KERNFORSCHUNGSZENTRUM

KARLSRUHE

Januar 1972

KFK 1536

Institut für Reaktorentwicklung

Algorithmen zur Verarbeitung von Baumstrukturen und ihre Anwendung in ICES

U. Schumann, E. G. Schlechtendahl

GESELLSCHAFT FÜR KERNFORSCHUNG M. B. H.

KARLSRUHE

GESELLSCHAFT FÜR KERNFORSCHUNG M. B. H.
KARLSRUHE

KERNFORSCHUNGSZENTRUM KARLSRUHE

Januar 1972 KFK 1536

Institut für Reaktorentwicklung

Algorithmen zur Verarbeitung von Baumstrukturen und ihre Anwendung in ICES

U. Schumann
E. G. Schlechtendahl

Gesellschaft für Kernforschung m.b.H., Karlsruhe

Zusammenfassung

Die Definitionen von Baumstrukturen und ihrer Abarbeitungsfolgen werden diskutiert. Es wird unterschieden zwischen Mengen - und Ordnungsbäumen. Ein vollständiger Satz von Algorithmen zur Verarbeitung von Baumstrukturen und ihre Programmierung in ICETRAN, dem Fortran-Dialekt des ICES-Systems, wird beschrieben.
Unter Verwendung dieses Unterprogrammsatzes wurden Verwaltungsroutinen erstellt, mit denen beliebig aufgebaute DYNAMIC ARRAYS (die dynamischen und virtuellen Speicherfelder des ICES-Systems) ausgedruckt, geschrieben, gelesen und kopiert werden können.

Summary

The definitions of tree structures and their traversal orders are discussed. We distinguish set trees from ordered trees.A complete set of algorithms for the manipulation of tree structures and their realizations in ICETRAN, the FORTRAN-dialekt of the ICES-System, is described.
Using this subprogram - package some utility-like routines are established, which are able to print, write, read and copy general structured DYNAMIC ARRAYS (the dynamical and virtual storage arrays of ICES).

I n h a l t s v e r z e i c h n i s

Zusammenfassung	II	
Teil I - Einleitung	1	
	1	Baumstrukturen
	2	DYNAMIC ARRAYs
	4	Schlußfolgerungen
Teil II - Baumstrukturen Definitionen, Algorithmen und ICETRAN-Unterprogramme	7	
	7	Definitionen
	9	Darstellung einer Baumstruktur im Speicher
	12	Algorithmen zum Erzeugen, Verknüpfen, Trennen und Löschen von Knoten in einer Baumstrukturtabelle
	15	Algorithmen zur Bestimmung der Ebene eines Knotens sowie zur Abarbeitung in Pre-, Post- oder Endorder
	20	Programmbeschreibung
Teil III - Anwendung der Baumstrukturroutinen zur Verarbeitung von DYNAMIC ARRAYs	28	
	28	Allgemeines zu DYNAMIC ARRAYs und ihrer Verarbeitung
	29	Einzelheiten der Erstellung und Abarbeitung der Baumstrukturtabelle
	31	Einzelheiten der Erstellung der Zustandsbeschreibungstabelle
	32	Programmbeschreibung
Teil IV-ICETRAN-Listen	36	
Literatur	37	
Tabelle 1	40	Fehlermeldungen
Anhang 1	A-1	Allgemeines zu DYNAMIC ARRAYs
Abbildung 1-16	-	

I. Einleitung*)

1. Baumstrukturen

Unter einer Struktur versteht man eine Menge von Objekten mit definierten Relationen. Spezielle Strukturen sind lineare Felder oder Vektoren, Matrizen, Listen, Ringstrukturen [2,5], Plexe [3] und Baumstrukturen.

Baumstrukturen (vergl.Abb.1) sind solche Strukturen, deren zugehöriger Graph das Aussehen eines Baumes hat, bei dem von der Wurzel aus alle Äste über genau nur einen Weg erreicht werden können (genauere Definition siehe II.Kap.1)

Baumstrukturen treten in zahlreichen Zusammenhängen auf. Einige Beispiele sind:

> Stücklisten, Familienstammbaum, die Kapiteleinteilung eines Buches [5], Darstellung eines algebraischen Ausdrucks für eine symbolische Manipulation [3] oder für die Umsetzung in Maschinenanweisungen innerhalb eines Compilers, Darstellung der Zusammensetzung einer graphischen Abbildung in Computer Graphics [2,17], Information Retrieval [1], Zugriffsbäume in Speicherstrukturen usw..

In einem Rechner kann letztlich nur auf lineare Felder zugegriffen werden, da der Adressenraum der üblichen Rechner Vektorstruktur besitzt. Eine Rechnerstruktur, die Baumstrukturen von der Hardware bereitstellt, wurde von Berkling [19] vorgeschlagen, ist aber bisher nicht implementiert worden. Baumstrukturen müssen daher nach gewissen Regeln in lineare Felder umgesetzt werden. Hierzu gibt es verschiedene Verfahren und Algorithmen [1, 5, 6]. Diese Algorithmen werden jedoch zumeist nur in formaler Weise beschrieben, und, obwohl sie sicherlich an vielen Stellen zumindest teilweise für spezielle Anwendungen implementiert sind, es gibt kein allgemein verfügbares Unterprogrammpaket, mit dem

---

*) Teile dieses Berichtes wurden auf der Eighth Semi-Anual ICES Users Group Conference, San Francisco, USA (20.-21.Jan.72) vorgetragen [22]

Zum Druck eingereicht am 8.2.1972

z.B. ein Fortran-Programmierer die in seinem Problem auftretenden Baumstrukturen definieren und verarbeiten kann, ohne die, z.T. recht komplizierten Algorithmen erst selbst programmieren zu müssen. Eine Ausnahme bildet hier das PL/1-Programm TREEPAK [17] für die Manipulation von Baumstrukturen am Bildschirm im interaktiven Rechnerbetrieb.
Ein konkreter Bedarf nach derartigen Unterprogrammen trat auf, als ein Programm zur Manipulation grafischer Abbildungen als Subsystem GRAPHIC des ICES-Systems [7,8,18] erstellt werden sollte [9].
Es wurde daher ein Unterprogrammpaket zur Verwaltung von Baumstrukturen in ICETRAN [10], der Programmiersprache des ICES-Systems aufgebaut, das im Abschnitt II detailliert beschrieben ist.

2. DYNAMIC ARRAYS

Einer der wichtigsten Gründe für die Wahl des ICES-Systems als Grundlage für das zu erstellende Programm war das Konzept des DYNAMIC ARRAY (="DA") in ICETRAN.

Eine genauere Übersicht über die Eigenschaften dieser DAs wird in [7,8,10,11] (siehe auch Anhang 1) gegeben. Mit DAs ist es möglich

a) Kernspeicherplatz dynamisch zu belegen
b) mehrdimensionale Matrizen zu definieren, die nicht notwendig rechteckig angelegt sein müssen (möglich sind z.B.Dreiecksmatrizen und auch jede beliebige andere Form)
c) Felder mit dynamischer Größe zu spezifizieren: ein Feld, das zu klein ist, wird vom ICES-Kern automatisch vergrößert, wenn der Programmierer dies vorgesehen hat.
d) Externe Direktzugriffsspeicher als virtuelle Kernspeicher bereitzustellen: der ICES Kern verlagert bei Überlauf des direkt zugreifbaren Kernspeichers momentan nicht benötigte Felder auf die externen Dateien und holt diese bei Bedarf wieder in den Kernspeicher zurück; dies geschieht völlig automatisch; der Programmierer kann diese Kernspeicherverwaltung in ihrer Effektivität durch Angabe von Prioritäten und Kennzeichnung aktuell nicht benötigter Felder beeinflussen.

Da in DAs Subarrays mit Daten unterschiedlichen Modes (Integer, Real, Double, Pointer, Alpha) gefüllt werden können und DAs (wie unter b) erwähnt) von komplizierter, unregelmäßiger Struktur sein können, wird der Programmierer bei der Entwicklung eines Programms häufig an dem aktuellen Inhalt und der Struktur seiner DAs interessiert sein, um seinen Programmablauf kontrollieren zu können.

Es gibt jedoch in ICETRAN keine Möglichkeit, einen DA mit einem einfachen Statement auf einen File zu schreiben; hierzu muß der Programmierer für jeden DA individuell angepaßte Programmteile erstellen. Das gleiche gilt für das Kopieren (Umspeichern) von DAs oder Teilen hiervon. Die ICETRAN-Sprache selbst gestattet nur, Operationen(z.B.arithmetische Operationen, Ein-/Ausgabeoperationen) mit den Elementen von DAs zu formulieren (Ausnahmen stellen die Argumentübergabe an Unterprogramme, sowie der SWITCH-Befehl, mit dem zwei Zeiger auf einzelne DAs vertauscht werden können, dar). Es entstand daher der Wunsch, allgemein verwendbare Unterprogramme verfügbar zu haben, die folgende Aufgaben erfüllen können:

a) Ausdrucken eines beliebigen DA mit allen seinen Subarrays in lesbarem Format

b) Ausgabe eines DA auf eine sequentielle Datei (z.B. Magnetband)

c) Einlesen eines DA von einer sequentiellen Datei

d) Kopieren eines DA in einen anderen.

Es zeigte sich, daß hierbei vorteilhaft auf die Routinen zur Manipulation von Baumstrukturen zurückgegriffen werden kann, da die in einem DA repräsentierte Struktur ebenfalls eine Baumstruktur ist. Unter Verwendung der in Teil II beschriebenen Unterprogramme wurden daher die in Teil III beschriebenen Unterprogramme zur Erfüllung der obengenannten Aufgaben erstellt.

3. Schlußfolgerungen

3.1 Diskussion der Routinen zur Verwaltung von Baumstrukturen

Die in Teil II angegebenen Algorithmen lassen sich aufgrund ihrer allgemeinen Darstellung (nach Knuth [5]) auch in anderen Programmsprachen als ICETRAN programmieren . Einige Routinen wurden von einem der Autoren in FORTRAN erstellt. ICETRAN erlaubt jedoch die Datenstrukturen in dynamischer Weise im Kernspeicher anzulegen und bei Bedarf zu vergrößern.
Außer bei der Verwendung in den Routinen zur Verwaltung von DYNAMIC-ARRAYs ist vorgesehen, die beschriebenen Programme innerhalb der z.Z. noch bearbeiteten Programmnachrichtenverwaltung [13] zur Definition und Auswertung von logischen Ausdrücken, deren Wert über Abbruch oder Fortsetzung der Programmausführung nach einer Fehlermeldung entscheidet, anzuwenden; dort sind die Baumstrukturen nicht in Preorderfolge (vergl.Teil I, Kap.5) sondern in Endorderfolge (siehe gleiches Kapitel) abzuarbeiten.

Bei der speziellen Verwendung nach Teil III zeigte sich, daß der Zeiger auf den "linken Bruder" nicht, ein Zeiger auf den Nachfolger gemäß Preorderfolge dagegen sehr häufig benötigt wird (letzterer ist hier nur algorithmisch implementiert). Andererseits würde dieser Zeiger in der erwähnten Programmnachrichtenverwaltung besser durch einen solchen für die Endorderfolge ersetzt. Erstrebenswert sind daher Algorithmen, bei denen die Datenstruktur zur Beschreibung der Baumstruktur je nach den Operationen zur Ausführungszeit vom Benutzer definiert oder gar automatisch ermittelt werden kann.

3.2 Diskussion der Routinen zur Verwaltung der DYNAMIC ARRAYs

Die beschriebenen Routinen haben sich bereits bewährt. Insbesondere ist DYARPR für den Programmierer neuer ICES-Subsysteme eine willkommene Hilfe bei der Analyse seiner Datenfelder und der Suche nach Fehlern.

DYARWR und DYARRE können verwendet werden, um von den Datenstrukturen eines ICES-Subsystems, Kopien langfristig und für jeden Benutzer einzeln auf Magnetbändern zu halten. Der Verwendung von Platten für diesen Zweck sind in der Regel durch die damit verbundenen Kosten Grenzen gesetzt. Auf diese Weise bilden also sequentielle Datenträger, wie z.B.Magnetbänder, die logische Erweiterung der Direktzugriffsspeicher; es wäre denkbar, auf diese Weise einen virtuellen sekundären Direktzugriffsspeicher aufzubauen.

Allerdings zeigen die bisherigen Erfahrungen in der Verwendung der erstellten Routinen, daß die Routinen für die standardmäßige Verarbeitung sehr großer DAs durch Rückgriff auf Assembler-Routinen und direkten Eingriff in die ICES-internen Datenstrukturen effektiver gestaltet werden sollten.

3.3 Allgemeine Bemerkungen zu ICETRAN

Bei der Erstellung der hier beschriebenen Routinen wurden folgende Eigenschaften von ICES bzw.ICETRAN als unangenehm empfunden:

- Es ist nicht möglich, Unterprogramme rekursiv aufzurufen, wie z.B. in PL/1. Dies ist allerdings in FORTRAN, der Grundlage von ICETRAN ebenfalls nicht möglich.
- Die POINTER der DYNAMIC ARRAYs enthalten fast alle relevanten Angaben über die Eigenschaften der Subarrays. Die DYNAMIC ARRAYs sind in diesem Sinne also selbstbeschreibende Datenstrukturen. Leider wird jedoch nicht der Mode der Daten (INTEGER, REAL, DØUBLEPRECISIØN oder ALPHA) sondern nur der Type (HALF, FULL, DØUBLE, POINTER) der Datenwörter registriert. In der Routine DYARPR mußten daher die Daten jeweils in zwei Formaten ausgedruckt werden, da allein anhand der Daten und der Type- Angabe der Mode nicht eindeutig festgestellt werden kann. Es sind maximal nur zwei Formate (und nicht vier), weil folgende Zuordnungen gegeben sind:

Word-Type (Wortlänge)	möglicher Mode
HALF (2 byte)	INTEGER
FULL (4 byte)	INTEGER oder REAL
DØUBLE (8 byte)	DØUBLE PRECISION oder ALPHA

Der die gedruckte Ausgabe betrachtende Mensch wird jedoch in der Regel anhand der Wertgrößenordnungen sofort erkennen, welcher Mode vorliegt (Vergl.Abb.15).

- Einige der vom ICES-Kern bereitgestellten Utility-Routinen (z.B.ISTAT,IDEF) erwarten variabel viele Argumentenlisten. Argumentenlisten lassen sich in den meisten Sprachen nicht in variabler Länge programmieren, es sei denn, man verwendet spezielle Assembler-Routinen [15]. Auch variable Indexlisten waren bei den Anwendungen nach Teil III wünschenswert und konnten nur durch direkten Zugriff auf ICES-Routinen verwirklicht werden.

- Eine verbesserte ICES-Version sollte die in Teil III beschriebenen Operationen vom Systemkern her bereitstellen und dafür spezielle Syntaxelemente in ICETRAN vorsehen. Beispielsweise wären für zwei DAs a und b folgende Statements wünschenswert:

```
b = a        (statt CALL DYARCØ (a,b)        )
PRINT(nf)a   (statt CALL DYARPR (a, 6, 'a') )
WRITE(nf)a   (statt CALL DYARWR (a, nf)      )
READ (nf)a   (statt CALL DYARRE (a, nf)      )
```

3.4 Schluß

Die beschriebenen Routinen wurden ausführlich in einem speziellen ICES-Subsystem TREE getestet. Sie stehen anderen ICES-Benutzern und sonstigen Interessenten zur Verfügung.
Die hier und in [14] beschriebenen Unterprogrammpakete sind Beispiele dafür, daß auch in ICES nicht nur komplette Subsysteme sondern auch einzelne Unterprogramme für die Erstellung neuer Subsysteme bereitgestellt werden sollten, ebenso wie in anderen Programmiersprachen für bestimmte Aufgaben besonders effektiv gestaltete Unterprogramme in Unterprogrammbibliotheken bereitgestellt und verwaltet werden (z.B.in Fortran und PL/1 das SSP [20]).

II. Baumstrukturen, Definitionen, Algorithmen und ICETRAN-Unterprogramme

1. Definitionen

Strukturen lassen sich als gerichtete Graphen darstellen [1,12], bestehend aus Objekten oder Knoten (nodes) und Assoziationen oder Relationen (branches, pointer, Zeiger) zwischen Paaren von Objekten. Die Relationen sind gerichtet, d.h.ein Zeiger geht aus von einem Knoten i und zeigt auf einen Knoten j. Wenn eine Relation definiert ist zwischen Knoten p_{i-1} und p_i für i=2, 3, ..., n, sagt man [4], ein Weg (path) der Länge n verbindet Knoten p_1 mit Knoten p_n. Die Knoten p_1 und p_n werden Endknoten (terminal nodes) des Weges genannt. Ein nichttrivialer (d.h. nicht von p_j direkt nach p_j führender) Weg, dessen Endknoten identisch sind, ist ein Ring (circuit).

Nach Salton [1] ist eine Baumstruktur (Baum, tree) definiert als ein gerichteter Graph, bei dem auf jeden Knoten maximal ein Zeiger zeigt und der keine Ringe enthält (vergl.Abb.1).
Es wird sich zeigen, daß diese Definition nicht in allen Fällen geeignet ist, da man gewisse Bäume mit Ringen darstellen kann (polynäre Mengenbäume, siehe unten). In [5] wird eine rekursive Definition von Baumstrukturen oder "trees" angegeben:
Eine Baumstruktur besteht aus einer endlichen Menge T von Objekten, wobei

a) es ein spezielles Objekt in T gibt, genannt "Wurzel" des Baumes und

b) die verbleibenden Objekte aufteilbar sind in m≥ o sich nicht überschneidende Mengen T_1, T_2, ..., T_m, von denen jede wiederum ein Baum ist.

Dies ist die Definition eines Mengenbaumes, da hier lediglich die Mengenzugehörigkeit, aber nicht die relative Ordnung beschrieben wird. Dies wird insbesondere durch die Veranschaulichung eines Mengenbaumes nach obigem Beispiel in Abb.2 verdeutlicht. Wenn man zwischen einer Relation zum ersten, zweiten, ...m-ten Baum unterscheidet, gelangt man zu einem Ordnungsbaum. Hier ist eine Vertauschung der einzelnen Teilbäume nicht willkürlich zulässig (vergl.Abb.3).

Man unterscheidet zudem zwischen binären und polynären Bäumen, je nachdem $m \leq 2$ oder $m \leq k$ mit $2 < k$ ist. Polynäre Bäume lassen sich jedoch mit binären Bäumen darstellen [5], also mit solchen Bäumen, von deren Knoten maximal zwei Zeigen ausgehen. Man muß hier jedoch den Zeigern unterschiedliche Bedeutung zuordnen [12], d.h.binäre Ordnungsbäume verwenden. Hierbei zeigt ein Zeiger (im folgenden senkrecht gezeichnet) auf den Baum, zu dem der Knoten selbst die Wurzel ist, während der andere Zeiger (im folgenden waagerecht gezeichnet) auf einen Knoten zeigt, der mit dem ersten Knoten eine gemeinsame Wurzel besitzt.

Je nachdem es sich um einen polynären Mengen- oder Ordnungsbaum handelt, wird man für den Zeiger, der auf Knoten mit gleicher Wurzel zeigt, zulassen, daß er zu einem Weg gehört, der ein Ring ist oder nicht, da nur bei Strukturen, die keine Ringe enthalten, die relative Position eindeutig angebbar ist, andererseits aber dort, wo die Position nicht von Bedeutung ist, Ringe einfachere Algorithmen zulassen [5]:

Polynäre Mengenbäume lassen sich durch binäre Ordnungsbäume darstellen (siehe Abb.4), bei denen der Zeiger, der auf Knoten mit gleicher Wurzel zeigt (waagerecht) zu dem Ring gehört, der alle Knoten mit gemeinsamer Wurzel verbindet, während der andere Zeiger (senkrecht) auf einen Ring von Knoten (genauer auf einen Knoten des Rings) zeigt, zu denen der Knoten selbst die Wurzel darstellt.

Polynäre Mengenbäume lassen sich durch binäre Ordnungsbäume beschreiben (siehe Abb.5), bei denen der Zeiger (waagerecht), der auf den nächsten Knoten mit gleicher Wurzel zeigt, zu einem Weg gehört, der von einem ersten Knoten ausgeht und zu allen weiteren Knoten mit gleicher Wurzel in eindeutiger Weise führt, während der andere (senkrechte)Zeiger auf den ersten Knoten eines derartigen Weges zeigt, zu dem der betrachtete Knoten selbst die Wurzel darstellt.

Zur Beschreibung der Relation eines binären Baumes werden ausgezeichnete Objekte häufig mit besonderen Namen bezeichnet, die entweder mehr formaler Herkunft sind oder den Bezeichnungen in einer Familie entlehnt sind (vergl.Abb.6), wobei man üblicherweise die Bezeichnungen der männlichen Familienmitglieder verwendet. Im folgenden werden die

formalen und (in Klammern) die "familiären" Begriffe definiert und später parallel verwendet.

Eine Struktur (Familie) besteht aus Objekten (Personen) mit definierten Relationen (Abstammungsangaben). Die Wurzel der Baumstruktur ist die Konfiguration (der Patriarch).
Zu jedem Objekt (Person) existiert ein Superobjekt (Vater)mit Ausnahme zur Konfiguration (Patriarch). Von Objekten ist mit senkrechter Relation nach unten eine Menge von Subobjekten (Söhnen) zu erreichen, wobei die senkrechte Relation das 1.Subobjekt (1.Sohn) willkürlich (Mengenbaum) oder gezielt (Ordnungsbaum) auszeichnet. Waagerechte Relationen verbinden Objekte mit anderen Objekten zu dem gleichen Superobjekt (Personen mit Brüdern). Objekte die keine Subobjekte besitzen nennt man Elemente (Personen ohne Söhne nennt man Kinder). Die Objektebene oder der Level (Generation) ist mit einer Zahl bezeichnet, deren Wert sich durch folgende Definition ergibt:
Die Objektebene (Generation) der Konfiguration (Patriarch) wird mit 0 bezeichnet; wenn ein Objekt (eine Person) zur Objektebene (Generation) n gehört, dann gehören die Subobjekte (Söhne) zur Objektebene (Generation) n+1. Die Zahl der Subobjekte (Söhne) eines Objektes (Person) bezeichnet man mit Grad.
Die folgenden Algorithmen gelten für polynäre Mengenbäume, da sie es nicht gestatten anzugeben, als wievielter Sohn ein Knoten einem Vater zuzuordnen ist.

2. Darstellung einer Baumstruktur im Speicher

2.1 Semantik der Datenstruktur

Zunächst wird hier die Semantik der Baumstrukturdarstellung beschrieben, die mit den unten beschriebenen Algorithmen verarbeitet werden können. Den Begriff Semantik hat Earley [12] für diesen Zusammenhang definiert:" The semantics of a data structure is the essential meaning of how its data is stored, how it is accessed, and how the structure may be changed."

Für eine Baumstruktur, die in flexibler und einfacher Weise aufgebaut, verändert und verarbeitet werden soll, reichen die Zeiger von dem Vater zu den Söhnen nicht aus; beispielsweise kann hierbei nicht oder nur durch einen Suchprozess festgestellt werden, wer der Vater eines Knotens ist. Es werden daher hier noch weitere Zeiger mit speziellen Bedeutungen verwendet. Die Baumstruktur wird hier aus Knoten aufgebaut, von denen folgende Zeiger ausgehen (vergl. Abb.7):

1S ist der Zeiger auf den Knoten, der der erste Sohn des Ausgangsknotens ist; falls kein Sohn existiert, zeigt 1S auf NULL.

V ist der Zeiger auf den Knoten, der der Vater des Ausgangsknotens ist; falls kein Vater existiert, zeigt V auf NULL.

RB ist der Zeiger auf den rechten Bruder, d.h.auf den nächsten Knoten in der (Ring)-Folge der Knoten mit gemeinsamen Vater. Vom letzten Knoten zeigt RB auf den ersten Knoten.

LB ist der Zeiger auf den linken Bruder, d.h.auf den vorhergehenden Knoten in der (Ring-)Folge der Knoten mit gemeinsamen Vater. Falls der Knoten selbst erster Sohn ist, zeigt LB auf den letzten Knoten der Folge.

Abb.7 zeigt ebenfalls den Sonderfall, daß eine Baumstruktur nur aus einem Knoten besteht.

Es wurde noch mit weiteren Zeigern experimentiert, insbesondere solchen Zeigern, die in Form eines Rings alle Knoten auf einem Weg verbinden, der einer bestimmten Abarbeitungssequenz entspricht (z.B. Postorderfolge, s.u.). Solche zusätzlichen Ringe müssen jedoch, wenn bei Änderungen der Struktur ein umständliches Suchen vermieden werden soll, stets als doppelt verkettete Ringe aufgebaut sein. Bei den bisher aufgetretenen Anwendungen erbrachten derartige Ringe jedoch nicht in allen Anwendungsfällen Vereinfachungen der Verarbeitungsroutinen oder Verkleinerungen der Rechenzeit. Aus Speicherplatzgründen wurde daher

auf die zusätzlichen Zeiger verzichtet.

2.2 Implementierung der Datenstruktur

Earley [12] definiert:
"The implementation is how the semantic is realized in a machine".
Gemäß Salton [1] werden die Baumstrukturen hier in einer Tabelle abgespeichert, deren Spalten Vektoren eines DYNAMIC ARRAY sind. Die i-te Zeile dieser Tabelle mit dem Index I=1, 2, ..., L enthält die Zeiger eines Knotens. Der Knoten ist selbst durch diesen Index identifiziert. Die Zeile enthält vier ganze, nicht negative Zahlen, gemäß:

Knoten I:

I1S(I)	IRB(I)	ILB(I)	IV(I)

Jede dieser Zahlen repräsentiert einen Zeiger, der von dem Knoten mit dem Index I ausgeht und auf den Knoten zeigt, dessen Index gleich dem Wert der entsprechenden Zahl ist. Der NULL-Zeiger ist durch die Zahl 0 repräsentiert. Die Zuordnung zu den oben definierten Zeigern erfolgt gemäß

I1S(I)	⇦	1S	des Knotens mit Index I
IRB(I)	⇨	RB	
ILB(I)	⇦	LB	
IV(I)	⇧	V	

Die Zeiger sind hier also durch Zahlenwerte an bestimmten Positionen dargestellt. Gemäß Salton [1] und Wedekind [6] kann man die Zeiger auch dadurch implementieren, daß man gewisse Regeln für die Reihenfolge der Knoten innerhalb der Tabelle definiert und dadurch die Zeiger bestimmt. Diese Form ist zwar speicherplatzsparend aber umständlich zu programmieren und aufwendig in der Rechenzeit, wenn die Struktur oft verändert werden soll. Allgemein muß man feststellen, daß die optimale Implementierung einer Baumstruktur stets einen Kompromiß darstellt zwischen den Forderungen nach minimalem Speicherplatz- und Rechenzeitbedarf. Dieser Kompromiß hängt zudem entscheidend von den speziellen Operationen, die auf die Baumstruktur angewendet werden sollen, ab. Da die Algorithmen hier in ICETRAN [10] programmiert werden, einer Programmsprache, die einen virtuellen Speicher kennt,

wurde auf einen kleinen Speicherplatzbedarf weniger Wert gelegt. Die Implementierungsform erlaubt es, Knoten an beliebiger Stelle innerhalb der Tabelle anzuordnen und insbesondere Knoten auch an solchen Stellen zu "löschen", die von besetzten Stellen benachbart sind, ohne daß sogleich ein Umspeichern der folgenden oder vorhergehenden Knoten in der Tabelle notwendig ist. Stattdessen sind jedoch Vorrichtungen erforderlich, die die Verwaltung der freien Tabellenplätze erlauben, wie z.B. ein Garbage Collector [5]; da hier jedoch der Speicherplatz der freien Plätze zur Verfügung steht, werden die freien Plätze in einem Keller (stack [5, S.234]), der als doppelt verknüpfte Liste (double linked list [5, S.278]) implementiert ist, registriert.
Bei der Initialisierung des Strukturspeichers werden zunächst alle Plätze dem Keller zugeordnet; die Verknüpfung geschieht mit Hilfe der Zeiger I1S und IV (vergl.Abb.8)
MFREE zeigt jeweils auf den ersten freien Platz
I1S(I) zeigt auf den nach I nächsten freien Platz
IV(I) zeigt auf den dem freien Platz I vorhergehenden Platz.
Die gesamte Länge der Tabelle wird in einer Variablen LENGTH gespeichert (LENGTH $\geq$ 1).

3. Algorithmen zum Erzeugen, Verknüpfen, Trennen und Löschen von Knoten in einer Baumstrukturtabelle

Zur Darstellung der Algorithmen werden hier nach Knuth [5] vollständige Sätze verwendet, da dadurch die Algorithmen am leichtesten zu verstehen sind. Sätze in eckigen Klammern sind Erläuterungen. "Springe nach Schritt i" soll heißen, daß der Algorithmus mit den im i-ten Absatz beschriebenen Vorschriften fortgesetzt werden soll. Als Vergleichsoperatoren werden verwendet gleich(=), nicht gleich($\neq$) sowie die üblichen $>$, $\geq$, $\leq$, $<$. Das Zeichen = wird außerdem als Zuweisungszeichen verwendet (a=b heißt: a erhält den Wert von b zugewiesen). Verwechslungen sind aufgrund des Kontextes ausgeschlossen.

3.1 Erzeugung eines Knotens

1. /Ist die Tabelle voll?/ Falls MFREE=0 ist, Fehlermeldung IERR=2.
2. /Der neue Knoten erhält den Index K in der Tabelle/
   K = MFREE.
3. /Entfernung des Knotens K aus dem Keller der freien Plätze/
   MFREE=I1S(K); falls MFREE ¬=0, IV(MFREE)=0.
4. /Ausgangszustand der Zeiger setzen/
   I1S(K)=0; IV(K)=0; IRB(K)=K; ILB(K)=K .

K wird als Ergebnis geliefert.

3.2 Verknüpfen zweier Knoten (zu einem Mengenbaum)

Der bereits definierte Knoten mit Index K soll an den bereits definierten Knoten L so angehängt werden, daß K der Sohn von L wird. K darf keinen Vater besitzen und L darf nicht ein Kind von K sein. (K wird stets der letzte Sohn von L - dies ist eine willkürliche, aber determinierende Festlegung).

1. /Ist K ein gültiger Index?/
   Falls (K ≤ 0 oder K > LENGTH,)Fehlermeldung IERR=3.
2. /Ist L ein gültiger Index?/
   Falls (L ≤ 0 oder L > LENGTH), Fehlermeldung IERR=4 . M = IV(K)
3. /Ist K ein Patriarch?/
   Falls M > 0, Fehlermeldung IERR=5.
4. /K darf nicht Mitglied sein der Sippe, der L angehört (sonst entstehen Ringe); da K keinen Vater hat, kann K nur dann Mitglied der Sippe von L sein, wenn K deren Patriarch ist, daher wird jetzt der Patriarch M von L gesucht:/ M = L;
   Führe Schritt 5. so oft aus, bis M=0 ist.

5. IALTR = M; M = IV(M)

6. /IALTR enthält den Index des Patriarchen der Sippe von L/
   Falls K=IALTR ist, Fehlermeldung IERR=6.

7. /Verknüpfung/
   IV(K) = L; Falls I1S(L)=0, I1S(L)=K; sonst IALTR=ILB(I1S(L);
   ILB(I1S(L))=K; IRB(K)=I1S(L); IRB(IALTR)=K; ILB(K) = IALTR).

3.3 Trennen von K aus der Struktur

Der Knoten K wird mit allen seinen Subobjekten aus der Struktur, zu der er gehört, herausgelöst.

1. /Ist K ein gültiger Index?/
   Falls ($K \leq 0$ oder $K > LENGTH$), Fehlermeldung IERR=3.

2. /Bestimmung des Vaters L von K/
   L = IV(K).

3. /Hatte K überhaupt einen Vater?/
   Falls L=0 ist, Fehlermeldung IERR=7

4. /Abtrennung/
   IV(K) = 0.
   /War K erster Sohn?/
   Falls K = I1S(L) weiter mit Schritt 6.

5. /K war 1.Sohn; war er auch einziger Sohn?/
   Falls K=IRB(K), (I1S(L)=0; weiter bei Schritt 7)
   sonst I1S(L)=IRB(K);

6. IRB(ILB(K)) = IRB(K)
   ILB(IRB(K)) = ILB(K)

7. /IRB und ILB zeigen auf sich selbst/
   IRB(K) = K
   ILB(K) = K

3.4 Löschen eines Knotens

Der Knoten mit Index K wird gelöscht. Hierbei wird vorausgesetzt, daß er zuvor von allen anderen Knoten gemäß 3.3 abgetrennt wurde.

1. /Ist K ein gültiger Index?/
   Falls (K≤0 oder K>LENGTH, IERR=3.

2. /Besitzt K weder Väter noch Söhne?/
   Falls ¬ (I1S(K)=0 und IV(K)=0), Fehlermeldung IERR=8.

3. /Einbau des Knotens in den Keller der freien Knoten/
   IV(K)=0;
   I1S(K)=MFREE;
   Falls MFREE ¬=0,IV(MFREE)=K;
   MFREE=K.

4. Algorithmen zur Bestimmung der Ebene eines Knotens sowie zur Abarbeitung in Pre-, Post- oder Endorder

4.1 Definitionen

1. Die Objektebene oder Generation wurde in Kap.1 definiert.

2. Häufig ist es erforderlich, eine Baumstruktur in einer bestimmten Weise "zu durchwandern" und dabei die einzelnen Knoten in einer systematischen Folge zu notieren oder zu verarbeiten, so daß jeder Knoten nur einmal verarbeitet wird.
   Drei Arten der Verarbeitungsfolgen ("Traversing") wurden von Knuth /5, S.316, 334/ in rekursiver Weise definiert und mit "Preorder traversal, Postorder traversal und Endorder traversal" bezeichnet.
   Die rekursive Definition lautet wie folgt:
   Preorder:
   Verarbeite jeden Knoten vor seinen Söhnen und diese vor seinen rechten Brüdern.
   Postorder:
   Verarbeite jeden Knoten nach seinen Söhnen aber vor seinen rechten Brüdern.

Endorder:

Verarbeite jeden Knoten nach seinen Söhnen und nach seinen rechten Brüdern.
Eine andere Bezeichnungsmöglichkeit wäre daher:

Preorder	-	Vorvorfolge	(Prepreorder)
Postorder	-	Nachvorfolge	(Postpreorder)
Endorder	-	Nachnachfolge	(Postpostorder)

Offenbar gibt es noch eine weitere Kombination:

Vornachfolge (Prepostorder)
Verarbeite jeden Knoten vor seinen Söhnen aber nach seinen rechten Brüdern.
Diese Kombination ist jedoch die gleiche, die man bei Postorder erhalten würde, wenn "rechter Bruder" durch "linker Bruder" ersetzt würde. Hierdurch ergeben sich auch zu Pre- und Endorder symmetrische Folgen. Andersartige Folgen erhält man, wenn nicht nur die unmittelbar mit einem Knoten in Relation stehenden Nachbarknoten berücksichtigt werden; beispielsweise könnte man eine Baumstruktur Ebene nach Ebene von links nach rechts und oben nach unten oder jeweils umgekehrt abarbeiten.
Pre-, Post- und Endorder sind also nur einige Spezialfälle der möglichen Verarbeitungsfolgen; sie treten jedoch in vielen Fällen auf und sollen daher hier alleine berücksichtigt werden.

Da die hier verwendete Programmiersprache ICETRAN [10], wie viele andere Sprachen (z.B.Fortran) keine rekursiven Unterprogrammaufrufe erlaubt, werden im folgenden zudem direkte Definitionen gegeben:

Die drei Verarbeitungsfolgen lassen sich zurückführen auf ein dichtes Umfahren des Graphen des Baumes im Gegenuhrzeigersinn von Wurzel bis Wurzel (siehe Abb.9) [19, 21], Auf diesem Weg werden die Knoten markiert und für die Verarbeitungsfolge notiert bzw.verarbeitet. Einmal markierte Knoten werden nicht noch einmal notiert.

Preorder: Die Knoten werden markiert, sobald sie in beliebiger Richtung verlassen werden.

Postorder: Die Knoten werden markiert, wenn sie nach rechts unten oder nach oben verlassen werden.

Endorder: Die Knoten werden markiert, wenn sie nach oben verlassen werden.

(Die oben erwähnte Prepostorder-Folge läßt sich hier nicht verwirklichen; sie setzt ein Umlaufen des Baumes im Uhrzeigersinn voraus).

Abb.10 zeigt für eine spezielle Baumstruktur die sich so ergebenden Folgen.
Die Preorderfolge wird bei der Verarbeitung der DYNAMIC ARRAYs eine wesentliche Rolle spielen (siehe Teil III). Die Endorderfolge ist beispielsweise zu verwenden, wenn man einen nicht nur aus einem Knoten bestehenden Baum mit den in Kap.3.3 und 3.4 beschriebenen Algorithmen auftrennen und löschen will.

4.2 Algorithmen

4.2.1 Bestimmung der Ebene LEVEL des Knotens K

1. /Ausgangswert setzen/
   LEVEL=0; I=K.

2. /Wandern bis zur Wurzel des Baumes/
   Falls IV(I) = 0 springe zu Schritt 3;
   LEVEL=LEVEL+1; I=IV(I);
   Wiederhole diesen Schritt.

3. /LEVEL enthält den gesuchten Ebenenwert von Knoten K/

4.2.2 Erzeugung der Verarbeitungsfolgen

Die folgenden Algorithmen erwarten, daß ihnen in LAST der Index des zuletzt verarbeiteten Knotens geliefert wird; falls mit der Abarbeitung erst begonnen wird, muß LAST den negativen Wert des Index des Knotens

enthalten, der der Patriarch der Struktur ist. Falls mit einem Knoten begonnen wird, der selbst noch einen Vater hat, ist die erzeugte Abarbeitungsfolge fehlerhaft und erst beendet, wenn bei der Umwanderung des Graphen die Wurzel selbst erreicht wurde.
Die Algorithmen liefern als Ergebnis entweder den auf den letzten Knoten folgenden Knoten der Folge (oder den ersten) oder den Wert 0 als Hinweis darauf, daß der zuletzt verarbeitete Knoten der letzte zu verarbeitende Knoten der Folge war.

Die hier beschriebenen Algorithmen verändern die Struktur selbst in keiner Weise, kommen also insbesondere ohne eine Markierung aus.

a) Preorder

1) ⌜Ist LAST der Startindex oder kein gültiger Index ?⌟
   Falls LAST=0, Fehlermeldung IERR=12
   Falls LAST<0, NEW=-LAST springe nach Schritt 5.
   L = LAST.

2) ⌜LAST>0; hat der Knoten LAST einen Sohn?⌟
   NEW = I1S(L). Falls NEW>0, springe nach Schritt 5.

3) ⌜Hat der Knoten L einen Vater? wenn nein, dann ist das Ende der Folge erreicht.⌟
   NEW = IV(L); Falls NEW=0? springe nach Schritt 5.

4) ⌜Hat der Knoten L einen rechten Bruder, der nicht 1.Sohn ist?⌟
   IIV=NEW; NEW=IRB(L); Falls NEW=I1S(IIV) springe nach Schritt 3.

5) ISTPRE=NEW ⌜Ergebnis⌟

b) Postorder

1) ⌜Ist LAST der Startindex oder kein gültiger Index?⌟
   Falls LAST=0, Fehlermeldung IERR=11;
   falls LAST < 0,(NEW=-LAST,Springe nach Schritt 6).

2) /Hat LAST einen Vater? Wenn nein, dann ist das Ende der Folge erreicht./
K = IV(LAST)
Falls K=0 ist, ISTPOS=L /Ergebnis/.

3) /Hat LAST einen rechten Bruder, der 1. Sohn ist?/
NEW=IRB(LAST); Falls NEW=I1S(K), springe nach 7.

4) /NEW ist nicht 1.Sohn; nimm den 1.Sohn von NEW/
K=I1S(NEW); Falls K=0 ist, nimm NEW, d.h.springe nach Schritt 7.

5) /der 1.Sohn existiert/
NEW=K.

6) /suche den 1. Sohn, der keine weiteren Söhne hat./
K=I1S(NEW); Falls K>0, springe nach Schritt 5.

7) ISTPOS=NEW /Ergebnis/.

c) Endorder

1) /Ist LAST der Startindex oder kein gültiger Index?/
Falls LAST=0, Fehlermeldung IERR=13;
Falls LAST<0, (L=LAST, Falls I1S(L)=0, springe nach Schritt 10.
andernfalls springe nach Schritt 7).

2) /Hat LAST noch einen Vater oder ist es das letzte Element der Folge?/
IIV=IV(LAST); Falls IIV=0, (L=0, springe nach Schritt 10).

3) /Hat LAST einen linken Bruder oder war er selbst 1.Sohn?/
Falls LAST ≠I1S(IIV), (L=ILB(LAST), springe nach Schritt 10),

4) /LAST war 1. Sohn. Hat der Vater einen rechten Bruder?/
Falls IRB(IIV)=IIV, (L=IIV, springe nach Schritt 10).

5) /IIV hat einen rechten Bruder und folglich auch einen Vater; Ist der rechte Bruder des Vaters 1.Sohn, dann steht der Vater am weitesten rechts./
Falls I1S(IV(IIV))=IRB(IIV)(L=IIV, springe nach Schritt 10).

6) /Der Vater von LAST hat noch rechte Brüder; verarbeite diese/
Springe nach Schritt 8 .

7) /Hierhin kommt man bei Beginn der Folge, wenn das Startelement (L=-LAST) Söhne hat./
L=I1S(L).

8) /Durchwandern der Folge der 1. Söhne/
K=I1S(L);
Falls K=0,(L=K, wiederhole diesen Schritt).

9) /Der betrachtete Knoten L hat keine Söhne; jetzt wird der rechte Bruder gesucht und falls vorhanden, dessen Folge 1. Söhne durchwandert./
K=IRB(L), Falls K=I1S(IV(L)) springe nach Schritt 10.;
andernfalls (L=K, springe nach Schritt 8.)

10) /Ergebnis/
ISTPOS=L.

5. Programmbeschreibung

5.1 Allgemeines

5.1.1 Programmiersprache ICETRAN

Die vorstehend beschriebenen Algorithmen zum Aufbau, zur Veränderung und Abarbeitung von Baumstrukturen wurden in ICETRAN /10/der Programmiersprache des ICES-Systems /7,8/programmiert. ICETRAN ist eine Erweiterung von FORTRAN-IV (Basic), u.a. um einige spezielle Statements für dynamische und beliebig strukturierte Speicherfelder, die DAs.

Unter eventuellem Verzicht auf diese Flexibilität lassen sich die Algorithmen jedoch auch in allen anderen Programmiersprachen erstellen.

5.1.2 Allgemeine Konventionen, Fehlerbehandlungen

Alle Subroutinen-Namen beginnen mit den Buchstaben ST, alle Funktionen (außer LEVEL) mit den Buchstaben IST. Fehlermeldungen werden über ERROR2, eine spezielle Subroutine einer Programmnachrichtenverwaltung [13] abgesetzt. Der Aufruf von ERROR2 hat die Form

```
CALL  ERROR2(s, nr )
```

s ist eine INTEGER-Variable, die eine Aussage über die Bedeutung dieses Fehlers für die weitere Programmausführung hat.
Für $0 \leq s \leq 7$ kann das Programm weiterrechnen (die Nachricht bildet lediglich eine Warnung), für $8 \leq s \leq 16$ soll die Programmausführung abgebrochen werden, da die weitere Rechnung zu nicht tolerierbaren Fehlern führt.

nr ist eine INTEGER-Variable, die eine Identifikation des Fehlers darstellt. Die hier verwendeten Werte für diese Nachrichtennummer nr sind in Tabelle 1 angegeben.

Fast alle Routinen besitzen als erstes ausführbares Statement

```
CALL    START ('name')
```

und vor dem RETURN

```
CALL    ZIEL
```

name ist hierbei stets der Name des Unterprogramms, das START ruft. Auch diese beiden Routinen sind zwei Routinen der erwähnten Programmnachrichtenverwaltung. Sie erlauben den Ausdruck eines Unterprogrammtrace und die Zuordnung von Programmnachrichten zu den Unterprogrammen. ERROR2 kann durch eine Routine ersetzt werden, die die übergebene Nachrichtennummer ausdruckt und bei $s \geq 8$ die Programmausführung abbricht (STOP oder ERROR RETURN).

START und ZIEL sind dann nicht notwendig erforderlich und können durch DUMMY-Routinen ersetzt werden.

Alle Routinen sind "reusable", d.h.sie verhalten sich unabhängig davon, ob sie zum ersten Mal oder zum zweiten Mal aufgerufen wurden. Außerdem können mit den Routinen in einem Programm mehrere, voneinander unabhängige Baumstrukturtabellen verwaltet werden, da alle Felder und Kontrollvariablen als Argument an die Routinen übergeben werden. Dennoch sind die Argumentenlisten sehr kurz und einfach, da alle Felder und Kontrollvariablen, die zur Baumstrukturtabelle gehören, über einen Zeiger eines DYNAMIC ARRAY erreicht werden (ähnliches wurde in [14] für HASH-Tabellen verwirklicht). Alle von den Routinen benötigten Daten werden über die Argumentenliste übergeben. Der COMMON wird hierzu nicht verwendet. Ferner wird nur, wo besonders angegeben, auf Files geschrieben oder von Files gelesen.

5.1.3 Aufbau der Baumstrukturtabelle

Die Baumstrukturtabelle ("ST") ist als DYNAMIC ARRAY angelegt. Dieser wird in der Subroutine STINIT wie folgt definiert:

```
      DYNAMIC ARRAY ST(I)
      DEFINE ST,5,PØINTER, HIGH
      DEFINE ST(1),2,LØW
      DØ 3I=2,5
    3 DEFINE ST(I),LENGTH, HALF, STEP=50, LØW
```

LENGTH ist die vom Benutzer zu spezifizierende Anfangslänge der Tabelle. Falls sie nicht ausreicht, wird die Tabelle vom ICES-Kern automatisch um 50 Plätze (entsprechend STEP=50) verlängert. Abb.11 zeigt die so definierte Tabellenstruktur grafisch.
Die Baumstrukturtabelle ST wird mit Hilfe des Unterprogramms STRUCT (s.u.) mit Anfangswerten initialisiert, die alle Tabellenplätze dem Keller der freien Plätze zuordnet, wie Abb.8 zeigt.

5.2 Beschreibung der Anwendungsvorschriften der einzelnen Unterprogramme

5.2.1 STINIT, Initialisierung der Baumstrukturtabelle

Aufruf:

```
CALL STINIT (st, lstart, length)
```

(statt CALL auch LINK, bzw.BRANCH mit entsprechend erweiterter Argumentenliste)

Argumente:

st	DA(I)*	Baumstrukturtabelle
lstart	I	erster, zu initialisierender Tabellenplatz (Eingabe)
length	I	letzter, zu initialisierender Tabellenplatz (Eingabe)

Wirkung:

Wenn lstart $\leq$ 1 ist, wird der DYNAMIC ARRAY st zunächst zerstört und dann definiert mit der Länge length.
Die Plätze lstart bis length werden mit Anfangswerten gemäß Abb.8 gefüllt.

5.2.2 STCREA, Erzeugung eines einzelnen Knotens

Aufruf: CALL STCREA(st, k)

Argumente:

st	DA(I)	Baumstrukturtabelle
k	I	Tabellenindex des erzeugten Knotens (Ausgabe)

Wirkung:

STCREA erzeugt einen Baumstrukturknoten gemäß Kap.3.1 und liefert als Ergebnis den Index des neu erzeugten Knotens in k. Falls die Tabelle keine freien Plätze mehr enthielt, wurde zuvor die Tabelle in ihrer Länge verdoppelt (maximal jedoch um 50 Plätze).

5.2.3 STADD, Addieren eines Knotens als Sohn an einen Vaterknoten

Aufruf: CALL STADD(st, k, l)

---

*) DA(I) = DYNAMIC ARRAY mit MODE INTEGER
I = INTEGER (full word)

Argumente:

st	DA(I)	Baumstrukturtabelle
k	I	Index des Sohnknotens (Eingabe). $0 < k \leq$ length
l	I	Index des Vaterknotens (Eingabe) $0 < l \leq$ length siehe 5.2.1)

Wirkung:

Die Knoten k und l in st werden so verknüpft, daß eine Baumstruktur entsteht, in der l der Vater von k bzw. k der letzte Sohn von l ist.

5.2.4 STCUT, Abtrennen einer Baumstruktur mit Patriarchen

Aufruf: CALL STCUT(st, k)

Argumente:

st	DA(I)	Baumstrukturtabelle
k	I	Index des Patriarchen (Eingabe) $0 < k \leq$ length (length siehe 5.2.1)

Wirkung:
Falls der Knoten k einen Vater besitzt (in der Baumstrukturtabelle st), wird der Knoten k mit allen seinen Söhnen vom Vater getrennt und als selbständige Baumstruktur gehalten.
Falls k keinen Vater besitzt, erscheint eine Fehlermeldung und es geschieht sonst nichts .

5.2.5 STERAS, Löschen eines Baumstrukturknotens

Aufruf: CALL STERAS (st, k)

Argumente:

st	DA(I)	Baumstrukturtabelle
k	I	Index des zu löschenden Knotens (Eingabe) $0 < k \leq$ length(length siehe 5.2.1)

Wirkung:

Der Knoten mit Index k wird, falls er weder Söhne noch Väter besitzt gelöscht und der frei werdende Platz dem Keller der freien Plätze zugeordnet; andernfalls erscheint eine Fehlermeldung und sonst geschieht nichts.

5.2.6 STRUCT Unterprogramm für STINIT, STCREA, STADD, STCUT und STERAS

Aufruf: CALL STRUCT(ks, k, l, length, mfree, i1s, irb, ilb, iv, ierr)

Argumente:

ks	I	Steuerparameter (Eingabe) s.u.
k	I	Tabellenindex
l	I	Tabellenindex
length	I	Tabellenlänge
mfree	I	oberster freier Platz des Kellers
i1s	DA(I)	Feld der 1. Söhne
irb	DA(I)	Feld der rechten Brüder
ilb	DA(I)	Feld der linken Brüder
iv	DA(I)	Feld der Väter
ierr	I	Fehlermeldungsnummer(siehe Tabelle 1) (Ausgabe) $1 \leq$ ierr ≤ 8

Bemerkung:

STRUCT dient als ausführendes Unterprogramm für die im Titel genannten Routinen. Wegen der langen Argumentenliste sollte es nicht direkt vom Benutzer verwendet werden.

Wirkung:

Die Wirkung wird durch ks gesteuert:

ks = 1 Initialisierung der Tabellenplätze zwischen den Indices k und l (einschließlich); die DYNAMIC ARRAYS müssen vorher definiert worden sein (siehe STINIT).

ks = 2 Erzeugen eines einzelnen Knotens (siehe STCREA)

ks = 3 Addieren des Knotens k als Sohn von Knoten l (siehe STADD)

ks = 4 Herauslösen von k aus l (siehe STCUT)

ks = 5 Löschen des Knotens mit Index k (siehe STERAS)

5.2.7 STPRNT, Ausgabe einer Tabelle auf den Drucker

Aufruf: CALL STPRNT (st, nf, text)

Argumente:

st	DA(I)	Baumstrukturtabelle
nf	I	File-Nr.der Druckereinheit (im OS-360 Standard nf=6)(Eingabe)
text	ALPHA 8*	Text aus 8 druckbaren Zeichen zu Identifikation der Tabelle.

Wirkung:

STPRNT erzeugt eine Druckliste von der Baumstrukturtabelle st auf File nf mit text in der Überschrift (siehe Abb.12). Falls st nicht definiert war, erscheint eine entsprechende Meldung auf File nf.

5.2.8 LEVEL, Bestimmung der Ebene eines Knotens in einer Baumstruktur

Aufruf: (als Function)
LEVEL (st, k) (Typ I)

Argumente:

st	DA(I)	Baumstrukturtabelle
k	I	Knotenindex (Eingabe)

* ALPHA 8 ist ein DOUBLE PRECISION Wort, mit bis zu 8 Characters gefüllt

Wirkung:

LEVEL liefert die Zahl der Väter, die ein Knoten k besitzt.

5.2.9 ISTPRE, ISTPOS, ISTEND, Bestimmung des Folgeindex einer Preorder-, Postorder- oder Endorder-Folge.

Aufruf: ISTPRE (st, last)
ISTPOS (st, last) (Function vom Typ I)
ISTEND (st, last)

Argumente:

st	DA(I)	Baumstrukturtabelle
last	I	letzter Index der Folge (Eingabe)

Wirkung:

a) last = 0 Fehlermeldung

b) last < 0 Das erste Element der Folge wird bestimmt, wobei IABS(last) der Index des Patriarchen der abzuarbeitenden Baumstruktur sein muß. Die Funktionen liefern den Index des ersten Elements.

c) last > 0 Der Index des auf den Knoten mit Index last folgende Knoten wird als Funktionswert geliefert.

Je nachdem ISTPRE, ISTPOS oder ISTEND aufgerufen wird, wird die Preorder-, Postorder- oder Endorder-Folge geliefert.

d) Falls last der letzte Knoten der betrachteten Folge ist, ist der Funktionswert gleich Null.

III. Anwendung der Baumstrukturroutinen zur Verarbeitung von DYNAMIC ARRAYs

(Abkürzung: DA = DYNAMIC ARRAY)

1. Allgemeines zu DYNAMIC ARRAYs und ihrer Verarbeitung

Das DYNAMIC ARRAY-Konzept in ICES ist ausführlich beschrieben in [7, 8, 10, 11]. Der Leser, dem dieses Konzept unbekannt ist, sei verwiesen auf die kurze Beschreibung in Anhang 1.

Wie Abb.13 zeigt, kann ein DA als Baumstruktur interpretiert werden. Es ist ein Ordnungsbaum, da eine Vertauschung der Äste einer Vertauschung von Indices entspricht. Dennoch kann dieser Ordnungsbaum mit den im Teil II beschriebenen Algorithmen für Mengenbäume dargestellt werden, weil die einzelnen Knoten in Preorder (Kap.II-4.1) erzeugt (und nicht gelöscht) werden und die dazugehörigen Algorithmen (siehe Kap.II-3.2) einen neuen Knoten stets als letzten Sohn einem Vater zuordnen.

Für die Verarbeitung eines DA wird mit dem Unterprogramm DYARST die durch den DA repräsentierte Baumstruktur in eine Baumstrukturtabelle gemäß Teil II abgebildet und steht dann Verarbeitungsroutinen zur Verfügung. Gleichzeitig wird eine "Zustandsbeschreibungstabelle" angelegt. Diese Zustandsbeschreibungstabelle enthält an der Position, deren Index dem entsprechenden Index des Knotens in der Baumstrukturtabelle entspricht, die Zustandsbeschreibung des Knotens. Diese Zustandsbeschreibung enthält die in Anhang 1 beschriebenen Angaben, wie sie zu jedem Pointer mit den Routinen ISTAT und IDEF ermittelt werden können, sowie zudem die Liste der Indices und ihre Anzahl nz mit denen dieser Subarray vom ICETRAN-Programmierer identifiziert wird; bei "$A(1,2)" also die Zahlen 1 und 2 (nz=2), bei dem DA "A" die leere Liste (nz=0). Nicht dagegen werden die aktuellen Adressen der Subarrays in die Zustandsbeschreibung aufgenommen. Abb.14 zeigt den Aufbau der Zustandsbeschreibungstabelle für die Baumstruktur nach Abb.13.(Die Bezeichnungen der Spalten sind in Anhang 1 erläutert.) Baumstrukturtabelle ("BST") und Zustandsbeschreibungstabelle ("ZBT")

bilden die Grundlage der hier vorgestellten Routinen zum Ausdrucken, Schreiben auf sequentielle Datei, Lesen von sequentieller Datei und Kopieren beliebig strukturierter und mit beliebigen Daten gefüllter DYNAMIC ARRAYs. Beim Schreiben eines DAs werden nicht nur die Daten des DA auf die Datei geschrieben, sondern zugleich auch ihre BST und ZBT. Soll der DA später wieder eingelesen werden, so kann anhand der eingelesenen BST und ZBT ein DA genau wie der ursprüngliche definiert werden (DEFINE...) und sodann mit den Daten von der Datei gefüllt werden.

Für das Ausdrucken und Kopieren eines DA spielen die BST und ZBT nur eine temporäre Rolle; ihre Verwendung erlaubt es, die entsprechenden Algorithmen sehr einfach zu gestalten.

2. Einzelheiten der Erstellung und Abarbeitung der Baumstruktur-tabelle (BST)

Definiert ein Programmierer einen DA, so muß er zunächst den Basepointer (ohne Indices) definieren, sodann dessen Subarrays und weiter deren Subarrays. Der Programmierer darf nicht zunächst bei den Subarrays beginnen und dann zu dem Basepointer übergehen; dies ist vom ICES-Kern nicht zugelassen. Eine sinnvolle Abarbeitungssequenz ist daher die Preorderfolge (siehe Teil II), bei der ein Baumstrukturknoten vor seinen Söhnen und diese vor seinen rechten Brüdern verarbeitet werden.
In dem folgenden Algorithmus wird daher mit STINIT (siehe Teil II) eine BST initialisiert, mit STCREA werden Baumstrukturknoten zu jedem Pointer des DA erzeugt und mit STAD© verknüpft. Die Funktion ISTPRE wird zur Erzeugung der Preorderfolge verwendet, LEVEL liefert die Ebene jeden Knotens. Die Zustandsbeschreibung wird mit den im folgenden Kapitel beschriebenen Routinen ermittelt.

1. /Erzeugung der Baumwurzel/
   Erzeuge(mit STCREA) einen Knoten und speichere den Knotenindex in IBASE.

2. /Dieser Knoten entspricht dem Base-Pointer; ermittle dessen Zustand/
   Ermittle die Zustandbeschreibung des Base-Pointer und speichere diese an die Position IBASE der Zustandstabelle.

3. Falls der Base-Pointer nicht definiert ist oder auf ein Datensubarray (d.h.kein Pointerarray) zeigt, springe nach 9. /Ende/.

4. /K wird der Vaterindex/ K = IBASE.

5. /Erzeuge soviele Knoten, wie der Subarray Pointer enthält und hänge sie als Söhne an den Vater K/
   II = Länge des Subarrays (aus Zustandstabelle, Position K);
   LEV = LEVEL (K)+1;
   Für I=1 (Schritt 1) bis II:
   (Erzeuge Knoten und speichere Index in L;
   Verknüpfe Knoten L mit K so, daß L Sohn von K wird;
   Ermittle die Zustandsbeschreibung des i-ten Subarrayelements und speichere diese an die Position L der Zustandstabelle
   /hierbei wird LEV benötigt/)

6. L = Nachfolger von K /L=ISTPRE(ST,K)/ in Preorderfolge.

7. /Ist die Folge beendet ?/
   Falls L=0, springe nach 9./Ende/.

8. Falls der entsprechende Pointer keinen definierten Subarray oder einen Datensubarray besitzt, springe nach 6; andernfalls springe nach 5.

9. /Ende; BST und ZBT sind vollständig/

Nachdem so die BST und ZBT existieren, kann mit der Abarbeitung, d.h.dem Drucken, Schreiben oder Kopieren begonnen werden. Beim Lesen wird die BST und ZBT nicht erzeugt, sondern vor den Daten gelesen. Die Abarbeitung erfolgt nun ebenfalls in Preorderfolge. Die Einzelheiten der Abarbeitung sind den Programmlisten zu entnehmen; erwähnt sei hier nur ein spezielles Problem:

Es ist erforderlich, Subarrays ansprechen zu können mit variabel vielen Indices:

z.B. A, A(1),....,A(1,2,3,4,5,6), usw.

In ICETRAN ist es wie in den meisten anderen Programmiersprachen nicht möglich, einen Array mit variabel vielen Indices zu definieren. Aus diesem Grunde wurde ein spezielles Unterprogramm SWSUB erstellt, daß es erlaubt, einen Subarray eines DA mit einem nichtindizierten temporären DA zu "switchen", d.h. die entsprechenden Adresspointer zu vertauschen. Dieses Unterprogramm erwartet in einem speziellen Eingabevektor die Indices des Subarrays und führt den "SWITCH" unter Umgehung des ICETRAN-Procompilers dann direkt mit der ICES-Routine QQSWCH [18] aus.

3. Einzelheiten der Erstellung der Zustandsbeschreibungstabelle (ZBT)

Mit Hilfe der Subroutinen IDDEF und DEF wird eine bestimmte Position innerhalb der ZBT gefüllt.

Die Indexliste wird in IDDEF erstellt; sie ergibt sich aus der Indexliste des Vaters, verlängert um den laufenden Index des behandelten Pointers im Vatersubarray.

Die übrigen Angaben der Zustandsbeschreibung werden mit den ICETRAN-Routinen [10] ISTAT und IDEF ermittelt. Diese erzeugten jedoch spezielle Probleme: ISTAT und IDEF erwarten eine variabel lange Argumentenliste; die letzten Argumente sind jeweils die Indices des entsprechenden Subarrays und die Anzahl dieser Indices ist gerade gleich der Ebene des zugehörigen Knotens in der Baumstruktur.

Wollte man hier alleine auf ICETRAN-Ebene programmieren, so müßte man soviele Aufrufe von IDEF und ISTAT programmieren, wie Ebenen möglich sind und je nach aktueller Ebene zu den richtigen Aufrufen verzweigen. Da dies sehr aufwendig ist, wurde hier eine Assembler-Routine ARLIST verwendet, die in [15] genau beschrieben ist.

Bemerkt sei schließlich noch, daß die ZBT auf wesentlich kleinerem Raum abgespeichert werden könnte, wenn es möglich wäre, das Pointerwort direkt in die Tabelle zu speichern und erst dann zu analysieren, wenn die darin enthaltene Information für die Verarbeitung benötigt wird.

4. Programmbeschreibung

4.1 Allgemeine Konventionen, Fehlerbehandlung

Die Unterprogramme zur Verwaltung von DYNAMIC ARRAYs sind mit 4 Ausnahmen in ICETRAN programmiert. Lediglich das Unterprogramm ARLIST [15] ist eine Assemblerroutine, und CONVXI, PUTBYT, GETDBY sind FORTRAN-H-Programme.

Alle Subroutinennamen, die für den Benutzer interessant sind, beginnen mit den Buchstaben DYAR (als Abkürzung für DYNAMIC ARRAY). Fehlermeldungen werden über die Subroutine ERRØR2 an eine Programmnachrichtenverwaltung [13] übermittelt. Die Nachrichtenummern sind in Tabelle 1 erläutert. ERRØR2 sowie die Unterprogramme START und ZIEL wurden zudem bereits in Kap.II-5.1.2 erläutert.
Alle Routinen sind "reusable"(vergl.II-5.1.2). Sie können mehrere voneinander unabhängige DYNAMIC ARRAYs verwalten. Anstelle eines Base-Pointers kann auch jeder Subarray übergeben werden (z.B.$A(5,2). Alle, von den Routinen benötigten Daten werden über die Argumentenliste übergeben. Der COMMON wird hierzu nicht verwendet. Ferner wird nur auf die in der Argumentenliste spezifizierten Files geschrieben (außer bei eventuellen Fehlern in ERRØR2).

Die Eingabe-DAs werden nach ihrer Abarbeitung "released"[10] .

4.2 Beschreibung der Anwendungsvorschriften der einzelnen Unterprogramme

4.2.1 DYARPR, Erzeugung eines Dumps auf dem Drucker

Aufruf: CALL* DYARPR (a, nf, name)

Argumente:

a	DA** Pointer eines DYNAMIC ARRAY oder Subarray
nf	I ** File-Nummer der Drucker-Einheit
name	ALPHA 8** Ein Text aus maximal 8 Zeichen, zur Identifikation der Arrays

Wirkung:

DYARPR analysiert den angegebenen DYNAMIC ARRAY a und erzeugt hierzu auf dem Drucker mit File-Nummer nf einen lesbaren DUMP. Beispiele zeigen Abb.15. Der in name enthaltene Text erscheint in der Überschrift und in der letzten Zeile der DUMP-Ausgabe.

4.2.2 DYARWR, Schreiben eines DYNAMIC ARRAY auf eine sequentielle, externe Datei

Aufruf: CALL DYARWR(a,nf)

Argumente: siehe 4.2.1

Wirkung:

DYARWR analysiert den angegebenen DYNAMIK ARRAY a und schreibt anschließend auf den File mit der Nummer nf

---

*) Statt CALL ist auch LINK bzw.BRANCH (mit entsprechend erweiterter Argumentenliste) möglich.

**) DA = DANAMIC ARRAY-PØINTER
I = INTEGER (Fullword)
ALPHA 8 = DOUBLE PRECISION mit 8 Characters gefüllt

a) die Strukturtabelle, die die Baumstruktur des Arrays beschreibt
b) die Zustandstabelle
c) die Werte der Datenfelder des DYNAMIC ARRAY.

Geschrieben wird im A-Format mit einer Satzlänge von 80 Bytes
(RECFM=FB,LRECL=80,BLKSIZE=n*80,n≥1)
DYARWR enthält kein REWIND-Statement. Das Ausgabeformat ist so, daß der ARRAY a anschließend nach einer Rewind-Operation wieder mit DYARRE gelesen werden kann

4.2.3 DYARRE, Lesen eines DYNAMIC ARRAY von einer mit DYARWR beschriebenen, sequentiellen, externen Datei

Aufruf: CALL DYARRE (a, nf)

Argumente: siehe 4.2.1

Wirkung:

DYARRE liest von dem File mit der Nummer nf die zuvor mit DYARWR geschriebene Beschreibung eines DYNAMIC ARRAY und erzeugt danach einen entsprechenden Array, der an den mit a übergebenen Pointer gehängt wird. Der Array a entspricht anschließend in allen Einzelheiten dem Array, der mit DYARWR geschrieben wurde.
DYARRE enthält keine REWIND-Operation.
Der File nf muß die DCB Parameter RECFM = F oder FB und LRECL=80 besitzen.

4.2.4 DYARCØ, Kopieren eines DYNAMIC ARRAY

Aufruf: CALL DYARCØ (a, b)

Argumente:

a	DA	Der zu kopierende DYNAMIC ARRAY oder Subarray
b	DA	Der DYNAMIC ARRAY oder Subarray, der die Kopie aufnehmen soll.

Wirkung:

DYARCØ kopiert den DYNAMIC ARRAY a mit allen seinen Subarrays in den DYNAMIC ARRAY b. Der alte Inhalt von b wird zerstört, a bleibt unverändert. DYARCØ (a,b) entspricht also einem Zuweisungsstatement:

b = a

Hierbei können a und b selbst Subarrays unterschiedlicher Ebene sein; z.B.

CALL DYARCØ (A,$B(2,4))

4.2.5 DYARST, Erzeugung der Struktur- und Zustandsbeschreibung eines DYNAMIC ARRAY

Aufruf: CALL DYARST (a, st, id)

Argumente:

a	DA	der zu analysierende DYNAMIC ARRAY
st	DA(I)	Baumstrukturtabelle
id	DA(I)	Zustandsbeschreibungstabelle

Bemerkung:

DYARST wird von den zuvor erläuterten Unterprogrammen DYARPR, DYARWR, DYARCØ zur Analyse des DYNAMIC ARRAY a verwendet. Für den normalen Benutzer dürfte eine direkte Verwendung DYARST nicht sinnvoll sein.

Wirkung:

DYARST erzeugt zu einem DYNAMIC ARRAY a die Baumstrukturbeschreibung in der Baumstrukturtabelle st und die Zustandsbeschreibung in der zugehörigen Tabelle id. Der alte Inhalt von st und id wird zerstört; a bleibt unverändert.

4.2.6 SWSUB, Switch zwischen einem DYNAMIC ARRAY und einem mit variabel vielen Indices indizierten Subarray

Aufruf: CALL SWSUB(a, s, n, lev)

Argumente:

a	DA	1. DYNAMIC ARRAY
s	DA	2. DYNAMIC ARRAY
n	DA(I)	Indexliste
lev	I	Zahl der Indices

Wirkung:

SWSUB bewirkt einen SWITCH* gemäß :

SWITCH(S, A(N(1),N(2),...,N(LEV))

Hierbei kann LEV $\geq$ 0 eine variable Größe sein. Dies ist bei dem ICETRAN-SWITCH-Befehl nicht möglich, wird aber in den bisher beschriebenen Routinen häufig benötigt.

IV. ICETRAN - Programmlisten

Aus Platzgründen werden die ICETRAN-Quellprogramme hier nicht repro - duziert. Sie können bei folgender Adresse angefordert werden:

Institut für Reaktorentwicklung
Gesellschaft für Kernforschung mbH.,
D-75 Karlsruhe
Postfach 3640

*) Ein SWITCH(a,b) bewirkt die Vertauschung der Subarrays von a und b durch Vertauschung der entsprechenden Pointer.

L i t e r a t u r

[1] G. Salton: Manipulation of Trees in Information Retrieval
Comm.ACM 5 (1962), 103-114

[2] J.Encarnacao: Datenstrukturen für graphische Informationsverarbeitung.aus W.Giloi, Gesellschaft für Informatik e.V. (ed.), Bericht Nr.2, Symposium über Computer Graphics, Berlin 19.-21.Oktober 1971, S.15 - 49

[3] D.T.Ross: A Generalized Techniques for Symbol Manipulation and Numerical Calculation.Comm. ACM 4 (1961), 147-150

[4] K. E. Iverson: A programming notation for trees.
Research Report RC-390, IBM Research Center (Jan.1961)

[5] D. E. Knuth: The Art of Computer Programming
Vol.1: Fundamental Algorithmus, Chapter 2
Addision-Wesley Publ.,Reading Mass.(1969)

[6] H. Wedekind: Datenorganisation
W.de Gruyter, Berlin (1970)

[7] D. Roos (ed): ICES System: General Description
MIT, CESL September 1967, R67-49

[8] D. Roos: ICES System Design
The MIT Press, 2nd.ed.(1967)

[9] U. Schumann: Systematische Unterteilung von graphischen Objekten nach ihrem Bezugssystem und deren gemeinsame Darstellung.
W. Giloi, Gesellschaft für Informatik e.V.(ed): Bericht Nr.2, Symposium über Computer Graphics Berlin 19.-21.Oktober 1971, S.51-60

[10] J. C. Jordan (ed.): ICES: Programmers Reference Manual MIT, Department of Civil Engineering Report R67-50 Oktober 1967

[11] J.M.Sussmann: Primary Memory Management in ICES MIT, CESL Report R67-68 (1967)

[12] J. Earley: Toward an Understanding of Data Structures Comm. ACM 14, 10 (1971), 617-627

[13] A. Pee, U. Schumann: Ein System zur Verwaltung von Programmnachrichten und seine Implementierung in ICETRAN (in Vorbereitung)

[14] U. Schumann: Verwaltung von dynamischen Symboltabellen nach der HASH-Methode mit ICETRAN-Unterprogrammen Kernforschungszentrum Karlsruhe, Externer Bericht 8/71-7 Oktober 1971

[15] A. Pee, U. Schumann: Vorteile der Dynamisierung von Argumentenlisten und deren Verwirklichung in Fortran und Icetran. Kernforschungszentrum Karlsruhe, KFK 1488, Oktober 1971

[16] C.M.Holifield: Graphic display and manipulation of tree-type data structures.
Naval Postgraduate School AD-709091,Dec.1969

[17] R. Williams: A Survey and an annotated bibliography of data structures for computer graphics systems
New York University AD697800, Sept.1969

[18] IBM-Contributed Program Library
ICES/360 Source Basic System and Language Processors V1M1 (Correction 7/69); Prog.Ord.No.360D- .03.0.005 (1968)

[19] K.J.Berkling: A computing machine based on tree structures
IEEE Transactions on Computers, C-20,4(1971),404-418

[20] IBM: System/360 Scientific Subroutine Package
(Fortran: 360A-CM-03X,PL/1:360A-CM-07X)
Form H20-0205 und H20-0586

[21] D.T.Ross: The AED approach to generalized Computer-aided design
Proc.ACM 2nd Annual Conf.(1967)

[22] U. Schumann, E.G.Schlechtendahl: ICETRAN-Treestructure Routines to Save Direct Access Space and Debugging Time

Eighth Semi-Anual ICES Users Group Conference, San Francisco (USA),20.-21.Jan.1972 (Manuskript bei den Autoren erhältlich).

T a b e l l e 1

Erläuterungen der Fehlernachrichten

Fehler-* nummer (nr)	Sevirity-Code s	Quelle	Erläuterung
1	0	STRUCT	STRUCT wurde ohne Fehler beendet;(diese Meldung wird nicht über ERROR2 sondern über das Argument IERR weitergegeben)
2	4	STRUCT	Baumstrukturtabelle ist voll oder LENGTH < 1; (STCREA verlängert die Tabelle)
3	4	STRUCT	Index K ist ungültig: K ≤ 0 oder K > LENGTH
4	4	STRUCT	Index L ist ungültig: L ≤ 0 oder L > LENGTH
5	4	STRUCT	L soll Vater von K werden, K hat aber schon einen Vater
6	4	STRUCT	L soll Vater von K werden, K ist aber der Patriarch der Sippe in der L enthalten ist.
7	4	STRUCT	K soll von L gelöst werden, K hat aber keinen Vater
8	4	STRUCT	K soll gelöscht werden, ist aber nicht von Vater oder Sohn getrennt
9	-	-	nicht definiert
10	-	-	nicht definiert
11	12	ISTPOS	Index LAST=0 und daher ungültig

* Siehe Beschreibung von ERROR2, Kap.II.6.1.2

nr	s	Quelle	Erläuterung
12	12	ISTPRE	Index LAST=0 und daher ungültig
13	12	ISTEND	Index LAST=0 " " "
14	12	DEF	Anzahl LEV der Indices des DA. A ist außerhalb des zuläßigen Bereiches: LEV < 0 oder LEV > 30
15	12	SWSUB	Anzahl LEV der Indices des DA. A ist LEV < 0 oder LEV > 30
16	4	DYARPR	Die Indexliste benötigt zum Drucken mehr als 68 Zeichen; es werden nur die ersten 68 gedruckt.
17	12	DEF	Subarray ist länger als $32767 (=2^{15}-1)$
18	12	STINIT	Baumstrukturtabelle soll länger als 32767 werden (Fehler in LENGTH).

(Für $0 \leq s < 8$ kann weitergerechnet werden,
für $8 \leq s \leq 16$ soll die Programmausführung in ERROR2 abgebrochen werden).

A n h a n g 1

Allgemeines zu DYNAMIC ARRAYs [7,8,10,11]

1. Die sprachlichen Möglichkeiten der DYNAMIC ARRAYs

Ein DYNAMIC ARRAY kann beispielsweise mit folgenden Statements deklariert werden [10] :

```
DYNAMIC ARRAY    (A, DA(D), A(I))
DEFINE  A, 2, PØINTER, HIGH
DEFINE  A (1), 2, PØINTER, LOW
DEFINE  A (1,1), 1, FULL
DEFINE  A (1,2), 2, DØUBLE
DEFINE  A (2), 1, HALF, LOW, STEP=5
```

Das erste Statement deklariert die Variablen mit den Namen A, DA und IA als DYNAMIC ARRAYs, die als EQUIVALENCE (im Fortran-Sinne) überlagert sein sollen. Bei Zugriffen auf Daten der DAs ist A der Mode REAL, DA der Mode DOUBLE (PRECISION) und IA der Mode INTEGER zugeordnet.

Mit den folgenden Statements wird die Struktur des DA definiert, wie sie in Abb.13 dargestellt ist [8]. Die Angaben PØINTER, FULL, DØUBLE und HALF spezifizieren den Typ des jeweiligen Subarrays (im wesentlichen die Wortlänge). Mit HIGH und LOW (Standard:LOW) wird die Priorität des Subarrays bezüglich seines Verbleibens im Kernspeicher bei Kernspeicherüberlauf festgelegt. STEP=n mit n=1,5,10, 20,50,100 oder 200 spezifiziert die Länge,um die der Subarray automatisch verlängert werden soll, wenn die bis dahin verwendete Länge nicht ausreicht(falls keine STEP-Angabe existiert, wird der Subarray nicht verlängert).

Die DEFINE-Statements bewirken noch nicht die Belegung von Kernspeicherplätzen für den Subarray. Dies geschieht erst bei der ersten Referenz auf ein Element des Subarrays, z.B.bei

A (2,1) = 5.

Vor einer derartigen Referenz ist der entsprechende Subarray intern als "unbenutzt oder zerstört" (unused as yet or destroyed) deklariert. Durch eine derartige Referenz wird der Subarray als "im Kernspeicher (in Core)" deklariert. Bei einer eventuellen Verlagerung des Subarray auf den Hintergrundspeicher erhält der Subarray die Deklaration "auf Hintergrundspeicher (on secondary storage)".

Weitere Statements zur Beeinflussung von DAs sind

a) RELEASE A
   Hierdurch wird A als im Moment nicht benötigt deklariert.

b) DESTROY A
   Hierdurch wird der von A belegte Platz freigegeben.

2. Der Inhalt der POINTER von DYNAMIC ARRAYs

Alle Angaben, die zur Deklaration eines Subarrays gehören (mit Ausnahme des Mode) werden in dem POINTER untergebracht. Der POINTER besteht aus zwei Wörtern zu je 32 Bits. Ein Teil des POINTERs enthält die Addresse des Subarrays, der Rest enthält eine Zustandsbeschreibung des Subarrays. Die genaue Struktur des POINTERs ist [8, 11] zu entnehmen. Die genaue Kenntnis ist jedoch nicht erforderlich, da dem ICETRAN-Programmierer zwei Routinen zur Verfügung stehen, die es erlauben, den Subarrayzustand in ICETRAN festzustellen. Die beiden Routinen haben die Namen ISTAT und IDEF und sind wie folgt aufzurufen [10, S.81] :

a) ISTAT:

```
is = ISTAT (nr, name, i, j, ...)
```

is definiert den Ort des Subarrays:

- is = 0 - in core
- is = 1 - on secondary storage(on disk)
- is = 2 - unused as yet or destroyed

nr definiert den RELEASE-Status:

- nr = 0 - unreleased
- nr = 1 - released

name ist der Name der DA-Variablen

i, j,.... sind die Indices des entsprechenden Subarrays (überflüssig für einen "Base"-Pointer)

b) IDEF:

```
ii = IDEF (n1,n2,n3,name, i,j,...)
```

ii ist die Länge des Subarrays (in Array-Elementen)

n1 definiert die Wortlänge und Typ:

n1	= 0	PØINTER
n1	= 1	HALF
n1	= 2	FULL
n1	= 3	DØUBLE

n2 bezeichnet die Priorität

n2	= 0	LOW
n2	= 1	HIGH

n3 bezeichnet den Vergrößerungs-Step und liefert entsprechend einen der Werte

0, 1, 5, 10, 20, 50, 100 oder 200.

name sowie i,j,...sind wie unter a) definiert.

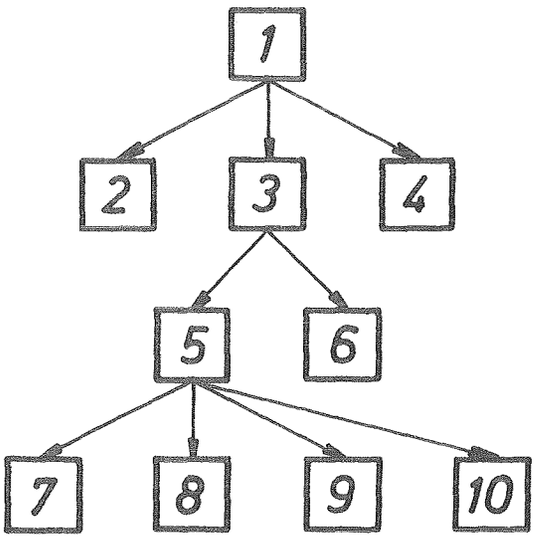


Abb.1: Beispiel einer Baumstruktur

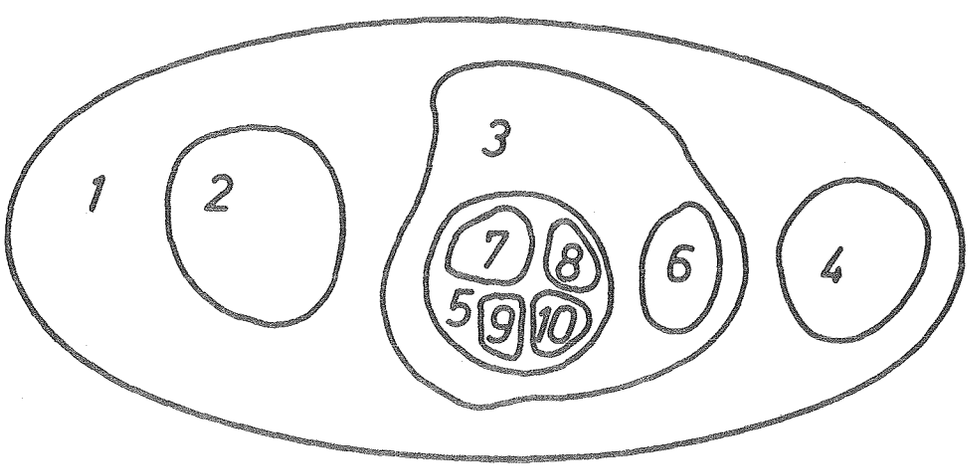


Abb.2: Baum als "Mengenbaum"

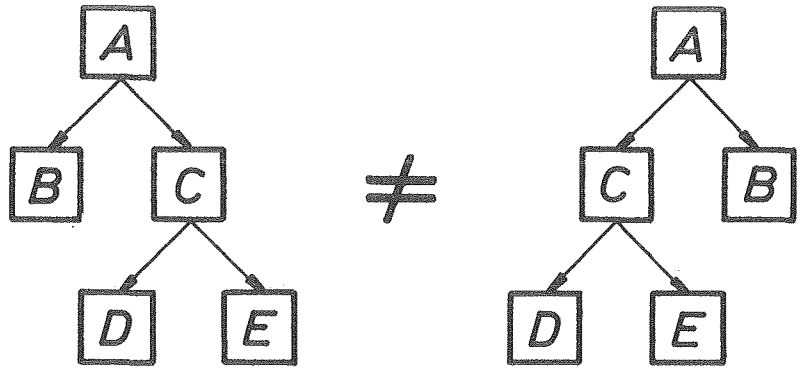


Abb.3: Baum als "Ordnungsbaum"

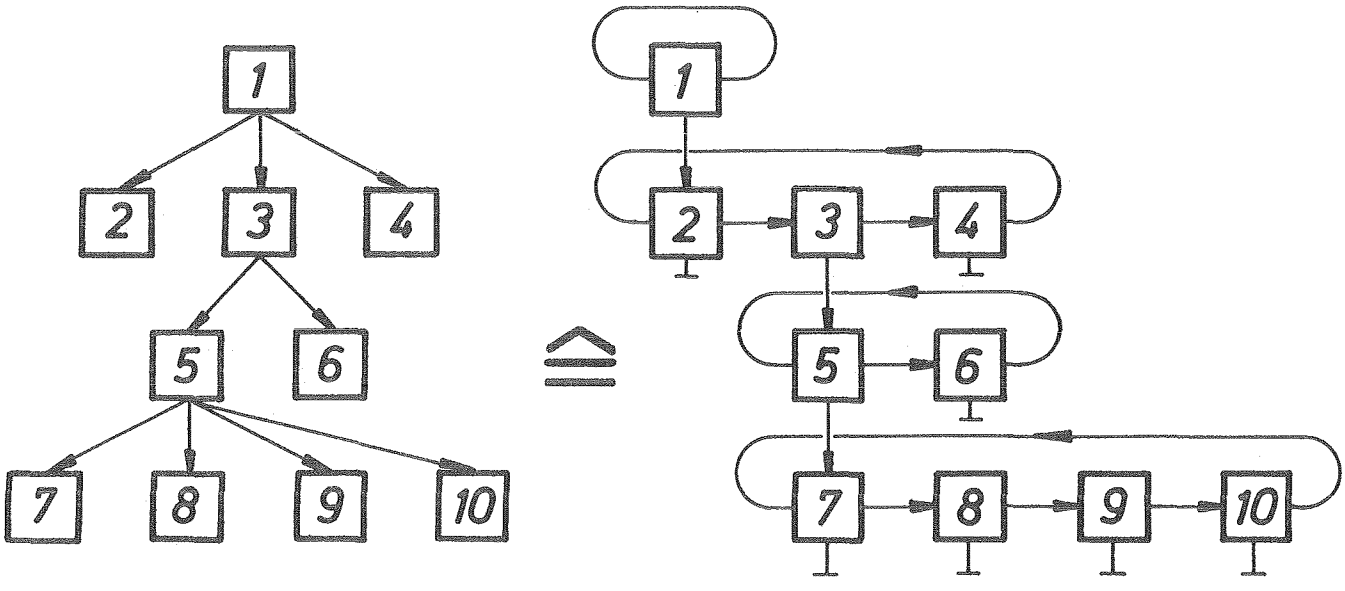


Abb.4: Darstellung eines polynären Mengenbaums durch binäre Knoten mit Ringen

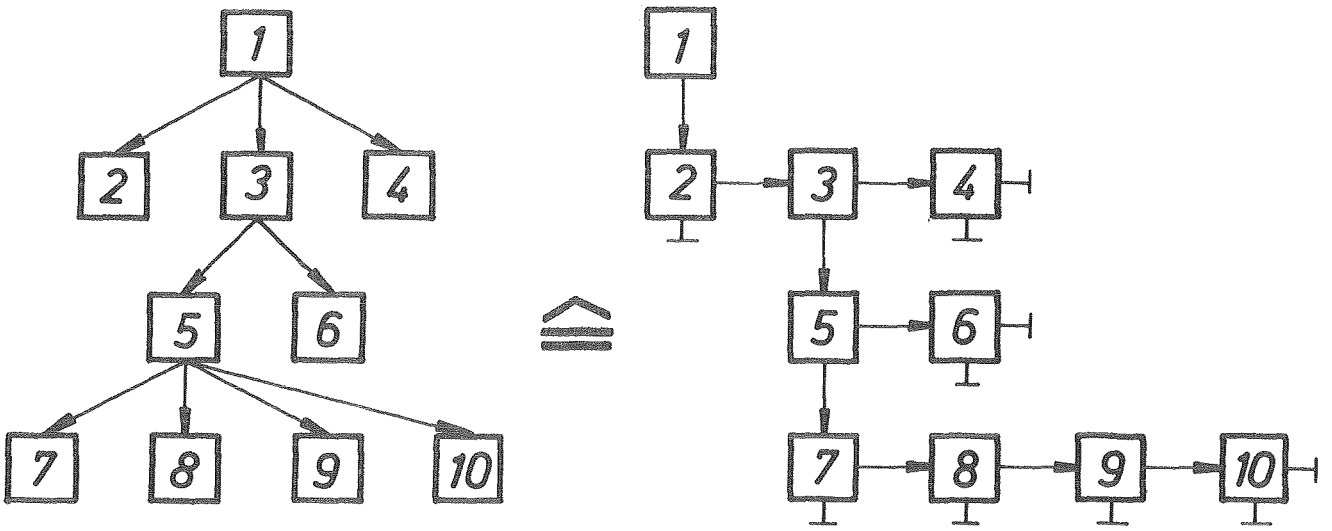


Abb.5: Darstellung eines polynären Ordnungsbaumes durch binäre Knoten ohne Ringe

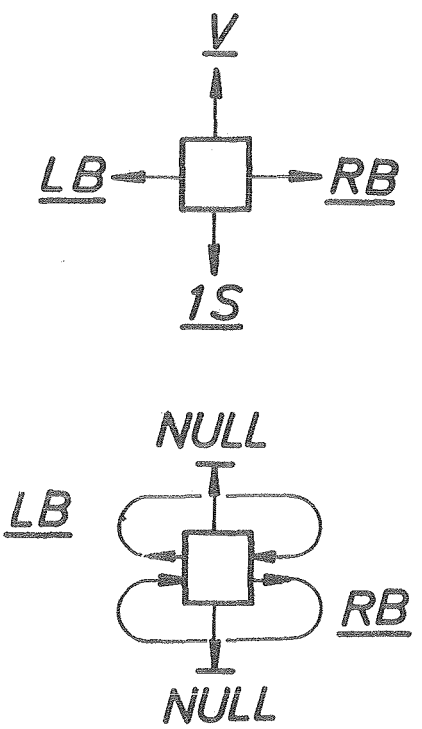


Abb.7: Die verwendeten Zeiger

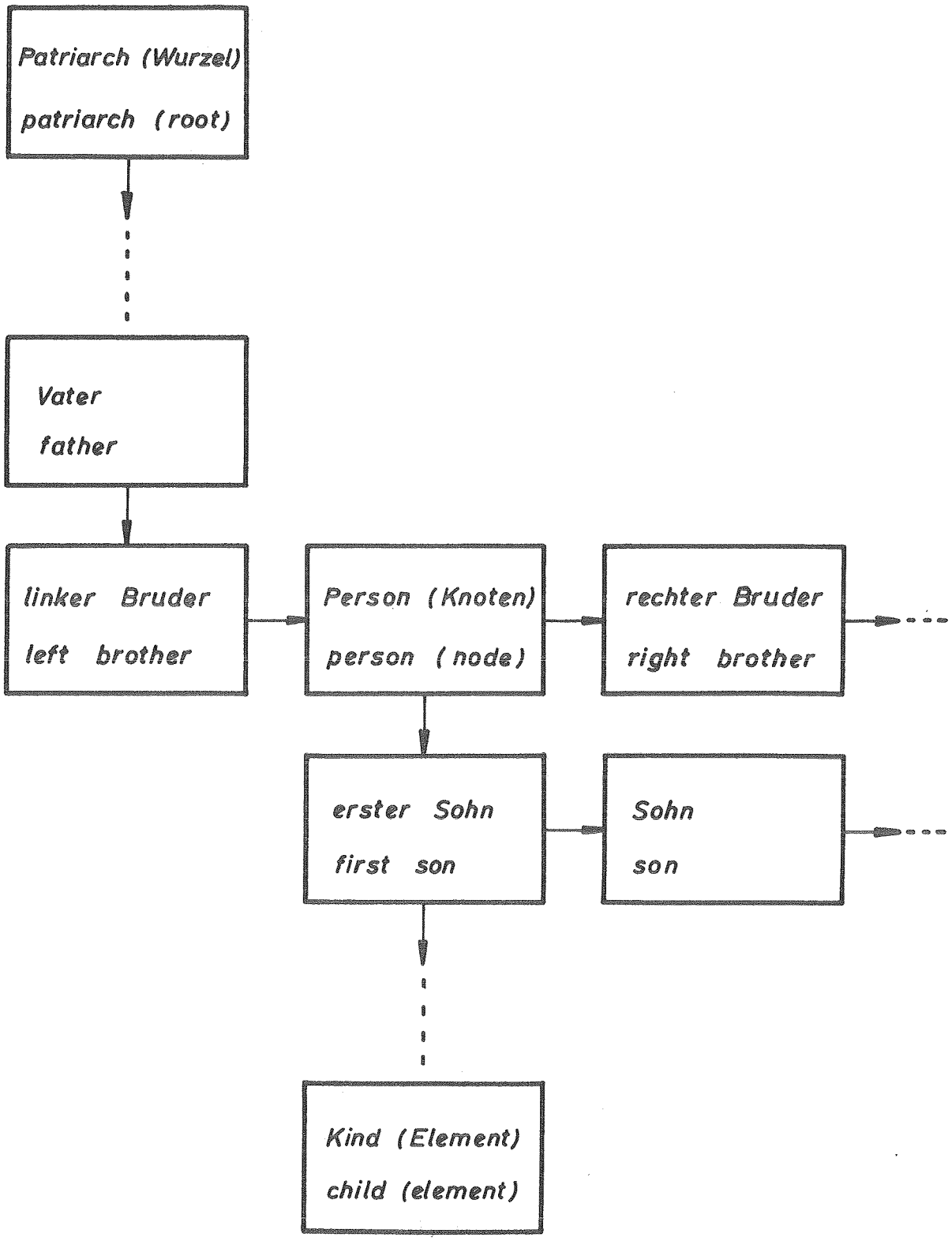


Abb.6: Baum als Familie; Bezeichnungen von Nachbarknoten und ausgezeichneten Knoten des Baums

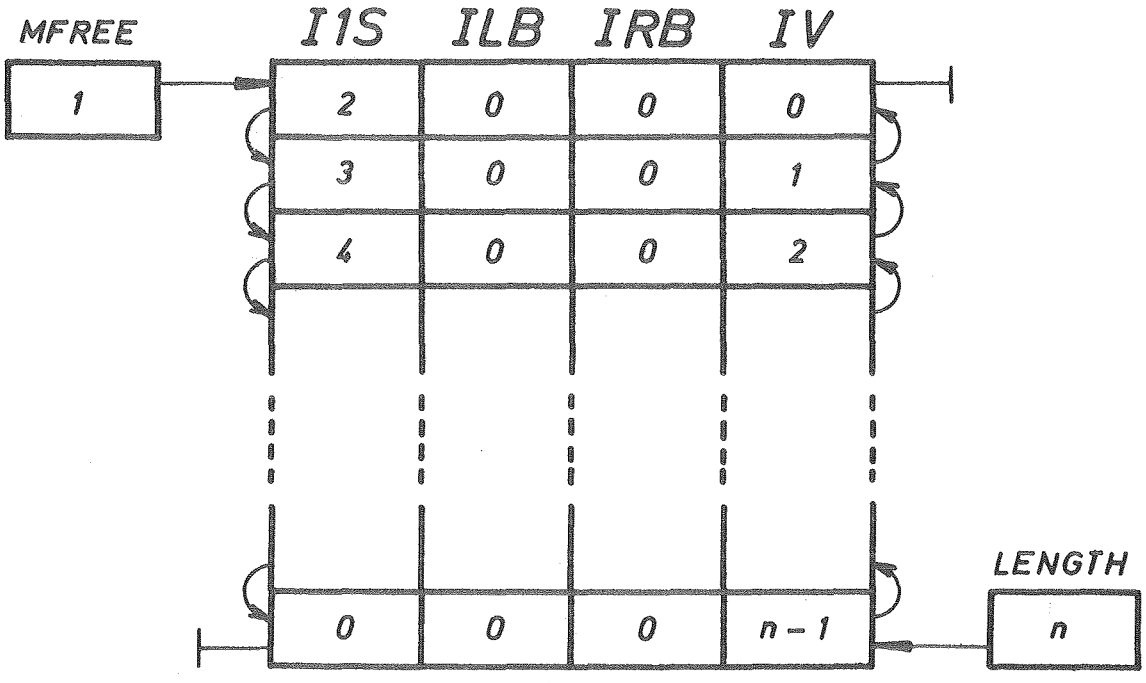


Abb.8: Die Anfangswerte der Baumstrukturtabelle
(Verkettung der Knoten in einer Liste der freien Knoten)

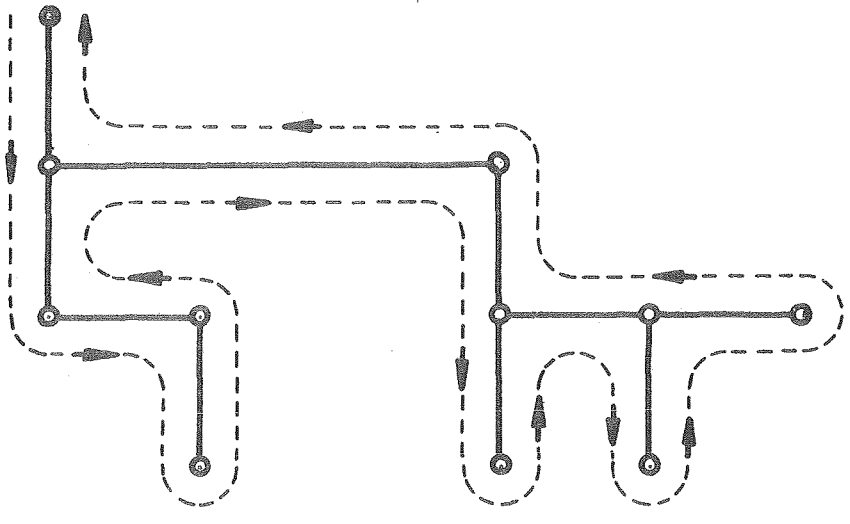
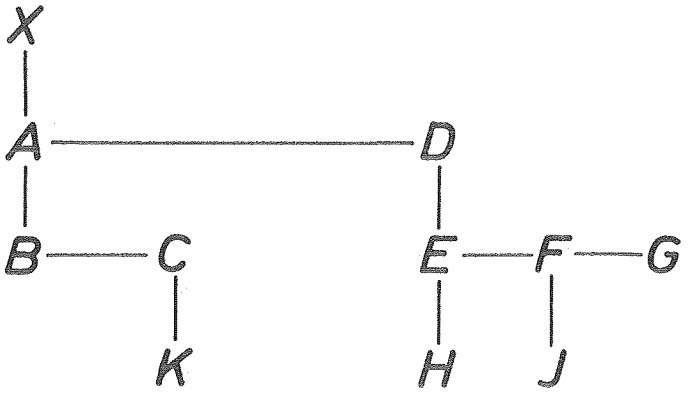


Abb.9: Zur Definition der Abarbeitungsfolgen



Preorder	X	A	B	C	K	D	E	H	F	J	G
Postorder	B	K	C	A	H	E	J	F	G	D	X
Endorder	K	C	B	H	J	G	F	E	D	A	X

Abb.10: Beispiel eines Baums und seiner Abarbeitungsfolgen

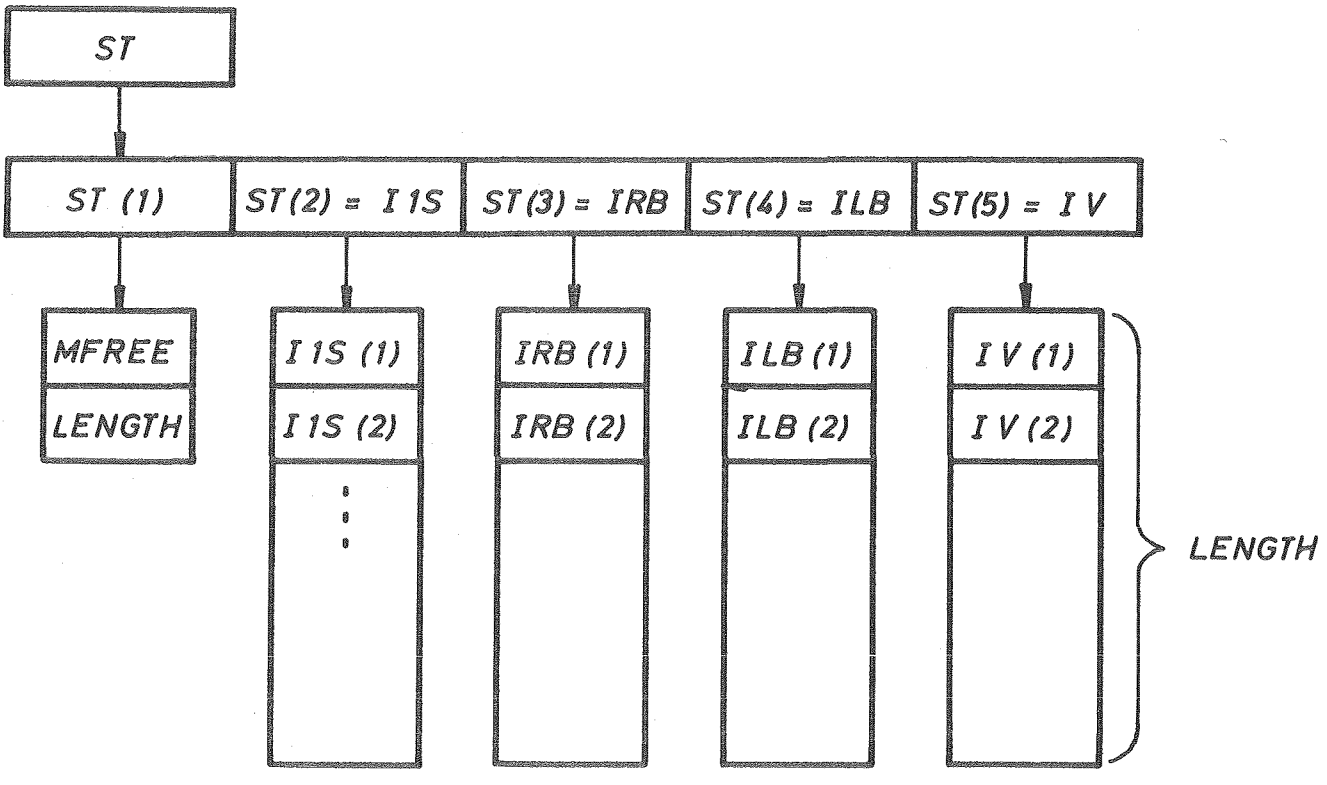


Abb.11: Struktur des DYNAMIC ARRAY ST

```
*STRUCTUR TEST1                    MFREE=      8          LENGTH=      10
*NUMMER      1   2   3   4   5   6   7   8   9  10
*1.SOHN      2   3   0   5   0   0   1   9  10   0
*R.BRUDER    1   4   3   6   5   2   7   0   0   0
*L.BRUDER    1   6   3   2   5   4   7   0   0   0
*VATER       7   1   2   1   4   1   0   0   8   9

STRUCTUR TEST3     IST NICHT DEFINED
```

Abb.12: Beispielsausgabe von STPRNT.

a) definierte Tabelle 'TEST1'
b) undefinierte Tabelle 'TEST3'

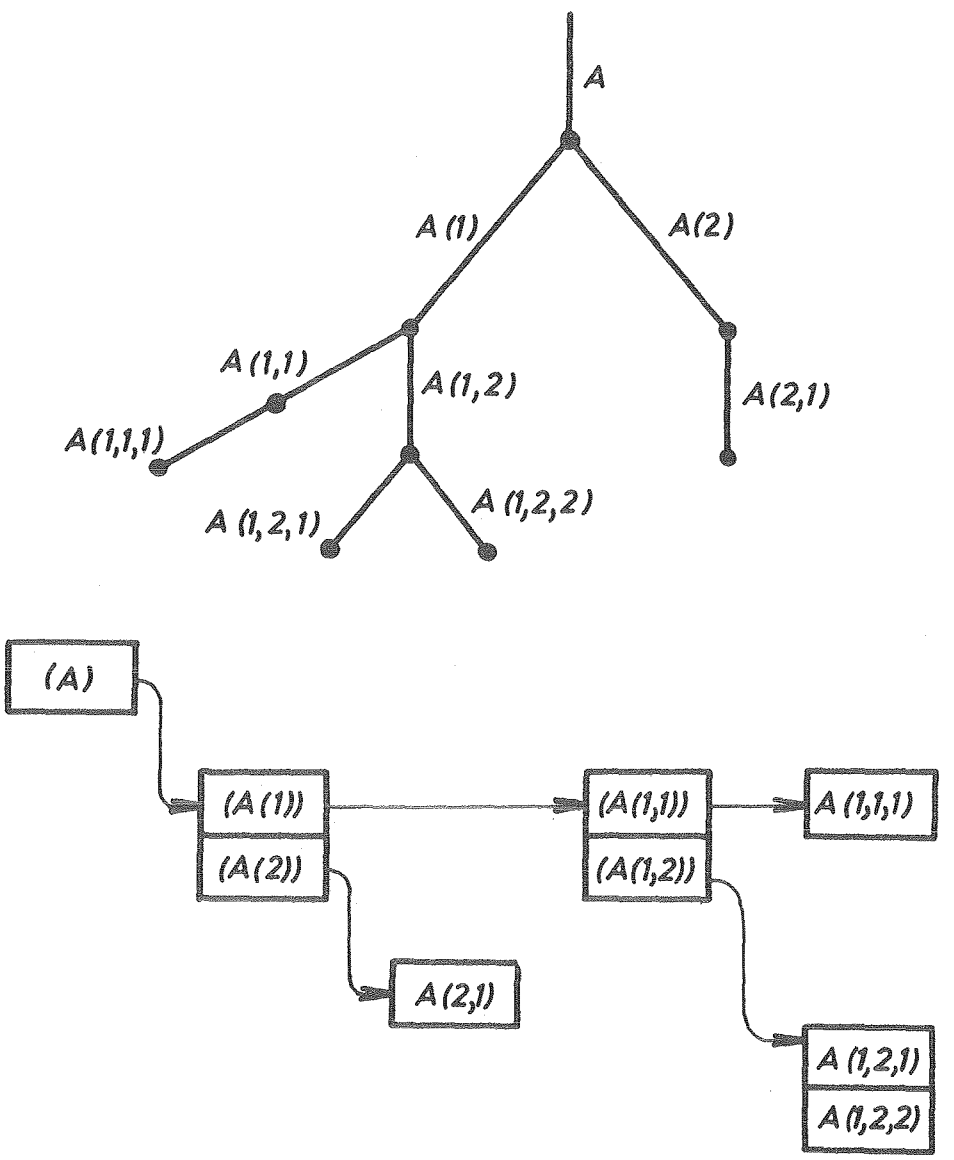


Abb.13: Beispiel einer DYNAMIC ARRAY-Baumstruktur [8]

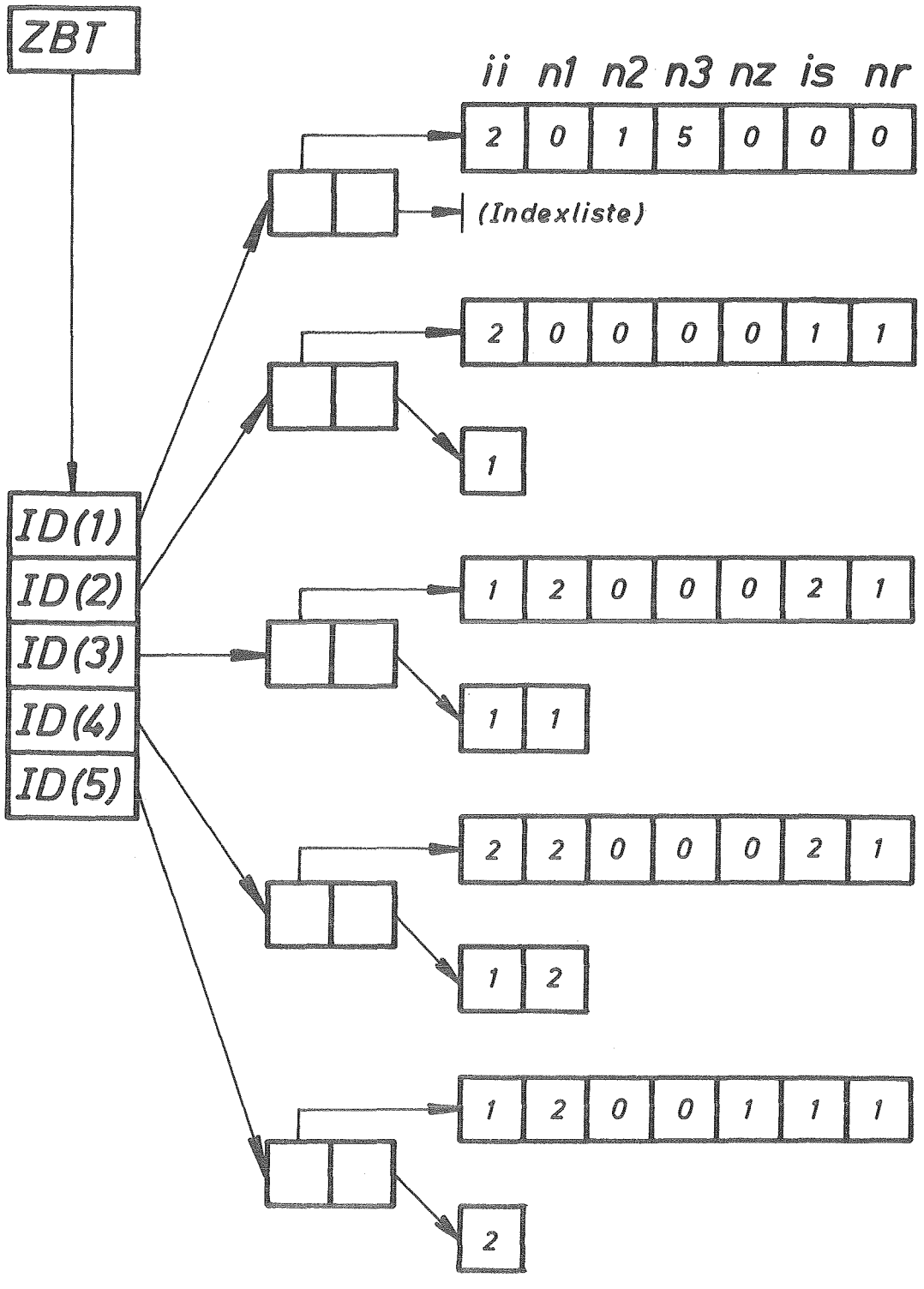


Abb.14: Struktur und möglicher Inhalt der "Zustandsbeschreibungstabelle" für den DYNAMIC ARRAY gem.Abb.13

```
*** DYNAMIC ARRAY TEST     PRINT-DUMP

*LEVEL SUBARRAY                                                                                                    LENGTH TYPE    PRIORITY GROWTH-STEP LOCATION STATUS
*    0 BASE                                                                                                             3 POINTER HIGH            5     IN CORE  UNRELEASED
*    1 (1)                                                                                                             30 HALF    LOW            0     IN CORE  RELEASED
    -9    -8    -7    -6    -5    -4    -3    -2    -1     0     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15    16    17    18    19    20
*    1 (2)                                                                                                              2 POINTER HIGH            1     IN CORE  UNRELEASED
*    2 (2,1)                                                                                                            2 FULL    LOW             0     IN CORE  UNRELEASED
 -0.1000000E 01  0.4280156E 00                                                                                    -1055916032 1080922735
*    2 (2,2)                                                                                                            9 FULL    LOW             0     IN CORE  UNRELEASED
 -0.7237005E 76  0.1544267E-83  0.9265604E-83  0.2522303E-82  0.5250509E-82                                          -2          3          18          49          102
  0.9426003E-82  0.1533972E-81  0.2331844E-81  0.3366503E-81                                                        183         298         453         656
*    1 (3)                                                                                                              2 POINTER LOW             0     IN CORE  UNRELEASED
*    2 (3,1)                                                                                                            2 POINTER LOW             0     IN CORE  UNRELEASED
*    3 (3,1,1)                                                                                                          3 DOUBLE  LOW             0     IN CORE  UNRELEASED
  0.9999999999999990 00  0.2000000000000000D 01  0.3000000000000000D 01
*    3 (3,1,2)                                                                                                          4 DOUBLE  LOW             0     IN CORE  UNRELEASED
 -0.1077395446804680 43 -0.1077395446804680 43 -0.5255241011601570 11 -0.2283475321156840 09    TEST1     TEST2     ICES      GRDYARPR
*    2 (3,2)                                                                                                            0                       0     UNUSED
*END OF DUMP OF DYNAMIC ARRAY TEST
```

Abb.15: Beispiel der Ausgabe von DYARPR(a,b,'TEST')

```
      DYNAMIC ARRAY(TE,ITE,DTE(D))
```

```
      DEFINE TE,3,POINTER,STEP=5,HIGH
      DEFINEITE(1),30,HALF
      DO 4001 I=1,30
4001  ITE(1,I)=(I-10)
      RELEASE ITE(1)
      DEFINE TE(2),2,POINTER,HIGH,STEP=1
      DEFINE TE(2,1),2
      TE(2,1,1)=-1.
      TE(2,1,2)= 0.42801569
      DEFINE ITE(2,2),9
      DO 4002 I=1,9
4002  ITE(2,2,I)= I-3+I*I*I-I*I
      DEFINE TE(3),2,POINTER
      DEFINE TE(3,1),2,POINTER
      DEFINEDTE(3,1,1),3,COUBLE
      DO 4003 I=1,3
4003  DTE(3,1,1,I)=I*1.DO
      DEFINE DTE(3,1,2),4,DOUBLE
      DTE(3,1,2,1)='TEST1'
      DTE(3,1,2,2)='TEST2'
      DTE(3,1,2,3)='ICES'
      DTE(3,1,2,4)='GRDYARPR'
      LINK DYARPR(TE,6,'TEST')
```

Abb.16: Programmabschnitt zur Definition des DYNAMIC ARRAY, dessen Inhalt mit DYARPR gemäß Abb.15 ausgedruckt wird