2\_Ptr := new STRING(1 .. 10); -- erlaubt, aber CONSTRAINT\_ERROR

Ist der Zieluntertyp eines Zugriffstyps ein unbeschränkter Reihungstyp, können auch einzelne Zugriffsobjekte mit einer Indexeinschränkung versehen werden.



Z3\_Ptr : STRING\_REF(1 .. 20); -- nur die Zugriffsobjekte

-- sind beschränkt

Z4\_Ptr : STRING\_20\_REF(1 .. 10); -- nicht erlaubt

In diesem Fall kann Z3 nur auf Objekte verweisen, die der angegebenen Indexeinschränkung genügen. Die Kollektion selbst wird dadurch nicht eingeschränkt und dient weiterhin der Speicherung beliebig langer Zeichenketten.

Z3\_Ptr := new STRING (1..10); -- CONSTRAINT\_ERROR

Zeiger auf unbeschränkte Verbunde

Ist der Zieluntertyp ein Verbund mit vorbesetzten Diskriminanten, so kann auch hier der Ziel­typ bereits einge­schränkt werden:

type TEXT\_TR (Laenge : NAT100\_T := 20 ) is record

Inhalt : STRING(1 .. Laenge);

end record;

type TEXT\_REF is access TEXT\_TR; -- (1)

type TEXT\_20\_REF is access TEXT\_TR(20); -- (2)

Die Kollektion (1) kann beliebige große Verbunde vom Datentyp TEXT\_TR aufneh­men, während die Kollektion (2) nur der Speicherung solcher Verbunde dient, die der Indexeinschränkung genügen.



Bei der Allokation wird ggf. der initialisierende Diskriminantenwert implizit an­genom­men.

Dabei ist zu beachten, dass dynamische Zielobjekte im Gegen­satz zu statischen Variablen immer beschränkt sind.

T1\_Ptr : TEXT\_REF;

T2\_Ptr : TEXT\_20\_REF;

T3 \_Ptr : TEXT\_REF (20); -- nur die Objekte sind beschraenkt

T1\_Ptr := new TEXT\_TR; -- impliziter Diskriminantenwert Laenge = 20

T1\_Ptr := new TEXT\_TR (10); -- expl. Diskriminantenwert Laenge = 10

T1\_Ptr.All := (5, "Hallo"); -- CONSTRAINT\_ERROR, da Zielobjekte

-- immer beschraenkt sind

T2\_Ptr := new TEXT\_TR; -- CONSTRAINT\_ERROR, impliziter

-- Diskriminantenwert 0 und Einschraenkung widersprechen sich

T2\_Ptr := new TEXT\_TR(20); -- OK

T1\_Ptr := T2\_Ptr; -- nicht erlaubt, da unterschiedliche Typen

Index- und Diskriminanteneinschränkungen sind die einzigen erlaubten Formen von Untertypeinschränkungen eines Zieluntertyps.

### Freigabe von angefordertem Speicher

Ausnahme STORAGE\_  
ERROR

Die bisherige, vereinfachte Betrachtungsweise des Speichers als unerschöpflicher Pool, aus dem der Allokator new beliebig schöpfen kann, ist für Programme mit ge­ringem Speicherbedarf in Relation zum verfügbaren Speicher ausreichend. Trifft diese Einschränkung jedoch nicht zu, so kann die Speicherplatzgrenze erreicht werden, was zu einem STORAGE\_ERROR führt.

garbage collection

In einigen Ada-Implementierungen wird diesem Problem durch einen Mechanismus vorgebeugt, der als Speicherbereinigung (engl. garbage collection) bezeichnet wird:

automatische Deallokation

Wird eine dynamische Variable nicht mehr durch einen Verweis referenziert, so wird sie automatisch in den zur Verfügung stehenden freien Speicher zurückgeführt (dieser Vorgang wird auch als Deallokation bezeich­net). Der Vorteil der Speicher­bereinigung ist, dass ausschließlich nicht referenzierte dynamische Variablen deallokiert werden. Fehler im Bereich der Programmierung (Deallokation von refe­renzierten Variablen) sind also ausgeschlossen. Der Nachteil besteht darin, dass die Speicherbereinigung in der Regel nicht permanent, sondern periodisch durchgeführt wird, ggf. auch zu einem laufzeitkritischen Zeitpunkt.

pragma CONTROLLED

Um eine garbage collection durch das Ada-Laufzeitsystem für einen Zeigertyp explizit auszuschließen, kann das Pragma CONTROLLED verwendet werden.

type STRING\_REF is access STRING;

pragma CONTROLLED (STRING\_REF);

Das Programm ist jetzt selbst für die eventuelle Freigabe nicht mehr genutzten Speicherplatzes zuständig. Dazu bietet Ada eine generische Prozedur Unchecked\_Deallocation an, die die Möglichkeit kontrollierter Deallokation bietet.



generic

type OBJECT (<>) is limited private;

type NAME is access OBJECT;

procedure Unchecked\_Deallocation (X : in out NAME);

Die beiden generischen Parameter stellen den Zieltyp der zu deallokierenden Variablen und den entsprechenden Zugriffstyp dar.

Der Aufruf der entsprechen­den Freigabeprozedur mit einer dynamischen Variablen be­wirkt zweierlei. Zum einen ist von da an der Zeigerwert null, zum anderen wird das Objekt, auf das der Zeiger verwiesen hat, als frei gekenn­zeichnet.



Die Verwendung dieser Pro­zedur hat aber mit der nötigen Vor­sicht zu geschehen. Hat man versäumt, andere Zeiger, die auf dieses Objekt deuten, zu löschen oder umzusetzen, so entstehen bei Zugriff über diese Zeiger unvorhersehbare Ereignisse. Eine Überprüfung, ob noch an­dere Zeiger auf das zu löschende Objekt verweisen, findet nicht statt (unchecked).

Beispiel:

with Ada.Unchecked\_Deallocation;

with Ada.Text\_IO;

procedure Test is

type STRING\_REF is access STRING;

pragma CONTROLLED (STRING\_REF);

X\_Ptr, Y\_Ptr : STRING\_REF;

procedure Freigabe is

new Ada.Unchecked\_Deallocation (STRING, STRING\_REF);

begin

X\_Ptr := new STRING'("Hallo Welt");

Y\_Ptr := X\_Ptr;

Freigabe(X\_Ptr);

...

Ada.Text\_IO.Put\_Line(Y\_Ptr.All); -- frei haengender Zeiger

-- Wirkung ist nicht definiert!

end Test;



Wichtigste Voraussetzung für die Anwendung dieser Prozedur ist ein Zeiger, der auf das zu löschende Element zeigt. Elemente oder Elementketten, deren Referenz im Programmlauf verloren ging, können nicht explizit freigegeben werden.

Jedes größere Servicepaket insbesondere mit abstrakten Datenobjekten sollte seine Halden selbständig verwalten. Zur Demonstration hierfür wurde die Prozedur Entnahme aus dem Paket Men­gen\_Paket (siehe Abschnitt 1.3.5) um die Freigabeprozedur erweitert.

with Ada.Unchecked\_Deallocation;

package body Mengen\_Paket is

procedure Freigabe is new Unchecked\_Deallocation

(Object => KNOTEN\_T, Name => KNOTEN\_REF);

...;

procedure Entnahme (

M : in out MENGE;

E : in out ELEMENT) is

Hilf\_Ptr : KNOTEN\_REF;

begin -- Entnahme

if M = Leere\_Menge then

raise Zugriff\_Auf\_Leere\_Menge;

else

Zuweisung (E, M.Start\_Element.Info);

Hilf\_Ptr := M.Start\_Element;

M.Start\_Element := M.Start\_Element.Nachfolger;

M.Anzahl\_Elemente := M.Anzahl\_Elemente - 1;

Freigabe (Hilf\_Ptr);

end if;

end Entnahme;

end Mengen\_Paket;

Im Folgenden wird bei allen Beispielprogrammen und -algorithmen im Skript die Freigabe explizit durchgeführt.

## Einsatzmöglichkeiten dynamischer Datenstrukturen



Die Verwendung von dynamischen Datenstrukturen hat sich für eine spezielle Klasse von Algorithmen als vorteilhaft erwiesen. Die im Folgenden geschil­derten Problemstel­lungen stellen ein Auswahl aus den umfangreichen Einsatzmöglichkeiten dar:

1. Haltung äquivalenter Daten innerhalb verschiedener Daten­struk­turen

Die Realisierung kann derart erfolgen, dass redundante Daten nur einmal in Form von dynamische erzeugten Zielobjekten gehalten werden (d.h. im Speicher nur einmal existent sind) und von verschiedenen Daten­strukturen mittels Zu­griffs­objekten auf diese verwiesen wird.

type EINKOMMENSPROFIL\_TR is record

Gehalt : POSITIVE;

Anzahl\_Zahlungen : POSITIVE;

...

end record;

type EINKOMMENSPROFIL\_REF is access EINKOMMENSPROFIL\_TR;

type MITARBEITER\_TR is record

PK : PERSONENKENNZIFFER\_T;

Daten : ...;

Gehaltsangaben : EINKOMMENSPROFIL\_REF;

end record;

type BELEGSCHAFT\_TA is array (POSITIVE range <>)

of MITARBEITER\_TR;



1. Kopieren umfangreicher Datenstrukturen

Der Laufzeitaufwand für Kopiervorgänge kann dadurch reduziert werden, dass Daten in Zielobjekten mit entsprechenden Verweisen darauf ge­halten werden. Dadurch reduziert sich das Kopieren auf eine einfache Zuweisungsoperation auf Zeigervariablen. Vor allem bei Datenbankanwendungen werden solche Zeiger benutzt, um die Zugriffszeiten gering zu halten.

1. Haltung von Daten, deren Umfang zum Zeitpunkt der Pro­gram­mierung nicht absehbar ist

Die Kapazität statischer Datenstrukturen wird zum Zeitpunkt der Programmierung festgelegt. Somit steht der Programmierer vor der Alterna­tive:

* Planung von „durchschnittlichem“ Datenaufkommen: Das Pro­gramm ist großen Anforderungen nicht gewachsen
* Planung des „maximalen“ Datenaufkommens: Unnötig große Speicherplatzbelegung für die Masse der Anforderungen.

Dieses Problem stellt sich nicht bei der Verwendung von dynami­schen Daten­strukturen, die zur Programmlaufzeit bedarfsgerecht angefor­dert werden können.

Beispiel zu 2 und 3

-- Datenstruktur eines Texteditors

type STRING\_REF is access STRING;

type TEXTSEITE\_TA is array (POSITIVE range <>) of STRING\_REF;

H\_Ptr : STRING\_REF;

Seite : TEXTSEITE\_TA(1 .. 60);

Puffer : STRING(1 .. 256);

I : POSITIVE := 2;

J : POSITIVE := 3;

Letzter\_Index : NATURAL := 0;

-- Vertauschen der Zeilen i und j (Beispiel zu 2):

H\_Ptr := Seite(I); -- nur die Zeiger werden vertauscht

Seite(I) := Seite(J);

Seite(j) := H\_Ptr;



-- Einlesen (Beispiel zu 3):

for I in Seite'Range loop

Ada.Text\_IO.Get\_Line (Puffer, Letzter\_Index);

if Puffer'Last = Letzter\_Index then

Ada.Test\_IO.Skip\_Line;

end if;

-- bedarfsgerechte Anforderung von Speicher:

Seite(I) := new STRING'(Puffer(1.. Letzter\_Index));

end loop;

...

1. Erzwingen von Seiteneffekten

Bei der Übergabe von zusammengesetzten Datentypen können bei vorzeitigem Verlassen des entsprechenden Unterprogramms imple­mentierungsab­hängige Effekte auftreten:

type REIHE\_TA is array (1 .. 100\_000) of ...;

type REIHE\_REF is access REIHE\_TA;

procedure P1 (R : in out REIHE\_TA) is

begin -- P1

for I in R'Range loop

Aktion (R(I));

end loop;

end P1;

procedure P2 (R : in REIHE\_REF) is

begin -- P2

for I in R.All'Range loop

Aktion (R(I));

end loop;

end P2;

Wird in Aktion bei der Bearbeitung des 98.000ten Datensatz eine Aus­nahme ausgelöst, so sind in P2 die ersten Datensätze garantiert bearbei­tet, in P1 dagegen hängt die Bearbeitung von der Implemen­tierung des Übergabemechanismus ab (call-by-reference, copy-in-copy-back).



Es ist damit außerdem möglich, Objekte an Unterprogramme zu überge­ben, die trotz des Modus in indirekt verändert werden können.



1. Bildung rekursiver Datenstrukturen.

Rekursive Datenstrukturen sind Datenstrukturen, die sich in ihrer Typvereinbarung durch sich selbst definieren. Dies ist für statische Datenstrukturen nicht möglich. Bei­spielsweise können keine Verbunde definiert werden, die Kom­ponenten desselben Verbundtyps enthalten, da der Platzbedarf einer solchen statischen Datenstruktur unendlich wäre.

type PERSON\_TR is record

Name,

Vorname : STRING(1 .. 20);

Vater,

Mutter : **PERSON\_TR**; -- nicht erlaubt

end record;

Für dynamische Datenstrukturen ist dies jedoch möglich.

Auf den folgenden Seiten werden drei rekursive Datenstrukturen vorgestellt:

* Listen
* Bäume
* Graphen

## Listen

Eine Liste ist eine Folge von mehreren Elementen, die linear an­geordnet sind. Die Art der Elemente spielt für die Handhabung einer Liste keine Rolle. Wenn eine Liste keine Elemente enthält, so wird sie als leere Liste bezeichnet.

### Speicherung verketteter Listen

Jedes Listenelement besteht aus der abzuspeichernden Information und aus einem Verweis (definiert die Rekursion) auf das folgende Listenelement. Der Verweis auf das erste Listenelement wird in einem besonderen Zugriffsobjekt, dem Startzeiger abgespei­chert. Die weiteren Listenelemente können über die Verweise der einzelnen Elemente erreicht werden. Die Verweiskomponente des letzten Listenelementes enthält den Wert null, um das Listenende zu kennzeich­nen.



Warum ist eine Liste eine rekursive Datenstruktur?

Jede Teilliste stellt ihrerseits wieder eine Liste dar. Jedes Listenelement zeigt auf den Anfang einer evtl. auch leeren Restliste.

Rekursive Definition:

* Eine leere Liste (Startzeiger = null) ist eine Liste
* Eine nicht-leere Liste besteht aus einem Listen­ele­ment, das einen Start­zeiger auf eine (kürzere) Liste enthält.

### Unvollständige Typdefinition

Um das vorherige Schaubild als Datenstruktur anzulegen, müssten bei dynamischen Datenstrukturen Verbunde definiert werden, die als eine Komponente den Zeigertyp für denselben Verbund­typ enthalten.

type ELEM\_TR is record

Info : INTEGER;

Next\_Ptr : **ELEM\_REF**; -- Fehler, da ZEIGER unbekannt

end record;

type ELEM\_REF is access ELEM\_TR;

unvollständige Typdefinition

Diese Typdefinitionen sind nicht korrekt, da nur vorher definierte Typenbezeichner verwendet werden dürfen. Auch ein Vertauschen der beiden Typdeklarationen bringt keine Lösung, da beide Typen sich gegenseitig benötigen. Ada bietet hierfür die unvollständige Typdefinition als Lösung an:



unvollständige\_Typvereinbarung ::=

**type** Bezeichner [ Diskrimantenanteil ]**;**



Bei einer **unvollständigen** Typvereinbarung folgt nach dem Schlüsselwort type nur der Typbezeichner. Der Typbezeichner darf nur für die Definition von Zugriffstypen als Zieltyp verwendet werden. Diskriminanten von Elementtypen müssen auch im unvollständigen Teil vollständig sichtbar sein (analog zum ADT).



type ELEM\_TR;

type ELEM\_REF is access ELEM\_TR;

type ELEM\_TR is record

Info : INTEGER;

Next\_Ptr : ELEM\_REF;

end record;

Jetzt kann die Zugriffstypdefinition angegeben werden, da der dazu benötigte Typbe­zeichner ELEM\_TR bekannt ist. Im Anschluss wird die vollständige Typdeklaration sofort nachgeholt.

Jetzt können Zugriffsobjekte vom Typ ELEM\_REF und Zielobjekte vom Typ ELEM\_TR dynamisch angefordert werden.

-- Deklaration von Zugriffsobjekten:

Startzeiger, Pointer : ELEM\_REF; -- (0)

-- Erzeugung von drei Zielobjekten mit Initialisierung:

Startzeiger := new ELEM\_TR'(100, Startzeiger); -- (1)

Startzeiger := new ELEM\_TR'(200, Startzeiger); -- (2)

Pointer := Startzeiger;

Startzeiger := new ELEM\_TR'(300, Startzeiger); -- (3)

-- Zuweisung an die Verbundkomponente Info:

Pointer.All.Info := 250;

-- Zeiger auf das dritte Element der Liste setzen:

Pointer := Startzeiger.All.Next\_Ptr.All.Next\_Ptr; -- (4)





### Operationen auf Listen



Im Folgenden wollen wir uns mit dem Operationen

* Durchsuchen
* Einfügen
* Löschen

beschäftigen. Dabei wollen wir von folgender Datenstruktur ausgehen, die zur Implementierung einer sortierten Liste dient:



type ELEM\_TR;

type ELEM\_REF is access ELEM\_TR;

type ELEM\_TR is record

Schl : SCHLUESSEL\_T; -- Sortierkriterium mit definiertem "<="

Info : INFORMATION\_T; -- nicht naeher definiert

Next\_Ptr : ELEM\_REF;

end record;

Durchlaufen



Beim Durchlaufen wollen wir auf jedes Listenelement einen einheitlichen Algorith­mus Aktion auf die jeweilige Information anwenden.

generic

with procedure **Aktion** (

Schl : in SCHLUESSEL\_T;

Info : in out INFORMATION\_T) is <>;

procedure Durchlaufen (Start\_Ptr : in ELEM\_REF);

procedure Durchlaufen (Start\_Ptr : in ELEM\_REF) is

Hilfs\_Ptr : ELEM\_REF := Start\_Ptr;

begin -- Durchlaufen

while Hilfs\_Ptr /= null loop

**Aktion (Hilfs\_Ptr.All.Schl, Hilfs\_Ptr.All.Info);**

-- Zeiger "weiterschalten"

Hilfs\_Ptr := Hilfs\_Ptr.All.Next\_Ptr;

end loop;

end Durchlaufen;

Iterator

Die oben angegebene generische Prozedur dient also als Implementie­rung einer generischen Schleife über alle Listenelemente und wird entsprechend als Iterator bezeichnet.

Einfügen



Beim Einfügen in eine geordnete Liste muss zunächst die Einfügestelle gefunden werden. Anschließend wird das neue Listenelement „eingekettet“.

Ein Einfügen eines Elements mitten in der Liste stellt sich wie folgt dar:



Das Ergebnis nach der Einkettung:



Bei der Suche ist zu beachten, dass ein Verweis auf das Vorgänger­element der Ein­fügestelle benötigt wird, da dessen Nachfolgerverweis auf das Einfügeelement „umgebogen“ werden muss.

Neben dem in der Zeichnung dargestellten Einfügevorgang, gibt es noch drei Spezial­fälle, die auf den ersten Blick unterschiedliche Aktionen notwendig machen.

* Einfügen in eine leere Liste

Der Zeiger Start hatte den Wert null. Der Zeiger muss auf das neue Element „umgebogen“ werden.

* Einfügen am Anfang einer Liste

Die Nachfolgerkomponente des neuen Elementes bekommt den In­halt des Zeigers Start und Start selbst muss auf das neue Element verweisen.

* Einfügen am Ende einer Liste

Die Verweiskomponente des neuen Elementes ist mit null zu initiali­sieren und die Verweiskomponente des letzten Elementes zeigt auf das neue Element.

Wenn man die beiden ersten Fälle genauer betrachtet, kommt man zu dem Ergebnis, dass sie zu einem Fall „Einfügen am Anfang einer Liste“ zusammengefasst werden können. Ebenso kann der Fall 3 wie ein „Einfügen innerhalb einer Liste“ behandelt werden.

sortiertes Einfügen in eine lineare Liste



procedure Einfuegen (

Liste\_Ptr : in out ELEM\_REF;

Schl : in SCHLUESSEL\_T;

Info : in INFORMATION\_T) is

Vorgaenger\_Ptr,

Einfuege\_Ptr : ELEM\_REF := Liste\_Ptr;

begin -- Einfuegen

-- Einfuegestelle suchen:

while Einfuege\_Ptr /= null and then Einfuege\_Ptr.Schl <= Schl

loop

Vorgaenger\_Ptr := Einfuege\_Ptr;

Einfuege\_Ptr := Einfuege\_Ptr.Next\_Ptr;

end loop;

-- Einketten:

if Vorgaenger\_Ptr = Einfuege\_Ptr then

-- Einfuegen am Listenanfang oder leere Liste

Liste\_Ptr := new ELEM\_TR'(Schl, Info, Liste\_Ptr­);

else

-- Einfuegen innerhalb der Liste oder am Listenende

Vorgaenger\_Ptr.Next\_Ptr :=

new ELEM\_TR'(Schl, Info, Einfuege\_Ptr);

end if;

end Einfuegen;

Ausfügen



Beim Ausfügen aus einer linearen Liste wollen wir alle Listenelemente mit einem gegebenen Schlüssel aus der Liste löschen.

Ist ein auszufügendes Element gefunden, hat die Verweiskomponente des Vorgängers auf das Element zu zeigen, welches dem auszufügendem Element folgt. Das auszufügende Element sollte danach explizit freigegeben werden:

Freigabe von dynamischen Listenelementen

procedure Freigabe is new Unchecked\_Deallocation(ELEM\_TR, ELEM\_REF);



Auch hier sind neben o.a. Fall die folgenden Spezialfälle zu beachten, die besondere Beachtung erfordern:

* Ausfügen in einer leeren Liste

Der Zeiger Start hat den Wert null und es müssen keine weiteren Aktionen durchgeführt werden.

* Ausfügen am Anfang einer Liste

Der Zeiger Start hat jetzt auf das zweite Listenelement zu zeigen.

* Ausfügen am Ende einer Liste

Die Verweiskomponente des vorletzten Listenelementes ist mit null zu initialisieren.

Auch beim Ausfügen können jeweils zwei Fälle zusammengefasst werden.

Löschen in einer geordneten linearen Liste



Schlüssel

eindeutig

procedure Ausfuegen (

Liste\_Ptr : in out ELEM\_REF;

Schl : in SCHLUESSEL\_T) is

Vorg\_Ptr, Loesch\_Ptr : ELEM\_REF := Liste\_Ptr;

begin -- Ausfuegen

-- Ausfuegestelle suchen:

while Loesch\_Ptr /= null and then Loesch\_Ptr.Schl < Schl loop

Vorg\_Ptr := Loesch\_Ptr;

Loesch\_Ptr := Loesch\_Ptr.Next\_Ptr;

end loop;

-- das Elemente mit dem Schluessel Schl loeschen:

if Loesch\_Ptr /= null and then Loesch\_Ptr.Schl = Schl then

if Vorg\_Ptr = Loesch\_Ptr then

-- Loeschen am Listenanfang

Liste\_Ptr := Liste\_Ptr.Next\_Ptr;

else

-- Loeschen innerhalb der Liste oder am Listenende

Vorg\_Ptr.Next\_Ptr := Loesch\_Ptr.Next\_Ptr;

end if;

Freigabe(Loesch\_Ptr);

end if;

end Ausfuegen;

Wiederholtes Löschen in einer geordneten linearen Liste



Schlüssel

nicht eindeutig

procedure Ausfuegen (

Liste\_Ptr : in out ELEM\_REF;

Schl : in SCHLUESSEL\_T) is

Vorg\_Ptr, Loesch\_Ptr : ELEM\_REF := Liste\_Ptr;

begin – Ausfuegen

-- Ausfuegestelle suchen:

while Loesch\_Ptr /= null and then Loesch\_Ptr.Schl < Schl loop

Vorg\_Ptr := Loesch\_Ptr;

Loesch\_Ptr := Loesch\_Ptr.Next\_Ptr;

end loop;

-- alle Elemente mit dem Schluessel Schl loeschen:

while Loesch\_Ptr /= null and then Loesch\_Ptr.Schl = Schl loop

if Vorg\_Ptr = Loesch\_Ptr then -- Loeschen am Listenanfang

Liste\_Ptr := Liste\_Ptr.Next\_Ptr;

Freigabe(Loesch\_Ptr);

Vorg\_Ptr := Liste\_Ptr;

Loesch\_Ptr := Liste\_Ptr;

else -- Loeschen innerhalb der Liste oder am Listenende

Vorg\_Ptr.Next\_Ptr := Loesch\_Ptr.Next\_Ptr;

Freigabe(Loesch\_Ptr);

Loesch\_Ptr := Vorg\_Ptr.Next\_Ptr;

end if;

end loop;

end Ausfuegen;

Alle iterativen Listenoperationen lassen sich auch rekursiv lösen. Im Folgenden sind für lineare Listen die rekursiven Versionen für Durchlaufen, Einfügen und Ausfügen aufgeführt:



Rekursives Durchlaufen einer linearen Liste und Anwendung einer generisch übergebenen Aktion:

generic

with procedure Aktion (

Schl : in SCHLUESSEL\_T;

Info : in out INFORMATION\_T);

procedure Durchlaufen (Liste\_Ptr : in ELEM\_REF);

rekursives Durchlaufen

procedure Durchlaufen (Liste\_Ptr : in ELEM\_REF) is

begin

if Liste\_Ptr /= null then

Aktion (Liste\_Ptr.Schl, Liste\_Ptr.Info);

**Durchlaufen(Liste\_Ptr.Next\_Ptr);**

end if;

end Durchlaufen;

Rekursives Einfügen in eine sortierte, lineare Liste:

rekursives Einfügen

procedure Einfuegen (

Liste\_Ptr : in out ELEM\_REF;

Schl : in SCHLUESSEL\_T;

Info : in INFORMATION\_T) is

begin

if Liste\_Ptr = null or else Schl <= Liste\_Ptr.Schl then

Liste\_Ptr := new ELEM\_TR'(Schl, Info, Liste\_Ptr);

else

**Einfuegen(Liste\_Ptr.Next\_Ptr, Schl, Info);**

end if;

end Einfuegen;

Einmaliges rekursives Löschen aus einer sortierten, linearen Liste:

rekursives   
Ausfügen

procedure Ausfuegen (

Liste\_Ptr : in out ELEM\_REF;

Schl : in SCHLUESSEL\_T) is

Hilf\_Ptr : ELEM\_REF := Liste\_Ptr; -- nur für Freigabe

begin

if Liste\_Ptr /= null then

if Schl = Liste\_Ptr.Schl then

Liste\_Ptr := Liste\_Ptr.Next\_Ptr;

Freigabe(Hilf\_Ptr);

else

**Ausfuegen(Liste\_Ptr.Next\_Ptr, Schl);**

end if;

end if;

end Ausfuegen;

Wiederholtes rekursives Löschen aus einer sortierten, linearen Liste:

rekursives

mehrfaches

Ausfügen

procedure AusfuegenWdhg (

Liste\_Ptr : in out ELEM\_REF;

Schl : in SCHLUESSEL\_T) is

Hilf\_Ptr : ELEM\_REF := Liste\_Ptr; -- nur für Freigabe noetig

begin

if Liste\_Ptr /= null then

**AusfuegenWdhg (Liste\_Ptr.Next\_Ptr, Schl);**

if Schl = Liste\_Ptr.Schl then

Liste\_Ptr := Liste\_Ptr.Next\_Ptr;

Freigabe(Hilf\_Ptr);

end if;

end if;

end AusfuegenWdhg;

### Shallow Copy vs Deep Copy – Zwei Kopierverfahren

Auf der Basis von einfachen Listenstrukturen wie oben eingeführt sollen nun Gedanken zu Kopiervorgängen angestellt werden. Dazu wird zunächst ein neuer Listenzeiger deklariert.

Kopie\_der\_Liste : ELEM\_REF;

Die einfachste Form die Liste über den neu eingeführten Zeiger zugänglich zu machen stellt die folgende Anweisung dar:

Kopie\_der\_Liste\_Ptr := Start\_Ptr;

Damit wurde eine sogenannte flache Kopie (respektive Shallow Copy) durchgeführt. Eine Kopie in dieser Art und Weise birgt Vor- und Nachteile. Sicherlich wird diese Anweisung zunächst Speicher- wie auch Laufzeitoptimal sein. Betrachten wir dazu konträr dennoch folgendes Verfahren unter Verwendung der im Unterricht eingeführten Schnittstelle Einfügen\_hinten:

Hilf\_Ptr := Start\_Ptr;

while Hilf\_Ptr /= null loop

Einfuegen\_hinten( Liste => Kopie\_der\_Liste,

Schl => Hilf\_Ptr.Schl,

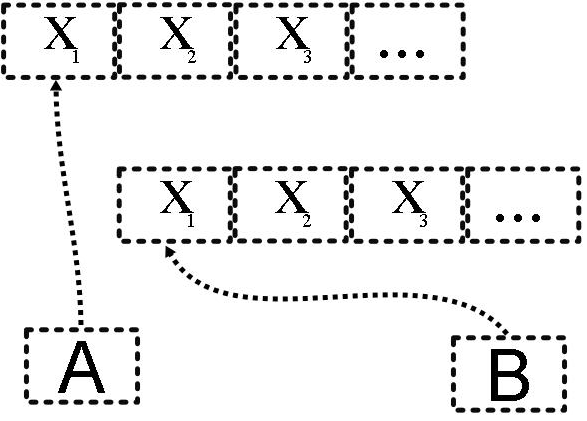
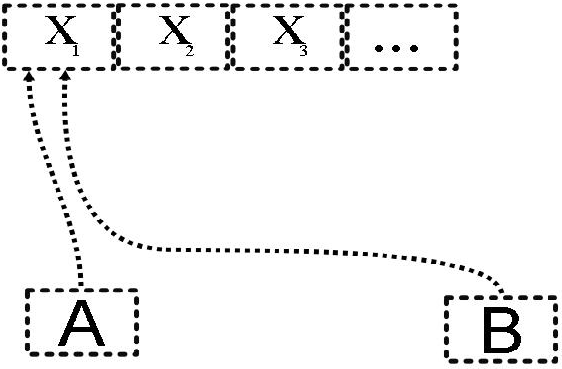
Info => Hilf\_Ptr.Info);

Hilf\_Ptr := Hilf\_Ptr.Next\_Ptr;

end loop;

Mit diesen Anweisungen wurde eine Tiefenkopie (respektive Deep Copy) durchgeführt. Es wurde elementweise eine komplette Kopie der Liste kreiert, die ein eigenständiges Objekt zur Manipulation darstellt.

Eine Grafik die die beiden Verfahren noch einmal voneinander abgrenzt:

Deep Copy Shallow Copy

### Anwendungsbeispiel: Mengen

Auf der Basis einer solchen linearen Listenstruktur und den darauf aufgezeigten grundle­genden Algorithmen können je nach Aufgabenstellung Operationen bzw. Prozeduren festgelegt werden.

Mengen

Lineare Listen lassen sich auch zur Implementierung von Mengen heranziehen. Für das Verwalten von Mengen eines beliebigen Datentyps wäre zum Beispiel fol­gender Aufbau der Liste möglich:



Wir haben hier neben den Listenelemen­ten noch ein Startelement. Dieses Startele­ment beinhaltet neben der Verweiskomponente zusätzlich die In­formation, wie viele Elemente in der Menge enthalten sind.

Mengen­verwaltung



Die mögliche Implementierung einer Mengenverwaltung benutzt oben dargestellte Datenstrukturen in einem generischen Paket. Das Paket stellt nach Ausprägung den abstrakten Datentyp MENGE zur Verfügung. Durch Übergabe von geeigneten Unterprogrammen ist ins­besondere auch die Instanzierung des Pakets mit dem Elementtyp MENGE selbst möglich, so dass sogar Mengen von Mengen (Potenzmengen) verwaltet werden kön­nen.

Die Realisierung des Paketes wird in einer begleitenden Übung gefordert.

## Bäume

Die bisher in diesem Kapitel dargestellten Datenstrukturen waren ausschließlich linear und eignen sich aus diesem Grunde nicht zur Darstellung hierarchischer Abhängigkeiten zwischen Datenelementen, wie z.B. bei der Darstellung von Stammbäumen.

Baum

Aus diesem Grund wird im folgenden Abschnitt die nicht-lineare Datenstruktur Baum vorgestellt.



Ein gerichteter Graph heißt (gerichteter) Baum, wenn gilt:

Wurzel

* Es gibt genau einen Knoten, der keinen Vorgänger hat. Dieser Knoten heißt Wurzel des Baumes.
* Jeder Knoten, mit Ausnahme der Wurzel, hat genau einen Vorgänger.
* Jeder Knoten ist von der Wurzel aus auf genau einem Weg erreichbar.

Für einen Baum gilt:

Ast

* Ein Weg vom Wurzelknoten zu einem Knoten, der keine Nachfolger hat, wird als Ast bezeichnet.

Tiefe

* Die Tiefe eines Baumes ist die Anzahl der Knoten des Asts mit den meisten Knoten.

Teilbaum

* Die Söhne eines gegebenen Knotens bilden die Wurzeln von sogenannten Teil­bäumen.

Blatt

* Ein Knoten ohne weitere Söhne heißt Blatt.

leerer Baum

* Ein Graph ohne Knoten wird als leerer Baum bezeichnet.

Binärbaum



Im Folgenden werden wir als Spezialform von Bäumen den Binärbaum betrachten. Ein Binärbaum ist ein Baum, dessen Knoten maximal zwei Söhne haben können.

Ein Binärbaum ist wie folgt rekursiv definiert:

1. Ein leerer Baum ist ein Binärbaum.
2. Ein Knoten K ist ein Binärbaum.
3. Sind B1 und B2 Binärbäume und K ein Knoten, dann ist auch folgen­der Graph ein Binärbaum:



Alle rekursiven Algorithmen auf Bäume basieren auf dieser abstrakten Betrachtungsweise.

### Speicherung von Binärbäumen



Ein Binärbaum kann folgendermaßen implementiert werden:



Jeder Knoten (Vater) des Baumes wird durch ein dynamisches Daten­objekt darge­stellt, das Informationen und jeweils einen Verweis auf jeden der beiden mög­lichen Nachfolger (Söhne) ent­hält. Ist kein Sohn mehr vorhanden, ist die jeweilige Zugriffskomponente auf null gesetzt.

Deklarationen für Binärbäume



type KNOTEN\_TR;

type KNOTEN\_REF is access KNOTEN\_TR;

type KNOTEN\_TR is record

Schl : SCHLUESSEL\_T; -- Sortierkriterium z.B. POSITIVE

Info : INFORMATION\_T; -- Zusatzmerkmal z.B. CHARACTER

Links\_Ptr,

Rechts\_Ptr : KNOTEN\_REF;

end record;

geordneter Binärbaum



Man spricht von einem **geordneten** Binärbaum, wenn die Schlüssel der Söhne eines Knotens in einer geordneten Beziehung zueinander und zum Knoten selbst stehen. Dies ist z.B. der Fall, wenn der linke Sohn kleiner oder gleich dem Vaterknoten ist und dieser kleiner als sein rechter Sohn ist.



Wir werden uns im Folgenden ausschließlich mit geordneten Binärbäumen beschäfti­gen.

### Operationen auf Binärbäume

Durchlaufen

Beim Durchlaufen eines (geordneten) Binärbaums wird, je nach der konkreten Aufgabenstellung, entweder der gesamte Baum (also jeder Knoten) oder nur ein Teil eines Baumes durchlaufen. Da wir entgegen dem Durchlauf-Algorithmus bei Listen bei Binärbäumen zwei Verweise auf Nachfolgeknoten haben, kommen wir hier auf die oben angesprochene rekursive Be­trachtungsweise von Bäumen zurück.

rekursives Durchlaufen eines Baums

generic

with procedure Aktion (

Schl : in SCHLUESSEL\_T;

Info : in out INFORMATION\_T) is <>;

procedure Durchlaufen (Wurzel\_Ptr : in KNOTEN\_REF);



procedure Durchlaufen (Wurzel\_Ptr : KNOTEN\_REF) is

begin -- Durchlaufen

if Wurzel\_Ptr /= null then

Durchlaufen (Wurzel\_Ptr.All.Links\_Ptr);

Aktion (Wurzel\_Ptr.All.Schl, Wurzel\_Ptr.All.Info);

Durchlaufen (Wurzel\_Ptr.All.Rechts\_Ptr);

end if;

end Durchlaufen;

In-Order-Durchlauf



In-Order-Durchlauf



Diese Art des Durchlaufens eines Binärbaums wird auch als In-Order-Durchlauf bezeichnet, da jeweils der linke Teilbaum zuerst besucht wird, dann der Knoten bearbeitet und anschließend der rechte Teilbaum be­sucht wird. Bei einem geordneten Binärbaum werden bei dieser Durchlauf­form die Knoten in Sortierreihenfolge bearbeitet.

Entsprechend sind die folgenden Durchlaufformen definiert:

Pre-Order-Durchlauf

* Pre-Order-Durchlauf

Zuerst wird der Knoten bearbeitet, dann wird der linke Teil­baum und anschließend der rechte Teilbaum besucht.

Nützlich zum Erstellen einer 1:1 Kopie (tiefe Kopie) des Baumes.

Aktion (Wurzel\_Ptr.Schl, Wurzel\_Ptr.Info);

Durchlaufen (Wurzel\_Ptr.Links\_Ptr);

Durchlaufen (Wurzel\_Ptr.Rechts\_Ptr);



Post-Order-Durchlauf

* Post-Order-Durchlauf

Zuerst wird der linke Teilbaum und anschließend der rechte Teilbaum besucht und danach erst der Knoten bearbeitet.

Nützlich zum Löschen des gesamten Baumes, da nur Blätter gelöscht werden müssen.

Durchlaufen (Wurzel\_Ptr.Links\_Ptr);

Durchlaufen (Wurzel\_Ptr.Rechts\_Ptr);

Aktion (Wurzel\_Ptr.Schl, Wurzel\_Ptr.Info);



Einfügen in Binärbaum



Beim Einfügen von Knoten in einen geordneten Binärbaum müssen die Knoten gemäß einem Schlüssel (Sortierkriterium) so eingeordnet werden, dass die gegebene Ordnung gewahrt bleibt.

Wir wollen unsere zugrundelie­gende Ordnung definieren, die einen geordneten Binärbaum ausmacht:





Als neue Zahl soll die 37 eingefügt werden.





Beim Einfügen wird von der Wurzel ausgehend entschieden, ob das Suchen im linken oder rechten Teilbaum fortgesetzt wird. Ist der Schlüssel­wert des einzufügenden Knotens kleiner gleich dem der Wur­zel des aktuellen Teilbaums, wird in den linken Teilbaum gewechselt, ansonsten in den rechten. Dort wird nach dem gleichen Verfahren weiter entschieden. Dieses Suchverfahren sucht rekursiv weiter, bis kein weiterer Teilbaum mehr vorhanden ist. Ist der aktuelle Zeigerwert null, ist die Einfügestelle gefunden und dieser (ehemals null-) Zeiger wird mit einem Verweis auf den neuen Knoten besetzt. Für unseren speziellen Fall erfolgt das Einfügen über die Knoten mit den Zahlen **38**, **14**, **23**.

Einfügen in   
Binärbaum



procedure Einfuegen (

Wurzel\_Ptr : in out KNOTEN\_REF;

Schl : in SCHLUESSEL\_T;

Info : in INFORMATION\_T) is

begin -- Einfuegen

if Wurzel\_Ptr = null then -- Einfuegestelle gefunden

Wurzel\_Ptr := new KNOTEN\_TR'(Schl, Info, null, null);

else -- Weitersuchen

if Schl <= Wurzel\_Ptr.Schl then -- im linken Teilbaum

Einfuegen (Wurzel\_Ptr.Links\_Ptr, Schl, Info);

else -- im rechten Teilbaum

Einfuegen (Wurzel\_Ptr.Rechts\_Ptr, Schl, Info);

end if;

end if;

end Einfuegen;

Löschen im Binärbaum



Das Entfernen eines Knotens aus einem Baum ist nicht so einfach, da hier Beson­derheiten auftreten können, die eine unterschiedliche Behandlung erfordern. Auf Grund dieser Besonderheit gehen wir beim Löschen von einem eindeutigen Schlüssel aus.

Zuerst muss das zu löschende Element im Baum gesucht werden (z.B. durch In-Order-Durchlauf mit Vergleich der Schlüssel). Haben wir das auszufügende Element gefunden, sind für das weitere Vorgehen drei Fälle zu unterscheiden:

Löschknoten hat

keinen Nachfolger

* das Element besitzt keinen Nachfolger (Blatt):

Der entsprechende Zeiger (linker oder rechter Verweis) des Vorgängerknotens wird durch null ersetzt.

einen Nachfolger

* das Element besitzt einen Nachfolger:

Der entsprechende Zeiger des Vorgängerknotens wird durch den Verweis auf diesen einen Nachfolgerknoten ersetzt.

zwei Nachfolger

* das Element besitzt zwei Nachfolger:

In diesem Fall ersetzen wir das zu entfernende Element, indem wir es durch das kleinste Element des rechten (bzw. größte Element des linken) Teilbaums ersetzen. Dieses Element hat höchstens einen Nachfolger.

Wieder wollen wir die Vorgehensweise an einem Beispiel illustrieren:



Der Knoten mit dem Schlüssel 14 soll ausgefügt werden. Er wird, da er genau zwei Nachfolger besitzt, durch das kleinste Element des rechten Teilbaums (Schlüssel 18) ersetzt.



procedure Loeschen (

Wurzel\_Ptr : in out KNOTEN\_REF;

Schl : in SCHLUESSEL\_T) is

Hilf\_Ptr : KNOTEN\_REF := Wurzel\_Ptr;

begin -- Loeschen

if Wurzel\_Ptr /= null then

if Schl < Wurzel\_Ptr.Schl then

-- Weitersuchen im linken Teilbaum

Loeschen (Wurzel\_Ptr.Links\_Ptr, Schl);

elsif Schl > Wurzel\_Ptr.Schl then

-- Weitersuchen im rechten Teilbaum

Loeschen (Wurzel\_Ptr.Rechts\_Ptr, Schl);

else -- Knoten gefunden und loeschen

if Wurzel\_Ptr.Links\_Ptr = null and   
 Wurzel\_Ptr.Rechts\_Ptr = null then

-- das Element besitzt keine Nachfolger

Freigabe(Wurzel\_Ptr);

elsif Wurzel\_Ptr.Rechts\_Ptr = null then

-- nur linker Sohn vorhanden

Wurzel\_Ptr := Wurzel\_Ptr.Links\_Ptr;

Freigabe(Hilf\_Ptr);

elsif Wurzel\_Ptr.Links\_Ptr = null then

-- nur rechter Sohn vorhanden

Wurzel\_Ptr := Wurzel\_Ptr.Rechts\_Ptr;

Freigabe(Hilf\_Ptr);

else -- zwei Soehne vorhanden

-- 1. in rechten Teilbaum gehen

Hilf\_Ptr := Wurzel\_Ptr.Rechts\_Ptr;

-- 2. dort kleinstes Element finden

while Hilf\_Ptr.Links\_Ptr /= null loop

Hilf\_Ptr := Hilf\_Ptr.Links\_Ptr;

end loop;

-- 3. Inhalt des kleinsten Elementes uebertragen

Wurzel\_Ptr.Schl := Hilf\_Ptr.Schl;

Wurzel\_Ptr.Info := Hilf\_Ptr.Info;

-- 4. kleinstes Element im rechten Teilbaum löschen

-– (hat höchstens 1 Nachfolger)

Loeschen (Wurzel\_Ptr.Rechts\_Ptr, Hilf\_Ptr.Schl);

end if;

end if;

end if;

end Loeschen;

## Graphen

Graph =

Knoten +

Kanten



Ein Graph ist anschaulich gesprochen ein Gebilde, das aus Punkten bzw. Knoten und Linien bzw. Kanten besteht, welche entweder zwei Knoten oder einen Knoten mit sich selbst (Schlinge) verbinden. Ist den Kanten eine Orientierung zugewiesen, heißen sie gerichtete Kanten oder Pfeile und der Graph wird als gerichteter Graph bezeichnet.

Wir wollen uns im Folgenden ausschließlich mit gerichteten Graphen beschäfti­gen.

Gerichteter Graph

Mathematisch (und wie wir später noch sehen werden auch programmtechnisch) lässt sich ein gerichteter Graph *G* darstellen durch:

* die Menge der Knoten *V* (engl. vertices) und
* die Menge der Kanten *E* (engl. edges), wobei jeder Kante   
  ge­nau ein geordnetes Knotenpaar zu­geordnet ist.

Beispiel

Der durch die Mengen



festgelegte gerichtete Graph sieht folgendermaßen aus:



Die Knoten und Kanten von Graphen können zusätzlich noch mit Informationen versehen werden.

Beispiele für die Anwendung von Graphen sind:

* Entfernungstabellen
* Projektplanung
* Simulation des Betriebes von Rechenanlagen

### Speicherung von Graphen

Im Folgenden wird die Speicherung von gerichteten Graphen mit Hilfe   
dy­namischer Datenstrukturen vorgestellt. Dabei sind zum einen Knoten und zum anderen Kanten abzuspeichern. Beides erfolgt in Form von Listen. Aus Gründen der Abstraktion wird außerdem noch der Datentyp GRAPH\_T eingeführt.



Die Knoten eines Graphen werden als einfach verkettete Liste dargestellt. Dabei enthält jeder Knoten einen Verweis auf eine Kantenliste der Kanten, dessen Start­knoten er ist. In der Kantenliste selbst werden die Kanten durch Verweise auf die Zielknoten repräsentiert.



Die Bedeutung der einzelnen Komponenten sind den entsprechenden Typdefinitionen zu entnehmen (die Reihenfolge entspricht der der grafischen Darstellung):

type KNOTEN\_TR;

type KNOTEN\_REF is access KNOTEN\_TR;

type KANTEN\_TR;

type KANTEN\_REF is access KANTEN\_TR;

type KNOTEN\_TR is record

Nummer : KNOTEN\_NUMMERN\_T; -- Knotennummer zur

-- Identifikation.

Info : KNOTEN\_INFO\_T; -- Informationsmenge.

Markierung : NATURAL := 0; -- fuer verschiedene

-- Algorithmen muessen

-- Knoten markiert werden.

Kantenliste\_Ptr : KANTEN\_REF; -- Verweis auf das erste

-- Element der Kantenliste,

-- dessen Startknoten dieser

-- Knoten ist.

Next\_Ptr : KNOTEN\_REF; -- naechster Knoten in

-- der Knotenliste.

end record;

type KANTEN\_TR is record

Endknoten\_Ptr : KNOTEN\_REF; -- Verweis auf den Endknoten

-- der Kante in Knotenliste.

Info : KANTEN\_INFO\_T; -- Informationsmenge.

Markierung : NATURAL := 0; -- fuer verschiedene

-- Algorithmen muessen

-- Kanten markiert werden.

Next\_Ptr : KANTEN\_REF; -- naechste Kante in

-- der Kantenliste.

end record;

subtype GRAPH\_T is KNOTEN\_REF;

### Operationen auf Graphen

* Einfügen von Knoten

Einfügen von

Knoten



Die Knoten eines Graphen werden in Form einer Liste abgespei­chert. Der Vorgang des Einfügens eines Knotens verläuft analog zum unsortierten Einfügen in eine verkettete Liste. Eine Sor­tierung muss nicht be­achtet werden, allerdings darf der Knoten noch nicht vorhanden sein.

procedure Einfuegen (

In\_Den\_Graph : in out GRAPH\_T;

Nummer : in KNOTEN\_NUMMERN\_T;

Info : in KNOTEN\_INFO\_T) is

Aktueller\_Knoten\_Ptr : KNOTEN\_REF := In\_Den\_Graph;

begin -- Einfuegen

-- Ueberpruefen, ob Knoten bereits vorhanden ist:

while Aktueller\_Knoten\_Ptr /= null loop

if Aktueller\_Knoten\_Ptr.Nummer = Nummer then

raise Knoten\_Bereits\_Vorhanden;

end if;

Aktueller\_Knoten\_Ptr := Aktueller\_Knoten\_Ptr.Next\_Ptr;

end loop;

-- Knoten vor die Knotenliste vorhaengen:

In\_Den\_Graph := new KNOTEN\_TR'(

Next\_Ptr => In\_Den\_Graph,

Nummer => Nummer,

Info => Info,

Markierung => 0,

Kantenliste\_Ptr => null);

end Einfuegen;

* Einfügen von Kanten

Einfügen von

Kanten



Beim Einfügen einer Kante wird zunächst der Startknoten der neuen Kante gesucht und anschließend der Endknoten. Dann wird in die Kantenliste des Startknotens ein neues Listenelement eingefügt. Hier muss ebenfalls keine Sortierung beachtet werden. Die Eindeu­tigkeit der Kante muss aber überprüft werden.

procedure Einfuegen (

In\_Den\_Graph : in GRAPH\_T;

Start\_Knoten\_Nummer,

End\_Knoten\_Nummer : in KNOTEN\_NUMMERN\_T;

Info : in KANTEN\_INFO\_T) is

Start\_Knoten\_Ptr,

End\_Knoten\_Ptr : KNOTEN\_REF;

Aktueller\_Knoten\_Ptr : KNOTEN\_REF := In\_Den\_Graph;

Aktuelle\_Kante\_Ptr : KANTEN\_REF;

begin -- Einfuegen

-- Start- und Endknoten suchen:

while Aktueller\_Knoten\_Ptr /= null loop

if Aktueller\_Knoten\_Ptr.Nummer = Start\_Knoten\_Nummer then

Start\_Knoten\_Ptr := Aktueller\_Knoten\_Ptr;

end if;

if Aktueller\_Knoten\_Ptr.Nummer = End\_Knoten\_Nummer then

End\_Knoten\_Ptr := Aktueller\_Knoten\_Ptr;

end if;

Aktueller\_Knoten\_Ptr := Aktueller\_Knoten\_Ptr.Next\_Ptr;

end loop;

if Start\_Knoten\_Ptr = null or End\_Knoten\_Ptr = null then

raise Knoten\_Nicht\_Vorhanden;

end if;

-- Ueberpruefen, ob Kante bereits vorhanden :

Aktuelle\_Kante\_Ptr := Start\_Knoten\_Ptr.Kantenliste\_Ptr;

while Aktuelle\_Kante\_Ptr /= null loop

if Aktuelle\_Kante\_Ptr.Endknoten\_Ptr = End\_Knoten\_Ptr then

raise Kante\_Bereits\_Vorhanden;

end if;

Aktuelle\_Kante\_Ptr := Aktuelle\_Kante\_Ptr.Next\_Ptr;

end loop;

-- neue Kante vorn in die Kantenliste des Startknoten

-- einhaengen:

Start\_Knoten\_Ptr.Kantenliste\_Ptr := new KANTEN\_TR'(

Endknoten\_Ptr => End\_Knoten\_Ptr,

Info => Info,

Markierung => 0,

Next\_Ptr => Start\_Knoten\_Ptr.Kantenliste\_Ptr);

end Einfuegen;

* Ausfügen von Knoten

Ausfügen von

Knoten

Beim Ausfügen eines Knotens müssen auch alle Kanten entfernt werden, die den auszufügenden Knoten als Start- oder Endknoten haben. Vorher müssen noch die Freigabe-Prozeduren festgelegt werden.

Freigabe-Prozeduren

procedure Freigabe\_Knoten is

new Unchecked\_Deallocation(KNOTEN\_TR, KNOTEN\_REF);

procedure Freigabe\_Kante is

new Unchecked\_Deallocation(KANTEN\_TR, KANTEN\_REF);

procedure Alle\_Kanten\_Loeschen(  
 Kanten\_Listen\_Ptr : in out KANTEN\_REF) is

Hilfs\_Ptr : KANTEN\_REF;

begin

while Kanten\_Listen\_Ptr /= null loop

Hilfs\_Ptr := Kanten\_Listen\_Ptr;

Kanten\_Listen\_Ptr := Kanten\_Listen\_Ptr.Next\_Ptr;

-- Kante wird geloescht

Freigabe\_Kante(Hilfs\_Ptr);

end loop;

end Alle\_Kanten\_Loeschen;

procedure Ausfuegen (

Aus\_Dem\_Graph : in out GRAPH\_T;

Nummer : in KNOTEN\_NUMMERN\_T) is

Ausfuege\_Knoten\_Ptr,

Vorgaenger\_Knoten\_Ptr : KNOTEN\_REF := Aus\_Dem\_Graph;

Aktueller\_Knoten\_Ptr : KNOTEN\_REF;

Ausfuege\_Kante\_Ptr,

Vorgaenger\_Kante\_Ptr : KANTEN\_REF;

begin -- Ausfuegen

-- Knoten suchen:

while Ausfuege\_Knoten\_Ptr /= null

and then Ausfuege\_Knoten\_Ptr.Nummer /= Nummer loop

Vorgaenger\_Knoten\_Ptr := Ausfuege\_Knoten\_Ptr;

Ausfuege\_Knoten\_Ptr := Ausfuege\_Knoten\_Ptr.Next\_Ptr;

end loop;

if Ausfuege\_Knoten\_Ptr = null then

raise Knoten\_Nicht\_Vorhanden;

end if;

-- Knoten ausketten; damit sind auch alle Kanten entfernt,

-- die diesen Knoten als Startknoten haben:

if Ausfuege\_Knoten\_Ptr = Vorgaenger\_Knoten\_Ptr then

Aus\_Dem\_Graph := Aus\_Dem\_Graph.Next\_Ptr;

else

Vorgaenger\_Knoten\_Ptr.Next\_Ptr:= Ausfuege\_Knoten\_Ptr.Next\_Ptr;

end if;

-- Alle Kanten aus Ausfuege-Knoten ausfuegen und loeschen

Alle\_Kanten\_Loeschen(Ausfuege\_Knoten\_Ptr.Next\_Ptr);

-- in allen Kantenlisten die Kanten ausfuegen, die den

-- Knoten als Endknoten haben:

Aktueller\_Knoten\_Ptr := Aus\_Dem\_Graph;

while Aktueller\_Knoten\_Ptr /= null loop -- alle Knoten

-- abarbeiten

-- fuer jeden Knoten die Kantenliste pruefen:

Ausfuege\_Kante\_Ptr := Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr;

Vorgaenger\_Kante\_Ptr := Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr;

-- Suche nach Kanten, die den auszufuegenden Knoten als

-- Endknoten haben:

while Ausfuege\_Kante\_Ptr /= null loop

if Ausfuege\_Kante\_Ptr.Endknoten\_Ptr = Ausfuege\_Knoten\_Ptr   
 then

-- Kante ausfuegen

if Vorgaenger\_Kante\_Ptr = Ausfuege\_Kante\_Ptr then

Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr :=

Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr.Next\_Ptr;

else

Vorgaenger\_Kante\_Ptr.Next\_Ptr :=   
 Ausfuege\_Kante\_Ptr.Next\_Ptr;

end if;

-- Kante wird geloescht

Freigabe\_Kante(Ausfuege\_Kante\_Ptr);

Ausfuege\_Kante\_Ptr := Vorgaenger\_Kante\_Ptr.Next\_Ptr;

else

Vorgaenger\_Kante\_Ptr := Ausfuege\_Kante\_Ptr;

Ausfuege\_Kante\_Ptr := Ausfuege\_Kante\_Ptr.Next\_Ptr;

end if;

end loop;

Aktueller\_Knoten\_Ptr := Aktueller\_Knoten\_Ptr.Next\_Ptr;

end loop;

-- Knoten wird geloescht

Freigabe\_Knoten(Ausfuege\_Knoten\_Ptr);

end Ausfuegen;

* Ausfügen von Kanten

Ausfügen von

Kanten



Zunächst wird der Startknoten der auszufügenden Kante gesucht. Anschließend wird in seiner Kantenliste nach der Kante gesucht, die auf den entsprechenden End­knoten verweist. Diese Kante wird ausgefügt.

procedure Ausfuegen (

Aus\_Dem\_Graph : in out GRAPH\_T;

Start\_Knoten\_Nummer,

End\_Knoten\_Nummer : in KNOTEN\_NUMMERN\_T) is

Aktueller\_Knoten\_Ptr : KNOTEN\_REF := Aus\_Dem\_Graph;

Ausfuege\_Kante\_Ptr,

Vorgaenger\_Kante\_Ptr : KANTEN\_REF;

begin -- Ausfuegen

-- Startknoten suchen:

while Aktueller\_Knoten\_Ptr /= null and then

Aktueller\_Knoten\_Ptr.Nummer /= Start\_Knoten\_Nummer loop

Aktueller\_Knoten\_Ptr := Aktueller\_Knoten\_Ptr.Next\_Ptr;

end loop;

if Aktueller\_Knoten\_Ptr = null then

raise Kante\_Nicht\_Vorhanden;

end if;

-- Endknoten in der Kantenliste des Startknotens suchen:

Ausfuege\_Kante\_Ptr := Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr;

Vorgaenger\_Kante\_Ptr := Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr;

while Ausfuege\_Kante\_Ptr /= null and then

Ausfuege\_Kante\_Ptr.Endknoten\_Ptr.Nummer /=

End\_Knoten\_Nummer loop

Vorgaenger\_Kante\_Ptr := Ausfuege\_Kante\_Ptr;

Ausfuege\_Kante\_Ptr := Ausfuege\_Kante\_Ptr.Next\_Ptr;

end loop;

-- Kante ausfuegen, aber nur, falls vorhanden:

if Ausfuege\_Kante\_Ptr = null then

raise Kante\_Nicht\_Vorhanden;

elsif Ausfuege\_Kante\_Ptr = Vorgaenger\_Kante\_Ptr then -- 1. Kante

Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr :=

Aktueller\_Knoten\_Ptr.Kantenliste\_Ptr.Next\_Ptr;

else

Vorgaenger\_Kante\_Ptr.Next\_Ptr := Ausfuege\_Kante\_Ptr.Next\_Ptr;

end if;

-- Kante wird geloescht

Freigabe(Ausfuege\_Kante\_Ptr);

end Ausfuegen;

### Autorouter

Straßenkarte

Graphen bestehen aus einer Menge von Knoten und einer Menge von Kanten. Wie man Knoten und Kanten deutet, hängt von der praktischen Anwendung ab. Eine mögliche Anwendung ist zum Beispiel die Speicherung einer Straßenkarte: Knoten sind dann Städte, Straßenkreuzungen, Straßenauf- und -abfahrten, u.ä., Kanten ihrerseits stellen Autobahnen, Bundesstraßen u.ä. dar. Dazu muss man die Datenstruktur GRAPHEN aus den vorigen Kapiteln nur ein wenig ergänzen und anpassen. Es werden Namen für Knoten und Kanten, eine Entfernungsangabe für die Kanten und eine Koordinatenangabe für die (später vielleicht implementierte) Graphikdarstellung der Knoten eingeführt werden. Zur Vereinfachung soll jede Kante zusätzlich einen Zeiger auf den Startknoten enthalten. Somit ergibt sich folgende Änderung in der Datenstruktur:

type STRING\_REF is access STRING; -- fuer Strassen-/ Ortsnamen

type KILOMETER\_T is new NATURAL; -- fuer Strassenlaenge

type KOORDINATEN\_TR is record -- Koordinate der Orte

X, Y : NATURAL := 0;

end record;

type KNOTEN\_TR is record

Name\_Ptr : STRING\_REF;

Koordinate : KOORDINATEN\_T;

Kantenliste\_Ptr : KANTEN\_REF;

Next\_Ptr : KNOTEN\_REF;

end record;

type KANTEN\_TR is record

Name\_Ptr : STRING\_REF;

Laenge : KILOMETER\_T;

Endknoten\_Ptr : KNOTEN\_REF;

Startknoten\_Ptr : KNOTEN\_REF;

Next\_Ptr : KANTEN\_REF;

end record;

In eine solche Datenstruktur GRAPHEN lassen sich theoretisch beliebig viele Knoten und davon abgehende Kanten aufnehmen, so dass man im Prinzip ein komplettes Straßennetz abbilden kann.

Bestimmung des kürzesten Wegs

Sind in einem solchen Straßennetz alle Angaben aufgenommen, spricht nichts dagegen, alle Wege oder sinnvoller, den kürzesten Weg zwischen zwei verschiedenen Städten zu bestimmen. Im Handel existieren bereits diverse kommerzielle Lösungen dieses Problems (Autoroute express, Map&Guide u.ä.).



Entscheidungs­baumverfahren

Wir wollen im Folgenden die Bestimmung des optimalen Wegs zwischen zwei Knoten selbst durchführen. Dazu geeignet ist das Entscheidungsbaumverfahren (siehe Teil 3), welches aber noch auf die Eigenheiten einer dynamischen Datenstruktur angepasst werden muss. Unser ursprüngliches Verfahren basiert auf diskreten Entscheidungen, die auf diskreten Stufen rekursiv gefällt werden, wobei aber unzulässige und nicht sinnvolle Teilausschnitte des Entscheidungsbaums aus der Suche nach der optimalen Lösung ausgeschlossen werden.

Im Fall des Straßengraphen sind die Entscheidungen die Straßen, die ein Weg umfassen soll. Eine Stufe ist dabei als die Nummer der nächsten Entscheidung zu betrachten. Dabei wird für jede Stufe die Entscheidung getroffen, welcher Kantenzeiger als nächster in die Lösungsmenge aufzunehmen ist.

Lösungsmenge

Die Lösungsmenge ist also eine Reihung aus Kantenzeigern:



zulässig,  
noch sinnvoll

Eine Teillösung ist dann zulässig, wenn keine Kante und auch kein Knoten mehrfach vorhanden ist. Eine Teillösung ist dann noch sinnvoll, wenn die Summe der Teilstreckenentfernungen der Teillösung besser als die der optimalen Lösung ist.

Im folgenden Listing wird ein wie oben beschriebener Graph vereinbart und mit einer Reihe von Hilfsoperationen versehen. Die Straßennetzdaten werden aus einer Textdatei geladen, was aber erst im nächsten Ausbildungsteil von Interesse ist.

with Ada.Text\_IO;

procedure Autorouter is

--------------------------------------------------------------

-- Definitionen fuer den Datentyp GRAPHEN:

--------------------------------------------------------------

type KNOTEN\_TR; -- Staedte, Kreuzungen,...

type KNOTEN\_REF is access KNOTEN\_TR;

type KANTEN\_TR; -- Autobahnen, Bundesstraßen

type KANTEN\_REF is access KANTEN\_TR;

subtype GRAPH\_T is KNOTEN\_REF;

type STRING\_REF is access STRING; -- Knoten und Kanten besitzen

-- einen Namen als Kennzeichnung

type KOORDINATEN\_TR is record -- auf einer Landkarte hat jeder X, Y : NATURAL := 0; -- Knoten eine Punktkoordinate (X/Y).

end record;

type KILOMETER\_T is new NATURAL; -- Kanten besitzen eine Laenge

-- in km.

type KNOTEN\_TR is record

Name\_Ptr : STRING\_REF;

Koordinate : KOORDINATEN\_T;

Kantenliste\_Ptr : KANTEN\_REF;

Next\_Ptr : KNOTEN\_REF;

end record;

type KANTEN\_TR is record

Name\_Ptr : STRING\_REF;

Laenge : KILOMETER\_T;

Endknoten\_Ptr : KNOTEN\_REF;

Startknoten\_Ptr : KNOTEN\_REF;

Next\_Ptr : KANTEN\_REF;

end record;

Strassennetz\_Ptr : GRAPH\_T;

Knoten\_Fehler : exception;

--------------------------------------------------------------

-- Definitionen fuer das Entscheidungsbaumverfahren:

--------------------------------------------------------------

type LOESUNGSMENGE\_TA is array (POSITIVE range <>) of

KANTEN\_REF;

-- fuer jede Stufe wird Entscheidung gefaellt, welche Kante

-- als naechstes in die Loesung mit aufgenommen wird.

Loesungs\_Fehler : exception;

--------------------------------------------------------------

-- Funktion, die in einem Graph den Knoten

-- mit einem bestimmten Namen sucht:

--------------------------------------------------------------

function Knoten\_Ptr (G : GRAPH\_T; Name : STRING)

return KNOTEN\_REF is

Hilf\_Knoten\_Ptr : KNOTEN\_REF := G;

begin

while Hilf\_Knoten\_Ptr /= null and then   
 Hilf\_Knoten\_Ptr.Name\_Ptr.All/= Name loop

Hilf\_Knoten\_Ptr := Hilf\_Knoten\_Ptr.Next\_Ptr;

end loop;

return Hilf\_Knoten\_Ptr;

end Knoten\_Ptr;

--------------------------------------------------------------

-- Prozedur zum Einfuegen einer Kante mit Namen und Laenge

-- zwischen zwei namentlich bekannten Knoten in einem Graphen:

--------------------------------------------------------------

procedure Kante\_Einfuegen (

G : in out GRAPH\_T;

Von,

Nach : in STRING;

Name : in STRING := "";

Laenge : in KILOMETER\_T := 0) is

Von\_Knoten\_Ptr : KNOTEN\_REF := Knoten\_Ptr(G, Von);

Nach\_Knoten\_Ptr : KNOTEN\_REF := Knoten\_Ptr(G, Nach);

begin

if Von\_Knoten\_Ptr = null or Nach\_Knoten\_Ptr = null then

raise Knoten\_Fehler;

end if;

Von\_Knoten\_Ptr.Kantenliste\_Ptr :=

new KANTEN\_TR'(Name\_Ptr =>new STRING'(Name),

Laenge => Laenge,

Endknoten\_Ptr => Nach\_Knoten\_Ptr,

Startknoten\_Ptr => Von\_Knoten\_Ptr,

Next\_Ptr => Von\_Knoten\_Ptr.Kantenliste\_Ptr);

end Kante\_Einfuegen;

--------------------------------------------------------------

-- Prozedur zum Einfuegen eines Knoten mit Namen und

-- KOORDINATEN\_TR in einen Graphen:

--------------------------------------------------------------

procedure Knoten\_Einfuegen (

G : in out GRAPH\_T;

Name : in STRING := "";

Koordinate : in KOORDINATEN\_TR := (0, 0)) is

begin –- Knoten\_Einfuegen

if Knoten\_Ptr(G, Name) = null then

G := new KNOTEN\_TR'(Name\_Ptr => new STRING'(Name),

Koordinate => Koordinate,

Kantenliste\_Ptr => null,

Next\_Ptr => G);

end if;

end Knoten\_Einfuegen;

--------------------------------------------------------------

-- Prozedur zum Laden eines Graphen aus einer Textdatei:

-- Textdateien und ihre Behandlung sind Inhalt von Teil 7:

--------------------------------------------------------------

procedure Graph\_Laden (

G : in out GRAPH\_T;

Name : in STRING) is separate;

--------------------------------------------------------------

-- zum Ausgeben einer LOESUNGSMENGE\_TA von Kanten in Form:

-- 1.: B17 von Augsburg nach Landsberg - 20 km

-- 2.: B12/A96 von Landsberg nach Muenchen - 60 km

-- Entfernung: 80 km

--------------------------------------------------------------

procedure Put (L : in LOESUNGSMENGE\_TA) is

Km : KILOMETER\_T := 0;

begin

Ada.Text\_IO.New\_Line;

Ada.Text\_IO.Put\_Line ("Optimale Wegstrecke:");

for I in L'Range loop

Ada.Text\_IO.Put\_Line

(INTEGER'Image(I) & ".: " & L(I).Name\_Ptr.All &

" von " & L(I).Startknoten\_Ptr.Name\_Ptr.All &

" nach " & L(I).Endknoten\_Ptr.Name\_Ptr.All &

" -" & KILOMETER\_T'Image(L(I).Laenge) & " km" );

Km := Km + L(I).Laenge;

end loop;

Ada.Text\_IO.Put\_Line ("Entfernung:" & KILOMETER\_T'Image(Km)

& " km");

Ada.Text\_IO.New\_Line;

end Put;

--------------------------------------------------------------

-- Funktion Optimal\_Weg, die durch Entscheidungsbaumverfahren

-- aus einem gegebenen Graphen den kuerzesten Weg zwischen 2

-- namentlich bekannten Knoten bestimmt. Dabei sollen keine

-- Wege mit mehr als Max\_Kanten Kanten untersucht werden:

--------------------------------------------------------------

function Optimalweg (

G : GRAPH\_T;

Von,

Nach : STRING;

Max\_Kanten : POSITIVE := 30)

return LOESUNGSMENGE\_TA is

Akt\_Loesung : LOESUNGSMENGE\_TA (1 .. Max\_Kanten);

Opt\_Loesung : LOESUNGSMENGE\_TA (1 .. Max\_Kanten);

Opt\_Last : NATURAL := Opt\_Loesung'First - 1;

Opt\_Wert : KILOMETER\_T;

Lsg\_Gefunden : BOOLEAN := FALSE;

Von\_Knoten\_Ptr : KNOTEN\_REF := Knoten\_Ptr(G, Von);

Nach\_Knoten\_Ptr : KNOTEN\_REF := Knoten\_Ptr(G, Nach);

-----------------------------------------------------------

function Wert (L : LOESUNGSMENGE\_TA) return KILOMETER\_T is

Km : KILOMETER\_T := 0;

begin

for I in L'Range loop

Km := Km + L(I).Laenge;

end loop;

return Km;

end Wert;

-----------------------------------------------------------

function Zulaessig (L : LOESUNGSMENGE\_TA) return BOOLEAN is

Ist\_Zulaessig : BOOLEAN := True;

begin

if L'Length > 1 then

-- keine Kante darf mehr als einmal vorkommen:

for I in L'First .. L'Last - 1 loop

if L(I) = L(L'Last) then

Ist\_Zulaessig := False;

end if;

end loop;

-- kein Knoten darf mehr als einmal vorkommen,

-- weder der Von-Knoten:

if L(L'Last).Endknoten\_Ptr = Von\_Knoten\_Ptr then

Ist\_Zulaessig := False;

end if;

-- ... noch sonst ein Zwischenknoten auf dem Weg:

if Ist\_Zulaessig then

for I in L'First .. L'Last-1 loop

if L(I).Endknoten\_Ptr = L(L'Last).Endknoten\_Ptr   
 then

Ist\_Zulaessig := False;

end if;

end loop;

end if;

end if;

return Ist\_Zulaessig;

end Zulaessig;

-----------------------------------------------------------

function Sinnvoll (L : LOESUNGSMENGE\_TA) return BOOLEAN is

begin

-- eine Loesung L ist sinnvoll, wenn sie nicht eine

-- groessere Laenge hat als die optimale Loesung:

return not Lsg\_Gefunden or else Wert(L) < Opt\_Wert;

end Sinnvoll;

-----------------------------------------------------------

procedure Treffe\_Entscheidung (

Fuer\_Ptr : in KNOTEN\_REF;

Stufe : in POSITIVE) is

Akt\_Kante\_Ptr : KANTEN\_REF := Fuer\_Ptr.Kantenliste\_Ptr;

begin

-- Fuer den uebergebenen Knoten wird sukzessive jede

-- abgehende Kante auf Zulaessigkeit und Zielerreichung

-- ueberprueft:

while Akt\_Kante\_Ptr /= null loop

Akt\_Loesung (Stufe) := Akt\_Kante\_Ptr;

if Zulaessig (Akt\_Loesung(1 .. Stufe)) and then

Sinnvoll (Akt\_Loesung(1 .. Stufe)) then

if Akt\_Kante\_Ptr.Endknoten\_Ptr = Nach\_Knoten\_Ptr then

Opt\_Loesung := Akt\_Loesung;

Opt\_Last := Stufe;

Opt\_Wert := Wert(Opt\_Loesung(1..Opt\_Last));

Lsg\_Gefunden := True;

elsif Stufe < Max\_Kanten then

Treffe\_Entscheidung(

Fuer\_Ptr => Akt\_Kante\_Ptr.Endknoten\_Ptr,

Stufe => Stufe + 1);

end if;

end if;

Akt\_Kante\_Ptr := Akt\_Kante\_Ptr.Next\_Ptr;

end loop;

end Treffe\_Entscheidung;

-----------------------------------------------------------

begin -- Optimalweg

-- die Knotenzeiger zu den Knotennamen bestimmen:

if Von\_Knoten\_Ptr = null or Nach\_Knoten\_Ptr = null then

raise Knoten\_Fehler;

end if;

Treffe\_Entscheidung (Fuer\_Ptr => Von\_Knoten\_Ptr, Stufe => 1);

if not Lsg\_Gefunden then

raise Loesungs\_Fehler;

end if;

return Opt\_Loesung(1 .. Opt\_Last);

end Optimalweg;

--------------------------------------------------------------

begin -- Autorouter

Graph\_Laden (

G => Strassennetz,

Name => "GRAPHEN.TXT");

Put (Optimalweg (

G => Strassennetz\_Ptr,

Von => "Augsburg",

Nach => "Muenchen" ) );

exception

when Knoten\_Fehler =>

Ada.Text\_IO.Put\_Line ("\*\*\* Knoten\_Fehler \*\*\* Ein

entsprechender Knoten fehlt");

when Loesungs\_Fehler =>

Ada.Text\_IO.Put\_Line ("\*\*\* Loesungs\_Fehler \*\*\* Es wurde keine

Loesung gefunden");

end Autorouter;

## Erweiterter Zugriff

### Zugriffe auf statisch deklarierte Objekte

allgemeine Zugriffstypen

Zugriffsobjekte der bisher beschriebenen Zugriffstyp können nur auf dynamisch erzeugte Objekte zugreifen. In Ada 95 wurde zusätzlich die Möglichkeit geschaffen auch auf bereits deklarierte (also statische) Objekte zuzugreifen. Man spricht in diesem Zusammenhang von allgemeinen Zugriffstypen. Hierfür muss eine zusätzliche Modifikation der Zugriffstypen und Zielobjekte erfolgen.

Zunächst noch einmal die vollständige BNF für Zugriffstypen:

typ\_definition ::=

**type** identifier **is** access\_type\_definition

access\_type\_definition ::=

access\_to\_object\_definition

| access\_to\_subprogram\_definition

access\_to\_object\_definition ::=

**access** [general\_access\_modifier] subtype\_indication



general\_access\_modifier ::= **all** | **constant**

Die Zugriffserweiterung erfolgt in 3 Schritten:

1. Verallgemeinerung des Zugriffstyps

Zunächst muss der Zugriffstyp mit dem Schlüsselwort all vereinbart werden:

type ALLGEMEINER\_ZUGRIFF\_REF is access all TYP\_T;

Pointer : ALLGEMEINER\_ZUGRIFF\_REF;

1. Sichtbarkeit der Zielobjektadresse

Um auf bereits vereinbarte Objekte des Typs TYP\_T über einen Verweis zuzugreifen, müssen diese mit dem Schlüsselwort aliased vereinbart werden.

-- die Variable ist jetzt auch über Zugriffe veränderbar

Objekt : aliased TYP\_T;

1. Abgabe der Zielobjektadresse an das Zugriffsobjekt

Die so deklarierten statischen Objekte sind in der Lage ihre Adresse abzugeben. Diese Absonderung erfolgt unter Verwendung des Zugriffsattribut 'Access.



-- der Pointer verweist nun auf die Adresse von Object

Pointer := Objekt'Access;

Zielobjekte können somit alle statischen Variablen und Konstanten sein, so ein Zugriffstyp für den Zielobjekttyp existiert und die Zielobjekte zur Abgabe ihrer Adresse bereit sind.

-- der Zeiger kann zwischen den beiden Konstanten wechseln

-- der Inhalt der Zielobjekte kann aber nicht verändert werden

ConstObjekt\_1 : aliased TYP\_T := ???;

ConstObjekt\_2 : aliased TYP\_T := ???;

Pointer := ConstObjekt\_1'access;

Pointer := ConstObjekt\_2'access;

Der Zeiger kann zwischen den Objekten wechseln. Der Inhalt der Zielobjekte bleibt aber konstant. Sobald das Zielobjekt wieder eine Variable ist, darf diese selbstverständlich verändert werden.

ausschließlich  
lesender   
Zugriff

Die Bearbeitung der Zielobjekte kann aber auch grundsätzlich auf nur lesende Zugriffe eingeschränkt werden (read only access). Dies erfolgt in der Typvereinbarung durch das Schlüsselwort constant anstelle von all. Dieser lesende Zugriff kann sowohl auf Variablen als auch auf Konstanten erfolgen. Die Veränderbarkeit des Zeigers selbst ist von dessen konkreter Deklaration abhängig.

type LESENDER\_ZUGRIFF\_REF is access **constant** TYP\_T;

-- Vereinbarung eines variabler Zeiger

LPointer : LESENDER\_ZUGRIFF\_REF;

-- lesender Zugriff auf eine Variable (read only)

LPointer := Objekt'access;

-- Vereinbarung eines konstanten Zeigers

CPointer : **constant** LESENDER\_ZUGRIFF\_REF := Objekt'access;

### Globaler Zugriff auf lokale Objekte

Das Sprachkonzept von Ada verhindert Zeiger, die auf nicht gültige Adressen verweisen – außer, der Programmierer übernimmt die Verantwortung und prägt eine Ada.Unchecked\_Deallocation aus und gibt ungeprüft Zeigerwerte frei.

Auch der folgende Versuch, einen „frei hängenden“ Zeiger zu erhalten, wird vom Compiler abgelehnt:

package Data is

type INT\_REF is access all Integer;

Ref1\_Ptr : INT\_REF;

end Data;

...

with Data;

procedure P is

K : aliased Integer; -- lokales Objekt

Ref2\_Ptr : Data.INT\_REF; -- globaler Zugriffstyp

begin

Ref2\_Ptr := K'Access; -- illegal, denn dann waere auch:

Data.Ref1\_Ptr := Ref2\_Ptr; -- moeglich!

end P;

...

In der Prozedur P kann mit Ref2\_Ptr, das ein Objekt des globalen Zeigertypen im Paket Data ist, nicht auf ein lokales Objekt K gezeigt werden.



Das Problem ist dabei zuerst nicht offensichtlich, denn Ref2\_Ptr und K haben die gleiche Lebensdauer. Allerdings wird schon diese Zuweisung abgelehnt.

Würde das nicht geschehen, könnte nämlich die zweite Zuweisung auf den globalen Zeiger Ref1\_Ptr angewiesen werden, wogegen syntaktisch nichts spricht. Nach Ablauf der Prozedur wäre aber genau der frei hängende Zeiger entstanden: Ref1\_Ptr existiert noch, aber das Objekt, auf dass er zeigt, nicht mehr! Der Compiler weist also die erste Zuweisung zurück.

### Zugriffe als formale Zugriffsparameter

Die statische Gültigkeitsregeln verhindern die Anwendung von allgemeinen Zugriffen insbesondere in Unterprogrammen, obwohl der Sicherheitsanspruch von Ada nicht geschmälert wird.

So ist beispielsweise die Auswertung eines Zugriffsattributes innerhalb eines Unterprogramms nicht möglich, wenn das Objekt nicht im Gültigkeitsbereich des Unterprogramms liegt, obwohl ein lokal erzeugter allgemeiner Zugriffstyp auf solche Objekte keine Gefahr eines „*hängenden*“ Zeigers bedeuten.

Zugriffs-parametertyp

Für solche, nicht wenig vorkommenden Fälle wurde eine neue Form von Formalparametern eingeführt, deren Typ ein sogenannter Zugriffsparametertyp ist. Der Modus eines Parameterzugriffstyps ist immer in und wird nicht explizit angegeben. Der formale Parameter ist dabei stets ein Objekt eines anonymen Zugriffstyps.

type TYP\_T is ...;

type ALLGEMEINER\_ZUGRIFF\_REF is access **all** TYP\_T;

Statisches\_Objekt : aliased TYP\_T;

Dynamisches\_Objekt : ALLGEMEINER\_ZUGRIFF\_REF := new TYP\_T'(...);

-- Parameteruebergabe mit statischer Bindung

procedure Mach\_Was (T : in out TYP\_T);

-- gültige Unterprogrammaufrufe mit Call-by-value

Mach\_Was (Statische\_Objekt);

Mach\_Was (Dynamisches\_Objekt.All);

-- Parameteruebergabe mittels eines allgemeinen Zugriff

procedure Mach\_Was (T : access TYP\_T);



--gueltige Unterprogrammaufrufe mit Call-by reference

Mach\_Was (Statische\_Objekt'access);

Mach\_Was (Dynamisches\_Objekt);

### Zugriffe auf Unterprogramme

Ada 83 sah ausschließlich eine statische Bindung eines Unterprogramm-Aufrufs an eine entsprechende Unterprogrammdeklaration vor.

Zeiger   
auf Unterprogramme

Die Gründe für diese Restriktion konnten sich bei der Weiterentwicklung der Sprache nicht mehr durchsetzen und so werden in Ada95 Zugriffstypen auf Unterprogramme zur Verfügung gestellt. Damit können flexibel Zugriffe auf Unterprogramme eingerichtet werden.

typ\_definition ::=

**type** identifier **is** access\_type\_definition

access\_type\_definition ::=



access\_to\_object\_definition

| access\_to\_subprogram\_definition

access\_to\_subprogram\_definition ::=

**access** [protected] procedure parameter\_profile

**|** **access** [protected] function parameter\_and\_result\_profile

Beispiel

function Sin (X : FLOAT) return FLOAT; -- Funktionspezifikation

function Cos (X : FLOAT) return FLOAT; -- Funktionspezifikation

-- Zugriffstyp mit einem bestimmten Profil

type WINKEL\_FUNKT\_REF is access function (F : FLOAT) return FLOAT;

-- Verweis auf eine Funktion mit diesem Profil

W\_Ptr : WINKEL\_FUNKT\_REF;

X, Phi : FLOAT;

-- W verweist jetzt auf Sin

W\_Ptr := Sin'access;

-- Funktionsaufruf über einen Zugriff;

X := W\_Ptr(Phi);

...;

-- Jetzt verweist W auf Cos

W\_Ptr := Cos'access;

X := W\_Ptr(Phi);

Damit können Unterprogramme mit gleichem Parameterprofil in Reihungen und Listen verwaltet werden. Diese Technik wird insbesondere in der Systemverwaltung und Menüsteuerung eingesetzt.

procedure Init;

procedure Start;

procedure Connect;

procedure Update;

type AKTIONS\_REF is access procedure;

-- anonymes Feldobjekt

AutoStart : array (1..10) of AKTIONS\_REF;

-- nur die zu aktivierenden Aktionen werden adressiert

Autostart (1) := Init'access;

Autostart (2) := Start'access;

for I in Autostart'range loop

if Autostart(I) /= null then

Autostart(I).all; -- Schreibweise .all ist für parameterlose

-- Unterprogramme nötig

end if;

end loop;

Index

.

**.all** 7, 9

A

allgemeine Zugriffstypen 57

*Allokator* 5

Ast 31

*Ausfügen aus einem Binärbaum* 38

Ausfügen aus einem Graph 46, 48

Ausfügen aus einer Liste 25

**Autoroute** 49

B

Baum 31

Binärbaum 32

Blatt 31

C

CONTROLLED 12

D

**Deklaration von Zugriffstypen** 3

*dereferenzieren* 7

*Dereferenzierung* 9

Durchlaufen einer Liste 22

Durchlaufen eines Binärbaums 34

*dynamisches Datenobjekt* 1

E

Einfügen in eine Liste 23

Einfügen in einen Binärbaum 36

Einfügen in einen Graph 44, 45

Einsatzmöglichkeiten 15

Einschränkung 6

Entscheidungsbaum 50

G

garbage collection 12

geordneter Binärbaum 33

gerichtete Kante 41

gerichteter Graph 41

*Gleichheit von Zeigern* 2

Graph 41, 49

I

In-Order-Durchlauf 35

Iterator 22

K

Kante 41

Knoten 41

Kollektion 6

L

leerer Baum 31

lineare Liste 19

Liste 19, 42

M

Menge 30

Mengenverwaltung 30

N

**new** 5, 12

**null** 2, 4

O

*Operationen auf Zugriffsobjekte* 4

P

Pointer 1

Post-Order-Durchlauf 35

Potenzmenge 30

Pragma CONTROLLED 12

Pre-Order-Durchlauf 35

R

read only access 58

redundante Daten 15

Referenz 1, 7

Rekursive Datenstrukturen 18

Rekursives Durchlaufen 27

Rekursives Einfügen 27

Rekursives Löschen 28

S

Schlinge 41

Seiteneffekt 17

Speicherbereinigung 12

Startzeiger 19

statisches Datenobjekt 1

*STORAGE\_ERROR* 12

T

Teilbaum 31

Tiefe 31

U

unbeschränkter Typ 10

Unchecked\_Deallocation 13

unvollständige Typdefinition 20

V

Verweis 1

*Vorinitialisierung* 4

W

Wurzel 31

Z

Zeiger 1, 4

*Zeiger auf unbeschränkte Reihungen* 10

*Zeiger auf unbeschränkte Verbunde* 11

Zieltyp 3

Zieluntertyp 3

Zugriffsattribut 58

Zugriffsobjekt 1

Zugriffsobjekte 4

Zugriffstyp 3

Zugriffswert 1

zusammengesetzter Datentyp 9