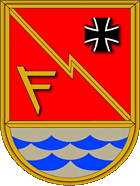
Schule Informationstechnik   
 der Bundeswehr



**Sprachausbildung Ada**



**„Dynamische Datenstrukturen“**

– Übungen –

Aufgabe 1 - Einführung

**A\_Ptr**, **B\_Ptr**, **C\_Ptr** und **D\_Ptr** seien Zeigervariablen, die auf Objekte vom Typ **INTEGER** zeigen.

type INTEGER\_REF is access INTEGER;

A\_Ptr, B\_Ptr, C\_Ptr, D\_Ptr : INTEGER\_REF;

Stellen Sie das Resultat folgender Anweisungen graphisch dar.

A\_Ptr := new INTEGER'(3);

B\_Ptr := new INTEGER'(3);

C\_Ptr := A\_Ptr;

D\_Ptr := new INTEGER;

D\_Ptr.All := A\_Ptr.all;

C\_Ptr.All := 1;

Aufgabe 2 - Einführung II

with Ada.Text\_IO;

procedure Uebung2 is

-- Typen

type STRING\_REF is access STRING;

subtype NAME\_TA is STRING(1..10);

subtype PLZ\_TA is STRING(1..5);

type WOHNORTE\_TR is record

Strasse\_Ptr : STRING\_REF;

HausNr : POSITIVE := 1;

Plz : PLZ\_TA := (others => '0');

end record;

type WOHNORTE\_REF is access WOHNORTE\_TR;

type ADRESSE\_TA is array (POSITIVE range <>) of WOHNORTE\_REF;

type PERSON\_TR is record

Vorname\_Ptr : STRING\_REF;

Nachname : NAME\_TA := (others => ' ');

Adressen : ADRESSE\_TA(1..3);

end record;

type AUSWAHL\_T is (J, N);

-- Pakete

package AUSWAHL\_IO is new Ada.Text\_IO.Enumeration\_IO(AUSWAHL\_T);

package I\_IO is new Ada.Text\_IO.Integer\_IO(INTEGER);

-- Unterprogramme

-- Spezifikationen

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

-- GebeWohnort --

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

-- Beschreibung: --

-- Funktion fragt die Daten (Komponenten) eines Wohnortes --

-- von Tastatur ab und gibt diesen Wohnort als Zeiger --

-- zurueck. --

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

-- function GebeWohnort return WOHNORTE\_REF; -- zu Punkt 3

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

-- Put --

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

-- Beschreibung: --

-- Prozedure gibt die Daten eines Wohnortes auf dem --

-- Bildschirm aus. --

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

-- procedure Put(Wohnort : in WOHNORTE\_TR); -- zu Punkt 5.1

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

-- Put --

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

-- Beschreibung: --

-- Prozedure gibt die Daten aller Wohnorte auf dem --

-- Bildschirm aus. --

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

-- procedure Put(Adressen : in ADRESSE\_TA); -- zu Punkt 5.1.2

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

-- Put --

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

-- Beschreibung: --

-- Prozedure gibt alle Daten einer Person auf dem --

-- Bildschirm aus. --

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

-- procedure Put(Person : in PERSON\_TR); -- zu Punkt 5.2

-- Bodies

-- 1. Aufgabe: eine Person von PERSON\_TR auf dem STACK (statischen Speicher) anlegen

-- 1.1 einen Wohnort auf dem HEAP (dynamischen Speicher) anlegen

-- 1.2 Zeichnen Sie den aktuellen Zustand

begin

-- 2. Weisen Sie der Person einen Vornamen und einen Nachnamen zu

-- 3. Nutzer die Moeglichkeit geben, bis zu drei Wohnorte der Person hinzuzufuegen

-- 3.1 Wollen Sie einen weiteren Wohnort eingeben (J/N)?

-- 3.2 Nach Eingabe von drei Wohnorten kein weiteres Nachfragen implementieren

-- 4. Wohnort (von 1.1) der Person als zusaetzlichen Wohnort aufnehmen, wenn

-- dies nicht moeglich, da alle drei Wohnorte schon belegt sind, nachfragen

-- welcher ersetzt werden soll?

-- 5. Ausgaben erzeugen

-- 5.1 Prozedur schreiben, die einen Wohnort (WOHNORT\_TR) ausgibt

-- 5.1.1 Ausgabe des Wohnortes (von 1.1)

-- 5.1.2 Prozedur schreiben, die alle Wohnorte der Person ausgibt

-- 5.2 Prozedur schreiben, die die gesamten Personendaten (PERSON\_TR) ausgibt

null;

end Uebung2;

Hinweis: Datei ist als Vorlage vorhanden.

Aufgabe 3 - Dynamische Strings

Schreiben Sie alle notwendigen Vereinbarungen zur Bildung eines Reihungsobjektes, das in der Lage ist, die Namen von 12 Lehrgangsteilnehmern als Zeichenketten aufzunehmen. Die Zeichenketten sollen dabei nur so lang als nötig für die einzelnen Namen sein, d.h. keine überflüssigen Leerzeichen enthalten.

a) Integrieren Sie diese Vereinbarungen in ein vollständiges Ada-Programm, welches diese 12 Namen einliest und in der Reihe ablegt und am Ende zum Test wieder ausgibt.

b) Modifizieren Sie Ihr Programm (neues HP schreiben, auf Basis von a), so dass die Anzahl der Teilnehmer vorab abgefragt werden und daraufhin erst das Reihungsobjekt dynamisch erzeugt wird.

c) Nach der erfolgten Eingabe der Namen und anschließender Ausgabe aller Namen sind zuletzt wieder alle dynamisch erzeugten Objekte wieder zu löschen (Freigabe von dynamischen Speicher).

Aufgabe 4 - Listengröße

Schreiben Sie eine Funktion, die die Anzahl der Listenelemente in einer gegebenen linearen Liste zählt. Achten Sie auch darauf, dass die Liste auch leer sein könnte.

function Anzahl\_Elemente (Start : ELEM\_REF) return NATURAL;

Benutzen Sie dabei folgende Typvereinbarungen:

type ELEM\_TR;

type ELEM\_REF is access ELEM\_TR;

type ELEM\_TR is record

Info : CHARACTER;

Naechster\_Ptr : ELEM\_REF;

end record;

Aufgabe 5 - Einfache Listenverwaltung

Erstellen Sie ein **generisches** Paket Listenverwaltung, dass diverse Service-Prozeduren für dynamische Listen anbieten:

* Einfügen eines Elementes am Listenanfang (LIFO)
* Einfügen eines Elementes am Listenende (FIFO)
* Ausführen einer vom Nutzer vorgegebenen Aktion (für ein Element), angewendet auf alle Listenelemente (generische Schleife)

Der Nutzer hat pro Paketausprägung nur **eine** einzige Liste zur Verfügung und kennt die Struktur dieser Liste nicht. Dafür gibt er einen **beliebigen** **Elementtyp** vor, der in der Liste gespeichert werden soll.

Erstellen Sie ein Hauptprogramm: main\_hoersaal\_liste, das in der Lage ist, Lehrgangsteilnehmer (Änderung des HP von Aufgabe 2) einliest, diese in der Listenverwaltung hinterlegt und abschließend alle Namen der Teilnehmer wieder ausgibt. Folgende Einzelschritte sind zu erfüllen:

* Einbinden des Pakets: Listenverwaltung
* Ausprägen des generischen Pakets auf Strings mit unterschiedlicher Länge (STRING\_REF) für die Namen der Teilnehmer
* Erstellen Sie eine Prozedur, die die Anzahl der Teilnehmer ermittelt (zum Ausprägen der generischen Schleife)
* Erstellen Sie eine Prozedur für die Ausgabe der Namen

(zum Ausprägen der generischen Schleife)

* Prägen Sie die generische Schleife aus,
  + zum Abfragen der Anzahl der Teilnehmer und
  + für die Ausgabe der Namen
* Das Einlesen der Namen soll in einer Schleife erfolgen, diese ist mit einer Beenden-Bedingung zu verlassen.
  + Namen abfragen und im Paket hinterlegen. Die Namen sollen in der Reihenfolge ausgegeben werden, wie sie eingelesen wurden.
  + Abfrage: Sollen weitere Teilnehmer eingelesen werden, wenn ja, dann 1.Schritt wiederholen ansonsten die Schleife verlassen.
* Ausgabe der Teilnehmer mit Hilfe der generischen Schleife. Die Bildschirmdarstellung ist verbindlich. Nutzer-Eingaben sind *kursiv* geschrieben!

Geben Sie den 1. Namen ein: *Max*

Moechten Sie weitere Teilnehmer eingeben? (Ja/Nein) *ja*

Geben Sie den 2. Namen ein: *Michael*

Moechten Sie weitere Teilnehmer eingeben? (Ja/Nein) *Ja*

Geben Sie den 3. Namen ein: *Maria*

Moechten Sie weitere Teilnehmer eingeben? (Ja/Nein) *Nein*

Es sind 3 Teilnehmer im Hoersaal.

Max

Michael

Maria

Aufgabe 6 - Flavius-Methode

Der jüdische Geschichtsschreiber Flavius Josephus berichtet, dass er zusammen mit 40 anderen Juden in einen Keller geflüchtet sei, als die Römer eine jüdische Stadt eroberten. Um den Feinden nicht in die Hände zu fallen, wollten sie sich gegenseitig umbringen. Josephus war dagegen, konnte die anderen aber nicht umstimmen. Um dem Tod zu entgehen, schlug er vor, man solle sich im Kreis aufstellen und reihum abzählen. Jeder Dritte sollte auf der Stelle umgebracht werden. Sein Vorschlag wurde akzeptiert, und Josephus stellte sich natürlich so in den Kreis, dass er als letztes übrig und so am Leben blieb.

Schreiben Sie ein vollständiges Ada-Programm, dass eine Folge von Namen für die beteiligten Personen einliest. Die Liste soll die Namen ohne überflüssige Leerzeichen in der Reihenfolge des Einlesens enthalten.

Beenden Sie das Einlesen, wenn das Zeichen **'\_'** einzeln als Name angegeben wurde. Schließen Sie sodann den Kreis und beginnen Sie mit dem Abzählen bei der Person, die als erstes eingelesen wurde.

Entfernen Sie nacheinander die jeweils dritte Person aus dem Kreis und geben Sie deren Namen auf dem Bildschirm aus.

Aufgabe 7 - Rückwärtskette

Neben den bisher vorgestellten einfach verketteten Listen kann auch für bestimmte Aufgabenstellungen eine doppelt verkettete Liste notwendig sein.



In dieser vorwärts und rückwärts verketteten Liste hat jedes Element zwei Zeiger. Ein Zeiger stellt die Verbindung zum Nachfolgeelement, der andere Zeiger die zum Vorgängerelement her.

Schreiben Sie eine Prozedur, die zu einer einfachen Liste parallel die Rückwärtskettung aufbaut und deren Anfang in der Variablen Ende zurückgibt. Die Vorwärtsverkettung ist also bereits durchgeführt und die Variable **Start** zeigt auf den Anfang dieser Verkettung.

procedure Rueck (Start\_Ptr : in ELEM\_REF; Ende\_Ptr : out ELEM\_REF);

Benutzen Sie dabei folgende Typdeklaration:

subtype INFORMATION\_T is STRING (1.. 10);

type ELEM\_TR;

type ELEM\_REF is access ELEM\_TR;

type ELEM\_TR is record

Info : INFORMATION\_T;

Vorgaenger\_Ptr,

Nachfolger\_Ptr : ELEM\_REF;

end record;

Aufgabe 8 - Polynomliste

Ein Polynom hat folgenden Aufbau

*f* (x) = 14.3 x4 + 3.7 x2 - 10.9 x + 4.0

Solch ein Polynom kann als sortierte Liste repräsentiert werden, wobei jedes Listenelement einen Summanden darstellt, dessen Koeffizient ungleich 0 ist. Ein Listenelement beinhaltet neben der Verweiskomponente den Koeffizienten und den Exponenten des Summanden. Der Summand mit dem höchsten Exponenten ist zuerst in der Liste gespeichert. Die anderen Summanden folgen in absteigender Reihenfolge, wobei jeder Exponent nur einmal vorkommt.

Das obige Polynom ist dann folgendermaßen als Liste abgespeichert:



Erstellen Sie eine geeignete Datenstruktur, die ein Polynom aufnehmen kann (Die Koeffizienten des Polynoms sind zur Vereinfachung vom Datentyp FLOAT, die Exponenten vom Datentyp NATURAL).

Schreiben Sie eine Funktion, welche den Wert eines Polynoms für einen gegebenen Wert x berechnet. Als Eingangsparameter werden der Funktion die Verweisadresse auf das Polynom und der Wert von x mitgegeben.

Aufgabe 9 - Mengenverwaltung

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.

In einer Menge darf jedes Element nur einmal vorkommen.

Mengen­verwaltung



Gefordert ist die Implementierung einer Mengenverwaltung in einem generischen Paket. Das Paket stellt nach Ausprägung den abstrakten Datentyp MENGE zur Verfügung. Der Paketnutzer darf lediglich Objekte vom Typ MENGE und die im Paket angebotenen Unterprogramme und Konstanten verwenden:

* Zuweisung von zwei Mengen (Von - Nach)
* Aufnahme eines Elementes in eine Menge (am Anfang oder Ende)
* Entfernen eines Elementes aus einer Menge
* Entnahme des ersten Elementes aus einer Menge (zerstörendes Lesen)
* Prüfung auf Gleichheit zweier Mengen
* Prüfung, ob ein Element in einer Menge enthalten ist
* Bestimmung der Anzahl der Elemente in einer Menge
* Zusatz: generische Schleife für alle Elemente einer Menge und Ausprägung im Hauptprogramm mit einer Put-Aktion
* Zusatz: nachträgliches Sortieren einer Menge, so dass die Elemente aufsteigend angeordnet sind. Dabei dürfen nicht die Elemente ausgetauscht werden - die Sortierung ist allein durch Zeigertausch zu realisieren.

Berücksichtigen Sie dabei Deallokation von nicht mehr benötigtem Speicher sowie das Auslösen von Fehlern.

Für das Testen des Paketes wird ein Hauptprogramm zur Verfügung gestellt.

Aufgabe 10 - Blätter zählen

Als Blätter in einem Binärbaum werden die Knoten bezeichnet, die keine Nachfolger haben.

Folgender Binärbaum hat z.B. 4 Blätter:



Schreiben Sie eine Funktion, die zu einem beliebigen Binärbaum die Anzahl der Blätter liefert.

function Anzahl\_Blaetter (Wurzel\_Ptr : in KNOTEN\_REF) return   
 NATURAL;

Benutzen Sie dabei folgende Typvereinbarungen:

type KNOTEN\_TR;

type KNOTEN\_REF is access KNOTEN\_TR;

type KNOTEN\_TR is record

Info : CHARACTER;

Links\_Ptr,

Rechts\_Ptr : KNOTEN\_REF;

end record ;

Aufgabe 11 - Baumhöhe

Die Höhe eines Binärbaumes ist definiert als die Anzahl der Knoten des längsten Astes.

Die Höhe kennzeichnet damit auch die maximale Verzweigungstiefe des Binärbaumes.

Schreiben Sie, ausgehend von den Definitionen der Aufgabe 10, eine Funktion, die zu einem beliebigen Binärbaum dessen Höhe liefert.

function Hoehe (Wurzel\_Ptr : KNOTEN\_REF) return NATURAL;

Aufgabe 12 - Character-Baum

Schreiben Sie ein vollständiges Ada-Programm, das feststellt, wie oft verschiedene Zeichen in einem einzulesenden Text vorkommen.

Realisieren Sie die Lösung mit Hilfe eines Binärbaumes, in den jedes neu vorkommende Zeichen als Knoten geordnet eingefügt wird. Für jedes weitere Exemplar eines Zeichens soll nur der Zähler des bereits vorhandenen Knotens erhöht werden. Der Binärbaum ist damit in der Ordnung des Aufzählungstyps CHARACTER sortiert.

* Lesen Sie zuerst eine beliebige Zeichenkette ein.
* Fügen sie jedes Zeichen schrittweise in den Binärbaum ein.
* Geben Sie anschließend alle vorgekommenen Zeichen mit ihrer Anzahl in der Ordnung des Typs CHARACTER nach einander aus.

Orientieren Sie sich an folgendem Beispiel:

Eingelesene Zeichenkette: "KERL 11 TEST"



Ausgabe:

2

1 2

E 2

K 1

L 1

R 1

S 1

T 2

Für die Übungen zur Datenstruktur Graph stehen die folgenden Typdefinitionen als global vereinbart zur Verfügung.

subtype KNOTEN\_NUMMERN\_T is POSITIVE;

type KNOTEN\_INFO\_REF is access STRING;

pragma CONTROLLED(KNOTEN\_INFO\_REF);

type KANTEN\_INFO\_REF is access STRING;

pragma CONTROLLED(KANTEN\_INFO\_REF);

type KNOTEN\_TR;

type KNOTEN\_REF is access KNOTEN\_TR;

pragma CONTROLLED(KNOTEN\_REF);

type KANTEN\_TR;

type KANTEN\_REF is access KANTEN\_TR;

pragma CONTROLLED(KANTEN\_REF);

type KNOTEN\_TR is record

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

-- der Knotenliste

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

-- Identifikation

Info\_Ptr : KNOTEN\_INFO\_REF; -- Informationsmenge

Markierung : NATURAL := 0; -- optional, zur

-- Markierung von

-- Knoten

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

-- Element der Kantenliste

end record;

type KANTEN\_TR is record

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

-- der Kante in der Knotenliste

Info\_Ptr : KANTEN\_INFO\_REF; -- Informationsmenge

Markierung : NATURAL := 0; -- optional, zur Markierung von

-- Kanten

Naechster\_Ptr : KANTEN\_REF; -- naechste Kante in der

-- Kantenliste

end record;

subtype GRAPH\_T is KNOTEN\_REF;

Aufgabe 13 - Kanten zählen

In einem Graph werden die Verbindungen zwischen den Knoten als Kanten dargestellt.

Folgender Graph hat z.B. 8 Kanten:



Schreiben Sie eine Funktion, die zu einem Graphen die Gesamtzahl der Kanten ermittelt.

function Anzahl\_Kanten (G : in GRAPH\_T) return NATURAL;

Aufgabe 14 - Ungerichteter Graph

Zur simulierten Darstellung von ungerichteten Graphen mit Hilfe der Datenstrukturen für gerichtete Graphen ist es notwendig, zu jeder gerichteten Kante eine Kante in Gegenrichtung zusätzlich aufzunehmen, sofern sie noch nicht vorhanden ist. Die neuen Kanten sollen dabei den Informationsgehalt ihrer Gegenkante übernehmen.

Gerichteter Graph: Ungerichteter Graph:

Schreiben Sie eine Prozedur, die einen gegebenen gerichteten Graphen in einen ungerichteten Graphen überführt.

procedure Bilde\_ungerichteten\_Graphen (G : in GRAPH\_T);

Aufgabe 15 - Graphen-Markierungen löschen

Häufig müssen bei Algorithmen für Graphen Knoten und/ oder Kanten markiert werden. Dafür ist in den Typdefinitionen für Knoten und Kanten die Komponente Markierung vorgesehen.

Schreiben Sie eine Prozedur, die in einem Graphen alle Markierungen bei Knoten und Kanten auf 0 setzt.

procedure Markierungen\_Loeschen (G : in GRAPH\_T);

Aufgabe 16 - Isolierte Knoten

Ein Knoten, der niemals Endknoten und niemals Startknoten einer Kante zu einem anderen Knoten ist, wird als isolierter Knoten bezeichnet.

Folgender Graph besitzt einen isolierten Knoten (Nr. 5):



Schreiben Sie eine Funktion, die für einen gegebenen Graphen feststellt, ob es isolierte Knoten gibt.

function Isolierte\_Knoten (G : GRAPH\_T) return BOOLEAN;

Für die Implementierung steht die Prozedur aus Aufgabe 15 zur Verfügung.

procedure Markierungen\_Loeschen (G : in GRAPH\_T);

Praktikum 1 – Gefädelter Binärbaum



Beim Durchsuchen von Binärbäumen werden im allgemeinen rekursive Algorithmen verwendet. Hierbei entsteht meist ein erheblicher Aufwand (Zeit und Speicherplatz) für Inkarnationen der rekursiven Unterprogramme, der bei entarteten Binärbäumen gelegentlich eine Bearbeitung unmöglich macht.

Es gibt allerdings noch eine andere Art, einen Binärbaum hinsichtlich dieser Auf­gabenstellung besser zu organisieren.

Ein gefädelter Binärbaum ist ein Binärbaum, der in den **null**-Verweisen der Knoten jeweils die Adresse des Vorgängers (für linke Verweise) bzw. Nachfolgers (für rechte Ver­weise) enthält. Damit enthalten nur zwei Verweise (im größten und kleinsten Schlüssel) den **null**-Verweis.

Es muss zwischen einem „gefädelten“ und einem „echten“ Verweis unterschieden werden. Dazu wird in jedem Knoten zu jedem Verweis eine boolesche Komponente vorgesehen.



type SCHLUESSEL\_T is range 1 .. 1000;

subtype INFORMATION\_TA is STRING(1..5);

type KNOTEN\_TR;

type BAUM\_REF is access KNOTEN\_TR;

type KNOTEN\_TR is record

Schl : SCHLUESSEL\_T;

Info : INFORMATION\_TA;

Links\_Ptr,

Rechts\_Ptr : BAUM\_REF;

Links\_Echt,

Rechts\_Echt : BOOLEAN;

end record;

Ihre Aufgabe:

Schreiben Sie ein Paket, das das abstrakte Datenobjekt B vom Typ BAUM\_REF beinhaltet und die DatentypenSCHLUESSEL\_T und INFORMATION\_TA zur Verfügung stellt. Der Datentyp BAUM\_REF ist intern als gefädelter Binärbaum mit aufsteigender Sortierreihenfolge zu realisieren. Folgende Operationen sind dem Paketbenutzer anzubieten:

* Suchen nach einem Schlüssel und Ausgabe der Information. Ist der Schlüssel nicht vorhanden, ist eine entsprechende Ausnahme auszulösen.
* Einfügen eines neuen Datensatzes (Schlüssel und Information). Ist der Schlüssel bereits vorhanden, ist auch hier eine entsprechende Ausnahme auszulösen.
* Iterative Ausgabe aller Datensätze (Schlüssel und Information) in der Sortierreihenfolge unter Nutzung der gefädelten Verweise.
* Rekursive Ausgabe aller Datensätze (Schlüssel und Information) in der Sortierreihenfolge, wie bei Bäumen üblich.

Bei den Operationen Suchen und Iterative Ausgabe sollen die gefädelten Verweise zur Vermeidung der rekursiven Aufrufe benutzt werden.

Durchlaufalgorithmus unter Ausnutzung der gefädelten Verweise:

-- Wurzel in Hilfsvariable uebernehmen

Hilf\_Ptr : BAUM\_REF := Wurzel;

Weiter : BOOLEAN := True; -- Schleifenbedingung

if Hilf\_Ptr /= null then -- leerer Baum?

while Weiter loop

-- nach links gehen, bis ein unechter Verweis kommt

-- (gefaedelt oder null)

while Hilf\_Ptr.Links\_Echt loop

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

end loop;

-- Aktion auf H.Schl und H.Info

-- nach rechts gehen, bis ein echter Verweis kommt. Dabei auf

-- jeden besuchten Knoten die Aktion auffuehren

while not Hilf\_Ptr.Rechts\_Echt and Weiter loop

if Hilf\_Ptr.Rechts\_Ptr = null then -- Verarbeitungsende?

Weiter := False;

else

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

-- Aktion auf Hilf\_Ptr.Schl und Hilf\_Ptr.Info

end if;

end loop;

if Weiter then

-- Hilfsvariable auf den echten Verweis weiterschalten

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

end if;

end loop;

else

-- Aktionen fuer leeren Baum

end if;

Hinweis:

* Die verbleibenden **null-**Verweise des größten bzw. kleinsten Elements werden als nicht echte Verweise betrachtet

Praktikum 2 – Dreifachverkettung



Hinter dem Bearbeiten häufig benutzter dynamischer Datenstrukturen, wie Listen und Bäume verschiedenster Art, steckt immer eine ähnliche Idee, nämlich ein Element über Operationen zu manipulieren.

Folglich laufen die entsprechenden Algorithmen zur Manipulation immer nach einem gleichartigen Muster ab, die sich nur durch den problemabhängigen Aufbau der Listen und Bäume unterscheiden.

So findet das Anfügen und Entfernen bei Datenkellern und Warteschlangen nur an den Enden statt. Bei der alphabetisch sortierten Liste ist es jedoch sinnvoll, einen neuen Knoten an einer beliebigen Stelle einzufügen oder zu entfernen.

Solche nach einer Ordnungsrelation (Datenkeller => LIFO; Warteschlangen => FIFO; Sortierte Liste => Alphabet) angeordneten linearen Listen heißen „geordnete Listen“. Neben Listen, die nur gleichartige Verknüpfungen aufweisen, gibt es auch geordnete Listen, deren Knoten verschiedenartige Nachfolger haben können, aller­dings von jeder Sorte höchstens einen. Zum Beispiel den alphabetisch nächsten Kunden oder einen Kunden mit geringerem Umsatz. Diese Listen nennt man mehrfach ge­ordnete lineare Listen (MGLL).

Mehrfach verkettete Listen kommen zum Beispiel bei der Realisierung einer dynamischen Spei­cherverwaltung mit Speicherbereinigung vor. Aber auch bei der Verwaltung von Prozessen und ihren Prioritäten bei Multiuser-/ Multitasking-Betriebssystemen oder von Cache-Algorithmen spielen MGLLs eine wichtige Rolle.

Da das Arbeiten mit mehrfach verketteten Strukturen als fehlerträchtig gilt, ist es sinnvoll, die entsprechenden Algorithmen zur Manipulation in einem Paket anzubieten.

Ihre Aufgabe:

Schreiben Sie den Paketrumpf zu folgender generischer Paket­spezifikation. Das Paket bietet als abstraktes Objekt eine dreifach verkettete Liste an.

generic

type INFORMATION is private;

type SCHLUESSEL is private;

with function "<=" (A, B : SCHLUESSEL) return BOOLEAN is <>;

package DVL is

type VERFAHREN\_T is (FIFO, LIFO, ALPHA);

procedure Aufnahme (

S : in SCHLUESSEL;

I : in INFORMATION);

procedure Loeschen (

S : in SCHLUESSEL);

generic

with procedure Aktion (

S : in SCHLUESSEL;

I : in out INFORMATION);

procedure Fuer\_Alle\_Elemente (V : in VERFAHREN);

generic

with procedure Aktion (

I : in out INFORMATION);

procedure Fuer\_Einen\_Schluessel (

V : in VERFAHREN;

S : in SCHLUESSEL);

procedure Entnahme (

V : in VERFAHREN;

S : out SCHLUESSEL;

I : out INFORMATION);

function Leer return BOOLEAN;

Zugriffsfehler : exception;

end DVL;

Anmerkungen zu Paketspezifikation bzw. -rumpf:

* Die einzelnen Elemente in den Listen bestehen aus einer beliebigen Information und einem beliebigen Schlüssel. Der Schlüssel unterliegt keiner Eindeutigkeit.
* Für jedes Verfahren gibt es einen Startzeiger und die einzelnen Listenelemente haben für jedes Verfahren einen Nachfolge­eintrag. Folgende Verfahren werden angeboten:
  + LIFO-Prinzip
  + FIFO-Prinzip
  + sortiert nach der Operation "<=" für einen beliebigen Schlüssel
* Folgende Operationen werden angeboten:
  + Aufnahme eines Elements in alle Listen
  + Entfernen (evtl. mehrfaches) von Elementen mit einem bestimmten Schlüssel aus allen Listen
  + Ausführen einer Aktion auf alle Elemente der Liste unter Angabe des Verfahrens
  + Ausführen einer Aktion auf die Elemente eines bestimmten Schlüssels innerhalb eines vorgegebenen Verfahrens
  + Entnahme des ersten bzw. kleinsten Elements aus der Liste eines angegebenen Verfahrens
  + Überprüfung, ob alle Listen leer sind
  + Eine Ausnahme für Zugriffsfehler