Datenstrukturen und Algorithmen

Universität Lüneburg

Prof. Dr. rer. nat. Ulrich Hoffmann

Dezember 2005

Das vorliegende Skript dient als begleitende Unterlage für die Veranstaltungen *Datenstrukturen und Algorithmen* und *Systemnahe Programmierung* an der Universität Lüneburg. Die Durcharbeitung des Skripts ersetzt nicht den Besuch der Veranstaltung, da dort zusätzlich Zusammenhänge, ergänzende Sachverhalte und im Skript noch nicht ausgeführte Inhalte behandelt werden.

Inhaltsverzeichnis

Literaturauswahl

[AHU] Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

[A/.] Ausiello, G.; Crescenzi, P.; Gambosi, G.; Kann, V.; Marchetti-Spaccamela, A.; Protasi, M.: **Complexity and Approximation**, Springer, 1999.

[BLU] Blum, N.: **Theoretische Informatik**, Oldenbourg, 1998.

[CHA] Chaudhuri, P.: **Parallel Algorithms**, Prentice Hall, 1992.

[D/K] Doberenz, W.; Kowalski, T.: **Borland Delphi 7, Grundlagen und Profiwissen**, Carl Hanser, 2002.

[G/D] Güting, R.H.; Dieker, S.: **Datenstrukturen und Algorithmen**, 2. Aufl., Teubner, 2003.

[HER] Herrmann, P.: **Rechnerarchitektur**, 3. Aufl., Vieweg, 2002.

[HOF] Hoffmann, U.: **Einführung in die systemnahe Programmierung**, de Gruyter, 1990.

[HRO] Hromkovič, J.: **Randomisierte Algorithmen**, Teubner, 2004.

[IOP] International Organisation for Standardization: **Specification for Computer Programming Language Pascal**, ISO 7185, 1985.

[KN1] Knuth, D.E.: **The Art of Computer Programming Vol. 1: Fundamental Algorithms**, 3. Aufl., Addison-Wesley, 1997.

[KN2] Knuth, D.E.: **The Art of Computer Programming Vol. 3: Searching and Sorting**, 2. Aufl., Addison-Wesley, 1998.

[KN3] Knuth, D.E.: **Arithmetik**, Springer, 2001.

[OTT] Ottmann, T. (Hrsg.): **Prinzipien des Algorithmenentwurfs**, Spektrum, 1998.

[O/W] Ottmann, T.; Widmayer, P.: **Algorithmen und Datenstrukturen**, 4. Aufl., Spektrum, 2002.

[PAS] Object-Pascal-Sprachdefinition in der Hilfefunktion der Borland Delphi Entwicklungsumgebung

[P/Z] Pratt, T.; Zelkowitz, M.: **Programmiersprachen**, Prentice Hall, 1998.

[S/G] Solymosi, A.; Grude, U.: **Grundkurs Algorithmen und Datenstrukturen in JAVA**, 3. Aufl., Vieweg, 2002.

[THI] Thies, K.-D.: **80486 Systemsoftware-Entwicklung**, Hanser, 1992.

[W/C] Wilson, L.B; Clark, R.G.: **Comparative Programming Languages**, Addison-Wesley, 1988.

1 Einleitung

Das vorliegende Skript behandelt Datenstrukturen und Algorithmen als Grundlage praktischer Anwendungen. Von den heute üblicherweise verwendeten Programmiersprachen mit ihren Entwicklungsumgebungen werden viele dieser Datenstrukturen in Form von Objektklassen zur Verfügung gestellt. Allerdings bleibt die Implementierung dem Anwender weitgehend verborgen. Eine entsprechende Aussage gilt für eine Reihe von Basisalgorithmen, die den Anwendungen in Form von Methoden zugänglich gemacht werden. Häufig ist es jedoch angebracht, über Detailkenntnisse zu verfügen, insbesondere dann, wenn aus mehreren möglichen Methoden die für die jeweilige Anwendung geeignete Methode auszuwählen ist. Hierbei sind sowohl Effizienzgesichtspunkte als auch Implementierungsaspekte zu berücksichtigen.

Das Skript dient als Lehr- und Lernunterlage für die Veranstaltungen „Datenstrukturen und Algorithmen“. Aus dem breiten Spektrum möglicher Themen wird dazu eine (subjektive) Auswahl getroffen. Ihr Fokus liegt dabei auf der Vermittlung von Kenntnissen, die letztlich dazu führen sollen, Anwendungsprobleme in bessere Programme umzusetzen, die grundlegende Datenstrukturen und Basisalgorithmen verwenden. Dazu ist es erforderlich, diese Datenstrukturen und Basisalgorithmen im Detail zu verstehen. Weiterhin werden allgemeingültige Lösungsmethoden exemplarisch vorgestellt.

Die getroffene Auswahl beinhaltet

- die Darstellung systemnaher Aspekte der Programmierung (rechnerinterne Darstellung von Daten, Darstellung von Daten in höheren Programmiersprachen und ihre Abbildung auf rechnerinterne Objekte, Implementierungskonzepte in höheren Programmiersprachen) in den Kapiteln 2 und 3
- die Bereitstellung von Hilfsmitteln für die Analyse von Algorithmen, insbesondere der Komplexität ihrer Ausführung, ohne in die Tiefe der Theoretischen Informatik einzusteigen in Kapitel 4
- die Darstellung von Basisdatenstrukturen und –algorithmen in den Kapiteln 5 und 6
- die Behandlung allgemeiner Lösungsstrategien an ausgewählten Beispielen in Kapitel 7
- die Behandlung spezieller Lösungsansätze in Kapitel 8.

2 Datenobjekte

Grundlage jeder höheren Programmiersprache ist das darin enthaltene Datenkonzept. Programmiersprachen wie Pascal, C++ oder Java verfügen über ein ausgeprägtes Typkonzept, das die Verwendung selbstdefinierter Datentypen erlaubt, andererseits aber die Regeln der Typisierung einzuhalten erzwingt. Eine Umgehung des Typkonzepts, wie es in COBOL oder C oder besonders in Assemblersprachen durch Überlagerung von Datendefinitionen üblich ist, erhöht nicht die Flexibilität, wohl aber die Fehleranfälligkeit der Programmierung. Die Zuverlässigkeit eines Programms wird im allgemeinen dadurch erhöht, dass der Compiler die Kompatibilität verwendeter Daten, die Einhaltung von Feldgrenzen oder Wertebereichen schon zur Übersetzungszeit des Programms prüft. Andererseits gibt es Situationen in der systemnahen Programmierung, in denen die Einhaltung eines stringenten Typkonzepts problematisch erscheint. Hier haben sich beispielsweise die semantischen Spracherweiterungen des Pointerkonzepts, die ein Dialekt wie Object Pascal [PAS, D/K] gegenüber dem in [IOP] definierten Pascal-Standard enthält, als sehr nützlich erwiesen.

In den folgenden Unterkapiteln werden Datenobjekte auf zwei Abstraktionsebenen betrachtet:

- in einer möglichen rechnerinternen Darstellung, beispielsweise in seiner Darstellung im Arbeitsspeicher oder der Peripherie eines Rechners. Diese Darstellungsform ist maschinenabhängig. Die Kenntnis der Umsetzung von Anwenderdefinitionen in maschinennahe Darstellung ist spätestens bei der Fehlersuche hilfreich bzw. zwingend erforderlich, nämlich dann, wenn ein Speicherauszug eines Programms zu interpretieren ist
- in Form der Deklarationen in einer höheren Programmiersprache. Diese Abstraktionsebene ist anwendungsbezogen und (bis auf wenige Details) maschinenunabhängig, auch wenn es je nach Programmiersprache noch implizite Beziehungen zu einer Modellvorstellung eines möglichen verwendeten Rechners gibt.

Ein Datenobjekt „existiert“ erst, wenn es eine Adresse im virtuellen Adressraum hat oder wenn ihm Speicherplatz auf der Peripherie zugewiesen wurde. Sein Datentyp beschreibt, auf welche Weise es im Rechner abgelegt wird und welche Operationen mit ihm erlaubt sind. Die Abstraktionsebene der internen Darstellung gibt daher ein Datenobjekt vollständig wieder. Die höheren Abstraktionsebenen, die Datenobjekte in der Assembler-Sprache des eingesetzten Rechners oder in einer höheren Programmiersprache deklarieren, *beschreiben* lediglich Datenobjekte und die Art, wie sie später zur Laufzeit „ins Leben gerufen“ und wertmäßig manipuliert werden. Durch eine Deklaration eines Datenobjekts in einem Programm, das in einer höheren Programmiersprache geschrieben ist, wird noch kein Datenobjekt erzeugt, auch wenn der geläufige Sprachgebrauch einer Deklaration diese Vorstellung suggeriert. Im folgenden kann diese Vorstellung der Erzeugung eines Datenobjekts durch eine Deklaration in einer Programmiersprache trotzdem zugrunde gelegt werden. Ein Anwender möchte die internen Details seiner deklarierten Datenobjekte ja nicht sehen; daher hat er seine Problemlösung in

einer höheren Programmiersprache formuliert. Aus seiner Sicht wird durch eine Deklaration ein Datenobjekt erzeugt.

Ein Datenobjekt zeichnet sich durch verschiedene Charakteristika aus (Abbildung 2-1).

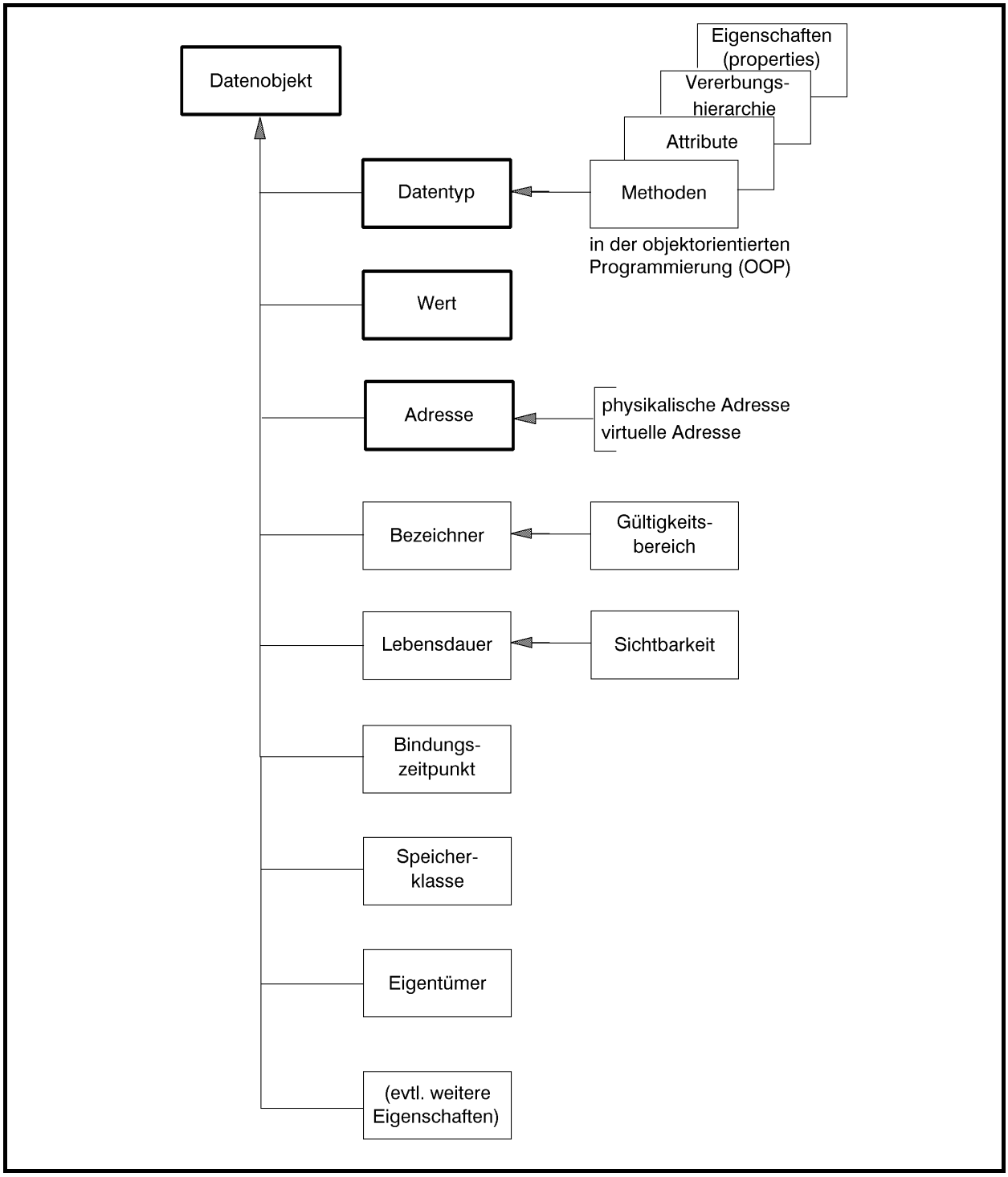


Abbildung 2-1: Charakteristische Eigenschaften eines Datenobjekts

Die minimale Menge dieser Eigenschaften (stark umrandeten Felder in Abbildung 2-1) beinhaltet:

- den **Datentyp des Datenobjekts**, d.h. die Menge der Attribute, insbesondere die Darstellungsform, den möglichen Wertebereich und die mit dem Objekt erlaubten Operationen. In der **objektorientierten Programmierung** lassen sich durch den Datentyp neben den Grundoperationen weitere Operationen in Form von Methoden und weitere typabhängige Attribute wie Vererbungshierarchien definieren
- den **gegenwärtige Wert des Datenobjekts**, der im Lauf der Zeit durch das Programm verändert werden kann. Vor der ersten Wertzuweisung an das Datenobjekt ist der Wert im allgemeinen undefiniert
- die **Adresse des Datenobjekts**, d.h. die Stelle, an der es sich im System befindet; diese Adresse ist eindeutig bestimmt, auch wenn sie dem Anwender nicht immer direkt zugänglich ist.

Auf der Abstraktionsebene einer höheren Programmiersprache kommen zusätzliche Eigenschaften eines Datenobjekts hinzu wie

- der **Bezeichner (Name) des Datenobjekts**, der angibt, wie das Datenobjekt im Programm angesprochen wird. Der **Gültigkeitsbereich des Bezeichners** regelt die Verwendbarkeit dieses Bezeichners
- die **Lebensdauer des Datenobjekts**, die die Zeitspanne umfasst, in der das Datenobjekt im System existiert; dabei kann es möglich sein, dass ein Datenobjekt zeitweise nicht sichtbar ist, d.h. dass es nicht angesprochen bzw. adressiert werden kann.

Weitere Charakteristika, die nicht alle Programmiersprachen vorsehen, sind beispielsweise

- der **Bindungszeitpunkt des Bezeichners** an das Datenobjekts: in einigen Programmiersprachen wird ein Datenobjekt an einen Bezeichner erst dann gebunden, d.h. eingerichtet, wenn eine Anweisung zum ersten Mal durchlaufen wird, die den Bezeichner enthält. Je nach Situation wird jetzt erst der Datentyp festgelegt und damit implizit die mit dem Datenobjekt anschließend erlaubten Operationen. Ein derartiges Verhalten ist natürlich nur in nicht streng typisierten Programmiersprachen möglich, in denen Typfestlegungen erst zur Laufzeit erfolgen, oder falls der betroffene Sprachdialekt eine Aufweichung eines an sich stringenten Typkonzepts zulässt
- die **Speicherklasse**, die bestimmt, in welchem Teil des Rechners das Datenobjekt angelegt werden soll
- der **Eigentümer**, d.h. derjenige Systemteil, der die Rechte bezüglich des Zugriffs auf das Datenobjekt kontrolliert.

Eine der wichtigsten Eigenschaften eines Datenobjekts ist sein **Datentyp**, da dieser letztlich festlegt, welche Operationen mit dem Datenobjekt erlaubt sind. Die übliche Art, ein Datenobjekt in einem Programm zu definieren (deklarieren), das in einer höheren Programmiersprache geschrieben ist, besteht daher in der Angabe des Typs des Objekts und damit implizit der zulässigen Operationen und Methoden, die auf das Datenobjekt dieses Typs anwendbar sind, und der Angabe des Bezeichners für das Datenobjekt, etwa in der in Pascal üblichen Form:

als Variable	als typisierte Konstante

`TYPE data_typ = ...;`
 	
`VAR datenobjekt : data_typ;`
	`TYPE data_typ = ...;`
	
	`CONST datenobjekt : datentyp = ...`
	`{ Anfangswert };`

Abbildung 2-2 gibt einen Überblick über die wichtigsten Datentypen in einem IT-System. Unter **Grunddatentypen** werden dabei Datentypen verstanden, über die die meisten IT-Systeme (in unterschiedlicher Ausprägung) verfügen. Diese sind auch in Programmiersprachen verwendbar, wobei zusätzlich **abgeleitete Datentypen** hinzukommen. **Anwendungsorientierte Datentypen** werden häufig von Entwicklungsumgebungen oder Spracherweiterungen in Form von Programmbibliotheken bereitgestellt und erlauben unter Verbergen der internen Details die einfache Nutzung komplexer Funktionalität.

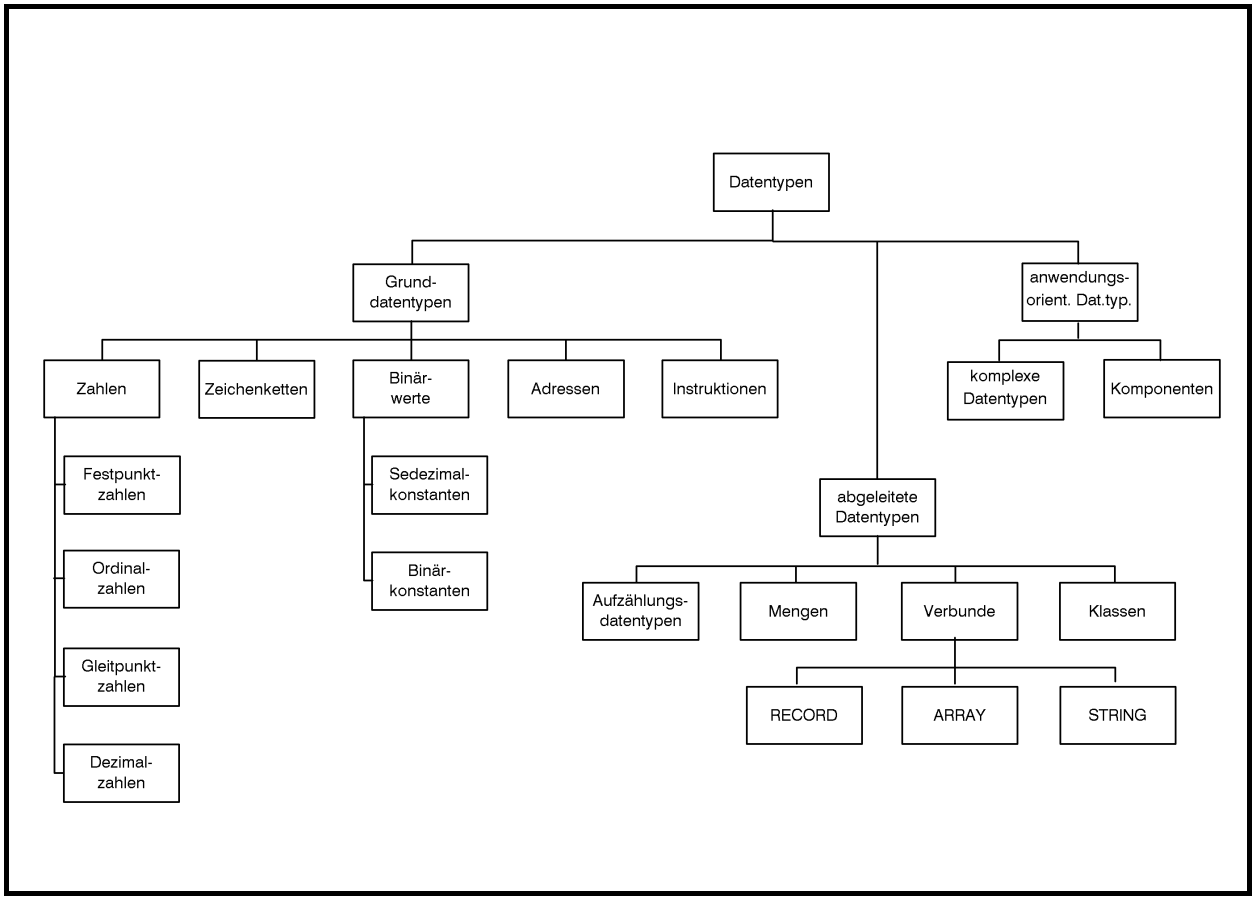


Abbildung 2-2: Datentypen in einem IT-System

2.1 Grundlegende Datentypen eines Rechners

Dieses Kapitel beschreibt die Grunddatentypen, über die ein Rechner üblicherweise verfügt und auf denen die Datentypen auf höheren Abstraktionsebenen aufbauen. Ein Datenobjekt eines Programms, das in einer höheren Programmiersprache definiert und eventuell anwendungsspezifisch strukturiert ist, muss letztlich auf ein (eventuell zusammengesetztes) **internes Datenobjekt** übersetzt werden. Die Art der Abbildung und damit der internen Datendarstellung ist Rechnertyp-abhängig. Exemplarisch sollen hier der INTEL 80x86-Prozessorfamilie und die in der kommerziellen Großrechnerwelt häufig anzutreffenden Bytemaschinen vom Typ IBM /390 gegenübergestellt werden ([THI], [HOF]). Beide Rechner verwenden ähnliche Grunddatentypen, die sich jedoch an einigen Stellen wesentlich unterscheiden.

Die kleinste adressierbare Einheit ist das **Byte** mit jeweils 8 **Bits**. Die einzelnen Bits innerhalb eines Bytes, auch wenn sie nicht einzeln durch Adressen ansprechbar sind, sind ***von rechts nach links*** von 0 bis 7 durchnumeriert. Die Nummern bezeichnet man als **Bitpositionen** (innerhalb des Bytes). Ein Bit kann die Werte 0 oder 1 annehmen, so dass für den Inhalt eines Bytes $2^8 = 256$ verschiedene Bitmuster möglich sind. Eine Bitfolge der Form $[b_7 b_6 \ldots b_1 b_0]$ kann als Binärzahl mit Wert $\sum_{i=0}^{7} b_i \cdot 2^i$ interpretiert werden. Ein Byte kann also in dieser Interpretation die Dezimalwerte 0, 1, ..., 255 enthalten.

Üblicherweise fasst man *zur Schreibvereinfachung und zur besseren Lesbarkeit* jeweils 4 nebeneinanderliegende Bitwerte (von rechts beginnend, wobei eventuell links führende binäre Nullen zur Auffüllung gedacht werden müssen) zu einer Sedezimalziffer zusammen und erhält damit die gleichwertige Sedezimaldarstellung. Als Ziffern kommen in der Sedezimaldarstellung die sechzehn Zeichen 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F vor.

Um zu unterscheiden, ob eine Zahl als Binär- , Sedezimal- bzw. Dezimaldarstellung anzusehen ist, wird in nichteindeutigen Situationen die Basis 2, 16 bzw. 10 als Index angehängt.

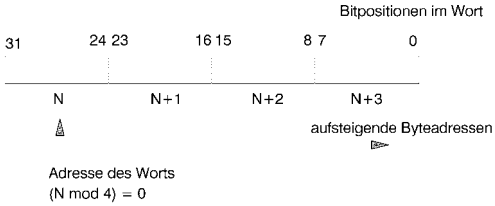
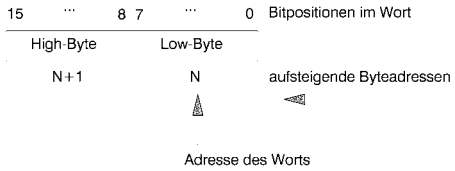
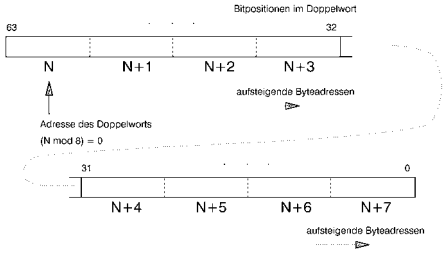
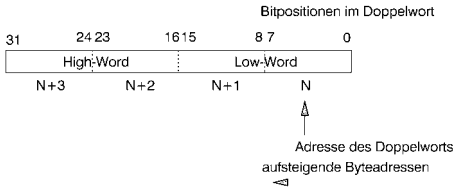
Byteinhalt		
Binärdarstellung (Basis 2) $[b_7 b_6 \ldots b_1 b_0]_2$	Sedezimaldarstellung (Basis 16) $[s_1 s_0]_{16}$	Dezimalwert (Basis 10) $\sum_{i=0}^{7} b_i \cdot 2^i$
0000 0000	00	0
0000 0001	01	1
0000 0010	02	2
0000 0011	03	3
0000 0100	04	4
0000 0101	05	5
0000 0110	06	6
0000 0111	07	7
0000 1000	08	8
0000 1001	09	9
0000 1010	0A	10
0000 1011	0B	11
0000 1100	0C	12
0000 1101	0D	13
0000 1110	0E	14
0000 1111	0F	15
0001 0000	10	16
0001 0001	11	17
...	...	...
0001 1111	1F	31
0010 0000	20	32
0010 0001	21	33
...	...	...
0010 1111	2F	47
0011 0000	30	48
0011 0001	31	49
...	...	...
0011 1111	3F	63
...	...	...
1111 0000	F0	240
1111 0001	F1	241
...	...	...
1111 1110	FE	254
1111 1111	FF	255

Häufig werden aufeinanderfolgende Bytes zu den weiteren logischen Einheiten **Halbwort**, **Wort**, **Doppelwort** und **Quadwort** zusammengefasst. Innerhalb dieser Einheiten werden wieder einzelne Bitpositionen von 0 beginnend gezählt. Zu beachten ist die **Byteanordnung**: beim IBM /390-Rechner liegen die höherwertigen Bits an den niedrigen Adressen, während

beim INTEL 80x86-Prozessor die höherwertigen Bits an den höheren Adressen stehen. **Eine Gruppe (Halbwort, Wort, Doppelwort bzw. Quadwort) kann als ganzes durch eine Adresse angesprochen werden.** Diese Adresse meint in beiden Architekturen den Anfang der Gruppe, d.h. die kleinste Byteadresse innerhalb der Gruppe. Bei IBM /390 ist dies das Byte mit den höchstwertigen Bits, bei INTEL 80x86 das Byte mit den niedrigstwertigen Bits; entsprechend nennt man die IBM /390-Rechner **Big-Endian**- und die INTEL 80x86-Rechner **Little-Endian**-Maschinen[1]. Außerdem meinen die jeweiligen Gruppierungsbegriffe in den Architekturen unterschiedliche Byteanzahlen. Abbildung 2.1-1 fasst die Gruppierungen zusammen.

	Big Endian (z.B. IBM /390)	Little Endian (z.B. INTEL 80x86)
Byte	7 . . . 0 Bitpositionen im Byte	7 . . . 0 Bitpositionen im Byte
Halbwort	2 hintereinanderliegende Bytes, deren erstes bei einer geraden Adresse beginnt **(Ausrichtung auf Halbwortgrenze**). 15 ··· 8 7 ··· 0 Bitpositionen im Halbwort N N+1 aufsteigende Byteadressen (N mod 2) = 0	---

[1] Die Ausdrücke “Big Endian” und “Little Endian” entstammen dem Buch “Gullivers Reisen” von Jonathan Swift (1727). In diesem Buch wird der Krieg zwischen den Königreichen Lilliput und Blefuscu beschrieben. Der Krieg war ausgebrochen, weil beide Parteien sich nicht darauf einigen konnten, ob beim Frühstück das Frühstücksei am spitzen oder am stumpfen Ende aufgeschlagen werden sollte (siehe auch [HER]).

Wort	4 hintereinanderliegende Bytes (2 Halbworte), deren erstes bei einer durch 4 teilbaren Adresse beginnt (**Ausrichtung auf Wortgrenze**). Zu beachten ist, dass das Byte, das die höchste Bitposition im Wort enthält, auf der niedrigsten Byteadresse innerhalb des Worts steht. 	2 hintereinanderliegende Bytes, die an einer beliebigen Adresse beginnen können. Das Byte, das Bitposition 0 enthält, heißt **Low-Byte**, das Byte mit Bitposition 15 **High-Byte**. In der Reihenfolge der Byteadressierung gilt der Grundsatz „***Low-Byte first***“. Beginnt ein Wort an einer geraden Adresse, so heißt es **ausgerichtetes Wort** (word aligned). 
Doppelwort	8 hintereinanderliegende Bytes (2 Worte), deren erstes bei einer durch 8 teilbaren Adresse beginnt (**Ausrichtung auf Doppelwortgrenze**) 	4 hintereinanderliegende Bytes (2 Worte), die an einer beliebigen Adresse beginnen können. Das Wort, das Bitposition 0 enthält, **heißt Low-Word**, das Wort mit Bitposition 31 **High-Word**. In der Reihenfolge der Byteadressierung gilt der Grundsatz „***Low-Word first***“ (und innerhalb eines Worts „***Low-Byte first***“). Beginnt ein Doppelwort an einer durch 4 teilbaren Adresse, so heißt es **ausgerichtetes Doppelwort** (doubleword aligned). 

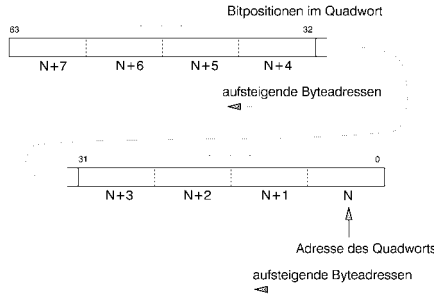
Quadwort	---	8 hintereinanderliegende Bytes (2 Doppelworte), die an einer beliebigen Adresse beginnen können. Das Wort, das Bitposition 0 bis 31 enthält, heißt **Low-Doubleword**, das Wort mit Bitposition 32 bis 63 **High-Doubleword**. 
Jedes Byte in einem Halbwort, Wort bzw. Doppelwort hat weiterhin seine eigene Adresse. Die Adresse des ersten Bytes entspricht der Adresse des Halbworts, Worts, Doppelworts bzw. Quadworts. Die Ausrichtung der jeweiligen Bytegruppe, die beim INTEL 80x86-Rechner nicht notwendig ist, beschleunigt den Arbeitsspeicherzugriff: Obwohl die Adressierung einzelner Bytes aus einem Programm heraus möglich, werden über den Bus immer Doppelworte übertragen, die an Doppelwortgrenze liegen. Das bedeutet, dass bei dem Zugriff auf eine nicht ausgerichtete Bytegruppe u.U. zwei Datentransfer erforderlich sind, während bei Ausrichtung nur ein Datentransfer ausreicht.		

Abbildung 2.1-1: Byte, Halbwort, Wort, Doppelwort, Quadwort

Auf Basis dieser Strukturierung in Bytes, Worte usw. werden **Grunddatentypen** eines IT-Systems definiert.

Das Problem der Byteanordnung (Big Endian versus Little Endian) logisch zusammen gehöriger Bytes, etwa in einem Wort, stellt sich spätestens dann, wenn Daten zwischen Maschinen unterschiedlicher Byteanordnung ausgetauscht werden sollen. So ist Programmcode, der für eine Big-Endian-Maschine erzeugt wurde, nicht kompatibel mit Programmcode für eine Little-Endian-Maschine, schon deshalb nicht, weil die in Worten abgelegten Adressen sich in der Byte-Reihenfolge unterscheiden. Entsprechendes gilt für die Daten. Enthalten etwa die aufeinanderfolgenden Bytes an den Adressen N und $N + 1$ die Folge von Sedezimalwerten $7A12_{16}$ und soll diese Folge als Festpunktzahl interpretiert werden, so hat sie in einer Big-Endian-Maschine den Wert $7 \cdot 16^3 + 10 \cdot 16^2 + 1 \cdot 16^1 + 2 \cdot 16^0 = 31.250$, in einer Little-Endian-Maschine den Wert $1 \cdot 16^3 + 2 \cdot 16^2 + 7 \cdot 16^1 + 10 \cdot 16^0 = 4.730$. Hersteller von Computer-Netzwerkarchitekturen haben sich übrigens darauf geeinigt, alle zu übertragenden Daten im Netz bitseriell in Big-Endian-Weise zu senden.

2.1.1 Datentyp Zahl

Der Datentyp **Zahl** (Festpunktzahl, Ordinalzahl, Gleitpunktzahl, gepackte und ungepackte Dezimalzahl) dient der Darstellung von Zahlen, die, bis auf ungepackte Dezimalzahlen, in arithmetischen Operationen verwendet werden können. Dabei können Festpunkt- und Dezimalzahlen ganze Zahlen (positive und negative Zahlen) darstellen, Ordinalzahlen nur positive ganze Zahlen; die erlaubten Operationen sind i.a. Addition und Subtraktion, Multiplikation, ganzzahlige Division mit Rest und mod-Bildung. Häufig gibt es für diese Datentypen Umwandlungsoperationen in den Datentyp Gleitpunktzahl. Gleitpunktzahlen werden zur Approximation gebrochener und reeller Zahlen verwendet. Erlaubte Operationen sind neben den üblichen arithmetischen Operationen Umwandlungen in Festpunktzahlen unter Weglassung des gebrochenen Anteils.

Ein gemäß dem Datentyp **Festpunktzahl** oder **Ordinalzahl** definiertes Datenobjekt belegt je nach Rechnertyp und zusätzlicher Festlegung ein Byte, Halbwort, Wort, Doppelwort oder Quadwort (z.B. IBM /390: Halbwort, Wort; INTEL 80x86: Byte, Wort, Doppelwort, Quadwort). Eine Festpunktzahl stellt eine ganze Zahl dar, eine Ordinalzahl eine nichtnegative ganze Zahl. Negative Zahlen in Festpunktdarstellung werden als 2er-Komplemente positiver Zahlen behandelt. Die möglichen Wertebereiche von Festpunkt- bzw. Ordinalzahlen zeigt folgende Tabelle:

intern verwendete Stellenzahl (Bits)	n	$n = 8$	$n = 16$	$n = 32$	$n = 64$
Festpunktzahl	$-2^{n-1}, \ldots, +2^{n-1} - 1$	–128, ..., 127	–32.768, ..., 32.767	–2.147.483.648, ..., 2.147.483.647	$-2^{63}, \ldots, 2^{63} - 1$; $2^{63} > 9{,}22 \cdot 10^{18}$
Ordinalzahl	$0, \ldots, 2^n - 1$	0, ..., 255	0, ..., 65.535	0, ..., 4.294.947.295	

Der Datentyp **Gleitpunktzahl** zur Approximation einer reellen Zahl durch eine rationale Zahl mit endlichem gebrochenem Anteil stellt eine Zahl x in der Form

$$x = \pm[a_0, a_{-1}a_{-2}a_{-3} \ldots a_{-k}]_B \cdot B^s \quad \text{mit} \quad a_i \in \{0, 1, \ldots, B-1\}$$

dar. Hierbei bezeichnet ± das **Vorzeichen** der Zahl x, $a_0 a_{-1} a_{-2} a_{-3} \ldots a_{-k}$ die **Mantisse** (in der INTEL 80x86-Architektur **Signifikand** genannt), B die **Basis** der Zahlendarstellung und s den **Exponenten** der Zahl. Durch geeignete **Normierung (Normalisierung)**, etwa durch die Festlegung

$0 \le a_0, a_{-1}a_{-2}a_{-3} \ldots a_{-k} < 1$ (wie bei der IBM /390-Architektur) bzw.

$1 \le a_0, a_{-1}a_{-2}a_{-3} \ldots a_{-k} < 2$ (wie bei der INTEL 80x86-Architektur)

ist diese Darstellung für $x \neq 0$ eindeutig.

Der Datentyp Gleitpunktzahl in der IBM /390-Architektur unterscheidet sich in einigen Details wesentlich vom Datentyp Gleitpunktzahl der INTEL 80x86-Architektur.

A. Datentyp Gleitpunktzahl in der IBM /390-Architektur

Hier lautet die Basis $B = 16$.

Ein Datenobjekt vom Datentyp Gleitpunktzahl wird in einem Wort, einem Doppelwort oder zwei aufeinander folgenden Doppelworten abgelegt (siehe auch Kapitel 2.1). Man spricht dann von einer **kurzen Gleitpunktzahl** (**Short Real**), einer **langen Gleitpunktzahl** (**Long Real**) bzw. einer **Gleitpunktzahl im erweiterten Format** (**Extended Real**).

Damit der Exponent eindeutig bestimmt ist, wird eine **Normalisierung** durchgeführt: Dies geschieht durch Verschieben des „sedezimalen Kommas“ in der Mantisse und Anpassung des Exponenten, so dass $a_0 = 0$ und $a_{-1} \neq 0$ gilt. Durch die Normalisierung wird eine höhere Genauigkeit bei der Zahlendarstellung erreicht; außerdem erwarten manche Maschineninstruktionen normalisierte Operanden, um normalisierte Ergebnisse zu erzeugen. Die **Mantisse** $a_0 a_{-1} a_{-2} a_{-3} \ldots a_{-k}$ (mit $a_0 = 0$ und $a_{-1} \neq 0$) wird also als

$$\sum_{i=1}^{k} a_{-i} \cdot 16^{-i}$$

interpretiert.

Für die Mantisse werden $k = 6$, $k = 14$ bzw. $k = 28$ Sedezimalstellen vorgesehen, wobei $a_0 = 0$ nicht abgespeichert wird. Je mehr Stellen sie enthält, umso genauer ist die Approximation der dargestellten Zahl an die darzustellende Zahl.

Das **Vorzeichen** wird durch ein Bit repräsentiert, und zwar bedeutet 0_2 das positive Vorzeichen und 1_2 das negative Vorzeichen. Eine negative Zahl wird nicht im Zweierkomplement dargestellt; die Mantisse ist also immer positiv.

Statt des tatsächlichen **Exponenten** s einer Zahl wird deren **Charakteristik** $c = s + b$ (ohne Übertrag) gespeichert. Die Zahl b heißt **Bias**. Die Funktion des Bias besteht darin, statt eines möglichen positiven oder negativen Exponenten immer eine nichtnegative Charakteristik (vorzeichenlos) abzuspeichern, so dass Gleitpunktzahlen mit gleichem Format und Vorzeichen wie vorzeichenlose Festpunktzahlen verglichen werden können.

Zu beachten ist, dass die Anzahl der Bits für die Charakteristik im wesentlichen die Größe des darstellbaren Wertebereichs festlegt. In der IBM /390-Architektur werden für den Exponenten bzw. die Charakteristik bei allen drei Gleitpunktdatenformaten 7 Bits vorgesehen. Der Exponent als Festpunktzahl kann also die Werte -64_{10} bis $+63_{10}$ annehmen. Für alle möglichen Gleitpunktdatenformate in dieser Architektur lautet der Bias $b = 64_{10}$. Die folgende Tabelle zeigt den Zusammenhang zwischen Exponent und Charakteristik.

$b = 64_{10}$ (7 Binärstellen für den Exponenten)			
Exponent s		Charakteristik $c = s + b$	
dezimal	binär	binär	sedezimal
– 64	1000000	0000000	00
– 63	1000001	0000001	01
...	...	...	...
– 1	1111111	0111111	3F
0	0000000	1000000	40
1	0000001	1000001	41
2	0000010	1000010	42
...	...	...	...
62	0111110	1111110	7E
63	0111111	1111111	7F

Vereinbarungsgemäß hat der Wert $x = 0$ das Vorzeichen +, die Mantisse 00 ... 00 und die Charakteristik 0, so dass der Wert $x = 0$ in Festpunkt- und Gleitkommadarstellung übereinstimmt.

Abbildung 2.1.1-1 fasst die drei Gleitpunktdatenformate in der IBM /390-Architektur zusammen.

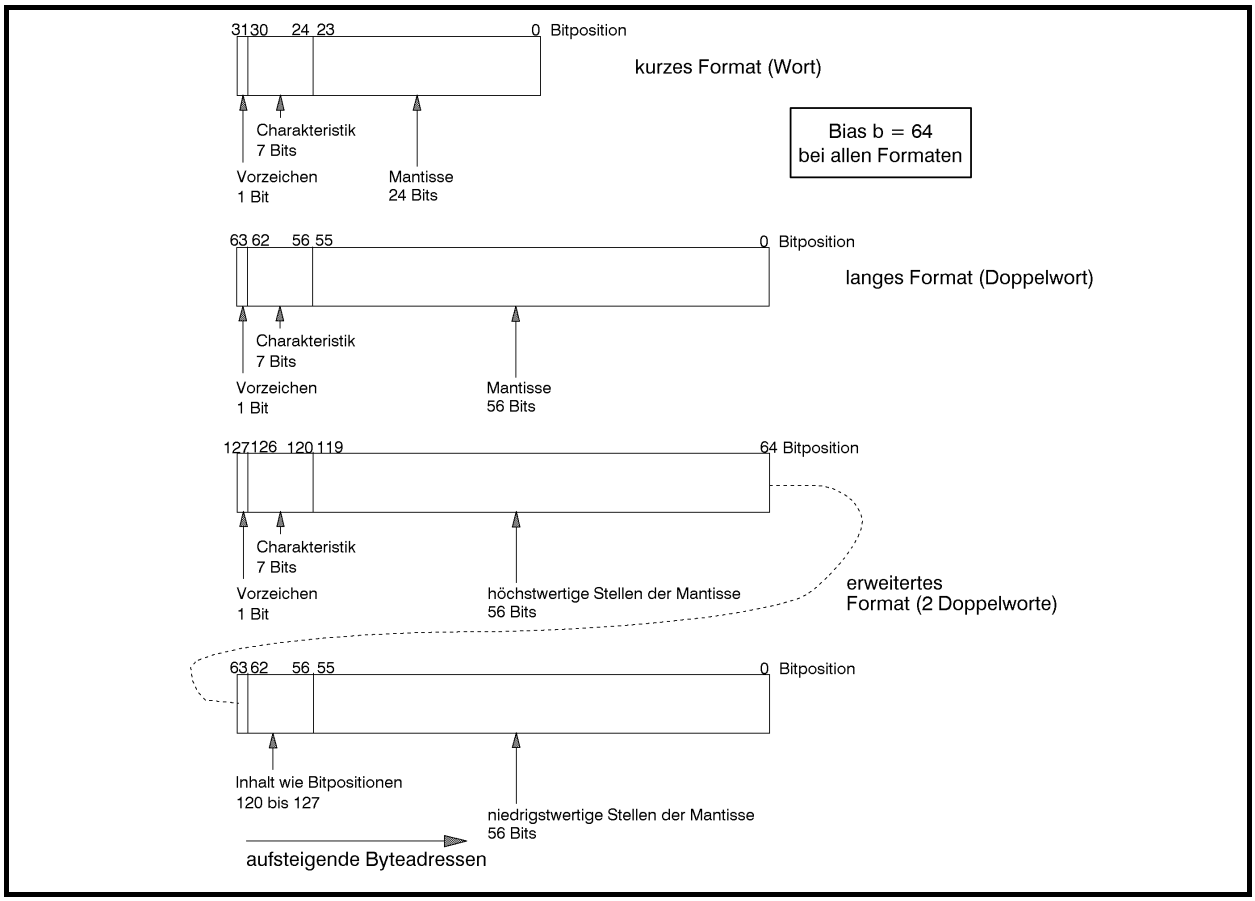


Abbildung 2.1.1-1: Gleitpunktdatenformate in der IBM /390-Architektur

Einige Zahlenbeispiele sollen die rechnerinterne Darstellung von Gleitpunktzahlen (hier für das kurze Format mit 6 Sedezimalziffern in der Mantisse) erläutern:

1. $46{,}415_{10} = 2E{,}6A3D70A3D70...{}_{16} = 2E{,}6\overline{A3D70}...{}_{16}$ (exakter Wert). Diese Zahl wird durch $0{,}2E6A3D_{16} \cdot 16^2 = 46{,}41499328613_{10}$ angenähert:
   Vorzeichen: +;
   Exponent: 2; Charakteristik: 1000010_2;
   Mantisse: 2E 6A $3D_{16}$ = 0010 1110 0110 1010 0011 1101_2;
   interne Darstellung: 0100 0010 0010 1110 0110 1010 0011 1101_2 = 42 2E 6A $3D_{16}$

2. $-0{,}03125_{10} = -0{,}8_{16} \cdot 16^{-1}$:
   Vorzeichen: –;
   Exponent: – 1; Charakteristik: 0111111_2;
   Mantisse: 80 00 00_{16} = 1000 0000 0000 0000 0000 0000_2;
   interne Darstellung: 1011 1111 1000 0000 0000 0000 0000 0000_2 = BF 80 00 00_{16}

3. $-213{,}04_{10} = -D5{,}0A3D70A3D7...{}_{16} = -D5{,}\overline{0A3D7}...{}_{16}$ (exakter Wert). Diese Zahl wird durch $-0{,}D50A3D_{16} \cdot 16^2 = -213{,}0399932861_{10}$ angenähert:
   Vorzeichen: –;
   Exponent: 2; Charakteristik: 1000010_2;

Mantisse: D5 0A $3D_{16}$ = 1101 0101 0000 1010 0011 1101_2;
interne Darstellung: 1100 0010 1101 0101 0000 1010 0011 1101_2 = C2 D5 0A $3D_{16}$.

Die im Gleitpunktdatenformat „kurzes Format“ darstellbare *größte Zahl* ist gleich
7F FF FF FF_{16} (Vorzeichen: +; Charakteristik: $1111111_2 = 127_{10}$, entsprechend dem Exponenten 63_{10}); diese Zahl hat den Wert
$0{,}FFFFFF_{16} \cdot 16^{63} = \left(1-16^{-6}\right)\cdot 16^{63} \approx 7{,}237005145973_{10} \cdot 10^{75}$.

Die im Gleitpunktdatenformat „kurzes Format“ darstellbare *kleinste Zahl* ist gleich
FF FF FF FF_{16} (Vorzeichen: –; Charakteristik: $1111111_2 = 127_{10}$ entsprechend dem Exponenten 63_{10}); diese Zahl hat den Wert
$-\,0{,}FFFFFF_{16} \cdot 16^{63} = -\left(1-16^{-6}\right)\cdot 16^{63} \approx -7{,}237005145973_{10} \cdot 10^{75}$.

Die im Gleitpunktdatenformat „kurzes Format“ darstellbare *kleinste positive Zahl größer als Null* hat (bedingt durch die Normalisierung) die interne Darstellung 00 10 00 00_{16} (Vorzeichen: +; Charakteristik = 0000000_2, entsprechend dem Exponenten -64_{10}); diese Zahl hat den Wert $+0{,}1_{16} \cdot 16^{-64_{10}} = 16^{-65_{10}} \approx 5{,}397605346934_{10} \cdot 10^{-79}$.

Für den **absoluten Wertebereich** W_{kG} der Zahlen im Gleitpunktdatenformat „kurzes Format“ gilt hier also (alle Angaben als Dezimalwerte):
$16^{-65} \le W_{kG} \le \left(1-16^{-6}\right)\cdot 16^{63}$,
$16^{-65} \approx 5{,}397605346934 \cdot 10^{-79}$ und $\left(1-16^{-6}\right)\cdot 16^{63} \approx 7{,}237005145973 \cdot 10^{75}$.
Das bedeutet an der unteren Grenze des Wertebereichs, dass die nächstkleinere (gültige) Zahl die Zahl 0 ist. Es wird natürlich jeweils vorausgesetzt, dass die Gleitpunktzahl normalisiert ist[2].

Für den absoluten Wertebereich W_{lG} der Zahlen im Gleitpunktdatenformat „langes Format“ bzw. für den absoluten Wertebereich W_{eG} der Zahlen im Gleitpunktdatenformat „erweitertes Format“ gelten die Werte (alle Angaben als Dezimalwerte):
$16^{-65} \le W_{lG} \le \left(1-16^{-14}\right)\cdot 16^{63} \approx 7{,}237005577331 \cdot 10^{75}$
bzw.
$16^{-65} \le W_{eG} \le \left(1-16^{-14}\right)\cdot 16^{63} \approx 7{,}237005577331 \cdot 10^{75}$,
also nahezu die gleichen Werte. Die Unterschiede liegen nicht im Wertebereich, sondern in der **Genauigkeit**, mit der man mit Hilfe eines Datenobjekts vom Typ Gleitpunktzahl eine reelle Zahl approximiert. Diese Genauigkeit wird durch den kleinstmöglichen Abstand Δ_{min}

---

2 Lässt man diese Forderung fallen, dann ergibt sich für die kleinste positive Gleitpunktzahl im kurzen Format die Darstellung 00 00 00 01_{16} (Vorzeichen: +; Charakteristik: 0000000_2 entsprechend dem Exponenten -64^{10}); diese Zahl hat den Wert $+0{,}000001_{16} \cdot 16^{-64} = 16^{-70} \approx 5{,}14755759_{10} \cdot 10^{-85}$.

zwischen zwei Zahlen im jeweiligen Gleitpunktdatenformat bestimmt; je dichter zwei Zahlen zusammenliegen, d.h. je kleiner dieser Abstand ist, umso genauer ist die Approximation. Dieser kleinste Abstand beträgt

beim kurzen Format: $0{,}000001_{16} \cdot 16^{-64_{10}} = 16^{-70_{10}} \approx 5{,}147557589468 \cdot 10^{-85}$,

beim langen Format: $0{,}00000000000001_{16} \cdot 16^{-64_{10}} = 16^{-78_{10}} \approx 1{,}198509146801 \cdot 10^{-94}$,

beim erweiterten Format: $0{,}\underbrace{000\ldots0001_{16}}_{28\,\text{Sedezimalziffern}} \cdot 16^{-64_{10}} = 16^{-92_{10}} \approx 1{,}663265562503 \cdot 10^{-111}$

B. Datentyp Gleitpunktzahl in der INTEL 80x86-Architektur

Hier lautet die Basis $B = 2$.

In der INTEL 80x86-Architektur, die sich in der Gleitpunktdatenformat am IEEE-Standard 754-1985 orientiert, liegt eine Gleitpunktzahl im **kurzen Format** (**Short Real**) in einem Doppelwort (4 Bytes), eine Gleitpunktzahl im **langen Format** (**Long Real**) in einem Quadwort (8 Bytes) und eine Gleitpunktzahl im **erweiterten Format** (auch **Temporary Real** genannt) in einem Feld von 10 zusammenhängenden Bytes. Die eingebaute Gleitkomma-Prozessoreinheit arbeitet mit diesem Format.

Die **Mantisse (Signifikand)** $a_0 a_{-1} a_{-2} a_{-3} \ldots a_{-k}$ als eine Ziffernfolge von Binärziffern wird so normalisiert, dass $a_0 = 1_2$ gilt, d.h. die Mantisse wird als

$$1 + \sum_{i=1}^{k} a_{-i} \cdot 2^{-i}$$

interpretiert. Das Bit $a_0 = 1_2$ wird im kurzen und langen Format nicht mit gespeichert.

Ein Bit repräsentiert das **Vorzeichen**, und zwar bedeutet 0_2 wieder das positive Vorzeichen und 1_2 das negative Vorzeichen. Statt des tatsächlichen Exponenten s einer Zahl wird deren Charakteristik $c = s + b$ (ohne Übertrag), hier als **Biased Exponent** bezeichnet, gespeichert, Die Größe des **Bias** b hängt vom jeweiligen Datentyp ab. Maximale und minimale Exponenten, sowie den jeweiligen Bias kann man der folgenden Tabelle entnehmen:

	Short Real	Long Real	Temporary Real
Breite insgesamt (Bits)	32	64	80
Breite der Mantisse (Bits)	23 + 1 (*)	52 + 1 (*)	64
Breite des biased Exponenten (Bits)	8	11	15
maximaler Exponent	127	1.023	16.383
minimaler Exponent	– 126	– 1.022	– 16.382
Bias	127	1.023	16.383
(*) Die führende 1_2 wird nicht mitgespeichert.			

Abbildung 2.1.1-2 fasst die drei Gleitpunktdatenformate in der INTEL 80x86-Architektur zusammen.

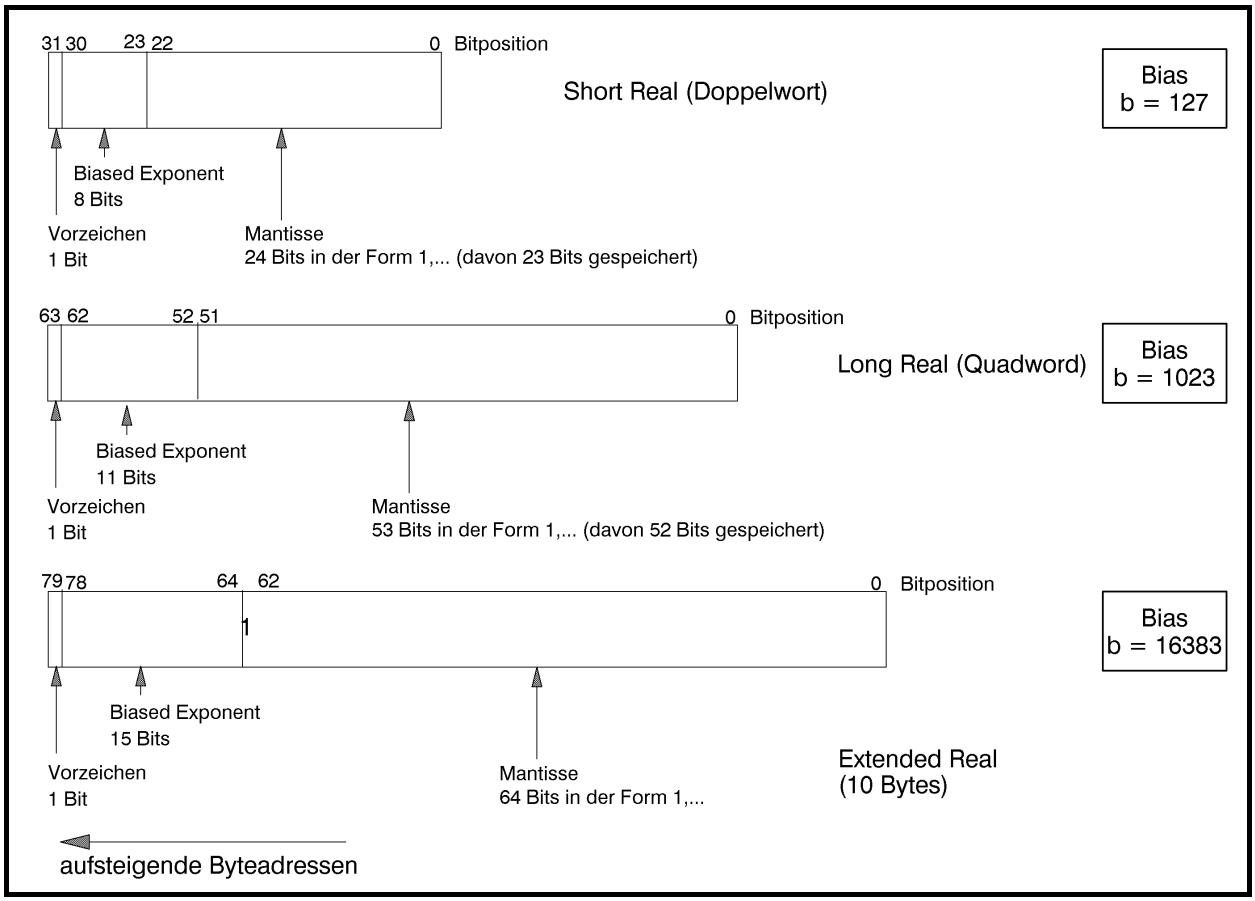


Abbildung 2.1.1-2: Gleitpunktdatenformate in der INTEL 80x86-Architektur

Die unter Punkt A. aufgeführten Zahlenbeispiele sollen auf die Gleitpunktdatenformate der INTEL 80x86-Architektur übertragen werden (auch hier nur für das kurze Format mit 32 Bits):

1. $46{,}415_{10} = 2E{,}6A3D70A3D70...{}_{16} = 2E{,}6\overline{A3D70}...{}_{16}$ (exakter Wert). Diese Zahl wird zunächst in Binärdarstellung notiert und normalisiert:

   $$2E{,}6\overline{A3D70}...{}_{16} = 00101110{,}0110\overline{1010001111010111000 0}...{}_2$$
   $$= 1{,}01110011010100011110101110000...{}_2 \cdot 2^{5_{10}}$$

   Für die Mantisse werden 23 Bits (beginnend nach dem Komma) genommen, so dass die Zahl $46{,}415_{10}$ durch den Wert $1{,}01110011010100011110101 \cdot 2^{5_{10}} = 46{,}41499710083_{10}$ approximiert wird.
   Vorzeichen: +;
   Exponent: 5_{10}; biased Exponent: 10000100_2;
   Mantisse: $01110011010100011110101_2$;
   interne Darstellung: $01000010001110011010100011110101_2 = 42\ 39\ A8\ F5_{16}$;

offensichtlich unterscheidet sich diese Darstellung wesentlich von der internen Darstellung bei der IBM /390-Architektur (dort lautet die interne Darstellung 42 2E 6A $3D_{16}$).

2. $-0{,}03125_{10} = -0{,}08_{16} = -0{,}00001000_2 = -1{,}000\ldots_2 \cdot 2^{-5_{10}}$
   Vorzeichen: –;
   Exponent: – 5; biased Exponent: 01111010_2;
   Mantisse: $00000000000000000000000_2$;
   interne Darstellung: $10111101000000000000000000000000_2$ = BD 00 00 00_{16}

3. $-213{,}04_{10} = -D5{,}0A3D70A3D7\ldots_{16} = -D5{,}\overline{0A3D7}\ldots_{16}$ (exakter Wert). Diese Zahl wird zunächst wieder in Binärdarstellung notiert und normalisiert:
   $-D5{,}\overline{0A3D7}\ldots_{16} = -11010101{,}\overline{00001010001111010111}\ldots_2$
   $= -1{,}10101010\overline{00001010001111010111}\ldots_2 \cdot 2^{7_{10}}$
   Für die Mantisse werden 23 Bits (beginnend nach dem Komma) genommen, so dass die Zahl $-213{,}04_{10}$ durch den Wert
   $-1{,}10101010000101000111101_2 \cdot 2^{7_{10}} = -213{,}0399932861_{10}$ approximiert wird.
   Vorzeichen: –;
   Exponent: 7; biased Exponent: 10000110_2;
   Mantisse: $10101010000101000111101_2$;
   interne Darstellung: $11000011010101010000101000111101_2$ = C3 55 0A $3D_{16}$.

Die im Gleitpunktdatenformat „kurzes Format“ darstellbare *größte Zahl* ist jetzt gleich $01111111011111111111111111111111_2$ = 7F 7F FF FF_{16} (Vorzeichen: +; biased Exponent: $11111110_2 = 254_{10}$ entsprechend dem Exponenten 127_{10}); diese Zahl hat den Wert $+1{,}11111111111111111111111_2 \cdot 2^{127_{10}} = \left(2^{24}-1\right)\cdot 2^{104_{10}} \approx 3{,}402823466385_{10} \cdot 10^{38}$.

Die *kleinste Zahl* im kurzen Gleitpunktdatenformat ist FF 7F FF FF_{16} entsprechend $-3{,}402823466385_{10} \cdot 10^{38}$.

Die mit dem Datentyp kurze Gleitpunktzahl darstellbare *kleinste positive Zahl größer als Null* ist (bedingt durch die Normalisierung)
$00000000100000000000000000000000_2$ = 00 80 00 00_{16}
(Vorzeichen: +; biased Exponent: 00000001_2, entsprechend dem Exponenten -126_{10});
diese Zahl hat den Wert $+1{,}0_2 \cdot 2^{-126_{10}} = 1{,}175494350822_{10} \cdot 10^{-38}$.

Entsprechend lassen sich die absoluten Wertebereiche für die übrigen Datenformate der Gleitpunktzahlen errechnen (siehe Abbildung 2.1.1-3).

Die **Genauigkeit**, die den kleinstmöglichen Abstand Δ_{min} zwischen zwei Zahlen vom Gleitpunktdatenformat bestimmt, ist beim kurzen Format gleich

$$2^{-126_{10}} \cdot 0,\underbrace{0\ldots01}_{23\text{ Binärziffern}} = 2^{-149_{10}} \approx 1{,}401298464325_{10} \cdot 10^{-45}.$$

Auf gleiche Weise ergibt sich Δ_{min} bei den anderen Gleitpunktdatenformaten:

beim langen Format: $2^{-1074_{10}} \approx 4{,}9407_{10} \cdot 10^{-324}$,

beim erweiterten Format: $2^{-16445_{10}} \approx 3{,}6452_{10} \cdot 10^{-4951}$.

Abbildung 2.1.1-3 fasst die Ergebnisse für Gleitpunktdatenformate zusammen.

absoluter Wertebereich W (normalisiert[3])		
	IBM /390	INTEL 80x86
kurzes Format	$5{,}3976_{10} \cdot 10^{-79} < W < 7{,}2371_{10} \cdot 10^{75}$	$1{,}1755_{10} \cdot 10^{-38} < W < 3{,}4029_{10} \cdot 10^{38}$
langes Format	$5{,}3976_{10} \cdot 10^{-79} < W < 7{,}2371_{10} \cdot 10^{75}$	$2{,}2250_{10} \cdot 10^{-308} < W < 1{,}7977_{10} \cdot 10^{308}$
erw. Format	$5{,}3976_{10} \cdot 10^{-79} < W < 7{,}2371_{10} \cdot 10^{75}$	$3{,}3621_{10} \cdot 10^{-4932} < W < 1{,}1898_{10} \cdot 10^{4932}$
Genauigkeit (kleinstmöglicher Abstand zwischen zwei Zahlen)		
	IBM /390	INTEL 80x86
kurzes Format	$\approx 5{,}1476_{10} \cdot 10^{-85}$	$\approx 1{,}4013_{10} \cdot 10^{-45}$
langes Format	$\approx 1{,}1985_{10} \cdot 10^{-94}$	$\approx 4{,}9407_{10} \cdot 10^{-324}$
erw. Format	$\approx 4{,}2580_{10} \cdot 10^{-109}$	$\approx 3{,}6452_{10} \cdot 10^{-4951}$
spezielle Werte (hier nur für Short Real)		
	IBM /390	INTEL 80x86
0	$00\ 00\ 00\ 00_{16}$	$+0 = 00\ 00\ 00\ 00_{16}$ $-0 = 80\ 00\ 00\ 00_{16}$
$+\infty$		$7F\ 80\ 00\ 00_{16}$
$-\infty$		$FF\ 80\ 00\ 00_{16}$
NaN		$7F\ 80\ 00\ 01_{16}, \ldots, 7F\ FF\ FF\ FF_{16}$, $FF\ 80\ 00\ 01_{16}, \ldots, FF\ FF\ FF\ FF_{16}$
unbestimmt		$FF\ 00\ 00\ 00_{16}$
	→ aufsteigende Adressen	← aufsteigende Adressen

Abbildung 2.1.1-3: Wertebereiche der Gleitpunktdatenformate in der INTEL 80x86-Architektur

3 An der Obergrenze ist im Faktor die letzte Dezimalstelle nach dem Komma des Faktors aufgerundet, an der Untergrenze abgerundet.

Einige Bitmuster stellen die speziellen Werte der **NaN**s (**Not a Number**) dar: Der biased Exponent enthält nur 1_2, und die Mantisse kann jeden Wert außer $10...0_2$ annehmen. Es gibt zwei Typen von NaNs: signalisierende NaNs (höchstwertiges Mantissenbit = 1_2) und stille NaNs (höchstwertiges Mantissenbit = 0_2). Eine **signalisierende NaN** wird von der CPU nie erzeugt und kann daher von einem Programm verwendet werden, um besondere Ausnahmesituationen zu behandeln, die in den Mantissenbits angezeigt werden können. **Stille NaNs** werden von der CPU bei den verschiedenen Ausnahmesituationen „ungültige Operation“ erzeugt (die stille NaN „undefiniert“).

Ein weiterer Datentyp zur internen Darstellung von Zahlen ist der Datentyp **Dezimalzahl** mit den beiden Untertypen gepackte Dezimalzahl bzw. ungepackte Dezimalzahl. Beide Datentypen dienen der Darstellung ganzer Zahlen im Dezimalformat, d.h. mit den Dezimalziffern 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Die IBM /390-Architektur besitzt eine ausgeprägte Unterstützung der Dezimalarithmetik durch entsprechende Maschineninstruktionen im Gegensatz zur INTEL 80x86-Architektur.

Bei einer Zahl vom Datentyp **gepackte Dezimalzahl** wird jede Dezimalziffer (die ja auch als Sedezimalziffer angesehen werden kann) so abgelegt, dass jeweils zwei Dezimalziffern der Zahl ein Byte belegen.

A. Datentyp gepackte Dezimalzahl in der IBM /390-Architektur

Das Vorzeichen der Zahl wird hier im äußerst rechten Byte (höchste Adresse in der internen Darstellung der Zahl) in der rechten Ziffernstelle, d.h. der kleinsten Ziffernposition, kodiert: dazu wird für das positive Vorzeichen die Sedezimalziffer C_{16} angehängt, für das negative Vorzeichen die Sedezimalziffer D_{16}. Die Sedezimalziffern A_{16} und E_{16} werden ebenfalls als Kodierungen des positiven Vorzeichens, die Sedezimalziffer B_{16} als Kodierung des negativen Vorzeichens interpretiert.

Die Zahl wird eventuell links mit einer führenden 0 aufgefüllt, so dass immer eine volle Anzahl von Bytes entsteht. Zur Darstellung einer *n*-stelligen Dezimalzahl als gepackte Dezimalzahl werden $\lceil (n+1)/2 \rceil$ viele Bytes benötigt. Es besteht die Einschränkung, dass diese Stellenzahl maximal 16 sein darf. Eine Ausrichtung auf Halbwort-, Wort- oder Doppelwortgrenze gibt es nicht.

B. Datentyp gepackte Dezimalzahl in der INTEL 80x86-Architektur

Die INTEL 80x86-Architektur unterstützt keine Vorzeichendarstellung (außer in einem speziellen Untertyp der Gleitkomma-Verarbeitungseinheit).

Bei einer Zahl vom Datentyp **ungepackte Dezimalzahl** belegt jede Dezimalziffer (die wieder auch als Sedezimalziffer angesehen werden kann) ein Byte, wobei die Dezimalziffer in den Bitpositionen 0 bis 3 steht; die Bitpositionen 4 bis 7 enthalten die Sedezimalziffer F_{16} (IBM /390) bzw. einen beliebigen Wert (INTEL 80x86). Das Vorzeichen wird bei IBM /390 im Byte der niedrigstwertigen Stelle kodiert, und zwar wird anstelle der Sedezimalziffer F_{16} bei einer positiven Zahl die Sedezimalziffer C_{16} genommen, bei einer negativen Zahl anstelle der Sedezimalziffer F_{16} die Sedezimalziffer D_{16}. Die INTEL 80x86-Architektur unterstützt kein Vorzeichen.

Eine *n*-stellige Zahl belegt *n* Bytes, so dass Zahlen beliebiger Größe darstellbar sind.

2.1.2 Datentyp Zeichenkette

Der Datentyp **Zeichenkette (String)** beschreibt eine Aneinanderreihung von Bytes, die einen beliebigen Inhalt aufnehmen können. Meist enthält eine Zeichenkette die rechnerinterne Darstellung druckbarer Zeichen, wobei unterschiedliche Zeichenkodierungen verwendet werden. Die IBM /390-Architektur setzt standardmäßig den EBCDI-Code (extended binary coded decimal interchange code) zur Darstellung von Zeichen ein, in der INTEL 80x86-Architektur wird der ASCII-Code (American standard code for information interchange) verwendet. Beide Zeichensätze kodieren Ziffern, Buchstaben, Sonderzeichen und Steuerzeichen, die z.B. in der Textverarbeitung und dem Datentransfer vorkommen. Beide Codes definieren sowohl **internationale** als auch **nationale Versionen**.

Der **EBCDI-Code** (extended binary coded decimal interchange code) benutzt die 8 Bits eines Bytes, wobei einige Bitkombinationen im Code nicht definiert sind. Ein Zeichen besteht dabei aus einer linken Sedezimalziffer, dem **Zonenteil**, und einer als **Zifferntеil** bezeichneten rechten Sedezimalziffer. Die folgende Tabelle zeigt den EBCDI-Code in der internationalen und der deutschen Version. Ein zu kodierendes Zeichen dabei wieder im Schnittpunkt einer Spalte und einer Zeile. Einige Bytewerte sind in der internationalen bzw. deutschen Version mit doppelten Bedeutungen belegt (z.B. bedeutet der Bytewert $7C_{16}$ entweder das Zeichen @ oder das Zeichen §); welche Bedeutung im Einzelfall genommen wird, ist maschinenabhängig. Die durch einen Text von 2 oder 3 Buchstaben beschriebenen „Zeichen“ stellen definierte Steuerzeichen dar, denen kein druckbares Zeichen entspricht.

EBCDI-Code

rechtes Halbbyte (Ziffernteil)

linkes Halbbyte (Zonenteil)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL				PF	HT	LC	DEL								
1					RES	NL	BS	IL								
2					BYP	LF	EOB	PRE			SM					
3					PN	RS	UC	EOT								
4	SP										¢	.	<	(	+	| ö
5	&										! ÷	$¤	*	)	;	¬
6	-	/									^	,	%	_	>	?
7											:	# £	@ §	'	=	"
8		a	b	c	d	e	F	g	h	i						
9		j	k	l	m	n	O	p	q	r						
A			s	t	u	v	W	x	y	z						
B												[Ä	\ Ö	] Ü		
C		A	B	C	D	E	F	G	H	I						
D		J	K	L	M	N	O	P	Q	R						
E			S	T	U	V	W	X	Y	Z						
F	0	1	2	3	4	5	6	7	8	9		{ ä		} ü		~ ß

Bedeutung der Steuerzeichen:

NUL	(Füllzeichen)	EOB	Blockende
PF	Stanzer aus	PRE	(Bedeutungsänderung der beiden Folgezeichen)
HAT	Horizontaltabulator		
LC	Kleinbuchstaben	PN	Stanzer ein
DEL	Löschen	RS	Leser Stop
RES	Sonderfolgenende	UC	Großbuchstaben
NL	Zeilenvorschub mit Wagenrücklauf	EOT	Ende der Übertragung
BS	Rückwärtsschritt	SM	Betriebsartenänderung
IL	Leerlauf	SP	Leerzeichen
LF	Zeilenvorschub		

Abbildung 2.1.2-1: EBCDI-Code

Der **ASCII-Code** (American standard code for information interchange) ist ein 7-Bit-Code, definiert also 128 Zeichen. Das im Byteformat freie achte Bit kann als Prüfbit, für programmtechnische Zwecke oder zur Erweiterung auf einen 8-Bit-Code, z.B. für verschiedene Drucker-Zeichensätze oder länder- und sprachspezifische Zeichensätze, verwendet werden. Die folgende Tabelle beinhaltet den ASCII-Code in seiner internationalen Ausprägung (die ersten 128 Zeichen), erweitert um Zeichen, die in westlichen Sprachen benötigt werden (ISO-8859-1, *Latin1*). Ein zu kodierendes Zeichen wieder steht im Schnittpunkt einer Zeile und einer Spalte. Die linke Sedezimalziffer ist in der Zeilenbeschriftung abzulesen, die rechte Sedezimalziffer in der Spaltenbeschriftung. Die durch einen Text von 2 oder 3 Buchstaben beschriebenen „Zeichen“ stellen definierte Steuerzeichen dar, denen kein druckbares Zeichen entspricht.

ASCII-Code

rechtes Halbbyte (Ziffernteil)

linkes Halbbyte (Zonenteil)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	QS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	x	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	¯	¯	¡	«	»
B						Á	Â	À	©					¢	¥	
C							ã	Ã								¤
D	_	_	Ê	Ë	È	¡	Í	Î	Ï					I	Ì	
E	Ó	ß	Ô	Ò	õ	Õ	µ	_	_	Ú	Û	Ù	y	Y	¯	´
F	-	±		_	¶	§	÷	¸	°	¨	·	¹	³	²		

Bedeutung der Steuerzeichen:

NUL	(Füllzeichen, Null)	DCx, x=1..4	Gerätesteuerung (Device Control)
SOH	Anfang des Kopfes (Start of Heading)		
STX	Anfang des Textes (Start of Text)	NAK	Negative Rückmeldung (Negative Acknowledge)
ETX	Ende des Textes (End of Text)		
EOT	Ende der Übertragung (End of Transmission)	SYN	Synchronisation (Synchronous Idle)
		ETB	Ende des Datenübertragungsblocks (End of Transmission Block)
ENQ	Stationsaufforderung (Enquiry)		
ACK	Positive Rückmeldung (Acknowledge)	CAN	Ungültig (Cancel)
BEL	Klingel (Bell)	EM	Ende der Aufzeichnung (End of Medium)
BS	Rückwärtsschritt (Backspace)		
HAT	Horizontal-Tabulator (Horizontal Tabulation	SUB	Substitution (Substitute Character)
		ESC	Umschaltung (Escape)
LF	Zeilenvorschub (Line Feed)	FS	Hauptgruppen-Trennung (File Separator)
VT	Vertikal-Tabulator (Vertical Tabulation)		
FF	Formularvorschub (Form Feed)	GS	Gruppen-Trennung (Group Separator)
CR	Wagenrücklauf (Carriage Return)	RS	Untergruppen-Trennung (Record Separator)
SO	Dauerumschaltung (Shift-out)		
SI	Rückschaltung (Shift-in)	US	Teilgruppen-Trennung (Unit Separator)
DLE	Datenübertragungsumschaltung (Data Link Escape)	DEL	Löschen (Delete)

Abbildung 2.1.2-2: ASCII-Code

Während der EBCDI- und der ASCII-Code für die Darstellung eines Zeichens ein Byte, also 8 Bits verwenden, definiert der in neueren Betriebssystemen (z.B. WINDOWS-NT, -CE, -XP, -2000, Linux, Solaris) und in den Web-Datenformaten HTML und XML intern eingesetzte 16-Bit-**Unicode** einen Zeichensatz mit 16 Bits pro Zeichen, so dass 2^{16} = 65.536 verschiedene

Zeichen möglich und damit Zeichensätze asiatischer, arabischer und europäischer Sprachen repräsentierbar sind. Nicht alle definierten Zeichen sind eigenständige Schriftzeichen, sondern Bausteine, aus denen sich zusammen mit einem Buchstabe ein gültiges Zeichen ergibt.

Die Norm ISO/IEC 10646 definiert einen 32-Bit-Code, den **UCS-4-Octett-Code** (universal character set). Mit ihm sind 2^{32} = 4.294.967.296 verschiedene Zeichen möglich, so dass die Schriftzeichen aller lebenden und toten Sprachen mit diesem Code darstellbar sind.

2.1.3 Datentyp Binärwert

Neben den in druckbaren Zeichenketten vorkommenden Bitkombinationen gibt es Bitkombinationen in einem Byte, denen keine Bedeutung fest zugeordnet sind und die weder als Zeichen einer druckbaren Zeichenkette noch als Zahlen sinnvoll interpretiert werden können. Ein Datenobjekt vom Datentyp **Binärwert** kann als Wert eine beliebige Bitkombinationen annehmen und eine Anzahl von Bytes belegen, die durch die Definition des Datenobjekts festgelegt ist. Die Interpretation hängt von der Anwendung ab. Daher stellt der Datentyp Binärwert den allgemeinsten Datentyp dar.

Je nach Interpretation der vorkommenden Bitkombinationen spricht man auch von **Sedezimalkonstanten** bzw. **Binärkonstanten**.

2.1.4 Datentyp Adresse

Der Datentyp **Adresse** dient der Darstellung virtueller Adressen in einem Programm. Dieser Datentyp ist abhängig vom Rechnertyp bzw. vom verwendeten Speichermodell. Beispielsweise verwenden die Speichermodelle des INTEL 80x86-Rechners sehr unterschiedliche Arten der internen Adressdarstellung und –interpretation (Abbildung 2.1.4-1); dabei wird noch zwischen Near und Far Pointern unterschieden. Die IBM /390-Architektur wiederum verwendet in Maschinenbefehlen ein Adressformat, das sich aus zwei Teilen zusammensetzt: einer CPU-Registernummer und einem als Displacement bezeichneten Offset. Der Inhalt des angesprochenen Registers wird Adresse interpretiert, zu der der Offset hinzuaddiert wird und so eine lineare virtuelle Adresse liefert. Der Anwender ist dafür verantwortlich, dass das CPU-Register mit einem „sinnvollen“ Wert geladen wird. Diese Vorgehensweise bezieht sich natürlich auf die Programmierung auf Maschinenebene; auf der Programmierebene einer höheren Programmiersprache werden diese Details durch die Sprachkonstrukte der Programmiersprache verborgen. Auf weitere Details soll hier daher verzichtet werden.

INTEL 80x86		
	Near Pointer	Far Pointer

Real Mode
 	
offset = Adresse innerhalb eines Segments
 	
physikalische Adresse = Segmentbasisadresse + offset (20 Bits)
	Format:
	`[offset] 16 Bits`
	
	Die 16 höherwertigen Bits der Segmentbasisadresse werden zur Laufzeit aus dem Inhalt eines Segmentregisters der CPU (z.B. CS, DS, SS) genommen und um 4 binäre 0_2 rechts ergänzt (16 + 4 Bits)
		Format:
		`[segmentnummer] 16 Bits`
		`[offset] 16 Bits`
		Segmentbasisadresse = segmentnummer * 16 (20 Bits);
		durch den Ladevorgang sind zur Laufzeit alle Segmentnummern der EXE-Datei um die Startsegmentnummer des Programms erhöht worden

Protected Mode
 	
Offset = Adresse innerhalb eines Segments
 	
Physikalische Adresse = Segmentbasisadresse + offset (32 Bits)
	Format:
	`[offset] 32 Bits`
	
	Die Segmentbasisadresse steht in einem Deskriptorregister der CPU (zusammen mit weiteren Informationen)
		Format:
		`[segmentselektor] 16 Bits`
		`[offset] 32 Bits`
		Der Segmentselektor wird als Index in eine Deskriptortabelle interpretiert; der so indizierte Eintrag enthält (mit weiteren Informationen) die Segmentbasisadresse

Flaches Speichermodell physikalische Adresse = Ergebnis des auf die Linearadresse angewendeten Pagingverfahrens (32 Bits)	Format: `[linearadresse] 32 Bits`	

Abbildung 2.1.4-1: Datentyp Adresse (Beispiel: INTEL 80x86-Architektur)

2.1.5 Datentyp Instruktion

Der Datentyp **Instruktion** beschreibt die interne Darstellung einer Maschineninstruktion. Diese besitzen ein festes Format (meist mehrere Bytes, weiter unterschieden nach Instruktionstypen), in denen die jeweilige Maschineninstruktion und die beteiligten Operanden kodiert sind. Operanden werden über virtuelle Adressen, als Direktwerte, über Registerbezeichnungen der CPU usw. angesprochen. Meist enthält eine Maschineninstruktion abhängig vom Rechnertyp zwischen einer und drei Operandenadressen. Eine weiterführende Behandlung des Themas Instruktionen ist der Assemblerprogrammierung vorbehalten.

2.2 Wichtige Datentypen in einer höheren Programmiersprache

In diesem Kapitel werden die Definitionsmöglichkeiten für Datenobjekte am Beispiel der höheren Programmiersprache Pascal (als Pseudocode) exemplarisch an ausgewählten Datentypen näher betrachtet. Dabei werden diese Definitionsmöglichkeiten nur soweit behandelt, wie sie in den folgenden Kapiteln benötigt werden. Weitere Details, insbesondere, wenn Pascal als Implementierungssprache für konkrete Anwendungen eingesetzt werden soll, kann man den einschlägigen Manuals oder der umfangreichen Literatur zu Pascal entnehmen ([PAS], [D/K]).

Während sich Datendefinitionen in einigen höheren Programmiersprachen sehr an der internen Datendarstellung orientieren (z.B. bei COBOL denke man an die PICTURE- und USAGE-Klauseln und ihre Abbildung auf interne Datenobjekte, die direkt auf ASSEMBLER-Deklarationen der IBM /390-Architektur übertragbar sind, vgl. [HOF]), abstrahiert Pascal weitgehend von der zugrundeliegenden Rechnerarchitektur und gewährleistet damit eine gewisse Portabilität, zumindest auf Quellprogrammebene. Die Möglichkeiten der Datendefinition in einem Pascal-Programm orientieren sich wesentlich mehr an den Belangen der Anwendungen als an den Konzepten, die von einer Hardware-Architektur bereitgestellt werden. In der Praxis jedoch, insbesondere bei Problemen der systemnahen Programmierung, ist eine Hinwendung zu mehr rechnerorientierten Deklarationsmöglichkeiten eventuell erwünscht.

Wie bereits anfangs beschrieben besteht die übliche Art, ein Datenobjekt in einer höheren Programmiersprache zu deklarieren, in der Angabe des Typs des Objekts und damit implizit der zulässigen Operationen und Methoden, die auf das Datenobjekt dieses Typs anwendbar sind, und der Angabe des Bezeichners für das Datenobjekt, etwa in der Form:

```
TYPE data_typ = ...;

VAR  datenobjekt : data_typ;
```

Zunächst sollen die (in Pascal) vordefinierten Datentypen an Beispielen erläutert werden. Einen Ausschnitt aus den Syntaxdiagrammen der wichtigsten Typdeklarationen zeigt Abbildung 2.2-1. Offensichtlich kann ein definierter Typ mittels seines Typbezeichners für weitere Definitionen verwendet werden. Außerdem gibt es durch Schlüsselworte der Sprache vordefinierte Typbezeichner.

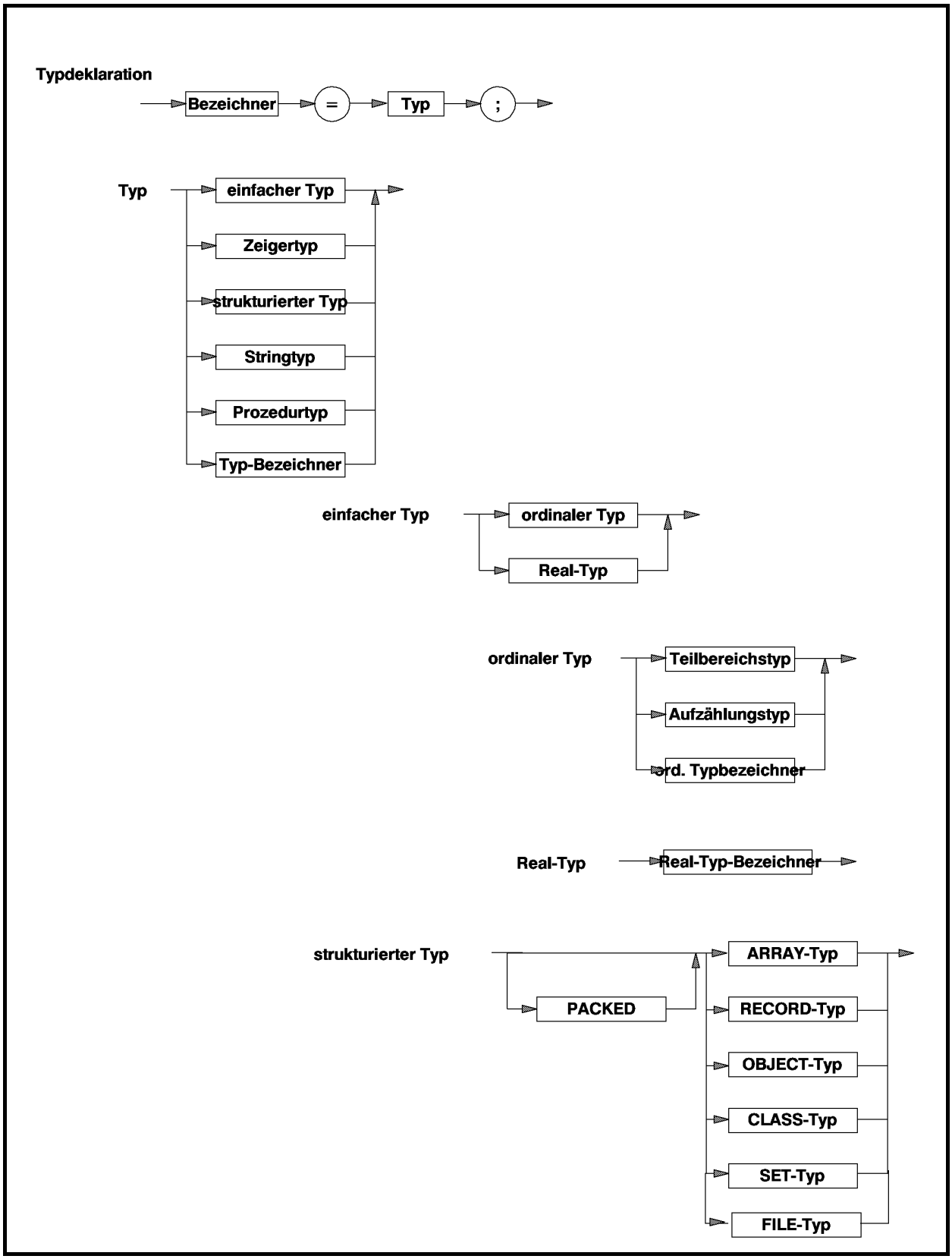


Abbildung 2.2-1: Datentypdeklarationen in Pascal (Ausschnitt)

Die Beschreibung in den folgenden Unterkapiteln orientiert sich an Abbildung 2-2. Dort werden Datentypen unterschieden nach Grunddatentypen, die in den meisten IT-Systemen vorhanden sind, abgeleiteten Datentypen, deren Deklarationsform von der jeweiligen Programmiersprache abhängt, und anwendungsorientierten Datentypen, wie sie häufig von Entwick-

lungsumgebungen bzw. Spracherweiterungen in Form von Bibliotheken bereitgestellt werden. Der Grunddatentyp `Pointer` und die abgeleiteten Datentypen `OBJECT` und `CLASS` werden dabei wegen ihrer besonderen Bedeutung in gesonderten Unterkapiteln behandelt.

2.2.1 Grunddatentypen

Zur Definition von Datenobjekten vom Datentyp **Festpunktzahl** (zur Darstellung ganzer Zahlen, wobei negative Zahlen als 2er-Komplemente positiver Zahlen behandelt werden) oder vom Datentyp **Ordinalzahl** (zur Darstellung vorzeichenloser nichtnegativer Zahlen) sind in Pascal einige Datentypen implizit vordefiniert, deren Bezeichner folgender Tabelle entnommen werden können:

Typ	internes Format (INTEL 80x86)	Wertebereich	interner Datentyp (INTEL 80x86)
`INTEGER`	Wort[4] Doppelwort[5]	–32.768, ..., 32.767 –2.147.483.648, ..., 2.147.483.647	Festpunktzahl, 16 Bits Festpunktzahl, 32 Bits
`ShortInt`	Byte	–128, ..., 128	Festpunktzahl, 8 Bits
`SmallInt`	Wort	–32.768, ..., 32.767	Festpunktzahl, 32 Bits
`LongInt`	Doppelwort	–2.147.483.648, ..., 2.147.483.647	Festpunktzahl, 32 Bits
`Int64`	Quadwort	-2^{63} , ..., $2^{63} - 1$; $2^{63} > 9{,}22 \cdot 10^{18}$	Festpunktzahl, 64 Bits
`Byte`	Byte	0, ..., 255	Ordinalzahl, 8 Bits
`Word`	Wort	0, ..., 65.535	Ordinalzahl, 16 Bits
`LongWord`	Doppelwort	0, ..., 4.294.967.295	Ordinalzahl, 32 Bits
`Cardinal`	Doppelwort	0, ..., 4.294.967.295	Ordinalzahl, 32 Bits

Zur Definition von Datenobjekten vom Datentyp **Gleitpunktzahl** (zur Approximation reeller Zahlen) werden reelle Datentypen bereitgestellt, die auf der internen Darstellung von Gleitpunktzahlen in der INTEL 80x86-Architektur aufbauen (vgl. Abbildung 2.1.1-2). Folgende Tabelle zeigt die unterschiedlichen reelen Datentypen:

4 Bei 16-Bit-Compiler
5 Bei 32-Bit-Compiler

Typ	internes Format (INTEL 80x86)	AbsoluterWertebereich, kleinstmöglicher Abstand zweier Zahlen	interner Datentyp (INTEL 80x86)
`REAL`	Quadwort	$2{,}2250_{10}\cdot 10^{-308} < W < 1{,}7977_{10}\cdot 10^{308}$, $\approx 4{,}9407_{10}\cdot 10^{-324}$	Long Real, 64 Bits
`Single`	Doppelwort	$1{,}1755_{10}\cdot 10^{-38} < W < 3{,}4029_{10}\cdot 10^{38}$, $\approx 1{,}4013_{10}\cdot 10^{-45}$	Short Real, 32 Bits
`Double`	Quadwort	$2{,}2250_{10}\cdot 10^{-308} < W < 1{,}7977_{10}\cdot 10^{308}$, $\approx 4{,}9407_{10}\cdot 10^{-324}$	Long Real, 64 Bits
`Extended`		$3{,}3621_{10}\cdot 10^{-4932} < W < 1{,}1898_{10}\cdot 10^{4932}$, $\approx 3{,}6452_{10}\cdot 10^{-4951}$	Extended Real, 80 Bits
`Real48`	Quadwort	$2{,}9387_{10}\cdot 10^{-39} < W < 1{,}7015_{10}\cdot 10^{38}$, $\approx 5{,}3455_{10}\cdot 10^{-51}$	---, 48 Bits
`Comp`, `Currency`	Ordinaltypen, die als reelle Datentypen klassifiziert werden, weil ihr Verhalten nicht Ordinaltypen entspricht (sie werden im folgenden nicht weiter betrachtet)		

Der Datentyp `Real48` ist aus Abwärtskompatibilitätsgründen zu älteren (Borland-) Pascalversionen eingeführt. Ihm entspricht kein internes Format der INTEL 80x86-Architektur. Daher laufen die entsprechenden Operationen langsamer ab als mit anderen Gleitpunktdatentypen. Intern belegt ein Datenobjekt vom Datentyp `Real48` 6 Bytes (48 Bits), und zwar 1 Bit für das Vorzeichen, 8 Bits für den biased Exponenten und 39 Bits für die Mantisse. Der Bias ist so gewählt, dass die Charakteristik c einer von 0 verschiedene `Real48`-Zahl die Bedingung $0 < c \leq 255$ erfüllt. Jede Charakteristik $c = 0$ führt zur Interpretation 0 des Werts für das Datenobjekt. Der Datentyp `Real48` kennt weder nicht-normalisierte Zahlen noch NaNs oder Unendlichkeiten: nicht-normalisierte Zahlen werden auf 0 gesetzt, NaNs und Unendlichkeiten erzeugen einen Überlauf.

Die in arithmetischen Ausdrücken mit Variablen vom Festpunktzahl-, Ordinalzahl- und Gleitpunktdatentyp zulässigen wichtigsten Operatoren werden in Abbildung 2.2.1-1 zusammengefasst. Dabei steht wie üblich ein binärer Operator zwischen zwei Operanden, ein unärer Operator vor einem einzelnen Operanden. Ein Integer-Datentyp bezeichnet dabei einen der oben aufgeführten Festpunkt- bzw. Ordinaldatentypen, ein Real-Datentyp einen der obigen Gleitpunktdatentypen. Zusätzlich werden die zu verwendeten Vergleichsoperatoren genannt.

Binäre Operatoren				
Operator	Operation	Operandentyp	Ergebnistyp	Bemerkung
+ bzw. − bzw. *	Addition bzw. Subtraktion bzw. Multiplikation	Integer Real	Integer Real	
/	Division	Integer Real	Real Real	Laufzeitfehler bei Division durch 0
DIV	ganzzahlige Division	Integer	Integer	i DIV j ist gleich $\lfloor i / j \rfloor$; Laufzeitfehler bei Division durch $j = 0$
MOD	Modulo	Integer	Integer	i MOD j ist gleich $i - (i \text{ DIV } j) \cdot j$
Unäre Operatoren				
+	Identität	Integer Real	Integer Real	
−	Negation	Integer Real	Integer Real	
Logische Operatoren				
NOT	bitweise Negation	Integer	Integer	NOT 0_2 = 1_2, NOT 1_2 = 0_2
AND	bitweises UND	Integer	Integer	0_2 AND 0_2 = 0_2, 0_2 AND 1_2 = 0_2, 1_2 AND 0_2 = 0_2, 1_2 AND 1_2 = 1_2
OR	bitweises ODER	Integer	Integer	0_2 OR 0_2 = 0_2, 0_2 OR 1_2 = 1_2, 1_2 OR 0_2 = 1_2, 1_2 OR 1_2 = 1_2
XOR	bitweises EXKLUSIVES ODER	Integer	Integer	0_2 XOR 0_2 = 0_2, 0_2 XOR 1_2 = 1_2, 1_2 XOR 0_2 = 1_2, 1_2 XOR 1_2 = 0_2
SHL	bitweise Linksverschiebung	Integer	Integer	i SHL j verschiebt den Wert von i um so viele Bitpositionen nach links, wie der Wert von j angibt, d.h. i SHL j ist gleich $i \cdot 2^j$
SHR	bitweise Rechtsverschiebung	Integer	Integer	i SHR j verschiebt den Wert von i um so viele Bitpositionen nach rechts, wie der Wert von j angibt, d.h. i SHR j ist gleich i DIV 2^j
Vergleichsoperatoren				
= gleich <> ungleich		< kleiner > größer <= kleiner/gleich >= größer/gleich		

Abbildung 2.2.1-1: Operatoren mit Integer- und Real-Datentypen

2.2.2 Abgeleitete Datentypen

Der Datentyp **Aufzählungstyp** dient der Darstellung geordneter Mengen. Er wird durch die Benennung der einzelnen Elemente definiert, wobei die Ordinalzahlen der Elemente (beginnend beim Wert 0) durch die Reihenfolge festgelegt werden. Durch die Deklaration sind die Elemente ebenfalls als Konstanten festgelegt. Beispielsweise definiert der Datentyp

```
TYPE studiengang_typ = (Informatik, BWL, Mathematik, sonstiger);
```

einen Aufzählungstyp und die vier Konstanten `Informatik`, `BWL`, `Mathematik` und `sonstiger`; dabei hat die Konstante `Informatik` den Wert 0, die Konstante `BWL` den Wert 1, die Konstante `Mathematik` den Wert 2 und die Konstante `sonstiger` den Wert 3.

Die Pascal-Standardfunktion **`Ord`** kann auf den Wert eines Datenobjekts vom Aufzählungsdatentyp angewendet werden und liefert dessen Ordinalzahl.

Zwischen den Konstanten eines Aufzählungstyp gelten die üblichen Ordnungsrelationen (Vergleiche).

Die interne Darstellung eines Datenobjekts mit Aufzählungstyp ist ein Byte, wenn die Aufzählung weniger als 256 Elemente enthält, sonst ein Wort (2 Bytes). Jeder Wert eines Datenobjekts von einem Aufzählungstyp wird intern durch seine Ordinalzahl repräsentiert.

Boolesche Datentypen sind `BOOLEAN`, `ByteBool`, `WordBool` und `LongBool` mit der internen Darstellung in einem Byte (`BOOLEAN`, `ByteBool`) bzw. einem Wort (`WordBool`) bzw. einem Doppelwort (`LongBool`). Der in allen Anwendungen vorzuziehende Boolesche Datentyp ist `BOOLEAN`, der als Aufzählungstyp durch

```
TYPE BOOLEAN = (FALSE, TRUE);
```

definiert ist. Bei allen anderen Booleschen Datentypen wird der ordinale Wert 0 als `FALSE`, ein ordinaler Wert ungleich 0 als `TRUE` interpretiert (eine Reminiszenz an die Sprache C und an WINDOWS).

Datenobjekte mit Booleschem Datentyp können durch **logische Operatoren** miteinander verknüpft werden (Abbildung Abbildung 2.2.2-1).

Operator	Operation	Operanden- und Ergebnistyp	
NOT	logische Negation	Boolesch	NOT FALSE = TRUE, NOT TRUE = FALSE
AND	logisches UND	Boolesch	FALSE AND FALSE = FALSE, FALSE AND TRUE = FALSE, TRUE AND FALSE = FALSE, TRUE AND TRUE = TRUE
OR	logisches ODER	Boolesch	FALSE OR FALSE = FALSE, FALSE OR TRUE = TRUE, TRUE OR FALSE = TRUE, TRUE OR TRUE = TRUE
XOR	logisches EXKLUSIVES ODER	Boolesch	FALSE XOR FALSE = FALSE, FALSE XOR TRUE = TRUE, TRUE XOR FALSE = TRUE, TRUE XOR TRUE = FALSE

Abbildung 2.2.2-1: Logische Operatoren bei Booleschen Datentypen

Ein **Teilbereichstyp** umfasst einen ordinalen Wertebereich und wird durch die Angaben des niedrigsten und des höchsten Werts (jeweils als Konstanten) festgelegt, z.B. ist nach Definition des obigen Aufzählungstyps `studiengang_typ` die Definition des Teilbereichstyps

```
TYPE stud_gang_typ = BWL .. sonstiger;
```

möglich. Die interne Darstellung hängt von den Werten der unteren und oberen Grenze ab; alle Werte eines Teilbereichtyps müssen darstellbar sein.

Mit Hilfe der Pascal-Standardfunktion **`Chr`**, die auf den `INTEGER`-Werten 0 bis 255 definiert ist und das der jeweiligen Zahl zugeordnete Zeichen (EBCDI- oder ASCII-Code, je nach Rechnertyp) liefert, ist der Datentyp `CHAR` als Teilbereichstyp durch

```
TYPE CHAR = Chr(0) .. Chr(255);
```

definiert.

Ein Datenobjekt mit **Mengendatentyp** stellt eine (Teil-) Menge im mathematischen Sinn aus Elementen einer Grundmenge dar. Der Datentyp der Grundmenge ist ein Ordinaltyp, dessen Werte sich im Bereich 0..255 bewegen. Ein Mengendatentyp wird als `SET OF ...` deklariert.

Im folgenden Beispiel werden der Ordinaltyp `farbe` einer Grundmenge und ein Mengendatentyp `farb_menge` definiert. Der Ordinaltyp `farbe` gibt die möglichen Elemente der Grundmenge an. Datenobjekte mit Datentyp `farb_menge`, wie die Variablen `f1`, `f2` und `re-`

genbogen_farben, repräsentieren Teilmengen aller möglichen Farbwerte, die in der Grundmenge vorkommen:

```
TYPE farbe = (schwarz, weiss, rot, orange, gelb,
              gruen, blau, indigo, violett,
              sonstige_Farbe);

     farb_menge = SET OF farbe;

VAR  f1, f2, regenbogenfarben : farb_menge;
```

Mit diesen Deklarationen sind beispielsweise die Anweisungen

```
f1                := [schwarz, weiss];
f2                := [schwarz..sonstige_Farbe];
regenbogenfarben := f2 - f1;
f1                := []    { leere Menge };
```

möglich. Die Konstruktion `[...]` stellt eine Konstante mit Mengendatentyp dar. Datenobjekte mit Mengendatentyp können mit den Operatoren + (Vereinigung von Mengen) − (Differenz von Mengen) und * (Schnitt von Mengen) verknüpft werden.

Ein Datenobjekt mit Namen `set_var` und Mengendatentyp wird intern als Bit-Feld, das nach Bedarf eventuell mehrere Bytes umfasst, gespeichert, wobei die Position des jeweiligen Bits (in der Zählung von rechts nach links beginnend mit Position 0) für die Ordinalität und sein Wert für das Vorhandensein des entsprechenden Elements der Grundmenge steht: Der Bitwert 0_2 besagt, dass das der Bitposition entsprechende Element der Grundmenge in `set_var` nicht vorhanden ist, der Bitwert 1_2 besagt, dass es in `set_var` vorkommt.

Es seien ord_{min} und ord_{max} die kleinste bzw. die größte Ordinalzahl innerhalb der Grundmenge. Die Anzahl benötigter Bytes zur Darstellung eines Datenobjekts mit Mengendatentyp errechnet sich dann zu

$$(ord_{max} \text{ DIV } 8) - (ord_{min} \text{ DIV } 8) + 1.$$

Die im folgenden beschriebenen abgeleiteten Datentypen werden als **Verbunde** bezeichnet.

Ein **`RECORD`-Typ** enthält eine festgelegte Anzahl an Komponenten. Die Deklaration legt die Typen und die Bezeichner für jede Komponente fest. Er enthält einen invarianten Teil, an den sich eventuell ein varianter Teil anschließt. Formal wird ein `RECORD`-Typ mit *n* Komponeten durch

```
TYPE data_typ = RECORD
                  komponente_1 : datentyp_1;
                  ...
                  komponente_n : datentyp_n
                END;
```

definiert.

Im folgenden Beispiel dient der durch

```
TYPE funktions_typ = (Verwaltungsmitarbeiter,
                      gewerblicher_Mitarbeiter,
                      leitender_Angestellter);
```

deklarierte Aufzählungstyp der Klassifizierung der Funktionen eines Mitarbeiters in einem Unternehmen. Als mögliche Personalnummer eines Mitarbeiters sei der Zahlbereich 10000 bis 99999 zugelassen:

```
TYPE personalnr_bereich = 10000..99999.
```

Um die Höhe des Gehalts eines Mitarbeiters, seines Krankenkassenbeitrags und den Namen der Krankenkasse abzulegen, der er angehört (wobei ein leitender Angestellter lediglich einen Krankenkassenzuschuss erhält, ohne den Namen der Krankenkasse festzuhalten), wird der `RECORD`-Typ

```
TYPE ma_typ = RECORD
                  { invarianter Teil: }
                personalnummer : personalnr_bereich;
                gehalt         : REAL;
                  { varianter Teil: }
                CASE funktion: funktions_typ OF
                  Verwaltungsmitarbeiter,
                  gewerblicher_Mitarbeiter:
                    (beitrag      : REAL;
                     krankenkasse : STRING [15]);
                  leitender_Angestellter:
                    (zuschuss     : REAL)
              END;
```

deklariert. Den durch

```
VAR mitarbeiter_1, mitarbeiter_2 : ma_typ;
```

deklarierten Variablen werden Werte zugewiesen:

```
WITH mitarbeiter_1 DO
  BEGIN
    personalnummer := 12000;
    gehalt         := 4940.00;
    funktion       := gewerblicher_Mitarbeiter;
    beitrag        := 642.20;
    krankenkasse   := 'XYZ Kasse'
  END;

WITH mitarbeiter_2 DO
  BEGIN
    personalnummer := 25000;
    gehalt         := 9500.00;
    funktion       := leitender_Angestellter;
    zuschuss       := 210.00
  END;
```

Ein **`ARRAY`-Typ** spezifiziert die Struktur eines Felds, indem der Datentyp der einzelnen Feldelemente, deren Anzahl (in Form einer Dimensionsangabe mit unterer und oberer Indexgrenze für jede Dimension) und der Datentyp der Indizes (Ordinaltyp) festgelegt wird. Pascal akzeptiert nur statische Felder, so dass der Speicherplatzbedarf eines Datenobjekts mit `ARRAY`-Typ zur Übersetzungszeit festliegt.

Der Datentyp einer Arbeitsgruppe mit 20 Mitarbeitern könnte etwa durch

```
TYPE arbeitsgruppen_typ = ARRAY [1..20] OF ma_typ;
```

deklariert werden.

Ein mehrdimensionales Feld kann als Datenobjekt mit (eindimensionalen) `ARRAY`-Typ definiert werden, deren Feldelemente wieder vom `ARRAY`-Typ sind. So führen die in der folgenden Deklaration vorkommenden `ARRAY`-Typen `feld_typ_1` und `feld_typ_2` zu Datenobjekten mit derselben internen Darstellung.

```
TYPE  anzahl = 1..10;

      feld_typ_1 = ARRAY [anzahl] OF
                     ARRAY [BOOLEAN] OF
                       ARRAY [1..10] OF REAL;
      feld_typ_2 = ARRAY [anzahl, BOOLEAN, 1..10] OF REAL;
```

Für die Behandlung variabel langer Zeichenketten ist der Datentyp **`STRING`-Typ** vorgesehen. Die Deklaration

```
VAR data : STRING [n];
```

definiert ein Datenobjekt, dessen Werte Zeichenketten der maximalen Länge von *n* Zeichen sein können. Fehlt die Angabe von *n*, so wird implizit der maximal mögliche Wert für *n* genommen (bei Pascal *n* = 255).

Zur internen Darstellung des Datenobjekts werden *n* + 1 aufeinanderfolgende Bytes belegt, die von 0 bis *n* indiziert sind. Das Byte an der Position 0 (an der niedrigsten Adresse) enthält als (vorzeichenlose) Ordinalzahl die Länge der gerade im Datenobjekt gespeicherten Zeichenkette. Die Pascal-Standardfunktion **`Length`** liefert diese Längenangabe. Die Zeichen der Zeichenkette belegen die Bytes mit Index 1 bis *n* (aufsteigende Adressen).

Zwei Zeichenketten sind gleich, wenn ihre aktuellen Längen und ihre Werte übereinstimmen. Eine Zeichenkette *X* ist kleiner als eine Zeichenkette *Y*, wenn entweder die Länge von *X* kleiner als die Länge von *Y* ist oder wenn sie bei gleicher Länge bis zu einer Position *k* dieselben Zeichen enthalten und der (ASCII-) Code des Zeichens an der Position *k* + 1 von *X* kleiner ist als der entsprechende Code an der Position *k* + 1 von *Y*. Das kleinste Datenobjekt mit `STRING`-Typ ist ein **Nullstring**, d.h. eine Zeichenkette der Länge 0.

Eine Reihe von Standardfunktionen zur Manipulation von Datenobjekten mit `STRING` -Typ definieren Funktionen zur Aneinanderreihung von Zeichenketten (Konkatination), das Entfernen von Teilen aus einer Zeichenkette, das Einfügen von Zeichenketten in eine andere ab einer gegebenen Position und das Suchen der kleinsten Position eines vorgegebenen Zeichens in einer Zeichenkette. Die Konkatination zweier Zeichenketten bewirkt auch durch der Operator +. Die Anweisungen

```
xzeichen := '123 A';
xzeichen := '### ' + xzeichen + '???';
```

setzen die Variable `xzeichen` mit `STRING`-Typ auf den Wert `### 123 A???`.

Eine weitere Möglichkeit der Deklaration eines Datenobjekts mit Zeichenkettenwerten ist durch

```
VAR data : PACKED ARRAY [index_typ] OF CHAR;
```

gegeben, wobei `index_typ` ein Teilbereichstyp ist. Die interne Darstellung besteht aus einem Datenobjekt mit Datentyp Zeichenkette, die so viele Bytes belegt, wie die Differenz der Ober- und Untergrenze von `index_typ` angibt.

Die Deklaration

```
VAR data : ARRAY [0.. idx] OF CHAR;
```

definiert ein Datenobjekt mit **nullbasierten `STRING`-Typ (nullterminierten Zeichenkette**), wobei *idx* ein ganzzahliger von 0 verschiedener Ordinalwert ist. Dieses Datenobjekt kann eine nullterminierte Zeichenkette aufnehmen, d.h. eine Zeichenkette, deren Ende durch das Zeichen `#0` definiert ist. Eine derartige Zeichenkette kann eine maximale Länge von 65.535 Zeichen haben. Spezielle Standardfunktionen erlauben Operationen mit nullterminierten Zeichenketten und Datenobjekten (siehe [PAS]).

2.2.3 Zeigerdatentypen

Datenobjekte, die über die bisher beschriebenen Datentypen deklariert werden, belegen intern eine durch den jeweiligen Datentyp bestimmte Anzahl von Bytes. Diese Speicherplatzbelegung erfolgt automatisch ohne Zutun des Anwenders, sobald der Block (Prozedur, Programmteil), der die Deklaration enthält, „betreten“ wird. Selbst wenn ein Datenobjekt dann aufgrund der Programmlogik während der Laufzeit gar nicht verwendet wird, belegt es trotzdem Speicherplatz. Eine Möglichkeit, einem Datenobjekt Speicherplatz dynamisch erst während der Laufzeit nach Bedarf zuzuweisen bzw. zu entziehen und damit seine Lebensdauer in Abhängigkeit von der Anwendung zu begrenzen, besteht in der Verwendung von Zeigertypen und einem „indirekten Zugriff“ auf das jeweilige Datenobjekt. Dazu wird neben dem betreffenden Datenobjekt `D` ein weiteres Datenobjekt `D_ptr` angelegt, das die Adresse von `D` aufnimmt, sobald `D` auch wirklich benötigt und Speicherplatz bereitgestellt wird. Der Wert von `D_ptr` wird als **Zeiger (Pointer) auf** `D` bezeichnet; der Datentyp von `D_ptr` ist ein **Zeigerdatentyp (Pointertyp)**.

Ein Datenobjekt mit Zeigerdatentyp kann als Wert eine Adresse, d.h. eine Referenz auf ein weiteres Datenobjekt, oder den Wert **`NIL`** enthalten. Der Wert `NIL` repräsentiert hier einen Adressverweis, der „nirgendwo hinzeigt“ (undefinierte Adresse). Intern belegt ein Datenobjekt mit Zeigerdatentyp so viele Bytes, wie zur internen Darstellung einer Adresse notwendig sind.

Die Syntax des Zeigerdatentyp wird an folgenden Beispiel deutlich:

```
TYPE Pointertyp = ^Tgrund_typ;
          { ^ bedeutet Verweis auf ein Datenobjekt,
            das den auf ^ folgenden Datentyp besitzt }
     Tgrund_typ  = ...;

VAR  datenobjekt : Pointertyp;
```

Das Datenobjekt mit Bezeichner `datenobjekt` und dem Zeigertyp `Pointertyp` kann die Adresse eines anderen Datenobjekts aufnehmen, das den Datentyp `Tgrund_typ` hat, oder den Wert `NIL`.

Für ein so deklariertes Datenobjekt mit Pointertyp sind nur wenige Operationen zugelassen:

- Zuweisung der Adresse eines Datenobjekts mit Datentyp `Tgrund_typ` oder die Zuweisung des Werts `NIL`
- Vergleich auf Wertgleichheit mit einem anderen Datenobjekt mit Datentyp `Pointertyp` oder mit dem Wert `NIL`
- „Freigabe“ des Datenobjekts, dessen Adresse gerade als Wert in `datenobjekt` enthalten ist.

Ist das Datenobjekt mit Namen `ref_data` durch

```
VAR  ref_data : Pointertyp   { siehe oben };
```

definiert, so sind beispielsweise die Operationen

```
ref_data    := NIL;
datenobjekt := ref_data;
...
IF ref_data = NIL THEN ... ELSE ...;
...
IF datenobjekt <> ref_data THEN ...;
```

erlaubt.

Durch den Aufruf der Pascal-Standardprozedur **`New`** in der Form

```
New (datenobjekt);
```

wird ein neues Datenobjekt mit Datentyp `Tgrund_typ` erzeugt und dessen Adresse (als Wert) dem Datenobjekt mit Bezeichner `datenobjekt` zugewiesen. Dieses neue Datenobjekt liegt dann in einem für dynamisch erzeugte Datenobjekte vorgesehenen (virtuellen) Speicherbereich, dem Heap (siehe Kapitel 3.1).

Ein Zugriff auf den Wert dieses neu erzeugten Datenobjekts erfolgt ausschließlich über die in `datenobjekt` abgelegte Adresse; es hat (anders als die „normalen“ Datenobjekte) keinen Bezeichner. Das Datenobjekt, dessen Adresse in `datenobjekt` liegt, ist durch die syntaktische Konstruktion

```
datenobjekt^
```

erreichbar.

Ist der Datentyp `Tgrund_typ` wie oben durch

```
TYPE Tgrund_typ = RECORD
                    wert   : LongInt;
                    belegt : BOOLEAN
                  END;
```

definiert, so werden die Werte 100 und `TRUE` in die Komponenten `wert` bzw. `belegt` durch

```
datenobjekt^.wert   := 100;
datenobjekt^.belegt := TRUE;
```

gesetzt; die Anweisung

```
IF datenobjekt^.wert > 100 THEN ... ELSE ...;
```

prüft den Wert der Komponente `wert`.

Die Pascal-Standardprozedur **`Dispose`** in der Form

```
Dispose (datenobjekt);
```

gibt das Datenobjekt, dessen Adresse in `datenobjekt` steht, frei, d.h. dieses Datenobjekt existiert nach Ausführung der Prozedur `Dispose` nicht mehr.

Durch die Deklarationen

```
TYPE Pointertyp = ^Tgrund_typ;
     Tgrund_typ  = ...;

VAR  datenobjekt : Pointertyp;
```

wird also noch kein Datenobjekt mit Datentyp `Tgrund_typ` deklariert, sondern nur ein Datenobjekt, das die *Adresse* eines Datenobjekts mit Datentyp `Tgrund_typ` aufnehmen kann. Erst durch den Aufruf der Prozedur

```
New (datenobjekt);
```

wird es eingerichtet, seine Adresse in mit Datentyp `datenobjekt` abgelegt, und es beginnt seine Lebensdauer. Sie endet mit der Ausführung der Prozedur `Dispose`, der als Parameter

die Adresse des Datenobjekts mitgegeben ist; ein Zugriff auf dieses Datenobjekt ist nun nicht mehr möglich.

Der Heap belegt in der Regel einen Speicherplatz, dessen Größe durch Steuerungsparameter bestimmt wird. Bei Programmstart ist der gesamte Heap als frei gekennzeichnet. Die Ausführung von `New` reserviert fest den Speicherplatz für ein neues Datenobjekt (z.B. mit Datentyp `grund_typ`). Der so reservierte Platz innerhalb des Heaps wird erst durch `Dispose` zur weiteren Verwendung wieder freigegeben. Auf diese Weise enthält der Heap reservierte und freie Bereiche, deren Größen und Verteilung sich während der Programmlaufzeit dynamisch ändern. Das Laufzeitsystem führt eine Liste der nicht-reservierten Bereiche (Anfangsadresse und Anzahl Bytes) im Heap. Die Einträge dieser Liste sind über Adressverweise logisch miteinander verkettet und nach Lückengröße geordnet. Bei Aufruf von `New` wird zunächst versucht, für das neu einzurichtende Datenobjekt eine durch vorhergehende Dispose-Aufrufe entstandene Lücke zu nutzen. Schließt sich bei einem Dispose-Aufruf die neu entstehende Lücke nahtlos an eine bereits existierende Lücke an, so werden diese Bereiche zu einem zusammenhängenden freien Bereich zusammengelegt. Aus Performancegründen wird auf komplexere Verfahren (Garbage Collection) zur Heaporganisation meist verzichtet.

Das Beispiel in Abbildung 2.2.3-1 zeigt schematisch das Zusammenspiel von `New` und `Dispose` und die Speicherbelegung bei einem Programmausschnitt. Durch eine fehlerhafte Reihenfolge im Gebrauch der Prozedur `New` (zu sehen beim Übergang von Bild (b) nach Bild (c)) ist der Adressverweis auf das Datenobjekt mit `wert`-Komponente 100 überschrieben worden. Der Speicherplatz, den dieses Datenobjekt im Heap belegt, kann nicht mehr durch `Dispose` freigegeben werden; er bleibt während der gesamten Programmlaufzeit reserviert.

```
PROGRAM beispiel;

  TYPE Pointertyp = ^Tgrund_typ;

        Tgrund_typ = RECORD
                       wert   : INTEGER;
                       belegt : BOOLEAN
                     END;

  VAR  datenobjekt : Pointertyp;
       ref_data    : Pointertyp;
  ...
  BEGIN

   New (datenobjekt);
   datenobjekt^.wert   := 100;
   datenobjekt^.belegt := TRUE;    { Bild (a) }

   New (ref_data);
   ref_data^.wert   := 200;
   ref_data^.belegt := TRUE;       { Bild (b) }

   New (datenobjekt);
   datenobjekt^.wert   := 300;
   datenobjekt^.belegt := TRUE;

   Dispose (ref_data);             { Bild (c) }

   ref_data := datenobjekt;
   New (datenobjekt);
   datenobjekt^.wert   := 400;
   datenobjekt^.belegt := TRUE;    { Bild (d) }

   ...
  END;
```

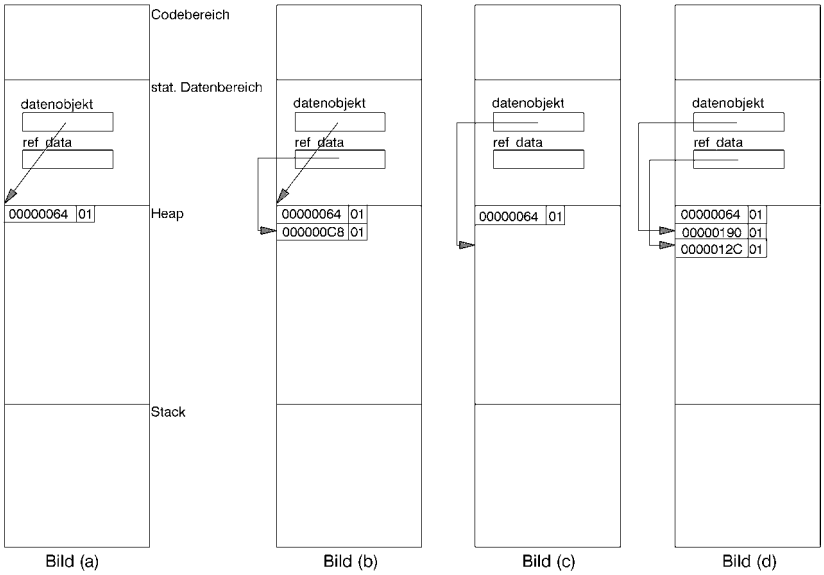


Abbildung 2.2.3-1: Belegung des Heaps (Beispiel)

Im folgenden Beispiel kommt es zu einem Laufzeitfehler, da auf ein nicht-existentes Datenobjekt zugegriffen wird.

```
PROGRAM achtung;

TYPE Pointertyp = ^Tgrund_typ;

        Tgrund_typ = RECORD
                       wert   : INTEGER;
                       belegt : BOOLEAN
                     END;

VAR  datenobjekt : Pointertyp;
     ref_data    : Pointertyp;

BEGIN
  New (datenobjekt);
  ref_data := datenobjekt
            { beide Datenobjekte enthalten
              dieselben Adressverweise       };
  Dispose (datenobjekt);
  ref_data^.wert := 100
            { das Datenobjekt, dessen Adresse
              in ref_data steht, existiert
              nicht mehr                     }
END.
```

Ein weiterer Zeigerdatentyp in Pascal ist der Datentyp **`Pointer`**, der für einen untypisierten Zeiger steht. Dieser Datentyp ist wie die Konstante `NIL` mit allen Zeigertypen kompatibel. Möchte man allerdings auf ein Datenobjekt, das durch den Wert eines Datenobjekts `Pointer` referenziert wird, zugreifen, so muss vorher eine explizite Typisierung vorgenommen werden. Das Vorgehen erläutert das folgende Beispiel:

```
...
TYPE Tgrund_typ    = RECORD
                       belegt : BOOLEAN;
                       wert   : INTEGER;
                     END;

     feld_typ     = ARRAY [1..300] OF Tgrund_typ;
     feld_typ_ptr = ^feld_typ;

     Buftyp = ARRAY[1..4096] OF Byte;

VAR  ptr    : Pointer;
     BufPtr : ^Buftyp;

BEGIN

  ...
  New(BufPtr);           { Puffer im Heap anlegen            }
  ptr := BufPtr;         { Puffer über ptr adressieren       }

  { Ein ARRAY mit Datentyp feld_typ über den Puffer legen,
    d.h. den Puffer, wie feld_typ angibt, strukturieren, und
    einen Wert in das erste Feldelement übertragen           }
  feld_typ_ptr(ptr)^[1].belegt := TRUE;
  feld_typ_ptr(ptr)^[1].wert   := 200;
  ...

END.
```

Über den **Adressoperator** @ kann die Adresse einer Variablen, Prozedur, Funktion oder Methode ermittelt und einem Zeiger zugeordnet werden.

2.2.4 Typkompatibilität

Bei Operationen, Wertzuweisungen und Prozedur- und Funktionsaufrufen müssen die Datentypen der beteiligten Datenobjekte Kompatibilitätsregeln genügen. Im folgenden Beispiel werden sechs Felder deklariert, die jeweils aus 5 Feldelementen mit Datentyp `INTEGER` bestehen:

```
TYPE feld_typ_1 = ARRAY [1..5] OF INTEGER;
     feld_typ_2 = ARRAY [1..5] OF INTEGER;

VAR  a    : feld_typ_1;
     b    : feld_typ_2;
     c    : ARRAY [1..5] OF INTEGER;
     d, e : ARRAY [1..5] OF INTEGER;
     f    : feld_typ_1;
```

In der internen Darstellung unterscheiden sich diese Felder nicht; sie sind **strukturäquivalent**. Pascal setzt für die Gleichheit von Datentypen jedoch **Namensäquivalenz** voraus, d.h. zwei Variablen haben nur dann einen gleichen Datentyp, wenn sie mit demselben Datentypbezeichner definiert wurden. Im Beispiel haben `a` und `f` bzw. `d` und `e` denselben Datentyp, `a`, `b`, `c` und `d` jedoch nicht (da `ARRAY [1..5] OF INTEGER` kein Datentyp*bezeichner* ist, haben `c` und `d` unterschiedliche Datentypen).

Für die Datentypen der Aktualparameter und der korrespondierenden Formalparameter bei Prozedur- und Funktionsaufrufen wird in Pascal Gleichheit im Sinne der Namensäquivalenz vorausgesetzt. Erweiterte Kompatibilitätsregeln gelten bei der Bildung von Ausdrücken, relationalen Operationen und Wertzuweisungen. Die Einhaltung dieser Regeln, von denen die wichtigsten in Abbildung 2.2.4-1 zusammengefasst sind, wird vom Compiler überprüft.

Zwei Datentypen sind kompatibel, wenn mindestens eine der Bedingungen erfüllt ist:
Beide Typen sind gleich (Namensäquivalenz). Beide Typen sind Real-Datentypen. Beide Typen sind Integer-Datentypen. Ein Typ ist ein Teilbereichstyp des anderen. Beide Typen sind Teilbereiche derselben Grundmenge. Beide Typen sind Mengendatentypen mit kompatiblen Datentypen der Grundmenge. Beide Typen sind `STRING`-Datentypen mit der gleichen Anzahl von Komponenten. Der eine Typ ist ein `STRING`-Datentyp und der andere ist ein `PACKED ARRAY`-Datentyp mit nicht mehr Zeichen, als im `STRING`-Datentyp aufgrund seiner Definition maximal zugelassen sind. Ein Typ ist der Datentyp `Pointer`, der andere ein beliebiger Zeigertyp.
Der Wert eines Datenobjekts mit Datentyp T_2 kann einem Datenobjekt mit Datentyp T_1 zugewiesen werden, wenn eine der folgenden Bedingungen erfüllt ist (**Zuweisungskompatibilität**):
T_1 und T_2 sind kompatible Ordinaltypen, und der Wert von T_2 liegt im möglichen Wertebereich von T_1. T_1 und T_2 sind kompatible Real-Datentypen, und der Wert von T_2 liegt im möglichen Wertebereich von T_1. T_1 hat den Datentyp `REAL` und T_2 den Datentyp `INTEGER`. T_1 und T_2 sind `STRING`-Datentypen. T_1 ist ein `STRING`-Datentyp, und T2 ist ein `CHAR`-Typ. T_1 ist ein `STRING`-Typ, und T_2 ist ist ein `PACKED ARRAY`-Typ mit nicht mehr Zeichen, als im `STRING`-Typ aufgrund seiner Definition maximal zugelassen sind. T_1 und T_2 sind kompatible Mengendatentypen, und alle Elemente des Werts von T_2 fallen in den möglichen Wertebereich von T_1. T_1 und T_2 sind kompatible Zeigerdatentypen. (Zusätzliche Regeln gelten für die objektorientierte Programmierung.)

Abbildung 2.2.4-1: Wichtige Typkompatibilitätsregeln

2.2.5 Aspekte der Objektorientierung und Klassendatentypen

In diesem Unterkapitel wird zunächst auf die grundlegenden Prinzipien der objektorientierten Programmierung eingegangen. Dabei wird vorausgesetzt, dass bereits Basiskenntnisse über objektorientierte Methodiken bekannt sind. Anschließend werden die syntaktischen Hilfsmittel zur objektorientierten Programmierung der Sprache Object Pascal im Überblick vorgestellt. Viele Details werden in den späteren Kapiteln im jeweiligen Zusammenhang behandelt.

In der **objektorientierten Modellierung (OOM)** und **der objektorientierten Programmierung (OOP)** wird ein Problem dadurch zu lösen versucht, dass man ein **Modell von miteinander kommunizierenden Objekten** aufstellt. Dazu identifiziert man aus dem jeweiligen Realitätsbereich **Objekte**, ihre gegenseitigen Abhängigkeiten und die mit den Objekten verknüpften **Operationen**, die hier **Methoden** genannt werden. Die Formulierung des Modells und seine Umsetzung in Programme erfolgt in einer geeigneten Programmiersprache.

OOP ist bedingt in „klassischen“ Programmiersprachen möglich. Das Gesamtkonzept wird aber durch spezielle Sprachen (Smalltalk, Flavor, Eifel usw.) anwendungsorientiert unterstützt. Andere Ansätze erweitern bekannte Sprachen. Beispiele sind C++, Java und Object Pascal. Allein eine Programmierung mit einer dieser Sprachen macht jedoch noch nicht die OOP aus.

Ein Charakteristikum der Objekte vieler Modelle ist die Tatsache, dass Objekte gemeinsame Eigenschaften aufweisen und zu einer **(Objekt-) Klasse** zusammengefasst werden. Alle Objekte einer Objektklasse haben daher dieselben Eigenschaften; insbesondere gleichen sich alle Methoden auf (d.h. Operationen mit) diesen Objekten. Eine Klasse definiert daher einen **Objekttyp (Klassentyp)**. Objekte unterschiedlicher Objektklassen können durchaus gemeinsame (Teil-) Eigenschaften besitzen; mindestens eine Eigenschaft trennt jedoch die Objekte der verschiedenen Objektklassen.

Es können **Hierarchien von Objektklassen** aufgebaut werden: Eine Objektklasse übernimmt alle Eigenschaften einer **übergeordneten Objektklasse** und fügt weitere Eigenschaften hinzu, spezialisiert also dadurch eine in der Hierarchie weiter oben stehende Objektklasse. Anwendbar sind alle Operationen, die in einer übergeordneten Objektklasse definiert sind, auch auf die Objekte einer dieser Klasse untergeordneten Objektklasse, falls die Operationen in der untergeordneten Klasse nicht neu definiert werden. Eine Objektklasse übernimmt ja von einer übergeordneten Objektklasse alle Eigenschaften; die dort definierten Operationen beziehen sich nur auf diese übernommenen Eigenschaften. Der Vorgang der Übernahme von Eigenschaften übergeordneter Objektklassen nennt man **Vererbung**. Die Art und Weise der Vererbung, z.B. die Vererbung aus einer einzigen Oberklasse (**einfache Vererbung**) oder mehreren Oberklassen (**multiple Vererbung**), und der Zeitpunkt der Vererbung, z.B. bei der Deklarati-

on eines Objekts (**statische** Vererbung) oder erst bei einer Operationsanwendung (**dynamische Vererbung**), prägen den objektorientierten Ansatz.

Jedes Objekt besteht aus:

- **Daten**, auch **Komponenten** genannt: sie besitzen einen Datentyp und können Werte annehmen, die während der Lebensdauer des Objekts veränderbar sind
- **Eigenschaften (properties)**: sie gleichen den Daten des Objekts, erlauben es aber zusätzlich, definierte Aktionen mit dem Lesen einer Eigenschaft bzw. dem Verändern einer Eigenschaft zu verbinden
- **Methoden**: Operationen, die auf die Komponenten und Eigenschaften des Objekts zugreifen und deren Werte eventuell verändern.

Zusätzlich wird jedes Objekt durch einen eindeutigen **Objektbezeichner** identifiziert. Die gegenwärtigen Werte der Daten (Komponenten) und Eigenschaften eines Objekts bestimmen seinen **Objektzustand**.

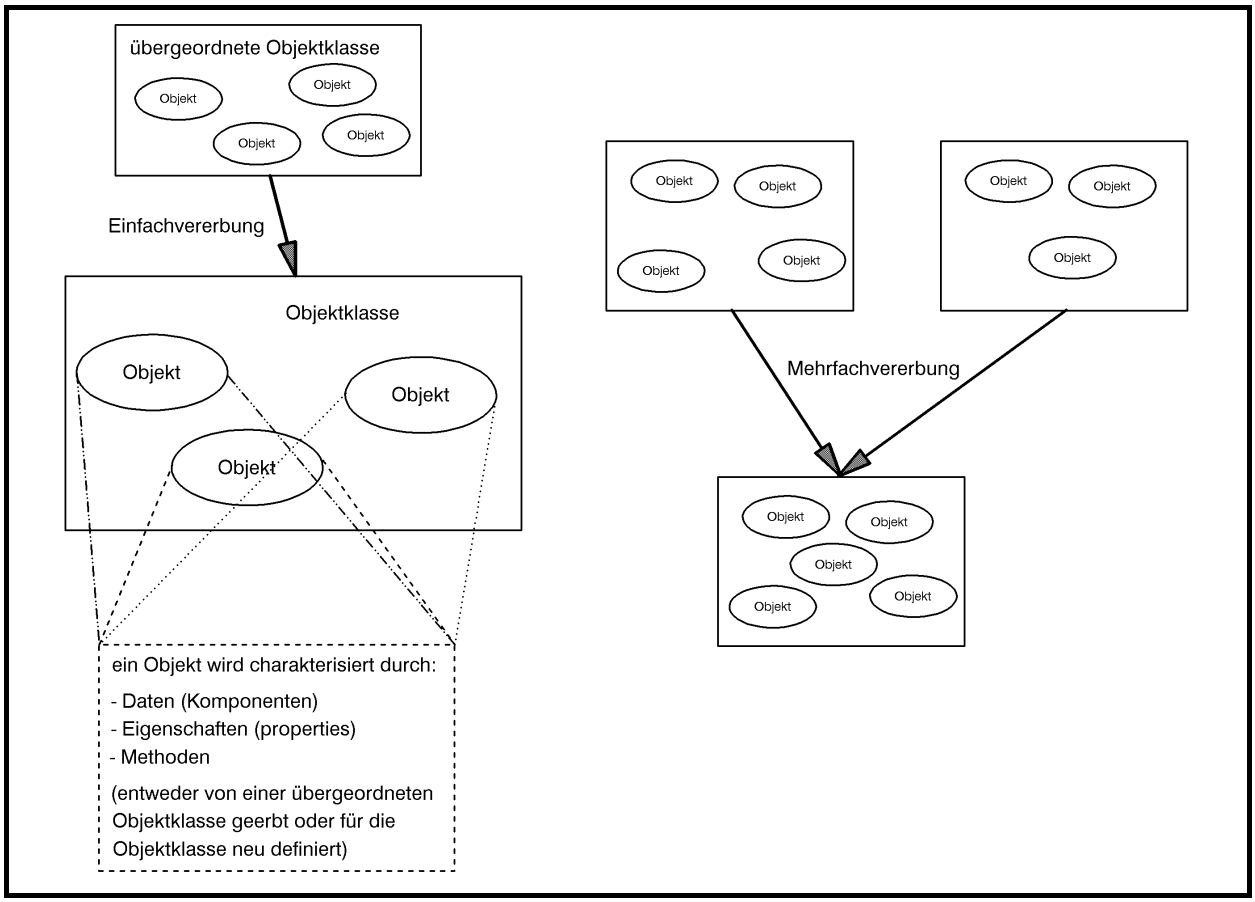


Abbildung 2.2.5-1: Objektkonzept der objektorientierten Programmierung

Prinzipiell besteht eine **vollständige Kapselung** der Daten und Eigenschaften eines Objekts nach außen hin, d.h. nur die für ein Objekt definierten Methoden können auf die einzelnen Komponenten des Objekts zugreifen und deren Werte verändern. Gerade dieser Aspekt ist jedoch in den Programmiersprachen, die erst später um die Konzepte der OOP erweitert wurden

wie C++, Java oder Object Pascal, nicht streng realisiert, so dass die Einhaltung der Kapselung dem Programmierer eine gewisse Programmierdisziplin auferlegt.

Die **Interaktion zwischen Objekten** erfolgt durch **Austausch von Nachrichten**: Ein Objekt ruft dazu eine Methode eines anderen Objekts auf und verändert dadurch dessen Objektzustand (und folglich damit den Zustand des Gesamtsystems). Das Zielobjekt reicht mit dem gleichen Mechanismus eventuell ein Resultat zurück. Das Versenden einer Nachricht ist damit mit dem Prozeduraufruf in einer konventionellen Programmiersprache vergleichbar. Dabei ist zu beachten, dass der gleiche Methodenbezeichner bei unterschiedlichen Objekten auch unterschiedliche Reaktionen hervorrufen kann und dieses sogar in „kontrolliertem Maß“ bei unterschiedlichen Objekten derselben Objekthierarchie. Dieses Konzept wird als **Polymorphie** bezeichnet.

In diesem Interaktionskonzept ist der Empfänger einer Nachricht dem Absender der Nachricht bekannt. Einige Systeme erlauben zusätzlich das Versenden von **Botschaften**, deren möglichen Empfänger der Absender nicht kennt: eine Botschaft wird durch einen **Botschaftsbezeichner** identifiziert und kann weitere Parameter enthalten. Eine Objektklasse hat zu diesem Botschaftsbezeichner eine Methode, die **Botschaftsbehandlungsmethode**, deklariert. Beim Absenden einer Botschaft durch ein Objekt wird zunächst in der eigenen Objektklasse nach der Botschaftsbehandlungsmethode zu dieser Botschaft gesucht und bei einer nicht erfolgreichen Suche durch die Objektklassenhierarchie gegangen, bis eine entsprechende Botschaftsbehandlungsmethode gefunden wurde.

Die Umsetzung dieses allgemeinen Konzepts wird im folgenden ausschnittsweise an der Sprache Object Pascal erläutert.

Ein **Klassentyp** (**Objekttyp**), Schlüsselwort **`CLASS`**, definiert eine Struktur von Feldern, Methoden und Eigenschaften:

- Ein **Feld** (Datenelement, Komponente) ist im wesentlichen eine Variable, die zu einem Objekt dieses Klassentyps gehört. Es definiert ein Datenelement, das in jedem Objekt dieses Objekttyps vorhanden ist
- Eine **Methode** ist eine Prozedur oder Funktion, die zu dem Klassentyp gehört
- Eine **Eigenschaft (property)** ist eine Schnittstelle zu den Daten eines Objekts (die oftmals in einem Feld gespeichert sind). Eigenschaften verfügen über Zugriffsangaben (`READ`, `WRITE`) und optionalen Angaben zur Verwaltung von Laufzeit-Typinformationen (`STORED`, `DEFAULT` oder `NODEFAULT`) und optionalen `IMPLEMENTS`-Angaben. Jede Eigenschaftsdeklaration muss zumindest einen `READ`- oder `WRITE`-Bezeichner enthalten. Die Zugriffsangaben bestimmen, wie ihre Daten gelesen und geändert werden. Sie erscheinen für die anderen Bestandteile eines Programms (außerhalb des Objekts) in vielerlei Hinsicht wie ein Feld. Eine Eigenschaft definiert (wie ein Feld) ein Attribut eines Objekts. Felder sind jedoch nur Speicherbereiche, die überprüft und geändert werden kön-

nen. Eigenschaften können hingegen mit Hilfe bestimmter Aktionen gelesen und geschrieben werden. Sie erlauben eine größere Kontrolle über den Zugriff auf die Attribute eines Objekts und ermöglichen das Berechnen von Attributen.

Für Felder, Methoden und Eigenschaften lassen sich mit Hilfe der Schlüsselwörter `PRIVATE`, `PUBLIC`, `PUBLISHED` und `PROTECTED` **Sichtbarkeitsregeln** vereinbaren:

- Auf ein `PRIVATE`-Element kann nur innerhalb des Moduls zugegriffen werden, in dem die Klasse deklariert ist. Mit anderen Worten: eine `PRIVATE`-Methode kann nicht von anderen Modulen aufgerufen werden, und als `PRIVATE` deklarierte Felder oder Eigenschaften können nicht von anderen Modulen gelesen oder geschrieben werden. Indem man verwandte Klassendeklarationen im selben Modul zusammenfasst, kann man diesen Klassen also den Zugriff auf alle `PRIVATE`-Elemente ermöglichen, ohne die Elemente anderen Modulen bekanntzumachen
- Ein `PUBLIC`-Element unterliegt keinerlei Zugriffsbeschränkungen. Es ist überall dort sichtbar, wo auf seine Klasse verwiesen werden kann
- Für `PUBLISHED`-Elemente gelten dieselben Sichtbarkeitsregeln wie für `PUBLIC`-Elemente, jedoch werden für `PUBLISHED`-Elemente Laufzeit-Typinformationen generiert. Sie ermöglichen einer Anwendung, die Felder und Eigenschaften eines Objekts dynamisch abzufragen und seine Methoden zu lokalisieren. Diese Eigenschaften werden von der Entwicklungsumgebung Delphi verwendet, um beim Speichern und Laden von Formulardateien auf die Werte von Eigenschaften zuzugreifen, Eigenschaften im Objektinspektor anzuzeigen und spezielle Methoden (sogenannte Ereignisbehandlungsroutinen) bestimmten Eigenschaften (den Ereignissen) zuzuordnen. Es gibt Einschränkungen bezüglich der Möglichkeiten, Elemente als `PUBLISHED` deklarieren zu können; auf Details wird hier nicht weiter eingegangen, da im folgenden Text nur `PUBLIC`- und `PRIVATE`-Elemente verwendet werden
- Ein `PROTECTED`-Element ist innerhalb des Moduls mit der Klassendeklaration und in allen abgeleiteten Klassen (unabhängig davon, in welchem Modul sie deklariert sind) sichtbar.

Der Ausgangspunkt der Klassenhierarchie, d.h. der „Urahn“ aller anderen Objekttypen, ist der Objekttyp `TObject`. Dieser kapselt das grundlegende Verhalten, das allen Objekten in Object Pascal gemeinsam ist. Mit den von `TObject` eingeführten Methoden lassen sich

- Objekte (**Objektinstanzen**) erzeugen, verwalten und auflösen. Dies geschieht durch Zuweisen, Initialisieren und Freigeben von Speicher für das Objekt
- auf objektspezifische Informationen über den Klassentyp und die Instanz zugreifen und Laufzeitinformationen von Eigenschaften verwalten, die als `PUBLISHED` deklariert sind
- Botschaften individuell bearbeiten.

Wenn bei der Deklaration eines neuen Objekttyps kein Vorfahr angegeben wird, setzt Object Pascal als Vorfahr automatisch die Klasse `TObject` ein.

Die Leistungsfähigkeit der Objekte beruht zu einem großen Teil auf den Methoden, die `TObject` einführt. Viele dieser Methoden sind für die interne Verwendung in der Delphi-Entwicklungsumgebung vorgesehen, nicht aber für einen direkten Aufruf durch den Anwender. Andere Methoden müssen in abgeleiteten Objekten und Komponenten, die ein komplexeres Verhalten zeigen, überschrieben, d.h. neu deklariert werden.

Das folgende Beispiel zeigt die Deklaration der Klasse `TListColumns` in Object Pascal:

```
TYPE
      TListColumns = CLASS (TCollection)
        PRIVATE
          FOwner : TCustomListView;
          FUNCTION GetItem (Index : INTEGER): TListColumn;
         PROCEDURE SetItem (Index : INTEGER;
                            Value : TListColumn);
        PROTECTED
          FUNCTION GetOwner : TPersistent; OVERRIDE;
          PROCEDURE Update (Item : TCollectionItem); OVERRIDE;
        PUBLIC
          CONSTRUCTOR Create (AOwner : TCustomListView);
          FUNCTION Add : TListColumn;
          PROPERTY Owner : TCustomListView READ FOwner;
          PROPERTY Items [Index : INTEGER] : TlistColumn
                     READ GetItem WRITE SetItem; DEFAULT;
      END;
```

`TListColumns` ist von einem zuvor deklarierten Klassentyp `TCollection`, die hier nicht angegeben wird, abgeleitet und erbt die meisten Elemente dieser Klasse. Zusätzlich werden mehrere Eigenschaften und Methoden einschließlich des Konstruktors `Create` definiert. Ein **Konstruktor** ist eine Methode mit einem speziellen Verhalten, die zur Erzeugung von Objektinstanzen dieses Typs aufgerufen wird. Der Destruktor `Destroy` wird ohne Änderung von `TCollection` übernommen und daher nicht erneut deklariert. Ein **Destruktor** ist eine spezielle Methode, die verwendet wird, wenn ein Objekt dieses Objekttyps aus dem System entfernt wird. Die Rollen von Konstruktoren und Destruktoren werden im Zusammenhang mit Betrachtungen zur Implementieren von Methoden in Kapitel 3.3 näher betrachtet.

Ausgehend von dieser Deklaration kann ein `TListColumns`-Objekt folgendermaßen erstellt werden:

```
VAR ListColumns : TListColumns;
```

```
ListColumns := TListColumns.Create(SomeListView);
```

`SomeListView` ist hierbei eine Variable, die ein `TCustomListView`-Objekt enthält.

Eine Objektklasse kann neben Feldern und Eigenschaften insbesondere Methoden an Nachfahren vererben und geerbte Komponenten überschreiben, d.h. mit demselben Bezeichner und veränderter Bedeutung neu definieren. Je nach Art der **Vererbung der Methoden** unterscheidet man

- statische Methoden
- virtuelle Methoden
- dynamische Methoden
- abstrakte Methoden.

Die Unterschiede werden besonders deutlich, wenn man die Prinzipien der Implementierungen der Methoden ansieht; dieses geschieht in Kapitel 3.3.

Leider ist beim Entwurf der Sprache Object Pascal eine **konzeptuelle Inkonsistenz** eingeführt worden, die sich auch bereits in C++ und dann in Java wiederfindet, nämlich die Vermischung von Variablendeklarationen und Variablenreferenzdeklarationen. Die folgende Betrachtung erläutert den Sachverhalt.

Durch die Deklarationen

```
TYPE Prec_typ = ^Trec_typ;
     Trec_typ = RECORD
                  zahl : INTEGER;
                  txt  : STRING [10];
                  ptr  : Prec_typ
                END;

VAR vrec : Trec_typ;
    vptr : Prec_typ;
```

werden zwei Variablen `vrec` und `vptr` deklariert. Die Variable `vrec` belegt so viel Speicherplatz, wie nötig ist, um ihre Komponenten `zahl` (eine Festpunktzahl mit 16 bzw. 32 Bits), `txt` (eine Zeichenkette mit 11 Bytes, einschließlich Byte 0, das die aktuelle Länge der Zeichenkette enthält) und `ptr` (eine Adresse etwa mit 16 bzw. 32 Bits je nach Rechnertyp) abzuspeichern. Die Variable `vptr` belegt lediglich den Speicherplatz, der für eine Adresse benötigt wird. Erst nach Ausführung von

```
New (vptr);
```

ist Speicherplatz für ein gemäß Datentyp `Trec_typ` strukturiertes Datenobjekt reserviert, dessen Adresse in der Variablen `vptr` steht. Eine Wertzuweisung etwa an die Komponente `zahl` in `vrec` erfolgt durch

```
vrec.zahl := 1000;
```

eine Wertzuweisung an die Komponente `zahl`, die im Datenobjekt vorkommt, das durch den Wert in der Variablen `vptr` adressiert wird, erfolgt durch

```
vptr^.zahl := 1000;
```

Es wird also *auch syntaktisch* strikt zwischen Datenobjekten und Referenzen auf Datenobjekten unterschieden, und die korrekte Verwendung der Variablen wird durch den Compiler geprüft. Ein Datenobjekt mit Objekttyp, das etwa wie folgt deklariert wird, müsste unter Beibehaltung des korrekten Sprachkonzepts analog behandelt werden:

```
TYPE  Tclass_typ = CLASS
        PRIVATE
          zahl : INTEGER;
          txt  : STRING [10];
          ptr  : Prec_typ;
        PUBLIC
          feld : INTEGER;
          CONSTRUCTOR Create;
             ...
      END;

VAR vclass : Tclass_typ;
```

Die Variable `vclass` müsste demnach ein Objekt vom Typ `Tclass_typ` benennen, insbesondere müsste für `vclass` so viel Speicherplatz alloziert werden, wie für die Aufnahme eines Objekts vom Typ `Tclass_typ` erforderlich ist. Stattdessen wird jedoch für `vclass` Speicherplatz für die *Adresse* eines Objekts vom Typ `Tclass_typ` bereitgestellt. Der Zugriff auf die Komponente `feld` erfolgt (nach Initialisierung durch den Konstruktor `Create` und unter Akzeptanz, dass `vclass` lediglich die Adresse eines Datenobjekts enthält) nicht durch

```
vclass^.feld := 1000;
```

sondern durch

```
vclass.feld := 1000;
```

Dieses Vermischen von Datenobjekten und Referenzen auf Datenobjekte und das Verbergen der „Natur“ einer Variablen, nämlich als Behälter für ein Datenobjekt bzw. für die Adresse

eines Datenobjekts, vor dem Anwender, findet in Object Pascal für alle durch eine CLASS-Deklaration vereinbarten Datenobjekte statt. Hier wird also das ansonsten stringente Typkonzept zugunsten einer vermeindlichen Anwendungsvereinfachung der Syntax und Semantik der Sprache durchbrochen. Um diesen Missstand zu vermeiden, trotzdem aber weiterhin die Vorteile der objektorientierten Programmierung wie Vererbung, Polymorphie usw. in Verbindung mit einer einfach zu verstehenden und streng typisierten Sprache wie Pascal nutzen zu können, wird in den nachfolgenden Beispielen und Beschreibungen, in denen Pascal als Pseudocode eingesetzt wird, das ältere Objektmodell verwendet, das in Object Pascal aus Gründen der Abwärtskompatibilität zu älteren Pascal-Versionen weiterhin unterstützt wird.

Die Deklaration eines Objekttyps erfolgt durch das Sprachkonstrukt **OBJECT**. Es entspricht dem Sprachkonstrukt CLASS, schränkt aber dessen Möglichkeiten ein: Es gibt lediglich statische, virtuelle und dynamische Methoden, d.h. keine abstrakten Methoden, und die Sichtbarkeitsregeln sind auf PRIVATE und PUBLIC beschränkt. Ansonsten ist das Vererbungskonzept gleich, d.h. alle Eigenschaften einer Klasse können an andere Klassen vererbt werden. Dazu wird der Bezeichner des Objekttyps, der die vererbende Klasse beschreibt, in Klammern an das Schlüsselwort OBJECT des Objekttyps angehängt, der die erbende Klasse charakterisiert.

Die syntaktische Form einer Objekttypdeklaration lautet:

```
TYPE objekttyp_bezeichner = OBJECT (bezeichner_des_Vorfahren)
                              < Komponenten des Objekttyps>;
                              < Methodenköpfe des Objekttyps >
                            END;
```

Ist keine vererbende Vorfahrklasse angegeben, so hat die neu deklarierte Objekttyp keinen Vorfahren (anders als im CLASS-Objektmodell, in dem in diesem Fall als Default-Vorfahre der Urahn-Objekttyp TObject eingesetzt wird, der bereits grundlegende Methoden zur Verfügung stellt, die in allen Klassen benötigt werden). Außerdem enthält ein Objekttyp in diesem Modell keine als Eigenschaften (properties) bezeichneten Komponenten.

Da alle Eigenschaften der Datenobjekte einer Klasse durch den zugehörigen Objekttyp festgelegt sind, kann man den Begriff der Vererbung auf die beschreibenden Objekttypen beziehen. Ein Objekttyp (genauer: die durch den Objekttyp beschriebene Objektklasse) erbt die Eigenschaft eines übergeordneten Objekttyps (genauer: der durch den Objekttyp beschriebenen übergeordneten Klasse). Die Hierarchie der Klassen spiegelt sich in der Hierarchie der die Klassen charakterisierenden Objekttypen wieder. *Im folgenden werden daher die Begriffe Objektklasse und Objekttyp synonym verwendet.*

Die Methoden in einer OBJECT-Deklaration werden lediglich durch ihren Prozedurkopf (Methodenkopf) benannt. Sie sind implizit FORWARD-Deklarationen; das bedeutet, dass die komplette Deklaration des Methodenrumpfs erst später innerhalb des Moduls erfolgt. Dabei wird

dann zur Unterscheidung gleichlautender Methodenbezeichner der Bezeichner des Objekttyps dem Methodenbezeichner vorangestellt und durch einen Punkt von ihm getrennt.

In diesem Objektmodell werden Objekte und Referenzen auf Objekte streng unterschieden, wie folgendes Beispiel zeigt:

```
TYPE  Pobject_typ = ^Tobject_typ;
      Tobject_typ = OBJECT
        PRIVATE
          zahl : INTEGER;
          txt  : STRING [10];
          ptr  : Prec_typ;
        PUBLIC
          feld : INTEGER;
          CONSTRUCTOR Create;
             ...
      END;

VAR vobject : Tobject_typ;
    pobject : Pobject_typ;
```

Die Variable `vobject` benennt ein Objekt vom Typ `Tobject_typ` und belegt so viel Speicherplatz, wie zur Aufnahme dieses Objektes erforderlich ist. Die Variable `pobject` belegt so viel Speicherplatz, wie für die Aufnahme der Adresse eines Objekts vom Typ `Tobject_typ` benötigt wird. Der Zugriff auf die Komponente `feld` der Variablen `vobject` erfolgt (nach Initialisierung durch den Konstruktor `Create`) durch

```
vobjekt.feld := 1000;
```

der Zugriff auf die entsprechende Komponente in dem durch die Variable `pobject` adressierten Objekt erfolgt (nach Initialisierung durch den Konstruktor `Create`) durch

```
pobject^.feld := 1000;
```

Wird Object Pascal als Implementierungssprache eines eventuell umfangreichen Projekts verwendet und nicht nur, wie in diesem Skript, als leicht lesbare Pseudocode-Sprache, empfiehlt es sich, anstelle des durch `OBJECT` definierten Objektmodells das anfangs beschriebene `CLASS`-Objektmodell einzusetzen. Auf diese Weise leitet sich jeder neu deklarierte Objekttyp direkt oder indirekt vom Objekttyp `TObject` ab und verfügt somit über integrierte Konstruktoren, Destruktoren oder andere grundlegende Methoden. Des weiteren sind als Vorteil die umfangreicheren Sichtbarkeitsregeln des `CLASS`-Objektmodells und die auf dem `CLASS`-Objektmodell aufbauenden neuesten Versionen der Entwicklungsumgebung Delphi zu nennen. Der konzeptuelle Nachteil der nicht konsequenten Einhaltung von einfachen und pointerbasierten Typen muss dann eben in Kauf genommen werden.

Weitere Details der objektorientierten Programmierung mit Pascal werden im Zusammenhang der Anwendungen in den folgenden Kapiteln behandelt.

2.2.6 Anwendungsorientierte Datentypen

Die aus den Grunddatentypen in einer Programmiersprache abgeleiteten Datentypen wie Felder (`ARRAY`), Zeichenketten (`STRING`), `RECORD`s oder Objektklassen (`CLASS` bzw. `OBJECT`) bilden die Basis zur Definition komplexerer Datentypen, die noch weiter auf anwendungsorientierte Belange eingehen. Derartige **anwendungsorientierte Datentypen**, deren Implementierung eventuell weiterführende Methoden erfordert, werden häufig in Form von Programmbibliotheken oder im Rahmen von Programmentwicklungsumgebungen bereitgestellt. Grundlage der technischen Realisierung bildet dabei die objektorientierte Programmiermethodik, d.h. der Einsatz von Objektklassentypen. Die internen Details der Implementierung bleiben dem Anwender dabei komplett verborgen. Die Funktionalität wird in Form einer definierten Schnittstelle geliefert.

In den Kapiteln 5 und 6 werden derartiger Datentypen und wichtige darauf basierende Algorithmen behandelt.

Fortgeschrittene Entwicklungsumgebungen wie Delphi ([D/K]) stellen heute den Anwendungen wiederverwendbare komplexe Komponenten zur Verfügung, die auf einfache Weise in die Anwendungen eingebaut werden können. Ein typisches Beispiel ist der Einsatz eines Dialogs zur Auswahl eines Verzeichnisses im Dateisystem eines Rechners.

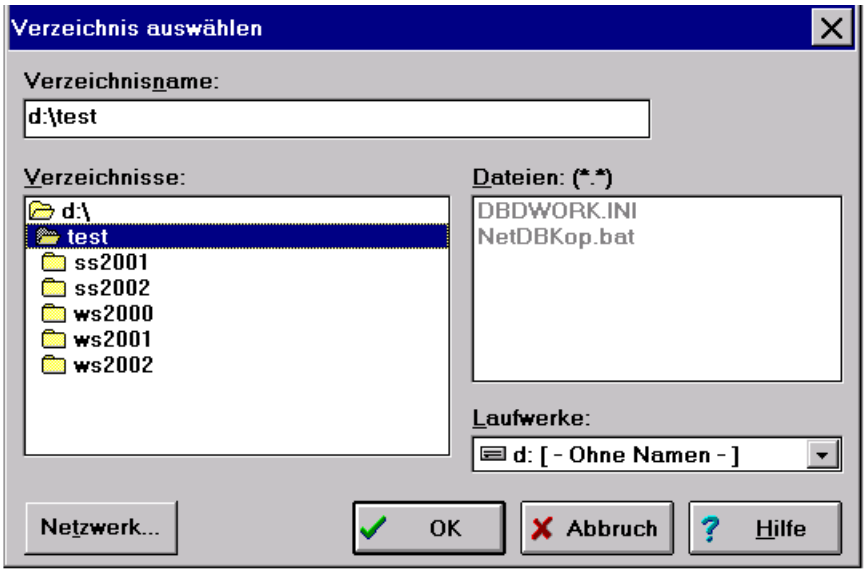


Abbildung 2.2.6-1: Beispiel eines vordefinierten Dialogelements

Derartige „Bildschirmmasken“ mit zugehöriger Dialogsteuerung und der dahinterliegenden Implementierung der angebotenen Funktionalität werden heute vordefinierten **Komponentenbibliotheken** entnommen (und müssen in keiner Weise selbst entwickelt werden). Prinzipiell verbirgt sich hinter einer derartigen Dialogkomponente ein komplexer Objekttyp, dessen Einsatz den Anwender von der Notwendigkeit befreit, sich mit den technischen Details der Realisierung abzumühen.

3 Programme

Im vorliegenden Kapitel werden Aspekte der Programmierung, d.h. der Realisierung eines dynamischen Ablaufs unter Einsatz einer Programmiersprache, beschrieben. Dazu werden in den folgenden Unterkapiteln einige wesentliche Aspekte einer Programmiersprache aus Anwendersicht wie das Prozedurkonzept und die Umsetzung von Vererbungsmethoden in der Objektorientierten Programmierung zusammengetragen.

Die Konzepte werden wieder im wesentlichen an der Sprache Pascal mit den dort definierten syntaktischen Bezeichnungen erläutert, sind aber leicht auf andere Sprachen übertragbar.

3.1 Programmstrukturen auf Anwenderebene

Sprachen wie Pascal, C, C++, ADA, Java usw. kann man als **blockstrukturierte Programmiersprachen** bezeichnen. Das bedeutet, dass ein Programm in syntaktische Einheiten, nämlich **Blöcke**, eingeteilt ist, die aus jeweils einen **Deklarationsteil** und einen **Anweisungsteil (Befehlsteil)** bestehen (Abbildung 3.1-1).

Ein Block, der ein komplettes Programm darstellt, unterscheidet sich von einem Block, der eine Prozedur (ein Unterprogramm) realisiert, durch das syntaktische Format der Anweisung, die den Block benennt: Bei einem Programm wird die Benennung durch eine Anweisung mit Schlüsselwort `PROGRAM` durchgeführt, bei einer Prozedur wird das Schlüsselwort `PROCEDURE` bzw. `FUNCTION` verwendet. In Kapitel 3.1.3 kommt dann noch das syntaktische Konstrukt einer Unit hinzu, die ebenfalls als Block aufgefasst werden kann.

Im **Deklarationsteil** werden Bezeichner festgelegt für

- Konstanten, die bei der Übersetzung des Programms direkt in Code übersetzt werden (**Konstantendeklarationsteil** beginnend mit dem Schlüsselwort `CONST`)
- Datentypen für selbstdefinierte Datentypen (**Typdeklarationsteil** beginnend mit dem Schlüsselwort `TYPE`)
- Datenobjekte (**Variablendeklarationsteil** beginnend mit dem Schlüsselwort `VAR`; entsprechend wird ein Datenobjekt auch Variable genannt)
- Prozeduren und Funktionen (**Prozedurdeklarationsteil**).

Aus historischen Gründen (veraltete Programmiertechnik) ist die Definition von Sprungzielen möglich, an die der Programmablauf mit Hilfe einer Sprunganweisung (`GOTO`) verzweigen

kann (**Labeldeklarationsteil** beginnend mit dem Schlüsselwort `LABEL`). Derartige Programmsprünge sind überflüssig und im Rahmen einer strukturierten Programmiermethodik nicht erlaubt.

Nicht alle Teile des Deklarationsteils müssen vorhanden sein. Sie können in beliebiger Reihenfolge und Wiederholung auftreten. Das Ende eines Teils wird durch das erneute Auftreten eines Schlüsselworts (`CONST`, `TYPE`, `VAR`, `PROCEDURE`, `FUNCTION`) erkannt. Bevor in einer Deklaration ein selbstdefinierter Bezeichner verwendet werden kann, muss er (bis auf gewisse Ausnahmen) deklariert worden sein.

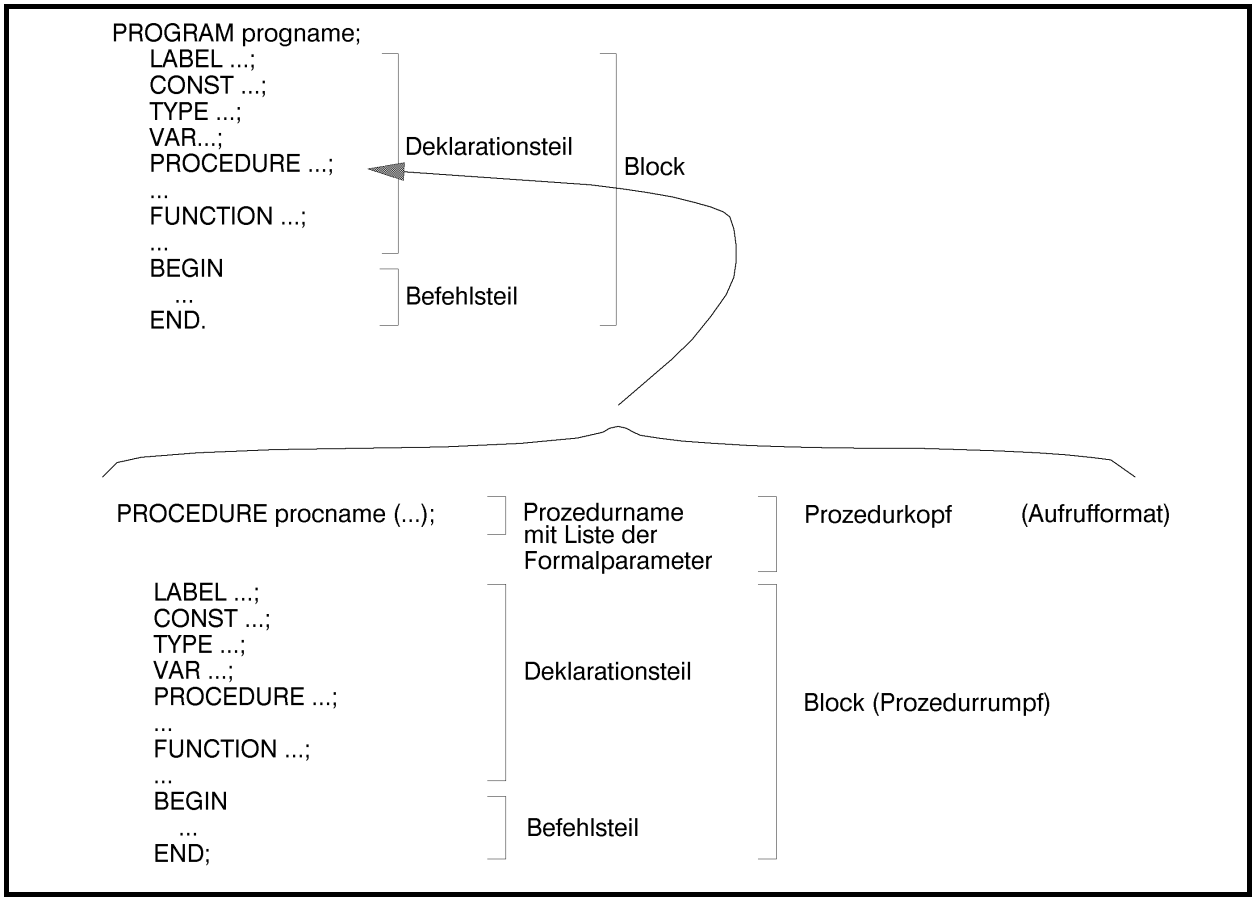


Abbildung 3.1-1: Block

Im **Befehlsteil** stehen die zu dem Block gehörenden Anweisungen. Der Befehlsteil ist in die Schlüsselwörter `BEGIN` und `END` eingeschlossen.

Eine Prozedurdeklaration kann selbst wieder Prozedurdeklarationen enthalten, die aus Blöcken bestehen, in denen weitere Prozedurdeklarationen vorkommen können usw. Blöcke können also **ineinandergeschachtelt** sein, aber auch nebeneinander liegen. Bei ineinandergeschachtelten Blöcken kann man dann von **äußeren** und **inneren Blöcken** sprechen: aus Sicht eines Blocks sind andere Blöcke, die in ihm definiert werden, und alle darin enthaltenen (eingebetteten) Blöcke innere Blöcke; die Blöcke, die einen definierten Block umfassen, sind aus der Sicht dieses Blocks äußere Blöcke. Ein aus Sicht eines Blocks äußerer Block heißt auch **übergeordneter Block**; entsprechend heißt ein innerer Block auch **untergeordneter Block**.

Für die Bildung von Bezeichnern gelten einige Regeln: So können sie beliebig lang sein; meist werden jedoch nur die ersten 255 Zeichen eines Bezeichners berücksichtigt. Ein Bezeichner beginnt mit einem Buchstaben (auch das Zeichen _ gilt als Buchstabe) und kann dann auch Ziffern enthalten. Groß- und Kleinschreibung wird (in Pascal) nicht unterschieden. Schlüsselworte der Programmiersprache sind als selbstdefinierte Bezeichner nicht zugelassen. ***Innerhalb eines Blocks müssen Bezeichner eindeutig sein.*** Die Regeln über den Gültigkeitsbereich von Bezeichnern (Kapitel 3.1.1) beschreiben das Verhalten bei Verwendung gleichlautender Bezeichner in ineinandergeschachtelten Blöcken.

3.1.1 Der Gültigkeitsbereich von Bezeichnern

Unterschiedliche Datenobjekte, untergeordnete Blöcke oder Datentypen in unterschiedlichen Blöcken können mit demselben Bezeichner versehen werden. Gleichlautende Bezeichner für verschiedene Datenobjekte, untergeordnete Blöcke und Typen im selben Block sind nicht zulässig.

In folgendem Beispiel (Abbildung 3.1.1-1) werden in den Variablendeklarationsteilen Datenobjekte mit zum Teil gleichlautenden Bezeichnern (und jeweiligem Datentyp `INTEGER`) deklariert. Zur Unterscheidung der Datenobjekte erhalten sie in der Abbildung die Bezeichnungen d1, ..., d11. Die *Bezeichner* der einzelnen Datenobjekte lauten:

in `p_program`: `v` für das Datenobjekt d1,
`w` für das Datenobjekt d2,

in `a_proc`: `a` für das Datenobjekt d3,
`b` für das Datenobjekt d4,

in `b_proc`: `a` für das Datenobjekt d5,
`i` für das Datenobjekt d6,
`w` für das Datenobjekt d7,

in `c_proc`: `a` für das Datenobjekt d8,
`v` für das Datenobjekt d9,

in `d_proc`: `v` für das Datenobjekt d10,
`w` für das Datenobjekt d11.

```
PROGRAM p_program;                                   Gültigkeitsbereiche
    { Variablendeklarationen von p_proc: }           der Bezeichner v
  VAR v : INTEGER       { Datenobjekt d1 };
      w : INTEGER       { Datenobjekt d2 };
    { Prozedurdeklarationen von p_proc: }
  PROCEDURE a_proc;
      { Variablendeklarationen von a_proc: }
    VAR a : INTEGER       { Datenobjekt d3 };
        b : INTEGER       { Datenobjekt d4 };
      { Prozedurdeklarationen von a_proc: }
    PROCEDURE b_proc;
        { Variablendeklarationen von b_proc: }
      VAR a : INTEGER       { Datenobjekt d5 };
          i : INTEGER       { Datenobjekt d6 };
          w : INTEGER       { Datenobjekt d7 };
        { Befehlsteil von b_proc: }
      BEGIN
        ...
        a := i + w + v + b;
        ...
      END { b_proc };
    PROCEDURE c_proc;
        { Variablendeklarationen von c_proc: }
      VAR a : INTEGER       { Datenobjekt d8 };
          v : INTEGER       { Datenobjekt d9 };
        { Befehlsteil von c_proc: }
      BEGIN
        ...
      END { c_proc };
      { Befehlsteil von a_proc: }
    BEGIN
      ...
      v := w;
      ...
      { Aufruf von } b_proc;
      ...
    END { a_proc };
  PROCEDURE d_proc;
      { Variablendeklarationen von d_proc: }
    VAR v : INTEGER       { Datenobjekt d10 };
        w : INTEGER       { Datenobjekt d11 };
      { Befehlsteil von d_proc: }
    BEGIN
      ...
      v := w;
      { Aufruf von } a_proc;
      ...
    END { d_proc };

    { Befehlsteil von p_program: }
  BEGIN
    ...
    { Aufruf von } d_proc;
    ...
  END { p_program }.
```

Abbildung 3.1.1-1: Gültigkeitsbereich von Bezeichnern (Beispiel)

Als generelle Regel für Bezeichner gilt:

Der **Gültigkeitsbereich eines Bezeichners** liegt zwischen dem Ort der Deklaration und dem Ende des die Deklaration enthaltenden Blocks, wobei alle Blöcke in den Gültigkeitsbereich mit eingeschlossen sind, die dieser Block umfasst. Allerdings gibt es von dieser Regel eine wesentliche Ausnahme: Wird ein Bezeichner in einem Block B definiert, so schließt diese Definition den Gültigkeitsbereich eines gleichlautenden Bezeichners in einem äußeren Block A für diesen Block B und seine eingebetteten Blöcke aus.

Anders gesagt: Wenn in einem Block A ein Bezeichner definiert wird und in einem inneren Block B eine weitere Definition mit demselben Bezeichner auftritt, so wird durch diese Neudefinition der innere Block B und alle darin eingebetteten Blöcke aus dem Gültigkeitsbereich des Bezeichners des äußeren Blocks A ausgeschlossen.

Bezeichner müssen deklariert sein, bevor sie benutzt werden können. Sie können in einem Block nur jeweils einmal deklariert werden, es sei denn, ihre Neudeklaration erfolgt innerhalb eines untergeordneten Blocks. Bezeichner sind von außerhalb des definierenden Blocks nicht „sichtbar“. ***Ein Block kapselt die Deklaration zu den äußeren Blöcken ab, so dass man einen in einem Block B definierten Bezeichner bezüglich eines diesen Block B umfassenden (äußeren) Blocks A als lokal im Block B ansehen kann.***

Ein lokaler Bezeichner eines Datenobjekts benennt eine **lokale Variable (lokales Datenobjekt)**. Bezüglich eines im Block B eingebetteten (inneren) Blocks C ist die Deklaration eines Bezeichners **global**. Ein globaler Bezeichner eines Datenobjekts benennt eine **globale Variable**. Die obige Ausnahmeregel besagt: ***Lokale Deklarationen überlagern globale Deklarationen.***

Für den Bezeichner einer Prozedur gilt in Übereinstimmung mit den obigen Regeln, dass er zum Prozedurrumpf selbst global und zu dem Block, der die Prozedurdefinition enthält, lokal ist. Der Gültigkeitsbereich eines Prozedurbezeichners umfasst also den Block, der die Prozedurdefinition enthält (als lokale Definition), den Prozedurrumpf und alle eingebetteten Blöcke (als globale Definition), wobei obige Ausnahmeregel zu beachten ist. Als Konsequenz ergibt sich beispielsweise, dass sich eine Prozedur in ihrem Prozedurrumpf selbst aufrufen kann, da der Prozedurname im Prozedurrumpf global bekannt ist.

Der Begriff des Gültigkeitsbereichs eines Bezeichners wird an den Bezeichnern der Datenobjekte (Variablen) in obigem Beispiel verdeutlicht:

Der Gültigkeitsbereich des Bezeichners `v` umfasst die Blöcke `p_program` als lokale Deklaration und die Blöcke `a_proc` und `b_proc` als globale Deklaration, da `v` in `p_program` definiert wird und `a_proc` und `b_proc` in `p_program` eingebettet sind und keine Neudefinition von `v` enthalten. Mit dem Bezeichner `v` wird also in `p_program`, `a_proc` und `b_proc` dasselbe Datenobjekt d1 angesprochen. Der Gültigkeitsbereich des Bezeichners `v` für das Daten-

objekt d1 aus der Definition in `p_program` umfasst aber nicht die Blöcke `c_proc` und `d_proc`, da hier jeweils Neudefinitionen des Bezeichners `v` vorkommen und damit jeweils unterschiedliche Datenobjekte (d9 bzw. d10) bezeichnen.

Wichtig ist, die *Reihenfolge* der Deklarationen in einem Block zu beachten: Im Beispiel der Abbildung 3.1.1-1 liegen die Blöcke `a_proc` und `d_proc` nebeneinander auf hierarchisch gleicher Stufe. Da die Deklaration des Bezeichners `a_proc` *vor* der Deklaration des Bezeichners `d_proc` erfolgt, liegt der Prozedurrumpf der Prozedur `d_proc` als zum Block `p_program` untergeordneter Block im Gültigkeitsbereich des Bezeichners `a_proc` (innerhalb des Prozedurrumpfs von `d_proc` ist der Bezeichner `a_proc` global). Das bedeutet, dass im Prozedurrumpf der Prozedur `d_proc` der Bezeichner `a_proc` „sichtbar“ ist und die Prozedur `a_proc` aufgerufen werden kann. Jedoch kann innerhalb der Prozedur `a_proc` kein Aufruf der Prozedur `d_proc` erfolgen, da der Prozedurrumpf von `a_proc` nicht im Gültigkeitsbereich des Bezeichners `d_proc` liegt. Eine analoge Situation herrscht aufgrund der Reihenfolge, in der die Bezeichner definiert werden, im Verhältnis von `b_proc` und `c_proc` zueinander: In `c_proc` könnte ein Aufruf von `b_proc` erfolgen, aber nicht umgekehrt.

Die Prozedur `b_proc` könnte die Prozedur `a_proc` aufrufen, da der Bezeichner `a_proc` global bezüglich `b_proc` ist (der Block `b_proc` ist im Block `a_proc` eingebettet). Ein Aufruf der Prozedur `d_proc` im Prozedurrumpf von `b_proc` ist aber nicht möglich, da der Gültigkeitsbereich des Bezeichners `d_proc` den Block `d_proc` (und alle eingebetteten Blöcke) umfasst und der Block `b_proc` nicht in `d_proc` enthalten ist.

Abbildung 3.1.1-2 fasst die Gültigkeitsbereiche der Bezeichner für Datenobjekte und die Zugriffsmöglichkeiten aus Abbildung 3.1.1-1 aus Sicht des jeweiligen Blocks (in Klammern stehen die mit dem jeweiligen Bezeichner angesprochenen Datenobjekte) zusammen. Zusätzlich sind die Gültigkeitsbereiche der Bezeichner der Prozeduren aufgeführt.

Man sieht also, dass der Bezeichner `a` in der Prozedur `b_proc` das lokale Datenobjekt d5 (und nicht etwa das Datenobjekt d3 wie in `a_proc`) bezeichnet. Entsprechend bezeichnet der Bezeichner `w` in `b_proc` das lokale Datenobjekt d7. Auf das Datenobjekt d2 ist aus `b_proc` heraus nicht zugreifbar, da der Bezeichner `w` lokal für d7 verwendet wird.

Block	sichtbare lokale Bezeichner	sichtbare globale Bezeichner
p_program	v (d1), w (d2); a_proc, d_proc	p_program
a_proc	a (d3), b (d4); b_proc, c_proc	v (d1), w (d2); a_proc, p_program
b_proc	a (d5), i (d6), w (d7)	b (d4), v (d1); b_proc, a_proc, p_program
c_proc	a (d8), v (d9)	b (d4), w (d2); c_proc, b_proc, a_proc, p_program
d_proc	v (d10), w (d11)	d_proc, a_proc, p_program

Block	Bezeichner	Gültigkeitsbereich
p_program	v (d1)	p_program, a_proc, b_proc
	w (d2)	p_program, a_proc, c_proc
	a_proc	p_program, a_proc, b_proc, c_proc, d_proc
	d_proc	p_program, d_proc
a_proc	a (d3)	a_proc
	b (d4)	a_proc, b_proc, c_proc
	b_proc	a_proc, b_proc, c_proc
	c_proc	a_proc, c_proc
b_proc	a (d5)	b_proc
	i (d6)	b_proc
	w (d7)	b_proc
c_proc	a (d8)	c_proc
	v (d9)	c_proc
d_proc	v (d10)	d_proc
	w (d11)	d_proc

Abbildung 3.1.1-2: Gültigkeitsbereich von Bezeichnern (Beispiel, Forts. von Abbildung 3.1.1-1)

3.1.2 Bemerkungen zur Lebensdauer von Datenobjekten

Ein wichtiges Charakteristikum eines Datenobjekts in blockorientierten Sprachen ist seine **Lebensdauer**. Während der Gültigkeitsbereich eines Datenobjektbezeichners eine statische Eigenschaft eines syntaktischen Konstrukts der Programmiersprache, nämlich eines Bezeichners, ist, beschreibt der Begriff der Lebensdauer eines Datenobjekts in einem Programm eine typische dynamische Eigenschaft.

Die Lebensdauer eines Datenobjekts beginnt, wenn diesem Datenobjekt Speicherplatz zugewiesen wird und endet, wenn dieser zugewiesene Speicherplatz zur anderweitigen Verwendung wieder freigegeben wird.

In einem Programm werden den Datenobjekten des Hauptprogramms (äußerster Programmblock) bei Programmstart Speicherplatz zugeordnet. Den Datenobjekten der Prozeduren und Funktionen wird erst im Augenblick des Prozedur- bzw. Funktionsaufrufs Speicherplatz zugeordnet. Es gilt daher für die Lebensdauer eines Datenobjekts:

Die **Lebensdauer eines Datenobjekts beginnt** zu dem Zeitpunkt, zu dem die erste Anweisungen (des Objektcodes) des Blocks ausgeführt wird, in dem das Datenobjekt deklariert ist. Sie **endet** mit dem Abschluss der letzten Anweisung (des Objektcodes) des Blocks, in dem das Datenobjekt deklariert ist. Das bedeutet beispielsweise, dass Datenobjekte, die in einer Prozedur definiert sind, die während eines Programmlaufs jedoch nicht aufgerufen wird, auch nicht „zu leben“ beginnen, d.h. dass ihnen kein Speicherplatz zugewiesen wird.

Die Lebensdauer der einzelnen Datenobjekte in Abbildung 3.1.1-1 während des Programmablaufs zeigt Abbildung 3.1.2-1. Der linke Teil gibt von oben nach unten der Linie folgend den zeitlichen Ablauf wieder; ein Knick nach rechts symbolisiert einen Unterprogrammsprung, ein Knick nach links eine Rückkehr in das rufende Programm. Im rechten Teil ist die Lebensdauer eines jeden Datenobjekts bei diesem Programmablauf durch eine senkrechte Linie dargestellt, deren Abschlusspunkte jeweils den Beginn bzw. das Ende der Lebensdauer markieren.

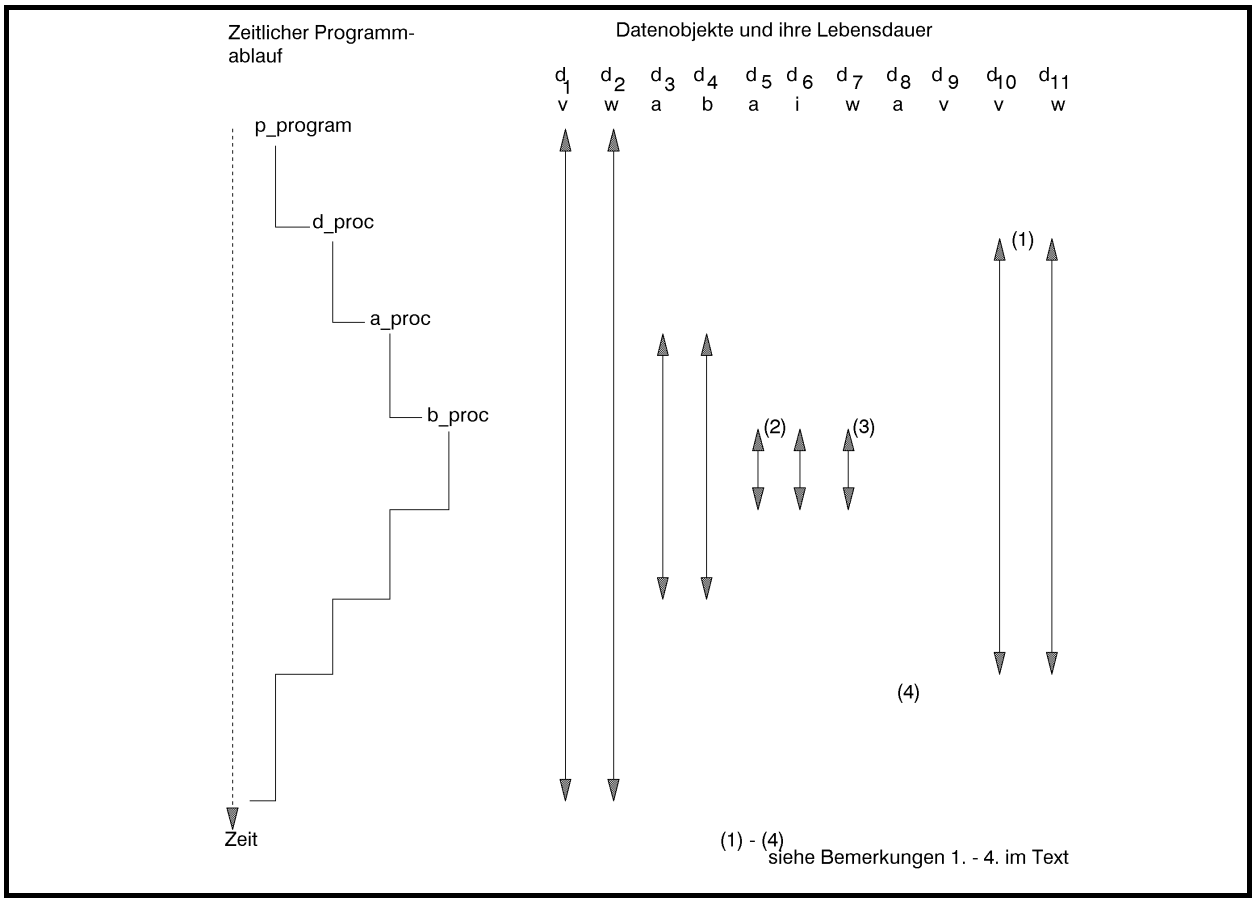


Abbildung 3.1.2-1: Lebensdauer von Datenobjekten (Beispiel)

Einige Bemerkungen erläutern die Zusammenhänge:

1. Der Beginn der Lebensdauer des Datenobjekts d10 mit Bezeichner `v` beendet nicht die Lebensdauer des gleichnamigen Datenobjekts d1, obwohl ein Zugriff mittels des Bezeichners `v` auf d1 in diesem Augenblick nicht möglich ist; die gleiche Aussage gilt für die Datenobjekte mit Bezeichner `w` (d2 und d11). Die Datenobjekte d1 und d2 sind in diesem Augenblick nicht sichtbar
2. Die Lebensdauer des Datenobjekts d3 ist nicht beendet, obwohl auf d3 mit dem Bezeichner `a` nicht zugegriffen werden kann
3. Mit dem Beginn der Lebensdauer kann über den Bezeichner `w` zur Zeit weder auf das Datenobjekt d2 noch auf das Datenobjekt d11 zugegriffen werden; ihre Lebensdauer ist aber noch nicht beendet
4. Die Datenobjekte d8 und d9 beginnen bei diesem Programmablauf nicht zu leben, da keine Anweisungen im sie definierenden Block `c_proc` ausgeführt werden.

3.1.3 Modularisierungskonzepte

Das Prozedurkonzept einer höheren Programmiersprache unterstützt die Aufteilung einer (komplexen) Programmieraufgabe in einzelne Teilaufgaben. Die Einbettung einzelner Prozeduren in andere Prozeduren ermöglicht eine Kapselung rein lokaler Teilaufgaben nach außen und eine strenge Top-Down-Modularisierung einer größeren Programmieraufgabe. Im bisher beschriebenen Konzept sind alle Prozeduren interne Unterprogramme, da sie innerhalb eines Hauptprogramms deklariert, d.h. im Quellcode des Hauptprogramms angegeben werden. Die Anforderungen, die Software-Engineering-Konzepte an eine Programmiersprache stellen (Top-down-Design, objektorientiertes Vorgehen mit Vererbung und Kapselung von Datenobjekten, Wiederverwendbarkeit von Code, Programmiersprachenunabhängigkeit von Modulen auf Objektcode-Ebene, automatische Versionskontrolle usw.), verlangen jedoch weitgehende Modularisierungsmöglichkeiten. Dazu gehören:

- **Automatisches Einfügen von Quellcode:** Quellcodeteile, die in mehreren Programmieraufgaben identisch vorkommen, müssen nur einmal erstellt werden und können dann, nachdem sie in einer eigenen Datei abgelegt wurden, ***vor der Übersetzung automatisch in Quellcodes verschiedener Programme hineinkopiert werden***
- **Aufruf externer Prozeduren:** Gelegentlich ist es notwendig, Programmteile (Prozeduren) aufzurufen, die spezielle Hardware-Eigenschaften der Maschine berücksichtigen und nur mit Maschinensprachbefehlen realisierbar sind
- **Einbindung von Laufzeitbibliotheken:** Eine Reihe von Prozeduren/Funktionen, die in vielen Programmen vorkommen, deren Implementierung häufig aber Rechner-abhängig ist, wird in Form einer Laufzeitbibliothek bereitgestellt. Die Laufzeitbibliothek ist Teil des Entwicklungssystems der Programmiersprache. Im Quelltext eines Programms kön-

nen die Prozeduren der Laufzeitbibliothek verwendet werden, ohne sie explizit zu deklarieren. Die Handbücher der Programmiersprache geben über Bezeichner, Aufrufformate, Bedeutung der Parameter und Randbedingungen Auskunft. Bei der Übersetzung wird der benötigte Code automatisch angebunden. Programmiersprachen wie C oder C++ haben standardisierte Laufzeitbibliotheken, die auf allen Rechnersystemen verfügbar sein müssen; Object Pascal hat auf die „INTEL-Welt“ abgestimmte Laufzeitbibliotheken für die verschiedenen Operationsmodi (Protected Mode, Flat Mode). Die Inhalte der Laufzeitbibliothek realisieren zum einen standardisierte Sprachelemente, z.B. mathematische Funktionen und Ein/Ausgabefunktionen, zum anderen stellen sie funktionale Spracherweiterungen, z.B. Schnittstellen zur Bildschirmsteuerung und eine Reihe „nützlicher“ Rechnerspezifischer Konstanten und Variablen bereit, die auf speziellen Betriebssystem- oder Rechnerfunktionen aufbauen. In Pascal ist die Laufzeitbibliothek (jedes Operationsmodus) technisch in Form einer Reihe von Units implementiert, die unten beschrieben werden

- **Verwendung dynamischer Bindebibliotheken:** Externe Referenzen erst zur Laufzeit (dynamisch) aufgelöst und die durch externe Referenzen angesprochenen Objektmoduln ***nach Bedarf*** angebunden. Dabei wird angegeben, wo die einzelnen Objektmoduln zu finden sind, und zwar durch Angabe entsprechender Bibliotheken (**dynamische Bindebibliothek, DLL, dynamic link library**). Wird auf eine externe Referenz Bezug genommen, so wird das entsprechende Objektmodul in der Bibliothek gesucht und angebunden (Auflösung externer Adressreferenzen). Der dynamische Bindevorgang kann entweder im Augenblick des Programmladens ablaufen (**dynamisches Binden zur Ladezeit, load-time dynamic linking**) oder erst zur Laufzeit, wenn auf die anzubindende externe Refererenz auch wirklich Bezug genommen wird (**dynamisches Binden zur Laufzeit, run-time dynamic linking**). Mit dem dynamischen Binden sind einige ***Vorteile*** verbunden: Ein Objektmodul wird erst dann angebunden, wenn es während der Laufzeit auch wirklich angesprochen wird; das ablauffähige Programm als gebundenes Programm ist vom Umfang her überschaubarer. Das System kann dynamisch rekonfiguriert bzw. erweitert werden, ohne dass das komplette System neu übersetzt oder gebunden zu werden braucht; es wird nur der veränderte Teil in der DLL ersetzt bzw. eine neue DLL angebunden; dieser Punkt ist besonders wichtig bei Änderung des Rechners und zur Garantie der Portierbarkeit von Softwaresystemen.

Als Beispiel eines sehr komfortablen Hilfsmittels zur Modularisierung kann das **Unit-Konzept in Pascal** angeführt werden.

Die Aufteilung eines Programms in getrennt übersetzbare Programmeinheiten (Moduln) und das anschließende Binden zu einem ablauffähigen Programm bietet eine Reihe offensichtlicher ***Vorteile***. Diese werden insbesondere dann sichtbar, wenn einige Programmteile in unterschiedlichen Programmiersprachen formuliert werden *müssen*, weil beispielsweise Anschlüsse an Betriebssystemdienste eine Maschinensprachenprogrammierung erfordern oder in einem Unterprogramm spezielle Hardwareeinrichtungen angesteuert werden sollen. Bei der Aufteilung einer größeren Programmieraufgabe in Teilaufgaben bietet es sich an, jede dieser Teil-

aufgaben in eine für die Teilaufgabe spezifische Menge getrennt übersetzter Moduln zu legen. Alle Moduln sind dann meist in derselben Programmiersprache geschrieben.

Ein ***Nachteil*** des Einsatzes getrennt übersetzter Moduln besteht (neben der eventuellen Komplexität des gegenseitigen Programmanschlusses) darin, dass es dem Compiler nicht möglich ist, die Datentypen der aktuellen Parameter in einem Aufruf eines getrennt übersetzten Unterprogramms daraufhin zu überprüfen, ob sie mit den Datentypen der korrespondierenden Formalparameter übereinstimmen. An dieser Stelle ist strenge Programmierdisziplin vom Anwender gefordert; sonst kann es zu Laufzeitfehlern oder logischen Programmfehlern kommen.

Um die Vorteile der Datentypüberprüfung mit den Vorteilen getrennt übersetzter Programmteile auf Sprachebene zu verbinden, sind zusätzliche Konzepte erforderlich. In Pascal (eingeführt in Turbo Pascal) ist dazu das **Unit-Konzept** realisiert. Dieses Sprachkonzept genügt zusätzlich der Forderung des Information Hidings, des Verbergens implementierungstechnischer Details einzelner externer Unterprogramme gegenüber dem Anwender.

Eine Unit besteht (auf Sprachebene) aus drei Teilen:

- dem **Interfaceteil** als „öffentlichen“ Teil der Unit; er enthält die Deklarationen von Datenobjekten (Datentyp- und Variablendeklarationen), Prozedur- und Funktionsköpfen, die von außen zugreifbar sind
- dem **Implementierungsteil** als „privaten“ Teil der Unit; er enthält den Programmcode zu den im Interfaceteil aufgeführten Prozedur- und Funktionsköpfen und eventuell zusätzliche nach außen nicht verfügbare Daten-, Prozedur- und Funktionsdeklarationen; auf keines der nur im Implementierungsteil deklarierten Datenobjekte und auf keine der nur hier deklarierten Prozeduren oder Funktionen kann von außen zugegriffen werden
- dem optionalen **Initialisierungsteil**, der Programmcode enthält, der beim Start des Programms, das diese Unit benutzt (siehe unten), durchlaufen wird und im wesentlichen zur Initialisierung der Datenstrukturen der Unit dient; hier sind beliebige Operationen im Rahmen des Pascal-Sprachumfangs erlaubt, die sowohl die „öffentlich“ als auch die „privat“ deklarierten Datenobjekte verwenden.

Im Interfaceteil werden die Schnittstellen (Aufrufformate) für Prozeduren und Funktionen, die die Unit bereitstellt, und „öffentliche“ Daten der Unit deklariert. Die Realisierung einer Prozedur bzw. Funktion, der Prozedur- bzw. Funktionsrumpf also, wird im Implementierungsteil nach außen vollständig verborgen.

Der **syntaktische Aufbau einer Unit** ist in Abbildung 3.1.3-1 dargestellt.

Eine Unit stellt ein eigenständiges Programm dar und führt zu einem eigenen (bezüglich anderer Programme getrennt übersetzten) Modul. Alle im Interfaceteil aufgeführten Deklarationen werden nach außen bekannt gegeben. Ein Programm kann eine Unit (mit Bezeichner `name`) als externes Programm durch Angabe von

```
USES name;
```

einbinden. Alle Bezeichner des Interfaceteils (aber nicht des Implementierungsteils) der angesprochenen Unit sind von dieser Programmstelle aus für das einbindende Programm global verfügbar und werden so behandelt, als wären sie an dieser Stelle im Programm deklariert worden (mit den entsprechenden Auswirkungen auf die Gültigkeit der Bezeichner). Insbesondere kann der Compiler prüfen, ob eine Prozedur, deren Prozedurkopf im Interfaceteil einer Unit deklariert ist und die von einem Programm, das diese Unit verwendet, aufgerufen wird, bezüglich der Datentypen der korrespondierenden Aktual- und Formalparameter richtig versorgt wird. Alle Bezeichner, die im Implementierungsteil einer Unit deklariert werden, sind wie der Code des Implementierungsteils nach außen abgekapselt (Information Hiding).

```
UNIT name;

INTERFACE

  USES <Liste weiterer von dieser Unit „öffentlich"
        verwendeter Units                          (*)>
       { optional } ;

  <„öffentliche" Deklarationen>;

IMPLEMENTATION

  USES <Liste weiterer von dieser Unit „privat"
        verwendeter Units                          (*)>
       { optional } ;

  <„private" Deklarationen                         (*)>;

  <Prozedur- und Funktionsrümpfe der im Interfaceteil
   aufgeführten Prozeduren und Funktionen          (*)>

BEGIN

  <Programmcode zur Initialisierung der
   Datenobjekte                                    (*)>
  { optional; der Initialisierungsteil umfasst alle
    Anweisungen von BEGIN (einschließlich) bis
    END (ausschließlich)                           }

END.
```

(*) Der Text in den spitzen Klammern (< ... >) beschreibt hier eine syntaktische Einheit.

Abbildung 3.1.3-1: Syntaktischer Aufbau einer Unit

Das folgende Beispiel erläutert Abhängigkeiten im Unit-Konzept:

Ein Programm mit Bezeichner `main` verwendet Prozeduren `high_proc` und `middle_proc`, die getrennt übersetzt in den Units mit Bezeichner `high` bzw. `middle` liegen. Innerhalb `UNIT high` wird ebenfalls die Prozedur `middle_proc` (aus `UNIT middle`) verwendet, die für `UNIT high` also extern ist. `UNIT middle` greift auf eine extern in `UNIT low` definierte Prozedur `low_proc` zu. Die entsprechenden Pascal-Programmteile und ihre gegenseitigen Abhängigkeiten sind in Abbildung 3.1.3-2 dargestellt. Zu beachten ist, dass der im Interfaceteil einer Unit aufgeführte Prozedurkopf einer Prozedur im Implementierungsteil wiederholt wird.

```
PROGRAM main;

  USES high, middle
             { Einbindung der „öffentlichen"
               Deklarationen der UNITs high und middle };

  VAR i: INTEGER;
      t : REAL;

BEGIN
  ...
  high_proc (i)  { in UNIT high   };
  ...
  middle_proc (t)  { in UNIT middle };
  ...
END.


UNIT high;

INTERFACE   { „öffentliche" Deklarationen }

  USES middle  { Zugriff auf die „öffentlichen"
                 Deklarationen der UNIT middle };

  PROCEDURE high_proc (p : INTEGER);
```

```
IMPLEMENTATION { „private" Deklarationen }

  PROCEDURE high_proc (p: INTEGER);

    VAR r_var : REAL { nach außen verborgen };

    BEGIN
     ...
     middle_proc (r_var);
     ...
    END  { high_proc };
  ...
END.

UNIT middle;

INTERFACE

  PROCEDURE middle_proc (r : REAL);

IMPLEMENTATION

  USES low;

  PROCEDURE privat_proc
              { „private", nach außen nicht
                 bekannte Prozedur der UNIT middle };
    BEGIN
    ...
    END  { privat_proc };

  PROCEDURE middle_proc (r: REAL);

    VAR in_char : CHAR;
    BEGIN
    ...
    low_proc (in_char);
    ...
    privat_proc  { Aufruf einer für die UNIT „privaten"
                   Prozedur innerhalb einer nach außen
                   bekannt gegebenen Prozedur           };
    ...
    END  { middle_proc };
  ...
END.
```

```
UNIT low;

INTERFACE

  PROCEDURE low_proc (VAR zeichen : CHAR);

IMPLEMENTATION

  PROCEDURE low_proc (VAR zeichen : CHAR);

    BEGIN
     ...
     zeichen := ...;
     ...
    END  { low_proc };
   ...
END.
```

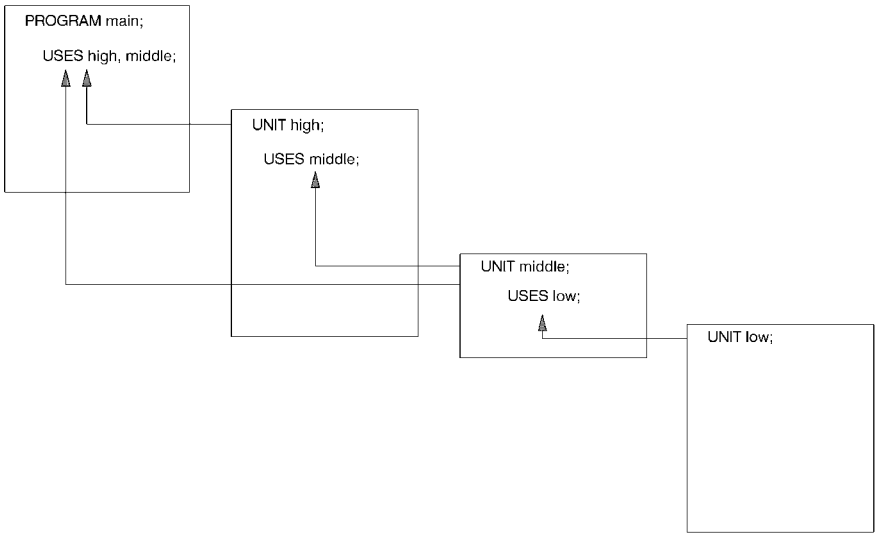


Abbildung 3.1.3-2: Units in Pascal

Bei derartigen Abhängigkeiten ist die Übersetzungsreihenfolge der einzelnen Programmteile (`PROGRAM` und `UNIT`) von Bedeutung: Verwendet ein Programmteil mit der `USES`-Anweisung andere Units, d.h. ist dieser Programmteil vom Vorhandensein der Deklarationen der so einzubindenden Units ***abhängig***, so müssen diese Units zuvor übersetzt worden sein. In obigem Beispiel lautet die Reihenfolge:

```
UNIT low,
```

```
UNIT middle,
UNIT high,
PROGRAM main.
```

Möchte man die oben beschriebenen Teile des Laufzeitsystems verwenden, muss die entsprechende Unit mit der `USES`-Anweisung einbinden; eine Ausnahme bildet `UNIT System`, die immer eingebunden ist.

Wird der Interfaceteil einer Unit geändert, so müssen alle Programmteile, die diese Unit mittels `USES`-Anweisung benutzen, neu kompiliert werden. Die integrierte Entwicklungsumgebung von Pascal unterstützt die automatische Recompilierung abhängiger Programmteile nach Quellcodeänderungen. Änderungen einer Unit im Implementierungs- oder Initialisierungsteil erfordern keine Rekompilation der abhängigen Programmteile. Nach Änderung und Rekompilation einer Unit und eventuell ihrer abhängigen Programmteile ist ein erneuter Bindevorgang notwendig. Selbstverständlich wird der Objektcode einer Unit, die in mehreren anderen Units per `USES`-Anweisung angesprochen wird, wie es im Beispiel mit `UNIT middle` geschieht, nur einmal angebunden. Außerdem bindet der Compiler nur diejenigen Teile einer Unit an, die auch verwendet werden.

3.1.4 Das Laufzeitlayout eines Programms

Nach der Übersetzung des Quellcodes eines Programms sind die Variablen je nach Typ und Stellung innerhalb des Programms eventuell in Speicherplatzreservierungen und alle Namensbezüge auf die Variablen in Adressbezüge umgesetzt worden. Für einige Variablen, z.B. die lokalen Variablen in Unterprogrammen und durch Pointer referenzierte Daten, erfolgt eine Speicherplatzreservierung erst zur Laufzeit gemäß des Lebensdauerkonzepts. Im folgenden wird exemplarisch das Laufzeit-Layout eines Pascal-Programms beschrieben[6]. Programmiersprachen wie C oder C++, allgemein sogenannte stackorientierte Programmiersprachen (siehe [P/Z]), haben ein ähnliches Laufzeit-Layout. Zur Vereinfachung wird angenommen, dass alle Programmteile statisch gebunden, d.h. dass also alle externen Referenzen durch Anbinden entsprechender Programm- und Datenteile aufgelöst wurden. Bei dynamischem Binden ergibt sich ein ähnliches Bild; allerdings sind Code- und statischer Datenbereich dann „noch nicht vollständig“.

---

6 Die Definition der Sprache Pascal macht natürlich keine Angaben über das Laufzeit-Layout. Dieses ist ein Thema der Realisierung der Sprache für einen speziellen Rechnertyp, d.h. ein Aspekt des Compilerbaus.

In einem Pascal-Programm kommen Datendeklarationen und die daraus erzeugten Datenobjekte an unterschiedlichen Positionen vor: im Hauptprogramm, in Unterprogrammen als Formalparameter und als lokale Daten und als dynamisch erzeugte Daten in allen Programmteilen. Je nach dem Ort dieser Deklaration im Quellprogramm finden sich die entsprechenden Datenreservierungen im Laufzeit-Layout an unterschiedlichen Stellen. Aus Programmiererersicht auf Sprachebene sind die Orte der Datenreservierungen im übersetzten Programm weitgehend von untergeordnetem Interesse. Daher benötigt der Anwender auf Pascal-Sprachebene, etwa durch Angaben von Speicherklassen für Variablen wie in C oder C++, keinen expliziten Einfluss auf deren Lage.

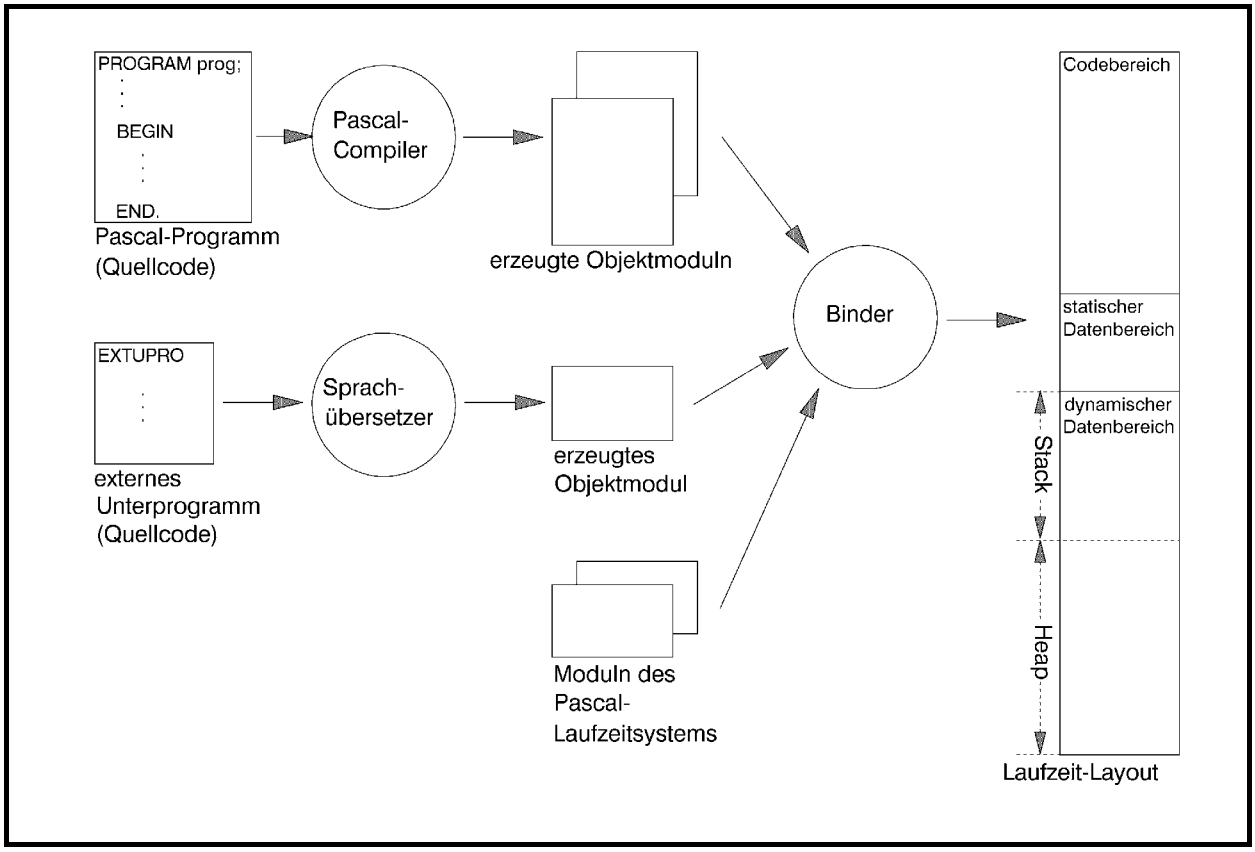


Abbildung 3.1.4-1: Laufzeit-Layout bei der Pascal-Programmierung

Das typische Laufzeit-Layout des virtuellen Adressraums bei der Pascal-Programmierung ist in Abbildung 3.1.4-1 dargestellt. Ein Pascal-Programm mit Bezeichner `prog`, das interne Unterprogramme, Datendefinitionen und den Aufruf eines externen Unterprogramms `EXTUPRO` enthält, wird vom Pascal-Compiler übersetzt. Das externe Unterprogramm, das in einer anderen Programmiersprache geschrieben sein kann, wird ebenfalls mit dem erforderlichen Sprachübersetzer übersetzt. Der Binder bindet neben den erzeugten Objektmoduln weitere Moduln des Laufzeitsystems von Pascal an.

Das ablauffähige Programm besteht aus folgenden unterscheidbaren Bereichen:

- Der **Codebereich** enthält alle ausführbaren Anweisungen des Programms `prog` und aller internen Prozeduren (aus `prog`) und externen Prozeduren (`EXTUPRO` und Prozeduren des Laufzeitsystems); außerdem sind alle deklarierten Konstanten in den Code eingearbeitet (ein Pascal-Compiler verarbeitet üblicherweise Konstantendefinitionen, indem er sie an den entsprechenden Stellen direkt in den Code einsetzt)
- Der **statische Datenbereich** enthält alle im Hauptprogramm `prog` definierten Variablen, sofern ihr Speicherplatzbedarf bereits zur Übersetzungszeit bekannt ist; über Pointer definierte (dynamische) Variablen gehören nicht dazu, wohl aber die Variablen, die die Pointer aufnehmen, die auf derartige Daten verweisen (sofern sie im Hauptprogramm definiert sind); sogenannte typisierte Konstanten des Hauptprogramms, d.h. Konstanten-Deklarationen, die mit Anfangswerten initialisiert werden, liegen ebenfalls im statischen Datenbereich, da sie wie Variablen behandelt werden
- Der **dynamische Datenbereich** ist in zwei Teile unterteilt: in den Heap und den Stack; die Gesamtgröße beider Bereiche lässt sich durch Compileroptionen festlegen; das Größenverhältnis beider Teilbereiche zueinander verändert sich dynamisch während der Laufzeit: dabei wächst im allgemeinen der Heap von niedrigen zu hohen Adressen und der Stack in umgekehrter Richtung (in einigen Implementationen wird die Richtung umgedreht); falls sich die Bereichsgrenzen während der Laufzeit überschneiden sollten, kommt es zu einem Laufzeitfehler mit Programmabbruch (heap overflow bzw. stack overflow); Heap und Stack sind komplett unterschiedlich organisiert und werden durch Routinen des Laufzeitsystems verwaltet.

Der **Heap** enthält alle während der Laufzeit durch die Pascal-Standardprozedur `New` eingerichtete Variablen, die über Pointer angesprochen werden.

Der **Stack** dient der Aufnahme aller lokalen Variablen einer Prozedur: Dazu gehören im wesentlichen alle Variablen, die innerhalb der Prozedur deklariert werden. Variablen, die in einer eingebetteten Prozedur deklariert werden, sind lokal zu dieser eingebetteten Prozedur.

Das hier beschriebene Layout zur Laufzeit ist nur als prinzipiell und beispielhaft zu betrachten. Je nach Sprachdialekt, eingesetztem Betriebssystem und Speichermodell des verwendeten Rechners gibt es Abweichungen (z.B. die relative Lage der einzelnen Bereiche zueinander).

3.2 Das Prozedurkonzept

Ein Unterprogramm wird in einer Programmiersprache meist als Prozedur deklariert. Pascal kennt als Prozedur das syntaktische Konstrukt der `PROCEDURE` und als Spezialform die mit Hilfe des Schlüsselworts `FUNCTION` vereinbarte Funktion. C kennt nur eine der Funktion entsprechende Form. In COBOL können „interne“ Prozeduren als Paragraphen oder Sections

(aufgrufen mit `PERFORM`) oder „externe“ Prozeduren (aufgerufen mit `CALL`) vereinbart werden.

Im folgenden wird neben einer allgemeinen Behandlung des Prozedurkonzepts einer höheren Programmiersprache im speziellen auf das Prozedurkonzept von Pascal eingegangen.

Eine **Prozedurdeklaration** besteht aus **Prozedurkopf** und **Prozedurrumpf**. In Pascal sieht die Prozedurdeklaration bezüglich des syntaktischen Aufbaus wie ein Programm aus und kann selbst wieder eingebettete Prozeduren enthalten (diese eingebettete Prozeduren sind dann nur innerhalb der Prozedur bekannt).

Der **Prozedurkopf („Musteranweisung“, Aufrufformat)** legt fest:

- mit welchem **Bezeichner** eine Prozedur aufgerufen wird
- die **Liste der Formalparameter** der Prozedur: hierbei wird für jeden Formalparameter
  - sein Bezeichner innerhalb der Prozedur
  - sein Datentyp
  - die Reihenfolge, in der er in der Liste der Formalparameter auftritt
  - die Art und Weise, wie über diesen Formalparameter Werte an die Prozedur bzw. aus der Prozedur zum Aufrufer zurück Werte übergeben werden können: die **Parameterübergabemethode**

  festgelegt. Jeder Formalparameter stellt eine innerhalb der Prozedur lokale Variable dar, die bei Aufruf der Prozedur vom Aufrufer mit Anfangswerten initialisiert wird. Nach außen hin, d.h. zum Block, der die Prozedurdeklaration enthält, ist lediglich der Prozedurbezeichner bekannt und nicht etwa der Bezeichner eines Formalparameters. Die Liste der Formalparameter kann auch fehlen; die Prozedur hat dann keine Formalparameter.

Das syntaktische Format einer Prozedurdeklaration in Pascal lautet für eine Prozedur mit Bezeichner `xyz`:

```
PROCEDURE xyz (liste_der_Formalparameter);
```

Der **Prozedurrumpf** enthält eventuell Deklarationen weiterer lokaler Bezeichner und Objekte (Konstanten, Variablen, Prozeduren, Funktionen usw.) und den Anweisungsteil, d.h. die Anweisungen, die bei Aufruf der Prozedur durchlaufen werden. Der Prozedurrumpf stellt einen Block dar; insbesondere gelten hier die Regeln über die Gültigkeit von Bezeichnern.

Zusammenfassend sind die (lokalen) Datenobjekte, auf die innerhalb des Prozedurrumpfs zugegriffen werden kann, ***die Formalparameter und die im Prozedurrumpf deklarierten Datenobjekte.*** Zusätzlich kann eine Prozedur eventuell auf für sie globalen Datenobjekte zugreifen, die in einem sie umfassenden Block deklariert sind (wenn diese nicht mit einem Bezeichner benannt werden, der innerhalb der Prozedur für ein lokales Datenobjekt verwendet wird).

Die Verwendung derartiger ***globaler Objekte sollte in einer Prozedur*** jedoch ***vermieden werden.***

Der **Aufruf einer Prozedur** erfolgt durch Nennung des zugehörigen Bezeichners und durch die Angabe der **Liste der Aktualparameter**, falls die Prozedur in ihrer Prozedurdeklaration Formalparameter vorsieht. ***Für jeden Formalparameter muss es einen korrespondierenden Aktualparameter geben, der im Datentyp mit dem Formalparameter übereinstimmt (kompatibel ist).*** Ein Aktualparameter ist meist ein Variablenbezeichner im Gültigkeitsbereich des Aufrufers; je nach Parameterübergabemethode ist eventuell auch eine Konstante oder ein arithmetischer Ausdruck zugelassen. Bei Prozeduraufruf wird der Aktualparameter „an das Unterprogramm übergeben"; ist der Aktualparameter ein arithmetischer Ausdruck, wird dieser vorher berechnet. Dann verzweigt der Programmfluss in die Anweisungen des bezeichneten Prozedurrumpfs. Nachdem dieser bis zum Ende durchlaufen ist, erfolgt ein **Rücksprung** genau hinter die Aufrufstelle der Prozedur. Nach dem Rücksprung sind die lokalen Datenobjekte der Prozedur nicht mehr verfügbar: ihre Lebensdauer, die bei Eintritt des Kontrollflusses in die Prozedur beginnt, ist beendet.

Eine spezielle Form einer Prozedur in Pascal ist eine Funktion. Eine Funktion besteht wie eine Prozedur aus einem Prozedurkopf und einem Prozedurrumpf. Die syntaktische Konstruktion lautet:

```
FUNCTION funktionsbezeichner (liste_der_formalparameter) :
                                                    typbezeichner;
```

z.B.

```
FUNCTION fakultaet (n : INTEGER) : INTEGER;
```

Der Prozedurkopf enthält den Funktionsbezeichner, die Liste der Formalparameter und einen Typbezeichner, der den Datentyp des Funktionsergebnisses festlegt. Für die Liste der Formalparameter gilt das gleiche wie für die Liste der Formalparameter bei einer „normalen" Prozedur. Innerhalb des Prozedurrumpfs gibt es eine Wertzuweisung, die auf der ***linken Seite*** den Funktionsbezeichner hat und diesem einen Wert vom Typ des Typbezeichners zuordnet. Dadurch wird der Funktionswert festgelegt, der als Rückgabeparameter der Funktion fungiert:

```
...
 funktionsbezeichner := ...;
...
```

Im aufrufenden Programm wird der Funktionsbezeichner auf der ***rechten Seite*** einer Anweisung in einem Ausdruck (wie eine Variable) verwendet. In diesem Fall verzweigt der Programmfluss wie bei einem Unterprogrammaufruf in den Prozedurrumpf der Funktion, durchläuft die entsprechenden Anweisungen und kehrt direkt hinter die Stelle zurück, an der beim

Aufrufer der Funktionsbezeichner steht. Als Besonderheit können Funktionen in Object Pascal zusätzlich wie normale Prozeduren aufgerufen werden, d.h. einfach durch Angabe des Funktionsbezeichners und nicht etwa auf der rechten Seite einer Wertzuweisung innerhalb eines Ausdrucks. In diesem Fall wird das Funktionsergebnis ignoriert.

Die Einführung des syntaktischen Konzepts der `FUNCTION` bedeutet in vielen Anwendungsfällen ein Vereinfachung in der Programmierung.

3.2.1 Parameterübergabe

Je nach Programmiersprache werden unterschiedliche Methoden verwendet, um Werte zwischen einer Prozedur und ihrem Aufrufer auszutauschen. Zunächst soll daher ein allgemeiner Überblick über die **Methoden der Parameterübergabe an Unterprogramme** gegeben werden, wie sie in höheren Programmiersprachen üblich sind. Dabei werden in diesem Kapitel weniger Implementierungsaspekte als vielmehr die **Semantik der Methoden zur Parameterübergabe aus Anwendersicht** behandelt.

Die Methoden lassen sich danach einteilen, ob sie dazu geeignet sind, Parameterwerte an eine Prozedur zu übergeben (Eingabe von Informationen in ein Unterprogramm), Parameterwerte aus einer Prozedur zurückzuerhalten (Rückgabe von Informationen aus einem Unterprogramm an den Aufrufer) oder ob sie „für beide Richtungen“ geeignet sind (Änderung von Informationen durch ein Unterprogramm).

Alle Methoden tragen Namen der Form „call-by-...“-Methode.

Eine für alle drei Teilaufgaben geeignete Methode ist die **call-by-reference-Methode**. Hierbei wird vom Compiler Code generiert, der beim Eintritt in die Prozedur die ***Adresse des korrespondierenden Aktualparameters*** an den Formalparameter (als Wert) übergibt. Im gerufenen Unterprogramm wird dann eine Bezugnahme auf den entsprechenden Formalparameter als Referenz auf die Position behandelt, deren Adresse übergeben wurde. Es wird also bei jedem Zugriff auf den Formalparameter tatsächlich eine indirekte Referenz über die bekanntgegebene Adresse auf den nicht-lokalen Aktualparameter durchgeführt. ***Eine als aktueller Parameter übergebene Variable wird somit direkt bei jeder Änderung vom Unterprogramm modifiziert.*** Diese semantische Regel bedeutet, dass durch die Wertzuweisung

```
form_param := wert_neu;
```

innerhalb einer Prozedur mit dem Formalparameter `form_param` nicht etwa das durch `form_param` bezeichnete Datenobjekt den neuen Wert `wert_neu` erhält, sondern dass der Wert von `form_param` als Adresse eines Datenobjekts (nämlich des korrespondierenden Ak-

tualparameters) interpretiert wird, dem nun der Wert `wert_neu` zugewiesen wird. Der Compiler hat bei der Übersetzung entsprechenden Code erzeugt. Analog wird durch die Auswertung bzw. des Lesens des Formalparameters `form_param`

```
IF form_param = ... THEN ...
```

innerhalb des Unterprogramms der Wert des korrespondierenden Aktualparameters ausgewertet.

Eine Besonderheit ist zu beachten: ***Wird innerhalb einer Prozedur ein call-by-reference-Parameter, z.B. mit Bezeichner*** `ref`***, als Aktualparameter für einen weiteren (untergeordneten) Prozeduraufruf nach dem call-by-reference-Prinzip verwendet, so wird nicht die Adresse von*** `ref` ***an diesen Prozeduraufruf weitergegeben, wie es die call-by-reference-Methode vorschreibt, sondern der Wert von*** `ref`**.** Dieser Wert ist die Adresse des „äußeren" Aktualparameters, die somit der untergeordneten Prozedur mitgeteilt wird. Damit wirkt sich jeder Zugriff innerhalb der untergeordneten Prozedur direkt auf den Wert des äußeren Aktualparameters aus.

In Pascal wird jeder **Variablenparameter**, d.h. jeder Formalparameter, der im Prozedurkopf durch das Schlüsselwort `VAR` gekennzeichnet ist, nach der call-by-reference-Methode behandelt (der dazu korrespondierende Aktualparameter muss eine Variable sein):

```
PROCEDURE xyz (VAR call_by_ref_parameter : ...);
```

Um nur aktuelle Parameterwerte an eine Prozedur zu übergeben, eignet sich neben der call-by-reference-Methode im wesentlichen die **call-by-value-Methode**. Hierbei ist ein Formalparameter eine lokale Variable, die bei Eintritt in die Prozedur mit dem ***Wert des korrespondierenden Aktualparameters*** durch eine Kopie dieses Werts initialisiert wird. Der Aktualparameter kann eine Variable, eine Konstante oder ein Ausdruck sein, da nur der Wert des aktuellen Parameters interessiert; bei einem Ausdruck als Aktualparameter wird dieser vorher ausgewertet. Die Prozedur verändert den Wert des Aktualparameters nicht, auch wenn sie innerhalb des Prozedurrumpfs den Formalparameter mit einem neuen Wert versieht. Es werden also unerwünschte Nebeneffekte für Eingabeparameter von vornherein vermieden. Ein Nachteil dieser Methode besteht darin, dass im Falle eines strukturierten Datenobjekts als Formal- und Aktualparameter, etwa im Fall eines Feldes, eine Kopie des kompletten Aktualparameters in den Formalparameter übertragen werden muss, was u.U. sehr speicherplatz- und zeitaufwendig ist. Daher werden in einigen Programmiersprachen als Ausnahme alle Wertparameter, die intern mehr als eine definierte Anzahl Bytes (beispielsweise 4 Bytes) belegen, nicht nach der call-by-value-Methode, sondern nach der call-by-reference-Methode übergeben, um genau die beschriebene Ineffizienz bei der Übergabe „großer" Aktualparameter zu vermeiden. Der vom Compiler für den Zugriff auf derartige Formalparameter erzeugt Code bewirkt jedoch, dass sich der Parameter „nach außen hin" wie ein normaler call-by-value-Parameter verhält, d.h.

dass insbesondere durch die Prozedur der Wert des korrespondierenden Aktualparameter beim Aufrufer nicht verändert wird.

Bezüglich der Übergabe von Werten als Eingabeinformationen in ein Unterprogramm bietet die call-by-value-Methode natürlich eine größere Sicherheit (z.B. gegen Programmierfehler) als die call-by-reference-Methode, da bei der call-by-reference-Methode der Aktualparameter direkt, bei der call-by-value-Methode nicht der Aktualparameter selbst, sondern nur eine Kopie seines Werts verändert wird.

In Pascal ist für jeden **Wertparameter** die call-by-value-Methode implementiert. Ein derartiger Formalparameter ist in der Liste der Formalparameter im Prozedurkopf *nicht* mit dem Schlüsselwort `VAR` versehen.

Bei der **call-by-constant-value-** und der **call-by-reference-value-Methode** als Spezialformen der call-by-value-Methode fungieren die Formalparameter als lokale Konstanten, deren Werte beim Eintritt in die Prozedur mit den Werten der korrespondierenden Aktualparameter initialisiert werden und die während des Ablaufs der Prozedur nicht geändert werden können. Object Pascal bezeichnet einen Formalparameter, der nach der call-by-constant-value-Methode behandelt wird, als **konstanten Parameter**; ein derartiger Formalparameter wird durch Voranstellen des Schlüsselworts `CONST` gekennzeichnet.

Die **call-by-content-Methode** (in COBOL-85) erweitert die ursprünglich in COBOL ausschließlich vorgesehene call-by-reference-Methode. Bei Aufruf einer Prozedur wird für jeden Aktualparameter eine für den Anwender nicht sichtbare Kopie angelegt. Diese Kopie wird dem Unterprogramm als Aktualparameter nach dem call-by-reference-Prinzip übergeben (d.h. es wird die Adresse dieser Kopie an das Unterprogramm übergeben), so dass innerhalb der Prozedur jeder Formalparameter als call-by-reference-Parameter behandelt werden kann. Zum Aufrufer stellt sich dieses Prinzip wie die call-by-value-Methode dar.

Bei der **call-by-result-Methode** wird der Formalparameter wie eine nicht-initialisierte lokale Variable benutzt, der während der Ausführung der Prozedur im Prozedurrumpf ein Anfangswert zugewiesen wird, der dann auch verändert werden kann. Am Ende der Prozedur wird der Wert des Formalparameters an den korrespondierenden Aktualparameter übergeben, der eine Variable (keine Konstante und kein Ausdruck) sein muss.

Die call-by-result-Methode wird beispielsweise in Ada bei der Implementierung von **out**-Parametern eingesetzt. Hierbei wird die Adresse des Aktualparameters, der zu einem Formalparameter gemäß call-by-result gehört, ***bei Eintritt*** in die Prozedur bestimmt; diese Adresse wird dann am Ende der Prozedur zur Rückgabe des Werts des Formalparameters verwendet. Andere Implementierungen dieser Methode sehen vor, diese Adresse erst unmittelbar vor Rückgabe des Werts des Formalparameters, also ***bei Verlassen*** der Prozedur, zu bestimmen. Beide Ansätze führen in einigen speziellen Situationen zu unterschiedlichen Ergebnissen.

Für Formalparameter, denen Werte von einem Aufrufer übergeben werden, die dann während der Ausführung der Prozedur verändert und an den Aufrufer zurückgegeben werden können, eignet sich neben der beschriebenen call-by-reference-Methode die **call-by-value-result-Methode** (in Ada für skalare **in-out**-Parameter). Bei dieser Methode wird der Formalparameter wieder als lokale Variable behandelt, die bei Eintritt in die Prozedur mit dem gegenwärtigen Wert des korrespondierende Aktualparameters initialisiert wird. Innerhalb der Prozedur beeinflussen Änderung des Werts des Formalparameters nur diese lokale Kopie (entsprechend der call-by-value-Methode). Am Ende der Prozedur wird der gegenwärtige Wert des Formalparameters an den Aktualparameter zurückgegeben.

Auch hier kommt es wie bei der call-by-result-Methode wieder darauf an, wann die Adresse des Formalparameters ermittelt wird, nämlich beim Eintritt in die Prozedur oder unmittelbar vor dem Verlassen des Unterprogramms.

In den meisten Fällen ergeben die call-by-value-result- und die call-by-reference-Methode bei vollständiger Abarbeitung einer Prozedur die gleichen Resultate. Ausnahmen stellen Situationen dar, in denen Interaktionen zwischen Parametern und nicht-lokalen Variablen vorkommen. Ein Programmierer sollte aber darauf achten, die Prozeduren so zu schreiben, dass ihre Ergebnisse nicht von den zugrundeliegenden Methoden der Parameterübergabe abhängen.

Call-by-value-result hat die gleichen Nachteile bezüglich der Ineffizienz bei strukturierten Datenobjekten als Parameter wie call-by-value. Andererseits hat call-by-value-result gegenüber call-by-reference den Vorteil, dass sich alle Referenzen innerhalb des Unterprogramms auf lokale Variablen (die Formalparameter) beziehen und damit schneller ablaufen, als die bei call-by-reference benutzten indirekten Referenzen auf nicht-lokale Objekte (die Aktualparameter). Eine weitere Konsequenz ergibt sich im Falle eines Abbruchs der Prozedur noch vor ihrem Ende aufgrund eines Fehlers: Bei call-by-reference kann der aktuelle Parameter inzwischen einen durch die Prozedur veränderten Wert haben, während bei call-by-value-result der Wert des Aktualparameters noch der gleiche ist, wie bei Eintritt in die Prozedur, da ja der Wert des Aktualparameters erst am Ende der Prozedur, das noch nicht erreicht ist, aktualisiert wird.

Abschließend soll noch die **call-by-name-Methode** erwähnt werden, die in ALGOL 60 definiert wurde, aber in heutigen Programmiersprachen nicht verwendet wird, da sie sehr viel aufwendiger zu implementieren ist als call-by-reference und ansonsten nur in ausgefallenen Situationen Vorteile bietet. Bei call-by-name wird die Adresse des Aktualparameters (anders als bei call-by-reference) erst dann bestimmt, wenn der korrespondierende Formalparameter auch benutzt wird, und zwar wird bei ***jeder Verwendung des Formalparameters*** während des Prozedurlaufs diese ***Auswertung erneut*** durchgeführt. Diese Methode ermöglicht es, dass unterschiedliche Adressen für verschiedene Zugriffe auf denselben Formalparameter während der Ausführung einer Prozedur verwendet werden.

Die folgende Tabelle fasst die Möglichkeiten der beschriebenen Methoden zur Parameterübergabe an Unterprogramme zusammen:

Methode call-by-	Parameter geeignet zur		
	Übergabe von aktuellen Werten an das Unterprogramm	Rückgabe von aktuellen Werten aus dem Unterprogramm	Übergabe und Rückgabe von aktuellen Werten an das bzw. aus dem Unterprogramm
reference	x	x	x
value / content	x		
constant-value	x		
reference-value	x		
result		x	
value-result	x	x	x

Das folgende Beispiel aus [W/C] erläutert die einzelnen Methoden noch einmal. Es zeigt in Abbildung 3.2.1-1 eine Prozedur mit Bezeichner `methoden`, die im Pascal-Stil geschrieben ist. In diesem Beispiel soll jedoch der Aktualparameter an die Prozedur nach den verschiedenen beschriebenen Methoden zu übergeben sein (also nicht nur nach der in Pascal in dieser Situation gemäß der Syntax verwendeten call-by-value-Methode). Je nach Methode ergeben sich u.U. verschiedene Resultate. Es muss betont werden, dass der „trickreiche“ Programmierstil dieses Beispiels zu vermeiden ist, zumal das Unterprogramm auch noch globale Variablen verwendet; das Beispiel dient lediglich der Demonstration der Unterschiede der einzelnen Methoden.

```
PROGRAM beispiel;

TYPE feld_typ = ARRAY [1..2] OF INTEGER;

VAR element : INTEGER;
    a       : feld_typ;

PROCEDURE methoden (x : INTEGER);

  BEGIN { methoden }
    a[1]    := 6;
    element := 2;
    x       := x + 3
  END   { methoden };

BEGIN { beispiel }
  a[1]    := 1;
  a[2]    := 2;
  element := 1;
  methoden (a[element])
END   { beispiel }.
```

Resultate bei den verschiedenen Methoden der Parameterübergabe:

Methode	a[1]	a[2]	element
call-by-reference	9	2	2
call-by-value	6	2	2
call-by-value-result (Parameterbestimmung bei Verlassen)	6	4	2
call-by-value-result (Parameterbestimmung bei Eintritt)	4	2	2
call-by-name	6	5	2

Abbildung 3.2.1-1: Parameterübergabemethoden (Beispiel)

Innerhalb von `methoden` werden `a[1]` und `element` als globale Variablen behandelt; je nach Methode kann auf `a[1]` bzw. `a[2]` über den Formalparameter `x` zugegriffen werden.

Wenn in diesem Beispiel der Parameter nach der call-by-result-Methode übergeben wird, entsteht ein Fehler, da der Formalparameter `x` vor seiner Verwendung auf der rechten Seite des Ausdrucks

```
x := x + 3;
```

nicht initialisiert worden ist.

Wenn der Parameter `x` nach der call-by-reference-Methode übergeben wird, kann auf `a[1]` auf zwei unterschiedliche Arten zugegriffen werden, nämlich entweder direkt als globales Datenobjekt in

```
a[1] := 6;
```

oder indirekt als Formalparameter in

```
x := x + 3;
```

Bei der call-by-value-Methode beeinflusst die Anweisung

```
x := x + 3;
```

den Wert von `a[1]`, des aktuellen Parameters, nicht, da in `methoden` mit einer lokalen Kopie von `a[1]` gearbeitet wird.

Die Ursache für Unterschiede, die sich bei den beiden call-by-value-result-Methoden ergeben, liegt im Bestimmungszeitpunkt der Adresse für die Rückgabe des Werts im Formalparameter an den korrespondierenden Aktualparameter. Wird diese Adresse, nämlich die Adresse des Aktualparamters `a[element]`, beim Eintritt in die Prozedur bestimmt, dann handelt es sich um die Adresse von `a[1]`. Wird die Adresse unmittelbar vor Verlassen der Prozedur ermittelt, dann ist aufgrund der Anweisung

```
element := 2;
```

die Adresse von `a[element]` die Adresse von `a[2]`. Der gegenwärtige Wert von `x` wird also entweder an `a[1]` oder an `a[2]` zurückgegeben, nachdem er in beiden Fällen bei Eintritt in die Prozedur mit dem Wert von `a[1]` initialisiert wurde.

Der Unterschied in den Ergebnissen der call-by-reference- und der call-by-name-Methode hat eine ähnliche Ursache: Der Formalparameter `x` wird bei call-by-reference *bei Eintritt in die Prozedur* mit der Adresse von `a[1]` versorgt wird, da `element` den Wert 1 enthält. Bei call-by-name wird der Formalparameter in der Anweisung

```
x := x + 3;
```

erneut berechnet, wobei für `x` der Aktualparameter `a[element]` steht und `element` inzwischen den Wert 2 hat.

3.2.2 Implementierungsprinzip von Prozeduren

Um einen korrekten Prozedurablauf, insbesondere die Übergabe von Aktualparametern, Handhabung lokaler Datenobjekte, Realisierung rekursiver Prozedurabläufe und Rücksprung zu gewährleisten, wird der Stack verwendet, über den jeder Thread eines Prozesses verfügt.

Der **Stack (Stapel, Keller)** ist ein Datenbereich mit (strukturierten und variabel langen) Einträgen, der so organisiert ist, dass immer nur auf den zuletzt eingetragenen Eintrag zugegriffen werden kann (lesen, entfernen). Der Stack kann durch die Reservierung eines fest vereinbarten Datenbereichs und eines global zugreifbaren Datenobjekts implementiert werden, der im folgenden mit dem Bezeichner **SP (Stackpointer)** bezeichnet wird. Befinden sich Daten im Stack, so belegen sie einen zusammenhängenden Bereich; der restliche Abschnitt des Stacks ist frei. Der Wert von SP identifiziert den zuletzt in den Stack aufgenommenen Eintrag. Die Einträge im Stack sind im allgemeinen strukturiert und unterschiedlich groß. Innerhalb jedes Eintrags steht an einer fest definierten Stelle ein Verweis auf den Eintrag, der zeitlich genau vorher in den Stack eingetragen wurde (bei gleichartig strukturierten Stackeinträgen kann man diese hintereinander schreiben; der vorherige Eintrag steht dann in einer konstanten Distanz, entsprechend der Länge eines Stackeintrags, zum Stackeintrag, der durch den Wert von SP identifiziert wird).

Mit einem Stack sind im wesentlichen drei Operationen mit folgender Wirkung auf den Stack verbunden:

- **Initialisierung des Stacks** als leerer Stack
- **Eintragen eines neuen Stackeintrags**; dabei wird der Inhalt von SP so aktualisiert, dass er nun den neuen Eintrag identifiziert
- **Entfernen des obersten Stackeintrags vom Stack**; dabei wird der Inhalt von SP entsprechend verändert, falls der Stack nicht leer ist; bei einem leeren Stack hat diese Operation keine Wirkung.

Zur Illustration der Anwendung des Stackprinzips im Zusammenhang mit dem Prozedurkonzept dienen wieder in Pascal geschriebene Beispiele. Es muss betont werden, dass es sich bei der im folgenden dargestellten Methode um *eine mögliche Umsetzung des Konzepts* handelt, die je nach Programmiersprache und Compiler variiert.

Die in einem Programm definierten Datenobjekte lassen sich bezüglich ihrer logischen Zugehörigkeit zu Programmteilen unterscheiden in

- ***globale Datenobjekte des Hauptprogramms***, die von allen Blöcken des Programms aus sichtbar, d.h. zugreifbar sind, wenn nicht innerhalb eines Blocks eine Deklaration mit demselben Bezeichner erfolgt

- ***globale Datenobjekte*** aus der Sicht einer Prozedur, nämlich solche, die in Blöcken deklariert sind, die die Prozedur (evtl. über mehrere Hierarchiestufen) einschachteln und in der Prozedur nicht durch Deklarationen mit denselben Bezeichnern überlagert werden
- ***lokale, innerhalb einer Prozedur oder Funktion definierte***, nur dieser Prozedur zugehörige ***Datenobjekte***; diese sind für alle darin eingebettete Prozeduren und Funktionen global, nach außen aber nicht sichtbar
- ***Formalparameter einer Prozedur oder Funktion***, die wie lokale Datenobjekte einer Prozedur betrachtet werden
- ***Aktualparameter für den Aufruf einer Prozedur oder Funktion***, die zum Aufrufer der Prozedur oder Funktion gehören.

Auf lokale Datenobjekte einer Prozedur (Funktionen werden wieder als syntaktisch spezielle Prozeduren angesehen) wird erst zugegriffen, wenn diese Prozedur während der Laufzeit auch aufgerufen wird. Daher braucht Speicherplatz für diese Datenobjekte erst bei Aufruf der Prozedur entsprechend dem Beginn ihrer Lebensdauer zugewiesen werden. Falls die Prozedur also gar nicht aufgerufen wird, braucht auch kein Speicherplatz für lokale Datenobjekte reserviert zu werden. Bei Verlassen der Prozedur endet die Lebensdauer der lokalen Datenobjekte, und damit kann der für sie verwendete Speicherplatz wieder freigegeben werden. Der Speicherplatz für globale Datenobjekte des Hauptprogramms wird beim Laden des Programms reserviert.

Bei Programmstart wird der Stack als leer initialisiert.

Bei jedem Aufruf einer Prozedur wird ***vom Aufrufer und von der aufgerufenen Prozedur*** ein neuer Stackeintrag erzeugt, der als **Aktivierungsrecord (des Aufrufs)** bezeichnetet wird. Die Größe des Aktivierungsrecords hängt von der Anzahl lokaler Variablen der Prozedur ab. Der Compiler generiert sowohl beim Aufrufer als auch bei der aufgerufenen Prozedur entsprechenden Code zur Erzeugung des Aktivierungsrecords. Der bei Einsprung in eine Prozedur erzeugte Aktivierungsrecord wird beim Rücksprung zum Aufrufer von der gerufenen Prozedur wieder entfernt.

Der Aktivierungsrecord eines Prozeduraufrufs enthält in einer implementierungsabhängigen Reihenfolge:

- die ***Rücksprungadresse zum Aufrufer***
- die **Speicherplatzreservierung für jedes lokale Datenobjekt**, d.h. für jeden ***Formalparameter*** und jede in der Prozedur deklarierte ***lokale Variable***; Formalparameter sind bei Start des Prozedurcodes bereits vom Aufrufer mit den Werten der Aktualparameter entsprechend der jeweiligen Parameterübergabemethode initialisiert worden
- das **Display**; hierbei handelt es sich um eine Tabelle aus Verweisen auf diejenigen, vorher im Stack eingetragenen Aktivierungsrecords, die aus Sicht des jeweiligen Prozeduraufrufs globale Datenobjekte enthalten; die Display-Informationen resultieren aus der Analyse

der statischen Blockstruktur, die zur Übersetzungszeit durchgeführt wird und werden zur Laufzeit der aufgerufenen Prozedur erzeugt (siehe unten)

- ***Einträge zur Verwaltung des Stacks***, insbesondere einen Adressverweis auf den vorherigen Stackeintrag.

Während des Prozedurablaufs können evtl. weitere Einträge im Stack vorgenommen werden, z.B.

- ***Sicherstellungsbereiche für Registerinhalte***: wenn innerhalb des Prozedurcodes Register verwendet werden, müssen deren Inhalte natürlich zuvor (lokal) gesichert und vor dem Rücksprung zum Aufrufer rekonstruiert werden, damit der Aufrufer dieselben Registerinhalte wie unmittelbar vor Eintritt in die Prozedur sieht; des weiteren werden die Register bei einem Threadwechsel im Stack des Threads gesichert
- Speicherplatz für die ***lokale Speicherung von Zwischenergebnissen***.

Diese möglichen zusätzlichen Stackeinträge bleiben in der folgenden Darstellung unberücksichtigt.

Der Wert des Stackpointers SP identifiztiert den „obersten“ Stackeintrag. Diese Identifizierung erfolgt in Form einer Adresse. Im folgenden bewegt sich der belegte Teil des Stacks beim Eintragen von Werten in Richtung niedrigerer Adressen und beim Entfernen von Einträgen in Richtung höherer Adressen.

Ein Aktivierungsrecord, der komplett erzeugt und im Stack abgelegt ist, besteht also selbst aus mehreren Komponenten, deren Anfänge jeweils einen relativen Abstand (Offset) zu der Adresse haben, die in SP steht. Um zu verdeutlichen, dass der Zugriff auf diese Komponenten eine lokale Operation auf der Ebene der jeweiligen Prozedur bzw. des Aufrufers ist, während die Manipulationen des SP-Werts global an den Stack gebunden sind, wird ein weiteres globales Datenobjekt verwendet, das **Basispointer (BP)** genannt werden soll. BP enthält eine Adresse, die in den Stack verweist, und zwar im wesentlichen die Adresse, die den Aktivierungsrecord in die beiden Teile teilt, die vom Aufrufer einer Prozedur bzw. von der Prozedur selbst generiert werden. Alle Einträge im Aktivierungsrecord unterhalb des BP-Werts (in Richtung größerer Adressen relativ zum BP-Wert) werden vom Aufrufer erzeugt. Alle Einträge oberhalb des BP-Werts (in Richtung absteigender Adressen relativ zum BP-Wert) werden von der gerufenen Prozedur angelegt (Abbildung 3.2.2-1).

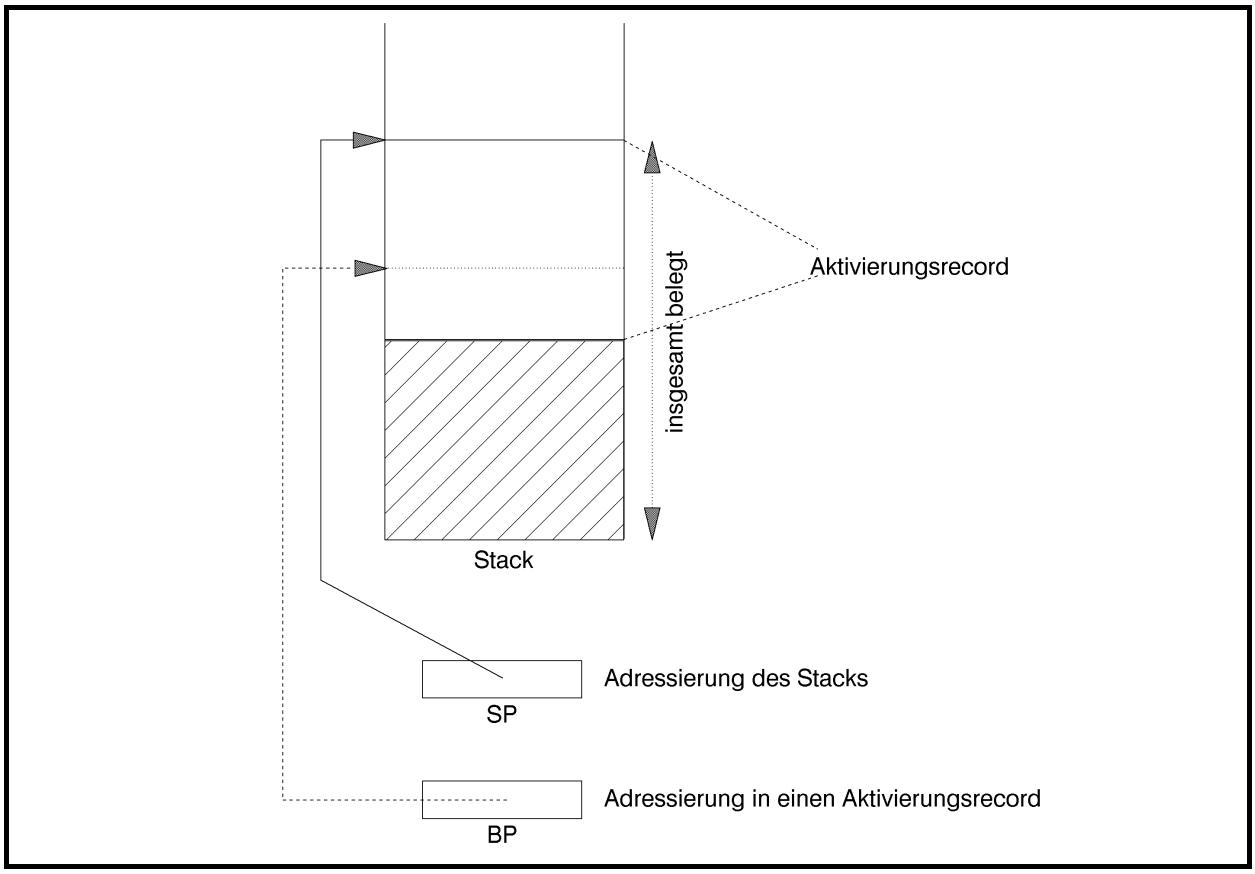


Abbildung 3.2.2-1: Aktivierungsrecord im Stack

Das Implementierungsprinzip soll zunächst am Layout eines Aktivierungsrecords für einen Aufruf einer Prozedur verdeutlicht werden, die nur auf ihre eigenen lokalen Variablen zugreift, d.h. auf die Variablen, die innerhalb der Prozedur oder als Formalparameter deklariert sind.

Die Prozedur

```
PROCEDURE proc (f1 : ...; ...; fn : ...;);
  { lokale Variablen von proc: }
  VAR v1 : ...;
      ...
      vm : ...;

  BEGIN { proc }
    ...
  END   { proc };
```

werde durch

```
proc (a1, ..., an);
```

aufgerufen. Die Aktualparameter $a_1, \ldots, a_n$ gehören zum Aufrufer. Die Details, die sich aus den Datentypen der einzelnen Datenobjekte und der Parameterübergabemethode ergeben, sollen zunächst nicht betrachtet werden.

Der Compiler hat für den Prozeduraufruf Code erzeugt, der nacheinander die Werte bzw. Adressen der Aktualparameter $a_1, \ldots, a_n$ (je nach Parameterübergabemethode) und die Rücksprungadresse auf den Stack transportiert. Der Speicherplatz für diese Werte bzw. Adressen der Aktualparameter auf dem Stack ist identisch mit dem Speicherplatz für die Formalparameter $f_1, \ldots, f_n$ der gerufenen Prozedur. Jetzt erfolgt eine Verzweigung in den Code von `proc` (Abbildung 3.2.2-2 (a)).

Der Code, der für das `BEGIN`-Statement in der Prozedur `proc` generiert wurde, bewirkt, dass der gegenwärtige BP-Wert auf dem Stack abgelegt wird und der BP mit dem jetzigen SP-Wert überschrieben wird. Dadurch wird eine Rückwärtsverkettung in den Aktivierungsrecord des Aufrufers hergestellt, und zwar an die Stelle, die ihrerseits eine Rückwärtsverkettung zu einem vorherigen BP-Wert enthält. Der neue BP-Wert ist der Anfang dieser Rückwärtsverkettung. Anschließend wird der SP-Wert soweit dekrementiert, wie Platz für die lokalen Variablen $v_1, \ldots, v_m$ der Prozedur `proc` benötigt wird, d.h. der Speicherplatz für $v_1, \ldots, v_m$ ist nun zugewiesen (Abbildung 3.2.2-2 (b)).

Am Ende von `proc` bewirkt der für das `END`-Statement vom Compiler eingesetzte Code, dass $v_1, \ldots, v_m$ vom Stack wieder entfernt werden, der BP mit der Rückwärtsverkettung wieder restauriert wird und ein Rücksprung an die Adresse erfolgt, die nun als oberster Eintrag auf dem Stack steht. Vorher werden noch die Rücksprungadresse und $a_1, \ldots, a_n$ bzw. $f_1, \ldots, f_n$ vom Stack genommen (Abbildung 3.2.2-2 (c)).

Offensichtlich kann innerhalb der Prozedur ein Zugriff auf ihre lokalen Variablen $v_1, \ldots, v_m$ und $f_1, \ldots, f_n$ ***relativ zum BP-Wert erfolgen***: $v_1, \ldots, v_m$ haben einen negativen, $f_1, \ldots, f_n$ einen positiven Offset zu BP. Dabei ist es unerheblich, wo genau innerhalb des Stacks der Aktivierungsrecord dieses Prozeduraufrufs liegt oder in welcher Rekursionstiefe der Prozeduraufruf erfolgte.

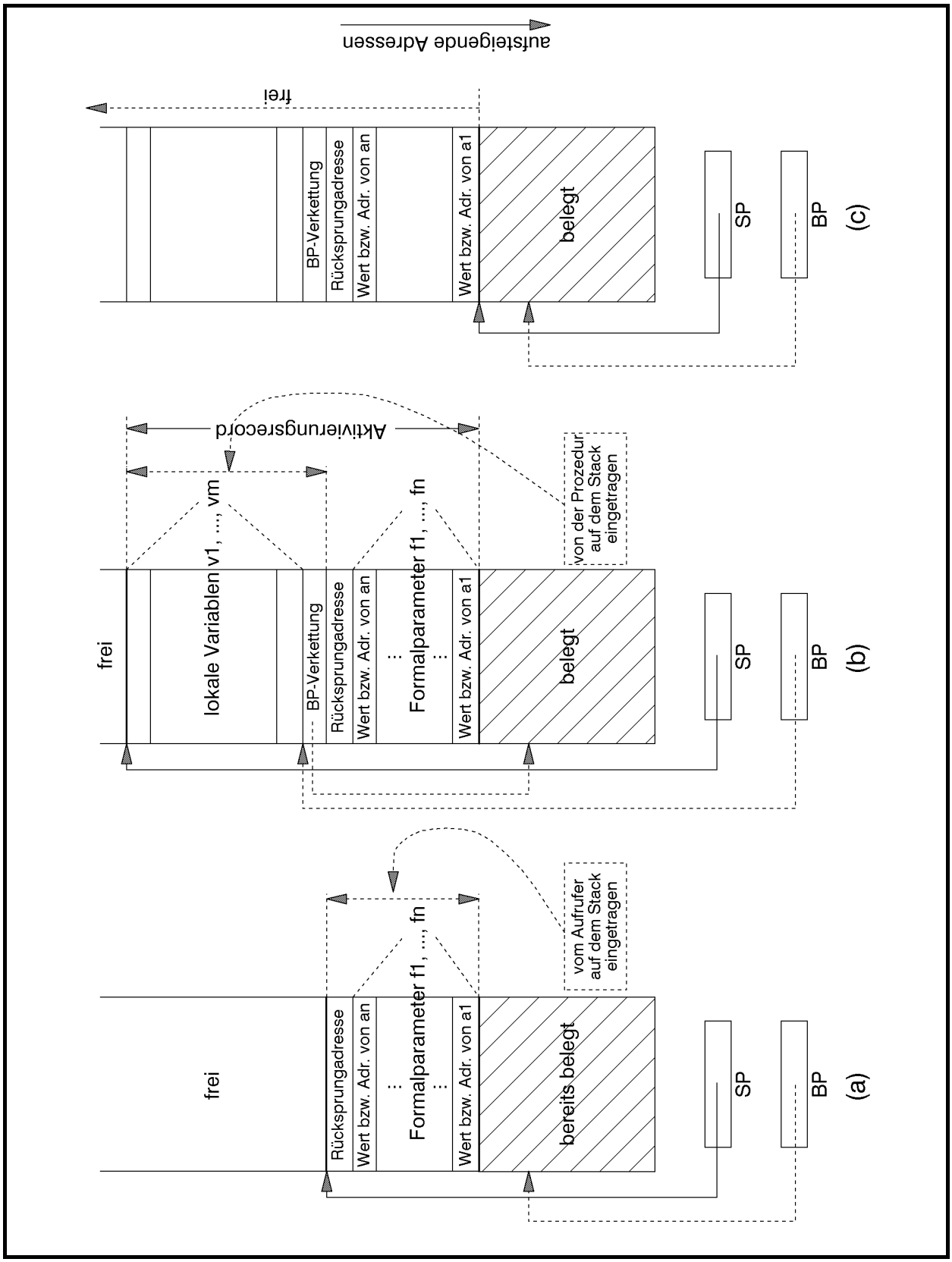


Abbildung 3.2.2-2: Stackbelegung bei einem Unterprogrammsprung (keine globalen Variablen)

Der Aktivierungsrecord eines Prozeduraufrufs wird etwas komplexer, wenn eine Prozedur auf globale Datenobjekte zugreifen kann, die in Prozeduren liegen, die die aufgerufene Prozedur einschachteln (auf Variablen des globalen Datenbereichs des Hauptprogramms besteht grund-

sätzlich Zugriffsberechtigung, falls diese innerhalb der Prozedur überhaupt benannt werden können). ***Die aus Sicht einer Prozedur globalen Variablen sind in der Menge der globalen und lokalen Variablen des sie umschließenden Blocks enthalten. Diese globalen Variablen befinden sich also im statischen Datenbereich oder in Aktivierungsrecords*** (zu umschließenden Prozeduraufrufen)***, die zum Aufrufzeitpunkt der Prozedur bereits im Stack liegen.*** Dort haben sie jeweils einen festen Offset zum Anfang des statischen Datenbereichs bzw. relativ zum BP-Wert des zugehörigen Aktivierungsrecords.

Als Beispiel wird wieder das Programm in Abbildung 3.1.1-1 betrachtet. Dabei soll jetzt zusätzlich angenommen werden, dass `p_program` selbst eine Prozedur und nicht ein Hauptprogramm ist, so dass die lokalen Variablen von `p_program` im Stack und nicht im globalen Datenbereich verwaltet werden. Die statische Blockstruktur der Prozeduren kann man dann auf unterschiedliche Weise darstellen, z.B. als ineinandergeschachtelte Blöcke oder als gerichteten Graph, dessen Kantenrichtung die mögliche Aufrufreihenfolge anzeigt (Abbildung 3.2.2-3). Eine gerichtete Kante dieses Graphen in vertikaler Richtung beschreibt dabei einen möglichen Aufruf einer eingeschachtelten Prozedur, in horizontaler Richtung einen Aufruf auf derselben Schachtelungstiefe.

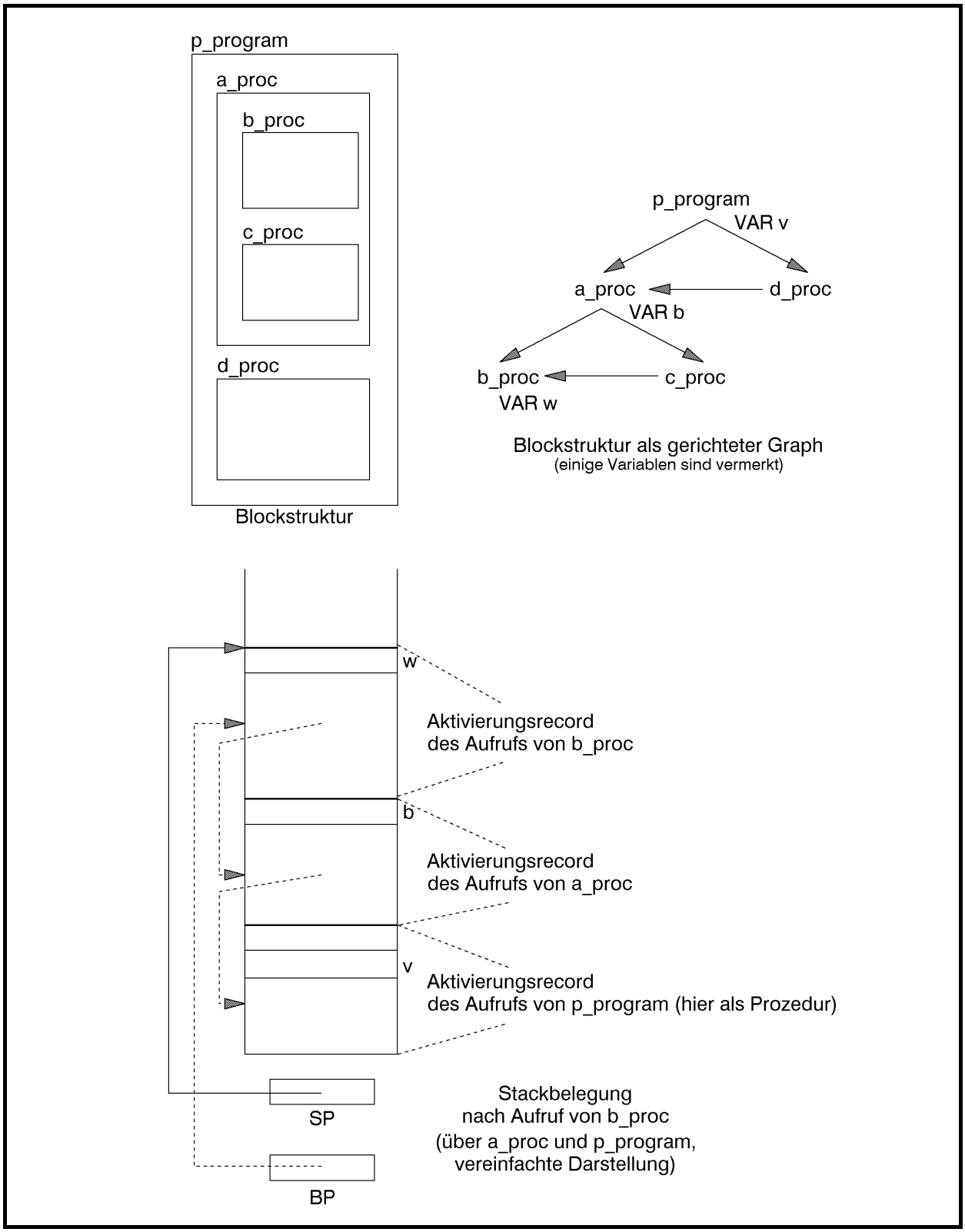


Abbildung 3.2.2-3: Blockstruktur (Beispiel aus Abbildung 3.1.1-1)

Der untere Teil von Abbildung 3.2.2-3 zeigt die Stackbelegung, wenn aus `p_program` heraus `a_proc` und von dort `b_proc` aufgerufen werden (diese Aufrufreihenfolge vereinfacht etwas die in Abbildung 3.1.1-1 angegebene Aufrufreihenfolge, in der ein Aufruf von `a_proc` aus `d_proc` heraus erfolgt; für den Augenblick soll diese etwas vereinfachte Aufrufreihenfolge angenommen werden). In Abbildung 3.2.2-3 sind exemplarisch Variablen wiedergegeben, auf

die beim Aufruf von `b_proc` zugegriffen werden kann: die Variable `w` (exakter: das mit `w` bezeichnete Datenobjekt d7) ist lokal in `b_proc`, also im Aktivierungsrecord, der über den gegenwärtigen BP-Wert angesprochen wird; die Variable `b` ist global für `b_proc` und lokal für `a_proc`; sie liegt im Aktivierungsrecord des `a_proc`-Aufrufs. Entsprechend befindet sich die Variable `v` im Aktivierungsrecord des `p_program`-Aufrufs.

Für jede Variable ist zur Übersetzungszeit aufgrund der Regeln über die Gültigkeitsbereich von Bezeichnern klar, über wieviele Stufen in der Schachtelungstiefe der Blockstruktur nach außen gegangen werden muss, um sie zu erreichen: 0 Stufen für lokale Variablen, *d* Stufen, falls die Variable im *d*-ten umfassenden Block liegt; hierbei wird von innen nach außen gezählt. Beispielsweise befindet sich für `b_proc` die Variable `w` null Stufen, die Variable `b` eine Stufe und die Variable `v` zwei Stufen weiter außen in der Blockstruktur. Für `a_proc` liegt die Variable `b` null Stufen und die Variable `v` nur eine Stufe weiter außen. Als offensichtliche Regel gilt: Liegt ein Datenobjekt für eine Prozedur innerhalb der Blockstruktur $d \geq 0$ Stufen weiter außen, so liegt es für eine direkt eingebettete Prozedur $d+1$ Stufen weiter außen, falls überhaupt eine Zugriffsmöglichkeit auf das Datenobjekt besteht.

Es sei noch einmal hervorgehoben, dass die im folgenden beschriebene Implementierung nur eine Möglichkeit aufzeigt, um das Konzept lokaler und globaler Variablen umzusetzen.

Für eine während der Laufzeit effiziente Handhabung des Zugriffs auf Datenobjekte in umfassenden Blöcken generiert der Compiler Code, der bei Eintritt in eine Prozedur (`BEGIN`-Statement) ausgeführt wird und ***der im Aktivierungsrecord der Prozedur zusätzliche Adressverweise auf diejenigen Aktivierungsrecords erzeugt, die aus Sicht der gerufenen Prozedur globale Datenobjekte in anderen Aktivierungsrecords enthalten.*** Die Menge dieser Adressverweise heißt **Display**. Alle Display-Informationen liegen im Aktivierungsrecord zwischen der BP-Rückwärtsverkettung und den lokalen Variablen. Wenn sich mehrere globale Datenobjekte in demselben umfassenden Block befinden, wird nur eine einzige Display-Information für die Datenobjekte dieses Blocks festgehalten. ***Es wird jeweils ein Verweis auf die BP-Rückwärtsverkettung innerhalb eines Aktivierungsrecords als jeweilige Display-Information genommen.*** Die Display-Informationen einer gerufenen Prozedur können während der Laufzeit bei Eintritt in die Prozedur aus der Display-Information des Aufrufers und der statischen Blockstruktur gewonnen werden. Da alle Blöcke eines Programms Zugriff auf den globalen Datenbereich des Programms besitzen, ist ein Verweis auf diesen Bereich als Teil des Displays überflüssig. Den so erweiterten Aufbau eines Aktivierungsrecords für einen Prozeduraufruf zeigt Abbildung 3.2.2.-4.

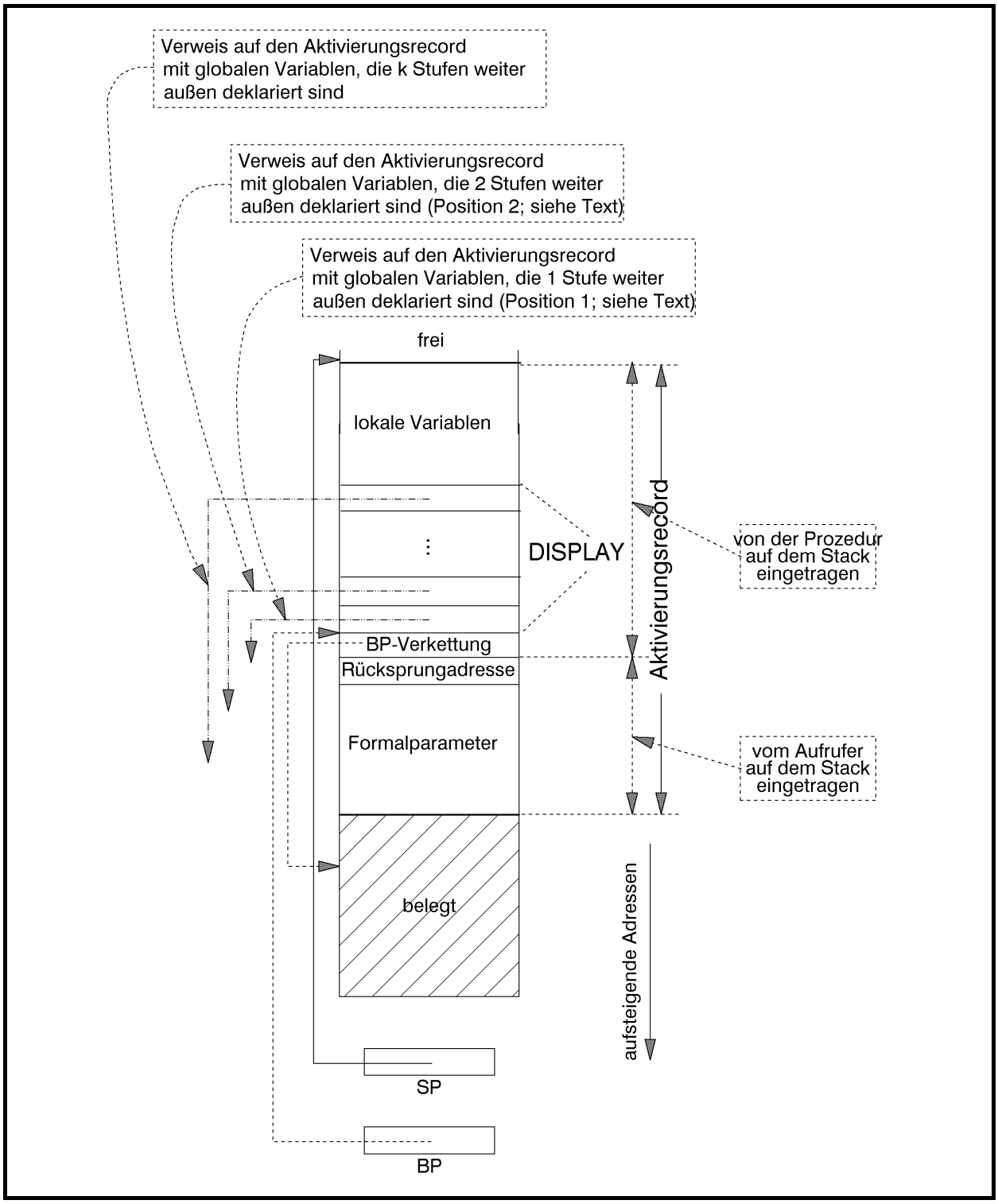


Abbildung 3.2.2-4: Um Display-Informationen erweiterter Aktivierungsrecord

Zur dynamischen Erzeugung der Display-Information während der Laufzeit wird jedem Prozeduraufruf als zusätzlicher verdeckter Parameter der BP-Wert mitgegeben, der die Lage desjenigen Aktivierungsrecords im Stack bestimmt, der ***zum nächsten umfassenden Prozeduraufruf*** (also in der Blockstruktur um eine Stufe weiter nach außen) gehört; er wird im folgenden als ***direkt übergeordneter Aktivierungsrecord*** bezeichnet. Dieser Adressverweis auf den direkt übergeordneten Aktivierungsrecord wird oberhalb der BP-Verkettung im Aktivierungsrecord der gerufenen Prozedur abgelegt (Position 1 in Abbildung 3.2.2-4). Er wird immer er-

zeugt, ganz gleich, ob auf dortige lokale Variablen zugegriffen wird oder nicht; falls es keinen übergeordneten Aktivierungsrecord gibt (in der Blockstruktur befindet sich dann die Prozedur in der äußersten Schachtelungstiefe), wird der Wert `NIL` erzeugt. Falls zusätzlich die lokalen Variablen in der nächsten Stufe nach außen innerhalb der Blockstruktur verwendet werden, erfolgt ein weiterer Display-Eintrag (Position 2 in Abbildung 3.2.2-4). Dieser kann aus dem direkt übergeordneten Aktivierungsrecord genommen werden: er liegt dort an der in Abbildung 3.2.2-4 mit Position 1 bezeichneten Stelle. Entsprechend werden eventuell weitere Display-Einträge aus dem direkt übergeordneten Aktivierungsrecord erzeugt. Aus wievielen Adressverweisen die Display-Information besteht und damit wo innerhalb eines Aktivierungsrecords relativ zum BP-Wert die lokalen Variablen liegen, ergibt sich aus der statischen Blockstruktur und liegt während der Übersetzungszeit fest.

Zur Erläuterung des Verfahrens wird noch einmal die Prozedur

```
PROCEDURE proc (f1 : ...; ...; fn : ...;);
  { lokale Variablen von proc: }
  VAR v1 : ...;
      ...
      vm : ...;

  BEGIN { proc }
    ...
  END   { proc };
```

herangezogen.

Die lokale Variable $\mathtt{v}_i$ belege $l_{\mathtt{proc},\mathtt{v}_i}$ viele Bytes. Zusätzlich habe die Prozedur `proc` nun Zugriff auf globale Variablen in d umfassenden Blöcken bzw. Stufen (gezählt von innen nach außen; lokale Variablen befinden sich auf Stufe 0). Dann werden zur Laufzeit d Display-Informationen, nämlich Adressverweise auf die Aktivierungsrecords der d umfassenden Blöcke, erzeugt. Der gesamte Display innerhalb des Aktivierungsrecords von `proc` belegt $\Delta_{\mathtt{proc}} = d \cdot a$ viele Bytes, wobei für eine Adresse a viele Bytes verwendet werden. Der (negativ zu nehmende) Offset $d_{\mathtt{proc},\mathtt{v}_i}$ der Variablen $\mathtt{v}_i$ relativ zum BP-Wert berechnet sich zu:

$$d_{\mathtt{proc},\mathtt{v}_1} = \Delta_{\mathtt{proc}} + l_{\mathtt{proc},\mathtt{v}_1},$$
$$d_{\mathtt{proc},\mathtt{v}_i} = d_{\mathtt{proc},\mathtt{v}_{i-1}} + l_{\mathtt{proc},\mathtt{v}_i} \quad \text{für } i = 2, ..., m.$$

Die Adresse der Variablen $\mathtt{v}_i$ lautet dann $\mathtt{BP} - d_{\mathtt{proc},\mathtt{v}_i}$.

Ist `v` eine aus `proc` global zugreifbare Variable in einem umfassenden Block `global_proc` auf Stufe $d' \leq d$ und hat dort den den Offset $d_{\mathtt{global\_proc},\mathtt{v}}$, so lautet die Adresse von `v`:

(BP- $d' \cdot a$)^- $d_{\texttt{global\_proc,v}}$.

Das Verfahren wird am Beispiel der Abbildungen 3.1.1-1 und 3.2.2-3 erläutert. Wieder wird angenommen, dass `p_program` selbst eine Prozedur auf äußerster Schachtelungstiefe der Blockstruktur ist. Weiter wird angenommen, dass ein Datenobjekt vom Typ `INTEGER` und ein Datenobjekt mit Adresstyp jeweils genau zwei Bytes belegen. Während der Übersetzungsphase hat der Compiler für jede Variable eine Reihe von Informationen erzeugt, z.B. auf welche Variablen aus einer Prozedur zugegriffen werden kann und auf welcher Stufe (nach außen gezählt) sie sich aus Sicht der jeweiligen Prozedur befindet. Insbesondere lässt sich die Adresse einer Variablen berechnen, jedoch nur relativ zum BP-Wert (im Aktivierungsrecord) des aktuellen Prozeduraufrufs.

`p_program` 1 Display-Eintrag = `NIL`: $\Delta_{\texttt{p\_program}} = 2$

`v` Stufe 0, $l_{\texttt{p\_program,v}} = 2$, $d_{\texttt{p\_program,v}} = \Delta_{\texttt{p\_program}} + l_{\texttt{p\_program,v}} = 4$,

Adresse `BP` – $d_{\texttt{p\_program,v}}$ = `BP-4`

`w` Stufe 0, $l_{\texttt{p\_program,w}} = 2$, $d_{\texttt{p\_program,w}} = d_{\texttt{p\_program,v}} + l_{\texttt{p\_program,w}} = 6$,

Adresse `BP` – $d_{\texttt{p\_program,w}}$ = `BP-6`

`a_proc` 1 Display-Eintrag (in den Aktivierungsrecord von `p_program`): $\Delta_{\texttt{a\_proc}} = 2$

`a` Stufe 0, $l_{\texttt{a\_proc,a}} = 2$, $d_{\texttt{a\_proc,a}} = \Delta_{\texttt{a\_proc}} + l_{\texttt{a\_proc,a}} = 4$

Adresse `BP` – $d_{\texttt{a\_proc,a}}$ = `BP-4`

`b` Stufe 0, $l_{\texttt{a\_proc,b}} = 2$, $d_{\texttt{a\_proc,b}} = d_{\texttt{a\_proc,a}} + l_{\texttt{a\_proc,b}} + = 6$,

Adresse `BP` – $d_{\texttt{a\_proc,b}}$ = `BP-6`

`v` Stufe 1, Adresse `(BP-1*2)^-` $d_{\texttt{p\_program,v}}$ = `(BP-2)^-4`

`w` Stufe 1, Adresse `(BP-1*2)^-` $d_{\texttt{p\_program,w}}$ = `(BP-2)^-6`

`b_proc` 2 Display-Einträge (in die Aktivierungsrecords von `a_proc` und von `p_program`): $\Delta_{\texttt{b\_proc}} = 4$

`a` Stufe 0, $l_{\texttt{b\_proc,a}} = 2$, $d_{\texttt{b\_proc,a}} = \Delta_{\texttt{b\_proc}} + l_{\texttt{b\_proc,a}} = 6$,

Adresse `BP` – $d_{\texttt{b\_proc,a}}$ = `BP-6`

`i` Stufe 0, $l_{\texttt{b\_proc,i}} = 2$, $d_{\texttt{b\_proc,i}} = d_{\texttt{b\_proc,a}} + l_{\texttt{b\_proc,i}} = 8$,

Adresse `BP` – $d_{\texttt{b\_proc,i}}$ = `BP-8`

`w` Stufe 0, $l_{\texttt{b\_proc,w}} = 2$, $d_{\texttt{b\_proc,w}} = d_{\texttt{b\_proc,i}} + l_{\texttt{b\_proc,i}} = 10$,

Adresse `BP` – $d_{\texttt{b\_proc,w}}$ = `BP-10`

`v` Stufe 2, Adresse `(BP-2*2)^-` $d_{\texttt{p\_program,v}}$ = `(BP-4)^-4`

`b` Stufe 1, Adresse `(BP-1*2)^-` $d_{\texttt{a\_proc,b}}$ = `(BP-2)^-6`

`c_proc` 2 Display-Einträge(in die Aktivierungsrecords von `a_proc` und von `p_program`): $\Delta_{\texttt{c\_proc}} = 4$

a Stufe 0, $l_{\texttt{c\_proc,a}} = 2$, $d_{\texttt{c\_proc,a}} = \Delta_{\texttt{c\_proc}} + l_{\texttt{c\_proc,a}} = 6$,

Adresse `BP` – $d_{\texttt{c\_proc,a}}$ = `BP-6`

v Stufe 0, $l_{\texttt{c\_proc,v}} = 2$, $d_{\texttt{c\_proc,v}} = d_{\texttt{c\_proc,a}} + l_{\texttt{c\_proc,v}} = 8$,

Adresse `BP` – $d_{\texttt{c\_proc,v}}$ = `BP-8`

w Stufe 2, Adresse `(BP-2*2)^-` $d_{\texttt{p\_program,w}}$ = `(BP-4)^-6`

b Stufe 1, Adresse `(BP-1*2)^-` $d_{\texttt{a\_proc,b}}$ = `(BP-2)^-6`

`d_proc` 1 Display-Eintrag (in den Aktivierungsrecord von `p_program`): $\Delta_{\texttt{d\_proc}} = 2$

v Stufe 0, $l_{\texttt{d\_proc,v}} = 2$, $d_{\texttt{d\_proc,v}} = \Delta_{\texttt{d\_proc}} + l_{\texttt{d\_proc,v}} = 4$,

Adresse `BP` – $d_{\texttt{d\_proc,v}}$ = `BP-4`

w Stufe 0, $l_{\texttt{d\_proc,w}} = 2$, $d_{\texttt{d\_proc,w}} = d_{\texttt{d\_proc,v}} + l_{\texttt{d\_proc,v}} = 6$,

Adresse `BP` – $d_{\texttt{d\_proc,w}}$ = `BP-6`.

Abbildung 3.2.2-5 zeigt die Stackbelegung, einschließlich der Display-Informationen, nach Eintritt in die Prozedur `b_proc`. Es wird jetzt die etwas komplexere Aufrufreihenfolge `p_program` - `d_proc` - `a_proc` - `c_proc` - `b_proc` (vgl. Abbildungen 3.1.1-1 und 3.2.2-3) angenommen. In den folgenden Abbildungen werden Einzeleinträge innerhalb des Stacks wieder als gleich große Kästchen gezeichnet, ohne die interne Speicherdarstellung der einzelnen Datentypen zu berücksichtigen.

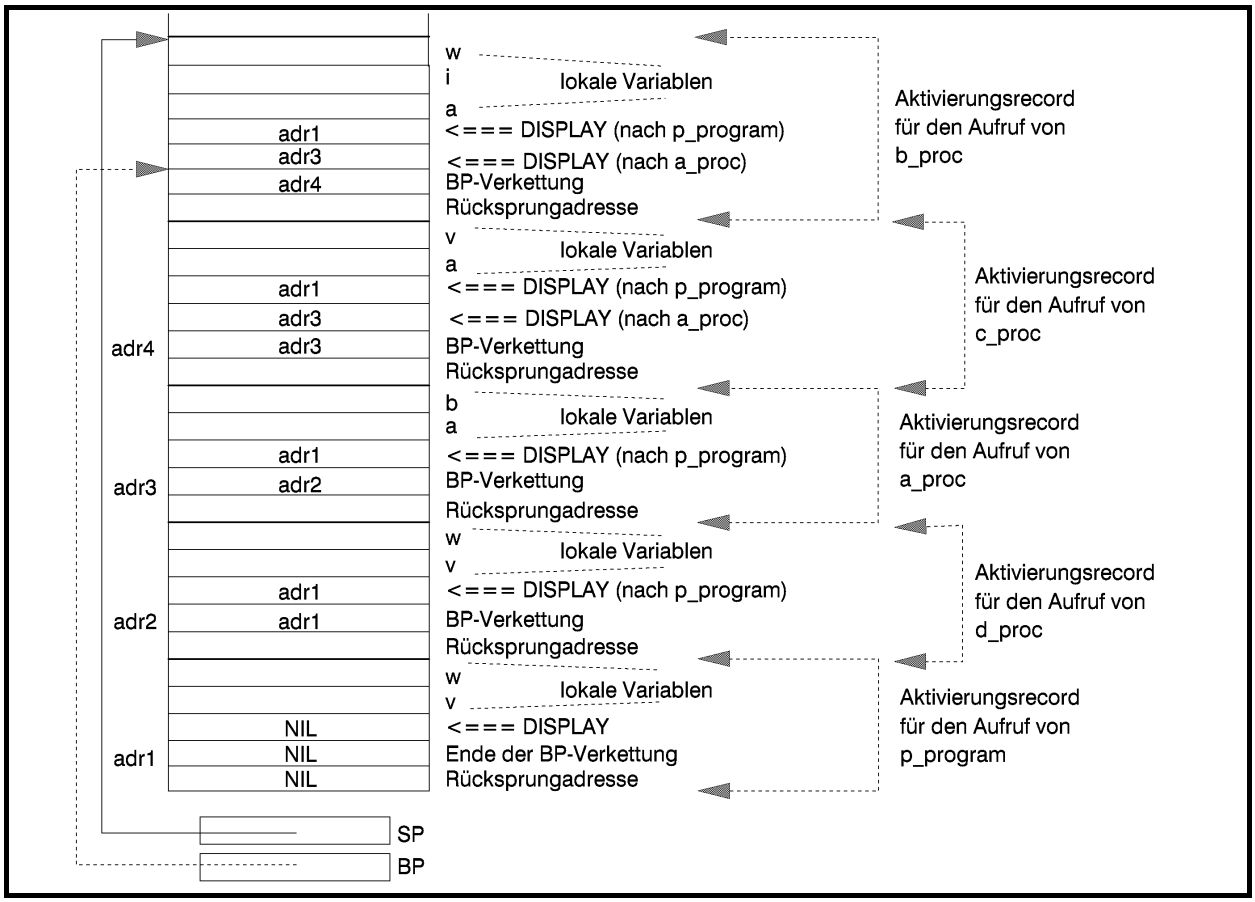


Abbildung 3.2.2-5: Beispiel einer Stackbelegung mit Display-Informationen

3.2.3 Mehrfachverwendbarkeit von Prozedurcode

Im Prozessmodell eines Betriebssystems kann eine Prozedur logisch in unterschiedlichen virtuellen Adressräumen liegen, und die zugehörigen Prozesse laufen parallel durch die Prozedur, ohne dass es zu einer gegenseitigen Beeinflussung kommt. Dafür sorgt ja gerade die Kontrolle des Betriebssystems. Das beschriebene Prozedurkonzept gewährleistet darüber hinaus, dass bei Einhaltung der im folgenden beschriebenen Programmierregeln auch unterschiedliche Threads innerhalb desselben Prozesses parallel durch den Code derselben Prozedur laufen können. Hierbei befindet sich die Prozedur nur ein einziges Mal im virtuellen Adressraum des Prozesses, zu dem die Threads gehören. Mit Parallelität ist hier gemeint, dass ein Thread in den Code einer Prozedur gelaufen ist und ein weiterer Thread den Code der Prozedur betritt, bevor der erste Thread die Prozedur wieder verlassen hat. Es muss natürlich sichergestellt sein, dass auch hier keine (ungewollte) gegenseitige Beeinflussung der Threads entsteht.

Unter folgenden Voraussetzung ist es unterschiedlichen Threads erlaubt, eine im virtuellen Adressraum des Prozesses nur ein einziges Mal vorhandene Prozedur parallel zu durchlaufen:

- ***jeder Thread verfügt*** neben seinem virtuellen Registersatz ***über einen eigenen Stack*** (für die Einrichtung eines „privaten“ Stacks für jeden Thread sorgt das Betriebssystem); als Stack bei der Umsetzung des Prozedurkonzepts wird immer der private Stack des gerade aktiven Threads genommen
- ***die Prozedur verwendet*** keine globalen Daten, sondern ***ausschließlich lokale Datenobjekte*** (Formalparameter und innerhalb der Prozedur deklarierte lokale Variablen).

Code, der von mehreren Threads (auch unterschiedlicher Prozesse) parallel durchlaufen werden kann, bezeichnet man als **parallel mehrfach-benutzbar (ablauf-invariant, reentrant)**.

Die Forderung, dass parallel mehrfach-benutzbarer Code keine globalen Datenobjekte verwendet, liegt auf der Hand: Setzt ein Thread den Wert einer globalen Variablen, so kann er nicht sicher sein, dass diese Variable beim nächsten Zugriff den von ihm gesetzten Wert noch enthält. Inzwischen kann ja ein Threadwechsel stattgefunden haben, und ein anderer Thread hat den Variablenwert verändert.

Der vom Compiler erzeugte Code, der bei Eintritt in die Prozedur durchlaufen wird, generiert einen Aktivierungsrecord mit den lokalen Datenobjekten der Prozedur auf dem Stack des aktiven Threads. Der Zugriff auf alle diese Datenobjekte erfolgt relativ zum Basispointer BP. Betritt ein weiterer Thread mit seinem eigenen Stack die Prozedur, wird ein weiterer Aktivierungsrecord nun im Stack dieses Threads erzeugt. Analog der Erzeugung einer weiteren Inkarnation lokaler Variablen bei jedem rekursiven Prozeduraufruf, erhält jeder Thread daher

seinen eigenen Satz lokaler Variablen (Abbildung 3.2.3-1). Natürlich muss das Betriebssystem bei einem Threadwechsel neben den jeweiligen virtuellen Registersätzen auch die privaten Stacks der Threads umschalten.

Prinzipiell ist es unerheblich, ob die Threads, die durch den parallel mehrfach-benutzbaren Code einer Prozedur laufen, demselben oder unterschiedlichen Prozessen angehören. Es ist daher dem Betriebssystem möglich, einen derartigen Code physisch auch nur ein einziges Mal in den Arbeitsspeicher zu laden, obwohl er logisch gleichzeitig eventuell in unterschiedlichen virtuellen Adressräumen und dort an unterschiedlichen virtuellen Adressen liegt. Man spricht hier von Code-Sharing. Um den Code nur einmal im Arbeitsspeicher zu halten und zugreifbar von unterschiedlichen Prozessen zu haben, werden spezielle Funktionen des Betriebssystems benötigt. Es muss beispielsweise verhindern, dass der von mehreren Prozessen benutzte Code aus dem Arbeitsspeicher des Rechners ausgelagert wird, solange noch ein Prozess auf ihn zugreift, auch wenn ein Prozess, der den Code mit anderen zusammen benutzt hat, beendet wird. Das Thema der Realisierung des Code-Sharings wird im Rahmen der Betriebssysteme behandelt.

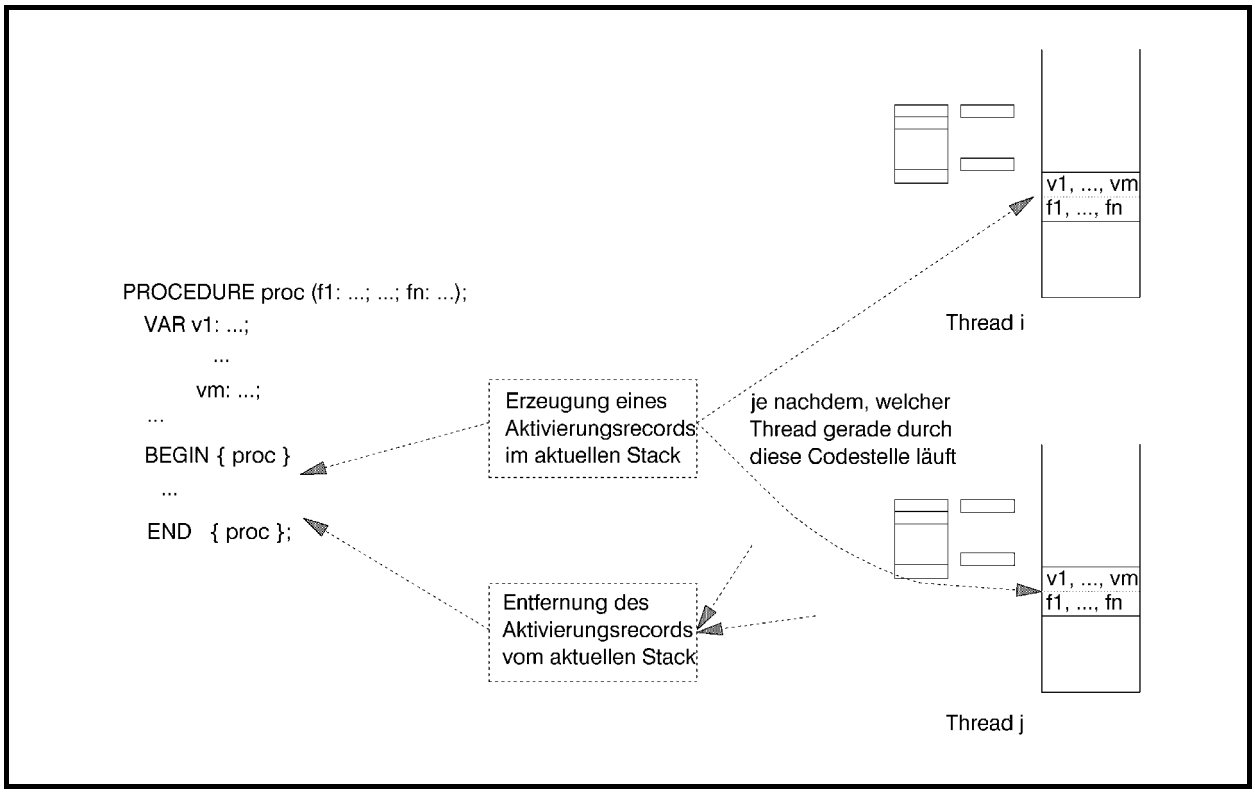


Abbildung 3.2.3-1: Parallel mehrfach-benutzbarer Code

3.3 Methoden in der Objektorientierten Programmierung

In diesem Kapitel werden einige Implementierungskonzepte im Zusammenhang mit der Handhabung von Methoden in der Objektorientierten Programmierung (OOP) behandelt, die das Verständnis der Unterscheidung von statischen, virtuellen und dynamischen Methoden und die Rolle von Konstruktoren und Destruktoren fördern können. Wieder wird die Sprache Object Pascal herangezogen, und zwar wird das durch `OBJECT` festgelegte Objektmodell genommen (die Begründung für die Wahl dieses Modells findet sich in Kapitel 2.2.5); die Ausführungen des vorliegenden Kapitels sind jedoch auch im durch `CLASS` festgelegten Objektmodells gültig. Die wichtigsten Merkmale der OOP, nämlich die (einfache) Vererbung und die Polymorphie von Methoden, findet man als Sprachkonzepte in Pascal in sehr „handlicher" Weise. Diese sollen im folgenden an Beispielen erläutert werden.

3.3.1 Statische Methoden

Das erste Beispiel ([HOF]) behandelt ein Modell[7], dessen Objekte ***Blöcke*** genannt werden. Jeder Block besteht aus zwei Teilen, einem ***Kopfteil*** und einem ***Nutzteil***, der ***Einzeleinträge*** enthält. Der Kopfteil selbst ist in vier Komponenten gegliedert. Sie enthalten Informationen, die Auskunft über die Anzahl von Einzeleinträgen im Nutzteil geben (in der Komponente `anz`) bzw. die Adressverweise auf weitere Blöcke darstellen (in den Komponenten `vorg`, `last` und `next`). Es gibt zwei Typen von Blöcken, die sich durch das Layout ihres Nutzteils unterscheiden (der Kopfteil ist bei allen Blöcken gleichartig strukturiert): Ein Block vom Typ ***Datenblock*** enthält in seinem Nutzteil Einträge der Form (*k*, *data*); die Datentypen und die Bedeutung der Einträge *k* und *data* spielen an dieser Stelle keine Rolle. Die Anzahlen der Einträge im Nutzteil verschiedener Datenblöcke können sich unterscheiden; sie sind jedoch durch den Wert eines globalen Parameters `_2v` beschränkt. Ein Block vom Typ ***Indexblock*** enthält in seinem Nutzteil einen Verweis (in Form einer Adresse) auf einen anderen Block und weitere Einträge der Form (*k*, *p*); die genauen Bedeutungen der Einträge *k* und *p* spielen hier ebenfalls keine Rolle. Auch hier können die Anzahlen der Einträge im Nutzteil verschiedener Indexblöcke unterschiedlich, aber beschränkt durch den Wert eines globalen Parameters `_2u` sein. Die Größen eines Daten- und eines Indexblocks können sich unterscheiden; gemeinsam ist ihnen nur, dass sie einen Kopfteil und einen Nutzteil besitzen.

---

7 Das Modell beschreibt die Speicherung von Datensätzen in Form höhenbalancierter Bäume. Auf die Datensätze sind Zugriffe über Primärschlüssel in der Weise möglich, dass die Zugriffszeit auf jeden Datensatz die gleiche Zeit in Anspruch nimmt.

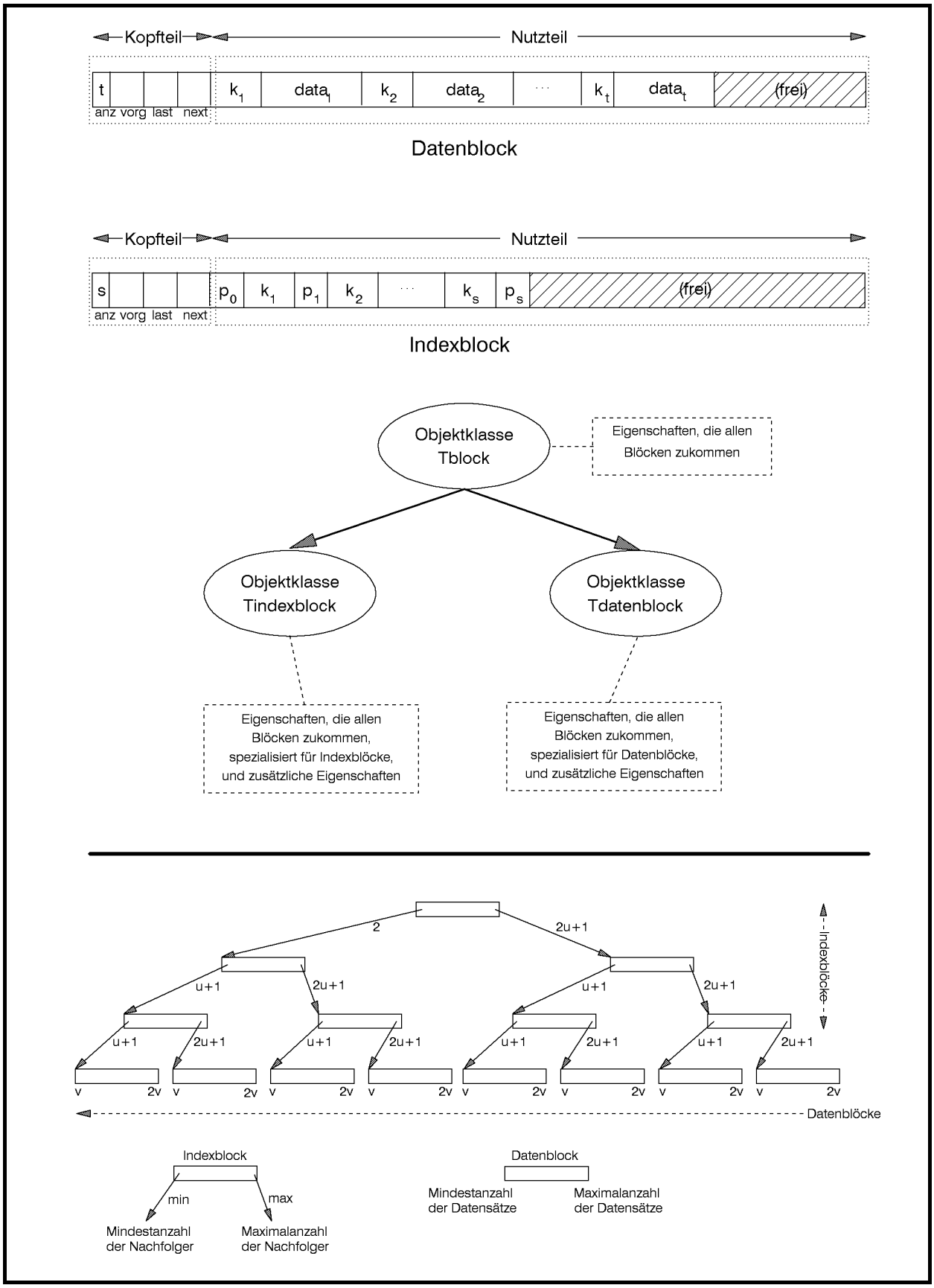


Abbildung 3.3.1-1: Blöcke (Beispiel)

Datenblöcke und Indexblöcke sind über die Adressverweise im Kopfteil logisch miteinander verknüpft (die Gesamtstruktur zeigt der untere Teil von Abbildung 3.3.1-1; sie ist für das Beispiel hier jedoch ohne Bedeutung).

Es ist also sinnvoll, Objektklassen `Tdatenblock` und `Tindexblock` zu definieren, die sich aus einer Objektklasse `Tblock` ableiten. In der Objektklasse `Tblock` wird der Kopfteil eines Blocks definiert; die unterschiedlichen Layouts der Daten- bzw. Indexblöcke werden zusätzlich in der Objektklasse `Tdatenblock` bzw. `Tindexblock` festgelegt. Die beiden Typen von Blöcken sind im oberen Teil von Abbildung 3.3.1-1 zu sehen.

Neue Blöcke können erzeugt und in die logische Struktur eingefügt bzw. aus der logischen Struktur entfernt und vernichtet werden. Einzeleinträge können in einen Datenblock eingefügt bzw. aus ihm entfernt werden. Das gleiche ist für Einzeleinträge in Indexblöcken möglich. Zusätzlich sollen die Inhalte eines Blocks (Kopfteil und Einzeleinträge in Datenblöcken bzw. Indexblöcken) angezeigt werden können.

In der Objektklasse `Tblock` werden also die Komponenten des Kopfteils und die Methoden

`init`	zur Erzeugung eines neuen Blocks und Initialisierung des Kopfteils
`insert`	zum Einfügen eines Blocks (Daten- oder Indexblock) in die Gesamtstruktur
`delete`	zum Entfernen eines Blocks aus der Gesamtstruktur und zur „Auflösung“ des Blocks
`display`	zum Anzeigen der Gesamtstruktur

definiert. Für die Objektklassen `Tdatenblock` bzw. `Tindexblock` gibt es neben den Komponenten des jeweiligen Nutzteils die Methoden

`init`	zur Initialisierung des jeweiligen Nutzteils
`insert_entry`	zum Einfügen eines Eintrags in den jeweiligen Nutzteil
`delete_entry`	zum Entfernen eines Eintrags aus dem jeweiligen Nutzteil
`display`	zum Anzeigen des jeweiligen Nutzteils

Folgende Deklarationen und Implementierungen könnten dann vorgenommen werden (einige Programmteile sind lediglich durch Punkte angedeutet).

```
CONST _2u             = ...;
      _2v             = ...;
      k_init_value    = ...;
      data_init_value = ...;

TYPE Tdata_typ        = ... ;
     Tkey_typ         = ...;
```

```
Tdata_entry_typ  = RECORD
                     k    : Tkey_typ;
                     data : Tdata_typ;
                   END;
Tdata_sect_typ   = ARRAY [1.._2v] OF Tdata_entry_typ;

Tindex_entry_typ = RECORD
                     k : Tkey_typ;
                     p : Pblock_ptr;
                   END;
Tindex_sect_typ  = ARRAY [1.._2u] OF Tindex_entry_typ;


Pblock = ^Tblock;
Tblock = OBJECT
           anz  : INTEGER;
           vorg : Pblock;
           last : Pblock;
           next : Pblock;
           { Methoden von block }
           PROCEDURE init;
           PROCEDURE insert;
           PROCEDURE delete;
           PROCEDURE display;
         END;

 Pdatenblock = ^Tdatenblock;
 Tdatenblock = OBJECT (Tblock)
                 data : Tdata_sect_typ;
                 { Methoden von Tdatenblock }
                 PROCEDURE init;
                 PROCEDURE insert_entry
                               (new_entry : Tdata_entry_typ);
                 PROCEDURE delete_entry
                              (del_entry : Tdata_entry_typ);
                 PROCEDURE display;
               END;

 Pindexblock = ^Tindexblock;
 Tindexblock = OBJECT (Tblock)
                 p_0   : Pblock;
                 index : Tindex_sect_typ;
                 { Methoden von indexblock }
                 PROCEDURE init;
                 PROCEDURE insert_entry
                              (new_entry: Tindex_entry_typ);
```

```
                      PROCEDURE delete_entry
                                  (del_entry: Tindex_entry_typ);
                      PROCEDURE display;
                    END;

   ...
{ ---------------------------------------------------------------- }

{ Implementierung der Methoden von Tblock }

PROCEDURE Tblock.init;

{ Initialisierung des Kopfteils }
  BEGIN { Tblock.init }
    anz  := 0;
    vorg := NIL;
    last := NIL;
    next := NIL;
  END   { Tblock.init };

PROCEDURE Tblock.insert;
  ...
PROCEDURE Tblock.delete;
  ...
PROCEDURE Tblock.display;
   ...
{ ---------------------------------------------------------------- }

{ Implementierung der Methoden von Tdatenblock }

PROCEDURE Tdatenblock.init;

  VAR idx : INTEGER;

  BEGIN { Tdatenblock.init }
    { Kopfteil initialisieren: }
    INHERITED init;
    { Nutzteil initialisieren: }
    FOR idx := 1 TO _2v DO
      BEGIN
        data[idx].k    := k_init_value;
        data[idx].data := data_init_value
      END
  END   { Tdatenblock.init };
```

```
PROCEDURE Tdatenblock.insert_entry
               (new_entry : Tdata_entry_typ);
   ...
PROCEDURE Tdatenblock.display;
   ...
PROCEDURE Tdatenblock.delete_entry
               (del_entry : Tdata_entry_typ);
   ...
{ ---------------------------------------------------------------- }

{ Implementierung der Methoden von Tindexblock }

PROCEDURE Tindexblock.init;

  VAR idx : INTEGER;

  BEGIN { Tindexblock.init }
    { Kopfteil initialisieren: }
    INHERITED init;
    { Nutzteil initialisieren: }
    p_0 := NIL;
    FOR idx := 1 TO _2u DO
      BEGIN
        index[idx].k := k_init_value;
        index[idx].p := NIL
      END
  END   { Tindexblock.init };

PROCEDURE Tindexblock.insert_entry
               (new_entry : Tindex_entry_typ);
   ...
PROCEDURE Tindexblock.delete_entry
               (del_entry : Tindex_entry_typ);
   ...
PROCEDURE Tindexblock.display;
   ...
{ ---------------------------------------------------------------- }
```

Objekte (Datenblock bzw. Indexblock) mit den Variablen

```
VAR d_block : Tdatenblock;
    i_block : Tindexblock;
```

werden dann beispielsweise initialisiert durch

```
                                  }
  d_block.init;
```

```
i_block.init;
```

Der Aufruf

```
INHERITED init;
```

in den jeweiligen `init`-Methoden des Daten- bzw. Indexblocks führt die geerbte `init`-Methode des Objekttyps `Tblock` aus, die den Kopfteil eines Blocks initialisiert.

Innerhalb der Definition einer Methode sind die Bezeichner für Komponenten des zugehörigen Objekttyps bekannt: Beispielsweise versorgt die Methode `block.init` die im Objekttyp `Tblock` definierten Komponenten eines Datenobjekts mit diesem Datentyp mit Anfangswerten. Die Definition einer Methode und des Objekttyps, der diese Methode enthält, liegen im selben **Gültigkeitsbereich**. Den Gültigkeitsbereich eines Bezeichners kann man auf den Modul (Unit) beschränken, der die Objekttypdeklaration enthält (siehe Kapitel 2.2.5): Eine Objekttypdeklaration und ihre Komponenten und Methoden sind in anderen Units bekannt, wenn die Objekttypdeklaration im Interfaceteil einer Unit steht. Um die Gültigkeit einiger Bezeichner innerhalb einer Objekttypdeklaration nur auf den Modul (Programm oder Unit) zu beschränken, der die Deklaration enthält, kann man diesen Bezeichnern das Schlüsselwort `PRIVATE` voranstellen. Dadurch sind diese `PRIVATE`-Bezeichner in anderen Moduln nicht bekannt. Folgende Bezeichner, die wieder allgemein öffentlich sein sollen, müssen mit dem Schlüsselwort `PUBLIC` eingeleitet werden. Von dieser Möglichkeit wird in den Beispielen in den folgenden Kapiteln Gebrauch gemacht.

Ein Vorteil der Vererbung wird deutlich: Wird das Layout bzw. eine Methode des Objekttyps `Tblock` geändert, so wird diese Änderung sofort auch in `Tdatenblock` und `Tindexblock` wirksam, ohne dass dort die Methoden angepasst zu werden brauchen. Die Änderungen des allgemeinen Objekttyps werden also an die spezielleren Nachkommen vererbt. Die Definition eines neuen Blocktyps, der ebenfalls aus Kopfteil und Nutzteil besteht, der sich von denjenigen der Daten- und Indexblöcke unterscheidet, ist ebenfalls ohne Änderung des bisherigen Quellcodes möglich. Ist der Name des neuen Objekttyps z.B. `Tneu_block`, so definiert

```
TYPE Tneu_block = OBJECT (Tblock)
                    { neuer Nutzteil: }
                   ...
                  { Methoden von Tneu_block, die sich auf
                    den neuen Nutzteil beziehen; für den
                    Zugriff auf den Kopfteil werden die
                    Methoden von Tblock genommen:       }
                   ...
                 END;
```

einen neuen Objekttyp.

Die so eingeführte Form der Vererbung nennt man **statische Vererbung**. Der Compiler kann zur Übersetzungszeit bereits alle weitergegebenen Eigenschaften von Objekttypen auflösen. Insbesondere wird jetzt bereits beim Aufruf einer Methode geprüft, ob diese für den entsprechenden Objekttyp definiert ist. Falls es zutrifft, erfolgt ein Unterprogrammsprung in den Objektcode der Methode. Ist keine Methode für den entsprechenden Objekttyp definiert, geht der Compiler zum direkt übergeordneten Objekttyp (der in Klammern hinter dem Schlüsselwort `OBJECT` steht) über und sucht hier nach der Definition der Methode. Ist die Suche auch hier erfolglos, wird die Suche nach dem Code der Methode auf den weiteren übergeordneten Objekttyp ausgedehnt, falls es ihn gibt. Erst wenn die Methode bis in die oberste Hierarchiestufe nicht gefunden wird, bricht der Compiler die Suche mit einer Fehlermeldung ab. Die Auflösung des Zugriffs auf Methoden während der Übersetzungszeit wird als **frühe Bindung** bezeichnet.

Ein wichtiger Randeffekt der statischen Vererbung von Methoden ist zu beachten. Wird in einem Objekttyp `To_1` eine Methode mit Bezeichner `xyz` definiert und ebenso in einem übergeordneten Objekttyp `To_2`, so wird bei Aufruf der Methode `xyz` mit einem Datenobjekt mit Objekttyp `To_1` die Methode `xyz` von `To_1` genommen. Ist andererseits eine Methode namens `meth` im übergeordneten Objekttyp `To_2`, aber nicht in `To_1` definiert, so wird bei Aufruf von `meth` mit einem Datenobjekt mit Objekttyp `To_1` die Methode `meth` aus `To_2` aktiviert. Verwendet `meth` seinerseits die Methode `xyz`, so wird bei Aufruf von `meth` die Methode `xyz` aus `To_2` genommen, auch wenn `meth` von einem Datenobjekt mit Objekttyp `To_1` aufgerufen wurde. Die Situation wird an folgendem Programmbeispiel und in der Abbildung 3.3.1-2 illustriert.

```
PROGRAM oop_stat;

TYPE To_2 = OBJECT
              { Komponenten von To_2:}
              flag : BOOLEAN;
              i    : INTEGER;
              { Methoden von To_2: }
              PROCEDURE init (wert : BOOLEAN;
                              x    : INTEGER);
              PROCEDURE del;
              PROCEDURE xyz;
              PROCEDURE meth;
            END;

     To_1 = OBJECT (To_2)
              { zusätzliche Komponenten von To_1:}
              j : INTEGER;
              { Methoden von To_1: }
              PROCEDURE init (f : BOOLEAN;
```

```
                              x : INTEGER;
                              y : INTEGER);
              PROCEDURE erase;
              PROCEDURE xyz;
            END;

{ ---------------------------------------------------------------- }
{ Implementierung der Methoden von To_2: }

PROCEDURE To_2.init (wert : BOOLEAN;
                     x    : INTEGER);
  BEGIN { To_2.init }
    flag := wert;
    i    := x
  END   { To_2.init };


PROCEDURE To_2.xyz;

  BEGIN { To_2.xyz }
    Writeln ('Methode xyz in o_2');
    Write ('     To_2-Objekt: ');
    IF flag = TRUE
    THEN Writeln ('flag = TRUE , i = ', i)
    ELSE Writeln ('flag = FALSE, i = ', i)
  END   { To_2.xyz };


PROCEDURE To_2.meth;
  BEGIN { To_2.meth }
    Writeln ('Methode meth');
    Writeln;
    xyz   { Aufruf der Methode xyz (aus To_2) };
  END   { To_2.meth };


PROCEDURE To_2.del;
  BEGIN { To_2.del }
    ...
  END   { To_2.del };

{ ---------------------------------------------------------------- }

{ Implementierung der Methoden von To_1: }

PROCEDURE To_1.init (f : BOOLEAN;
```

```
                    x : INTEGER;
                    y : INTEGER);
  BEGIN { To_1.init }
    flag := f;
    i    := x;
    IF flag
    THEN j := x
    ELSE j := x + y
  END   { To_1.init };


PROCEDURE To_1.erase;
  BEGIN { To_1.erase }
    ...
  END   { To_1.erase };


PROCEDURE To_1.xyz;
  BEGIN { To_1.xyz }
    Writeln ('Methode xyz in o_1');
    Writeln ('     o_1-Objekt: j = ', j);
    IF flag
    THEN Writeln( '                 flag = TRUE , i = ', i)
    ELSE Writeln( '                 flag = FALSE, i = ', i);
  END   { To_1.xyz };

{ ---------------------------------------------------------------- }

VAR o_1_objekt_a : To_1;
    o_1_objekt_b : To_1;
    o_2_objekt_1 : To_2;
    o_2_objekt_2 : To_2;


BEGIN { oop_stat }
  o_1_objekt_a.init (TRUE, 1, 100);
  o_1_objekt_b.init (FALSE, 2, 200);
  o_2_objekt_1.init (TRUE, 1000);
  o_2_objekt_2.init (FALSE, 2000);

  o_1_objekt_a.xyz  { <== Aufruf der Methode xyz aus To_1 };
  Writeln;
  o_1_objekt_b.xyz  { <== Aufruf der Methode xyz aus To_1 };
  Writeln;
  o_1_objekt_b.meth { <== Aufruf der Methode meth aus To_2
                          und dort der Methode xyz aus To_2};
```

```
  Writeln;
  o_2_objekt_1.xyz  { <== Aufruf der Methode xyz aus To_2 };
  Writeln;
  o_2_objekt_2.meth { <== Aufruf der Methode meth aus To_2
                          und dort der Methode xyz aus To_2};
  Writeln('====================')
END   { oop_stat }.
```

Ergebnis:

```
Methode xyz in o_1
     o_1-Objekt: j = 1
                 flag = TRUE , i = 1

Methode xyz in o_1
     o_1-Objekt: j = 202
                 flag = FALSE, i = 2

Methode meth

Methode xyz in o_2
     o_2-Objekt:      flag = FALSE, i = 2

Methode xyz in o_2
     o_2-Objekt:      flag = TRUE , i = 1000

Methode meth

Methode xyz in o_2
     o_2-Objekt:      flag = FALSE, i = 2000
```

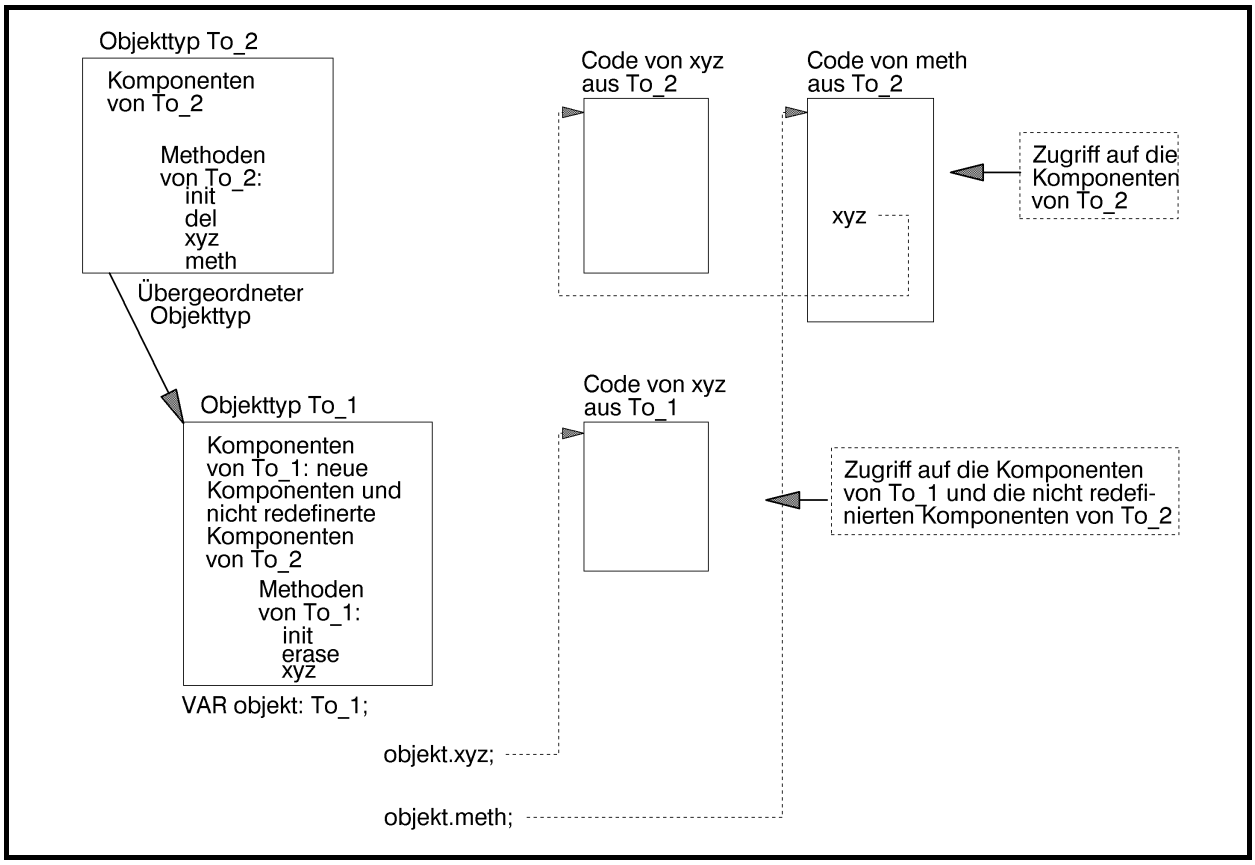


Abbildung 3.3.1-2: Statische Vererbung

3.3.2 Virtuelle und dynamische Methoden

Durch spezielle Mechanismen ist es jedoch möglich, eine Methode fest an ein Datenobjekt eines Objekttyps zu binden, der die Methode definiert. In obigem Beispiel heißt das, dass bei Anwendung der Methode `xyz` auf ein Datenobjekt vom Objekttyp `To_1` immer die Methode `To_1.xyz` genommen wird, bei Anwendung der Methode `xyz` auf ein Datenobjekt vom Objekttyp `To_2` immer die Methode `To_2.xyz` - und das auch, wenn `xyz` aus `meth` heraus aufgerufen wird. Dieses Konzept wird in Kapitel 2.2.5 als **Polymorphie** beschrieben: Eine Methode bekommt eine einzige Bezeichnung, die in der gesamten Objekttyphierarchie verwendet wird. Für jeden Objekttyp wird eine eigene Implementierung der Methode definiert. Wird die Methode mit einem Datenobjekt angestoßen, so wird zum Ablaufzeitpunkt dann jeweils diejenige Implementation der Methode genommen, die zu dem Objekttyp des Datenobjekts gehört. Die Bindung einer Methode an ein Datenobjekt erfolgt also erst zur Laufzeit und wird daher **späte Bindung** genannt.

Zur Realisierung der späten Bindung beinhaltet Pascal das Konzept der **virtuellen Methoden**. Die Methoden eines Objekttyps, die an ein Datenobjekt dieses Objekttyps erst zur Laufzeit angebunden werden, müssen als virtuell (Schlüsselwort `VIRTUAL`) gekennzeichnet werden. Der Compiler erzeugt für jeden Objekttyp, der virtuelle Methoden verwendet, eine eigene Ta-

belle mit den Adressen der Implementierungen der virtuellen Methoden, die **virtuelle Methodentabelle** des Objekttyps. ***Jedes Datenobjekt (Instanz) dieses Objekttyps muss vor Verwendung einer virtuellen Methode initialisiert werden***, indem es eine für den Objekttyp spezifische Methode aktiviert, die als **Konstruktor** (Schlüsselwort `CONSTRUCTOR` anstelle von `PROCEDURE`) bezeichnet wird. Dabei wird mit dem Datenobjekt ein Verweis auf die virtuelle Methodentabelle seines Objekttyps verbunden. Jedes einzelne Datenobjekt dieses Objekttyps muss zur Initialisierung den Konstruktor aufrufen, um die Verbindung mit der virtuellen Methodentabelle zu erhalten; eine einfache Wertzuweisung eines bereits initialisierten Datenobjekts auf ein noch nicht mit dem Konstruktor initialisierten Datenobjekts überträgt diese Verbindung nicht. Die Syntax eines Konstruktors unterscheidet sich bis auf das Schlüsselwort `CONSTRUCTOR` nicht von der einer Prozedur.

```
TYPE To_2 = OBJECT
              { Komponenten von To_2:}
              flag : BOOLEAN;
              i    : INTEGER;
              { Methoden von To_2: }
              CONSTRUCTOR init (wert : BOOLEAN;
                                x    : INTEGER);
              PROCEDURE del;
              PROCEDURE xyz; VIRTUAL;
              PROCEDURE meth;
            END;

     To_1 = OBJECT (To_2)
              { zusätzliche Komponenten von To_1:}
              j : INTEGER;
              { Methoden von To_1: }
              CONSTRUCTOR init (f : BOOLEAN;
                                x : INTEGER;
                                y : INTEGER);
              PROCEDURE erase;
              PROCEDURE xyz; VIRTUAL;
            END;
```

Entsprechend müssen auch die Prozedurköpfe bei den Implementierungen der Prozeduren `To_1.init` und `To_2.init` geändert werden:

```
CONSTRUCTOR To_2.init (wert : BOOLEAN;
                       x    : INTEGER);
```

und

```
CONSTRUCTOR To_1.init (f : BOOLEAN;
                       x : INTEGER;
```

```
                    y : INTEGER);
```

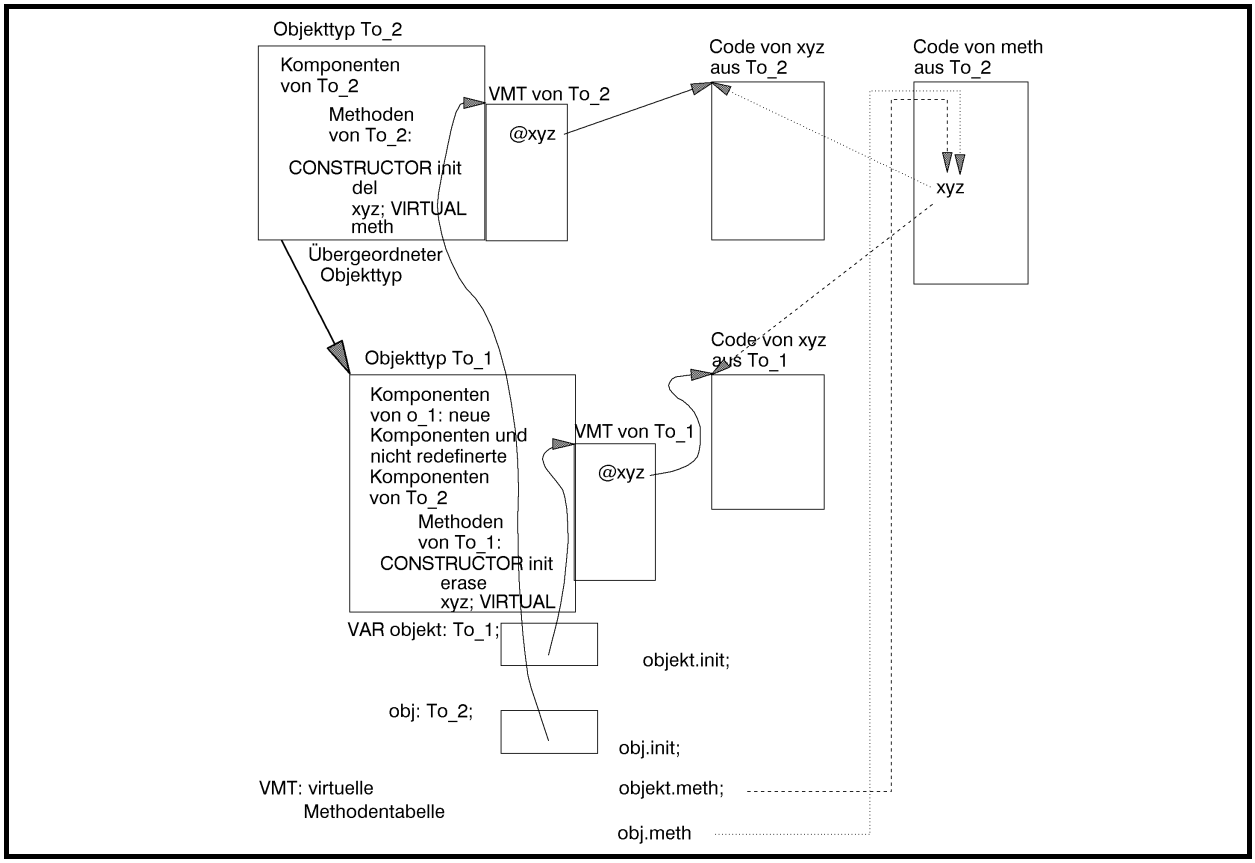


Abbildung 3.3.2-1: Virtuelle Methoden

Werden zwei Datenobjekt

```
VAR objekt: To_1;
    obj   : To_2;
```

deklariert und durch ihren Objekttyp-spezifischen Konstruktor initialisiert, z.B.:

```
objekt.init (TRUE, 1000, 2000);
obj.init (FALSE, 10);
```

so ruft die Methode `meth` für das Datenobjekt `objekt` die Methode `To_1.xyz` und für das Datenobjekt `obj` die Methode `To_2.xyz` auf (siehe Abbildung 3.3.2-1). Die Adresse der dem Objekttyp entsprechenden Methode `xyz` kann ja nun, auch im Aufruf von `meth`, während der Laufzeit aus der virtuellen Methodentabelle des Objekttyps genommen werden, die über den Verweis im Datenobjekt erreichbar ist.

Zusätzlich soll jetzt die Methode `del` von `To_2` als virtuelle Methode vereinbart werden:

```
TYPE To_2 = OBJECT
              { Komponenten von To_2:}
              flag : BOOLEAN;
              i    : INTEGER;
              { Methoden von To_2: }
              CONSTRUCTOR init (wert : BOOLEAN;
                                x    : INTEGER);
              PROCEDURE del; VIRTUAL;
              PROCEDURE xyz; VIRTUAL;
              PROCEDURE meth;
            END;
```

Das prinzipielle Format der internen Darstellung der Datenobjekte `objekt` und `obj` und des Datenobjekts `o_1_objekt_a` mit Objekttyp `To_1` zeigt Abbildung 3.3.2-2. Insbesondere ist die Methode `del` auch an `To_1` vererbt worden, d.h. es gibt einen entsprechenden Eintrag in der virtuellen Methodentabelle von `To_1`. Die Adressen nicht-virtueller Methoden tauchen nicht in den virtuellen Methodentabellen auf.

Auch wenn ein Objekttyp keine virtuellen Methoden enthält, kann ein Konstruktor definiert werden. Sein Aufruf entspricht dann einem normalen Methodenaufruf. Sobald ein Objekttyp oder einer seiner Vorfahren in der Vererbungshierarchie virtuelle Methoden enthält, ***muss*** vor der ersten Verwendung eines Datenobjekts mit diesem Objekttyp der Konstruktor aufgerufen werden. Prinzipiell können auch mehrere Konstruktoren für einen Objekttyp deklariert werden.

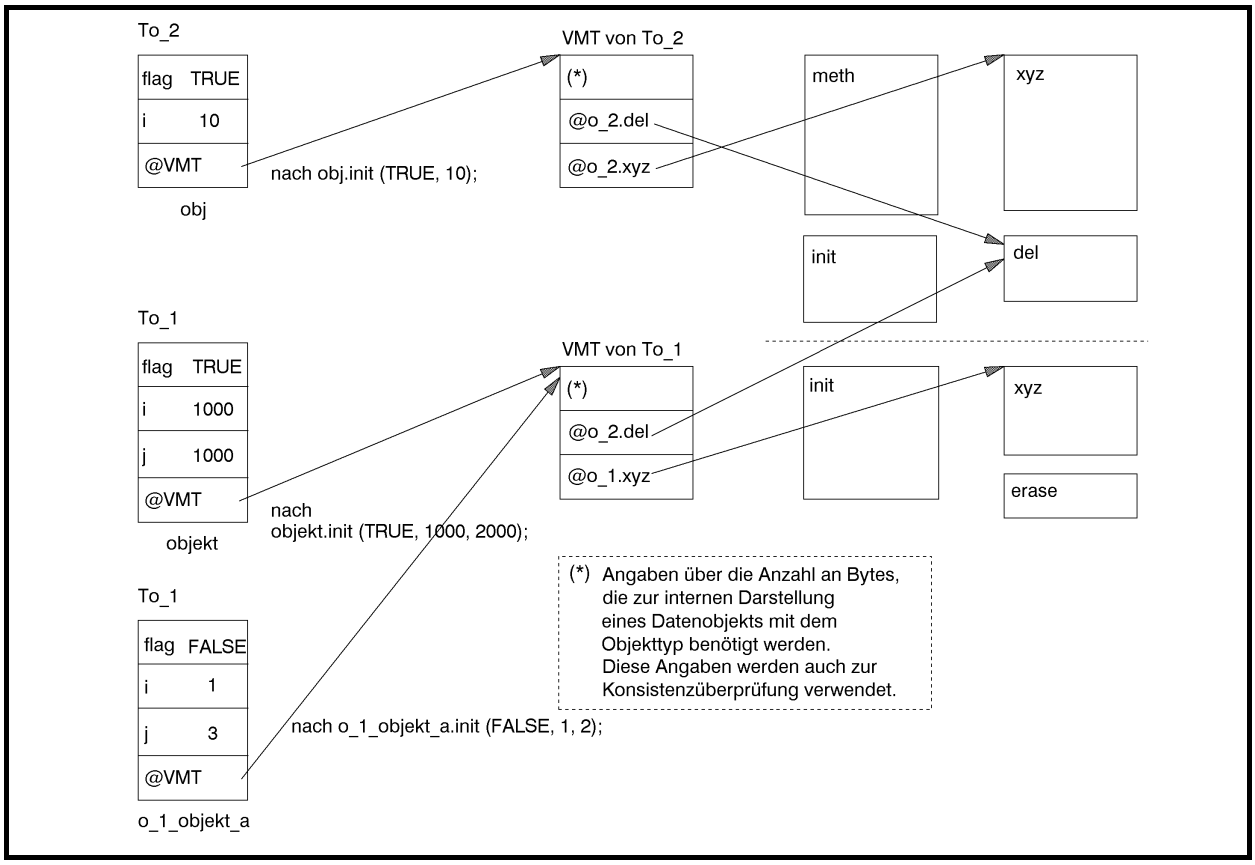


Abbildung 3.3.2-2: Interne Darstellung von Objekten mit virtuellen Methoden

In vielen Anwendungen belegen Datenobjekte aufgrund der großen Zahl ihrer Komponenten, einschließlich der aus übergeordneten Objektklassen eventuell über viele Stufen geerbten Komponenten, in ihrer internen Darstellung eine großen Speicherbereich. Daher ist es sinnvoll, derartige Datenobjekte erst während der Laufzeit als dynamische Datenobjekte einzurichten. Für den Umgang mit dynamischen Datenobjekten der OOP sind die Pascal-Standardprozeduren `New` und `Dispose` in ihrer Syntax und Semantik erweitert worden. Der folgende Programmausschnitt zeigt das Prinzip der Verwendung dynamischer Datenobjekte. Es wird eine Pointervariable deklariert und damit (als aktuellen Parameter) die Standardprozedur `New` aufgerufen. Diese legt das dynamische Datenobjekt (im Heap) an und gibt die Adresse des Datenobjekts als Wert der Pointervariablen zurück. Als weiterer Parameter der Prozedur `New` wird der Aufruf des Konstruktors für diesen Objekttyp mitgegeben, der automatisch von `New` nach Allokation des Speicherplatzes auf dem Heap für dieses neue Datenobjekt angestoßen wird und damit die Verbindung zur virtuellen Methodentabelle des Objekttyps herstellt.

```
TYPE Tobj = ^Pobj;
     Tobj = OBJECT
              { ... Komponenten des Objekttyps ...}
              CONSTRUCTOR init (< Liste der Formalparameter > );
              ...
              PROCEDURE proc (< Liste der Formalparameter > } );
              ...
```

```
              DESTRUCTOR done (< Liste der Formalparameter > );
                        { siehe unten }
            END;

VAR obj_ptr : Pobj;
...
BEGIN
  ...
  { Einrichten und initialisieren eines Objekts }
  New (obj_ptr, init (< Liste der Aktualparameter > ));
  ...
  { Aufruf der Methode proc für dieses Objekt }
  obj_ptr^.proc (< Liste der Aktualparameter > );
  ...
END;
```

Mit der Methode `Tobj.init (< Liste der Formalparameter > ))` als Konstruktor wird durch

```
New (obj_ptr, init (< Liste der Aktualparameter > ));
```

ein dynamisches Datenobjekt vom Objekttyp `Tobj` auf dem Heap angelegt, `obj_ptr` mit dessen Adresse versorgt und das Datenobjekt mit der Methode `init` initialisiert. Der Aufruf entspricht der Befehlsfolge

```
New (obj_ptr);
obj_ptr^.init (< Liste der Aktualparameter > );
```

Entsprechend ist die Pascal-Standardprozedur `Dispose` zur Freigabe des Speicherplatzes dynamisch erzeugter Variablen erweitert worden. In diesem Zusammenhang ist ein neuer Typ von Methoden wichtig, der **Destruktor**. Dabei handelt es sich um eine Methode innerhalb einer Objekttypdefinition (Schlüsselwort `DESTRUCTOR` anstelle von `PROCEDURE`). Ein Destruktor vereinigt die Freigabe des Speicherplatzes eines dynamisch erzeugten Datenobjekts mit dem entsprechenden Objekttyp auf dem Heap mit einer vom Benutzer definierten Methode, die unmittelbar vor der Speicherplatzfreigabe noch ausgeführt wird.

Anstelle eines abschließenden Aufrufs der Methode `Tobj.done` und anschließender Freigabe des durch `obj_ptr` adressierten Objekts auf dem Heap, etwa in der Form

```
obj_ptr^.done (< Liste der Aktualparameter > );
Dispose (obj_ptr);
```

kann man

```
Dispose (obj_ptr, done (< Liste der Aktualparameter > ));
```

verwenden.

Die virtuelle Methodentabelle eines Objekttyps enthält für jede virtuelle Methode einen Adressverweis auf den Code der virtuellen Methode. Wenn ein Objekttyp eine große Anzahl virtueller Methoden besitzt bzw. ererbt, wird dadurch viel Speicherplatz belegt. Auch wenn der Objekttyp nur wenige der ererbten virtuellen Methoden überschreibt, enthält seine virtuelle Methodentabelle Verweise auf alle Methoden, ob sie nun überschrieben wurden oder nicht. Durch die Verwendung **dynamischer Methoden** kann in dieser Situation der erforderliche Speicherplatz verkleinert werden. Die Syntax einer dynamischen Methode entspricht weitgehend einer virtuellen Methode; sie schreibt in der Deklaration neben dem Schlüsselwort `VIRTUAL` einen zusätzlichen numerischen dynamischen Methodenindex vor:

```
PROCEDURE methodenname (<parameter list>);
              VIRTUAL dynamischer_Methodenindex;
```

`dynamischer_Methodenindex` ist ein `INTEGER`-Wert, eine `INTEGER`-Konstante oder ein Ausdruck, der zu einem `INTEGER` -Wert ausgewertet wird (zwischen 1 und 65535). Überschreibt ein Nachkomme in der Vererbungshierarchie eine dynamische Methode, muss die Deklaration der überschreibenden Methode in Bezug auf Reihenfolge, Typ und Bezeichner der Parameter der Deklaration der überschriebenen Methode entsprechen. Auch sie muss das Schlüsselwort `VIRTUAL` angeben, dem der gleiche dynamische Methodenindexwert folgt, der in der Deklaration der überschriebenen Methode steht.

Bei der Verwendung dynamischer Methoden wird für einen Objekttyp neben seiner virtuellen Methodentabelle eine **dynamische Methodentabelle** generiert, die im wesentlichen Adressverweise auf die Methoden enthält, die auch wirklich überschrieben wurden und auf die dynamische Methodentabelle des direkten Vorfahren. Das folgende Beispiel erläutert das Prinzip. Der Objekttyp `TBase` vererbt Methoden an einen Objekttyp `TDerived`, der einige dieser Methoden überschreibt (`init`, `done`, `p10`, `p30`) und eine zusätzliche Methode (`p50`) deklariert:

```
TYPE TBase = OBJECT
               x : INTEGER;
               CONSTRUCTOR init;
               DESTRUCTOR done; VIRTUAL;
               PROCEDURE p10; VIRTUAL 10;
               PROCEDURE p20; VIRTUAL 20;
               PROCEDURE p30; VIRTUAL 30;
               PROCEDURE p40; VIRTUAL 40;
             END;
```

```
TDerived = OBJECT (TBase)
             y : INTEGER;
             CONSTRUCTOR init;
             DESTRUCTOR done; VIRTUAL;
             PROCEDURE p10; VIRTUAL 10;
             PROCEDURE p30; VIRTUAL 30;
             PROCEDURE p50; VIRTUAL 50;
           END;
```

Abbildung 3.3.2-3 zeigt im oberen Teil das Layout der virtuellen und dynamischen Methodentabellen der beiden Objekttypen. Im unteren Teil ist zum Vergleich das entsprechende Layout zu sehen, wenn anstelle dynamischer Methoden virtuelle Methoden verwendet, d.h. die dynamischen Methodenindizes weggelassen werden.

Der Aufruf einer dynamischen Methode ist zeitaufwendiger als der Aufruf einer virtuellen Methode: Die dynamischen Methodentabellen sind analog zur Vererbungshierarchie über Adressverweise miteinander verbunden. Eventuell ist die aufgerufene Methode über mehrere Hierarchiestufen geerbt, und ihre Adresse ist nur in der dortigen dynamischen Methodentabelle verzeichnet und nicht in der virtuellen Methodentabelle, die zum Objekttyp des aufgerufenen Objekts gehört. Um die Adresse der Methode zu finden, müssen u.U. die dynamischen Methodentabellen über mehrere Hierarchiestufen durchsucht werden. Durch das in Abbildung 3.3.2-3 gezeigte Zwischenspeicherfeld in einer dynamischen Methodentabelle wird dieser Suchvorgang beschleunigt (Details findet man beispielsweise in [PAS]).

Offensichtlich lohnt sich der Einsatz dynamischer anstelle virtueller Methoden nur, wenn, wenn sehr viele virtuelle Methoden von Vorfahren geerbt werden, die vom Objekttyp selbst überschrieben werden.

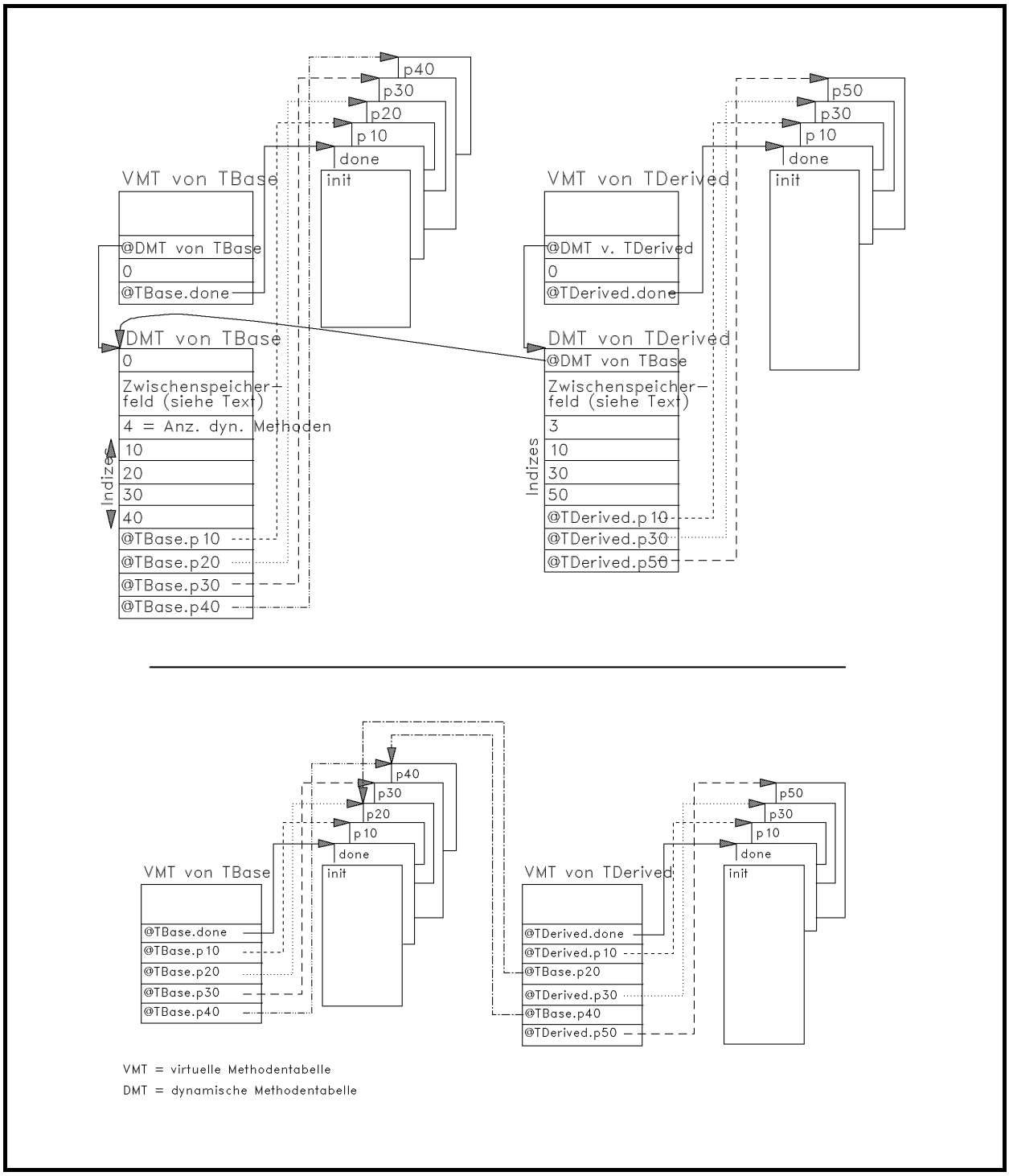


Abbildung 3.3.2-3: Dynamische Methodentabelle

4 Algorithmen

In der Theoretischen Informatik wird der Begriff des Algorithmus formal definiert, und es werden Modelle für den Vorgang der Berechnung aufgestellt und ihre Möglichkeiten und Grenzen untersucht. Im vorliegenden Text wird ein eher intuitiver und von praktischen Aufgabenstellungen geleiteter Zugang zu Algorithmen gewählt.

4.1 Ein intuitiver Algorithmusbegriff

Ein **Algorithmus** ist formal eine Verfahrensvorschrift (Prozedur, Berechnungsvorschrift), die aus einer endlichen Menge eindeutiger Regeln besteht, die eine endliche Aufeinanderfolge von Operationen spezifiziert, so dass eine Lösung zu einem Problem bzw. einer spezifischen Klasse von Problemen daraus erzielt wird.

Konkret kann man sich einen Algorithmus als ein Computerprogramm vorstellen, das in einer **Pascal-ähnlichen Programmiersprache** formuliert ist. Darunter versteht man Programmiersprachen, die

- Deklarationen von Datentypen und Variablen zulassen
- die üblichen arithmetischen Operationen mit Konstanten und Variablen und Wertzuweisungen an Variablen enthalten
- Kontrollstrukturen wie Sequenz (Hintereinanderreihung von Anweisungen, blockstrukturierte Anweisungen, Prozeduren), Alternativen (**IF** ... **THEN** ... **ELSE, CASE ... END**) und Schleifen (**WHILE** ... **DO** ..., **FOR** `i` := ... **TO** … **DO**..., **REPEAT** … **UNTIL** ...) besitzen.

Von einem Algorithmus erwartet man eine Reihe von **Eigenschaften**, damit er als „effektives Rechenverfahren“ gelten kann:

1. Die Verfahrensvorschrift (das Programm) soll aus einem endlichen Text bestehen.

2. Der Ablauf einer Berechnung soll schrittweise als Folge elementarer Rechenschritte erfolgen.

3. Das Verfahren soll **deterministisch** sein, d.h. in jedem Stadium einer Berechnung soll vollständig und eindeutig bestimmt sein, welcher elementare Rechenschritt als nächster getan wird. Ein Text „Bei Eingabe von x kann man für $f(x)$ einen von endlich vielen Werten als korrekten Wert aussuchen“ ist als Teil eines Algorithmus nicht zulässig.

4. Das Verfahren soll abgeschlossen sein, d.h. welcher Rechenschritt als nächster getan wird, soll ausschließlich von den Eingabewerten und den vorangegangenen berechneten Zwischenergebnissen abhängen.

Typische **Fragestellungen** bei einem gegebenen Algorithmus für eine Problemlösung sind:

- Hält der Algorithmus immer bei einer gültigen Eingabe nach endlich vielen Schritten an?
- Berechnet der Algorithmus bei einer gültigen Eingabe eine korrekte Antwort?

Die positive Beantwortung beider Fragen erfordert einen mathematischen **Korrektheitsbeweis** des Algorithmus. Bei positiver Beantwortung nur der zweiten Frage spricht man von **partieller Korrektheit**. Für die partielle Korrektheit ist lediglich nachzuweisen, dass der Algorithmus bei einer gültigen Eingabe, bei der er nach endlich vielen Schritten anhält, ein korrektes Ergebnis liefert.

- Wieviele Schritte benötigt der Algorithmus bei einer gültigen Eingabe **höchstens (worst case analysis)** bzw. **im Mittel (average case analysis)**, d.h. welche **(Zeit-) Komplexität** hat er im schlechtesten Fall bzw. im Mittel? Dabei ist es natürlich wichtig nachzuweisen, dass die Komplexität des Algorithmus von der jeweiligen Formulierungsgrundlage (Programmiersprache, Maschinenmodell) weitgehend unabhängig ist.

Entsprechend kann man nach dem benötigten **Speicherplatzbedarf (Platzkomplexität)** eines Algorithmus fragen.

Die Beantwortung dieser Fragen für den schlechtesten Fall gibt **obere Komplexitätsschranken** (Garantie für das Laufzeitverhalten bzw. den Speicherplatzbedarf) an.

- Gibt es zu einer Problemlösung eventuell ein „noch besseres“ Verfahren (mit weniger Rechenschritten, weniger Speicherplatzbedarf)? Wieviele Schritte wird jeder Algorithmus mindestens durchführen, der das vorgelegte Problem löst?

  Die Beantwortung dieser Frage liefert untere Komplexitätsschranken.

4.2 Problemklassen

In der Informatik beschäftigt man sich häufig damit, Problemstellungen mit Hilfe von Rechnerprogrammen zu lösen. Natürlich wird man dabei das Programm so entwerfen, dass es nicht nur die Lösung für eine einzige konkrete Problemstellung findet, sondern für eine ganze Klasse von Problemen, die natürlich alle „ähnlich“ spezifiziert sind. Beispielsweise wird ein Sortierverfahren in der Lage sein, nicht nur einen Satz von 100 Zahlen zu sortieren, sondern eine beliebige Anzahl von Zahlen, eventuell sogar von feiner strukturierten Objekten.

Ein **Problem** ist eine zu beantwortende Fragestellung, die von **Problemparametern** (Variablenwerten, Eingaben usw.) abhängt, deren genaue Werte in der Problembeschreibung zunächst unspezifiert sind, deren Typen und syntaktische Eigenschaften jedoch in die Problembeschreibung eingehen. Ein Problem wird beschrieben durch:

1. eine allgemeine Beschreibung aller Parameter, von der die Problemlösung abhängt; diese Beschreibung spezifiziert die **(Problem-) Instanz (Eingabeinstanz)**

2. die Eigenschaften, die die Antwort, d.h. die Problemlösung, haben soll.

Eine spezielle Problemstellung erhält man durch Konkretisierung einer Probleminstanz, d.h. durch die Angabe spezieller Parameterwerte in der Problembeschreibung.

Im folgenden werden einige grundlegende **Problemtypen** unterschieden und zunächst an Beispielen erläutert.

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem

Problem-
instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Gesuchte
Lösung: Eine **Tour** durch G, d.h. eine geschlossene Kantenfolge
$\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ mit $(v_{i_j}, v_{i_{j+1}}) \in E$ für $j = 1, ..., n-1$,
in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt, die minimale Kosten (unter allen möglichen Touren durch G) verursacht. Man kann o.B.d.A. $v_{i_1} = v_1$ setzen.

Die Kosten einer Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist dabei die Summe der Gewichte der Kanten auf der Tour, d.h. der Ausdruck

$$\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1}).$$

Das Problem des Handlungsreisenden findet vielerlei Anwendungen in den Bereichen

- Transportoptimierung:
  Ermittlung einer kostenminimalen Tour, die im Depot beginnt, $n-1$ Kunden erreicht und im Depot endet.
- Fließbandproduktion:
  Ein Roboterarm soll Schrauben an einem am Fließband produzierten Werkstück festdrehen. Der Arm startet in einer Ausgangsposition (über einer Schraube), bewegt sich dann von einer zur nächsten Schraube (insgesamt *n* Schrauben) und kehrt in die Ausgangsposition zurück.
- Produktionsumstellung:
  Eine Produktionsstätte stellt verschiedene Artikel mit denselben Maschinen her. Der Herstellungsprozess verläuft in Zyklen. Pro Zyklus werden *n* unterschiedliche Artikel produziert. Die Änderungskosten von der Produktion des Artikels v_i auf die des Artikels v_j betragen $w((v_i, v_j))$ (Geldeinheiten). Gesucht wird eine kostenminimale Produktionsfolge. Das Durchlaufen der Kante (v_i, v_j) entspricht dabei der Umstellung von Artikel v_i auf Artikel v_j. Gesucht ist eine Tour (zum Ausgangspunkt zurück), weil die Kosten des nächsten, hier des ersten, Zyklusstarts mit einbezogen werden müssen.

Problem des Handlungsreisenden auf Graphen als Berechnungsproblem

Problem-
instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Gesuchte
Lösung: Der Wert $\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$ einer kostenminimalen Tour

$\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ durch *G*.

Zu beachten ist hierbei, dass nicht eine kostenminimale Tour selbst gesucht wird, sondern lediglich der *Wert* einer kostenminimalen Tour. Eventuell ist es möglich, diesen Wert (durch geeignete Argumentationen und Hinweise) zu bestimmen, ohne eine kostenminimale Tour explizit anzugeben. Das Berechnungsproblem scheint daher „einfacher“ zu lösen zu sein als das Optimierungsproblem.

Problem des Handlungsreisenden auf Graphen als Entscheidungsproblem

Problem-
instanz: $[G,K]$,
$G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht; $K \in \mathbf{R}_{\geq 0}$

Gesuchte
Lösung: Die Antwort auf die Frage:
Gibt es eine kostenminimale Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ durch G, deren Wert $\leq K$ ist?

Bei dieser Problemstellung ist nicht einmal der Wert einer kostenoptimalen Tour gesucht, sondern lediglich eine Entscheidung, ob dieser Wert „nicht zu groß“, d.h. kleiner als eine vorgegebene Schranke ist. Das Entscheidungsproblem scheint daher „noch einfacher“ zu lösen zu sein als das Optimierungs- und das Berechnungsproblem.

Im vorliegenden Fall des Problems des Handlungsreisenden befindet man sich jedoch im Irrtum: In einem in der Theoretischen Informatik präzisierten Sinne sind bei diesem Problem alle Problemvarianten algorithmisch gleich schwierig zu lösen.

Das Beispiel lässt sich verallgemeinern, wobei im folgenden vom (vermeintlich) einfacheren Problemtyp zum komplexeren Problemtyp übergegangen wird.

Die Instanz x eines Problems Π ist eine endliche Zeichenkette über einem endlichen Alphabet Σ_Π, das dazu geeignet ist, derartige Problemstellungen zu formulieren, d.h. $x \in \Sigma_\Pi^*$.

Es werden folgende **Problemtypen** unterschieden:

Entscheidungsproblem Π:

Problem-
instanz: $x \in \Sigma_\Pi^*$
und Spezifikation einer Eigenschaft, die einer Auswahl von Elementen aus Σ_Π^* zukommt, d.h. die Spezifikation einer Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit
$L_\Pi = \{u \in \Sigma^* \mid u \text{ hat die beschriebene Eigenschaft}\}$

Gesuchte Lösung: Entscheidung „ja“, falls $x \in L_\Pi$ ist,
Entscheidung „nein“, falls $x \notin L_\Pi$ ist.

Bei der Lösung eines Entscheidungsproblems geht es also darum, bei Vorgabe einer Instanz $x \in \Sigma_\Pi^*$ zu entscheiden, ob x zur Menge L_Π gehört, d.h. eine genau spezifizierte Eigenschaft, die genau allen Elementen in L_Π zukommt, besitzt, oder nicht.

Es zeigt sich, dass der hier formulierte Begriff der Entscheidbarkeit sehr eng gefasst ist. Eine erweiterte Definition eines Entscheidungsproblems verlangt bei der Vorgabe einer Instanz $x \in \Sigma_\Pi^*$ nach endlicher Zeit lediglich eine positive Entscheidung „ja“, wenn $x \in L_\Pi$ ist. Ist $x \notin L_\Pi$, so kann die Entscheidung eventuell nicht in endlicher Zeit getroffen werden. Dieser Begriff der Entscheidbarkeit führt auf die rekursiv aufzählbaren bzw. auf die entscheidbaren Mengen; beide Begriffe sind Untersuchungsgegenstand der Theoretischen Informatik.

Berechnungsproblem Π:

Problem-instanz: $x \in \Sigma_\Pi^*$
und die Beschreibung einer Funktion $f : \Sigma_\Pi^* \to \Sigma'^*$.

Gesuchte Lösung: Berechneter Wert $f(x)$.

Optimierungsproblem Π:

Problem-instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ **eine Menge zulässiger Lösungen** zuordnet
3. Spezifikation einer **Zielfunktion** m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ einen Wert $m_\Pi(x, y)$, den **Wert einer zulässigen Lösung**, zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Gesuchte Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$) bzw. $m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).

Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

In dieser (formalen) Terminologie wird das Handlungsreisenden-Minimierungsproblem wie folgt formuliert:

Problem-
instanz:

1. $G = (V, E, w)$
   $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht
2. $\mathrm{SOL}(G) = \left\{ T \mid T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle \text{ ist eine Tour durch } G \right\}$
   ; eine Tour durch G ist eine geschlossene Kantenfolge, in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt
3. für $T \in \mathrm{SOL}(G)$, $T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle$, ist die Zielfunktion definiert durch $m(G, T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$
4. $goal = \min$

Gesuchte
Lösung: Eine Tour $T^* \in \mathrm{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G, T^*)$.

4.3 Komplexität eines Algorithmus

In der Theoretischen Informatik werden die Zeit- und Platzkomplexität eines Algorithmus exakt definiert. Im vorliegenden Zusammenhang soll die Definition dieser Begriffe eher pragmatisch erfolgen.

Mit der Definition einer Probleminstanz für ein zu lösendes Problem ist die Festlegung der Größe der Probleminstanz verbunden. Da Lösungsalgorithmen für ein Problem als Computerprogramme formuliert werden, ist es sinnvoll, die Eingabe für einen derartigen Lösungsalgorithmus als Zeichenkette, etwa als Zeichenkette aus Binärwerten, vorzugeben. Als Zeichenvorrat wird ein Alphabet gewählt, das zur Formulierung von Instanzen für dieses Problem „geeignet“ ist. Letztlich kann man natürlich die Buchstaben jedes endlichen Alphabets als Zeichenketten über $\{0, 1\}$ formulieren. Unter der **Größe der Probleminstanz** I versteht man

dann die Anzahl der Zeichen, z.B. Bits, um die Probleminstanz zu notieren bzw. in den Lösungsalgorithmus einzugeben, und bezeichnet sie mit $size(I)$.

Der Aufwand, der entsteht, um bei Eingabe der Probleminstanz I in einen Lösungsalgorithmus **A** eine Lösung zu erzielen, sei $t_{\mathbf{A}}(I)$. Dieser Aufwand setzt sich aus der Anzahl durchlaufener Anweisungen und dem Aufwand bei der Abarbeitung jeder einzelnen Anweisung zusammen. Der Wert $t_{\mathbf{A}}(I)$ wird in Abhängigkeit von $size(I)$ angegeben. Ist $L(n)$ die Menge aller Probleminstanzen I mit $size(I) \le n$, für die der Algorithmus **A** eine Lösung erzielt, dann ist

$$T_{\mathbf{A}}(n) = \max \left\{ t_{\mathbf{A}}(I) \,\middle|\, I \in L(n) \right\}$$

eine obere Schranke für den Aufwand des Algorithmus **A**, der bei Eingabe einer Probleminstanz bis zur Größe n entsteht. Diese Funktion wird **Zeitkomplexität (im schlechtesten Fall) des Algorithmus A** genannt. Entsprechend erhält man die **Platzkomplexität (im schlechtesten Fall) des Algorithmus A**, wenn man $t_{\mathbf{A}}(I)$ durch die Anzahl an Speicherplätzen ersetzt, die bei der Berechnung einer Lösung für die Instanz I mit $size(I) \le n$ benötigt werden.

Wird in der Definition von $T_{\mathbf{A}}(n)$ anstelle des Maximums der Erwartungswert über alle $I \in L(n)$ genommen, wobei eine entsprechende Verteilung angenommen wird, so erhält man die **mittlere Zeitkomplexität des Algorithmus A**. Entsprechend wird die **mittlere Platzkomplexität des Algorithmus A** definiert.

Das folgende Beispiel zeigt, dass man bei der Definition der Größe einer Probleminstanz und der Analyse eines Lösungsalgorithmus sorgfältig vorgehen muss.

Die Pascal-Prozedur `zweier_potenz` berechnet bei Eingabe einer Zahl $n > 0$ den Wert $c = 2^{2^n} - 1$.

```
PROCEDURE zweier_potenz (    n : INTEGER;
                         VAR c : INTEGER);

VAR idx : INTEGER;
    p   : INTEGER;

BEGIN {zweier-potenz }
  idx := n;                  { Anweisung 1 }
  p   := 2;                  { Anweisung 2 }
  WHILE idx > 0 DO           { Anweisung 3 }
    BEGIN
      p   := p*p;             { Anweisung 4 }
      idx := idx - 1;         { Anweisung 5 }
```

```
        END;
     c := p - 1;                    { Anweisung 6 }
END   { zweier-potenz };
```

Werden für die Aufwandsberechnung nur die Anzahl der ausgeführten Anweisungen gezählt, so ergeben sich bei Eingabe der Zahl $n > 0$ folgende Werte: Die Anweisungen 1, 2 und 6 werden jeweils einmal durchlaufen. Die Anweisungen 4 und 5 werden für jeden Wert von `idx` jeweils einmal durchlaufen, wobei `idx` nacheinander die Werte $n, n-1, ..., 1$ annimmt. Die Anweisung 3 wird $(n+1)$-mal ausgeführt. Die Anzahl durchlaufener Anweisungen bei Eingabe der Zahl $n > 0$ beträgt $3 \cdot n + 4$. Die Anzahl ist linear im Wert der Eingabe *n*. Das Ergebnis $c = 2^{2^n} - 1$ belegt jedoch 2^n viele binäre Einsen und benötigt zu seiner Erzeugung mindestens 2^n viele (elementare) Schritte. Vergrößert man die Eingabe *n* um eine Konstante *k*, d.h. betrachtet man die Eingabe $n + k$, so ist die Laufzeit gleich $3 \cdot (n+k) + 4$, also immer noch linear im Wert der Eingabe, während das Ergebnis um den *Faktor* 2^k größer wird.

Diese Anomalität wird durch folgende korrekte Laufzeituntersuchung behoben:

Als Größe der Eingabe wird nicht der Wert von *n* genommen, sondern die Anzahl der Bits, die benötigt werden, um *n* darzustellen, d.h. $size(n) = \lfloor \log_2(n) \rfloor + 1$. Als Aufwand für die Ausführung einer Anweisung wird die entstandene Anzahl elementarer Bitoperationen definiert.

In der Pascal-Prozedur `zweier_potenz` erfordert die Wertzuweisung in Anweisung 1 dann $size(n)$ viele Bitoperationen. Die Wertzuweisung in Anweisung 2 benötigt 2 Bitoperationen (Erzeugung der Zahl $2_{10} = 10_2$). Zur Berechnung der Anzahl der übrigen Bitoperationen wird auf Ergebnisse aus Kapitel 4.5 vorgegriffen. Die arithmetische Operation in Anweisung 6 benötigt $c_6 \cdot \log_2\left(2^{2^n}\right) = c_6 \cdot 2^n$ viele Bitoperationen mit einer Konstanten $c_6 > 0$. Die Anweisungen der Schleife werden insgesamt *n*-mal durchlaufen (Anweisung 3 sogar $(n+1)$-mal). Zu Beginn des *i*-ten Schleifendurchlaufs hat `p` den Wert $2^{2^{i-1}}$, und `idx` hat den Wert $n - i + 1$. Die Anzahl der Bitoperationen zur Durchführung der Anweisung 4 im *i*-ten Schleifendurchlauf ist daher von der Größe $c_4 \cdot 2^{2i}$ mit einer Konstanten $c_4 > 0$. In Anweisung 5 werden $c_5 \cdot \log_2(n - i + 1)$ Bitoperationen mit einer Konstanten $c_5 > 0$ ausgeführt. Insgesamt benötigt die Schleife eine Anzahl von Bitoperationen, die sich durch $c_5 \cdot \sum_{i=1}^{n} \log_2(n-i+1) + c_4 \cdot \sum_{i=1}^{n} 2^{2i} \le c_5 \cdot n \cdot \log_2(n) + c_4 \cdot 4/3 \cdot \left(2^{2n} - 1\right)$ abschätzen lässt. Da wegen $size(n) = \lfloor \log_2(n) \rfloor + 1$ der Zusammenhang $n = c_0 \cdot 2^{size(n)}$ mit einer Konstanten $c_0 > 0$ gilt, ist der Gesamtaufwand an Bitoperationen beschränkt durch einen Wert der Größe $c \cdot 2^{2^{size(n)}}$ mit einer Konstanten $c > 0$. Dieser Wert gibt die Realität exakt wieder.

Diese Art der Aufwandsabschätzung, nämlich die Untersuchung der Zahl (elementarer) Bitoperationen in Abhängigkeit von der Größe $size(I)$ einer Eingabeinstanz *I*, wird immer dann angewandt, wenn die Laufzeit des Lösungsalgorithmus von numerischen Werten in der Eingabeinstanz abhängt.

4.4 Größenordnung von Funktionen

Häufig ist man gar nicht am exakten Wert der Komplexität eines Algorithmus (in Abhängigkeit von der Größe der Eingabeinstanz) interessiert, sondern lediglich am „Typ" des entsprechenden funktionalen Zusammenhangs (beispielsweise lineare oder polynomielle oder exponentielle Laufzeit). So kann man Funktionenklassen unterscheiden, die sehr unterschiedliches Wachstumsverhalten zeigen. Die folgende Tabelle zeigt beispielsweise fünf Funktionen $h_i : \mathbf{R}_{>0} \to \mathbf{R},\ i = 1,\ \ldots,\ 5$ und einige ausgewählte (gerundete) Funktionswerte.

Spalte 1	Spalte 2	Spalte 3	Spalte 4	Spalte 5
i	$h_i(n)$	$h_i(10)$	$h_i(100)$	$h_i(1000)$
1	$\log_2(n)$	3,3219	6,6439	9,9658
2	$\sqrt{n}$	3,1623	10	31,6228
3	n	10	100	1000
4	n^2	100	10.000	1.000.000
5	2^n	1024	$1{,}2676506 \cdot 10^{30}$	$> 10^{693}$

Die folgende Tabelle zeigt noch einmal die fünf Funktionen $h_i : \mathbf{R}_{>0} \to \mathbf{R},\ i = 1,\ \ldots,\ 5$. Es sei $y_0 > 0$ ein fester Wert. Die dritte Spalte zeigt für jede der fünf Funktionen Werte n_i mit $h_i(n_i) = y_0$. In der vierten Spalte sind diejenigen Werte $\overline{n_i}$ aufgeführt, für die $h_i\left(\overline{n_i}\right) = 10 \cdot y_0$ gilt, d.h. dort ist angegeben, auf welchen Wert man n_i vergrößern muss, damit der Funktionswert auf den 10-fachen Wert wächst. Wie man sieht, muss bei der Logartihmusfunktion wegen ihres langsamen Wachstums der Wert stark vergrößert werden, während bei der schnell anwachsenden Exponentialfunktion nur eine additive konstante Steigerung um ca. 3,3 erforderlich ist.

Spalte 1	Spalte 2	Spalte 3	Spalte 4
i	$h_i(n)$	x_i mit $h_i(n_i) = y_0$	$\overline{x_i}$ mit $h_i\left(\overline{x_i}\right) = 10 \cdot y_0$
1	$\log_2(n)$	n_1	$(n_1)^{10}$
2	$\sqrt{n}$	n_2	$100 \cdot n_2$
3	n	n_3	$10 \cdot n_3$
4	n^2	n_4	$\approx 3{,}162 \cdot n_4$
5	2^n	n_5	$\approx n_5 + 3{,}322$

Offensichtlich unterscheiden sich die angegeben Funktionen in ihrem Wachstumsverhalten teilweise drastisch, wobei der Unterschied zwischen $h_3(n) = n$ und $h_4(n) = n^2$ weniger stark ausfällt als zwischen $h_4(n) = n^2$ und $h_5(n) = 2^n$.

In vielen Fällen führt die exakte Analyse des Laufzeitverhaltens eines Algorithmus auf relativ komplexe funktionale Zusammenhänge. Eventuell genügt es dabei sogar, lediglich eine obere Schranke für das Laufzeitverhalten anzugeben, wenn diese eine einfache funktionale Form aufweist. Anstelle des exakten Werts $T_{\mathbf{A}}(n)$ für die Zeitkomplexität ist man an einer funktional einfachen oberen Abschätzung $f(n)$ interessiert, die aber dann in „derselben Größenordnung“ liegen sollte. Diese Überlegungen führen auf den mathematischen Begriff der Ordnung einer Funktion.

Im folgenden seien $f : \mathbf{N} \to \mathbf{R}$ und $g : \mathbf{N} \to \mathbf{R}$ zwei Funktionen. Die Funktion *f* ist von der **(Größen-) Ordnung** $O(g(n))$, geschrieben $f(n) \in O(g(n))$, wenn gilt:
es gibt eine Konstante $c > 0$, die von n nicht abhängt, und ein $n_0 \in \mathbf{N}$, so dass $|f(n)| \le c \cdot |g(n)|$ ist für jedes $n \in \mathbf{N}$ mit $n \ge n_0$ gilt („, ..., so dass $|f(n)| \le c \cdot |g(n)|$ ist für fast alle $n \in \mathbf{N}$“ gilt, d.h. für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen).

$O(g(n))$ ist also eine Menge von Funktionen, nämlich die Menge aller Funktionen *f*, für die es jeweils eine Konstante $c > 0$ gibt, so dass $|f(n)| \le c \cdot |g(n)|$ ist für fast alle $n \in \mathbf{N}$ gilt.

Einige Regeln, die sich direkt aus der Definition der Größenordnung einer Funktion herleiten lassen, lauten:

$f(n) \in O(f(n))$

Für d = const. ist $d \cdot f(n) \in O(f(n))$

Es gelte $|f(n)| \le c \cdot |g(n)|$ für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen. Dann ist $O(f(n)) \subseteq O(g(n))$, insbesondere $f(n) \in O(g(n))$.

$O(O(f(n))) = O(f(n))$;
hierbei ist

$$O(O(f(n))) = \left\{ h \;\middle|\; \begin{array}{l} h : \mathbf{N} \to \mathbf{R} \text{ und es gibt eine Funktion } g \in O(f(n)) \text{ und eine Konstante } c > 0 \\ \text{mit } |h(n)| \le c \cdot |g(n)| \text{ für jedes } n \in \mathbf{N} \text{ bis auf höchstens endlich viele Ausnahmen} \end{array} \right\}$$

Im folgenden seien S_1 und S_2 zwei Mengen, deren Elemente miteinander arithmetisch verknüpft werden können, etwa durch den Operator $\circ$. Dann ist
$S_1 \circ S_2 = \{ s_1 \circ s_2 \mid s_1 \in S_1 \text{ und } s_2 \in S_2 \}$.
Mit dieser Notation gilt:

$$O(f(n)) \cdot O(g(n)) \subseteq O(f(n) \cdot g(n)),$$

$$O(f(n) \cdot g(n)) \subseteq |f(n)| \cdot O(g(n)),$$

$$O(f(n)) + O(g(n)) \subseteq O(|f(n)| + |g(n)|).$$

Im folgenden werden einige wichtige Eigenschaften des *O*-Operators in Form mathematischer Sätze angeführt und bewiesen.

1. Ist p ein Polynom vom Grade m, $p(n) = \sum_{i=0}^{m} a_i \cdot n^i$ mit $a_i \in \mathbf{R}$ und $a_m \ne 0$, so ist $p(n) \in O(n^k)$ für $k \ge m$,
   insbesondere $p(n) \in O(n^m)$ und $n^m \in O(n^k)$ für $k \ge m$.

2. Konvergiert $f(n) = \sum_{i=0}^{\infty} a_i \cdot n^i$ für $n \le r$, so ist für jedes $k \ge 0$:
   $f(n) = \sum_{i=0}^{k} a_i \cdot n^i + g(n)$ mit $g(n) \in O(n^{k+1})$.

Teil 1. des Satzes ergibt sich wie folgt: Mit $c = \max\{|a_0|, ..., |a_m|\}$ und $n \ge 2$ ist

$$|p(n)| \le \sum_{i=0}^{m} |a_i| \cdot n^i \le c \cdot \sum_{i=0}^{m} n^i = c \cdot \frac{n^{m+1} - 1}{n - 1} = c \cdot \frac{n^m - 1/n}{1 - 1/n} \le 2c \cdot n^m \le 2c \cdot n^k$$

für $k \ge m$, also nach Definition $p(n) \in O(n^k)$.

Teils 2. lässt sich folgendermaßen zeigen:

$$f(n) = \sum_{i=0}^{\infty} a_i \cdot n^i = \sum_{i=0}^{k} a_i \cdot n^i + \sum_{i=k+1}^{\infty} a_i \cdot n^i = \sum_{i=0}^{k} a_i \cdot n^i + n^{k+1} \cdot \sum_{i=k+1}^{\infty} a_i \cdot n^{i-(k+1)} .$$

Wegen der Konvergenz von f konvergiert die zweite Summe, d.h. mit

$$g(n) = n^{k+1} \cdot \sum_{i=k+1}^{\infty} a_i \cdot n^{i-(k+1)}$$

gilt

$$|g(n)| \le n^{k+1} \cdot \left| \sum_{i=k+1}^{\infty} a_i \cdot n^{i-(k+1)} \right| \le n^{k+1} \cdot \sum_{i=k+1}^{\infty} |a_i| \cdot n^{i-(k+1)} \le n^{k+1} \cdot C ,$$

also $g(n) \in O\left(n^{k+1}\right)$. ///

Aus der Analysis ist für $a > 1$ und $b > 1$ die Umrechnung $\log_a(n) = \dfrac{1}{\log_b(a)} \cdot \log_b(n)$ bekannt. Daher gilt

> Für $a > 1$ und $b > 1$ ist $\log_a(n) \in O(\log_b(n))$.
>
> Statt $O(\log_b(n))$ schreibt man $O(\log(n))$.

Wegen $e^{h(n)} = 2^{\log_2(e) \cdot h(n)}$ gilt

> $e^{h(n)} \in O\left(2^{O(h(n))}\right)$.

Da eine Exponentialfunktion der Form $f(n) = a^n$ mit $a > 1$ schneller wächst als jedes Polynom $p(n)$, genauer: $\lim\limits_{n \to \infty} \dfrac{|p(n)|}{a^n} = 0$, ist $a^n \notin O(p(n))$, jedoch $p(n) \in O\left(a^n\right)$.

Entsprechend gelten für jede Logarithmusfunktion $\log_a(n)$ mit $a > 1$ und jedes Polynom $p(n)$ die Beziehungen $p(n) \notin O(\log_a(n))$ und $\log_a(n) \in O(p(n))$.

Für jede Logarithmusfunktion $\log_a(n)$ mit $a > 1$ und jede Wurzelfunktion $\sqrt[m]{n}$ ist $\sqrt[m]{n} \notin O(\log_a(n))$, jedoch $\log_a(n) \in O\left(\sqrt[m]{n}\right)$.

Insgesamt ergibt sich damit

> Es seien $a > 1$, $b > 1$, $m \in \mathbf{N}_{>0}$, $p(n)$ ein Polynom; dann ist
> $O(\log_b(n)) \subset O\left(\sqrt[m]{n}\right) \subset O(p(n)) \subset O\left(a^n\right)$.

Bei der Analyse des Laufzeitverhaltens von Algorithmen treten häufig Funktionen auf, die arithmetische Verknüpfungen von Logarithmusfunktionen, Polynomen und Exponentialfunktionen sind. Man sagt, eine durch $f(n)$ gegebene Funktion hat **langsameres Wachstum** als eine durch $g(n)$ gegebene Funktion, wenn $O(f(n)) \subseteq O(g(n))$ ist. Entsprechend weist g schnelleres Wachstum als f auf. So sind die folgenden Funktionen gemäß ihrer Wachstumsgeschwindigkeit geordnet:

langsames Wachstum $\longrightarrow$ schnelles Wachsstum

$$(\log_2(n))^5 \longrightarrow n^2 \longrightarrow n^3 \longrightarrow \frac{n^4}{\log_2(n)} \text{ und}$$

$$(\log_2(n))^2 \longrightarrow n \longrightarrow n \cdot (\log_2(n))^3 \longrightarrow \frac{n^2}{\log_2(n)}$$

4.5 Zusammenfassung der Laufzeiten arithmetischer Operationen

Im vorliegenden Kapitel werden die Anzahlen der erforderlichen Bitoperationen **bei den gängigen arithmetischen Operationen** zusammengefasst. Dabei erfolgt eine Beschränkung auf natürliche Zahlen; negative ganze Zahlen werden in Komplementdarstellung behandelt.

Es seien x und y natürliche Zahlen mit $x \geq y$, $k = size(x)$ und $l = size(y)$. Die Binärdarstellungen seien $x = [x_{k-1}x_{k-2} \ldots x_1x_0]_2$ bzw. $y = [y_{l-1}y_{l-2} \ldots y_1y_0]_2$.

Addition $z := x + y$

Für $l \leq k$ wird y durch Voranstellung von $k-l$ führenden binären Nullen auf dieselbe Länge wie x gebracht, so dass $y = [y_{k-1}y_{k-2} \ldots y_1y_0]_2$ ist. Das Ergebnis z besitzt eventuell eine Binärstelle mehr als x: $z = [z_k z_{k-1} \ldots z_1 z_0]_2$. Bei der bitweisen stellengerechten Addition werden für Stelle i die Bits x_i und y_i und ein eventueller Übertrag aus der Stelle $i-1$ addiert. Wird mit $u = [u_k u_{k-1} \ldots u_1 u_0]$ die Bitfolge der Überträge aus der jeweils vorherigen Bitaddition bezeichnet, so ist
$u_0 = 0$, $z_k = u_k$ und $z_i = x_i + y_i + u_i \pmod 2$ für $i = 0, \ldots, k-1$.
Die Addition „+“ lässt sich auf die logischen Operationen $\wedge$, $\vee$ und $\oplus$ (Exklusiv-Oder-Operation (Addition modulo 2)[8] zurückführen, und damit auf die Operationen $\wedge$, $\vee$ und $\neg$:

[8] Die Operation „$\oplus$“ ist definiert durch $0 \oplus 0 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, $1 \oplus 1 = 0$. Es gilt $a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$.

$z_i = x_i \oplus y_i \oplus u_i$ und $u_{i+1} = (x_i \wedge y_i) \vee (u_i \wedge (x_i \vee y_i))$ für $i = 0, ..., k-1$.

Bei der Addition werden also $O(k) = O(\log(x))$ viele Bitoperationen ausgeführt.

Multiplikation $z := x \cdot y$

Das Ergebnis z besitzt eventuell doppelt so viele Binärstellen wie x: $z = [z_{2k-1} z_{2k-2} \ldots z_1 z_0]_2$. Gemäß der „Schulmethode“ wird folgender (Pseudo-) Code ausgeführt:

```
z := 0;                          { Anweisung 1 }
FOR i := l − 1 DOWNTO 0 DO       { Anweisung 2 }
  BEGIN
    SHL (z, 1);                  { Anweisung 3 }
    IF y_i = 1 THEN z := z + x;  { Anweisung 4 }
  END;
```

Der Aufwand für Anweisung 1 ist von der Ordnung $O(1)$. Die `FOR`-Schleife (Anweisungen 2 bis 4) wird l-mal durchlaufen, wobei jeweils ein Aufwand der Ordnung $O(size(z))$ entsteht. Wegen $size(z) \leq 2k$ und $l \leq k$ werden zur Multiplikation $O(k^2) = O((\log(x))^2)$ viele Bitoperationen ausgeführt.

Bemerkung: Der schnellste bekannte Multiplikationsalgorithmus benötigt unter Einsatz einer Variante der diskreten Fouriertransformation eine Anzahl an Bitoperationen der Ordnung $O(k \cdot \log(k) \cdot \log(\log(k)))$.[9]

9 Siehe Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

Ganzzahlige Division $z := \lfloor x/y \rfloor$ für $y \neq 0$

Im ersten Ansatz könnte die Berechnung von $\lfloor x/y \rfloor$ durch sukzessive Additionen erfolgen:

```
IF y <= x THEN BEGIN
                 z := -1;
                 w := 0;
                 WHILE w <= x DO
                   BEGIN
                     z := z + 1;
                     w := w + y;
                   END

               END
           ELSE z := 0;
```

Offensichtlich ist $z = \lfloor x/y \rfloor$. Die `WHILE`-Schleife wird $O(x/y)$-mal durchlaufen, wobei wegen $w \leq x$, $y \leq x$ und $z \leq x$ jeweils die Anzahl der Bitoperationen durch eine Größe der Ordnung $O(k) = O(size(x))$ begrenzt ist. Insgesamt ist die Anzahl der Bitoperationen nach diesem Verfahren von der Ordnung $O\left(k \cdot x/y\right)$, d.h. im ungünstigsten Fall, nämlich für kleine Werte von y, exponentiell von der Ordnung $O\left(k \cdot 2^k\right)$.

Der folgende Ansatz bringt eine Laufzeitverbesserung:

```
IF y <= x THEN BEGIN
                 v := x;
                 z := 0;
                 WHILE v >= y DO          { Schleife 1 }
                   BEGIN
                     w := y;
                     u := 0;
                     WHILE 2*w <= v DO    { Schleife 2 }
                       BEGIN
                          w := SHL (w, 1);  { w := 2 * w }
                          u := u + 1;
                       END;
                     z := z + SHL (1, u);
                     v := v - w;
                   END;
               END
           ELSE z := 0;
```

Die Korrektheit des Verfahrens sieht man wie folgt:

Schleife 1 werde t-mal durchlaufen. Die Werte der Variablen `u` am Ende der Schleife 2 seien $i_1, ..., i_t$. Dann gilt:

$2^{i_1} \cdot y \le x$ und $2^{i_1+1} \cdot y > x$,

$2^{i_2} \cdot y \le x - 2^{i_1} \cdot y$ und $2^{i_2+1} \cdot y > x - 2^{i_1} \cdot y$,

$2^{i_3} \cdot y \le x - 2^{i_1} \cdot y - 2^{i_2} \cdot y$ und $2^{i_3+1} \cdot y > x - 2^{i_1} \cdot y - 2^{i_2} \cdot y$,

...,

$2^{i_t} \cdot y \le x - 2^{i_1} \cdot y - 2^{i_2} \cdot y - ... - 2^{i_{t-1}} \cdot y$ und $2^{i_t+1} \cdot y > x - 2^{i_1} \cdot y - 2^{i_2} \cdot y - ... - 2^{i_{t-1}} \cdot y$.

Daraus folgt $i_1 > i_2 > ... i_{t-1} > i_t$, $\left(2^{i_1} + 2^{i_2} + 2^{i_3} + ... + 2^{i_t}\right) \cdot y \le x$ und

$\lfloor x/y \rfloor = 2^{i_1} + 2^{i_2} + 2^{i_3} + ... + 2^{i_t}$.

Es ist $t \in O(\log(\lfloor x/y \rfloor))$. Der Aufwand an Bitoperationen für einen Durchlauf der Schleife 2 kann durch einen Wert der Größenordnung $O(i_1 \cdot (\log(x) + \log(\lfloor x/y \rfloor)))$ abgeschätzt werden. Insgesamt ist damit die Anzahl an Bitoperationen beschränkt durch einen Ausdruck der Ordnung

$$O(t \cdot (\log(y) + i_1 \cdot (\log(x) + \log(\lfloor x/y \rfloor)) + \log(\lfloor x/y \rfloor) + \log(x))) \subseteq O((\log(\lfloor x/y \rfloor))^3 \subseteq O((\log(x))^3)).$$

Die Anzahl der erforderlichen Bitoperationen zur Bildung von $z = \lfloor x/y \rfloor$ lässt sich sogar durch Anwendung „subtilerer“ Verfahren auf die Anzahl der Bitoperationen bei der Multiplikation zurückführen[10], so dass die ganzzahlige Division ebenfalls mit einem Aufwand an Bitoperationen von der Ordnung $O(k^2) = O((\log(x)^2)$ bzw. sogar von der Ordnung $O(k \cdot \log(k) \cdot \log(\log(k)))$ durchgeführt werden kann.

Modulo-Operation $z := x \bmod y$

Wegen $x \bmod y = x - y \cdot \lfloor x/y \rfloor$ ist der Aufwand an Bitoperationen bei der Modulo-Operation von derselben Größenordnung wie die Multiplikation.

Exponentiation $z := x^y$ **und modulare Exponentiation** $z := x^y \bmod n$

Wegen $y = [y_{l-1} y_{l-2} ... y_1 y_0]_2$ ist

[10] Das Verfahren führt die Division auf die Multiplikation mit reziproken Werten zurück: $x/y = x \cdot (1/y)$ und $1/y$ kann mittels des Newtonverfahrens sehr schnell durch die Folge $(a_n)_{n \in \mathbf{N}}$ mit $a_{n+1} = a_n \cdot (2 - y \cdot a_n)$ approximiert werden. Siehe Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

$$x^y = x^{y_{l-1}\cdot 2^{l-1}+y_{l-2}\cdot 2^{l-2}+\ldots+y_1\cdot 2+y_0} = \prod_{i=0}^{l-1} x^{y_i\cdot 2^i} = \prod_{i=0}^{l-1}\left(x^{2^i}\right)^{y_i}.$$

Man könnte zur Berechnung von x^y zunächst l Zahlen e_i $(i = l-1, \ldots, 1, 0)$ bestimmen, die durch $e_i = \begin{cases} x^{2^i} & \text{für } y_i = 1 \\ 1 & \text{für } y_i = 0 \end{cases}$ definiert sind. Diese Zahlen werden dann zusammenmultipliziert. Die Berechnung einer Zahl $e_i = x^{2^i}$ könnte durch sukzessives Quadrieren erfolgen:

```
c := x;
FOR idx := 1 TO i DO c:=c*c;
e_i :=c;
```

Hierzu sind für $i = l-1, \ldots, 1, 0$ jeweils maximal i Multiplikationen, also insgesamt maximal $l(l-1)/2$ Multiplikationen, erforderlich. Anschließend müssten die l Zahlen e_i in $l-1$ Multiplikationen zusammenmultipliziert werden. Die Anzahl der Multiplikationen beträgt bei diesem Ansatz maximal $l(l-1)/2 + l - 1 = l(l+1)/2 - 1$, ist also von der Ordnung $O(l^2)$. Die Berechnung und das Zusammenmultiplizieren aller Werte e_i kann gleichzeitig erfolgen:

```
c := x;
FOR i := l-1 DOWNTO 1 DO
   BEGIN
      c:=c*c;
      IF y_{i-1} = 1 DO c := c * x;
   END;
```

Der abschließende Wert der Variablen `c` ist x^y. Offensichtlich kommt man mit $2(l-1)$, d.h. mit $O(l) = O(\log(y))$ vielen Multiplikationen aus. Belegt die größte bei dieser Berechnung auftretende Zahl höchstens h Binärstellen, so kann x^y mit höchstens $O(h^2 \cdot \log(y))$ vielen Bitoperationen berechnet werden.

Das Verfahren kann auf die Modulo-Arithmetik übertragen werden. Zur Berechnung von $x^y \bmod n$ wird dabei in obigem Verfahren jede Multiplikation `c:=c*c;` bzw. `c := c * x;` durch $c := c \cdot c \pmod n$; bzw. $c := c \cdot x \pmod n$; ersetzt. Alle bei der Berechnung auftretenden Zahlen belegen dann höchstens $h = \lfloor \log_2(n+1) \rfloor$ viele Binärstellen. Daher werden zur Berechnung von $x^y \bmod n$ höchstens $O((\log(n))^2 \cdot \log(y))$ viele Bitoperationen benötigt.

5 Anwendungsorientierte Basisdatenstrukturen

Das Thema dieses Kapitels sind Datenstrukturen mit den dazugehörigen Operationen, die in vielen Anwendungen häufig vorkommen. Sie werden in Form von Pascal-Deklarationen formuliert, da dadurch das Verständnis der Datenstrukturen erleichtert und der Anwendungsbezug deutlicher unterstrichen werden. Es gibt heute für viele Programmiersprachen mehr oder weniger umfangreiche Klassenbibliotheken, die vordefinierte Datenstrukturen bereitstellen. Beispiele sind die vordefinierten Objekttypen und die Visuellen Komponenten in Delphi ([D/K]) oder die umfangreichen Klassenbibliotheken in Java. In den folgenden Unterkapiteln soll eine Auswahl derartiger Datenstrukturen beschrieben werden, wobei der Schwerpunkt auf der Behandlung allgemeiner Basisstrukturen zur Manipulation von Klassen, bestehend aus einzelnen Datenobjekten, liegt und nicht etwa auf der Behandlung von Strukturen zur Gestaltung von Dialoganwendungen und Benutzeroberflächen.

Die Basis bildet eine Klasse von Objekten, deren innere Struktur hier nicht weiter betrachtet werden soll und die daher **Elemente** (auch **Einträge**) genannt werden. Die Objekttypdeklaration eines Elements einschließlich eines zugehörigen Pointertyps lautet

```
TYPE Pentry = ^Tentry;
     Tentry = OBJECT
                ...
              END;
```

Die Komponenten, die ein Objekt vom Objekttyp `Tentry` besitzt, und die Methoden, die auf ein derartiges Objekt anwendbar sind, sind zunächst irrelevant. Elemente können zu größeren **Datenstrukturen** zusammengefasst werden, auf denen bestimmte Operationen definiert sind, die für die Struktur charakteristisch sind. Eine derartige Struktur hat einen Objekttyp

```
TYPE TDatenstruktur = ...;
```

dessen Methoden den Zustand eines Objekts dieser Klasse, d.h. einer Zusammenfassung von Elementen, in der Regel beeinflusst, nicht aber den Zustand eines Elements in dieser Struktur (Abbildung 5-1).

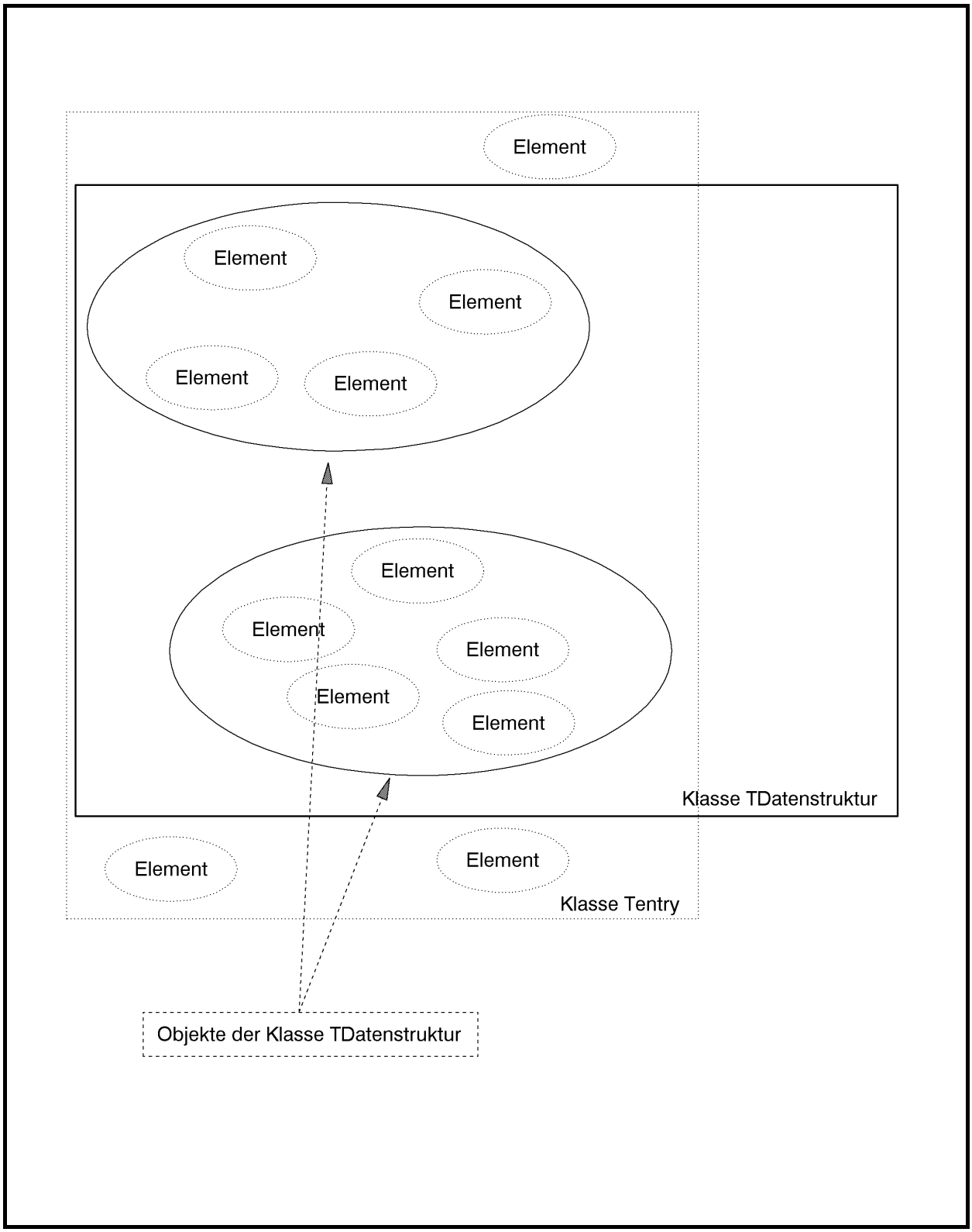


Abbildung 5-1: Elemente und Datenstrukturen

Abbildung 5-2 führt tabellarisch einige Beispiele für Elemente und Strukturen auf und benennt darauf definierte Methoden (Operationen). Die ***Methoden*** können ***in drei Typen eingeteilt*** werden:

- Methoden, die den Zustand eines Objekts des jeweiligen Objekttyps `TDatenstruktur` ändern, indem sie Elemente hinzufügen oder entfernen und damit den Zustand eines Objekts (der Datenstruktur) ändern
- Methoden, die den Zustand eines Objekts des jeweiligen Objekttyps `TDatenstruktur` anzeigen und den Zustand unverändert lassen
- Methoden, die zwei oder mehr Objekte des jeweiligen Objekttyps `TDatenstruktur` zu neuen Objekten vom Objekttyp `TDatenstruktur` verknüpfen bzw. aus einem Objekt neue Objekte generieren und dabei eventuell den Zustand des Ausgangsobjekts ändern.

Beispiele für den ersten Typ sind die Methoden ***insert*** und ***delete*** im Objekttyp `TListe`, für den zweiten Typ die Methoden ***is_member*** im Objekttyp `TListe` und ***min*** im Objekttyp `TPrio` und für den dritten Typ Methode ***concat*** zur Vereinigung zweier Objekte vom Objekttyp `TListe` und die Mengenoperationen $\cup$ und $\cap$ im Objekttyp `TMenge`.

Aus Anwendersicht ist die Implementation der Objekte in der Klasse `TDatenstruktur` ***ohne Belang. Die Möglichkeiten der Manipulationen (Operationen auf) der gesamten Struktur sind von Interesse.*** Natürlich kann eine schlechte Implementation zu einem unhandlichen und ineffizienten Verhalten führen. In den folgenden Unterkapiteln werden exemplarisch einige Implementationen vorgestellt.

Bezeichnung der Datenstruktur:	***Liste***, Objekttyp `TListe` (Kapitel 5.1.1)
Bezeichnung eines Elements:	Eintrag
Charakteristische Operationen:	
- ***init***:	Initialisieren als leere Liste
- ***insert***:	Einfügen eines Eintrags
- ***delete***:	Entfernen eines Eintrags
- ***count***:	Anzahl der Einträge in der Liste feststellen
- ***is_member***:	Feststellen, ob ein spezifizierter Eintrag in der Liste vorkommt
- ***concat***:	Anhängen einer weiteren Liste
- $\forall$**op** :	Ausführung der Operation **op** auf allen Elementen, die sich in der Liste befinden
Bemerkung: Die Datenstruktur Liste entspricht einer häufig auch als **Kollektion** bezeichneten Datenstruktur.	
Bezeichnung der Datenstruktur:	***(sortierbare) Liste***, Objekttyp `TsortiertListe`
Bezeichnung eines Elements:	Eintrag mit Sortierkriterium
Charakteristische Operationen:	wie Liste, zusätzlich:
- ***sort***:	Sortieren der Liste

Bezeichnung der Datenstruktur:	***FIFO-Warteschlange***, Objekttyp `TFIFO` (Kapitel 5.1.1)
Bezeichnung eines Elements:	Eintrag
charakteristische Operationen:	
- ***init***:	Initialisieren als leere FIFO-Warteschlange
- ***insert***:	Einfügen eines Eintrags
- ***delete***:	Entfernen des Eintrags, der sich am längsten unter allen Einträgen in der FIFO-Warteschlange befindet, und Rückgabe an den Aufrufer
- ***count***:	Anzahl der Einträge in der FIFO-Warteschlange feststellen
- ***is_member***:	Feststellen, ob ein spezifizierter Eintrag in der FIFO-Warteschlange vorkommt
- $\forall$**op** :	Ausführung der Operation **op** auf allen Elementen, die sich in der FIFO-Warteschlange befinden
Bezeichnung der Datenstruktur:	***Stack (LIFO-Warteschlange)***, Objekttyp `TStack` (Kapitel 5.1.1)
Bezeichnung eines Elements:	Eintrag
charakteristische Operationen:	
- ***init***:	Initialisieren als leeren Stack
- ***insert***:	Einfügen eines Eintrags
- ***delete***:	Entfernen des zuletzt eingefügten Eintrags unter allen Einträgen, die sich gerade im Stack befinden, und Rückgabe an den Aufrufer
- ***count***:	Anzahl der Einträge im Stack feststellen
- ***is_member***:	Feststellen, ob ein spezifizierter Eintrag im Stack vorkommt
- $\forall$**op** :	Ausführung der Operation **op** auf allen Elementen, die sich im Stack befinden
Bezeichnung der Datenstruktur:	***beschränkter Puffer***, Objekttyp `TPuffer` (Kapitel 5.1.2)
Bezeichnung eines Elements:	Eintrag
charakteristische Operationen:	
- ***init***:	Initialisieren als leere Datenstruktur mit einer festen Anzahl an Einträgen
- ***insert***:	Einfügen eines Eintrags auf die nächste freie Position; sind bereits alle Positionen belegt, wird der Eintrag, der sich am längsten im beschränkten Puffer befindet, überschrieben
- ***delete***:	Entfernen des Eintrags, der sich am längsten unter allen Einträgen im beschränkten Puffer befindet, und Rückgabe an den Aufrufer
- ***count***:	Anzahl der Einträge im beschränkten Puffer feststellen
- ***is_member***:	Feststellen, ob ein spezifizierter Eintrag im beschränkten Puffer vorkommt
- $\forall$**op** :	Ausführung der Operation **op** auf allen Elementen, die sich im beschränkten Puffer befinden

Bezeichnung der Datenstruktur:	***Hashtabelle***, Objekttyp `THashtab` (Kapitel 5.1.2)
Bezeichnung eines Elements:	Eintrag
charakteristische Operationen:	
- ***init***:	Initialisieren als leere Datenstruktur mit einer festen Größe
- ***insert***:	Einfügen eines Eintrags
- ***delete***:	Entfernen des Eintrags
- ***count***:	Anzahl der Einträge in der Hashtabelle feststellen
- ***is_member***:	Feststellen, ob ein spezifizierter Eintrag in der Hashtabelle vorkommt
- $\forall$**op** :	Ausführung der Operation **op** auf allen Elementen, die sich in der Hashtabelle befinden
Bezeichnung der Datenstruktur:	***Prioritätsschlange***, Objekttyp `TPrio` (Kapitel 5.2.1)
Bezeichnung eines Elements:	Eintrag mit Sortierkriterium
charakteristische Operationen:	
- ***init***:	Initialisieren als leere Prioritätsschlange
- ***insert***:	Einfügen eines Eintrags
- ***delete***:	Entfernen des spezifizierten Eintrags
- ***count***:	Anzahl der Einträge in der Prioritätsschlange feststellen
- ***is_member***:	Feststellen, ob ein spezifizierter Eintrag in der Prioritätsschlange vorkommt
- ***min***:	Ermitteln des Eintrags mit dem kleinsten Wert des Sortierkriteriums unter allen Einträgen, die sich gerade in der Prioritätsschlange befinden
Bezeichnung der Datenstruktur:	***Menge***, Objekttyp `TMenge` (Kapitel 5.1.3)
Bezeichnung eines Elements:	Element
charakteristische Operationen:	
- ***init***:	Initialisieren als leere Menge
- ***insert***:	Einfügen eines Elements in eine Menge, falls es dort noch nicht vorkommt
- ***delete***:	Entfernen eines spezifizierten Elements aus einer Menge, falls es dort vorkommt
- ***count***:	Anzahl der Elemente in der Menge feststellen
- $\in$	Feststellen, ob ein spezifiziertes Element in der Menge vorkommt
- $\subseteq$ bzw. =	Feststellen, ob eine Menge in einer anderen Menge enthalten ist bzw. ob zwei Mengen gleich sind
- $\cup$, $\cap$	Vereinigungs- und Schnittmengenbildung

Bezeichnung der Datenstruktur:	***(gerichteter oder ungerichteter oder gewichteter) Graph, Spezialisierung auf Bäume bzw. Binärbäume bzw. höhenbalancierte Bäume***, Objekttyp `TGraph` (Kapitel 5.2.3, 7.2, 7.3, 7.4)
Bezeichnung eines Elements:	Knoten, Kante (Knotenpaar)
charakteristische Operationen:	
- ***init***:	Initialisieren als leerer Graph
- ***insert_node***:	Einfügen eines neuen Knotens
- ***insert_edge***:	Einfügen einer neuen Kante
- ***delete_node***:	Entfernen eines spezifizierten Knotens und der mit ihm inzidierenden Kanten
- ***delete_edge***:	Entfernen einer spezifizierten Kante
- ***b_search***:	Durchlaufen des Graphen, beginnend bei einem spezifizierten Knoten, gemäß Breitensuche und Ermittlung der Kantenreihenfolge
- ***d_search***:	Durchlaufen des Graphen, beginnend bei einem spezifizierten Knoten, gemäß Tiefensuche und Ermittlung der Kantenreihenfolge
- ***tour***	Ermittlung einer Tour mit minimalem Gesamtgewicht der durchlaufenen Kanten
- ***span_tree***	Ermittlung eines aufspannenden Baums mit minimalem Gewicht
- ***min***:	Ermitteln der kürzesten Wegs zwischen zwei Knoten
- ***min_all***:	Ermitteln aller kürzesten Wege zwischen je zwei Knoten
Bezeichnung der Datenstruktur:	***Matrix***, Objekttyp `TMatrix` (in Basisform Kapitel 5.1.2)
Bezeichnung eines Elements:	Eintrag
charakteristische Operationen:	
- ***init***:	Initialisieren mit einem Anfangswert
- ***trans***:	Transponieren
- ***invert***:	Invertieren
- ***determ***:	Ermittlung der Determinante
- ***Eigenwert***:	Ermittlung des Eigenwerts
- ***arithm***:	Diverse arithmetische Matrixoperationen

Abbildung 5-2: Beispiele für Elemente und Strukturen

Zur Konkretisierung wird die Gesamtaufgabe in einzelne UNITs eingeteilt. Die Anwendung ist in Form eines Programms definiert, in der die oben beschriebenen Elemente vorkommen. Die Deklaration des Objekttyps eines Elements steht in der

```
UNIT Element;

INTERFACE

   { Objekttypdeklarationen (Bezeichner, Methoden) eines Elements: }
   TYPE Pentry = ^Tentry; { Pointer auf ein Element }
        Tentry = OBJECT
                   ...
                   CONSTRUCTOR init ( ... );
                     ...
                   PROCEDURE display; VIRTUAL;
                     ...
                   DESTRUCTOR  done; VIRTUAL;
                 END;


IMPLEMENTATION

CONSTRUCTOR Tentry.init ( ... );
  BEGIN { Tentry.init }
    ...
  END   { Tentry.init };

...

PROCEDURE Tentry.display;
  BEGIN { Tentry.display }
    ...
  END   { Tentry.display };

...

DESTRUCTOR Tentry.done;
  BEGIN { Tentry.done }
    ...
  END   { Tentry.done };

END.
```

Neben dem Objekttyp eines Elements wird ein Pointertyp auf ein Element deklariert, da dieser Pointertyp in den Anwendungen und Implementierungen der Datenstrukturen häufig benötigt wird. Als Methode des Objekttyps `Tentry` ist neben dem Konstruktor `Tentry.init`

(eine Formalparameterliste ist angedeutet) und dem Destruktor `Tentry.done`[11] eine Methode `Tentry.display` deklariert, die den Zustand eines Objekts, d.h. die Werte seiner Komponenten, anzeigt.

Die Deklaration (Bezeichner, Operationen usw.) und die Implementierung der Datenstruktur, zu der die Elemente zusammengefasst werden, befindet sich ebenfalls in einer UNIT, die im Prinzip folgendes Layout hat. Zu beachten ist, dass die Implementierung der Details nach außen verborgen wird:

```
UNIT Struktur;
      { Deklaration der gesamten Struktur,
        zu der Elemente zusammengefasst werden }

INTERFACE

  USES Element;

   { Objekttypdeklarationen (Bezeichner, Methoden) der Struktur: }
   TYPE PDatenstruktur = ^TDatenstruktur;
        TDatenstruktur = OBJECT
                           ...
                         END;

IMPLEMENTATION

  { Implementierung der Methoden }
  ...

END.
```

Das Anwendungsprogramm bindet dann diese UNITs ein:

```
PROGRAM Anwender;

  USES Element,
       Struktur;

  ...

BEGIN { Anwender }
  ...
END   { Anwender }.
```

---

11 Der Destruktor ist als virtuelle Methode definiert, da dies die Implementation eines Destruktors für abgeleitete Objekttypen erleichtert.

5.1 Lineare Datenstrukturen

Unter einer linearen Datenstruktur soll eine Datenstruktur verstanden werden, auf der die folgenden Operationen definiert sind:

- ***init***: Initialisieren der Datenstruktur als leer
- ***insert***: Einfügen eines weiteren Elements in die Datenstruktur
- ***delete***: Entfernen eines Elements aus der Datenstruktur, wobei definiert ist, ob ein genau bezeichnetes oder ein beliebiges Element entfernt wird; das zu entfernende Element wird je nach Definition der Operation an den Aufrufer übergeben
- ***op***: eventuell ***weitere Operationen, die nicht voraussetzen, dass zwischen den Elementen der Datenstruktur irgendwelche Ordnungsbeziehungen bestehen***.

Beispiele sind die in Abbildung 5-2 genannten Datenstrukturen Liste, FIFO-Warteschlange, Stack und Menge. Es sei darauf hingewiesen, dass der Begriff „lineare Datenstruktur“ in der Literatur unterschiedlich definiert oder mit dem Begriff „Liste“ synonym verwendet wird. Der Begriff assoziiert natürlich eine gängige Realisierungsform, nämlich als tabellarische oder als eine über Adressverweise verkettete Liste.

Einige Operationen verändern den gegenwärtigen Zustand eines Objekts der Datenstruktur. Um die korrekte Implementierung zu kontrollieren, wird eine zusätzliche Operation ***show*** definiert, die den gegenwärtigen Zustand eines Objekts der Datenstruktur anzeigt. Diese Operation beeinflusst den Zustand eines Objekts nicht.

5.1.1 Grundlegende Listenstrukturen

Zu den grundlegenden Listenstrukturen werden in diesem Kapitel die Datenstrukturen Liste, FIFO-Warteschlange und Stack zusammengefasst. Diese drei Datenstrukturen verfügen über verwandte Methoden, wobei die Basis der Methoden in der Datenstruktur Liste gelegt wird.

Für alle drei Datenstrukturen sind die Operationen ***init***, ***insert***, ***delete***, ***count***, ***is_member***, $\forall$**op** und ***show*** definiert (siehe Abbildung 5-2 und Kapitel 5.1), jedoch teilweise mit unterschiedlicher Wirkung. Des weiteren ist eine Methode ***choose*** definiert, die es erlaubt, die jeweilige Datenstruktur sequentiell zu durchlaufen und dabei einen Verweis auf ein gegenwärtig in der Datenstruktur befindliches Element zurückzuliefern. Hinzu kommt der Destruktor für den jeweiligen Objekttyp. Weiterhin verfügt die Datenstruktur Liste über eine Methode ***concat***, die für die beiden anderen Datenstrukturen nicht definiert ist.

Die Implementierungen aller drei Datenstrukturen werden in einer `UNIT Listenstrukturen` zusammengefasst. Dabei ist der Objekttyp `TListe` Grundlage für die abgeleiteten Objekttypen zur Definition der Datenstrukturen FIFO-Warteschlange (Objekttyp `TFIFO`) und Stack (Objekttyp `TStack`). Im Anschluss an den Programmcode werden Erläuterungen gegeben.

```
UNIT Listenstrukturen;

INTERFACE

  USES Element;

  TYPE Tfor_all_proc = PROCEDURE (entry_ptr : Pentry);
       TList_flag    = (c_first, c_next);

       PListe = ^TListe;
       TListe = OBJECT
                  PRIVATE
                    anz        : INTEGER;
                               { Anzahl der Listeneinträge       }
                    first      : Pointer;
                               { Anfang der Verkettung der
                                 Verweise auf Einträge der Liste }
                    lastchosen : Pointer;
                               { Eintrag, der beim letzten Aufruf
                                 der Methode choose ermittelt wurde}
                  PUBLIC
                    CONSTRUCTOR init;
                    PROCEDURE insert (entry_ptr : Pentry);
                    PROCEDURE delete (entry_ptr : Pentry);
                    FUNCTION count : INTEGER;
                    FUNCTION is_member (entry_ptr : Pentry)
                                                   : BOOLEAN;
                    PROCEDURE for_all (proc : Tfor_all_proc);
                    PROCEDURE concat (Liste_ptr : PListe);
                    PROCEDURE show; VIRTUAL;
                    FUNCTION choose (param : TList_flag) : Pentry;
                    DESTRUCTOR done; VIRTUAL;
                END;


       PFIFO = ^TFIFO;
       TFIFO = OBJECT (TListe)
                 PRIVATE
                   last  : Pointer;  { Ende der Verkettung        }
                 PUBLIC
                   CONSTRUCTOR init;
                   PROCEDURE insert (entry_ptr : Pentry);
                   PROCEDURE delete (VAR entry_ptr : Pentry);
                   PROCEDURE concat (Liste_ptr : PListe);
                                              { "Dummy-Methode" }
                   PROCEDURE choose;          { "Dummy-Methode" }
                   PROCEDURE show; VIRTUAL;
               END;
```

```
       PStack = ^TStack;
       TStack = OBJECT (Tliste)
                   PROCEDURE delete (VAR entry_ptr : Pentry);
                   PROCEDURE concat (Liste_ptr : PListe);
                                                { "Dummy-Methode" }
                   PROCEDURE choose;            { "Dummy-Methode" }
                END;


{ ************************************************************** }

IMPLEMENTATION

TYPE Plistentry = ^Tlistentry;
     Tlistentry = RECORD
                    entry_ptr : Pentry;
                    next      : Plistentry;
                    last      : Plistentry;
                  END;


CONSTRUCTOR TListe.init;

  BEGIN { TListe.init };
    first      := NIL;
    lastchosen := NIL;
    anz        := 0;
  END   { TListe.init };


PROCEDURE TListe.insert (entry_ptr : Pentry);

VAR p : PListentry;

  BEGIN { TListe.insert }
    New (p);
    p^.next      := first;
    p^.entry_ptr := entry_ptr;
    Inc(anz);
    first := p;
  END   { TListe.insert };


PROCEDURE TListe.delete (entry_ptr : Pentry);

VAR p           : PListentry;
    q           : PListentry;
    dispose_ptr : PListentry;

  BEGIN { TListe.delete }
    p := first;
    q := NIL;
    WHILE p <> NIL DO
      IF p^.entry_ptr = entry_ptr
      THEN BEGIN
             dispose_ptr := p;
```

```
            IF q = NIL THEN first   := p^.next
                       ELSE q^.next := p^.next;
            p := p^.next;
            IF dispose_ptr = lastchosen THEN lastchosen := q;
            Dispose (dispose_ptr);
            Dec (anz);
          END
          ELSE BEGIN
                 q := p;
                 p := p^.next;
               END;
  END   { TListe.delete };


FUNCTION TListe.count : INTEGER;

   BEGIN { TListe.count }
     count := anz;
   END   { TListe.count };


FUNCTION TListe.is_member (entry_ptr : Pentry) : BOOLEAN;

VAR p : PListentry;

  BEGIN { TListe.is_member }
    p         := first;
    is_member := FALSE;
    WHILE p <> NIL DO
      IF p^.entry_ptr = entry_ptr
      THEN BEGIN
             is_member := TRUE;
             Break;
           END
      ELSE p := p^.next;
  END   { TListe.is_member };


PROCEDURE TListe.concat (Liste_ptr : PListe);

VAR p : PListentry;

  BEGIN { TListe.concat }
    IF first = NIL
    THEN first := Liste_ptr^.first
    ELSE BEGIN
           p := first;
           WHILE p^.next <> NIL DO
             p := p^.next;
           p^.next := Liste_ptr^.first;
         END;
    anz := anz + Liste_ptr^.anz;
    Dispose (Liste_ptr, done);
  END   { TListe.concat };


PROCEDURE TListe.for_all (proc : Tfor_all_proc);
```

```
VAR p : PListentry;

  BEGIN { TListe.for_all }
    p := first;
    WHILE p <> NIL DO
      BEGIN
        proc(p^.entry_ptr);
        p := p^.next;
      END;
  END   { TListe.for_all };


PROCEDURE TListe.show;

VAR p : PListentry;

  BEGIN { TListe.show }
    p := first;
    WHILE p <> NIL DO
      BEGIN
        p^.entry_ptr^.display;
        p := p^.next;
      END;
  END   { TListe.show };



FUNCTION TListe.choose (param : TList_flag) : Pentry;

   BEGIN { TListe.choose }
     IF (param = c_first) OR (lastchosen = NIL)
     THEN lastchosen := first
     ELSE lastchosen := PListentry(lastchosen)^.next;
     IF lastchosen = NIL
     THEN choose := NIL
     ELSE choose := PListentry(lastchosen)^.entry_ptr;
   END   {TListe.choose };


DESTRUCTOR TListe.done;

VAR p : PListentry;
    q : PListentry;

  BEGIN { TListe.done }
    p := first;
    WHILE p <> NIL DO
      BEGIN
        q := p;
        p := p^.next;
        Dispose(q);
      END;
  END   { TListe.done };


{ ---------------------------------------------------------------- }
```

```
CONSTRUCTOR TFIFO.init;

  BEGIN { TFIFO.init };
    INHERITED init;
    last := NIL;
  END   { TFIFO.init };


PROCEDURE TFIFO.insert (entry_ptr : Pentry);

  BEGIN { TFIFO.insert }
    INHERITED insert (entry_ptr);
    Plistentry(first)^.last := NIL;
    IF last = NIL THEN last := first
                  ELSE Plistentry(first)^.next^.last := first;
  END   { TFIFO.insert };


PROCEDURE TFIFO.delete (VAR entry_ptr : Pentry);

VAR p : Plistentry;

  BEGIN { TFIFO.delete }
    IF last <> NIL
    THEN BEGIN
           entry_ptr := Plistentry(last)^.entry_ptr;
           p         := last;
           last      := Plistentry(last)^.last;
           Dispose (p);
           Dec (anz);
           IF last = NIL THEN first := NIL
                         ELSE Plistentry(last)^.next := NIL;
         END
    ELSE entry_ptr := NIL;
  END   { TFIFO.delete };


PROCEDURE TFIFO.concat (Liste_ptr : PListe);

  BEGIN { TFIFO.concat }
  END   { TFIFO.concat };


PROCEDURE TFIFO.choose;

  BEGIN { TFIFO.choose }
  END   { TFIFO.choose };


PROCEDURE TFIFO.show;

VAR p : Plistentry;

  BEGIN { TFIFO.show }
    p := last;
```

```
    WHILE p <> NIL DO
      BEGIN
        p^.entry_ptr^.display;
        p := p^.last;
      END;
  END   { TFIFO.show };


{ ---------------------------------------------------------------- }


PROCEDURE TStack.delete (VAR entry_ptr : Pentry);

VAR p : PListentry;

  BEGIN { TStack.delete }
    IF first <> NIL
    THEN BEGIN
           entry_ptr := PListentry(first)^.entry_ptr;
           p         := first;
           first     := PListentry(first)^.next;
           Dispose (p);
           Dec (anz);
         END
    ELSE entry_ptr := NIL;
  END   { TStack.delete };


PROCEDURE TStack.concat;

  BEGIN { TStack.concat }
  END   { TStack.concat };

PROCEDURE TStack.choose;

  BEGIN { TStack.choose }
  END   { TStack.choose };

{ ***************************************************************** }


END.
```

A. Datenstruktur Liste (Objekttyp `TListe`)

Eine Liste könnte als Datenstruktur aufgefasst werden, die die in ihr eingetragenen Elemente auch wirklich physisch enthält. In diesem Fall müssten die Elemente beim Eintragen von ihrem bisherigen Ort im System in die Liste transferiert werden (dieses Konzept ist mit implementationstechnischen Details im oberen Teil von Abbildung 5.1.1-1 dargestellt). Ein Element könnte sich dann nur höchstens in einer Liste befinden. Alternativ könnte man beim Eintragen eines Elements eine Kopie in die Liste übernehmen, so dass sich ein Element konzeptionell in mehreren Listen aufhalten könnte. In jedem Fall benötigt man für die Listenope-

rationen genauere Informationen über die Struktur eines Elements, zumindest Informationen über den für ein Element erforderlichen Speicherplatz in der Liste. Um hier größtmögliche Flexibilität zu wahren und die Methoden der Datenstruktur Liste möglichst einfach zu halten, werden die Einträge in einer Liste auf einheitliche Weise unabhängig vom Layout eines einzelnen Eintrags verwaltet: Die Liste enthält dabei *nicht* die in ihr eingetragenen Elemente, sondern lediglich Adressverweise auf diese Elemente. Dabei kommt es nicht darauf an, welche Struktur die eingetragenen Elemente wirklich besitzen.

Da keine Beziehungen zwischen den Einträgen einer Liste bestehen und auch die potentielle Anzahl an Listeneinträgen aus Sicht der Liste nicht bekannt ist, erscheint es angebracht, die Adressverweise auf die in einem Objekt der Datenstruktur Liste eingetragenen Elemente als verkettete Folge von dynamisch angelegten Einträgen zu organisieren. Im unteren Teil von Abbildung 5.1.1-1 ist das entsprechende Layout zu sehen. Dabei wird im Objekttyp `TListe` die Anzahl der Einträge in einer Liste und eine verkettete Folge von Adressverweisen auf die Einträge in der Liste implementiert; die Einträge selbst liegen nicht im Datenobjekt vom Typ `TListe`. Das Erzeugen und Vernichten eines Datenobjekt vom Typ `Tentry` liegt in der Verantwortung des Anwenders und wird vom Objekttyp `TListe` und seinen Methoden nicht kontrolliert.

Des weiteren ist zu beachten, dass in obiger Implementierung für die Adressverkettung innerhalb einer Liste sowohl eine Vorwärts- als auch eine Rückwärtsverkettung vorgesehen sind. Für die Datenstrukturen Liste und Stack wird nur die Vorwärtsverkettung benötigt; die Rückwärtsverkettung wird daher in Abbildung 5.1.1-1 nicht gezeigt. Die Rückwärtsverkettung wird zusätzlich für die Datenstruktur FIFO-Warteschlange benötigt, die sich aus der Datenstruktur Liste ableitet.

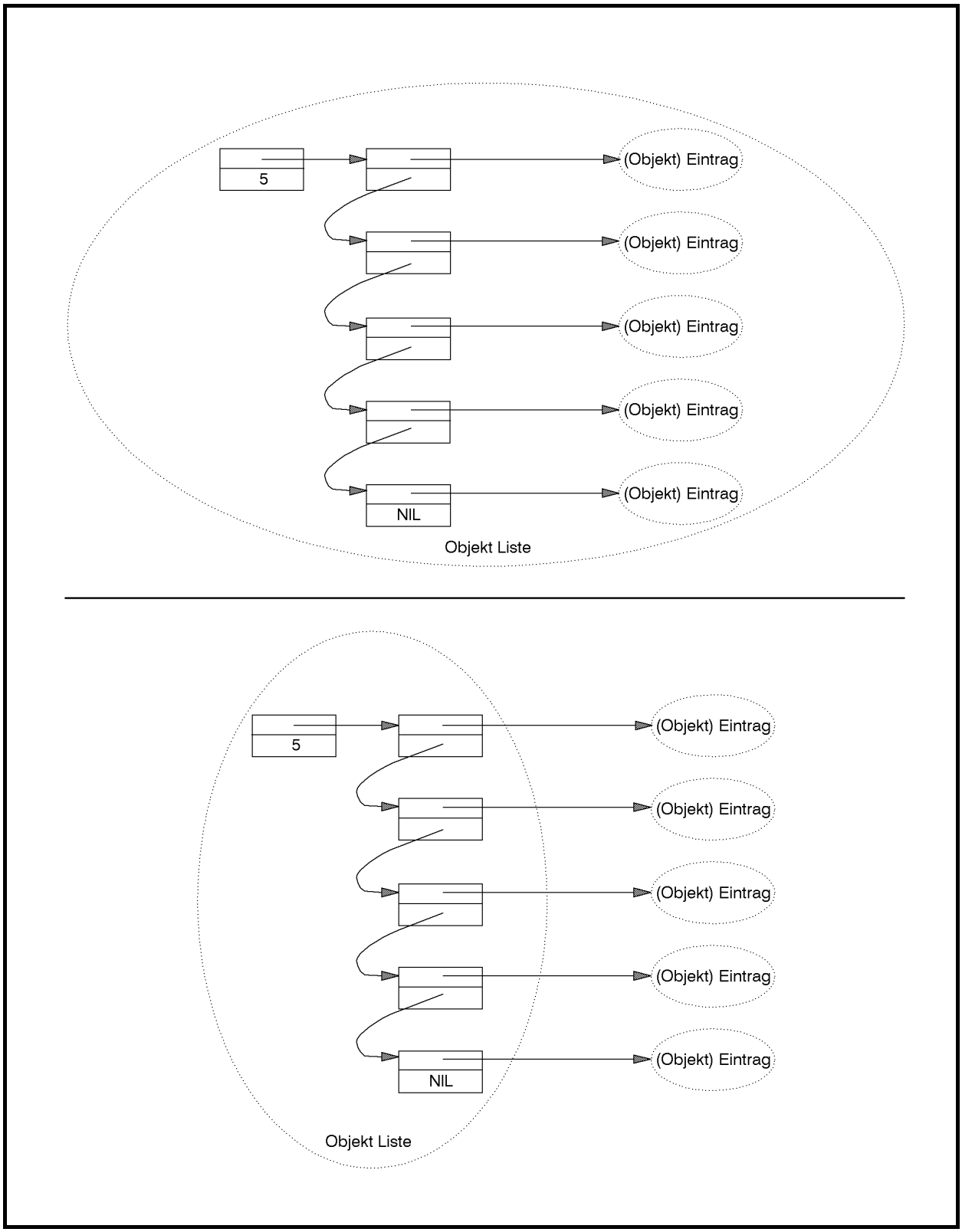


Abbildung 5.1.1-1: Liste

Die folgende Tabelle beschreibt die Benutzerschnittstelle des Objekttyps `TListe`.

Methode	Bedeutung
`CONSTRUCTOR Tliste.init;`	Die Liste wird als leere Liste initialisiert.
`PROCEDURE TListe.insert (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	Ein neues Element wird in die Liste aufgenommen.
`PROCEDURE TListe.delete (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu entfernende Element	Alle Vorkommen eines Elements werden aus der Liste entfernt.
`FUNCTION TListe.count: INTEGER;`	Die Funktion liefert die Anzahl der Einträge, die gegenwärtig in der Liste stehen.
`FUNCTION TListe.is_member (entry_ptr : Pentry) : BOOLEAN;` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu überprüfende Element	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element in der Liste vorkommt, ansonsten den Wert `FALSE`.
`PROCEDURE TListe.concat (Liste_ptr : PListe);` Bedeutung des Parameters: `Liste_ptr` Verweis auf die anzuhängende Liste	Hängt eine Liste an.
`PROCEDURE TListe.for_all (proc : Tfor_all_proc);` Bedeutung des Parameters: `proc` anzuwendende Elementmethode	Auf alle zur Zeit in der Liste eingetragenen Elemente wird die Operation (Elementmethode) `proc` angewendet
`PROCEDURE TListe.show;`	Alle zur Zeit in der Liste vorkommenden Elemente werden angezeigt.
`FUNCTION Tliste.choose (param : TList_flag) : Pentry;`	Bei jedem Aufruf wird ein Verweis auf einen Listeneintrag geliefert, wobei der Aktualparameterwert `c_first` angibt, ob der erste Eintrag ermittelt werden soll, oder der nächste Eintrag seit dem letzten Aufruf der Funktion (Aktualparameterwert `c_next`).
`DESTRUCTOR TListe.done;`	Die Liste wird aus dem System entfernt.

Das Einfügen eines Eintrags in die Datenstruktur Liste (Methode `TListe.insert`) erfordert die Erzeugung eines neuen Adressverweises auf den Eintrag und sein Einhängen in die Kette der Adressverweise auf die bisherigen Einträge. Günstigerweise wird der neue Adressverweis an den Anfang der Kette gehängt. Der Methode `TListe.insert` wird nicht das Objekt (Element) selbst, sondern ein Verweis auf das Objekt übergeben. Ein Element kann mehrmals in eine Liste eingefügt werden. Zur Entfernung eines Eintrags (Methode `TListe.delete`) wird zunächst der erste Verweis gesucht, der auf das Objekt (Element) zeigt, und dann entfernt. Al-

le weiteren Vorkommen eines Elements in der Liste werden entfernt. Die Methode `TListe.is_member` zur ***is_member***-Operation stellt fest, ob es einen Verweis in der Liste auf ein spezifiziertes Element gibt. Allen drei Methoden wird also das spezifizierte Element nicht direkt, sondern als Pointer auf das Element übergeben.

Die Implementierung der Operation $\forall$**op** erfolgt in der Methode `TListe.for_all`. Es wird vorausgesetzt, dass beim Aufruf dieser Methode in der Anwendung als Aktualparameter für den Formalparameter `proc` eine Prozedur vom Typ `Tfor_all_proc` übergeben wird, die innerhalb des aufrufenden Blocks deklariert ist.

Die ***choose***-Operation wird in der Methode `TListe.choose` implementiert. Die Komponente `lastchosen` zeigt beim Aufruf der Methode auf den Listeneintrag, der zuletzt ausgewählt wurde bzw. hat den Wert `NIL`, wenn noch kein Listeneintrag gewählt wurde. In der Methode `TListe.delete` wird berücksichtigt, dass im Falle des Entfernens des zuletzt ausgewählten Listeneintrags (`dispose_ptr = lastchosen`) die Komponente `lastchosen` auf den vorhergehenden Listeneintrag zurückgesetzt wird.

Die folgende Tabelle gibt eine Obergrenze für den Aufwand an, den ein jeweiliger Methodenaufruf benötigt, falls die Liste bereits n Elemente enthält.

Methode	Aufwand (worst case)
`TListe.init;` (bei leerer Liste)	$O(1)$
`Tliste.insert;`	$O(1)$
`TListe.delete;`	$O(n)$; dieser Aufwand ergibt sich (im ungünstigsten Fall) auch, wenn nur das erste Vorkommen des Elements entfernt wird
`TListe.count;`	$O(1)$
`TListe.is_member;`	$O(n)$
`TListe.for_all;`	$O(n \cdot A)$, wobei A den Aufwand für die einmalige Ausführung der Operation `proc` bezeichnet
`TListe.concat;`	$O(n)$
`TListe.show;`	$O(n)$
`TListe.choose;`	$O(1)$
`TListe.done;`	$O(n)$

B. Datenstruktur FIFO-Warteschlange (Objekttyp `TFIFO`)

Eine FIFO-Warteschlange verhält sich ähnlich einer Liste, nur wird jetzt ein neuer Eintrag durch die Operation ***insert*** so in ein Objekt eingefügt, dass ein Aufruf einer anschließenden Operation ***delete*** denjenigen Eintrag an den Aufrufer zurückliefert, der sich am längsten in der FIFO-Warteschlange aufhält. Um diese Bedingung zu erfüllen, wird ein neu aufzunehmendes Element stets an das Ende der aktuellen FIFO-Warteschlange angefügt, während die Operati-

on ***delete*** vom Anfang der FIFO-Warteschlange entfernt. Daher wird für diese Datenstruktur eine vorwärts und rückwärts gerichtete Adressverkettung aufgebaut. Der Aufwand, den ein Methodenaufruf `TFIFO.delete` benötigt, falls die FIFO-Warteschlange bereits *n* Elemente enthält, ist nun konstant (im Gegensatz zum entsprechenden Aufruf `TListe.delete`).

Der Objekttyp `TFIFO`, der den Datentyp FIFO-Warteschlange realisiert, leitet sich vom Objekttyp `TListe` ab, wobei die Methoden zur Implementierung der Operationen ***insert***, ***delete***, ***show*** und ***concat*** überschrieben werden, um das spezifische Verhalten einer FIFO-Warteschlange zu realisieren. Damit für eine FIFO-Warteschlange eine Operation ***concat***, wie sie für eine Liste definiert ist, nicht ausführbar ist, wird eine Methode `TFIFO.concat` definiert, deren Implementierung jedoch keine Statements enthält, die aber die von `TListe` geerbte Methode `TListe.concat` überschreibt und dadurch „unwirksam“ macht.

Die folgende Tabelle beschreibt die Benutzerschnittstelle des Objekttyps `TFIFO`.

Methode	Bedeutung
`CONSTRUCTOR TFIFO.init;`	Die FIFO-Warteschlange wird als leer initialisiert.
`PROCEDURE TFIFO.insert (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	Ein neues Element wird in die FIFO-Warteschlange aufgenommen.
`PROCEDURE TFIFO.delete (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu entfernende Element	Das Element, das sich am längsten in der FIFO-Warteschlange aufhält, wird an den Aufrufer zurückgeliefert und aus der FIFO-Warteschlange entfernt.
`FUNCTION TFIFO.count: INTEGER;`	Die Funktion liefert die Anzahl der Einträge, die gegenwärtig in der FIFO-Warteschlange stehen.
`FUNCTION TFIFO.is_member (entry_ptr : Pentry) : BOOLEAN;` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu überprüfende Element	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element in der FIFO-Warteschlange vorkommt, ansonsten den Wert `FALSE`.
`PROCEDURE TFIFO.for_all (proc : Tfor_all_proc);` Bedeutung des Parameters: `proc` anzuwendende Elementmethode	Auf alle zur Zeit in der FIFO-Warteschlange eingetragenen Elemente wird die Operation (Elementmethode) `proc` angewendet
`PROCEDURE TFIFO.show;`	Alle zur Zeit in der FIFO-Warteschlange vorkommenden Elemente werden angezeigt.
`DESTRUCTOR TFIFO.done;`	Die FIFO-Warteschlange wird aus dem System entfernt.

Abbildung 5.1.1-2 zeigt die interne Struktur der Implementierung eines Objekts vom Objekttyp `TFIFO`. Sie besteht aus zwei Verweisen, nämlich auf den Anfang bzw. das Ende der Liste, einer Komponente, die die aktuelle Anzahl an Einträgen festhält, und jeweils vor- und rückwärtsverketteten Adressverweisen auf die Elemente der FIFO-Warteschlange. Die Rückwärtsverkettung wird in `TFIFO.delete` verwendet, um direkt den nun als nächsten zu entfernenden Eintrag zu erhalten.

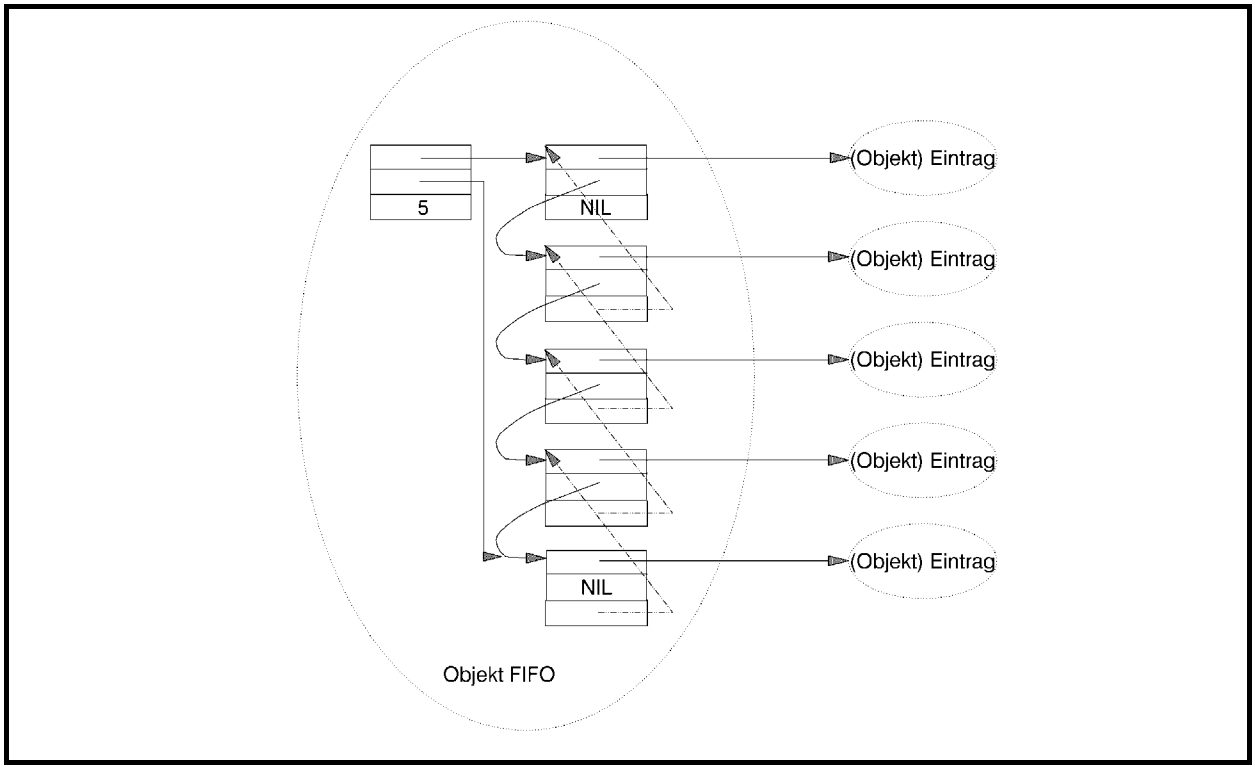


Abbildung 5.1.1-2: FIFO-Warteschlange

Der Aufwand, den ein jeweiliger Methodenaufruf benötigt, falls die FIFO-Warteschlange bereits *n* Elemente enthält, entspricht bis auf die Ausführung der Methode `TFIFO.delete` dem entsprechenden Aufwand bei `TListe`. Ein Aufruf der Methode `TFIFO.delete` ist von der Ordnung $O(1)$.

C. Datenstruktur Stack (Objekttyp `TStack`)

Die Datenstruktur Stack behandelt ihre Einträge nach dem last-in-first-out-Prinzip, d.h. derjenige Eintrag wird durch die Operation ***delete*** entfernt, der zuletzt eingetragen wurde.

Die interne Struktur eines Objekts vom Objekttyp `TStack` besteht aus einem Verweis, nämlich auf den Anfang der Liste, einer Komponente, die die aktuelle Anzahl an Einträgen festhält, und jeweils vorwärtsverketteten Adressverweisen auf die Elemente des Stacks wie im Objekttyp `TListe`. Die Methode `TListe.concat` wird wieder durch eine „leere“ Methode `TStack.concat` ersetzt, weil sie konzeptionell in der Datenstruktur Stack nicht vorkommt.

Die Benutzerschnittstelle des Objekttyps `TStack` lautet:

Methode	Bedeutung
`CONSTRUCTOR TStack.init;`	Der Stack wird als leer initialisiert.
`PROCEDURE TStack.insert (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	Ein neues Element wird in den Stack aufgenommen.
`PROCEDURE TStack.delete (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu entfernende Element	Das Element, das zuletzt in den Stack eingefügt wurde, wird an den Aufrufer zurückgeliefert und aus dem Stack entfernt.
`FUNCTION Tstack.count: INTEGER;`	Die Funktion liefert die Anzahl der Einträge, die gegenwärtig im Stack stehen.
`FUNCTION TStack.is_member (entry_ptr : Pentry) : BOOLEAN;` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu überprüfende Element	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element im Stack vorkommt, ansonsten den Wert `FALSE`.
`PROCEDURE TStack.for_all (proc : Tfor_all_proc);` Bedeutung des Parameters: `proc` anzuwendende Elementmethode	Auf alle zur Zeit im Stack eingetragenen Elemente wird die Operation (Elementmethode) `proc` angewendet
`PROCEDURE TStack.show;`	Alle zur Zeit im vorkommenden Elemente werden angezeigt.
`DESTRUCTOR TStack.done;`	Der Stack wird aus dem System entfernt.

Bezüglich des Aufwands, den ein einziger Methodenaufruf benötigt, falls der Stack bereits *n* Elemente enthält, gelten dieselben Abschätzungen wie bei der FIFO-Warteschlange.

5.1.2 ARRAY-basierte Listenstrukturen

In Kapitel 5.1.1 wurden Beispiele linearer Datenstrukturen vorgestellt, die auf dem Objekttyp `TListe` basieren und die Handhabung variabel langer Listen ermöglichen, indem nach Bedarf dynamisch neue Listeneinträge erzeugt oder bisherige Listeneinträge entfernt werden. Im vorliegenden Kapitel werden Beispiele linearer Datenstrukturen behandelt, die auf einer Implementierung als `ARRAY` basieren, nämlich die Datenstrukturen Vektor, Matrix, beschränkter Puffer und Hashtabelle. Wegen ihrer strukturellen Ähnlichkeit werden sie in einer `UNIT ARRAY_basierte_Liste` implementiert; anschließend werden wieder Erläuterungen gegeben:

```
UNIT ARRAY_basierte_Liste;

INTERFACE

USES Element,
     Listenstrukturen;

CONST max_dimension = ...; { maximale Dimension einer Matrix  }

TYPE PVektor = ^TVektor;
     TVektor = OBJECT
                 PRIVATE
                   dimension : INTEGER;
                   { ... des Vektors                  }
                   size      : INTEGER;
                   { Größe (in Bytes) eines Eintrags   }
                   vektor    : Pointer;
                   { Verweis auf den Vektor            }
                 PUBLIC
                   CONSTRUCTOR init (n      : INTEGER;
                                     VAR OK : BOOLEAN);
                   FUNCTION v (i : INTEGER) : INTEGER;
                   PROCEDURE setze (i    : INTEGER;
                                    wert : INTEGER);
                   DESTRUCTOR done;
               END;


     PPointerVektor = ^TPointerVektor;
     TPointerVektor = OBJECT (TVektor)
                        PUBLIC
                          FUNCTION v (i : INTEGER) : Pointer;
                          PROCEDURE setze (i    : INTEGER;
                                           wert : Pointer);
                      END;


     PMatrix = ^TMatrix;
     TMatrix = OBJECT
                 PRIVATE
                   dimension : INTEGER;
                   { ... der Matrix                  }
                   zeilen    : Pointer;
                   { Verweis auf das ARRAY der Zeilen }
                 PUBLIC
                   CONSTRUCTOR init (n      : INTEGER;
                                     VAR OK : BOOLEAN);
                   FUNCTION m (i : INTEGER;
                               j : INTEGER) : INTEGER;
                   PROCEDURE setze (i    : INTEGER;
                                    j    : INTEGER;
                                    wert : INTEGER);
                   DESTRUCTOR done;
               END;


     PPointerMatrix = ^TPointerMatrix;
```

```
TPointerMatrix = OBJECT (TMatrix)
                  PUBLIC
                    FUNCTION m (i : INTEGER;
                                j : INTEGER) : Pointer;
                    PROCEDURE setze (i    : INTEGER;
                                     j    : INTEGER;
                                     wert : Pointer);
                 END;


PPuffer = ^TPuffer;
TPuffer = OBJECT (TPointerVektor)
            PRIVATE
              start_idx  : INTEGER;
             { Position des gerade ersten Eintrags im Puffer }
              end_idx    : INTEGER;
             { Position des ersten freien Eintrags im Puffer }
             empty_flag : BOOLEAN;
             { Anzeige eines leeren Puffers                  }
            PUBLIC
              CONSTRUCTOR init (m      : INTEGER;
                                VAR OK : BOOLEAN);
              PROCEDURE insert (entry_ptr : Pentry);
              PROCEDURE delete (VAR entry_ptr : Pentry);
              FUNCTION count : INTEGER;
              FUNCTION is_member (entry_ptr : Pentry)
                                              : BOOLEAN;
              PROCEDURE for_all (proc : Tfor_all_proc);
              PROCEDURE show; VIRTUAL;
          END;


THashfunktion = FUNCTION (entry_ptr : Pentry) : INTEGER;

PHashtab = ^THashtab;
THashtab = OBJECT (TPuffer)
             PRIVATE
               hash : THashfunktion;
             PUBLIC
               CONSTRUCTOR init (m      : INTEGER;
                                 fkt    : THashfunktion;
                                 VAR OK : BOOLEAN);
               PROCEDURE insert (entry_ptr : Pentry);
               PROCEDURE delete (entry_ptr : Pentry);
               FUNCTION count : INTEGER;
               FUNCTION is_member (entry_ptr : Pentry)
                                              : BOOLEAN;
               PROCEDURE for_all (proc : Tfor_all_proc);
               FUNCTION choose (entry_ptr: Pentry;
                                flag     : TList_flag)
                                              : Pentry;
               PROCEDURE show; VIRTUAL;
               DESTRUCTOR done;
             END;

{ ********************************************************** }
```

```
IMPLEMENTATION

TYPE Zeilen_ARRAY  = ARRAY[1..max_dimension] OF Pointer;
     Spalten_ARRAY = ARRAY[1..max_dimension] OF INTEGER;

     TPuffer_array = ARRAY [0..(max_dimension-1)] OF Pointer;


{ ------------------------------------------------------------ }

CONSTRUCTOR TVektor.init (n      : INTEGER;
                          VAR OK : BOOLEAN);

VAR OldHeapFunc : Pointer;

BEGIN { TVektor.init }
  OK := TRUE;
  IF (n >= 1) AND (n <= max_dimension)
  THEN BEGIN
         IF SizeOf (INTEGER) > SizeOf (Pointer)
         THEN size := SizeOf (INTEGER)
         ELSE size := SizeOf (Pointer);
         vektor    := NIL;
         dimension := n;
         GetMem (vektor, size * dimension);
         IF vektor = NIL
         THEN OK := FALSE;
       END
  ELSE OK := FALSE;
END   { TVektor.init };


FUNCTION TVektor.v (i : INTEGER) : INTEGER;

BEGIN { TVektor.v }
  v := Spalten_ARRAY(vektor^)[i];
END   { TVektor.v };


PROCEDURE TVektor.setze (i    : INTEGER;
                         wert : INTEGER);

BEGIN { TVektor.setze }
  IF (i >= 1) AND (i <= dimension)
  THEN Spalten_ARRAY(vektor^)[i] := wert;
END   { TVektor.setze };


DESTRUCTOR TVektor.done;

BEGIN { TVektor.done }
  IF vektor <> NIL
  THEN FreeMem (vektor, size * dimension);
END   { TVektor.done };

{ ------------------------------------------------------------ }
```

```
FUNCTION TPointerVektor.v (i : INTEGER) : Pointer;

BEGIN { TPointerVektor.v }
  v := Zeilen_ARRAY(vektor^)[i];
END   { TPointerVektor.v };


PROCEDURE TPointerVektor.setze (i    : INTEGER;
                                wert : Pointer);

BEGIN { TPointerVektor.setze }
  IF (i >= 1) AND (i <= dimension)
  THEN Zeilen_ARRAY(vektor^)[i] := wert;
END   { TPointerVektor.setze };

{ ------------------------------------------------------------ }

CONSTRUCTOR TMatrix.init (n : INTEGER;
                          VAR OK : BOOLEAN);

VAR idx : INTEGER;
    jdx : INTEGER;

BEGIN { TMatrix.init }
  OK := TRUE;
  IF n <= max_dimension
  THEN BEGIN
         zeilen := NIL;
         dimension := n;
         GetMem (zeilen, SizeOf(Pointer) * dimension);
         IF zeilen <> NIL
         THEN BEGIN
                FOR idx := 1 TO n DO
                  BEGIN
                    New(PVektor(Zeilen_ARRAY(zeilen^)[idx]),
                        init (dimension, OK));
                    IF NOT OK
                    THEN BEGIN
                           FOR jdx := 1 TO idx - 1 DO
                             Dispose
                              (PVektor(Zeilen_ARRAY(zeilen^)[jdx]),
                               done);
                           FreeMem
                              (zeilen, SizeOf(Pointer) * dimension);
                           zeilen := NIL;
                           Break;
                         END;
                  END;
              END
         ELSE OK := FALSE;
       END
  ELSE OK := FALSE;
END   { TMatrix.init };


FUNCTION TMatrix.m (i : INTEGER;
```

```
                    j : INTEGER) : INTEGER;

BEGIN { TMatrix.m }
  m := PVektor(Zeilen_ARRAY(zeilen^)[i])^.v(j);
END   { TMatrix.m };


PROCEDURE TMatrix.setze (i    : INTEGER;
                         j    : INTEGER;
                         wert : INTEGER);

BEGIN { TMatrix.setze }
  IF (i >= 1) AND (i <= dimension)
  THEN PVektor(Zeilen_ARRAY(zeilen^)[i])^.setze (j, wert);
END   { TMatrix.setze };


DESTRUCTOR TMatrix.done;

VAR idx : INTEGER;

BEGIN { TMatrix.done }
  IF zeilen <> NIL
  THEN BEGIN
         FOR idx := 1 TO dimension DO
           Dispose (PVektor(Zeilen_ARRAY(zeilen^)[idx]), done);
       END;
END   { TMatrix.done };

{ ------------------------------------------------------------ }

FUNCTION TPointerMatrix.m (i : INTEGER;
                           j : INTEGER) : Pointer;

BEGIN { TPointerMatrix.m }
  m := PPointerVektor(Zeilen_ARRAY(zeilen^)[i])^.v(j);
END   { TPointerMatrix.m };


PROCEDURE TPointerMatrix.setze (i    : INTEGER;
                                j    : INTEGER;
                                wert : Pointer);

BEGIN { TPointerMatrix.setze }
  IF (i >= 1) AND (i <= dimension)
  THEN PPointerVektor(Zeilen_ARRAY(zeilen^)[i])^.setze (j, wert);
END   { TPointerMatrix.setze };

{ ------------------------------------------------------------ }


CONSTRUCTOR TPuffer.init (m : INTEGER; VAR OK : BOOLEAN);

BEGIN { Tpuffer.init }
  INHERITED init (m, OK);

  IF OK
```

```
  THEN BEGIN
         start_idx  := 0;
         end_idx    := 0;
         empty_flag := TRUE;
       END;

END   { TPuffer.init };


PROCEDURE TPuffer.insert (entry_ptr : Pentry);

BEGIN { TPuffer.insert }
  end_idx := (end_idx + 1) MOD dimension;
  Tpuffer_array(vektor^)[end_idx] := entry_ptr;
  IF empty_flag
  THEN empty_flag := FALSE
  ELSE IF end_idx = start_idx
       THEN start_idx := (start_idx + 1) MOD dimension;
END   { TPuffer.insert };


PROCEDURE TPuffer.delete (VAR entry_ptr : Pentry);

BEGIN { TPuffer.delete }
  IF NOT empty_flag
  THEN BEGIN
         entry_ptr := Tpuffer_array(vektor^)[start_idx];
         IF start_idx = end_idx THEN empty_flag := TRUE;
         start_idx := (start_idx + 1) MOD dimension;
       END;
END   { TPuffer.delete };


FUNCTION TPuffer.count : INTEGER;

BEGIN { TPuffer.count }
  IF empty_flag
  THEN count := 0
  ELSE BEGIN
         IF start_idx <= end_idx
         THEN count := end_idx - start_idx + 1
         ELSE count := end_idx + 1 + dimension - start_idx;
       END
END   { TPuffer.count };


FUNCTION TPuffer.is_member (entry_ptr : Pentry) : BOOLEAN;

VAR idx : INTEGER;
    OK  : BOOLEAN;

BEGIN { TPuffer.is_member }
  OK := FALSE;
  IF NOT empty_flag
  THEN BEGIN
         IF start_idx <= end_idx
         THEN FOR idx := start_idx TO end_idx DO
```

```
              BEGIN
                IF TPuffer_array(vektor^)[idx] = entry_ptr
                THEN BEGIN
                       OK := TRUE;
                       Break;
                     END;
              END
         ELSE BEGIN
                FOR idx := 0 TO end_idx DO
                 IF TPuffer_array(vektor^)[idx] = entry_ptr
                 THEN BEGIN
                       OK := TRUE;
                       Break;
                     END;
                IF NOT OK
                THEN FOR idx := start_idx TO dimension - 1 DO
                     IF TPuffer_array(vektor^)[idx] = entry_ptr
                     THEN BEGIN
                            OK := TRUE;
                            Break;
                          END;
             END;
       END;
  is_member := OK;
END   { TPuffer.is_member };


PROCEDURE TPuffer.for_all (proc : Tfor_all_proc);

VAR idx : INTEGER;
    OK  : BOOLEAN;

BEGIN { TPuffer.for_all }
  IF NOT empty_flag
  THEN BEGIN
         IF start_idx <= end_idx
         THEN FOR idx := start_idx TO end_idx DO
              proc(Pentry(TPuffer_array(vektor^)[idx]))
         ELSE BEGIN
                FOR idx := 0 TO end_idx DO
                  proc(Pentry(TPuffer_array(vektor^)[idx]));
                FOR idx := start_idx TO dimension - 1 DO
                  proc(Pentry(TPuffer_array(vektor^)[idx]));
             END;
       END;
END   { TPuffer.for_all };


PROCEDURE TPuffer.show;

VAR idx : INTEGER;

BEGIN { TPuffer.show };
  IF NOT empty_flag
  THEN BEGIN
         IF start_idx <= end_idx
         THEN FOR idx := start_idx TO end_idx DO
```

```
                Pentry(TPuffer_array(vektor^)[idx])^.display
          ELSE BEGIN
                 FOR idx := 0 TO end_idx DO
                  Pentry(TPuffer_array(vektor^)[idx])^.display;
                 FOR idx := start_idx TO dimension - 1 DO
                  Pentry(TPuffer_array(vektor^)[idx])^.display;
               END;
        END;
END   { TPuffer.show };

{ ------------------------------------------------------------ }

CONSTRUCTOR THashtab.init (m      : INTEGER;
                           fkt    : THashfunktion;
                           VAR OK : BOOLEAN);

VAR idx      : INTEGER;
    list_ptr : PListe;

BEGIN { THashtab.init }
  INHERITED init(m, OK);
  IF OK
  THEN BEGIN
         FOR idx := 0 TO m-1 DO
           BEGIN
             New (list_ptr, init);
             TPuffer_array(vektor^)[idx] := list_ptr;
           END;
         hash := fkt;
       END;
END   { THashtab.init };


PROCEDURE THashtab.insert (entry_ptr : Pentry);

VAR list_ptr : PListe;

BEGIN { THashtab.insert }
  list_ptr := TPuffer_array(vektor^)[hash(entry_ptr)];
  list_ptr^.insert (entry_ptr);
END   { THashtab.insert };


PROCEDURE THashtab.delete (entry_ptr : Pentry);

VAR list_ptr : PListe;

BEGIN { THashtab.delete }
  list_ptr := TPuffer_array(vektor^)[hash(entry_ptr)];
  list_ptr^.delete (entry_ptr);
END   { THashtab.delete };


FUNCTION THashtab.count : INTEGER;

VAR list_ptr : PListe;
    idx      : INTEGER;
```

```
    i        : INTEGER;

BEGIN { THashtab.count }
  i := 0;
  FOR idx := 0 TO dimension - 1 DO
    BEGIN
      list_ptr := TPuffer_array(vektor^)[idx];
      i := i + list_ptr^.count;
    END;
  count := i;
END   { THashtab.count };


FUNCTION THashtab.is_member (entry_ptr : Pentry) : BOOLEAN;

VAR list_ptr : PListe;
    idx      : INTEGER;

BEGIN { THashtab.is_member }
  is_member := FALSE;
  FOR idx := 0 TO dimension - 1 DO
    BEGIN
      list_ptr  := TPuffer_array(vektor^)[idx];
      is_member := list_ptr^.is_member(entry_ptr);
    END;
END   { THashtab.is_member };


PROCEDURE THashtab.for_all (proc : Tfor_all_proc);

VAR list_ptr : PListe;
    idx      : INTEGER;

BEGIN { THashtab.for_all }
  FOR idx := 0 TO dimension - 1 DO
    BEGIN
      list_ptr  := TPuffer_array(vektor^)[idx];
      list_ptr^.for_all(proc);
    END;
END   { THashtab.for_all };


FUNCTION THashtab.choose (entry_ptr: Pentry;
                          flag     : TList_flag) : Pentry;

VAR lst : PListe;

BEGIN { THashtab.choose }
  lst    := TPuffer_array(vektor^)[hash(entry_ptr)];
  choose := lst^.choose (flag);
END   { THashtab.choose };


PROCEDURE THashtab.show;

VAR list_ptr : PListe;
    idx      : INTEGER;
```

```
BEGIN { THashtab.show }
  FOR idx := 0 TO dimension - 1 DO
    BEGIN
      list_ptr  := TPuffer_array(vektor^)[idx];
      list_ptr^.show;
    END;
END   { THashtab.show };


DESTRUCTOR THashtab.done;

VAR idx      : INTEGER;
    list_ptr : PListe;

BEGIN { THashtab.done }
  IF vektor <> NIL
  THEN BEGIN
         FOR idx := 0 TO dimension - 1 DO
           BEGIN
             list_ptr := TPuffer_array(vektor^)[idx];
             list_ptr^.done;
           END;
       END;
  INHERITED done;
END   { THashtab.done };


{ ********************************************************** }


END.
```

A. Dynamisch festgelegter Vektor bzw. festgelegte Matrix

Die meisten Programmiersprachen erlauben es nicht, Vektoren und Matrizen zu deklarieren, deren Dimensionen erst während der Laufzeit festgelegt werden (dynamisch festgelegte `ARRAY`s). Es werden daher Objekttypen bereitgestellt, die diesem Manko in eingeschränkter Weise Abhilfe verschaffen. Die `UNIT ARRAY_basierte_Liste` enthält dazu zunächst Objekttypen, deren Methoden **Matrixoperationen für ein- bzw. zweidimensionale `INTEGER`- und Pointer-Matrizen** bereitstellen, **deren Dimension erst während der Laufzeit festgelegt wird**. Es besteht jedoch die Restriktion, dass die aktuelle Dimension einer derartigen Matrix eine feste Maximalgröße nicht überschreitet. Der Interfaceteil stellt diese Maximalgröße in einer Konstanten `max_dimension` zur Verfügung. Die Objekttypen heißen `TVektor`, `TPointerVektor`, `TMatrix` bzw. `TPointerMatrix`.

Die Benutzerschnittstelle des Objekttyps `TVektor` lautet:

Methode	Bedeutung
`CONSTRUCTOR TVektor.init (n : INTEGER; VAR OK : BOOLEAN);`	Der Vektor wird mit der angegebenen Anzahl an `INTEGER`-Elementen initialisiert.
Bedeutung der Parameter:	
`n`	Dimension des Vektors, $1 \leq n \leq$ `max_dimension`
`OK`	= `TRUE` steht für die erfolgreiche Initialisierung
`FUNCTION TVektor.v (i : INTEGER) : INTEGER;`	Die Funktion liefert die *i*-te Komponente des Vektors.
Bedeutung des Parameters:	
`i`	Nummer der Komponente
`PROCEDURE TVektor.setze (i : INTEGER; wert : INTEGER);`	Die Funktion setzt die *i*-te Komponente des Vektors auf den angegebenen Wert.
Bedeutung der Parameter:	
`i`	Nummer der Komponente
`wert`	neuer Werter der Komponente
`DESTRUCTOR TVektor.done;`	Der Vektor wird aus dem System entfernt.

Zur Einrichtung eines Vektors mit *n* `INTEGER`-Komponenten mit $1 \leq n \leq$ `max_dimension` wird im Konstruktor des Vektors dynamisch der erforderliche Speicherplatz angefordert und später im Destruktor wieder freigegeben. Hierbei wird gleich soviel Speicherplatz vereinbart, dass er auch zur Aufnahme eines Vektors aus *n* Pointern ausreicht. Die Größe (in Bytes) eines Eintrags im Vektor wird in der Komponente `size` festgehalten. Der Zugriff auf eine Komponente des Vektors erfolgt über den Verweis, der bei Einrichtung in der Komponente `vektor` im PRIVATE-Teil des Objekttyps `TVektor` abgelegt wurde, durch Typisierung mit dem Datentyp

```
TYPE Spalten_ARRAY = ARRAY[1..max_dimension] OF INTEGER;
```

Die ***i*-te Komponente des Vektors** ist `Spalten_ARRAY(vektor^)[i]`.

Zur Deklaration eines Vektors aus *n* Pointern wird der Objekttyp `TPointerVektor` verwendet. Er unterscheidet sich nicht in seiner Benutzerschnittstelle vom Objekttyp `TVektor`. Lediglich die Implementierungen der Zugriffsmethoden berücksichtigen `INTEGER`- bzw. Pointer-Einträge.

Die Benutzerschnittstelle des Objekttyps `TMatrix` lautet:

Methode	Bedeutung
`CONSTRUCTOR TMatrix.init (n : INTEGER; VAR OK : BOOLEAN);`	Die Matrix wird mit der angegebenen Anzahl an `INTEGER`-Elementen initialisiert.
Bedeutung der Parameter:	
`n`	Dimension der Matrix, $1 \leq n \leq$ `max_dimension`
`OK`	= `TRUE` steht für die erfolgreiche Initialisierung
`FUNCTION TMatrix.m (i : INTEGER; j : INTEGER) : INTEGER;`	Die Funktion liefert die Komponente in der *i*-ten Zeile und *j*-ten Spalte der Matrix.
Bedeutung der Parameter:	
`i`	Zeilennummer
`j`	Spaltennummer
`PROCEDURE TMatrix.setze (i : INTEGER; j : INTEGER wert : INTEGER);`	Die Funktion setzt die Komponente in der *i*-ten Zeile und *j*-ten Spalte der Matrix auf den angegebenen Wert.
Bedeutung der Parameter:	
`i`	Zeilennummer
`j`	Spaltennummer
`wert`	neuer Werter der Komponente
`DESTRUCTOR TMatrix.done;`	Die Matrix wird aus dem System entfernt.

Auf ähnliche Weise wie ein Vektor wird eine während der Laufzeit dynamisch vereinbarte (*n*, *n*)-Matrix implementiert (`TMatrix` für eine `INTEGER`-Matrix, `TPointerMatrix` für eine Matrix mit Pointer-Einträgen). Zur Adressierung der *n* Zeilen der Matrix wird Speicherplatz angefordert, der ein `ARRAY` aus *n* Pointern aufnehmen kann; er wird über die Komponente `zeilen` im PRIVATE-Teil des Objekttyps `TMatrix` adressiert. Der *i*-te Verweis dieses `ARRAY`s zeigt auf die *i*-te Zeile der Matrix, die ihrerseits als dynamisch eingerichtetet Vektor vom Objekttyp `TVektor` behandelt wird.

Die ***i*-te Zeile der Matrix** ist somit der Vektor

`PVektor(Zeilen_ARRAY(zeilen^)[i])` bzw.
`PPointerVektor(Zeilen_ARRAY(zeilen^)[i]).`

Das Lesen bzw. Setzen des Matrixelements in der *i*-ten Zeile und *j*-ten Spalte erfolgt über die Methoden des Objekttyps `TVektor`:

```
PVektor(Zeilen_ARRAY(zeilen^)[i])^.v(j),
```

```
PVektor(Zeilen_ARRAY(zeilen^)[i])^.setze (j, wert)
```
bzw.
```
PPointerVektor(Zeilen_ARRAY(zeilen^)[i])^.v(j),
PPointerVektor(Zeilen_ARRAY(zeilen^)[i])^.setze (j, wert).
```

B. Beschränkter Puffer

Ein beschränkter Puffer ist eine Liste, die bei der Initialisierung mit m leeren Plätzen eingerichtet und dann m Einträge aufnehmen kann. Wird versucht, einen weiteren Eintrag aufzunehmen, wird ein bereits im Puffer vorhandener Eintrag überschrieben. ***Es liegt also in der Verantwortung des Anwenders dafür zu sorgen, dass der Puffer nicht „überläuft“.***

In der oben angegebenen Implementierung des Objekttyps `TPuffer` in der `UNIT ARRAY_basierte_Liste` wird ein dynamisch angelegtes `ARRAY` verwendet. Wegen des speziellen Umgangs mit den Indizes (z.B. Numerierung von 0 bis $m-1$ bei m Einträgen) wird hier nicht ein dynamisch vereinbarter Vektor vom Objekttyp `TPointerVektor` genommen.

Der Maximalwert für die Anzahl der verwalteten Puffereinträge wird als Konstante im Interfaceteil der Unit definiert:

```
CONST max_dimension = ...;
```

Bei der Initialisierung wird ein `ARRAY` angelegt, das m Pointer aufnehmen kann (indiziert von 0 bis $m-1$). Zur Realisierung dieses Ansatzes wird der Objekttyp `Tpuffer` als direkter Nachfahre des Objekttyps `TPointerVektor` implementiert. Die Komponente `vektor` eines Objekts vom Typ `TPuffer` adressiert dieses `ARRAY` (aus `TPointervektor`). Es enthält Pointer auf diejenigen Einträge, die sich gerade im Puffer befinden, genauer: in der Pufferverwaltung. Der gegenwärtige Wert der Komponente `start_idx` indiziert die Position im `ARRAY`, die den Pointer auf den Eintrag enthält, der als nächstes aus der Pufferverwaltung entfernt werden soll. Entsprechend indiziert die Komponente `end_idx` die Position im `ARRAY`, die den Pointer auf den zuletzt aufgenommenen Eintrag. Abbildung 5.1.2-1 zeigt typische Belegungssituationen des Puffers.

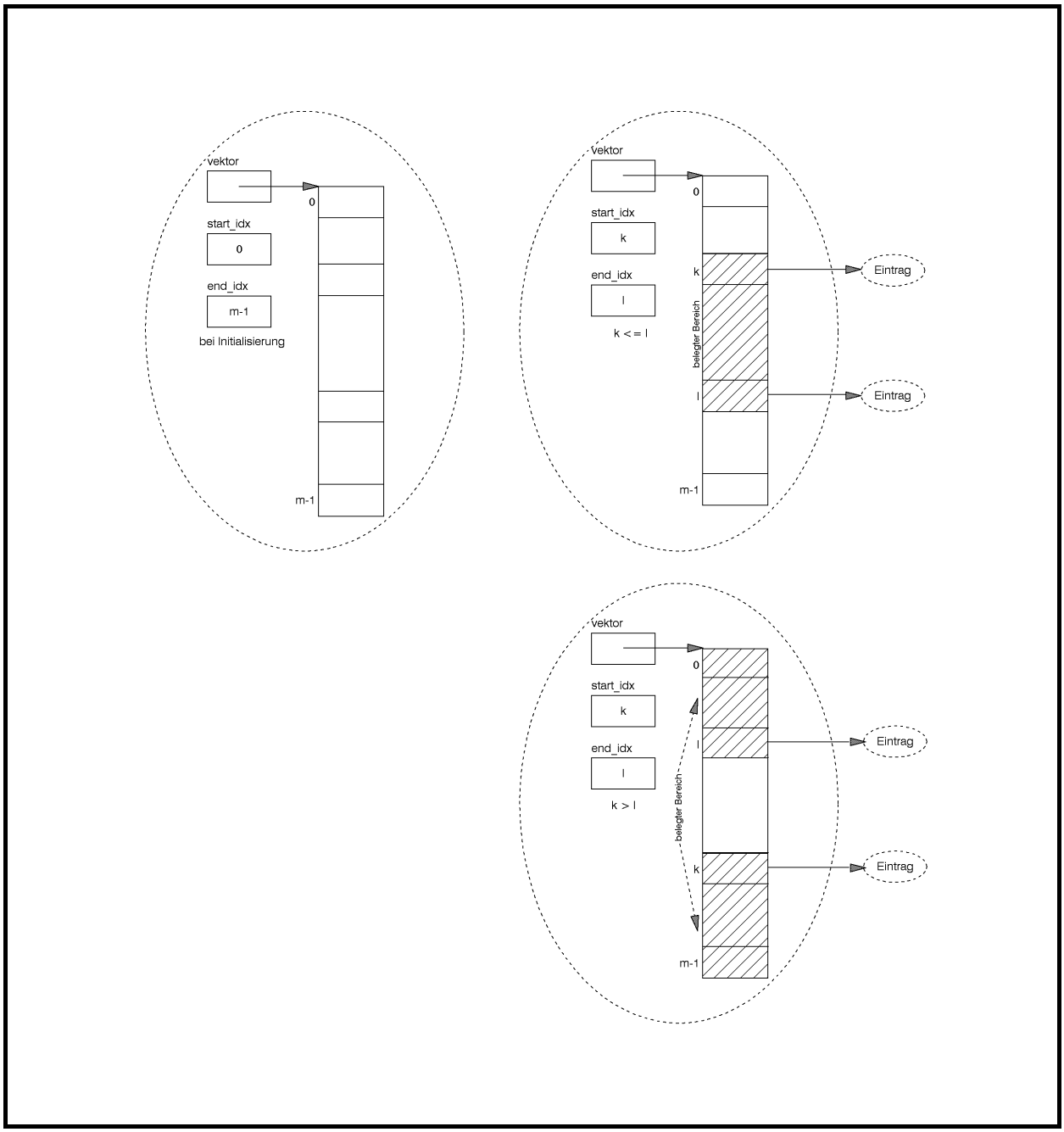


Abbildung 5.1.2-1: Beschränkter Puffer

Die folgende Tabelle präzisiert die Methodenbeschreibung des Objekttyps `TPuffer`.

Methode	Bedeutung
`CONSTRUCTOR TPuffer.init;`	Der beschränkte Puffer wird als leer initialisiert.
`PROCEDURE TPuffer.insert (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	Ein neues Element wird in den beschränkten Puffer aufgenommen.
`PROCEDURE TPuffer.delete (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu entfernende Element	Das Element, das sich am längsten im beschränkten Puffer aufhält, wird an den Aufrufer zurückgeliefert und aus dem beschränkten Puffer entfernt.
`FUNCTION Tpuffer.count: INTEGER;`	Die Funktion liefert die Anzahl der Einträge, die gegenwärtig im beschränkten Puffer stehen.
`FUNCTION TPuffer.is_member (entry_ptr : Pentry) : BOOLEAN;` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu überprüfende Element	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element im beschränkten Puffer vorkommt, ansonsten den Wert `FALSE`.
`PROCEDURE TPuffer.for_all (proc : Tfor_all_proc);` Bedeutung des Parameters: `proc` anzuwendende Elementmethode	Auf alle zur Zeit im beschränkten Puffer eingetragenen Elemente wird die Operation (Elementmethode) `proc` angewendet
`PROCEDURE TPuffer.show;`	Alle zur Zeit im beschränkten Puffer vorkommenden Elemente werden angezeigt.

Im Konstruktor `TPuffer.init` wird mit der von `TPointerVektor` geerbte Konstruktor aufgerufen, der genau soviel Speicherplatz auf dem Heap anfordert, wie zur Aufnahme von m Pointern erforderlich ist. Um einen Eintrag im `ARRAY` über einen Feldindex anzusprechen, wird der bereitgestellte Speicherplatz mit einem `ARRAY`-Datentyp überlagert (typisiert). Zu beachten ist hierbei, dass in den Methoden des Objekttyps `TPuffer` das `ARRAY` von 0 bis $m-1$ indiziert wird.

Da der Puffer maximal m Einträge enthält, lässt sich hier der Aufwand, den ein jeweiliger Methodenaufruf benötigt, durch eine Konstante abschätzen. Zu beachten ist jedoch, dass ohne weitere Kontrollen durch den Benutzer eventuell im Puffer stehende Einträge überschrieben werden.

C. Hashtabelle

Eine Hashtabelle dient der Verwaltung von Elementen vom Objekttyp `Tentry` wie mit einer Liste. Bei einer Hashtabelle erweist sich das ***mittlere Laufzeitverhalten*** bei der ***delete***-, ***is_member***- und $\forall$**op** -Operation jedoch als wesentlich besser gegenüber der Implementation des Objekttyps `TListe`. Eine Hashtabelle stellt eine Kombination aus einem beschränkten Puffer und einer Liste dar.

Die obige Unit zeigt eine Implementierung der Datenstruktur Hashtabelle durch den Objekttyp `THashtab`, die sich vom Objekttyp `TPuffer` ableitet. Ein Objekt vom Typ `THashtab` wird daher als festes `ARRAY` mit *m* Einträgen (numeriert von 0 bis $m-1$) implementiert, das über den Pointer `vektor` adressiert ist. Ein Eintrag in diesem `ARRAY` ist nun ein Pointer auf ein Objekt vom Typ `TListe`.

Es gibt eine Abbildung *h*, die jedem Objekt vom Typ `Tentry` einen Wert aus $[0..m-1]$ zuweist; diese Abbildung heißt **Hashfunktion**. Die Deklaration des Types der Hashfunktion ist

```
TYPE THashfunktion = FUNCTION (entry_ptr : Pentry) : INTEGER;
```

Der Anwender übergibt einem Objekt vom Typ `THashtab` die zu verwendende Hashfunktion bei dessen Einrichtung durch den Konstruktor.

Der *i*-te Eintrag `TPuffer_array(vektor^)[`i`]` in dem `ARRAY` adressiert diejenige Liste, in die alle Objekte vom Typ `Tentry` aufgenommen werden, für die die Hashfunktion den Wert *i* liefert.

Das Verfahren wird in Abbildung 5.1.2-2 erläutert. Die zu verwaltenden Elemente vom Typ `Tentry` werden hier durch natürliche Zahlen repräsentiert. Die Hashtabelle enthält $m = 7$ Einträge. Als Hashfunktion wird die durch

$$h(x) = (x \bmod m)$$

definierte Funktion genommen. Der linke Teil der Abbildung zeigt ein Objekt vom Typ `THashtab`, nachdem nacheinander die durch 3, 13, 7, 1, 18, 25, 29 und 9 repräsentierten Elemente vom Typ `Tentry` aufgenommen wurden. Der rechte Teil zeigt die Hashtabelle nach Entfernen der durch 29, 3 und 18 repräsentierten Elemente.

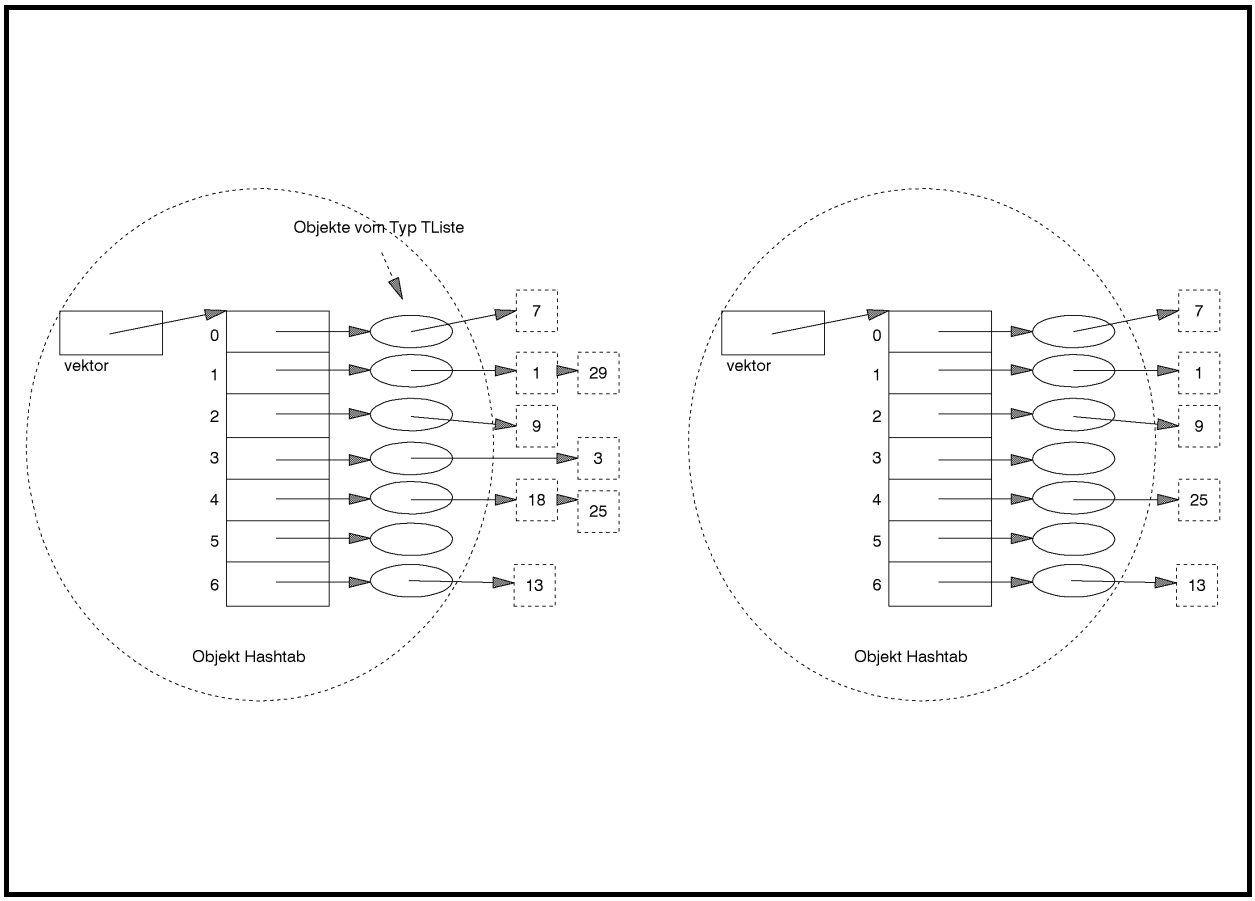


Abbildung 5.1.2-2: Hashtabelle

Das dargestellte Verfahren wird als **offenes Hashing** bezeichnet: ein Element kann immer aufgenommen werden, und zwar in die durch die Hashfunktion adressierte Liste, die ihrerseits beliebig erweiterbar ist. Im Gegensatz dazu wird beim **geschlossenen Hashing** der gesamte zur Verfügung stehende Speicherplatz bzw. die Anzahl der Einträge begrenzt: m Plätze sind dabei für das durch `vektor` adressierte Datenobjekt vorgesehen. Hier wird ein Element an der durch die Hashfunktion bestimmten Position eingetragen, wenn der Platz noch nicht belegt wird. Andernfalls wird das Element an einem freien Platz in einem „Überlaufbereich“ abgelegt, dessen Größe durch $(b-1)\cdot m$ Plätze (mit einer Konstanten b) begrenzt ist. Insgesamt können so in die Hashtabelle maximal $n \le m \cdot b$ viele Elemente aufgenommen werden. Beim geschlossenen Hashing stellt sich das Problem der **Behandlung von Kollisionen**, d.h. der Entwicklung einer geeigneten Strategie, die festlegt, an welcher Position des Überlaufbereichs ein Element abgelegt wird, falls der Platz an der durch die Hashfunktion bestimmten Position innerhalb des durch `vektor` adressierten Datenobjekts bereits belegt ist.

Die folgende Darstellung beschränkt sich auf das offene Hashing.

Die Benutzerschnittstelle für den Objekttyp `THashtab` lautet:

Methode	Bedeutung
`CONSTRUCTOR THashtab.init;`	Die Hashtabelle wird als leer initialisiert.
`PROCEDURE THashtab.insert (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das einzutragende Element	Ein neues Element wird in die Hashtabelle aufgenommen.
`PROCEDURE THashtab.delete (entry_ptr : Pentry);` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu entfernende Element	Das adressierte Element wird an den Aufrufer zurückgeliefert und aus der Hashtabelle entfernt.
`FUNCTION THashtab.count: INTEGER;`	Die Funktion liefert die Anzahl der Einträge, die gegenwärtig in der Hashtabelle stehen.
`FUNCTION THashtab.is_member (entry_ptr : Pentry) : BOOLEAN;` Bedeutung des Parameters: `entry_ptr` Verweis auf das zu überprüfende Element	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element in der Hashtabelle vorkommt, ansonsten den Wert `FALSE`.
`PROCEDURE THashtab.for_all (proc : Tfor_all_proc);` Bedeutung des Parameters: `proc` anzuwendende Elementmethode	Auf alle zur Zeit in der Hashtabelle eingetragenen Elemente wird die Operation (Elementmethode) `proc` angewendet
`FUNCTION THashtab.choose (entry_ptr : Pentry; param : TList_flag) : Pentry;`	Bei jedem Aufruf wird ein Eintrag mit dem Hashwert von `entry_ptr` geliefert, wobei `param` angibt, ob der erste Eintrag ermittelt werden soll (`c_first`), oder der nächste Eintrag seit dem letzten Aufruf von `choose` (`c_next`).
`PROCEDURE THashtab.show;`	Alle zur Zeit in der Hashtabelle vorkommenden Elemente werden angezeigt.
`DESTRUCTOR THashtab.done;`	Die Hashtabelle wird aus dem System entfernt.

Die Hashfunktion bestimmt, in welche (Teil-) Liste der Hashtabelle ein neu aufzunehmendes Element eingetragen wird. Bei ungünstiger Wahl der Hashfunktion werden viele Elemente in nur wenige Listen aufgenommen. In diesem Fall ist das Laufzeitverhalten einer Hashtabelle nicht besser als dasjenige der Datenstruktur `TListe`. Insbesondere werden im ungünstigsten Fall bei Vorhandensein von bereits n Elementen $O(n)$ viele Teilschritte für die einmalige Ausführung eines Aufrufs `THashtab.delete` ausgeführt.

Die Hashfunktion ist besonders gut, wenn die eingetragenen Elemente möglichst gleichmäßig über alle Listen verteilt werden. Es soll daher das ***mittlere Laufzeitverhalten*** von *n* ***insert***-, ***delete***- und ***is_member***-Operationen auf einer anfangs leeren Hashtabelle betrachtet werden. Dabei werden an die Hashfunktion folgende Bedingungen gestellt:

1. Die Berechnung eines Werts der Hashfunktion benötigt konstanten Aufwand
2. die Hashfunktion verteilt alle einzufügenden Elemente gleichmäßig über das Intervall $[0 .. m-1]$, d.h. für alle *i* und *j* aus dem Intervall $[0 .. m-1]$ gilt:
   $$\left\| \left| h^{-1}(i) \right| - \left| h^{-1}(j) \right| \right\| \leq 1$$
3. sämtliche mögliche einzufügenden, zu entfernenden oder zu suchenden Elemente sind mit gleicher Wahrscheinlichkeit Argument der nächsten Operation, d.h. gibt es *u* viele mögliche Elemente, so ist ein beliebiges Element mit Wahrscheinlichkeit $1/u$ Argument der *k*-ten Operation.

Bezeichnet x_k das Argument der *k*-ten Operation (*k* = 1, ..., *n*), dann gilt unter diesen Voraussetzungen: $P(h(x_k) = i) = P(h(x_k) = j) = 1/m$ (mit *i* = 0, ..., *m* – 1 und *j* = 0, ..., *m* – 1), d.h. alle Listen werden in der *k*-ten Operation mit gleicher Wahrscheinlichkeit $1/m$ ausgewählt. Unter diesen Voraussetzungen lässt sich zeigen ([BLU]): Der Erwartungswert für die von der Folge von *n* Einfüge- und Zugriffsoperationen benötigte Zeit ist $\leq (1 + \beta/2) \cdot n$ mit $\beta = n/m$.

Die Bedingungen 1 und 2 lassen sich durch Hashfunktionen der Form $h(x) = (x \text{ MOD } m)$ realisieren, wobei hierbei *m* sorgfältig gewählt werden muss (siehe [O/W]). Bedingung 3 ist in der Praxis i.a. nicht leicht zu erreichen, da sie ein bestimmtes Verhalten des Benutzers der Hashfunktion postuliert.

Wählt man *m* zu Beginn hinreichend groß, so lässt sich die Größe $\beta = n/m$ in obiger Abschätzung durch eine Konstante beschränken, so dass der obige Erwartungswert von der Ordnung $O(n)$ ist. In der Implementierung `TListe` benötigen *n* Aufrufe von `TListe.delete` (nachdem *n* Elemente eingefügt wurden) im ungünstigsten Fall $O(n^2)$ viele Verarbeitungsschritte.

Abschließend sei bemerkt, dass in der Literatur eine Reihe weiterer Hashverfahren beschrieben wird (vgl. z.B. [O/W] und [BLU]).

5.1.3 Mengen und Teilmengensysteme einer Grundmenge

Pascal und andere Sprachen haben einen im Sprachkonzept vorgesehenen Mengentyp (Kapitel 2.2.2). Eine Deklaration der Form

```
TYPE TMenge = SET OF typ;
```

erwartet, dass `typ` einen Ordinaltyp bezeichnet. Die üblichen Mengenoperationen lassen sich mit den definierten Operatoren realisieren. Sind beispielsweise die Variablen

```
VAR amenge : TMenge;
    bmenge : TMenge;
    cmenge : TMenge;
```

deklariert, so erfolgt die Initialisierung der Menge `amenge` als leere Menge durch

```
amenge := [];
```

Die Mengenoperationen $\cup$, $\cap$ und $\setminus$ lauten

```
cmenge := amenge + bmenge; { Vereinigung }
cmenge := amenge * bmenge; { Schnitt     }
cmenge := amenge - bmenge; { Differenz   }
```

Ist `x` eine Variable vom Typ der Elemente in `amenge`, etwa

```
VAR x : typ;
```

so realisiert

```
x IN amenge
```

die elementweise Enthaltenseinrelation $\in$; die Teilmengenrelation $\subseteq$ ist durch

```
bmenge <= cmenge
```

erklärt.

In einer Pascal-Menge können nur Elemente vom Ordinaltyp liegen. Das bedeutet, dass Mengen, die aus Objekten eines Objekttyps bestehen, nicht definierbar sind. Viele Anwendungen erfordern aber Definitionsmöglichkeiten, in denen **Mengen aus Objekten** eines (allgemei-

nen) Objekttyps bestehen. Es ist daher ein Objekttyp `TMenge` erforderlich, der die in Abbildung 5-2 beschriebenen Mengen mit entsprechenden Mengenoperationen zulässt. Eine Möglichkeit besteht darin, Mengen in Form von Listen wie in Kapitel 5.1.1 zu implementieren, d.h. als lineare Datenstruktur.

Einige Anwendungen erfordern die Implementation ***disjunkter Teilmengen einer Grundmenge mit speziellen Mengenoperationen***: Bei der Initialisierung wird eine Teilmenge als leere Menge oder als Menge erzeugt, die genau ein Element enthält. Eine Grundoperation vereinigt zwei disjunkte Teilmengen zu einer weiteren Menge. Typische Anwendungen führen auf eine als **Union-Find-Struktur** bezeichnete Datenstruktur. Dazu gehören Algorithmen, die Mengenpartitionen gemäß einer Äquivalenzrelation manipulieren, oder Algorithmen auf Graphen, z.B. bei der Suche von minimalen aufspannenden Bäumen in Kommunikationsnetzen (siehe [O/W]).

Eine Union-Find-Struktur besteht also aus einem Objekt, das disjunkte Teilmengen einer Grundmenge enthält; jede dieser Teilmengen enthält als Elemente Objekte vom Typ `Tentry` (Abbildung 5.1.3-1). Eine Union-Find-Struktur ist also ein **Mengensystem**. Die spezielle Form der Union-Find-Strukturen soll in diesem Kapitel betrachtet werden.

Die Implementierung erfolgt durch die Implementierung der im Mengensystem enthaltenen Teilmengen als dynamisch erzeugt Objekte. Die Organisation der Teilmengen im Mengensystem kann z.B. als Liste erfolgen; sie soll hier nicht weiter betrachtet werden. Vielmehr geht es um eine für die speziellen Operationen der Union-Find-Struktur geeignete Implementierung der Teilmengen, für die ein Objekttyp `TUnion_Find` deklariert wird.

Aus der Tatsache, dass mit disjunkten Teilmengen einer Grundmenge hantiert wird, erübrigt sich für `TUnion_Find` die Operation $\cap$, da diese immer die leere Menge ergibt. Die ***init***-Operation initialisiert eine Teilmenge als leere Menge oder als einelementige Teilmenge der Grundmenge; es muss ihr also als Parameter das eventuelle „Initialisierungselement“ mitgegeben werden, d.h. sie hat zur Erzeugung der Teilmenge das Aufrufformat ***init*** (a_i). Dabei soll vorausgesetzt werden, dass stets alle Initialisierungselemente neue, bisher noch nicht verwendete Elemente der Grundmenge sind. Die ***init***-Operation vergibt implizit auch eine ***Identifikation*** für die Teilmenge, und zwar wird zunächst als Identifikation der Teilmenge das Element genommen, das bei Initialisierung als einelementige Menge anfangs in ihr enthalten ist (die leere Menge bekommt keine explizite Identifikation, siehe Implementierung der Operationen). Das Element a_i in einer Teilmenge der Form $T = \{a_i\}$ dient also auch als Identifikation von T; gleichzeitig liegt innerhalb T eine Repräsentation des Elements a_i (wieder als Adressverweis auf a_i). Im Laufe der Lebensdauer einer Teilmenge, wenn sie nämlich mit anderen Teilmengen vereinigt wird und dann mehrere Elemente enthält, kann sich ihre Identifikation ändern (siehe unten).

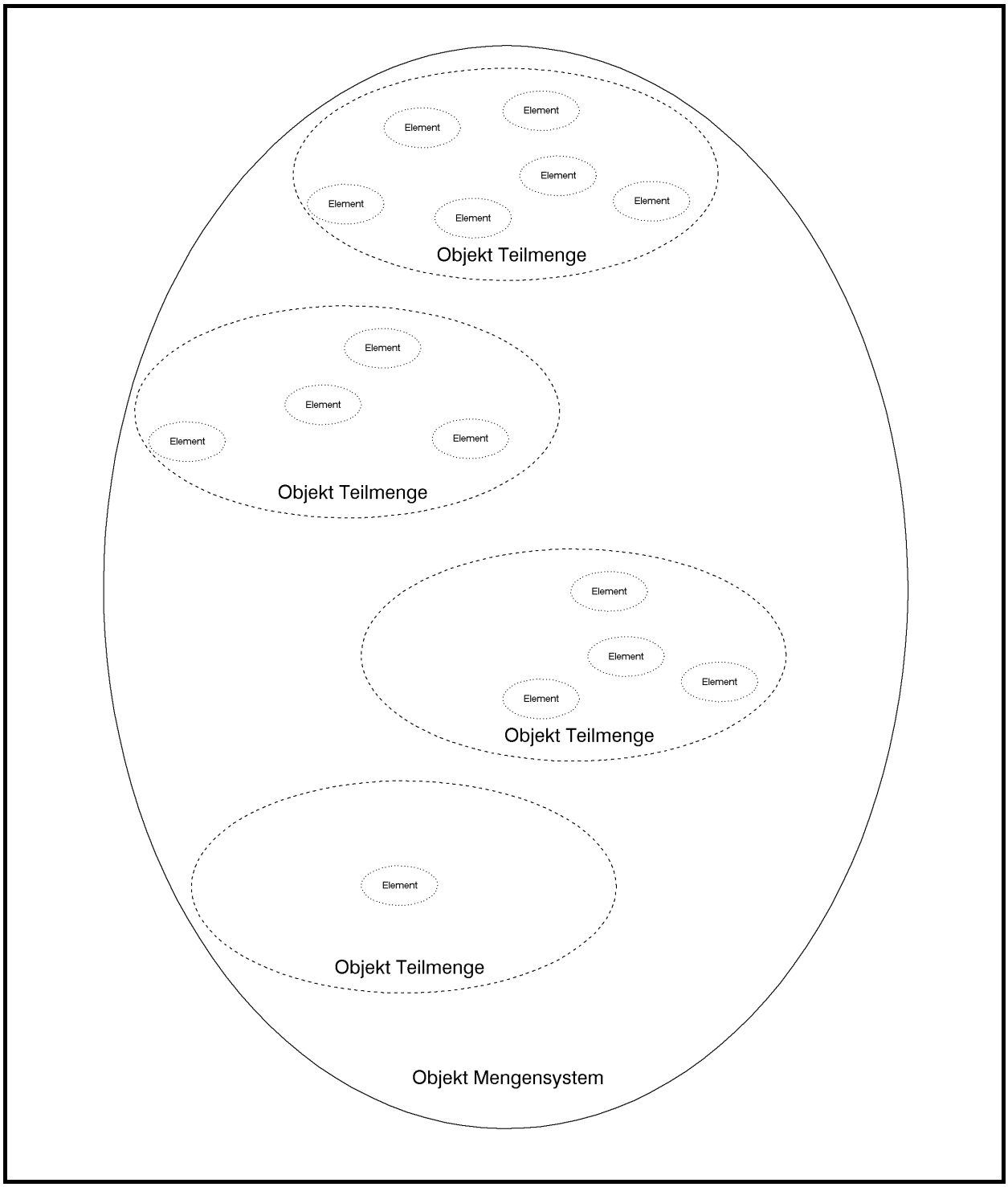


Abbildung 5.1.3-1: Union-Find-Struktur

In den Anwendungen ist es wichtig, für ein Element *x* die gegenwärtige Identifikation derjenigen Teilmenge zu ermitteln, die *x* enthält (diese ist eindeutig, da hier ja nur disjunkte Teilmengen der Grundmenge vorkommen). Dazu dient die Operation ***find*** (*x*). Zwei Elemente *x* und *y* liegen bei ***find*** (*x*) = ***find*** (*y*) in derselben Teilmenge. Für eine effiziente Implementierung der ***find***-Operation wird in die ***init***-Operation ein Rückgabewert eingebaut: ***init*** gibt einen Pointer auf die Repräsentation des Initialisierungselements in *T* zurück, ***der vom Anwender nicht geändert werden darf***. Mit Hilfe dieses Pointers durchsucht ein Aufruf ***find*** (*x*) die

Teilmenge nach deren Identifikation; der Pointer verweist auf die Repräsentation von *x*, da er bei der Initialisierung der Teilmenge { *x* }, die *x* zum ersten Mal enthielt, so erzeugt wurde.

Als weitere Mengenoperation ist die Vereinigung $T \cup S$ zweier Teilmengen *T* und *S* möglich. Durch die Annahmen über die ***init***-Operation werden Vereinigungsoperationen nur mit disjunkten Mengen durchgeführt. Die Operation ***union*** (*T*, *S*) bewirkt entweder $T := T \cup S$ oder $S := T \cup S$, je nachdem, ob *T* bzw. *S* mindestens so viele Elemente wie die andere Menge hat. Gleichzeitig bekommt die Vereinigungsmenge die Identifikation von *T* bzw. von *S*, und die andere Menge wird aus dem Mengensystem entfernt.

Auf eine Operation ***show***, die die Elemente einer Teilmenge anzeigt, soll hier verzichtet werden, da Teilmengen im folgenden als nichtlinerare Datenstrukturen implementiert werden und im Kapitel 5.2 entsprechende Implementierungen von ***show*** gezeigt werden.

Eine Teilmenge $T = \{a_1, ..., a_n\}$ der Grundmenge *M* kann konzeptionell als gerichteter Baum mit durch Elemente markierten Knoten angesehen werden, wobei die Wurzel des Baums mit einem der Elemente a_i und die übrigen Knoten mit a_j mit $j \neq i$ markiert sind. Von jedem Knoten, der mit a_j markiert ist, führt ein Pfad über eine oder mehrere Kante zu dem mit a_i markierten Knoten. Das Element a_i stellt auch die Identifikation von *T* dar. Abbildung 5.1.3-2 zeigt im oberen Teil Beispiele von Teilmengen S_1 = { 1, 7, 8, 9}, S_2 = { 2, 5, 10} und S_3 = { 3, 4, 6} der Grundmenge *M* = { 1, 2, ..., 10} und zwei prinzipielle Möglichkeiten, $S_1 \cup S_2$ darzustellen. In diesem Beispiel sind die Elemente als Objekte natürliche Zahlen. Die oben beschriebene Operation ***union*** (S_i, S_j) zur Vereinigung zweier Teilmengen S_i und S_j bildet die linke Alternative in Abbildung 5.1.3-2 nach: Es wird diejenige Teilmenge S_i an S_j als Unterbaum angehängt, die weniger Elemente enthält.

In Abbildung 5.1.3-2 ist ***find*** (9) auf der Teilmenge S_1 das Element 1 und ***find*** (10) auf der (linken Alternative) von $S_1 \cup S_2$ das Element 1, ebenso wie ***find*** (2) oder ***find*** (8).

Die Annahmen führen auf folgende `UNIT Unionfd` zur Implementierung einer Union-Find-Struktur auf einer Grundmenge aus Objekten vom Objekttyp `Tentry`. Die Teilmengen haben den Objekttyp `TUnion_Find`. Wieder werden in einem Objekt vom Typ `TUnion_Find` die in ihr enthaltenen Objekte nicht selbst, sondern Verweise auf die jeweiligen Objekte festgehalten: Der untere Teil von Abbildung 5.1.3-2 zeigt die Implementierung für eine einelementige Teilmenge und für $S_1 \cup S_2$. Die Erzeugung der Elemente liegt wieder in der Kontrolle der Anwendung. Alle weiteren Details können dem folgenden Code entnommen werden.

Die Implementierung des Destruktors `TUnion_Find.done` ist aus Gründen der Übersichtlichkeit nicht ausgeführt; hierfür empfiehlt es sich nämlich, die Einträge im Baum, die eine Teilmenge ausmachen, zusätzlich vorwärts zu verketten, damit sie von der Wurzel aus erreicht werden können.

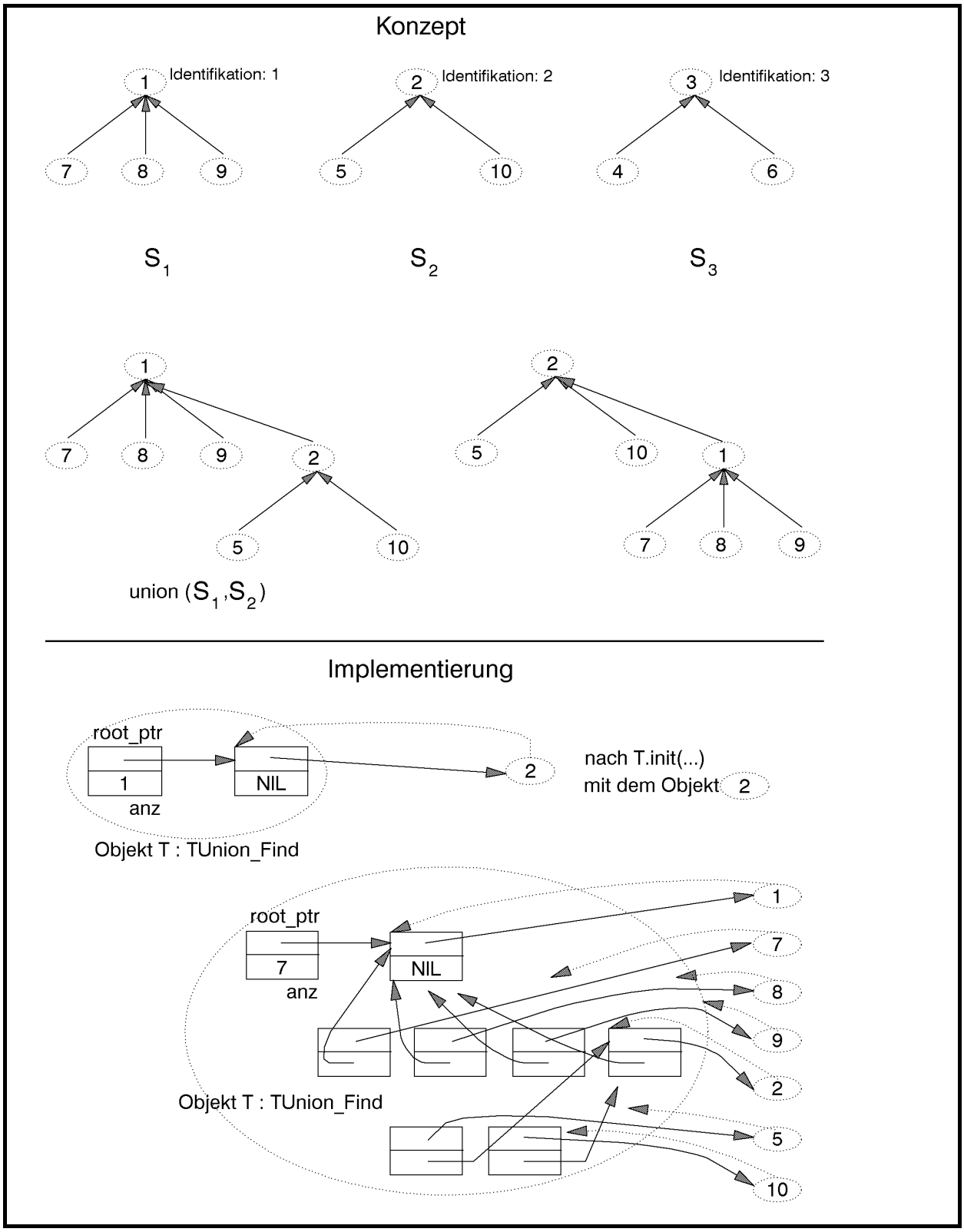


Abbildung 5.1.3-2: Teilmengen einer Grundmenge (Beispiel)

Die folgende Tabelle beschreibt die Benutzerschnittstelle von `UNIT Unionfd`.

Methode	Bedeutung
`CONSTRUCTOR TUnion_Find.init (entry_ptr : Pentry; VAR strukt_ptr : Pointer);`	Bei `entry_ptr ≠ NIL` wird eine Teilmenge mit genau einem Element eingerichtet; bei `entry_ptr = NIL` wird die leere Menge erzeugt.
Bedeutung des Parameters:	
`entry_ptr`	Verweis auf das Element der erzeugten Teilmenge (bei `entry_ptr ≠ NIL`)
`strukt_ptr`	Verweis auf die Repräsentation der erzeugten Teilmenge (gestrichelte Linie in Abbildung 5.1.3-2); *der Wert darf vom Anwender nicht geändert werden*
`FUNCTION TUnion_Find.count: INTEGER;`	Die Funktion liefert die Anzahl der Einträge, die sich gegenwärtig in der Teilmenge befinden.
`DESTRUCTOR TUnion_Find.done;`	Die Teilmenge wird aus dem System entfernt.
`FUNCTION find (element_ptr : Pointer) : Pentry;`	Die Funktion ermittelt einen Verweis auf das Element, das die Teilmenge identifiziert, die das durch `element_ptr` adressierte Element enthält.
Bedeutung des Parameters:	
`element_ptr`	Verweis auf ein Element
`PROCEDURE Union (VAR T: PUnion_Find; VAR S: PUnion_Find);`	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element im beschränkten Puffer vorkommt, ansonsten den Wert `FALSE`.
Bedeutung des Parameters:	
`T` bzw. `S`	Verweise auf die zu vereinigenden Teilmengen

```
UNIT Unionfd;

INTERFACE

  USES Element;

  TYPE PMengensystem = ^TMengensystem;
       TMengensystem = OBJECT
                         {...}
                       END;

       PUnion_Find = ^TUnion_Find;
       TUnion_Find = OBJECT
                     PRIVATE
                       root_ptr : Pointer;
                                  { Verweis auf das Element,
                                    das die Menge identifiziert  }
                       anz  : INTEGER;
                                  { Anzahl der Elemente der Menge}
```

```
                    PUBLIC
                      CONSTRUCTOR init (entry_ptr      : Pentry;
                                        VAR strukt_ptr : Pointer);
                      FUNCTION count : INTEGER;
                      DESTRUCTOR done; VIRTUAL;
                    END;

  FUNCTION find (element_ptr : Pointer) : Pentry;

  PROCEDURE Union (VAR T : PUnion_Find;
                   VAR S : PUnion_Find);
  { Bildet die Vereinigung von T und S in der Teilmenge, die
    mindestens so viele Elemente wie die andere Teilmenge
    enthält, die dann entfernt wird                            }

{ ************************************************************** }

IMPLEMENTATION

TYPE PUnion_Findentry = ^TUnion_Findentry;
     TUnion_Findentry = RECORD
                          entry_ptr : Pentry;
                          pred      : PUnion_Findentry;
                        END;


CONSTRUCTOR TUnion_Find.init (entry_ptr      : Pentry;
                              VAR strukt_ptr : Pointer);

  BEGIN { TUnion_Find.init };
    IF entry_ptr = NIL
    THEN BEGIN
           root_ptr   := NIL;
           anz        := 0;
           strukt_ptr := NIL;
         END
    ELSE BEGIN
           New (PUnion_Findentry(root_ptr));
           PUnion_Findentry(root_ptr)^.entry_ptr := entry_ptr;
           PUnion_Findentry(root_ptr)^.pred      := NIL;
           anz        := 1;
           strukt_ptr := root_ptr;
         END;
  END   { TUnion_Find.init };


FUNCTION TUnion_Find.count : INTEGER;

BEGIN { TUnion_Find.count }
  count := anz;
END   { TUnion_Find.count };


DESTRUCTOR TUnion_Find.done;

  BEGIN { TUnion_Find.done }
    {...}
```

```
  END   { TUnion_Find.done };

{ ---------------------------------------------------------------- }

FUNCTION find (element_ptr : Pointer) : Pentry;

VAR p : PUnion_Findentry;
    q : PUnion_Findentry;

  BEGIN { find }
    p := element_ptr;
    WHILE p^.pred <> NIL DO
      p := p^.pred;

    find := p^.entry_ptr;
  END   { find };


PROCEDURE Union (VAR T : PUnion_Find;
                 VAR S : PUnion_Find);

VAR U : PUnion_Find;
    V : PUnion_Find;

  BEGIN { Union }
    IF T^.anz >= S^.anz THEN BEGIN
                               U := T;
                               V := S;
                             END
                        ELSE BEGIN
                               U := S;
                               V := T;
                             END;
    U^.anz := U^.anz + V^.anz;
    PUnion_Findentry(V^.root_ptr)^.pred := U^.root_ptr;

    Dispose (V, done);
    V := NIL;

  END   { Union };


END.
```

Die folgenden Aufwandsabschätzung einer Folge von ***union***- und ***find***-Operationen legt eine Modifikation der Operation ***find*** nahe.

Die einmalige Durchführung der ***union***-Operation benötigt einen konstanten Aufwand. Beginnt man mit n Union-Find-Strukturen, die mit jeweils einem Element initialisiert wurden, und führt $(n-1)$-mal die ***union***-Operation aus, so haben alle zwischenzeitlich entstandenen Bäume höchstens eine Höhe h mit $h \leq \lfloor \log_2(n) \rfloor + 1$ (siehe [O/W]), so dass eine ***find***-Operation dann höchstens $O(\log(n))$ Schritte benötigt. Eine Verbesserung erhielte man, indem man bei der ***union***-Operation alle Elemente unter den obersten Knoten der größeren

Menge hängt, allerdings auf Kosten des konstanten Aufwands. Eine andere Möglichkeit besteht darin, einen Baum bei der ***find***-Operation zu reorganisieren, da er ja sowieso durchlaufen werden muss. Der erhöhte Aufwand, der dabei bei der ***find***-Operation entsteht, wirkt sich günstig auf die Höhe des Baums aus, so dass spätere ***find***-Operationen weniger aufwendig ablaufen. Daher wird in die Methode zur ***find***-Operation eine **Pfadkompression** eingebaut. Drei Ansätze sind in Abbildung 5.1.3-3 zu sehen. Die Objekte in einer Union-Find-Struktur werden wieder durch Knoten dargestellt, die mit dem Objekt bezeichnet sind. Das Objekt in der Wurzel ist die Identifikation einer Teilmenge. Es werde ***find*** (x) aufgerufen, so dass der „Einstieg" in den Baum beim Objekt x erfolgt und ein Pfad bis zur Wurzel (mit dem Objekt) v durchlaufen wird. Abbildung 5.1.3-3 zeigt das Beispiel einer Ausgangssituation, bei der auf dem Pfad von x nach v die Objekt y, z, w und u liegen. Weitere mögliche Teilbäume unter den Objekten sind durch Dreiecke angedeutet.

Ansätze zur Pfadkompression in Abbildung 5.1.3-3 sind:

- **Kollapsregel**: Es werden alle Knoten auf dem Pfad von x zur Wurzel direkt unter die Wurzel gehängt. Da man die Wurzel erst finden muss, wird der Baum dabei zweimal durchlaufen
- **Pfad-Splitting**: Während des Durchlaufens des Pfads von x nach v wird jeder Knoten (und der darunter hängende Teilbaum) unter seinen übernächsten Nachfolger gehängt
- **Pfad-Halbierung**: Während des Durchlaufens des Pfads von x nach v wird der i-te Knoten bei ungeradem i (und der darunter hängende Teilbaum) unter seinen übernächsten Nachfolger gehängt; die Position innerhalb des Baum des i-te Knotens bei geradem i (und der darunter hängende Teilbaum) bleibt unverändert.

Die Bedeutung der Kollapsregel lässt sich in folgender Aussage zusammenfassen (siehe [O/W]):

Werden n ***init***-Operationen zur Erzeugung einelementiger Teilmengen und anschließend höchstens $n-1$ ***union***-Operationen, gemischt mit einer Reihe von ***find***-Operationen durchgeführt, in der die Kollapsregel eingebaut ist, und ist die Gesamtanzahl der ***union***- und ***find***-Operationen $m \geq n$, so erfordern alle Operationen zusammen einen Gesamtaufwand von $O(m \cdot \alpha(m,n))$ Rechenschritten. Hierbei ist $\alpha(m,n)$ die (extrem langsam wachsende) inverse Funktion zur Ackermann-Funktion. Es ist z.B. $\alpha(m,n) \leq 3$ für $n < 2^{16} = 65.536$ und $\alpha(m,n) \leq 4$ für alle praktisch auftretenden Werte von m und n, so dass $\alpha(m,n)$ als praktisch konstant angesehen werden kann.

Bemerkung: Die Ackermannfunktion $A(i,j)$ ist für $i > 0$ und $j > 0$ definiert durch

$A(1,j) = 2^j$ für $j > 0$,

$A(i,1) = A(i-1,2)$ für $i > 1$,

$A(i,j) = A(i-1, A(i,j-1))$ für $i > 1$ und $j > 1$.

Die inverse Funktion zu $A(i, j)$ lautet für $m \geq n > 0$:

$$\alpha(m, n) = \min\{ i \mid i > 0 \text{ und } A(i, \lfloor m/n \rfloor) > \log_2(n) \}.$$

Die ***find***-Operation mit Kollapsregel lautet:

```
FUNCTION find (element_ptr : Pointer) : Pentry;

VAR p : PUnion_Findentry;
    q : PUnion_Findentry;

  BEGIN { find }
    p := element_ptr;
    WHILE p^.pred <> NIL DO
      p := p^.pred;

    { Kollapsregel }
    q := element_ptr;
    WHILE q <> p DO
      BEGIN
        q^.pred := p;
        q       := q^.pred;
      END;

    find := p^.entry_ptr;
  END   { find };
```

Eine alternative Möglichkeit zur Entscheidung, welche Teilmenge als Ergebnismenge der ***union***-Operation genommen wird, ist nicht die Anzahl der Elemente in den Teilmengen, sondern die Höhen der Bäume, die die Teilmengen darstellen. Dadurch gilt ebenfalls die obige Komplexitätsabschätzung.

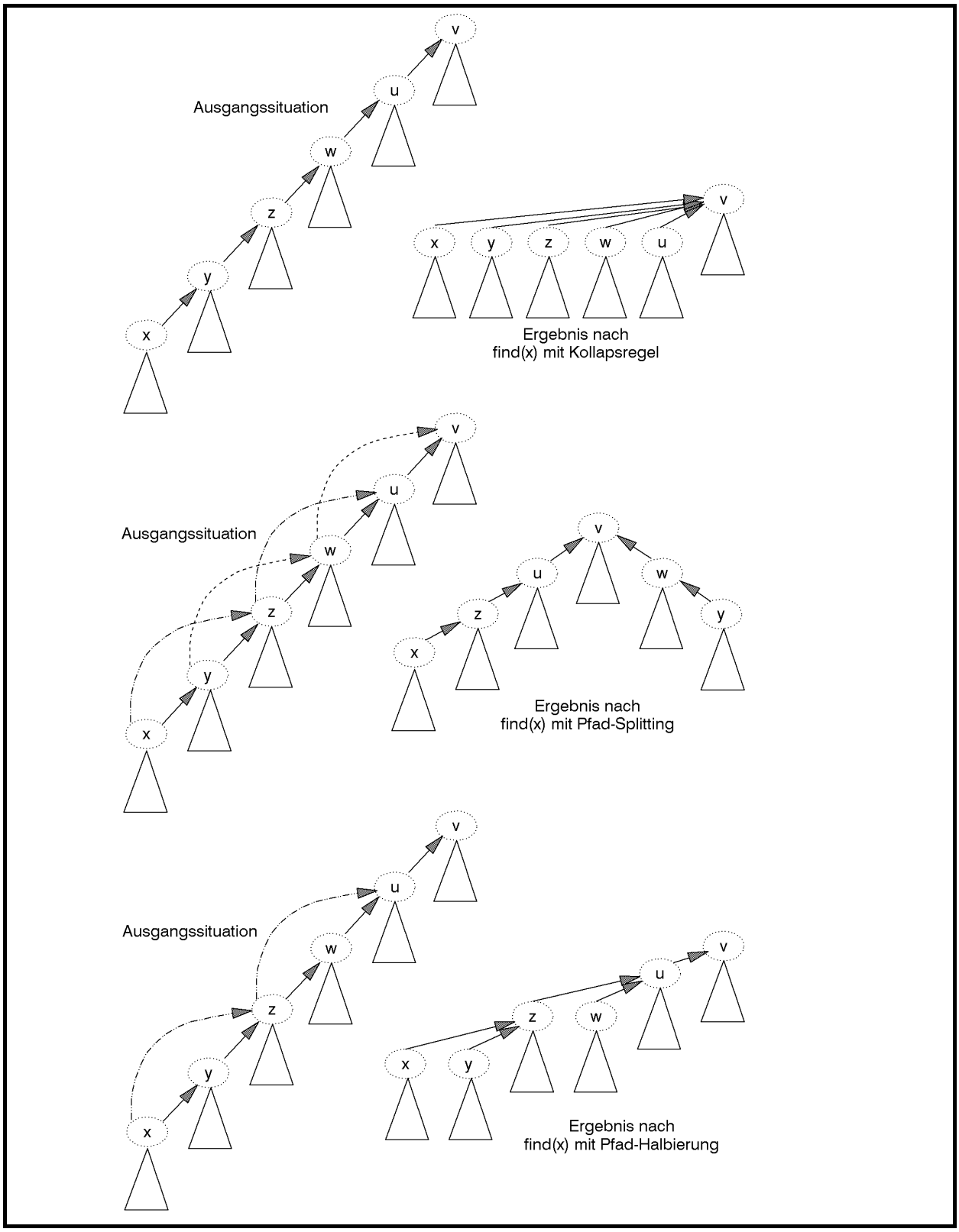


Abbildung 5.1.3-3: Pfadkompression

5.2 Nichtlineare Datenstrukturen

Unter einer **nichtlinearen Datenstruktur** werden alle übrigen Datenstrukturen zusammengefasst. Dazu gehören insbesondere diejenigen, auf denen Operationen definiert sind, die davon Gebrauch machen, dass auf den Elementen der Datenstruktur eine Ordnungsrelation ≤ (Sortierkriterium) erklärt ist. Die genaue Definition der Ordnungsrelation wird in der Implementation des Objekttyps `Tentry` verborgen. Zum Vergleich zweier Objekte wird der Objekttyp `Tentry` im Interfaceteil der `UNIT Element` um Methoden `kleiner`, `gleich` und `groesser` erweitert, deren Implementierung im Implementationteil der `Unit element` liegt:

```
FUNCTION { Tentry. } kleiner (obj : Tentry) : BOOLEAN;
         { liefert den Wert TRUE, falls der Zustand des
           Objekts kleiner als der Zustand des Objekts obj
            ist                                          }
FUNCTION { Tentry. } gleich (obj : Tentry) : BOOLEAN;
         { liefert den Wert TRUE, falls der Zustand des
           Objekts gleich dem Zustand des Objekts obj ist     }
FUNCTION { Tentry. } groesser (obj : Tentry) : BOOLEAN;
         { liefert den Wert TRUE, falls der Zustand des
           Objekts größer als der Zustand des Objekts obj ist }
```

Auch hier verändern einige Operationen den gegenwärtigen Zustand eines Objekts der Datenstruktur, so dass wieder eine Operation ***show*** definiert wird, die den gegenwärtigen Zustand eines Objekts der Datenstruktur anzeigt.

5.2.1 Prioritätsschlangen

Auf einer Prioritätsschlange sind die Operationen ***init***, ***insert***, ***delete***, ***count***, ***is_member*** und ***min*** definiert (vgl. Abbildung 5-2).

Typische Anwendungen einer Prioritätsschlange sind das Führen einer Symboltabelle in einem Compiler (das Ordnungskriterium ist die lexikographische Reihenfolge der Namen), die Verwaltung von Datensätzen in einer Datei, die durch einen Primärschlüssel (Ordnungskriterium) identifizierbar sind oder die Verwaltung prioritätengesteuerter Warteschlangen in einem Betriebssystem (das Ordnungskriterium ist die Priorität).

Die Operationen auf einer Prioritätsschlange sollen in ihrem Zeitaufwand nicht von den *Zuständen* der Objekte abhängen; es kann jedoch eine Abhängigkeit zwischen dem Zeitaufwand zur Durchführung einer Operation und der Anzahl der in der Datenstruktur vorhandenen Elemente bestehen.

Bei der Realisierung einer Prioritätsschlange kann man die Ordnungsrelation auf den einzelnen Elementen nutzen. Beispielsweise bietet sich an, die Prioritätsschlange als geordnete (lückenlose) Tabelle mit aufsteigendem Ordnungskriterium in den Elementen zu implementieren, vorausgesetzt, die maximale Anzahl an Elementen ist von vornherein bekannt. Die Realisierung der Operation ***min*** greift dann gerade auf den ersten Tabelleneintrag zu. Die Methoden, die die Operationen ***insert*** und ***delete*** implementieren, müssen sicherstellen, dass auch nach ihrer Ausführung alle Tabelleneinträge lückenlos in der Tabelle stehen und dass die Ordnung der Elemente in der Tabelle erhalten bleibt. Die Methode für die Operation ***insert*** muss also das neu einzutragende Element an die richtige Position innerhalb der bereits in der Prioritätsschlange vorhandenen Elemente einfügen. Daher werden eventuell eine Reihe von Tabelleneinträgen, nämlich diejenigen, die einen größeren Ordnungswert aufweisen als das hinzuzufügende Element, auf die jeweils nächste Position verschoben. Eine Verschiebung von Tabelleneinträgen, jetzt um eine Position nach vorn, ist in der Regel auch bei der zur Operation ***delete*** gehörenden Methode erforderlich. Die Methode zur Realisierung der Operation ***is_member*** kann von der Ordnung der Tabelleneinträge Gebrauch machen, indem eine Binärsuche auf der Tabelle durchgeführt wird.

Wegen des zeitlichen Aufwands bei der Durchführung mehrerer ***insert***- und ***delete***-Operationen in dieser Realisierung, der sich durch die Verschiebung der Tabelleneinträge ergibt, sind jedoch andere Implementationen einer Prioritätsschlange vorzuziehen. Im folgenden soll eine Prioritätsschlange als binärer Suchbaum realisiert werden. Hierbei findet ein Ausgleich zwischen dem Aufwand aller Operationen statt, die für eine Prioritätsschlange definiert sind. Andere Methoden werden in der angegebenen Literatur beschrieben.

Ein **binärer Suchbaum** besteht aus einem Binärbaum mit durch Werte des Ordnungskriteriums markierten Knoten und gerichteten Kanten. Die Struktur des Baums berücksichtigt die Ordnungsrelation ≤ der Elemente. Es gilt für die Knotenmarkierung m jedes Knotens K: Alle Knotenmarkierungen m_l im *linken Teilbaum* unterhalb von K (das ist der Teilbaum, dessen Wurzel der linke Nachfolger von K ist) erfüllen die Bedingung $m_l \leq m$, und alle Knotenmarkierungen m_r im *rechten Teilbaum* unterhalb von K (das ist der Teilbaum, dessen Wurzel der rechte Nachfolger von K ist) erfüllen die Bedingung $m_r > m$.

Die Anzahl der gerichteten Kanten, die von der Wurzel beginnend durchlaufen werden müssen, um einen Knoten zu erreichen, heißt der **Rang des Knotens**. Die **Höhe des Baums** wird durch den maximalen Rang + 1 definiert. Ein Baum, der nur aus der Wurzel besteht, hat demnach die Höhe 1.

Abbildung 5.2.1-1 zeigt einen binären Suchbaum. Die Knotenmarkierungen sind hier Elemente vom Typ `INTEGER`. Die Pfeile stellen die Kanten dar. Ihre Positionierung an den Knoten beschreibt die Nachfolgerrelation der Knoten (linker Nachfolger, rechter Nachfolger).

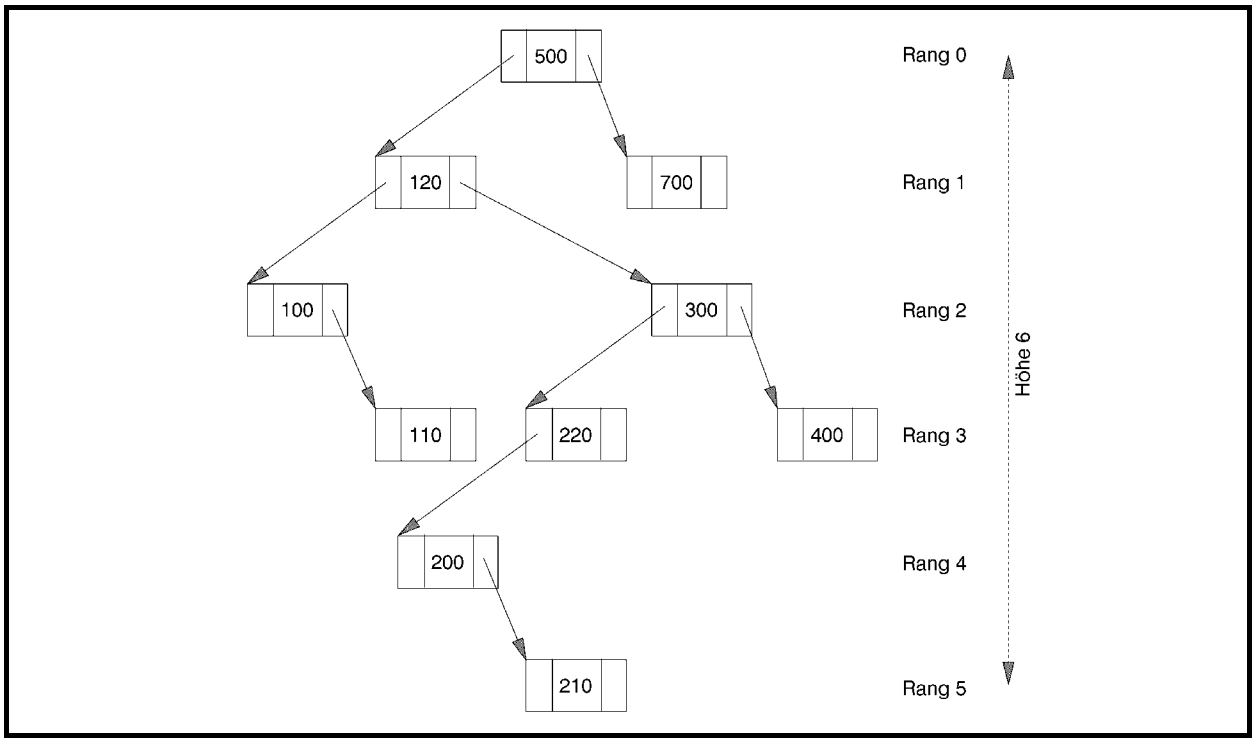


Abbildung 5.2.1-1: Binärer Suchbaum

Um eine Prioritätsschlange als binären Suchbaum zu realisieren, wird der Objekttyp `TPrio` deklariert. Ein Objekt dieses Typs bildet eine dynamisch erzeugte Verweisstruktur in Form eines binären Suchbaums, wobei die Knotenmarkierungen nicht die in ihr enthaltenen Elemente selbst, sondern (wie beim Objekttyp `TListe`) Verweise auf die entsprechenden Objekte sind. Mit der Bezeichnung *Element eines Knotens K* ist im folgenden das Element gemeint, auf das die Markierung im Knoten *K* verweist. Die Ordnungsrelation der Elemente wird dann offensichtlich auf die Knoten des binären Suchbaums übertragen, der zur Implementierung einer Prioritätsschlange aufgebaut wird.

Abbildung 5.2.1-2 zeigt ein Objekt vom Typ `TPrio` zur Implementierung der Elemente im binären Suchbaum von Abbildung 5.2.1-1. Das Ordnungskriterium der Elemente (Objekte) ist hier wieder vom Typ `INTEGER` und in den jeweiligen Objekten vermerkt.

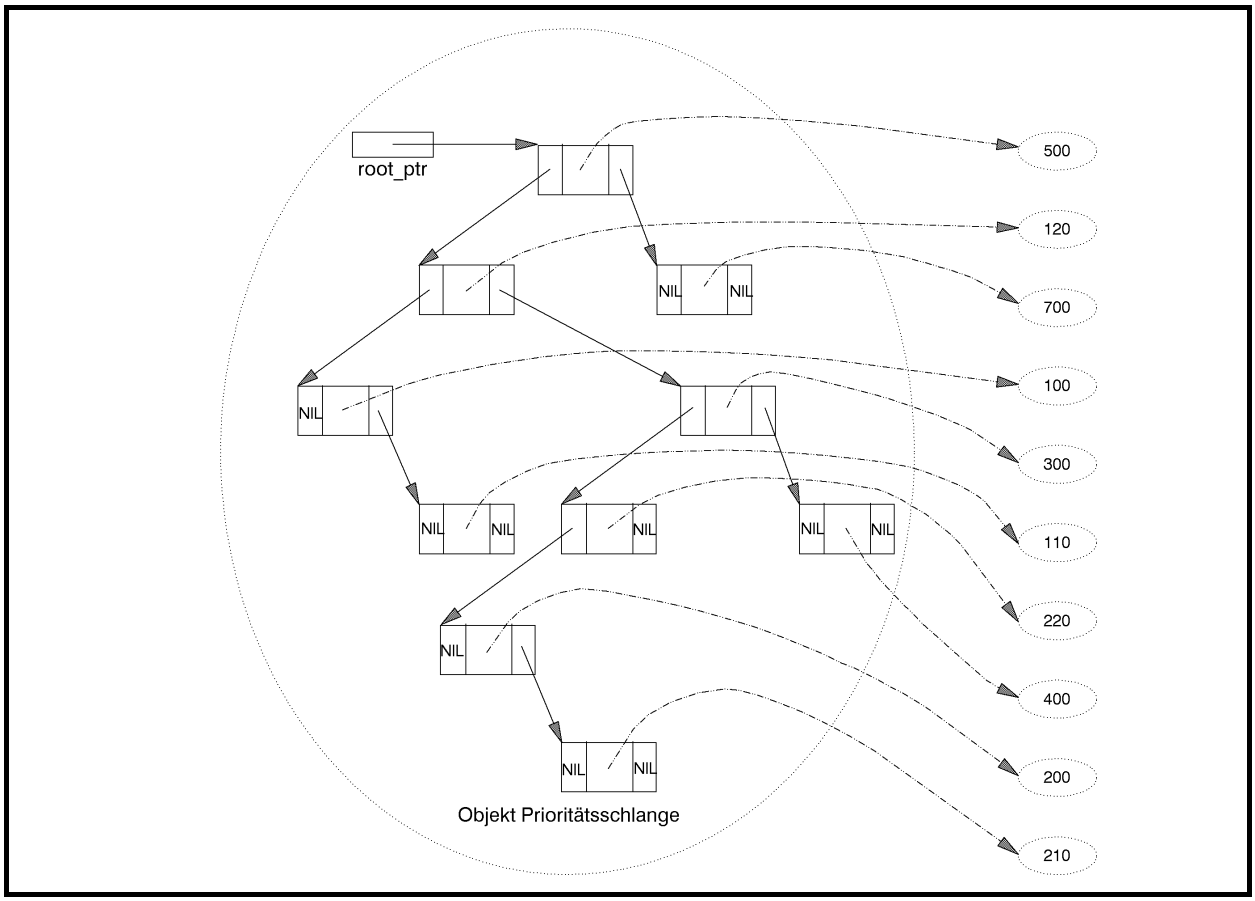


Abbildung 5.2.1-2: Prioritätsschlange (Beispiel)

Die folgende `UNIT Prio` implementiert die Datenstruktur Prioritätsschlange. Die Benutzerschnittstelle lautet:

Methode	Bedeutung
`CONSTRUCTOR` `TPrio.init;`	Die Prioritätsschlange wird als leer initialisiert.

`PROCEDURE`
`TPrio.insert`
`(entry_ptr : Pentry);`
Bedeutung des Parameters:
`entry_ptr` Verweis auf das einzutragende Element
	Ein neues Element wird in die Prioritätsschlange aufgenommen.

`PROCEDURE`
`TPrio.delete`
`(entry_ptr : Pentry);`
Bedeutung des Parameters:
`entry_ptr` Verweis auf das zu entfernende Element
	Das adressierte Element wird an den Aufrufer zurückgeliefert und aus der Prioritätsschlange entfernt.

`FUNCTION`
`TPrio.count:`
`INTEGER;`
	Die Funktion liefert die Anzahl der Einträge, die gegenwärtig in der Prioritätsschlange stehen.

`FUNCTION`
`TPrio.is_member`
`(entry_ptr : Pentry) :`
`BOOLEAN;`
Bedeutung des Parameters:
`entry_ptr` Verweis auf das zu überprüfende Element
	Die Funktion liefert den Wert `TRUE`, falls das durch `entry_ptr` adressierte Element in der Prioritätsschlange vorkommt, ansonsten den Wert `FALSE`.

`FUNCTION` `TPrio.min : Pentry;`	Die Funktion liefert einen Verweis auf das kleinste Element, das zur Zeit in der Prioritätsschlange vorkommt.
`DESTRUCTOR` `TPrio.done;`	Die Prioritätsschlange wird aus dem System entfernt.

Die Methode `TPrio.insert` zur Realisierung der ***insert***-Operation durchläuft den binären Suchbaum bei der Wurzel beginnend. Dabei wird bei jedem Knoten *K* geprüft, ob der Wert des Ordnungskriteriums im neu aufzunehmenden Element kleiner oder gleich dem Wert des Ordnungskriteriums des Elements des Knotens *K* ist oder nicht. Im ersten Fall wird zum linken Nachfolger von *K* verzweigt, im zweiten Fall zum rechten, bis ein Knoten gefunden ist, der an der betreffenden Nachfolgerstelle keinen Nachfolger besitzt. Hier wird ein neuer Knoten angehängt und mit einem Verweis auf das neu aufzunehmende Element markiert. Die Struktur des binären Suchbaums, der durch mehrere hintereinander ausgeführte `TPrio.insert`-Aufrufe entsteht, hängt wesentlich von der Reihenfolge der auftretenden Elemente und ihrer Ordnungskriterien ab.

In der Methode `TPrio.delete` zur Realisierung der ***delete***-Operation wird zunächst der Knoten gesucht, dessen Markierung auf das zu entfernende Element verweist. Falls ein derartiger Knoten existiert, wird geprüft, welche der drei möglichen Situationen, die in Abbildung

5.2.1-3 dargestellt sind, an diesem Knoten herrscht. Entsprechend wird anschließend ein Knoten aus dem binären Suchbaum gelöscht.

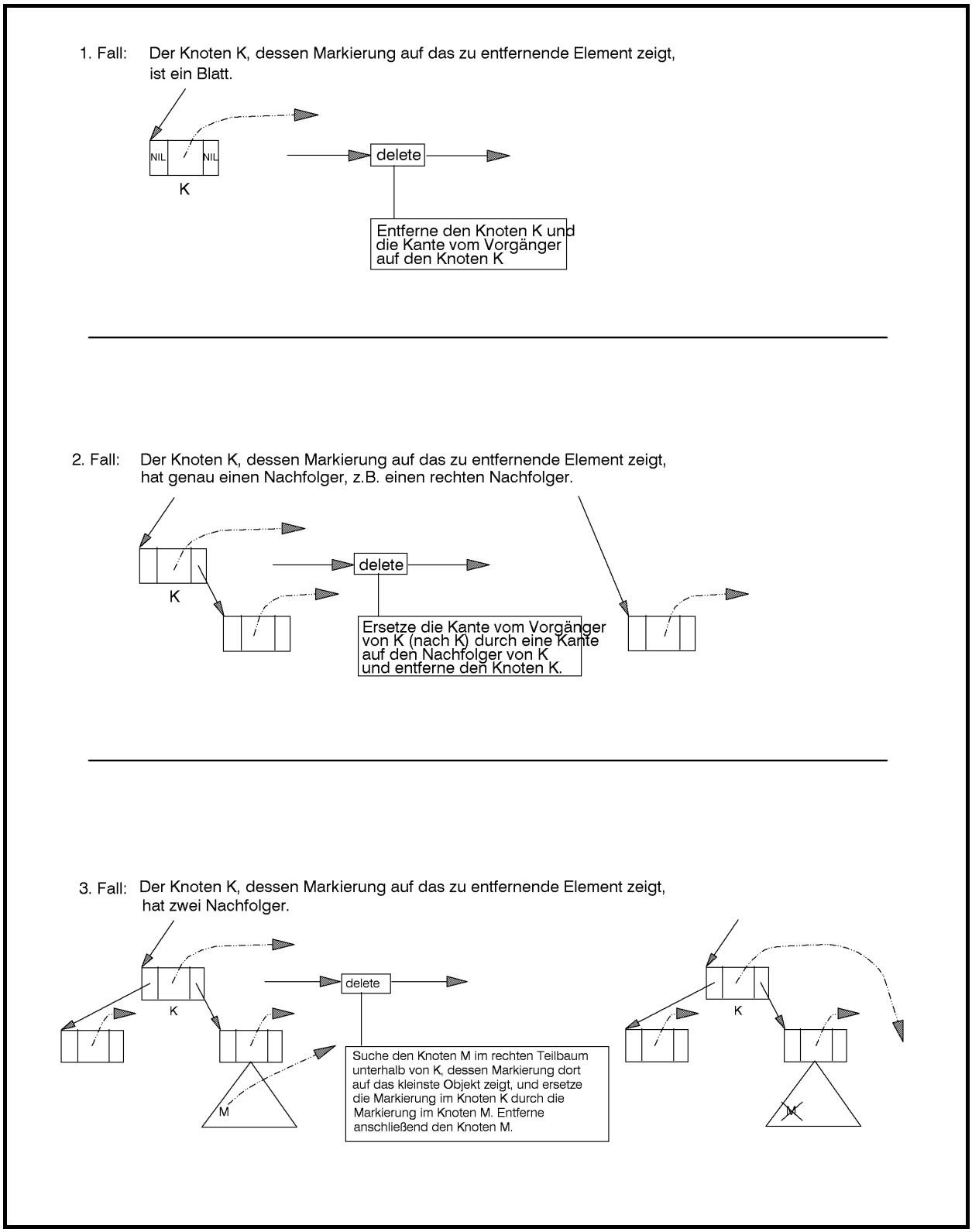


Abbildung 5.2.1-3: Methode `TPrio.delete`

Abbildung 5.2.1-4 zeigt die Prioritätsschlange aus Abbildung 5.2.1-3 nach einem Aufruf von `TPrio.delete` mit dem Element mit Ordnungskriteriums-Wert 120. Hier liegt der 3. Fall

vor: Nachdem der Knoten *K* des zu entfernenden Elements gefunden worden ist, wird der Knoten *M*, dessen Markierung auf das kleinste Objekt im rechten Teilbaum unterhalb *K* zeigt, gesucht; die im Knoten *M* stehende Markierung ersetzt die Markierung in *K*. Anschließend wird *M* entfernt. Offensichtlich wird durch `TPrio.delete` nicht notwendigerweise der Knoten entfernt, dessen Markierung auf das bezeichnete Element zeigt.

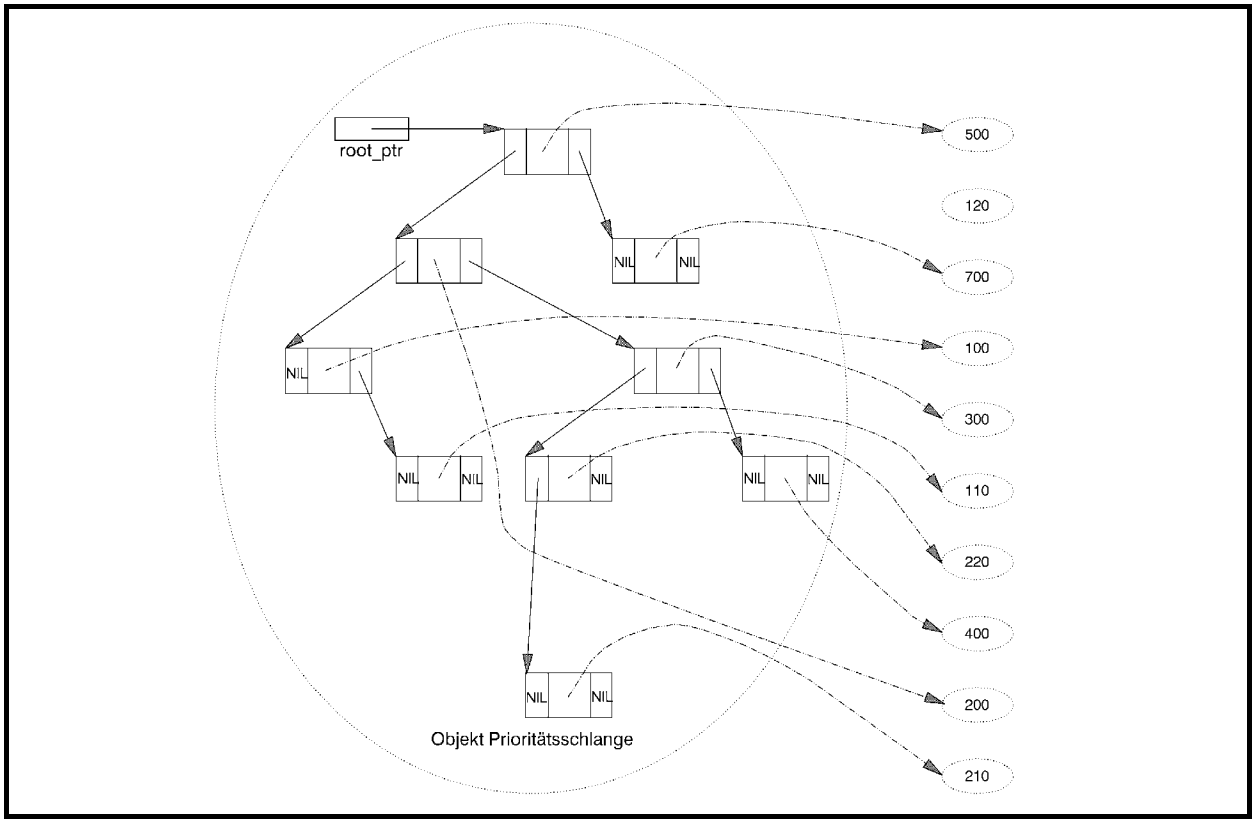


Abbildung 5.2.1-4: Ergebnis eines Aufrufs der Methode `TPrio.delete` (Beispiel)

Die zur ***is_member***-Operation gehörende Methode `TPrio.is_member` durchläuft den binären Suchbaum an der Wurzel beginnend ähnlich wie die Methode `TPrio.delete`. Dabei wird bei jedem Knoten geprüft, ob die dortige Markierung auf das im Aufruf der Methode TPrio.is_member spezifizierte Element zeigt oder nicht. Im ersten Fall wird das Vorhandensein des Elements festgestellt, im zweiten Fall wird zum linken Nachfolger verzweigt, wenn der Wert des Ordnungskriteriums des zu findenden Elements kleiner als das Ordnungskriterium im Element des gerade betrachteten Knotens ist, bzw. zum rechten Nachbarn, wenn er größer ist. Trifft man auf einen Knoten, der an der aufzusuchenden Nachfolgerstelle keinen Nachfolger besitzt, so ist das zu findende Element in der Prioritätsschlange nicht vorhanden.

Für die Realisierung der ***min***-Operation wird die Beobachtung genutzt, dass der Knoten, dessen Markierung auf das Element mit minimalem Wert des Ordnungskriteriums verweist, derjenige Knoten ist, der keinen linken Nachfolger hat, wenn man den Baum bei der Wurzel beginnend jeweils zum linken Nachfolger verzweigend durchläuft.

Eine Methode zur Realisierung der Operation ***show*** ist zunächst nicht implementiert; sie wird allgemein für Binärbäume in Kapitel 5.2.2 behandelt.

```
UNIT prio;

INTERFACE

   USES Element;

   TYPE PPrio = ^TPrio;
        TPrio = OBJECT
                  PRIVATE
                    root_ptr : Pointer;
                    anz      : INTEGER;
                  PUBLIC
                    CONSTRUCTOR init;
                    PROCEDURE insert (entry_ptr : Pentry);
                    PROCEDURE delete (entry_ptr : Pentry);
                    FUNCTION count : INTEGER;
                    FUNCTION is_member (entry_ptr : Pentry) :
                                                   BOOLEAN;
                    FUNCTION min : Pentry;
                    DESTRUCTOR done; VIRTUAL;
                END;

{ ************************************************************** }

IMPLEMENTATION

TYPE PPrioentry = ^TPrioentry;
     TPrioentry = RECORD
                    left   : PPrioentry;
                    marker : Pentry;
                    right  : PPrioentry;
                  END;


CONSTRUCTOR TPrio.init;

  BEGIN { TPrio.init }
    root_ptr := NIL;
    anz      := 0;
  END   { TPrio.init };


PROCEDURE TPrio.insert (entry_ptr : Pentry);

  PROCEDURE insert_ext (entry_ptr     : Pentry;
                        VAR tree_ptr  : PPrioentry);
  { fügt das Element, auf das entry_ptr zeigt, in den
    binären Suchbaum ein, der bei tree_ptr^ beginnt   }

    BEGIN { insert_ext }
      IF tree_ptr = NIL
      THEN BEGIN { einen neuen Knoten erzeugen und einfügen }
             New (tree_ptr);
```

```
            tree_ptr^.left   := NIL;
            tree_ptr^.marker := entry_ptr;
            tree_ptr^.right  := NIL;
          END
      ELSE       { Teilbaum suchen, in den der neue Knoten
                   eingefügt werden soll                    }
            IF tree_ptr^.marker^.kleiner (entry_ptr^)
            THEN insert_ext (entry_ptr, tree_ptr^.right)
            ELSE insert_ext (entry_ptr, tree_ptr^.left);
    END   { insert_ext };

  BEGIN { TPrio.insert }
    insert_ext (entry_ptr, PPrioentry(root_ptr));
    Inc(anz);
  END   { TPrio.insert };


PROCEDURE TPrio.delete (entry_ptr : Pentry);

  PROCEDURE delete_ext (entry_ptr : Pentry;
                        tree_ptr  : PPrioentry);
  { entfernt das Element, auf das entry_ptr zeigt, aus dem
    binären Suchbaum, der bei tree_ptr^ beginnt            }

  VAR mem_ptr  : PPrioentry;
      pred_ptr : PPrioentry;
      next_ptr : PPrioentry;

    BEGIN { delete_ext }
      { Knoten suchen, dessen Markierung auf das zu
        entfernende Element verweist, und dessen Vorgänger:}
      IF tree_ptr = NIL
      THEN { leerer Baum }
           mem_ptr := NIL
      ELSE BEGIN
             mem_ptr  := tree_ptr;
             pred_ptr := NIL;
             WHILE (mem_ptr <> NIL)
                   AND (mem_ptr^.marker <> entry_ptr) DO
               BEGIN
                 pred_ptr := mem_ptr;
                 IF NOT entry_ptr^.groesser (mem_ptr^.marker^)
                 THEN { im linken Teilbaum weitersuchen }
                      mem_ptr := mem_ptr^.left
                 ELSE { im rechten Teilbaum weitersuchen }
                      mem_ptr := mem_ptr^.right;
               END;
           END;

      IF mem_ptr <> NIL
      THEN BEGIN { mem_ptr zeigt auf den Knoten des zu entfernenden
                   Elements, pred_ptr auf dessen Vorgänger bzw. hat
                   den Wert NIL, wenn das Element der Wurzel zu
                   entfernen ist                                  }
             IF (mem_ptr^.left = NIL) OR (mem_ptr^.right = NIL)
             THEN BEGIN { 1. Fall oder 2. Fall }
                    IF mem_ptr^.right <> NIL
```

```
                    THEN next_ptr := mem_ptr^.right
                    ELSE next_ptr := mem_ptr^.left;
                    IF pred_ptr = NIL
                    THEN { Das Element der Wurzel ist zu entfernen }
                         tree_ptr := next_ptr
                    ELSE BEGIN
                           IF mem_ptr = pred_ptr^.left
                           THEN pred_ptr^.left  := next_ptr
                           ELSE pred_ptr^.right := next_ptr;
                         END;
                    Dec (anz);
                    { Speicherplatz freigeben }
                    Dispose (mem_ptr);
                  END   { 1. Fall oder 2. Fall }
              ELSE BEGIN { 3. Fall }
                    { Ermittlung des Knotens im rechten Teilbaum
                      unterhalb mem_ptr mit dem kleinsten Wert des
                      Ordnungskriteriums (Verweis in next_ptr)     }
                    next_ptr := mem_ptr^.right;
                    WHILE next_ptr^.left <> NIL DO
                      next_ptr := next_ptr^.left;
                    { Übernahme der dortigen Knotenmarkierung }
                    mem_ptr^.marker := next_ptr^.marker;
                    { Entfernung des überflüssigen Knotens aus dem
                      rechten Teilbaum unterhalb mem_ptr          }
                    delete_ext (next_ptr^.marker, mem_ptr^.right);
                  END   { 3. Fall };
            END;
    END   { delete_ext };

  BEGIN { TPrio.delete }
    delete_ext (entry_ptr, root_ptr);
  END   { TPrio.delete };


FUNCTION TPrio.count : INTEGER;

BEGIN { TPrio.count }
  count := anz;
END   { TPrio.count };


FUNCTION TPrio.is_member (entry_ptr : Pentry) : BOOLEAN;

  FUNCTION is_member_ext (entry_ptr : Pentry;
                          tree_ptr  : PPrioentry) : BOOLEAN;
  { sucht das Element entry in dem binären Suchbaum,
    der bei tree_ptr^ beginnt                        }

    BEGIN { is_member_ext }
      IF tree_ptr <> NIL
      THEN BEGIN
             IF tree_ptr^.marker = entry_ptr
             THEN is_member_ext := TRUE
             ELSE IF tree_ptr^.marker.kleiner (entry_ptr^)
             THEN is_member_ext
                       := is_member_ext (entry_ptr, tree_ptr^.right)
```

```
                  ELSE is_member_ext
                       := is_member_ext (entry_ptr, tree_ptr^.left);
           END
      ELSE is_member := FALSE;
    END   { is_member_ext };

  BEGIN { is_member }
    is_member := is_member_ext (entry_ptr, root_ptr);
  END   { is_member };


FUNCTION TPrio.min : Pentry;

VAR p : PPrioentry;

  BEGIN { TPrio.min }
    IF root_ptr = NIL
    THEN { leerer Baum } min := NIL
    ELSE BEGIN { im linken Teilbaum weitersuchen, bis es keinen
                 linken Nachfolger mehr gibt                    }
           p := root_ptr;
           WHILE p^.left <> NIL DO
             p := p^.left;
           min := p^.marker
         END;
  END   { TPrio.min };


DESTRUCTOR TPrio.done;

  BEGIN { TPrio.done }
    WHILE root_ptr <> NIL
      DO delete (PPrioentry(root_ptr)^.marker);
  END   { TPrio.done };

{ ************************************************************** }

END { prio }.
```

Der Vorteil der Implementation einer Prioritätsschlange durch einen binären Suchbaum liegt in der Einfachheit der Methoden. Bei gleichverteilten Werten des Ordnungskriteriums der Elemente (siehe unten) ist zu erwarten, dass die Struktur des binären Suchbaums ausgeglichen ist, d.h. dass möglichst viele Knoten auch zwei Nachfolger besitzen. Das kann natürlich nicht immer garantiert werden. Werden beispielsweise Elemente mit aufsteigenden bzw. absteigenden Werten des Ordnungskriteriums nacheinander in den binären Suchbaum aufgenommen, so entsteht eine lineare Kette von Knoten. Nachfolgende Operationen haben dann eine Laufzeit der Ordnung $O(n)$, falls n Elemente in der Struktur sind. Weiterhin wird bei der hier behandelten Implementierung angenommen, dass die Elemente selbst im Arbeitsspeicher liegen, dass also der Zugriff über den Verweis in einem Knoten „schnell“ erfolgt. Liegen jedoch die Elemente in einem Peripheriespeicher, so sind Plattenzugriffe erforderlich, und die Adressvergleiche zur Feststellung der Gleichheit zweier Elemente erscheint problematisch. Daher eignet sich ein binärer Suchbaum nur gut zur Implementierung „interner“ Prioritätsschlangen.

Die Komplexität der Operationen auf einer Prioritätsschlange hängt von der Anzahl der Knoten ab, die durchlaufen werden müssen, um die richtige Position eines Elements in der Prioritätsschlange zu finden. Um die **mittlere Anzahl der zu durchlaufenden Knoten** zu bestimmen, müssen einige Annahmen getroffen werden:

1. die Prioritätsschlange ist nur durch ***insert***-Operationen entstanden
2. in Bezug auf die Einfügereihenfolge seien alle Permutationen in der Menge des Sortierkriteriums der einzufügenden Elemente gleichwahrscheinlich.

Die mittlere Pfadlänge in der Prioritätsschlange ist dann ein Maß für die mittlere Anzahl der zu durchlaufenden Knoten. Bezeichnet $P(n)$ die mittlere Pfadlänge in der Prioritätsschlange mit n Elementen, wobei die beiden vorstehenden Annahmen zugrunde gelegt sind, dann gilt für große n [G/D]:

$$\lim_{n\to\infty} P(n) \le 1 + 2\cdot\ln(n+1) + 2\gamma - 3 \approx 1{,}386\cdot\log_2(n) + c$$

mit einer Konstanten $c > 0$, d.h. in einer zufällig erzeugten Prioritätsschlange ist (unter den genannten Voraussetzungen) die mittlere Komplexität von der Ordnung $O(\log(n))$ [12].

5.2.2 Durchlaufen von Binärbäumen

Der in Kapitel 5.2.1 definierte binäre Suchbaum (Objekttyp `TPrio`) ist ein Spezialfall eines Binärbaums, der im vorliegenden Abschnitt jetzt allgemeiner in `Unit tree` definiert wird. Da im vorliegenden Kapitel das Durchlaufen eines Binärbaums behandelt wird, sind nur die bezüglich dieser Operation relevanten Methoden angegeben. Hinzu kommt die Methoden zur Initialisierung der Struktur. Methoden zum Einfügen weiterer Knoten in die Struktur, zum Entfernen von Knoten aus der Struktur und der Destruktor zu Entfernung der gesamten Struktur werden im folgenden nicht weiter betrachtet; sie können analog zur Implementierung einer Prioritätsschlange erfogen (vgl. Kapitel 5.2.1). Der Schwerpunkt der Darstellung liegt auf Methoden zur Realisierung des Durchlaufens eines Baums und der Operation ***show***. Hierbei werden unterschiedliche Vorgehensweisen zum Durchlaufen des Baums betrachtet, die durch einen zusätzlichen Parameter in der zur ***show***-Operation gehörenden Implementierung identifiziert werden.

Details von `Unit tree` werden im Anschluss an den Programmcode erläutert.

---

12 Diese Aussage gilt nicht mehr, wenn sehr lange gemischte Folgen von ***insert***- und ***delete***-Operationen ausgeführt werden (siehe [G/D].

```
UNIT tree;

INTERFACE

   USES Element;

   TYPE show_typ = (praefix, symmetric, postfix, br_search);

   TYPE PTree = ^TTree;
        TTree = OBJECT
                  PRIVATE
                    root_ptr : Pointer;
                    anz      : INTEGER;
                  PUBLIC
                    CONSTRUCTOR init;
                     ...
                    PROCEDURE show (typ : show_typ); VIRTUAL;
                    ...
                END;

{ ************************************************************** }

IMPLEMENTATION

USES Listenstrukturen;

TYPE PTree_entry = ^TTree_entry;
     TTree_entry = RECORD
                     left   : PTree_entry;
                     marker : Pentry;
                     right  : PTree_entry;
                   END;


PROCEDURE show_praefix (ptr : PTree_entry);

  { Anzeigen aller Knoten in Präfixordnung, die im Baum stehen,
    auf dessen Wurzel ptr zeigt                                 }

  BEGIN { show_praefix }
    IF ptr <> NIL
    THEN BEGIN
           ptr^.marker^.display;
           show_praefix (ptr^.left);
           show_praefix (ptr^.right);
         END;
    END   { show_praefix };
```

```
PROCEDURE show_symmetric (ptr : PTree_entry);

  { Anzeigen aller Knoten in symmetrischer Ordnung,
    die im Baum stehen, auf dessen Wurzel ptr zeigt           }

  BEGIN { show_symmetric }
    IF ptr <> NIL
    THEN BEGIN
           show_symmetric (ptr^.left);
           ptr^.marker^.display;
           show_symmetric (ptr^.right);
         END;
    END   { show_symmetric };


PROCEDURE show_postfix (ptr : PTree_entry);

  { Anzeigen aller Knoten in Postfixordnung, die im Baum stehen,
    auf dessen Wurzel ptr zeigt                                 }

  BEGIN { show_postfix }
    IF ptr <> NIL
    THEN BEGIN
           show_postfix (ptr^.left);
           show_postfix (ptr^.right);
           ptr^.marker^.display;
         END;
  END   { show_postfix };


PROCEDURE show_br_search (ptr : PTree_entry);

VAR agenda_ptr : PFIFO;
    node_ptr   : PTree_entry;

  BEGIN { show_br_search }
    IF ptr <> NIL
    THEN BEGIN
           { Agenda einrichten }
           New (agenda_ptr, init);
           { Wurzel des Baums bearbeiten und Verweise auf
             weitere Knoten in die Agenda stellen; dann die
             Agenda abarbeiten                              }
           node_ptr := ptr;
           WHILE node_ptr <> NIL DO
```

```
            BEGIN
              { Element des Knotens anzeigen }
              node_ptr^.marker^.display;
              { Nachfolgerknoten in die Agenda stellen }
              IF node_ptr^.left <> NIL
              THEN agenda_ptr^.insert (Pentry(node_ptr^.left));
              IF node_ptr^.right <> NIL
              THEN agenda_ptr^.insert (Pentry(node_ptr^.right));
              { einen weiteren Knoten aus der Agenda holen }
              agenda_ptr^.delete (Pentry(node_ptr));
            END;
          { Agenda wieder entfernen }
          Dispose (agenda_ptr, done);
        END;
  END   { show_br_search };

{ ---------------------------------------------------------------- }

CONSTRUCTOR TTree.init;

  BEGIN { TTree.init }
    root_ptr := NIL;
    anz      := 0;
  END   { TTree.init };

...

PROCEDURE TTree.show (typ : show_typ);

BEGIN { TTree.show }
  CASE typ OF
  praefix   : show_praefix (root_ptr);
  symmetric : show_symmetric (root_ptr);
  postfix   : show_postfix (root_ptr);
  br_search : show_br_search (root_ptr);
  END;
END   {TTree.show };


...

END { tree }.
```

Der Objekttyp eines Binärbaums (im Interfaceteil von `Unit tree`) ist

```
TYPE PTree = ^TTree;
     TTree = OBJECT
```

```
PRIVATE
  root_ptr : Pointer;
  anz      : INTEGER;
PUBLIC
  CONSTRUCTOR init;
  ...
  PROCEDURE show (typ : show_typ); VIRTUAL;
  ...
 END;
```

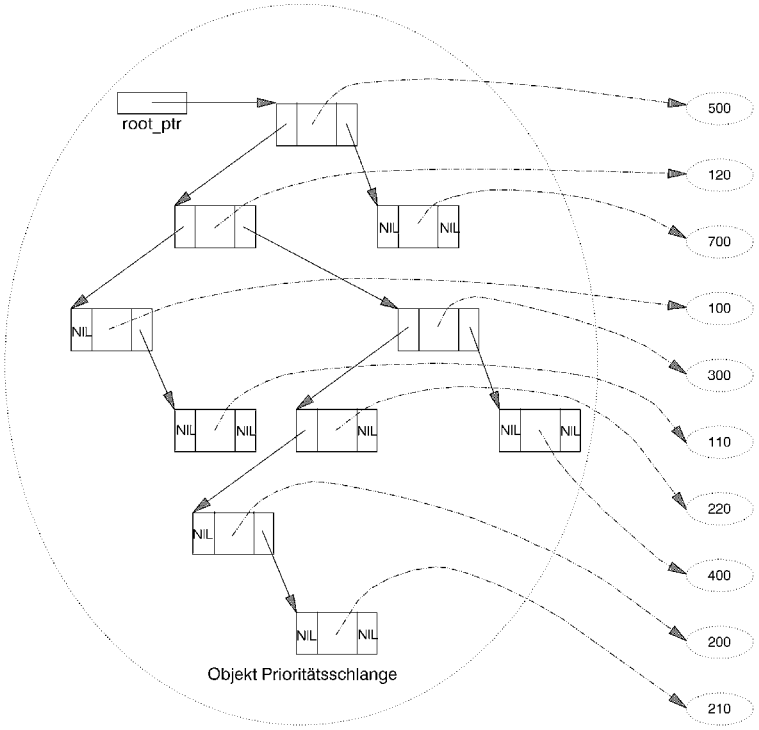
Ein Knoten des Binärbaums wird durch den Typ (im Implementierungsteil von `Unit Tree`)

```
TYPE PTree_entry = ^TTree_entry;
     TTree_entry = RECORD
                     left   : PTree_entry;
                     marker : Pentry;
                     right  : PTree_entry;
                   END;
```

beschrieben. Er hat einen Verweis auf zwei Nachfolger (einen linken und einen rechten Nachfolger), auf einen Nachfolger (einen linken oder einen rechten Nachfolger) oder keinen Nachfolger. In der Komponente steht ein Verweis auf ein Element, das zu diesem Knoten gehört (vgl. Abbildung 5.2.1-2). Die Wurzel des Baums ist über eine Pointervariable `root_ptr` erreichbar. Weitere Bedingungen über die Knotenmarkierungen (wie die Ordnungsrelation der Elemente, die zu den Knoten im linken bzw. rechten Teilbaum unterhalb eines Knotens gehören, wie bei einem binären Suchbaum) werden nicht berücksichtigt.

Eine Methode zur ***show***-Operation muss den gesamten Baum durchlaufen, wobei der Weg nur über die gerichteten Kanten erfolgen kann. Bei jedem Knoten wird das Element angezeigt, auf das die jeweilige Knotenmarkierung verweist. Dabei kann die „rekursive“ Struktur eines Baums genutzt werden: Ein nicht-leerer Baum besteht ja entweder nur aus der Wurzel *W* oder aus der Wurzel *W* und einem linken oder rechten Teilbaum, dessen Wurzel der linke bzw. rechte Nachfolger von *W* ist. Dabei kann einer dieser Teilbäume auch leer sein. Entsprechend wird man bei der Methode die Wurzel aufsuchen und die Methode dann (rekursiv) auf den linken bzw. rechten Teilbaum unterhalb der Wurzel anwenden. Je nachdem, in welcher Reihenfolge man in die Teilbäume unterhalb eines Knotens *W* geht und das Element von *W* anzeigt, ergibt sich eine andere Reihenfolge aller angezeigten Elemente. In `Unit tree` sind drei mögliche Realisierungen der Operation ***show*** angegeben. Man bezeichnet die Arbeitsweisen als Durchlaufen des **Baums in Präfixordnung**, **symmetrischer Ordnung** bzw. **Postfixordnung**. Entsprechende Prozeduren sind mit `show_praefix`, `show_symmetric` bzw. `show_postfix` bezeichnet und im Implementationsteil von `Unit tree` definiert. Alle drei Methoden führen auf dem Binärbaum eine **Tiefensuche (depth-first-search)** durch.

Die Ergebnisse der Methodenaufrufe `TTree.show` mit den für die Tiefensuche zu verwendenden Parametern `praefix`, `symmetric` bzw. `postfix` ist in Abbildung 5.2.2-1 wiedergegeben.



Anzeigen der Knotenmarkierungen in Präfixordnung

Der Aufruf von `TTree.show (praefix)` bzw. dann `show_praefix (root_ptr)` führt auf die Reihenfolge der angezeigten Knotenmarkierungen des binären Suchbaums:

500, 120, 100, 110, 300, 220, 200, 210, 400, 700

Anzeigen der Knotenmarkierungen in symmetrischer Ordnung

Der Aufruf von `TTree.show (symmetric)` bzw. dann `show_symmetric (root_ptr)` führt auf die Reihenfolge der angezeigten Knotenmarkierungen des binären Suchbaums:

100, 110, 120, 200, 210, 220, 300, 400, 500, 700

Anzeigen der Knotenmarkierungen in Postfixordnung

Der Aufruf von `TTree.show (postfix)` bzw. dann `show_postfix (root_ptr)` führt auf die Reihenfolge der angezeigten Knotenmarkierungen des binären Suchbaums:

110, 100, 210, 200, 220, 400, 300, 120, 700, 500

Abbildung 5.2.2-1: Tiefensuche auf einem Binärbaum

Eine andere Reihenfolge, um die Knotenmarkierungen anzuzeigen, spiegelt die Hierarchie in einem Baum wider: Zuerst wird das Element der Wurzel (Rang 0) angezeigt, anschließend die Elemente aller Nachfolger der Wurzel, d.h. aller Knoten mit Rang 1. Allgemein wird zu den Knoten des Rangs i übergegangen, nachdem alle Knoten des Rangs $i - 1$ besucht wurden $(1 \leq i \leq \text{Höhe des Baums})$. Diese Art des Durchlaufen eines Baums heißt **Breitensuche (breadth first search)**. Die Implementierung der Breitensuche auf einem Binärbaum erfolgt durch die Prozedur `show_br_search` (im Implementierungsteil der Unit `Tree`) und wird von außen durch einen Aufruf der Methode `TTree.show (br_search)` angestoßen.

Das Durchlaufen eines Baums gemäß Breitensuche kann die Verweisstruktur zwischen den Knoten nicht direkt nutzen, da es keine Verweise zu den Nachbarknoten auf gleichem Niveau gibt. Während des Ablaufs der Breitensuche werden sukzessive Verweise auf Knoten des Baums in eine ***Agenda***, ein Objekt vom Objekttyp FIFO-Warteschlange (`TYPE TFIFO`), eingefügt und entfernt:

Es wird die Wurzel bearbeitet und Verweise auf den oder die Nachfolger der Wurzel in die Agenda eingetragen. Solange die Agenda nicht leer ist, werden die folgenden drei Schritte durchgeführt:

1. ein Verweis auf einen Knoten K wird der Agenda entnommen;
2. der Knoten K wird bearbeitet, d.h. das zu K gehörende Element wird angezeigt
3. falls K einen oder zwei Nachfolger besitzt, werden Verweise auf den bzw. die Nachfolger von K in die Agenda eingetragen.

Denkt man sich die Knoten des Baums bei der Wurzel beginnend nach aufsteigenden Niveaus und auf jedem Niveau von links nach rechts durchnumeriert (Startnummer ist die Nummer 1), so enthält die Agenda (zwischen `last` und `first` und falls sie nicht bereits leer ist) zu jedem Zeitpunkt Verweise auf Knoten mit lückenlos aufsteigenden Nummern i, $i + 1$, ..., $i + k$ (Abbildung 5.2.2-2).

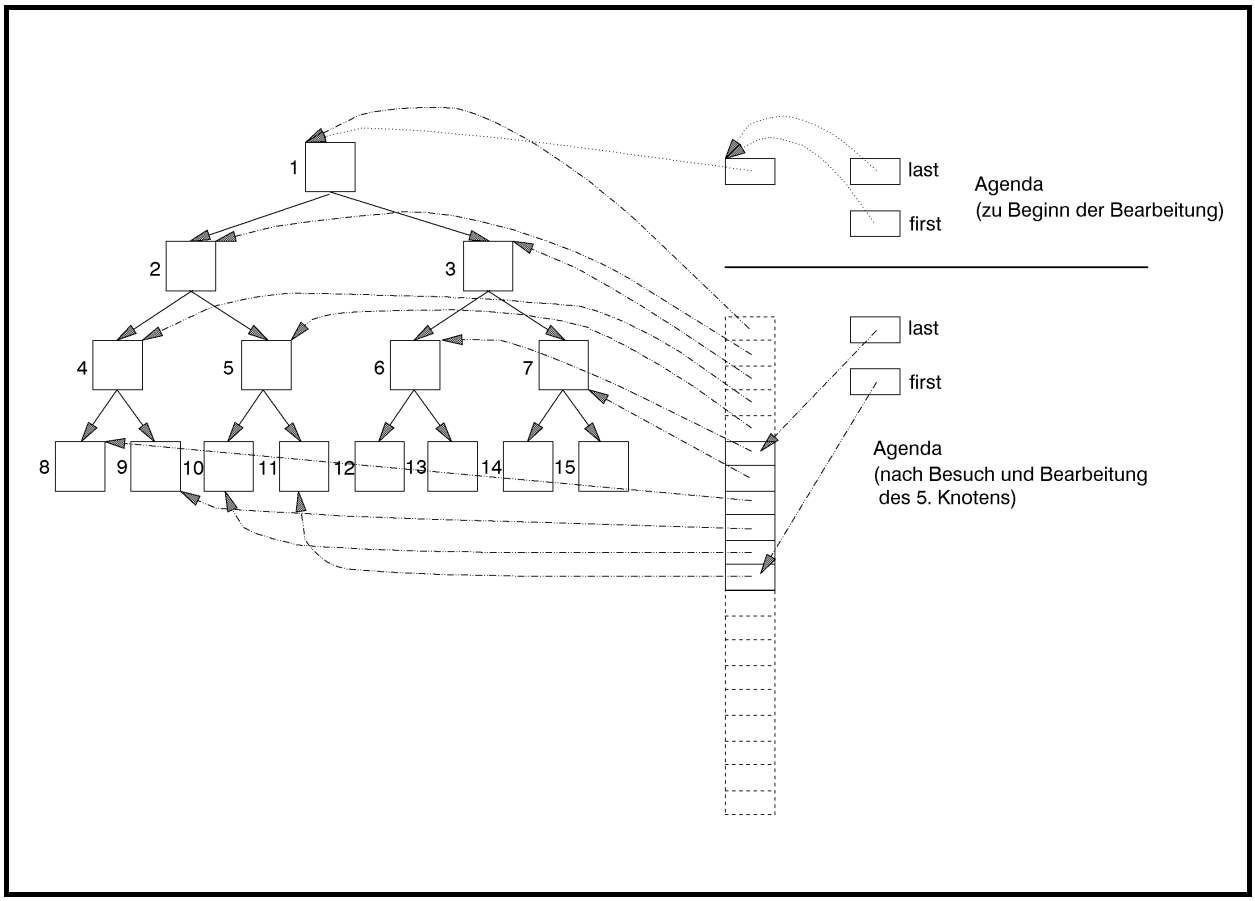


Abbildung 5.2.2-2: Breitensuche auf einem Binärbaum

Tiefensuche und Breitensuche können in ein einheitliches Konzept gefasst werden. Ersetzt man beispielswiese in der Prozedur `show_br_search` die Typdeklaration der Agenda durch

```
VAR agenda_ptr : PStack;
```

so liefert ein Aufruf von `show_br_search (root_ptr)` die Knotenmarkierungen in inverser Postfixordnung (im Beispiel der Abbildung 5.2.2-1 in der Reihenfolge 500, 700, 120, 300, 400, 220, 200, 210, 100, 110). Je nachdem, ob für die Agenda eine FIFO-Warteschlange oder ein Stack eingesetzt wird erhält man Breitensuche bzw. Tiefensuche).

5.2.3 Graphen

Zwei übliche Darstellungsformen eines gewichteten gerichteten Graphen $G = (V, E, w)$ mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, der Kantenmenge $E \subseteq V \times V$ und der Gewichtsfunktion $w : E \to \mathbf{R}$, die jeder Kante $e \in E$ ein Gewicht zuordnet, sind seine

- Adjanzenzmatrix
- Adjazenzliste (Knoten/Kantenliste).

Die **Adjazenzmatrix** $A(G)$ ist definiert durch $A(G) = \lfloor a_{i,j} \rfloor_{n \times n}$ mit

$$a_{i,j} = \begin{cases} w((v_i, v_j)) & \text{für } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases} \qquad \text{für } 1 \le i \le n, 1 \le j \le n .$$

Bei einem ungewichteten Graphen sind die Einträge der Adjazenzmatrix durch

$$a_{i,j} = \begin{cases} 1 & \text{für } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases} \qquad \text{für } 1 \le i \le n, 1 \le j \le n$$

definiert.

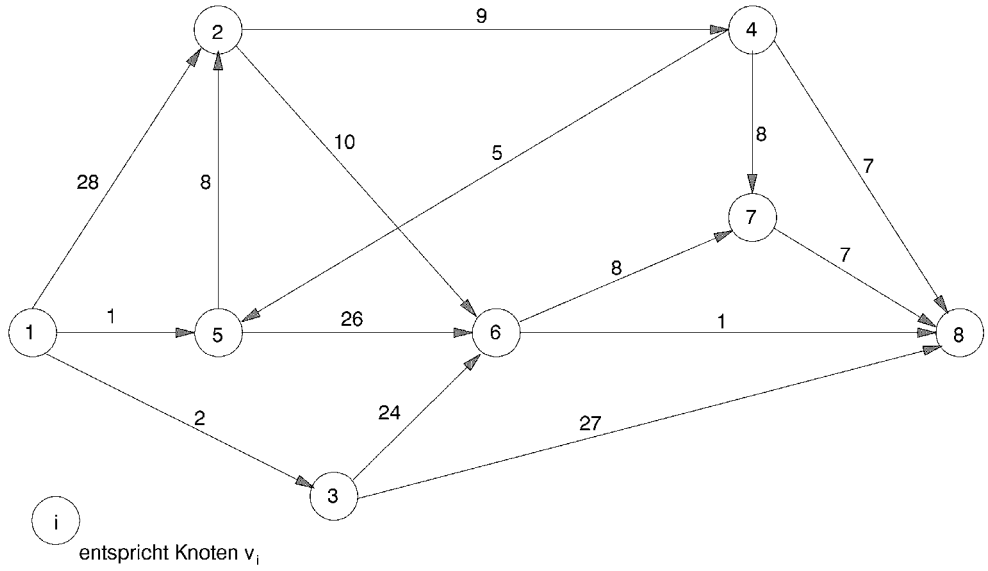
Die **Adjazenzliste (Knoten/Kantenliste)** verwaltet für jeden Knoten v_i eine Liste, die die direkten Nachfolgerknoten von v_i zusammen mit dem jeweiligen Gewicht der verbindenden Kante enthält.

Bei der Darstellung eines Graphen in Form seiner Adjazenzmatrix kann man in einer Laufzeit der Ordnung $O(1)$ feststellen, ob er eine Kante (v_i, v_j) enthält. Der Nachteil dieser Darstellung ist ihr Speicherplatzbedarf der Ordnung $O(n^2)$, vor allem dann, wenn die Anzahl der Kanten im Verhältnis zu n^2 sehr klein ist. Die Initialisierung der Adjazenzmatrix erfordert in jedem Fall mindestens n^2 viele Schritte. Das Auffinden aller k Nachfolger eines Knotens v_i benötigt $O(n)$ viele Schritte, d.h. eine Schrittzahl, die ungünstig ist, falls ein Knoten nur wenige Nachfolger besitzt.

Enthält der Graph n Knoten und e Kanten, dann ist der Platzbedarf der Darstellung mittels einer Adjazenzliste von der Ordnung $O(n+e)$. Das Auffinden aller k Nachfolger eines Knotens v_i benötigt $O(k)$ viele Schritte. Man kann jedoch nicht mehr in konstanter Zeit prüfen, ob zwei Knoten v_i und v_j durch eine Kante (v_i, v_j) verbunden sind.

Je nach durchzuführenden Operationen in der Anwendung ist eine der Darstellungsformen (Adjazenzmatrix oder Adjazenzliste) daher besser geeignet als die andere.

Abbildung 5.2.3-1 zeigt ein Beispiel.



Die Adjazenzmatrix lautet:

$$A(G) = \begin{bmatrix} \infty & 28 & 2 & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & \infty & 9 & \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 24 & \infty & 27 \\ \infty & \infty & \infty & \infty & 5 & \infty & 8 & 7 \\ \infty & 8 & \infty & \infty & \infty & 26 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 8 & 1 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Die Adjazenzliste lautet:

v_1: $[v_2, 28]$, $[v_3, 2]$, $[v_5, 1]$

v_2: $[v_4, 9]$, $[v_6, 10]$

v_3: $[v_6, 24]$, $[v_8, 27]$

v_4: $[v_5, 5]$, $[v_7, 8]$, $[v_8, 7]$

v_5: $[v_2, 8]$, $[v_6, 26]$

v_6: $[v_7, 8]$, $[v_8, 1]$

v_7: $[v_8, 7]$

v_8:

Abbildung 5.2.3-1: Gewichteter gerichteter Graph (Beispiel)

Im folgenden wird eine exemplarische Implementierung eines Graphen in einer `UNIT Graph` vorgestellt. Zunächst werden einige grundlegende Methoden behandelt wie das Durchlaufen eines Graphen nach unterschiedlichen Vorgehensweisen. In den späteren Kapiteln wird diese Unit durch weitere Methoden ergänzt. Außerdem erfolgt eine Beschränkung der Kantengewichte auf `INTEGER`-Werte.

Die Adjazenzmatrix wird als dynamisch vereinbarte zweidimensionale Matrix (siehe Kapitel 5.1.2) vereinbart. Dabei ist zu beachten, dass die Anzahl *n* der Knoten den Wert der Konstanten `max_dimension` nicht überschreitet.

Erläuterungen zum Programmcode werden im Anschluss gegeben.

```
UNIT Graph;

INTERFACE

   USES ARRAY_basierte_Liste;

   TYPE show_typ      = (tiefensuche, breitensuche);
        Knotenbereich = 1..max_dimension;

        PGraph = ^TGraph;
        TGraph = OBJECT
                  PRIVATE
                    n              : INTEGER;
                          { Anzahl der Knoten                }
                    Adjazenzmatrix : PMatrix;
                          { Adjazenzmatrix                   }
                    Adjazenzliste  : PHashtab;
                          { Adjazenzliste (Knoten/Kantenliste }
                    PROCEDURE t_suche (v : Knotenbereich);
                          { Tiefensuche vom Knoten v aus     }
                    PROCEDURE b_suche (v : Knotenbereich);
                          { Breitensuche vom Knoten v aus    }
                  PUBLIC
                    CONSTRUCTOR init (anz_knoten : INTEGER;
                                      anz_kanten : INTEGER;
                                      VAR OK     : BOOLEAN);
                    ...
                    PROCEDURE show (typ : show_typ); VIRTUAL;
                    ...
                    DESTRUCTOR done; VIRTUAL;
                 END;

{ ********************************************************** }
```

```
IMPLEMENTATION

USES SysUtils { Delphi - Entwicklungsumgebung },
     { Strings { Pascal - Entwicklungsumgebung ,}
     Listenstrukturen,
     Kante,
     Element;

CONST unendlich = 16353; { hier kann der maximal mögliche
                           INTEGER-Wert verwendet werden      }
      leer : PChar = '';

VAR besucht : PChar;

FUNCTION Hashfunktion (entry_ptr : Pentry) : INTEGER; FAR;
           { Die Funktion ermittelt den Wert der Hashfunktion
             für das durch entry_ptr adressierte Objekt       }
BEGIN { Hashfunktion }
  Hashfunktion := PKante(entry_ptr)^.Startknoten - 1;
           { Numerierung in der Hashtabelle: 0, ..., m-1      }
END   { Hashfunktion };

{ ------------------------------------------------------------ }

CONSTRUCTOR TGraph.init (anz_knoten : INTEGER;
                         anz_kanten : INTEGER;
                         VAR OK     : BOOLEAN);

VAR i   : Knotenbereich;
    j   : Knotenbereich;
    p   : PKante;
    v_i : Knotenbereich;
    v_j : Knotenbereich;
    w   : INTEGER;

BEGIN { TGraph.init }
  OK := TRUE;
  Adjazenzmatrix := NIL;
  Adjazenzliste  := NIL;

  IF anz_knoten <= max_dimension
  THEN BEGIN
         n := anz_knoten;
         New (Adjazenzliste, init(anz_knoten, Hashfunktion, OK));
         IF OK
         THEN New (Adjazenzmatrix, init (n, OK))
         ELSE Adjazenzmatrix := NIL;
```

```
        IF OK
        THEN BEGIN
               { Hier wird der Code zur Initialisierung der
                 Adjazenzliste bzw. der Adjazenzmatrix
                 eingefügt, z.B. (Pseudocode)                }
               FOR i := 1 TO n DO
                 FOR j := 1 TO n DO
                   Adjazenzmatrix^.setze (i, j, unendlich);
               FOR j := 1 TO anz_kanten DO
                 BEGIN
                   v_i := v_i;
                   v_j := v_j;
                   w   := w(v_i, v_j);
                   New (p, init (v_i, v_j, w));
                   Adjazenzliste^.insert (p);
                   Adjazenzmatrix^.setze (v_i, v_j, w);
                 END;
             END;
      END
  ELSE OK := FALSE;
END   { TGraph.init };

...

PROCEDURE TGraph.show (typ : show_typ);

VAR i : Knotenbereich;

  BEGIN { TGraph.show }
    besucht := StrAlloc (n + 1);
    StrCopy (besucht, leer);
    FOR i := 1 TO n DO
      StrCat (besucht, 'F');

    CASE typ OF
    tiefensuche  : FOR i := 1 TO n DO
                     t_suche (i);
    breitensuche : FOR i := 1 TO n DO
                     b_suche (i);
    END;
    StrDispose (besucht);
  END   {TGraph.show };


PROCEDURE TGraph.t_suche (v : Knotenbereich);
```

```
VAR ptr       : PKante;
    start_ptr : PKante;

BEGIN { TGraph.t_suche }
  New (start_ptr, init (v, 0, 0));
  IF besucht[v - 1] = 'F'
  THEN BEGIN
         -- Bearbeiten des Knotens, z.B. Anzeigen der Knotennummer;
         besucht[v - 1] := 'T';
         ptr := PKante(Adjazenzliste^.choose (start_ptr, c_first));
         WHILE ptr <> NIL DO
           BEGIN
             t_suche (ptr^.Endknoten);
             ptr := PKante(Adjazenzliste^.choose
                                          (start_ptr, c_next));
           END;
       END;
  Dispose (start_ptr, done);
END   { TGraph.t_suche };


PROCEDURE TGraph.b_suche (v : Knotenbereich);

VAR agenda_ptr : PFIFO;
    edge_ptr   : PKante;
    dummy_ptr  : PKante;
    continue   : BOOLEAN;

BEGIN { TGraph.b_suche }
  { Agenda einrichten }
  New (agenda_ptr, init);
  New (dummy_ptr, init (v, 0, 0));
  edge_ptr := dummy_ptr;

  WHILE edge_ptr <> NIL DO
    BEGIN
      IF besucht[v - 1] = 'F'
      THEN BEGIN
             -- Bearbeiten des Knotens, z.B. Anzeigen der Knotennummer;
             besucht[v - 1] := 'T';
             edge_ptr := PKante(Adjazenzliste^.choose
                                          (edge_ptr, c_first));
             WHILE edge_ptr <> NIL DO
               BEGIN
                 IF besucht[edge_ptr^.Endknoten - 1] = 'F'
                 THEN agenda_ptr^.insert (edge_ptr);
                 edge_ptr := PKante(Adjazenzliste^.choose
```

```
                                                (edge_ptr, c_next));
                END;
            END;
      agenda_ptr^.delete (Pentry(edge_ptr));
      IF edge_ptr <> NIL
      THEN BEGIN
             v := edge_ptr^.Endknoten;
             Dispose (dummy_ptr, done);
             New (dummy_ptr, init (v, 0, 0));
             edge_ptr := dummy_ptr;
            END;
    END;

  Dispose (dummy_ptr, done);
  Dispose (agenda_ptr, done);
 END   { TGraph.b_suche };

...

PROCEDURE Alle_Kanten_entfernen (entry_ptr : Pentry); FAR;

BEGIN { Alle_Kanten_entfernen }
  Dispose (PKante(entry_ptr), done);
END   { Alle_Kanten_entfernen };


DESTRUCTOR TGraph.done;

  BEGIN { TGraph.done }
    IF Adjazenzmatrix <> NIL
    THEN Dispose (Adjazenzmatrix, done);
    IF Adjazenzliste <> NIL
    THEN BEGIN
            Adjazenzliste^.for_all (Alle_Kanten_entfernen);
            Dispose (Adjazenzliste, done);
         END;
END   { TGraph.done };

{ ************************************************************ }

END { Graph }.
```

Die Benutzerschnittstelle des Objekttyps `TGraph` lautet:

Methode	Bedeutung
`CONSTRUCTOR TGraph.init (anz_knoten : INTEGER; anz_kanten : INTEGER; VAR OK : BOOLEAN);`	Der Graph wird mit der angegebenen Anzahl an Knoten und Kanten initialisiert und in das System eingelesen.
Bedeutung der Parameter:	
`anz_knoten`	Anzahl der Knoten des Graphen
`anz_kanten`	Anzahl der Kanten des Graphen
`OK`	= `TRUE` steht für die erfolgreiche Initialisierung
`PROCEDURE TGraph.show (typ : show_typ); VIRTUAL;`	Alle Knoten des Graphen werden angezeigt.
Bedeutung des Parameters:	
`typ`	Der Graph wird in Tiefensuche bzw. Breitensuche verarbeitet (siehe unten).
`DESTRUCTOR TGraph.done;`	Der Graph wird aus dem System entfernt.

Die Adjazenzmatrix wird mit Hilfe des Objekttyps `TMatrix` als dynamisches `Array` deklariert und ihre Adresse in der Komponente `Adjazenzmatrix` abgelegt. Die Adjazenzliste ist vom Typ `PHashtab`. Alle von einem Knoten v_i ausgehenden Kanten (v_i, v_{i_1}), ..., (v_i, v_{i_k}) werden mit Start- und Endknoten und dem jeweiligen Gewicht in der Hashtabelle mit Hashwert $i-1$ abgelegt. Hierbei wird vorausgesetzt, dass die Knoten von 1 bis n durchnumeriert sind. Als Hasfunktion für die Hashtabelle wird daher die oben angegebene Hashfunktion verwendet (zu beachten ist, dass die Hashtabelle intern die Numerierung 0, ..., $n-1$ verwendet).

Der Objekttyp einer Kante lautet `TKante` und ist in der folgenden Unit deklariert:

```
UNIT Kante;

INTERFACE

USES Element;

   { Objekttypdeklarationen (Bezeichner, Methoden) einer Kante:   }
   TYPE PKante = ^TKante;
        TKante = OBJECT (Tentry)
                   PRIVATE
                     v_i      : INTEGER;
                     v_j      : INTEGER;
                     w        : REAL;
                   PUBLIC
                     CONSTRUCTOR init (i : INTEGER;
```

```
                                        j : INTEGER;
                                        v : REAL);
                        FUNCTION Startknoten : INTEGER;
                        FUNCTION Endknoten : INTEGER;
                        FUNCTION Gewicht : REAL;
                        PROCEDURE display; VIRTUAL;
                    END;

IMPLEMENTATION

{ ************************************************************** }

CONSTRUCTOR TKante.init (i : INTEGER;
                         j : INTEGER;
                         v : REAL);
  BEGIN { TKante.init }
    INHERITED init;
    v_i := i;
    v_j := j;
    w   := v;
  END   { TKante.init };


FUNCTION TKante.Startknoten : INTEGER;

  BEGIN { TKante.Startknoten }
    Startknoten := v_i;
  END   { TKante.Startknoten };


FUNCTION TKante.Endknoten : INTEGER;

  BEGIN { TKante.Endknoten }
    Endknoten := v_j;
  END   { TKante.Endknoten };


FUNCTION TKante.Gewicht : REAL;

  BEGIN { TKante.Gewicht }
    Gewicht := w;
  END   { TKante.Gewicht };

PROCEDURE TKante.display;

  BEGIN { TKante.display }
    ...
```

```
  END   { TKante.display };

{ ************************************************************** }

END.
```

Die Benutzerschnittstelle des Objekttyps `TKante` lautet:

Methode	Bedeutung
`CONSTRUCTOR TKante.init (i : INTEGER; j : INTEGER; v : REAL);`	Die Kante wird initialisiert.
Bedeutung der Parameter:	
`i`	Startknoten der Kante
`j`	Endknoten der Kante
`v`	Kantengewicht
`FUNCTION TKante.Startknoten : INTEGER;`	Die Funktion liefert den Startknoten der Kante.
`FUNCTION TKante.Endknoten : INTEGER;`	Die Funktion liefert den Endknoten der Kante.
`FUNCTION TKante.Gewicht : Real;`	Die Funktion liefert das Gewicht der Kante.
`PROCEDURE TKante.display;`	Die Prozedur zeigt die Kante an (Start- und Endknoten, Gewicht);

Eine der wichtigsten Grundoperationen auf einem Graphen ist das **systematische Durchlaufen des Graphen** entlang seiner Kanten, beispielsweise um die Knoten (-nummern) anzuzeigen oder die Gewichte der durchlaufenen Kanten aufzusummieren. In Kapitel 5.2.2 wurde die Operation ***show*** für einen Binärbaum angegeben, die nacheinander die Knotenmarkierungen des Baums anzeigt. Im folgenden werden Prinzipien vorgestellt, nach denen (allgemeine) Graphen durchlaufen werden können.

Zunächst sollen Graphen betrachtet werden, in denen alle Knoten von einem definierten Knoten aus, der Wurzel, erreichbar sind. Derartige Graphen bezeichnet man als **Wurzelgraphen**. Ein Beispiel zeigt Abbildung 5.2.2-1 mit der durch 1 markierten Wurzel. Auch hier unterscheidet man wieder Tiefensuche und Breitensuche. Beide Methoden lassen sich mit Hilfe eines Baums erklären, der jedoch nicht explizit konstruiert wird, sondern nur als Gedankenmodell dient.

Der Graph $G = (V, E)$ besitze die Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und die Kantenmenge $E \subseteq V \times V$. Seine Wurzel sei r. Auf eine Gewichtung der Kanten sei hier

verzichtet, da nur das systematische Durchlaufen aller Knoten beschrieben werden soll. Die **Expansion** $X(v)$ von G in einem Knoten v ist der Baum, der wie folgt definiert wird:

(i) Hat v keinen Nachfolger, so ist $X(v)$ der Baum, der nur aus dem Knoten v besteht

(ii) Sind $v_1, ..., v_k$ alle Nachfolger des Knotens v, so ist $X(v)$ der Baum mit Wurzel v, an die die Expansionen $X(v_1), ..., X(v_k)$ angehängt sind.

Die Expansion $X(1)$ im Beispiel des Graphen aus Abbildung 5.2.3-1 ist in Abbildung 5.2.3-2 zu sehen.

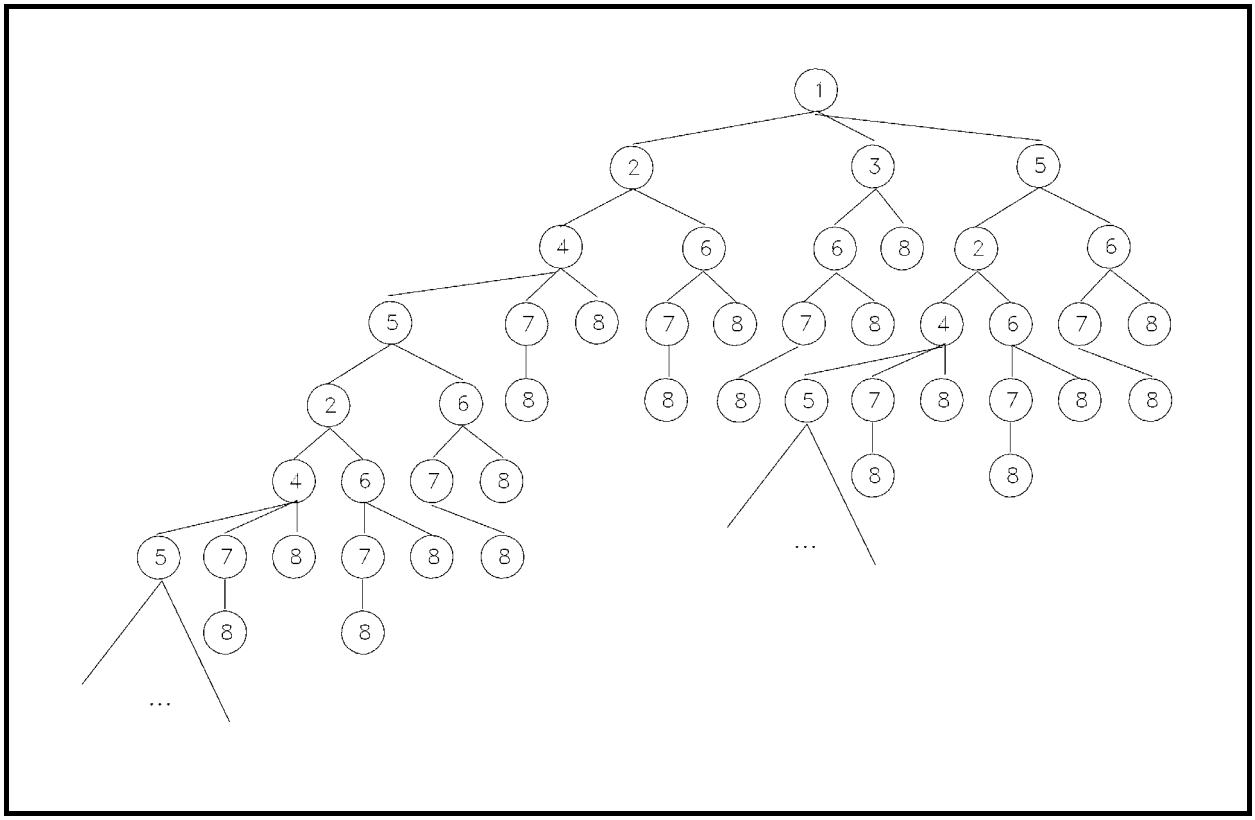


Abbildung 5.2.3-2: Expansion $X(1)$ des Graphen aus Abbildung 5.2.3-1

Die Expansion $X(r)$ stellt also die Menge aller von der Wurzel r ausgehenden Pfade in G dar. Sie ist ein unendlicher Baum, falls G Zyklen enthält. Die **Tiefensuche** in G ist das Durchlaufen der Expansion $X(r)$ in Präfixordnung, wobei jeweils in einem bereits besuchten Knoten von G abgebrochen wird. Die **Breitensuche** in G ist das Durchlaufen der Expansion $X(r)$ in Breitensuche, wobei ebenfalls jeweils in einem bereits besuchten Knoten von G abgebrochen wird.

Im Graphen der Abbildung 5.2.3-1 liefert Tiefensuche die Knotenreihenfolge 1, 2, 4, 5, 6, 7, 8, 3. Breitensuche liefert die Knotenreihenfolge 1, 2, 3, 5, 4, 6, 8, 7.

In einem **allgemeinen Graphen** (ohne Bedingung, dass alle Knoten von einer Wurzel aus erreichbar sind), können bei einem Durchlaufen unbesuchte Knoten übrig bleiben. Bei diesen ist dann jeweils ein neuer Durchlauf zu beginnen.

Tiefen- bzw. Breitensuche sind in den Methoden

```
PROCEDURE TGraph.t_suche (v : Knotenbereich);
```

bzw.

```
PROCEDURE TGraph.b_suche (v : Knotenbereich);
```

implementiert. Dabei wird vom Knoten `v` ausgegangen, d.h. die Expansion $X(\mathtt{v})$ verfolgt. Damit ein Knoten nicht mehrmals besucht wird, kontrolliert die Komponente `TGraph.besucht` (hier als nullterminierte Zeichenkette mit n Einträgen mit jeweiligem Wert `'F'` initialisiert), ob der Knoten mit der Nummer i bereits bearbeitet wurde; in diesem Fall wird `TGraph.besucht[`$i-1$`]` auf den Wert `'T'` gesetzt. Tiefen- und Breitensuche orientieren sich am Vorgehen bei Binärbäumen (Kapitel 5.2.2). Bei der Tiefensuche wird wie dort eine als Agenda bezeichnete FIFO-Warteschlange eingesetzt, die Informationen über die später noch zu besuchenden Knoten aufnimmt. In der vorliegenden Implementierung werden jedoch nicht die Knotennummern der unmittelbaren Nachfolger eines zu bearbeitenden Knotens in die Agenda eingetragen, sondern Verweise auf die verbindenden Kanten (falls der Endknoten noch nicht besucht worden ist); der (technische) Grund liegt in der Tatsache, dass in der Adjazenzliste komplette Kanten verwaltet werden.

6 Anwendungsorientierte Basisalgorithmen

Im vorliegenden Kapitel werden grundlegende Verfahren zum Sortieren und Suchen behandelt.

6.1 Sortierverfahren

Zu den wichtigsten anwendungsorientierten Basisalgorithmen zählen **Verfahren zur Sortierung einer ungeordneten Menge von Elementen**, auf denen eine Ordnungsrelation ≤ (Sortierkriterium) erklärt ist. Die Elemente haben mindestens zwei Komponenten: einen Primärschlüssel und einen Informationsteil, so dass ein Element in der Form $a = (key, info)$ darstellbar ist; die *key*-Komponente ist der Primärschlüssel, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \le a_j$ genau dann, wenn $key_i \le key_j$ ist.

Die **Sortieraufgabe** besteht im Entwurf eines Verfahrens, das bei Eingabe einer Folge $\langle a_1, ..., a_n \rangle$ von n Elementen der Form $a_i = (key_i, info_i)$ eine Umordnung $\pi : [1:n] \to [1:n]$ der Elemente ermittelt, so dass $\langle a_{\pi(1)}, ..., a_{\pi(n)} \rangle$ nach aufsteigenden Werten der *key*-Komponente sortiert ist, d.h. es gilt: $a_{\pi(i)} \le a_{\pi(i+1)}$ für $i = 1, ..., n-1$.

Die Elemente werden in Objekten vom Objekttyp `TSortentry` abgelegt, die sich in der `UNIT SortElement` befindet:

```
UNIT SortElement;

INTERFACE

   USES Element;

   TYPE Tkey  = { z.B. } INTEGER;
        Tinfo = ...;

        PSortentry = ^TSortentry;
        TSortentry = OBJECT  (Tentry)
                       PRIVATE
                         key  : Tkey;
                         info : Tinfo;
```

```
                        PUBLIC
                          CONSTRUCTOR init (k : Tkey;
                                            i : Tinfo );
                          FUNCTION kleiner (obj : TSortentry):
                                                    BOOLEAN;
                          FUNCTION gleich (obj : TSortentry) :
                                                    BOOLEAN;
                          FUNCTION groesser (obj : TSortentry) :
                                                    BOOLEAN;
                          FUNCTION getkey  : Tkey;
                          FUNCTION getinfo : Tinfo;
                          PROCEDURE show;
                      END;


{ ************************************************************ }

IMPLEMENTATION

CONSTRUCTOR TSortentry.init (k : Tkey;
                             i : Tinfo);
  BEGIN { TSortentry.init }
    key  := k;
    info := i;
  END   { TSortentry.init };


FUNCTION TSortentry.kleiner (obj : TSortentry) : BOOLEAN;

  BEGIN { TSortentry.kleiner }
    IF key < obj.key
    THEN kleiner := TRUE
    ELSE kleiner := FALSE;
  END   { TSortentry.kleiner };


FUNCTION TSortentry.gleich (obj : TSortentry) : BOOLEAN;

  BEGIN { TSortentry.gleich }
    IF key = obj.key
    THEN gleich := TRUE
    ELSE gleich := FALSE;
  END   { TSortentry.gleich };


FUNCTION TSortentry.groesser (obj : TSortentry) : BOOLEAN;
```

```
  BEGIN { TSortentry.groesser }
    IF key > obj.key
    THEN groesser := TRUE
    ELSE groesser := FALSE;
  END   { TSortentry.groesser };


FUNCTION TSortentry.getkey  : Tkey;

  BEGIN { TSortentry.getkey }
    getkey := key;
  END   { TSortentry.getkey };


FUNCTION TSortentry.getinfo  : Tinfo;

  BEGIN { TSortentry.getinfo }
    getinfo := info;
  END   { TSortentry.getinfo };


PROCEDURE TSortentry.show;

BEGIN { TSortentry.show }
  { ... }
END   { TSortentry.show };

{ ************************************************************ }

END.
```

Die folgende Tabelle beschreibt die Benutzerschnittstelle des Objekttyps `TSortentry`.

Methode	Bedeutung

`CONSTRUCTOR`
`TSortentry.init`
`k : Tkey;`
`i : Tinfo);`
Bedeutung der Parameter:
`k` Primärschlüssel
`i` Infoteil
	Ein Element wird initialisiert.

`FUNCTION`
`TSortentry.kleiner`
`(obj : TSortentry) :`
`BOOLEAN;`
Bedeutung des Parameters:
`obj` Vergleichselement
	Die Funktion liefert den Wert `TRUE`, falls der Primärschlüssel des Elements kleiner als der Primärschlüssel des im Parameter übergebenen Elements ist, ansonsten den Wert `FALSE`.

`FUNCTION`
`TSortentry.gleich`
`(obj : Tsortentry) :`
`BOOLEAN;`
Bedeutung des Parameters:
`obj` Vergleichselement
	Die Funktion liefert den Wert `TRUE`, falls der Primärschlüssel des Elements gleich dem Primärschlüssel des im Parameter übergebenen Elements ist, ansonsten den Wert `FALSE`.

`FUNCTION`
`TSortentry.groesser`
`(obj : TSortentry) :`
`BOOLEAN;`
Bedeutung des Parameters:
`obj` Vergleichselement
	Die Funktion liefert den Wert `TRUE`, falls der Primärschlüssel des Elements größer als der Primärschlüssel des im Parameter übergebenen Elements ist, ansonsten den Wert `FALSE`.

`PROCEDURE` `TSortentry.show;`	Die Komponten des Elements werden angezeigt.
`FUNCTION getkey :` `Tkey;`	Die Funktion liefert den Primärschlüsselwert eines Elements.
`FUNCTION getinfo :` `Tinfo;`	Die Funktion liefert den Informationsteil eines Elements.

Die zu sortierenden Elemente werden durch eine Anwendung erzeugt und an ein Objekt vom Objekttyp `TSortierfeld` (beispielsweise mit Hilfe des Konstruktors bei der Initialisierung des Objekts) übergeben. Ein derartiges Objekt wird im folgenden als **Sortierfeld** bezeichnet. Dieses verwaltet Verweise auf die zu sortierenden Elemente in einem `ARRAY`, das dynamisch während der Initialisierung erzeugt wird und über einen Pointer angesprochen wird, der sich im `PRIVATE`-Teil des Objekts befindet (ähnlich wie bei einer Hashtabelle, siehe Kapitel 5.1.2). Zusätzlich wird dort die Anzahl zu sortierender Elemente festgehalten. Ein Objekt vom Objekttyp `TSortierfeld` stellt über eine Typdeklaration

```
TYPE TSortiere = (bubble, quick, heap, bottum_up_heap);
```

und eine Methode

```
PROCEDURE sortieren (typ : TSortiere);
```

eine Reihe wichtiger Sortierverfahren zur Verfügung, die in den folgenden Unterkapiteln besprochen werden.

Abbildung 6.1-1 zeigt ein Objekt vom Objekttyp `TSortierfeld`, mit dessen Hilfe $n = 7$ Elemente sortiert werden können; die Elemente sind bereits dem Objekt übergeben worden, aber noch nicht sortiert (die Elemente sind hier als Quadrate wiedergegeben, die den jeweiligen Primärschlüsselwert beinhalten).

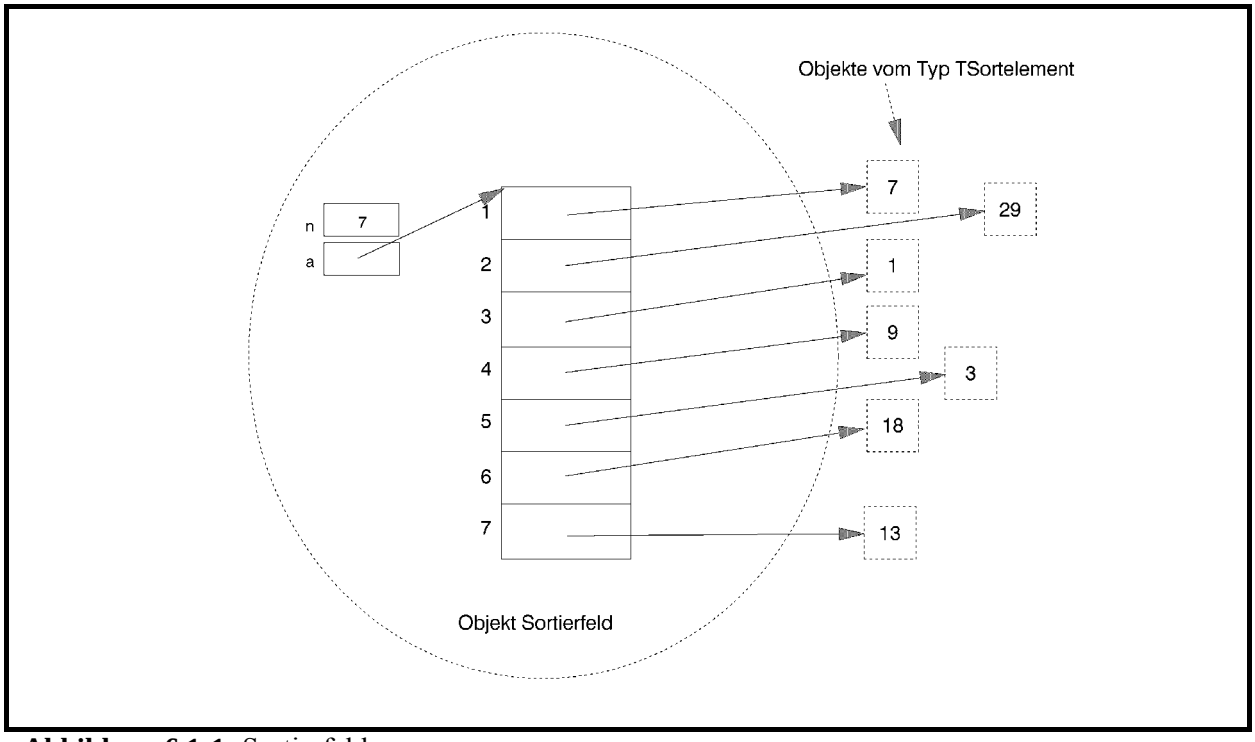


Abbildung 6.1-1: Sortierfeld

Im folgenden ist die `Unit Sortiere` in Ausschnitten aufgeführt, die den Objekttyp `TSortierfeld` bereitstellt.

Die Benutzerschnittstelle lautet:

Methode	Bedeutung
`CONSTRUCTOR Tsortierfeld.init (anz : INTEGER; VAR OK : BOOLEAN);`	Das Sortierfeld wird initialisiert.
Bedeutung der Parameter: `anz` Anzahl zu sortierender Elemente; `OK` = `TRUE`, falls die Initialisierung erfolgreich verlaufen ist	
`PROCEDURE Tsortierfeld.sortieren (typ : TSortiere);`	Die Elemente werden sortiert. Dabei sind Bubblesort, Quicksort, Heapsort und Bottum-up-Heapsort implementiert (siehe folgende Unterkapitel).
Bedeutung des Parameters: `typ` anzuwendendes Sortierverfahren	
`FUNCTION TSortierfeld.count : INTEGER;`	Die Funktion liefert die Anzahl zu sortierender Elemente
`PROCEDURE TSortierfeld.show (i : INTEGER);`	Die Methode zeigt ein Element an, das über das Sortierfeld verwaltet wird.
Bedeutung des Parameters: `i` Nummer des Elements innerhalb des Sortierfelds	
`DESTRUCTOR TSortierfeld.done;`	Das Sortierfeld wird aus dem System entfernt.

```
UNIT Sortiere;

INTERFACE

   TYPE TSortiere = (bubble, quick, heap, bottum_up_heap);

   TYPE PSortierfeld = ^TSortierfeld;
        TSortierfeld = OBJECT
                         PRIVATE
                           n : INTEGER;
                               { Anzahl zu sortierender Elemente   }
                           a : POINTER;
                               { Verweis auf das ARRAY mit den
                                 Verweisen auf die Elemente        }
                           PROCEDURE exchange (i : INTEGER;
                                               j : INTEGER);
                           { die Verweise an den Positionen i und j
                             werden vertauscht                     }
```

```
                        { Sortierverfahren: }
                            PROCEDURE bubblesort (lower : INTEGER;
                                                  upper : INTEGER);
                            PROCEDURE quicksort (lower : INTEGER;
                                                 upper : INTEGER);
                            PROCEDURE heapsort (typ   : TSortiere;
                                                lower : INTEGER;
                                                upper : INTEGER);
                          PUBLIC
                            CONSTRUCTOR init (anz    : INTEGER;
                                              VAR OK : BOOLEAN);
                            PROCEDURE sortieren (typ : TSortiere);
                            FUNCTION count : INTEGER;
                            PROCEDURE show (i : INTEGER);
                            DESTRUCTOR  done; VIRTUAL;
                      END;

{ ********************************************************** }

IMPLEMENTATION

USES SortElement;

CONST n_max = ...;   { max. Anzahl zu sortierender Element    }

TYPE Pfeld     = ^Tfeld;
     Tfeld     = ARRAY [1..n_max] OF PSortentry;
     Trec_proc = PROCEDURE (a : Pfeld;
                            i : INTEGER;
                            j : INTEGER);

{ ---------------------------------------------------------- }

CONSTRUCTOR TSortierfeld.init (anz    : INTEGER;
                               VAR OK : BOOLEAN);

VAR idx : INTEGER;

  BEGIN { TSortierfeld.init }
    OK := TRUE;
    IF anz <= n_max
    THEN BEGIN
           n := anz;
           GetMem (a, SizeOf(Pointer) * n);
           IF a = NIL
           THEN OK := FALSE
           ELSE BEGIN
```

```
                  { Hier wird der Code zur Initialisierung des
                    Felds zu Verwaltung des Sortiervorgangs
                    Eingefügt, etwa die Übergabe aller
                    Elemente an das Sortierfeld                }
                   ...
                END;
         END
    ELSE OK := FALSE;
  END   { TSortierfeld.init };


PROCEDURE TSortierfeld.exchange (i : INTEGER;
                                 j : INTEGER);

VAR ptr : PSortentry;

  BEGIN { TSortierfeld.exchange }
    IF (i >= 1) AND (i <= n) AND (j >= 1) AND (j <= n)
    THEN BEGIN
           ptr          := Tfeld(a^)[i];
           Tfeld(a^)[i] := Tfeld(a^)[j];
           Tfeld(a^)[j] := ptr;
         END;
  END   { TSortierfeld.exchange };


PROCEDURE TSortierfeld.bubblesort (lower : INTEGER;
                                   upper : INTEGER);
...
BEGIN { TSortierfeld.bubblesort }
  ...
END { TSortierfeld.bubblesort };


PROCEDURE TSortierfeld.quicksort (lower       : INTEGER;
                                  upper       : INTEGER);
...
BEGIN { TSortierfeld.quicksort }
...
END { TSortierfeld.quicksort };


PROCEDURE TSortierfeld.heapsort (typ   : TSortiere;
                                 lower : INTEGER;
                                 upper : INTEGER);
...
BEGIN { TSortierfeld.heapsort }
```

```
  CASE typ OF
  heap           : ...;
  bottum_up_heap : ...;
  ELSE Exit;
  END;
...
END   { TSortierfeld.heapsort };


PROCEDURE TSortierfeld.sortieren (typ : TSortiere);

  BEGIN { TSortierfeld.sortieren }
    CASE typ OF
    bubble         : bubblesort (1, n);
    quick          : BEGIN
                       Randomize;
                         { Zufallszahlengenerator für den
                           Quicksort initialisieren          }
                       quicksort (1, n);
                     END;
    heap,
    bottum_up_heap : heapsort (typ, 1, n);
    END;
  END   { TSortierfeld.sortieren };


FUNCTION TSortierfeld.count : INTEGER;

BEGIN { TSortierfeld.count }
  count := n;
END   { TSortierfeld.count };


PROCEDURE TSortierfeld.show (i : INTEGER);

BEGIN { TSortierfeld.show }
  IF i <= n THEN Tfeld(a^)[i]^.show
END   { TSortierfeld.show };


DESTRUCTOR TSortierfeld.done;

VAR idx : INTEGER;

  BEGIN { TSortierfeld.done }
    IF a <> NIL
    THEN FreeMem (a, SizeOf(Pointer) * n);
```

```
  END   { TSortierfeld.done };

{ ********************************************************** }

END.
```

Die implementierten Sortierverfahren zeigen unterschiedliches **Laufzeitverhalten**, sind andererseits aber auch unterschiedlich komplex bezüglich ihres im Programmcode niedergelegten Vorgehens. Das Laufzeitverhalten wird hierbei in Abhängigkeit von der Anzahl n der zu sortierenden Elemente gemessen, und zwar wird als Maßstab die Anzahl der Vergleiche von Elementen genommen, die während des Sortiervorgangs ausgeführt werden.

Eine wichtige Randbedingung aller in den folgenden Unterkapiteln behandelten Methoden ist die Forderung nach einem **internen Sortierverfahren**, d.h. die Sortierung soll innerhalb des Sortierfelds erfolgen, eventuell unter zu Hilfenahme weiterer Variablen, deren Anzahl jedoch konstant ist und keinesfalls in der Größenordnung n der zu sortierenden Elemente liegt. Daher scheiden Verfahren wie der Mergesort ([O/W]) aus.

Untere Schranken für das worst-case-Verhalten von Sortieralgorithmen **und das mittlere Laufzeitverhalten** sind aus der Literatur bekannt ([AHU], [KN2], [O/W]):

Jeder Algorithmus, der zur Sortierung von Elementen auf Vergleichen beruht, benötigt zur Sortierung von n Elementen im ungünstigsten Fall mindestens $C \cdot n \cdot \log(n)$ viele Vergleiche (mit einer Konstanten $C > 0$).

Jede Permutation der n zu sortierenden Elemente erscheine mit gleicher Wahrscheinlichkeit als Eingabe eines Sortieralgorithmus. Dann benötigt dieser im Mittel mindestens $C \cdot n \cdot \log(n)$ viele Vergleiche (mit einer Konstanten $C > 0$).

Obere Schranken für das Laufzeitverhalten werden durch die angegebenen Algorithmen festgelegt.

Bei der Beschreibung der Verfahren in den folgenden Unterkapiteln wird wiederholt die Formulierung verwendet, dass *das Sortierfeld (in einer dort angegebenen Reihenfolge) durchlaufen werde*. Gemeint ist damit, dass die Verweise im Sortierfeld (d.h. die `ARRAY`-Einträge `Tfeld(a^)[1]`, ..., `Tfeld(a^)[`n`]`) durchlaufen werden; meist werden dabei die Elemente `Tfeld(a^)[`i`]^` vom Objekttyp `TSortentry` mit anderen Elementen mit Hilfe der Methoden `TSortentry.kleiner`, `TSortentry.gleich` bzw. `TSortentry.groesser` verglichen. Entsprechend bedeutet die Formulierung, *ein Element werde an die Position i im Sortierfeld gebracht*, dass im Sortierfeld an Position i, d.h. in `Tfeld(a^)[`i`]`, ein Verweis auf das Objekt mit dem Element eingetragen wird.

6.1.1 Bubblesort

Der **Bubblesort** ist ein sehr einfaches Sortierverfahren. Das Sortierfeld wird mehrmals sequentiell durchlaufen. Beim ersten Durchlauf wird das größte Element an die höchste Position im Sortierfeld geschoben. Im zweiten Durchlauf wird das größte Element unter den verbleibenden Elementen an die zweithöchste Position geschoben usw. Das Verfahren ist in der Methode `TSortierfeld.bubblesort` (im Implementierungsteil der `UNIT Sortiere`) niedergelegt. Die Parameter `lower` und `upper` bestimmen den Abschnitt im Sortierfeld, der durchmustert werden soll.

```
PROCEDURE TSortierfeld.bubblesort (lower : INTEGER;
                                   upper : INTEGER);

VAR  idx : INTEGER;
     j   : INTEGER;

BEGIN { TSortierfeld.bubblesort }
  idx := upper;

  { Tfeld(a^)[idx+1]^, ..., Tfeld(a^)[upper]^ ist aufsteigend
    sortiert                                                  }
  WHILE idx > lower DO
    BEGIN
      FOR j := lower TO idx - 1  DO
        BEGIN
          IF Tfeld(a^)[j]^.groesser (Tfeld(a^)[idx]^)
          THEN { austauschen } exchange (idx, j);
        END;
      idx := idx - 1;
    END { WHILE }

END { TSortierfeld.bubblesort };
```

Ist der Parameter `lower` = l und der Parameter `upper` = u, so werden $u-l$ Durchläufe der `WHILE`–Schleife durchgeführt. Im i-ten Durchlauf hat die Variable `idx` den Wert `idx` = $u-i+1$. Im i-ten durchläuft die Variable `j` die Werte l, ..., $u-i$, d.h. im i-ten Durchlauf werden $u-i-l+1$ viele Vergleiche ausgeführt. Der Bubblesort wird mit den Parametern `lower` = 1 und `upper` = n gestartet, so dass die Gesamtzahl der Vergleiche

$$\sum_{i=1}^{n-1}(n-i)=\sum_{i=1}^{n-1}i=\frac{n\cdot(n-1)}{2}$$

beträgt, also von der Ordnung $O(n^2)$ ist.

Auch im Mittel ergibt sich diese Größenordnung der Laufzeit, da die Anzahl der Vergleiche nicht von den Primärschlüsselwerten der Elemente abhängt, die zu sortieren sind. Der Bubblesort ist also weit von der optimalen Worst-case-Schranke und der Schranke für die mittlere Laufzeit entfernt und daher nur zur Sortierung weniger Elemente geeignet.

6.1.2 Quicksort

Der **Quicksort** realisert die Idee der Divide-and-Conquer-Methode (siehe Kapitel 7.1).

Besteht die Folge $x = \langle a_1, ..., a_n \rangle$ der zu sortierenden Elemente nur aus einem einzigen Element, dann ist nichts zu tun. Sind mehr als nur ein Element zu sortieren, dann wird ein Element $a = (key, info)$ zufällig ausgewählt und an diejenige Position verschoben, an der es in der endgültigen Sortierung stehen wird. Diese Position wird dadurch bestimmt, dass die Folge $x = \langle a_1, ..., a_n \rangle$ in drei Teilfolgen $\langle b_1, ..., b_{m-1} \rangle$, $\langle a \rangle$ und $\langle b_{m+1}, ..., b_n \rangle$ aufgeteilt wird, die alle aus den ursprünglichen Elementen $a_1, ..., a_n$ in einer eventuell anderen Reihenfolge bestehen. Das Element *a* wird also an die Position *m* gebracht. Dabei gilt:

$b_i < a$ für $i = 1, ..., m - 1$ und $b_i \geq a$ für $i = m + 1, ..., 1$.

Über die Größenverhältnisse der Elemente $b_1, ..., b_{m-1}$ und $b_{m+1}, ..., b_n$ untereinander wird nichts gesagt. Das Problem der Sortierung der erzeugten kleineren Teilfolgen $\langle b_1, ..., b_{m-1} \rangle$ und $\langle b_{m+1}, ..., b_n \rangle$ wird (rekursiv) nach dem gleichen Prinzip gelöst. Zu beachten ist dabei, dass die Sortierung einer Teilfolge stets innerhalb des Folgenabschnitts erfolgt und nicht auf Teilfolgen zugreift, die außerhalb liegen.

Das Verfahren ist in der Methode `TSortierfeld.quicksort` (im Implementierungsteil der `UNIT Sortiere`) niedergelegt. Dabei wird vorausgesetzt, dass die zu sortierenden Elemente bereits an das Sortierfeld in der in Kapitel 6.1 beschriebenen Weise übergeben wurden. Außerdem wurde der eingesetzte Zufallszahlengenerator bereits initialisiert. Die Parameter `lower` und `upper` bestimmen den Abschnitt im Sortierfeld, der durchmustert werden soll.

```
PROCEDURE TSortierfeld.quicksort (lower : INTEGER;
                                  upper : INTEGER);

VAR  t           : PSortentry;
     m           : INTEGER;
     idx         : INTEGER;
     zufalls_idx : INTEGER;
```

```
BEGIN { TSortierfeld.quicksort }
  IF lower < upper
  THEN BEGIN
         { ein Feldelement zufällig aussuchen und mit
           Tfeld(a^)[lower]^ austauschen                      }
         zufalls_idx
             := lower + Trunc(Random * (upper - lower + 1));
         exchange (lower, zufalls_idx);

         t := Tfeld(a^)[lower];
         m := lower;

         FOR idx := lower + 1 TO upper DO
         { es gilt:
           Tfeld(a^)[lower+1]^ , ..., Tfeld(a^)[m]^ < t^
           und
           Tfeld(a^)[m+1]^, ... Tfeld(a^)[idx-1]^ >= t^       }
           IF Tfeld(a^)[idx]^.kleiner (t^)
           THEN BEGIN
                  m := m+1;
                  { vertauschen von Tfeld(a^)[m]^
                    und Tfeld(a^)[idx]^                       }
                  exchange (m, idx);
                END;

           { Tfeld(a^)[lower]^ und Tfeld(a^)[m]^ vertauschen }
           exchange (lower, m);

           { es gilt:
             Tfeld(a^)[lower]^, ...,
                                Tfeld(a^)[m-1]^ < Tfeld(a^)[m]^
             und
             Tfeld(a^)[m]^ <= Tfeld(a^)[m+1]^, ...,
                                Tfeld(a^)[upper]^              }

           quicksort (lower, m-1);
           quicksort (m+1, upper);

       END { IF}

END { TSortierfeld.quicksort };
```

Abbildung 6.1.2-1 zeigt schematisch den Ablauf des Verfahrens.

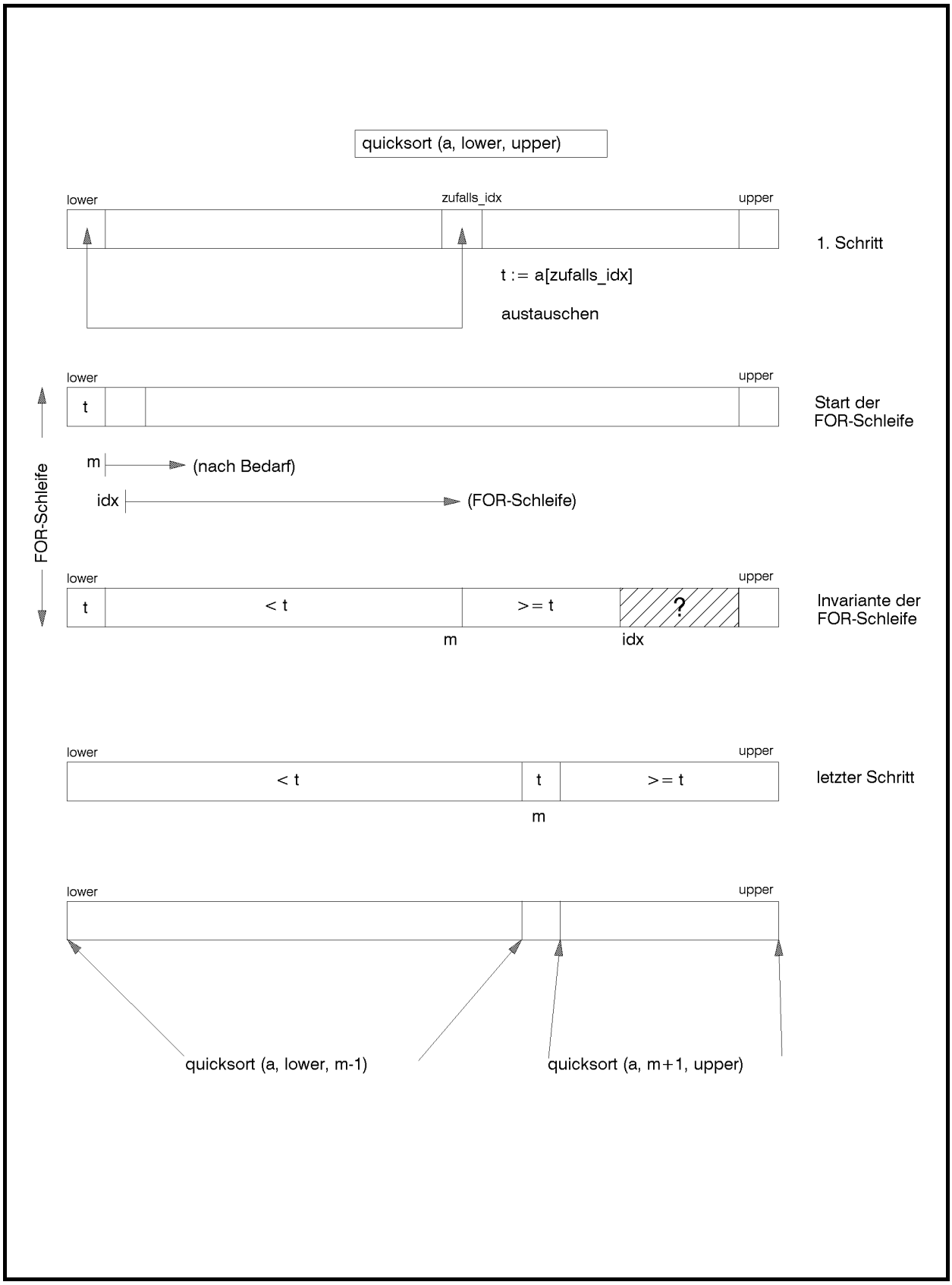


Abbildung 6.1.2-1: Quicksort

Der Algorithmus stoppt, da in jedem `quicksort`-Aufruf ein Element an die endgültige Position geschoben wird und die Anzahl der Elemente in den beiden rekursiven `quicksort`-Aufrufen in den restlichen Teilfogen zusammen um 1 verringert ist. Die Korrektheit des Algorithmus folgt aus der Invariante der `FOR`-Schleife.

Zur Ermittlung des Laufzeitverhaltens, gemessen in der Anzahl der durchgeführten Vergleiche bei Eingabe von n Elementen, wird zunächst der Spezialfall betrachtet, dass bei der Aufteilung der zu sortierenden (Teil-) Folge das Teilungselement nach der Verschiebung der übrigen Elemente stets in die Mitte der Folge gelangt, d.h. wird der Quicksort auf die Folge $x = \langle a_1, ..., a_n \rangle$ angewendet und entstehen dabei die drei Teilfolgen $\langle b_1, ..., b_{m-1} \rangle$, $\langle a \rangle$ und $\langle b_{m+1}, ..., b_n \rangle$, so enthalten die erste und dritte Teilfolge gleich viele Elemente. Wird der Quicksort nun rekursiv auf $\langle b_1, ..., b_{m-1} \rangle$ bzw. $\langle b_{m+1}, ..., b_n \rangle$ angewendet, so soll auch dort wieder das zufällig ausgewählte Teilungselement genau in die Mitte kommen. Diese Annahme soll für alle rekursiv untergeordneten Sortiervorgänge gelten. Dazu wird weiterhin angenommen, dass n die Form $n = 2^k - 1$ besitzt. Dann gilt für die Anzahl $QS(n)$ der Anzahl der Vergleiche zur Sortierung von n Elementen mit Quicksort:

$QS(1) = 0$ und

$$\begin{aligned} QS(n) &= QS\left(2^k - 1\right) \\ &= 2 \cdot QS\left({(n-1)}/{2} \right) + n - 1 \\ &= 2 \cdot QS\left(2^{k-1} - 1\right) + 2^k - 2. \end{aligned}$$

In allgemeiner Form ergibt sich damit

$$QS\left(2^k - 1\right) = 2^l \cdot QS\left(2^{k-l} - 1\right) + l \cdot 2^k - \sum_{i=1}^{l} 2^i .$$

Geht man bis $l = k - 1$, so ist hiermit $2^{k-l} - 1 = 1$, d.h. mit diesem Wert für l ist $QS\left(2^{k-l} - 1\right) = QS(1) = 0$. Damit ergibt sich

$$\begin{aligned} QS\left(2^k - 1\right) &= (k-1) \cdot 2^k - \sum_{i=1}^{k-1} 2^i \\ &= (k-1) \cdot 2^k - 2^k - 2, \end{aligned}$$

also $QS(n) \in O\left(n \cdot \log(n)\right)$ und daher optimales Laufzeitverhalten.

Dieser Spezialfall kann als „günstige“ Auswahl der Teilungselemente angesehen werden. Im allgemeinen kann man nicht erwarten, dass bei der Auswahl der Teilungselemente durchgängig zwei gleichlange Teilfolgen entstehen, da das Teilungselement genau in die Mitte zwischen beide Teilfolgen kommt. Die allgemeinere Abschätzung von $QS(n)$ lautet:

$QS(0) = QS(1) = 0$ und

$$\begin{aligned} QS(n) &\le n-1+\max\{QS(n_1)+QS(n_2)\,|\,n_1+n_2=n-1\} \\ &= n-1+\max\{QS(n_1)+QS(n-1-n_1)\,|\,n_1=0,\ldots,n-1\}. \end{aligned}$$

Diese Abschätzung liefert

$QS(n) \le \sum_{i=0}^{n-1} i$, also $QS(n) \in O(n^2)$.

Man kann sich leicht davon überzeugen, dass diese Anzahl an Vergleichen auch erreicht wird, nämlich dann, wenn bei der Aufteilung einer (Teil-) Folge stets eine der beiden entstehenden Teilfolgen leer ist, und die andere Teilfolge alle verbleibenden Elemente enthält.

Im Mittel verhält sich Quicksort weitaus günstiger:

Die mittlere Anzahl an Vergleichen zur Sortierung einer Zahlenfolge aus *n* Elementen ist von der Ordnung $O(n \cdot \log(n))$.

Der Quicksort verhält sich also im Mittel optimal. Dass das Verfahren mit der Methode `TSortierfeld.quicksort` in der Praxis ein Laufzeitverhalten zeigt, das fast alle anderen Sortierverfahren schlägt, liegt daran, dass die Wahrscheinlichkeit, eine „ungünstige“ Folge von Teilungselementen im Quicksort zu bekommen, äußerst gering ist gegenüber der Wahrscheinlichkeit, eine „günstige“ Folge von Teilungselementen auszuwählen. Das Laufzeitverhalten in der vorgestellten Implementierung tendiert durch die zufällige Auswahl des Teilungselements eher zum mittleren Laufzeitverhalten des Quicksorts.

6.1.3 Heapsort

Ein Sortierverfahren, das im schlechtesten Fall und im Mittel die minimale Zeitkomplexität der Ordnung $O(n \cdot \log(n))$ hat, ist die Sortierung gemäß **Heapsort**. Dieses Verfahren beruht auf der Manipulation einer für das Sortierproblem geeigneten Datenstruktur, einem Sortierfeld bzw. Binärbaum mit „Heap-Eigenschaft”.

Im vorliegenden Kapitel werden zwei Varianten des Heapsorts behandelt, nämlich der „klassische“ Heapsort, der theoretisch dem Quicksort in der Laufzeit überlegen ist, jedoch nicht in den praktischen Implementierungen, und der Bottom-up-Heapsort. Die mangelnde Überlegenheit des klassischen Heapsorts über den Quicksort (im worst-case-Verhalten) lässt sich mit den Argumenten am Ende von Kapitel 6.1.2 gegründen. Das mittlere Laufzeitverhalten, zu dem der Quicksort in der vorgestellten Implementierung tendiert, und das worst-case-Laufzeitverhalten des klassischen Heapsorts sind von der gleichen Größenordnung $O(n \cdot \log(n))$, jedoch sind hierbei die beteiligten Konstanten sehr unterschiedlich. Dieses Manko wird im Bottom-up-Heapsort auszugleichen versucht. Der Bottom-up-Heapsort als ei-

ne Variante des klassischen Heapsorts versucht hierbei einige Vorteile der zugrundeliegenden Datenstruktur optimal zu nutzen und verzichtet auf eine rekursive Implementierung des Verfahrens. Praktische Versuche zeigen, dass der Bottom-up-Heapsort nicht nur beweisbar theoretisch dem Quicksort überlegen ist, sondern auch praktisch (jedenfalls bei großen Eingabefolgen).

Zum Verständnis des Heapsorts sind einige grundsätzliche Bemerkungen angebracht:

Die zu sortierende Eingabefolge $\langle a_1, ..., a_n \rangle$ kann in einem `ARRAY` abgelegt bzw. wie in der vorliegenden Implementierung über Verweise angesprochen werden, die in einem `ARRAY` realisiert sind, hier im Sortierfeld `Tfeld(a^)`. Man kann die zu sortierende Folge auch als Knotenmarkierungen eines vollständigen Binärbaums betrachten. Ein **vollständiger Binärbaum** ist ein Binärbaum, dessen Niveaus vollständig gefüllt sind, eventuell mit Ausnahme des höchsten Niveaus; in diesem Fall sind alle Knoten (auf dem höchsten Niveau) möglichst weit links angeordnet. Man numeriert die Knoten des vollständigen Binärbaums beginnend bei der Wurzel nach aufsteigenden Niveaus und auf jedem Niveau von links nach rechts (diese Numerierung entspricht dem Durchlaufen des Binärbaums in Breitensuche, siehe Kapitel 5.2.2). Das Element a_i wird dann entweder im Feldelement mit Index i abgelegt bzw. über `Tfeld(a^)[` i`]` adressiert, oder es befindet sich als Knotenmarkierung im Knoten mit der Nummer i. Abbildung 6.1.3-1 zeigt ein Beispiel mit 10 Elementen.

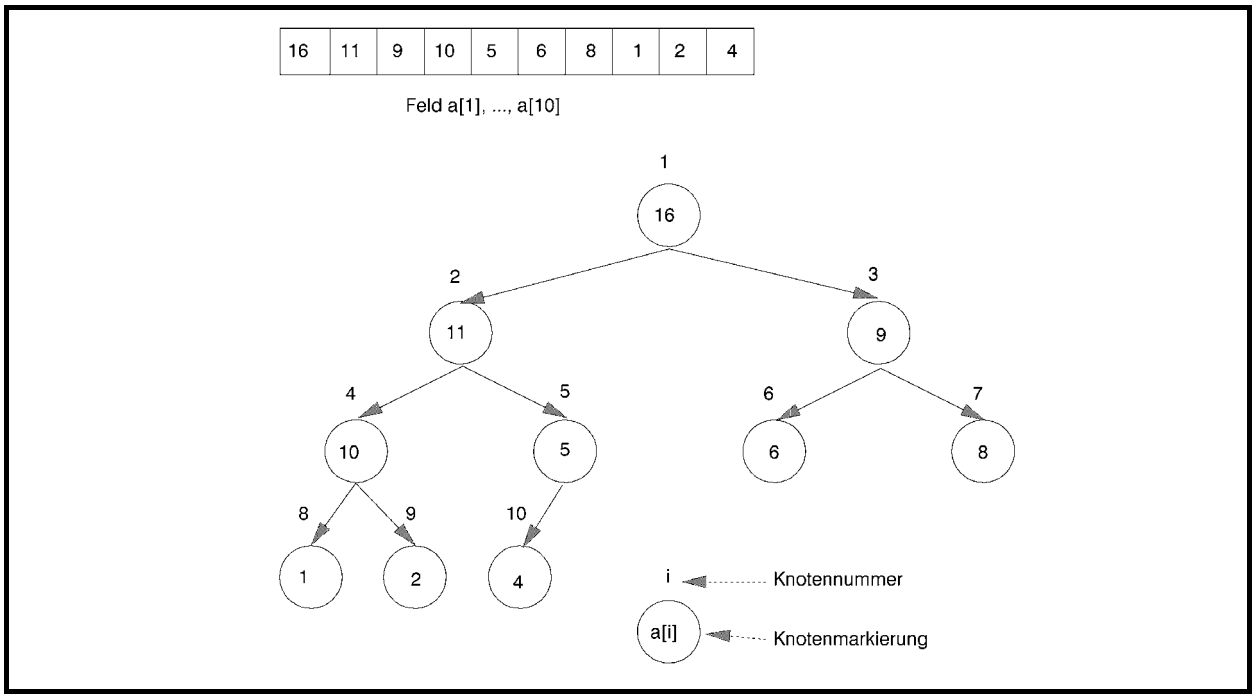


Abbildung 6.1.3-1: Korrespondenz zwischen einem `ARRAY` und einem vollständigen Binärbaum

Es besteht eine bijektive Beziehung zwischen den Indizes des `ARRAY`s und den so erzeugten Knotennummer im Binärbaum:

Der Binärbaum habe die Höhe *h*. Das maximale Niveau ist dann $m = h - 1$.

Jeder Index *i* im `ARRAY` mit $1 \le i \le n$ hat die Form $i = 2^j + l$ mit $0 \le j \le m$ und $0 \le l < 2^j$. Dann befindet sich das Element a_i im Knoten mit der Nummer *i*, der auf dem Niveau *j* und dort genau *l* Positionen von links (Zählung beginnend bei 0) steht.

Aus dem Index eines Elements im `ARRAY` lässt sich also bijektiv umkehrbar das Niveau und die Position des Knotens innerhalb dieses Niveaus im Binärbaum bestimmen, der mit dem Element markiert ist. Daher kann die Vorgehensweise des Heapsorts entweder durch seine Wirkung auf das `ARRAY` oder auf den Binärbaum beschrieben werden. Die erste Beschreibungsweise eignet sich besser für eine algorithmische Umsetzung, die zweite für eine Visualisierung.

Hat ein Knoten im Binärbaum die Nummer $i = 2^j + l$ mit $0 \le j \le m$ und $0 \le l < 2^j$ d.h. befindet er sich auf Niveau *j* an Position *l* von links, und hat er zwei direkte Nachfolger, so stehen diese auf Niveau $j+1$ und dort an den Positionen (Zählung beginnend bei 0) $2 \cdot l$ und $2 \cdot l + 1$. Daher haben die beiden Nachfolger die Nummern (und damit die Indizes im `ARRAY`) $2^{j+1} + 2 \cdot l = 2 \cdot i$ und $2^{j+1} + 2 \cdot l + 1 = 2 \cdot i + 1$.

Die Folge $\langle a_1, ..., a_n \rangle$ erfüllt die **Heap-Eigenschaft**, wenn gilt:
$a_i \ge a_{2i}$ für $1 \le i \le n/2$ und $a_i \ge a_{2i+1}$ für $1 \le i < n/2$.

Wird auf die Folge über ein Sortierfeld vom Datentyp `TSortierfeld` zugegriffen, so bedeutet die Heap-Eigenschaft

```
Tfeld(a^)[i]^.kleiner(Tfeld(a^)[2*i]^) = FALSE
```
und
```
Tfeld(a^)[i]^.kleiner(Tfeld(a^)[2*i + 1]^) = FALSE.
```

Im zugehörigen Binärbaum bedeutet die Heap-Eigenschaft, dass die Beschriftung eines Knotens (bis zum zweithöchsten Niveau) größer als die Beschriftungen der beiden Nachfolger bzw. des direkten Nachfolgers ist.

Der Heapsort arbeitet folgendermaßen:

Geht man davon aus, dass die zu sortierende Eingabefolge $\langle a_1, ..., a_n \rangle$ die Heap-Eigenschaft erfüllt und wird auf die Folge über ein Sortierfeld zugegriffen (gedanklich über einen vollständigen Binärbaum), so steht das in der Sortierung größte Element im Objekt `Tfeld(a^)[1]^` (bzw. in der Wurzel des Baums). Der Heapsort bringt dieses Element an die letzte Position der Folge, d.h. er tauscht `Tfeld(a^)[1]^` mit `Tfeld(a^)[n]^` aus (bzw.

vertauscht die Markierung der Wurzel mit der Markierung des Blattes mit der höchsten Nummer). Ab sofort wird das nun *n*-te Folgenglied nicht mehr berücksichtigt, da es an der korrekten Position steht (im Binärbaum wird das Blatt „abgeschnitten“). Durch den Austausch ist eventuell an Position 1 (bzw. an der Baumwurzel) die Heap-Eigenschaft verletzt. Durch Aufruf einer Prozedur `reconstruct_heap` im klassischen Heapsort bzw. `reconstruct_bottomup_heap` im Bottom-up-Heapsort wird die Heap-Eigenschaft im Feld wieder hergestellt. Durch Austausch des ersten Elements mit dem Element an der höchsten berücksichtigten Position (bzw. der Markierung der Wurzel mit der Markierung im Blatt mit der nun höchsten Nummer) wird das zweitgrößte Element der Eingabefolge an die in der Sortierfolge korrekte Position geschoben und dann von der weiteren Betrachtung ausgeschlossen (aus dem Binärbaum „abgeschnitten“). Das Verfahren wird wiederholt, bis nur noch ein zu berücksichtigendes Element (im Binärbaum die Wurzel) übrig bleibt.

Im allgemeinen muss man davon ausgehen, dass die ursprüngliche Eingabefolge die Heap-Eigenschaft nicht erfüllt. Daher wird die Eingabefolge zunächst so umsortiert, dass sie zum ersten Mal die Heap-Eigenschaft erfüllt. Der Vorgang lässt sich am besten im Bild des Binärbaums erklären: Beginnend beim Knoten mit der höchsten Nummer im höchsten komplett ausgefüllte Niveau wird in Richtung absteigender Knotennumerierung an jedem Knoten die Heap-Eigenschaft hergestellt, bis die Wurzel erreicht ist. Dazu werden die Prozeduren `reconstruct_heap` im klassischen Heapsort bzw. `reconstruct_bottomup_heap` im Bottom-up-Heapsort verwendet.

Der gesamte Ablauf ist in Abbildung 6.1.3-2 zusammengefasst.

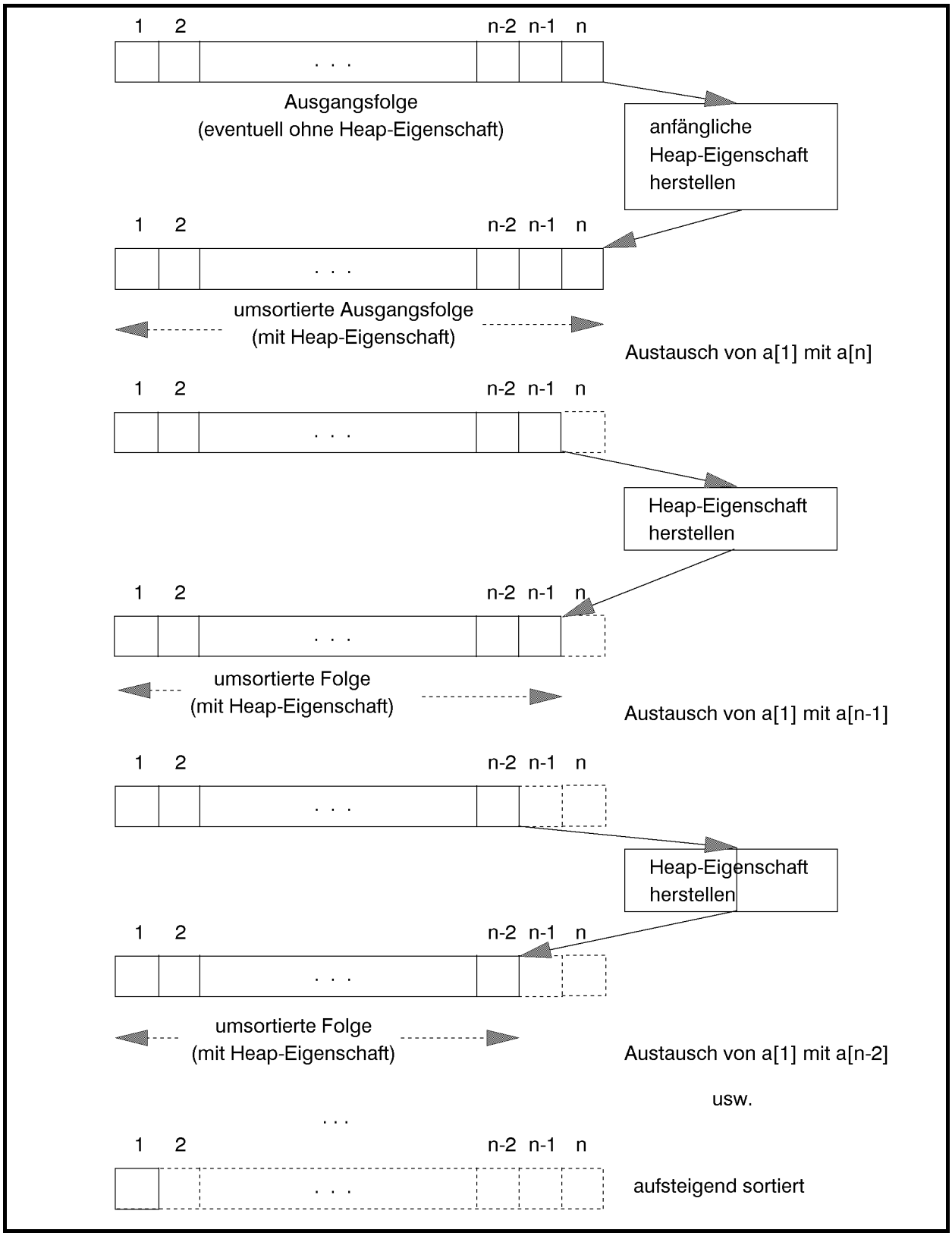


Abbildung 6.1.3-2: Heapsort

Das Verfahren ist in der Methode `TSortierfeld.heapsort` (im Implementierungsteil der `UNIT Sortiere`) niedergelegt. Dabei wird vorausgesetzt, dass die zu sortierenden Elemente bereits an das Sortierfeld in der in Kapitel 6.1 beschriebenen Weise übergeben wurden. Die Parameter `lower` und `upper` bestimmen den Abschnitt im Sortierfeld, der durchmustert wer-

den soll. Der Parameter `typ` unterscheidet den klassischen Heapsort vom Bottom-up-Heapsort.

```
PROCEDURE TSortierfeld.heapsort (typ   : TSortiere;
                                 lower : INTEGER;
                                 upper : INTEGER);

VAR idx         : INTEGER;
    reconstruct : Trec_proc;


BEGIN { TSortierfeld.heapsort }

  CASE typ OF
  heap           : reconstruct := reconstruct_heap;
  bottom_up_heap : reconstruct := reconstruct_bottomup_heap;
  ELSE Exit;
  END;

  { Tfeld(a^)[lower]^ , ... Tfeld(a^)[upper]^ in einen Heap
    umwandeln; dabei wird ausgenutzt, dass die Elemente mit
    "hohen" Indizes keine Nachfolger haben                    }

  FOR idx := upper DIV 2 DOWNTO lower DO
    reconstruct (Pfeld(a), idx, upper);

  { Heap sortieren }
  idx := upper;
  WHILE idx >= 2 DO
    BEGIN
      { Tfeld(a^)[lower]^ mit Tfeld(a^)[idx]^ vertauschen }
      exchange (lower, idx);
      idx := idx - 1;
      reconstruct (Pfeld(a), lower, idx);
    END;   { WHILE }

END   { TSortierfeld.heapsort };
```

Zunächst wird der **klassische Heapsort** beschrieben. Hier wird die Prozedur

```
PROCEDURE reconstruct_heap (a : Pfeld;
                            i : INTEGER;
                            j : INTEGER);     FAR;
```

verwendet, um in dem durch `a` adressierten Sortierfeld, und zwar in dem durch die Indizes `i` und `j` bestimmten Abschnitt einen Heap zu rekonstruieren. Dabei wird vorausgesetzt, dass in

diesem Abschnitt die Heap-Eigenschaft besteht, mit Ausnahme eventuell an der Position, die durch den Index `i` bestimmt ist. Das Vorgehen wird wieder anhand des Bilds des Binärbaums beschrieben:

Zunächst wird geprüft, ob der durch `i` bestimmte Knoten kein Blatt ist; nur in diesem Fall ist eventuell etwas zu tun. Dann wird geprüft, ob er zwei Nachfolger besitzt. In die Variable `idx` wird die Nummer des Knotens mit der größeren Markierung gesetzt. Falls die Heap-Eigenschaft mit dem durch `idx` angesprochenen Knoten verletzt ist, findet ein Austausch mit dieser Knotenmarkierung statt. Die ursprüngliche Markierung des Knotens mit der Nummer `i` fällt also auf das nächst höhere Niveau. Durch den Austausch kann natürlich an der Austauschstelle eine Verletzung der Heap-Eigenschaft eingetreten sein, so dass an dieser Stelle mit `reconstruct_heap` rekursiv „repariert“ wird. Auf diese Weise kann die anfängliche Markierung des Knotens mit der Nummer `i` bis in eine Blattposition sinken.

```
PROCEDURE reconstruct_heap (a : Pfeld;
                            i : INTEGER;
                            j : INTEGER);     FAR;
    { Die Prozedur ordnet die Elemente im Feld
      a^[i], ..., a^[j] so um, dass anschließend die
      Heap-Eigenschaft gilt:
      a^[k].key >= a^[2k].key   für i <= k <= j/2 und
      a^[k].key >= a^[2k+1].key für i <= k <  j/2             }

    VAR idx : INTEGER;
        ptr : Psortentry;

    BEGIN { reconstruct_heap }
      IF 2*i <= j    { d.h. a^[i] ist kein Blatt }
      THEN BEGIN
             IF 2*i + 1 <= j
             THEN BEGIN
                    IF a^[2*i]^.kleiner(a^[2*i + 1]^)
                    THEN idx := 2*i+1
                    ELSE idx := 2*i;
                  END
             ELSE idx := 2*i;
             IF a^[i]^.kleiner(a^[idx]^)
             THEN BEGIN
                    { a^[i]^ mit a^[idx] vertauschen }
                    ptr     := a^[idx];
                    a^[idx] := a^[i];
                    a^[i]   := ptr;
                    { die Heap-Eigenschaft im Teilbaum
                      wiederherstellen                      }
                    reconstruct_heap (a, idx, j);
                  END
```

```
        END
    END { reconstruct_heap } ;
```

Zur Abschätzung der Anzahl von Elementvergleichen bei Eingabe von n zu sortierenden Elementen werde $n = 2^m + 2^m - 1 = 2^{m+1} - 1$ angenommen, d.h. der Binärbaum ist auch auf dem höchsten Niveau m vollständig ausgefüllt. Der Heapsort wird in der Form

```
heapsort (heap, 1, n)
```

aufgerufen. Daher erfolgen zur anfänglichen Herstellung der Heap-Eigenschaft `reconstruct_heap`-Aufrufe, in denen der Formalparameter `i` den Wert i mit $i = 2^j + l$, $0 \le j < m$ und $0 \le l < 2^j$ hat und der Formalparameter `j` gleich n ist. Eine Knotenmarkierung (bei `i` = i) sinkt dabei maximal vom Niveau j auf das Niveau m. Bei jedem Niveauwechsel erfolgen maximal zwei Elementvergleiche. Daher werden für einen Knoten auf dem Niveau j maximal $2 \cdot (m - j)$ Elementvergleiche ausgeführt. Auf dem Niveau j stehen 2^j viele Knoten. Die Anzahl der Elementvergleiche zur anfänglichen Herstellung der Heap-Eigenschaft ist also beschränkt durch

$$\sum_{j=0}^{m-1} 2^j \cdot 2 \cdot (m - j).$$

Durch Anwendung der Gleichungen $\sum_{j=0}^{k} j \cdot 2^j = (k-1) \cdot 2^{k+1} + 2$ und $\sum_{j=0}^{k} 2^j = 2^{k+1} - 1$ ergibt sich

$$\sum_{j=0}^{m-1} 2^j \cdot 2 \cdot (m - j) = 2 \cdot (n - \log_2(n+1)),$$

also ein Wert von der Ordnung $O(n + \log(n))$.

Anschließend wird jeweils das Element, das die Wurzel des Binärbaums markiert, mit der Markierung des Knotens mit der gegenwärtig größten Nummer ausgetauscht. Diese Markierung befindet sich auf dem Niveau j mit $1 \le j \le m$, dem zur Zeit höchsten Niveau. Durch `reconstruct_heap`-Aufrufe wird diese Knotenmarkierung eventuell wieder bis zum Niveau j transportiert. Bei jedem in dieser Transportphase durchgeführten Niveauwechsel werden maximal zwei Elementvergleiche ausgeführt. Insgesamt sind dieses für eine transportierte Knotenmarkierung höchstens $2 \cdot j$ viele Elementvergleiche. Da das j-te Niveau 2^j viele Knoten enthält, ergibt sich eine obere Schranke für die Gesamtanzahl der Elementvergleiche zu

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

also von der Ordnung $O(n \cdot \log(n))$.

Der klassische Heapsort führt also im ungünstigsten Fall eine Anzahl von Elementvergleichen der Ordnung $O(n \cdot \log(n))$ aus.

Die mittlere Anzahl von Elementvergleichen lässt sich ebenfalls durch die Größenordnung $O(n \cdot \log(n))$ abschätzen: In der Heap-Sortierphase wird eine Knotenmarkierung zunächst von Niveau j in die Wurzel gebracht und kommt anschließend in einen „Zielknoten", der auf einem Niveau t mit $0 \le t \le j$ steht. In dieser Situation gibt es $2^j - 1 + l$ mit $0 \le l < 2^j$ viele mögliche Zielknoten ($2^j - 1$ Knoten bis zum Niveau $j-1$ und l viele Knoten auf Niveau j). Man nimmt man, dass die Wahrscheinlichkeit, in einem der in Frage kommenden Knoten zu landen, für alle Knoten gleich ist, d.h. sie beträgt $p = 1/(2^j - 1 + l) \le 1/(2^j - 1)$. Die mittlere Anzahl von Elementvergleichen ist dann beschränkt durch

$$\sum_{t=1}^{j} p \cdot 2t \cdot 2^t + 2p \le 2 \cdot \frac{(j-1)2^{j+1} + 3}{2^j - 1}.$$

Dieser Ausdruck ist von der Ordnung $O(j)$ bzw. $O(\log(n))$. Damit ergibt sich die angegebene Abschätzung für den mittleren Aufwand.

Der **Bottom-up-Heapsort** versucht, die charakteristischen Eigenschaften eines vollständigen Binärbaums mit Heap-Eigenschaft noch besser zu nutzen. Anstelle der `reconstruct_heap`-Prozedur wird dazu die `reconstruct_bottomup_heap`-Prozedur verwendet:

```
PROCEDURE reconstruct_bottomup_heap (a : Pfeld;
                                     i : INTEGER;
                                     j : INTEGER); FAR;

    { Die Prozedur ordnet die Elemente im Feld
      a^[i], ..., a^[j] so um, dass anschließend die
      Heap-Eigenschaft gilt:
      a^[k].key >= a^[2k].key   für i <= k <= j/2 und
      a^[k].key >= a^[2k+1].key für i <= k <  j/2            }

    VAR idx : INTEGER;
        k   : INTEGER;
        x   : Psortentry;

    FUNCTION bin (z : INTEGER) : INTEGER;
    { die Funktion liefert die Länge der Binärdarstellung der
      Zahl z                                                 }

      VAR i : INTEGER;

      BEGIN { bin }
        i := 0;
        WHILE  z <> 0 DO
        BEGIN
          i := i + 1;
          z := z SHR 1
```

```
      END { WHILE };
      bin := i
    END   { bin };

  BEGIN { reconstruct_bottomup_heap }

    { suche das spezielle Blatt }
    idx := i;
    WHILE 2*idx < j DO
    BEGIN
      IF NOT a^[2*idx]^.kleiner(a^[2*idx + 1]^)
      THEN idx := 2*idx
      ELSE idx := 2*idx + 1
    END;
    IF 2*idx = j THEN idx := j;

    { idx ist der Index des speziellen Blatts }

    WHILE (i < idx) AND NOT a^[i]^.kleiner (a^[idx]^) DO
       idx := idx SHR 1;

    { zyklischer Tausch auf dem speziellen Weg }
    x := a^[i];
    FOR k := bin(idx)-bin(i)-1 DOWNTO 0 DO
      a^[idx SHR (k+1)] := a^[idx SHR k];
    a^[idx] := x;

  END { reconstruct_bottomup_heap } ;
```

Die Prozedur `reconstruct_bottomup_heap` verfolgt folgende Strategie (Abbildung 6.1.3-3), hier nur für die Sortierphase des Heaps beschrieben, wenn also bereits die Heap-Eigenschaft zum ersten Mal hergestellt worden ist:

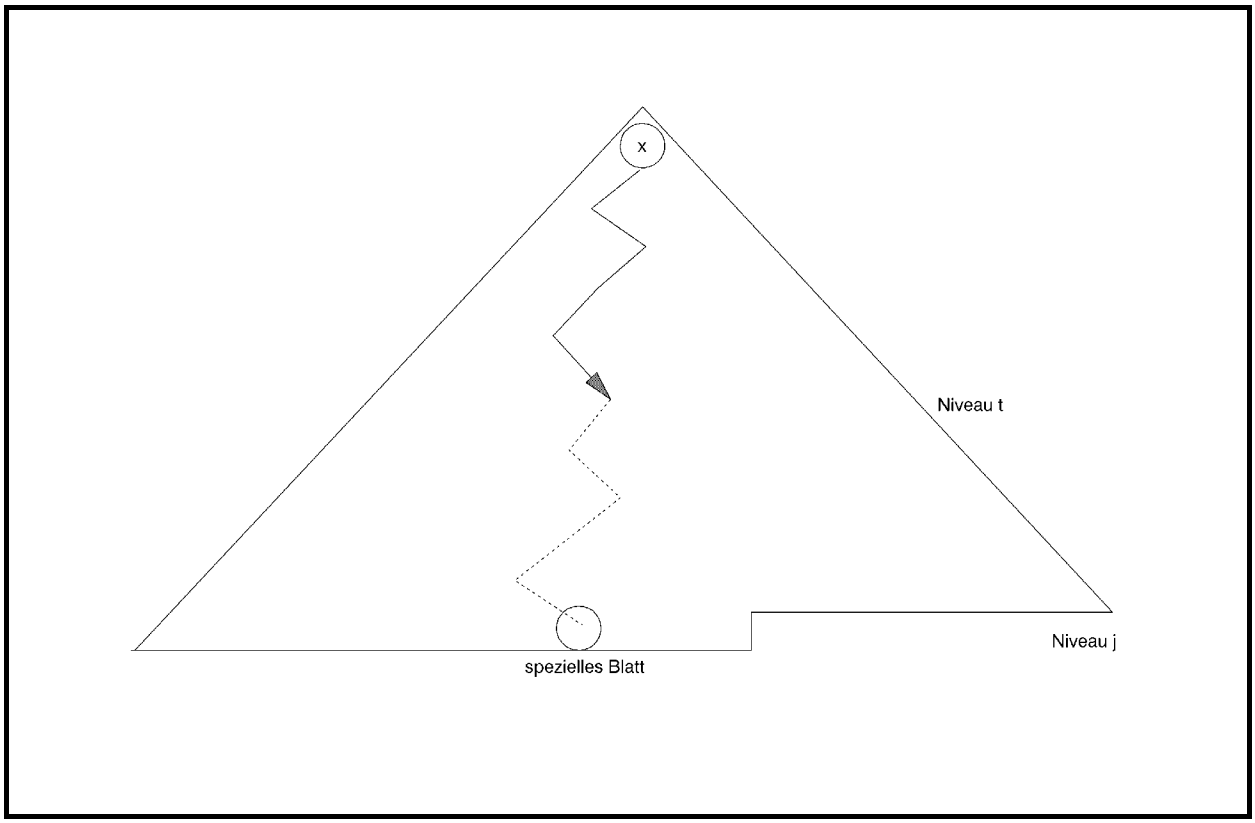


Abbildung 6.1.3-3: `reconstruct_bottomup_heap`

Nachdem ein Element *x* von Niveau *j* an die Wurzel des Baum gebracht wurde, wird es durch `reconstruct_heap` im klassischen Heapsort wieder auf ein Niveau *t* mit $0 \le t \le j$ transportiert. Bei jedem Transportschritt werden dabei zwei Elementvergleiche durchgeführt. Die Prozedur `reconstruct_bottomup_heap` sucht zunächst im noch zu sortierenden Teilbaum ausgehend von der Wurzel und jeweils übergehend zum Nachfolger mit der größeren Knotenmarkierung den als spezielles Blatt bezeichneten Knoten (der also im noch zu sortierenden Teilbaum keinen Nachfolger mehr besitzt) auf Niveau *j*. Der Pfad von der Wurzel zum speziellen Blatt wird als spezieller Pfad bezeichnet. In dieser Phase wird bei jedem Übergang auf das nächst höhere Niveau ein Elementvergleich durchgeführt. Anschließend wird ausgehend vom speziellen Blatt über die jeweiligen Vorgängerknoten dem speziellen Pfad in Richtung auf die Wurzel folgend die Position k_0 gefunden, an die die Markierung der Wurzel zu transportieren ist. Auch hierbei wird jeweils ein Elementvergleich durchgeführt. Der Vorgänger eines Knotens mit der Nummer *k* auf dem speziellen Pfad trägt die Nummer $\lfloor k/2 \rfloor$. Da auf dem speziellen Pfad (bis auf die Position an der Wurzel) die Heap-Eigenschaft gilt, ist die Position k_0 dadurch gekennzeichnet, dass hier zum ersten Mal $x < a_{k_0}$ gilt. In einem zyklischen Tausch wird *x* an die Position k_0 gebracht und alle anderen Element auf dem Anfangsstück des speziellen Pfades um eine Position in Richtung auf die Wurzel verschoben.

Zu beachten ist, dass `reconstruct_bottomup_heap` aus Gründen der Effizienzsteigerung nicht-rekursiv abläuft. Außerdem bestimmen `reconstruct_bottomup_heap` und `reconstruct_heap` denselben Zielknoten für den Austausch.

In der in Abbildung 6.1.3-3 dargestellten Situation wird die Markierung an der Wurzel von `reconstruct_bottomup_heap` und `reconstruct_heap` auf das Niveau t transportiert. Die Prozedur `reconstruct_heap` benötigt dazu $2t$ viele Elementvergleiche, `reconstruct_bottomup_heap` benötigt $j + j - t$ viele Elementvergleiche und ist damit effizienter, wenn möglichst häufig $t \geq \frac{2}{3} \cdot j$ ist. Da vor dem Austausch von x an die Wurzel die Heap-Eigenschaft galt und x aus einer Blattposition im noch zu sortierenden Teilbaum stammt, ist zu erwarten, dass x auch wieder in die Nähe des Niveaus zurück transportiert wird, d.h. dass t in der Nähe von j liegt. Da auf Niveau j im noch zu sortierenden Teilbaum mehr als die Hälfte aller restlichen Elemente liegt, wird die Bedingung $t \geq \frac{2}{3} \cdot j$ häufig erfüllt sein.

Genauere Analysen zeigen, dass der Bottom-up-Heapsort in der anfänglichen Aufbauphase, die zum ersten Mal die Heap-Eigenschaft erzeugt, eine Anzahl von Elementvergleichen hat, die zwischen $n - 2\lfloor \log_2(n) \rfloor$ und $n - 2$ liegt. Bottom-up-Heapsort zeigt insgesamt ein worst-case-Verhalten und ein mittleres Verhalten der Ordnung $O(n \cdot \log(n))$, ist aber insgesamt effizienter als der klassische Heapsort.

6.2 Suchverfahren

Im vorliegenden Kapitel werden Beispiele für Suchverfahren vorgestellt. Die Grundaufgabe besteht darin festzustellen, ob ein gegebenes Element a innerhalb einer Menge von n Elementen vorkommt.

Unterliegen die Elemente keiner Ordnungsrelation oder erfolgt der Zugriff auf die Elemente, ohne eine eventuelle Ordnungsrelation nutzen zu können, müssen eventuell alle n Elemente mit a verglichen werden, um festzustellen, dass a unter den Elementen nicht vorkommt. Beispiele zur Handhabung derartiger Situationen wurden in Kapitel 5.1 und den zugehörigen Unterkapiteln im Zusammenhang mit der ***is-member***-Operation beschrieben.

Unterliegen die Elemente einer Ordnungsrelation kann man beispielsweise eine Verwaltungsstruktur in Form eines Binärbaums zum Zugriff auf die Elemente aufbauen. Ein Beispiel ist die Prioritätsschlange aus Kapitel 5.2.1 mit ihrer ***is-member***-Operation. Allerdings kann auch bei einer Prioritätsschlange nicht ausgeschlossen werden, dass das Suchen eines Elements n Elementvergleiche erfordert, nämlich dann, wenn die Prioritätsschlange zu einer linearen Liste ausartet.

In den folgenden Unterkapiteln werden zwei Situationen unterschieden:

- auf die zu durchsuchenden Elemente erfolgt ein direkter Zugriff über einen Index (**internes Suchverfahren**), und die Elemente sind bereits gemäß eines Ordnungskriteriums sortiert
- die zu durchsuchenden Elemente liegen in externen Dateien und Zugriffsinformationen auf die Elemente werden ebenfalls in externen Dateien abgelegt (**externes Suchverfahren**).

Geht man davon aus, dass extern gespeicherte Elemente und ihre Zugriffsinformationen innerhalb der Dateien physikalisch geblockt werden, ist bei einem externen Suchverfahren die Anzahl von Zugriffen auf Datenblöcke bzw. deren rechnerinterner Transfer von primärem Interesse; die Anzahl erforderlicher Elementvergleiche kommt dann noch hinzu. Bei einem internen Suchverfahren zählt allein die Anzahl von Elementvergleichen.

6.2.1 Internes Suchen mittels Binärsuche

Die zu durchsuchenden Elemente sind wieder in Objekten vom Objekttyp `TSortentry` aus der `UNIT SortElement` abgelegt (siehe Kapitel 6.1). Das Element, nach dessen Vorkommen im Sortierfeld gesucht werden soll, befindet sich ebenfalls in einem Objekt vom Objekttyp `TSortentry` und wird über einen Verweis der Suchmethode übergeben. Das Element gilt als „gefunden“, wenn unter den Elemente, die über das Sortierfeld verwaltet werden, ein Element mit gleichem Primärschlüsselwert (`key`-Komponente) vorkommt.

Der Objekttyp `TSortfeld` des Sortierfelds wird im `PUBLIC`-Teil des Interface-Teils um eine Methode `TSortfeld.is_member` ergänzt, die die Methode `TSortfeld.binsearch` aufruft, die im `PRIVATE`-Teil des Interface-Teils der Deklaration des Objekttyps `TSortfeld` hinzugefügt wird und die Suche implementiert. ***Es wird vorausgesetzt, dass die Elemente im Sortierfeld gemäß dem Sortierkriterium aufsteigend sortiert sind***. Zur Umsetzung von `TSortfeld.is_member` wird **Binärsuche** durchgeführt.

Die Schnittstelle der Methode `TSortfeld.is_member` lautet:

Methode	Bedeutung
`FUNCTION Tsortfeld.is_member (entry_ptr : Psortentry; VAR position : INTEGER): BOOLEAN;`	Die Funktion liefert den Wert `TRUE`, falls unter den Elementen des Sortierfelds ein Element mit gleichem Primärschlüsselwert wie das durch `entry_ptr` adressierte Element vorkommt, ansonsten den Wert `FALSE`. **Achtung:** Die Funktion führt eine Binärsuche durch und erwartet, dass die Elemente des Sortierfelds gemäß dem Ordnungskriterium aufsteigend sortiert sind.

Bedeutung der Parameter:

`entry_ptr`	Verweis auf das zu überprüfende Element
`position`	Index des Elements im Sortierfeld, falls es vorkommt, ansonsten = –1

Der Code der Methoden `TSortfeld.is_member` und `TSortfeld.binsearch` lautet:

```
FUNCTION TSortierfeld.is_member (entry_ptr    : PSortentry;
                                 VAR position : INTEGER):
                                                   BOOLEAN;

BEGIN { TSortierfeld.is_member }
  position := binsearch (entry_ptr, 1, n);
  IF position >= 1
  THEN is_member := TRUE
  ELSE is_member := FALSE;
END   { TSortierfeld.is_member };

FUNCTION TSortierfeld.binsearch (entry_ptr : PSortentry;
                                 lower     : INTEGER;
                                 upper     : INTEGER) : INTEGER;

VAR middle   : INTEGER;

BEGIN { TSortierfeld.binsearch }
  binsearch := -1;
  IF lower < upper
  THEN BEGIN { der Feldausschnitt
               Tfeld(a^)[lower]^ , ... Tfeld(a^)[upper]^
               enthält mindestens 2 Elemente                 }
         middle := lower + ((upper - lower + 1) DIV 2);
         IF entry_ptr^.gleich (Tfeld(a^)[middle]^)
         THEN binsearch := middle
         ELSE BEGIN
                IF entry_ptr^.kleiner (Tfeld(a^)[middle]^)
```

```
                THEN binsearch
                      := binsearch (entry_ptr, lower, middle-1)
                ELSE binsearch
                      := binsearch (entry_ptr, middle + 1, upper);
              END
        END
  ELSE BEGIN
         IF entry_ptr^.gleich (Tfeld(a^)[lower]^)
         THEN binsearch := lower;
        END;
END   { TSortierfeld.binsearch };
```

Der in der Methode `TSortierfeld.binsearch` realisierte Algorithmus beruht wieder auf der Idee der Divide-and-Conquer-Methode: Wenn die sortierte Folge $\langle a_1, ..., a_n \rangle$ der Elemente leer ist, dann ist das Element *a*, nach dem gesucht wird, in ihr nicht enthalten, und die Entscheidung lautet `binsearch := FALSE`. Andernfalls wird das mittlere Element in $\langle a_1, ..., a_n \rangle$ daraufhin untersucht, ob die jeweiligen *key*-Komponenten übereinstimmen (bei einer geraden Anzahl von Elementen wird das erste Element der zweiten Hälfte genommen). Falls dieses zutrifft, ist die Suche beendet und die Position von *a* in $\langle a_1, ..., a_n \rangle$ bestimmt. Falls dieses nicht zutrifft, liegt *a*, wenn es überhaupt in $\langle a_1, ..., a_n \rangle$ vorkommt, im vorderen Abschnitt (nämlich dann, wenn die *key*-Komponente von *a* kleiner als die *key*-Komponente des Vergleichselements ist) bzw. im hinteren Abschnitt (nämlich dann, wenn die *key*-Komponente von *a* größer als die *key*-Komponente des Vergleichselements ist). Die Entscheidung, in welchem Abschnitt weiterzusuchen ist, kann jetzt getroffen werden. Gleichzeitig wird durch diese Entscheidung die Hälfte aller potentiell noch auf Übereinstimmung mit *a* zu überprüfenden Elemente in $\langle a_1, ..., a_n \rangle$ ausgeschlossen. Im Abschnitt, der weiter zu überprüfen ist, wird nach dem gleichen Prinzip (rekursiv) verfahren. Unter Umständen muss die Suche fortgesetzt werden, bis ein noch zu überprüfender Abschnitt innerhalb $\langle a_1, ..., a_n \rangle$ nur noch ein Element enthält.

Der Algorithmus stoppt, da in jedem `binsearch`-Aufruf, der einen Abschnitt der Länge *k* untersucht (*k* = `upper - lower + 1`), höchstens ein weiterer `binsearch`-Aufruf mit $\lfloor k/2 \rfloor$ vielen Elementen erfolgt. Die Korrektheit des Algorithmus ist offensichtlich.

Es bezeichne $V(n)$ die Anzahl erforderlicher Elementvergleiche, falls *n* Element nach dem Vorhandensein eines Elements durchsucht werden sollen. Die natürliche Zahl *m* werde so gewählt, dass $2^{m-1} \le n < 2^m$ ist. Dann lässt sich $V(n)$ wie folgt abschätzen:

Für $m = 1$ ist $n = 1$ und $V(1) = 1$.

Für $m \geq 2$ und $n = 2^{m-1}$ ist $V(n) = V\left(2^{m-1}\right) \leq 2 + V\left(2^{m-1}/2\right) = 2 + V\left(2^{m-2}\right)$, im allgemeinen $V(n) = V\left(2^{m-1}\right) \leq 2 \cdot (k-1) + V\left(2^{m-k}\right)$ mit $1 \leq k \leq m$. Für $k = m$, ist
$V(n) \leq 2 \cdot (m-1) + V(1) = 2m - 1 = 2 \cdot \log_2(n) + 1$.

Für $m \geq 2$ und $n = 2^m - 1$ ist $V(n) = V\left(2^m - 1\right) \leq 2 + V\left(2^{m-1} - 1\right) \leq 2 \cdot k + V\left(2^{m-k} - 1\right)$ mit $1 \leq k \leq m-1$. Setzt man $k = m-1$, so folgt ebenfalls
$V(n) \leq 2 \cdot (m-1) + V(1) = 2m - 1 = 2 \cdot \log_2(n) + 1$

Für $2^{m-1} < n < 2^m$ ist $V\left(2^{m-1}\right) \leq V(n) \leq V\left(2^m - 1\right)$, also $V(n) \leq 2 \cdot \log_2(n) + 1$.

Die Anzahl erforderlicher Elementvergleiche, falls n Element nach dem Vorhandensein eines Elements durchsucht werden sollen, ist also im ungünstigsten Fall von der Ordnung $O(\log(n))$. Dieses (worst-case-) Verhalten ist optimal.

Im Mittel wird man gelegentlich das gesuchte Element „etwas eher“ finden, es überwiegt aber die Komplexität der Suche nach einem Element, das nicht in der zu durchsuchenden Folge vorhanden ist, so dass hier eine Komplexität im Mittel der Ordnung $O(\log(n))$ folgt.

6.2.2 Externes Suchen mittels höhenbalancierter Bäume

Im vorliegenden Kapitel wird exemplarisch ein externes Suchverfahren beschrieben. Die Beschreibung erfolgt informell, d.h. ohne Angabe entsprechenden (Pseudo-) Codes; Details können in der angegebenen Literatur nachgelesen werden.

Bei einem externen Suchverfahren sind die Elemente in Form von Dateien auf einem externen Speichermedium, z.B. einem Plattenspeicher, abgelegt. Zusätzlich werden Zugriffsinformationen erzeugt und gepflegt, die ebenfalls extern verwaltet werden. Die extern gespeicherten Elemente und ihre Zugriffsinformationen innerhalb der Dateien sind physikalisch geblockt, so dass bei dem Suchverfahren die Anzahl von Zugriffen auf Datenblöcke bzw. deren rechnerinterner Transfer (neben der Anzahl erforderlicher Elementvergleiche) von primärem Interesse ist.

Die Organisation der Zugriffsinformationen in Hinblick auf effiziente Suchverfahren hängt wesentlich davon ab, in welcher Reihenfolge die Elemente in die Datenstruktur aufgenommen. Beispielsweise kann bei Verwendung eines binären Suchbaums eine lineare Liste oder auch ein vollständiger Binärbaum entstehen. Dabei ergeben sich dann Aufwandabschätzungen für das einmalige Suchen mit Ordnungen zwischen $O(n)$ und $O(\log(n))$, da der Zeitaufwand einer Such- und Einfügeoperation bei Einsatz eines Binärbaums zur Organisation der Zugriffsinformationen im wesentlichen von seiner Höhe abhängt. Insgesamt erfordert bei einem binären Suchbaum das Einfügen von n Elementen, ausgehend von einem leeren binären Suchbaum, im ungünstigsten Fall $O\left(n^2\right)$ viele Schritte. Es lässt sich zeigen ([AHU]), dass die

mittlere Anzahl von Schritten, um n Elemente mit zufälligen Werten des Ordnungskriteriums[13] in einen anfangs leeren binären Suchbaum einzufügen, von der Ordnung $O(n \cdot \log(n))$ ist. Weiter gilt, dass eine Folge von n ***insert***-, ***delete***-, ***is_member***- (und ***min***-) Operationen mit einem mittleren Zeitaufwand der Ordnung $O(n \cdot \log(n))$ ausgeführt werden kann. Es erscheint daher erstrebenswert, eine Datenstruktur zur Organisation der Zugriffsinformationen auf die Elemente als Binärbaum so zu entwerfen, dass durch die mehrfache Ausführung der für sie definierten Operationen jeweils ein **höhenbalancierter Baum** entsteht, d.h. für den sich die Pfadlängen von der Wurzel zu den einzelnen Blättern möglichst wenig unterscheiden. Der Idealfall ist der **vollkommen höhenbalancierte Baum**, bei dem für jeden Knoten die Anzahl der Knoten in seinem linken Teilbaum um höchstens 1 von der Anzahl der Knoten im rechten Teilbaum differiert. Als Konsequenz folgt, dass sich bei einem vollkommen höhenbalancierten Baum für jeden Knoten die Höhen seiner von ihm ausgehenden Teilbäume um höchstens 1 unterscheiden.

Diese Folgerung liefert eine notwendige, aber nicht hinreichende Bedingung für einen vollkommen höhenbalancierten Baum. Ein **AVL-Baum** (benannt nach G.M. Adelson-Velskii und E.M. Landis) beispielsweise ist dadurch definiert, dass sich für jeden Knoten die Höhen der beiden von ihm ausgehenden Teilbäume um höchstens 1 unterscheiden; er ist aber keineswegs vollkommen höhenbalanciert, wie das Beispiel in Abbildung 6.2.2-1 zeigt.

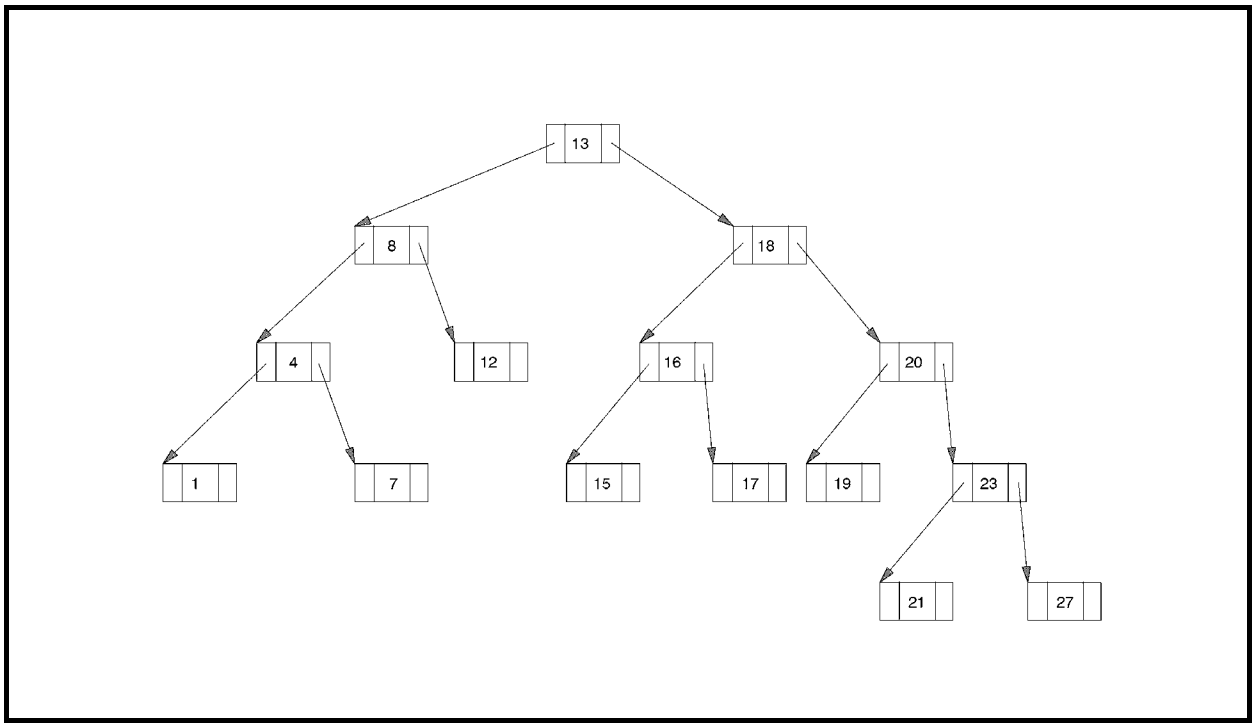


Abbildung 6.2.2-1: AVL-Baum

13 Eine Folge von Werten $a_1, ..., a_n$ ist eine zufällige Folge, wenn jedes a_i mit gleicher Wahrscheinlichkeit $1/n$ das j-kleinste ($1 \leq j \leq n$) ist.

Eine genaue Analyse der notwendigen Zugriffe, um bei einem binären Suchbaum mit n Knoten zu einem Knoten zu gelangen, zeigt die folgende Zusammenfassung (siehe angegebene Literatur).

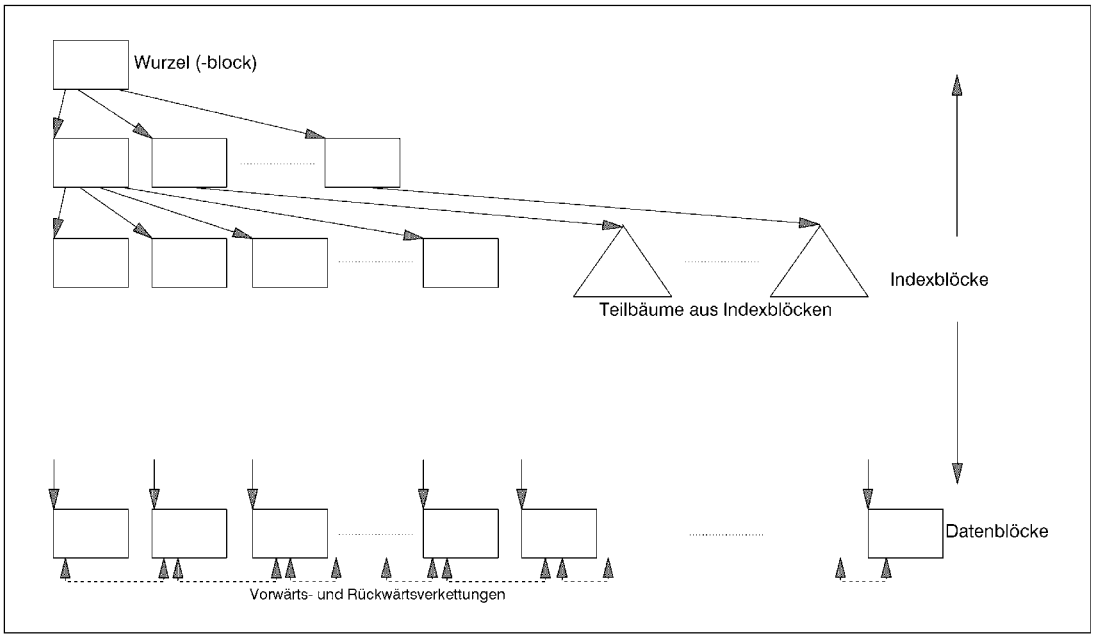
	maximale Anzahl $M(n)$	durchschnittliche Anzahl $D(n)$
binärer Suchbaum	n	$2(n+1)/n \cdot H_n - 3$ mit $H_n = \sum_{k=1}^{n} 1/k$. Für große n gilt $D(n) \approx 1{,}39 \cdot \log_2(n)$
vollkommen höhenbalancierter binärer Suchbaum	$\lfloor \log_2(n+1) \rfloor$	$1/n \cdot \left(1 + (n+1)M(n) - 2^{M(n)}\right)$. Für große n gilt $D(n) \approx \log_2(n) - 1$.
AVL-Baum	$< 1{,}44 \cdot \log_2(n+2)$	wie beim vollkommen höhenbalancierten binären Suchbaum

In praktischen Anwendungsfällen besteht ein Element der Datenstruktur häufig aus einem Datensatz, der über den Wert eines Primärschlüsselfelds eindeutig identifiziert wird und eine Reihe weiterer meist umfangreicher Komponenten beinhaltet. Die Datenstruktur stellt in diesem Fall eine (physikalische) Datei dar. Bei einer großen Anzahl von Datensätzen, die auf einem externen Speichermedium liegen, ist es im allgemeinen nicht angebracht, diese Datei in Form eines binären Suchbaums zu organisieren. Jeder Zugriff auf einen Datensatz, der dann in einem Knoten des binären Suchbaums liegt, erfordert einen Datentransfer zwischen Arbeitsspeicher und Peripherie. Der hierbei zu veranschlagende Zeitaufwand ist selbst im Idealfall des vollkommen höhenbalancierten binären Suchbaums in der Regel nicht akzeptabel. Aus den obigen Formeln wird ersichtlich, dass im Idealfall des vollkommen höhenbalancierten binären Suchbaums bei einer Datei mit beispielsweise 20.000 Datensätzen im Durchschnitt ca. 13,3 und bei 200.000 Datensätzen im Durchschnitt 16,6 (vergleichsweise sehr zeitaufwendige) Plattenzugriffe erforderlich sind, um zu einem Datensatz zu gelangen. In der Praxis ist dieser Wert unakzeptabel. Es sind daher andere Realisierungsformen für Dateien angebracht.

Von den in der Literatur (vgl. z.B. [AHU]) beschriebenen Vorschlägen ist das Konzept des B*-Baums zur Realisierung von Dateien von hoher praktischer Relevanz; es wird zur Speicherung von Datensätzen in ISAM-Dateien bzw. VSAM-Dateien in Großrechner-Betriebssystemen eingesetzt.

Die **satzorientierte Zugriffsmethode ISAM** verarbeitet Datensätze, die über Primärschlüsselwerte identifiziert werden. Physikalisch werden die Datensätze einer so bearbeiteten Datei (**ISAM-Datei**) geblockt, so dass i.a. ein Datenblock mehrere Datensätze enthält. Es gibt in einer ISAM-Datei zwei Typen von Blöcken (Abbildung 6.2.2-2):

- **Datenblöcke** enthalten die logischen Datensätze des Anwenders und Verwaltungsinformationen zur logischen Verkettung der Datenblöcke. Innerhalb eines Datenblocks sind die enthaltenen Sätze nach aufsteigenden Primärschlüsselwerten sortiert. Die Datenblöcke sind vorwärts bzw. rückwärts miteinander verkettet (Verweise über Blocknummern), und zwar zeigt die Vorwärtsverkettung auf den Datenblock, der in der Reihenfolge der Primärschlüsselwerte die nächsten Datensätze enthält; die Rückwärtsverkettung zeigt auf den Datenblock, der in dieser Reihenfolge die vorhergehenden Datensätze enthält. Bei einem Einstieg in die Datei in den Datenblock, der den Datensatz mit dem kleinsten Primärschlüsselwert enthält, ist daher eine sequentielle Satzverarbeitung möglich
- **Indexblöcke** enthalten neben Verwaltungsinformationen mögliche Primärschlüsselwerte von Datensätzen und Verweise auf andere Indexblöcke bzw. Datenblöcke.



Inhalt einer ISAM-Datei (SNI BS2000) bzw. VSAM-Datei (IBM MVS/ESA, IBM VSE)

Abbildung 6.2.2-2: ISAM-Datei

Eine ISAM-Datei ist in Form eines B*-Baums organisiert. Ein **B*-Baum** ist eine aus **Knoten** bestehende Datenstruktur, die miteinander hierarchisch in **Form eines Baums** verknüpft sind. Hierbei hat ein Knoten meist mehr als 2 Nachfolger.

Im Fall einer ISAM-Datei stellen die Indexblöcke die inneren Knoten und die Datenblöcke die Blätter des B*-Baums dar. Die Begriffe Knoten und Block werden im folgenden synonym verwendet.

Ein **B*-Baum** wird durch zwei **Parameter *u* und *v*** und durch die folgenden Eigenschaften charakterisiert; die Parameter u und v beschreiben dabei einen Mindestfüllungsgrad der Indexblöcke bzw. der Datenblöcke:

1. Alle Blätter haben denselben Abstand von der Wurzel d.h. die Anzahl der Knoten, die beginnend bei der Wurzel in Richtung der Kanten durchlaufen werden müssen, um ein Blatt zu erreichen ist für alle Blätter gleich
2. Die Wurzel ist ein Blatt oder hat mindestens 2 Nachfolger
3. Jeder innere Knoten außer der Wurzel hat mindestens $u + 1$ und höchstens $2u + 1$ Nachfolger; falls die Wurzel kein Blatt ist, hat sie mindestens 2 und höchstens $2u + 1$ Nachfolger
4. Jeder innere Knoten mit $s + 1$ Nachfolgern ist mit s Primärschlüsselwerten markiert; diese Primärschlüsselwerte sind aufsteigend sortiert
5. Jedes Blatt enthält, wenn es nicht die Wurzel ist, mindestens v und höchstens $2v$ Datensätze, die nach aufsteigenden Primärschlüsselwerten sortiert sind; ist die Wurzel ein Blatt, so enthält sie höchstens $2v$ nach aufsteigenden Primärschlüsselwerten sortierte Datensätze.

Wegen Bedingung 1. ist die Höhe des B*-Baums gleich der Anzahl der Knoten, die beginnend bei der Wurzel in Richtung der Kanten durchlaufen werden müssen, um ein Blatt zu erreichen. Bedingung 1. besagt, dass der Baum **vollständig höhenbalanciert** ist.

Ein **innerer Knoten (Indexblock)** lässt sich folgendermaßen skizzieren:

$$[p_0 \;\; k_1 \;\; p_1 \;\; k_2 \;\; \ldots \;\; k_s \;\; p_s].$$

Hierbei sind die Werte p_i Verweise (Pointer) auf die Nachfolger des Knotens und k_i Primärschlüsselwerte mit $k_1 \le k_2 \le \ldots \le k_s$, und es ist $u \le s \le 2u$. Jedes p_i zeigt auf einen untergeordneten Teilbaum T_{p_i}, der selbst wieder Primärschlüsselwerte enthält. Zu den obigen Bedingungen 1. - 5. kommt zusätzlich noch folgende Bedingung:

6. Alle in $T_{p_{i-1}}$ vorkommenden Primärschlüsselwerte sind kleiner oder gleich k_i, und k_i ist selbst größer als alle in T_{p_i} vorkommenden Primärschlüsselwerte.

Ein **Blatt (Datenblock)** hat die Darstellung (hier sind zur vereinfachten Darstellung nur die Primärschlüsselwerte der Datensätze angegeben)

$$[k_1 \;\; k_2 \;\; \ldots \;\; k_t]$$

mit $k_1 \le k_2 \le \ldots \le k_t$ und $v \le t \le 2v$.

Zum schnelleren Navigieren durch den B*-Baum sind je nach Implementierung die Knoten im Baum vorwärts und rückwärts verkettet, d.h. ein Knoten enthält neben den (Vorwärts-) Verweisen auf seine Nachfolgerknoten einen zusätzlichen (Rückwärts-) Verweis auf seinen Vorgängerknoten. Außerdem sind (wieder implementierungsabhängig) alle Knoten desselben Rangs als doppelt-verkettete Liste miteinander verknüpft.

Um auf einen Datensatz in einem B*-Baum zugreifen zu können, muss ein Pfad von der Wurzel des Baums bis zu dem Blatt (Datenblock) durchlaufen werden, der den Datensatz enthält. In der Regel bewirkt der Besuch eines jeden Knotens einen physikalischen Datentransfer.

Das **Aufsuchen** des Datenblocks, der einen **Datensatz** mit vorgegebenem Primärschlüsselwert *k* enthält bzw. bei Nichtvorhandensein in der Datei enthalten würde, wenn der Datensatz in der Datei vorkäme, kann folgendermaßen ablaufen: Ist die Wurzel ein Blatt, also selbst ein Datenblock, so ist damit der Datenblock gefunden. Ist die Wurzel kein Blatt, sondern ein Indexblock, so werden die Werte dieses Indexblocks mit *k* verglichen, bis derjenige Eintrag gefunden ist, für den $k_j < k \leq k_{j+1}$ gilt (für $k \leq k_1$ ist $j = 0$, für $k > k_s$ wird nur mit k_s verglichen und $j = s$ gesetzt). Dann wird im Indexblock, auf den p_j verweist, nach der gleichen Methode weitergesucht, bis man ein Blatt, also einen Datenblock erreicht hat. Dieses ist der Datenblock, der den Datensatz mit dem Primärschlüsselwert *k* enthält bzw. enthalten würde.

Alle **Routinen zur Manipulation** (Einfügen, Entfernen, Aufsuchen eines Datensatzes, Durchlaufen eines B*-Baums usw.) müssen sicherstellen, dass durch eine Manipulation im Baum die definierenden Eigenschaften nicht verletzt werden. Problematisch sind die Operationen zur Aufnahme weiterer Datensätze und zur Entfernung von Datensätzen aus der Datenstruktur, da diese Operationen sicherstellen müssen, dass die vollkommene Höhenbalance jeweils erhalten bleibt. Dabei tritt eventuell eine Situation ein, in der ein weiterer Datensatz in einen Datenblock einzufügen ist, der aber bereits vollständig belegt ist (**Überlauf**). Entsprechend kann eine Situation entstehen, in der ein Datensatz zu entfernen ist, so dass der betreffende Datenblock anschließend weniger als *v* Datensätze enthält (**Unterlauf**). Über- und Unterlaufsituationen gibt es auch im Zusammenhang mit Indexblöcken.

Die üblichen in der Literatur besprochenen Verfahren bewirken, dass der B*-Baum in Richtung von den Blättern zur Wurzel wächst.

Im folgenden Beispiel (Abbildung 6.2.2-3) werden Datensätze in einen anfangs leeren B*-Baum eingefügt und aus ihm entfernt. Dargestellt sind nur die Primärschlüsselwerte (hier key-Werte genannt) und die Baumstruktur ohne die zusätzlichen implementierungsabhängigen Verweise zur Erleichterung der Navigation durch den Baum. Die Parameter des B*-Baums lauten $u = v = 2$. Das Prinzip der Über- und Unterlaufbehandlung wird im wesentlichen Ablauf erläutert:

Um einen neuen Datensatz mit Primärschlüsselwert k einzufügen, wird zunächst der Datenblock aufgesucht (siehe oben), der in der Sortierreihenfolge den Datensatz aufnehmen müsste. Wenn dieser Datenblock noch nicht vollständig gefüllt ist, also noch nicht $2v$ Sätze enthält, wird der neue Datensatz an die in der Sortierreihenfolge richtigen Position eingefügt. Ist der Datenblock bereits vollständig belegt, so hat er die Form

$$[k_1 \;\; k_2 \;\; \ldots \;\; k_v \;\; k_{v+1} \;\; \ldots \;\; k_{2v}].$$

Da der neue Datensatz in diesen Datenblock gehört, gilt $k_1 \le k \le k_{2v}$. Damit der neue Datensatz eingefügt werden kann, wird der Datenblock in zwei Datenblöcke

$$[k_1 \;\; k_2 \;\; \ldots \;\; k_v] \text{ und } [k_{v+1} \;\; \ldots \;\; k_{2v}]$$

aufgeteilt und der neue Datensatz bei $k \le k_v$ in den ersten bzw. bei $k > k_v$ in den zweiten dieser aufgeteilten Datenblöcke an der in der Sortierung richtigen Position eingefügt. Außerdem wird der *größte Primärschlüsselwert im ersten* der aufgeteilten Datenblöcke *zusammen mit einem Verweis auf den zweiten der aufgeteilten Datenblöcke* in den übergeordneten Indexblock an die richtige Position einsortiert, die sich durch die Reihenfolge der Primärschlüsselwerte im Indexblock ergibt. In Abbildung 6.2.2-3 tritt diese Überlaufsituation z.B. beim Einfügen der Datensätze mit Primärschlüsselwerten 19 bzw. 12 ein.

Eine eventuelle Überlaufbehandlung in einem Indexblock wird analog behandelt, wobei sich diese u.U. rekursiv bis zur Aufnahme einer neuen Wurzel fortsetzt (in Abbildung 6.2.2-3 z.B. bei der Aufnahme der Datensätze mit Primärschlüssel 5 bzw. 12): Ein vollständig belegter Indexblock hat die Form

$$[p_0 \;\; k_1 \;\; p_1 \;\; k_2 \;\; \ldots \;\; k_{2u} \;\; p_{2u}];$$

der einzusortierende Eintrag sei $[k \;\; p]$. Man kann sich zunächst $[k \;\; p]$ in den vollen Indexblock an die größenmäßig richtige Stelle einsortiert denken, d.h. bei $k \le k_1$ entsteht dabei

$$[p_0 \;\; k \;\; p \;\; k_1 \;\; p_1 \;\; k_2 \;\; \ldots \;\; k_{2u} \;\; p_{2u}],$$

bei $k_j < k \le k_{j+1}$ für ein j mit $1 \le j < 2u$ entsteht

$$[p_0 \;\; k_1 \;\; p_1 \;\; \ldots \;\; k_j \;\; p_j \;\; k \;\; p \;\; k_{j+1} \;\; p_{j+1} \;\; \ldots \;\; k_{2u} \;\; p_{2u}]$$

und bei $k > k_{2u}$ entsteht

$$[p_0 \;\; k_1 \;\; p_1 \;\; k_2 \;\; \ldots \;\; k_{2u} \;\; p_{2u} \;\; k \;\; p].$$

Dieser (gedachte) Block enthält $2u+1$ Primärschlüsselwerte und $2u+2$ Verweise und kann daher als

$$[q_0 \;\; l_1 \;\; q_1 \;\; l_2 \;\; \ldots \;\; l_u \;\; q_u \;\; l_{u+1} \;\; q_{u+1} \;\; l_{u+2} \;\; \ldots \;\; l_{2u+1} \;\; q_{2u+1}]$$

geschrieben werden ($\{q_0, \ldots, q_{2u+1}\} = \{p_0, \ldots, p_{2u}, p\}$ und $\{l_1, \ldots, l_{2u+1}\} = \{k_1, \ldots, k_{2u}, k\}$). Er wird in zwei Indexblöcke

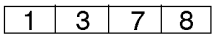
$$[q_0 \;\; l_1 \;\; q_1 \;\; l_2 \;\; \ldots \;\; l_u \;\; q_u] \text{ und } [q_{u+1} \;\; l_{u+2} \;\; \ldots \;\; l_{2u+1} \;\; q_{2u+1}]$$

aufgeteilt und der *Wert* l_{u+1} *zusammen mit einem Verweis auf den zweiten der aufgeteilten Indexblöcke* nach dem gleichen Prinzip in den Indexblock eingeordnet, der dem übergelaufenen Indexblock im B*-Baum übergeordnet war (evtl. muss dabei eine neue Wurzel angelegt werden).

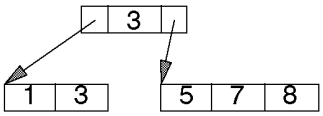
Eine Unterlaufbehandlung eines Datenblocks ist erforderlich, wenn er bei Entfernen eines Datensatzes weniger als v Datensätze enthalten würde. Er wird dann mit einem Nachbar-Datenblock zu einem Datenblock verschmolzen. Wenn dabei ein Überlauf entsteht, wird diese Situation wie oben behandelt. Aus dem übergeordneten Indexblock wird ein entsprechender Verweis entfernt, wobei auch hierbei ein Unterlauf eintreten kann, dem durch Zusammenlegen benachbarter Indexblöcke begegnet wird. Eine eventuelle Überlaufbehandlung beim Verschmelzen der Indexblöcke wird wie oben behandelt. In Abbildung 6.2.2-3 finden derartige komplexe Vorgänge bei der Entfernung des Datensatzes mit dem Primärschlüsselwert 2 statt.

Die Suchdauer nach einem vorgegebenen Datensatz bzw. das Entfernen oder Einfügen eines Datensatzes ist in einem B*-Baum proportional zur Länge eines Pfades von der Wurzel zu einem Blatt. Alle diese Pfade sind aufgrund der vollständigen Höhenbalance des Baums gleichlang und liefern somit einen Maßstab für die Zugriffsgeschwindigkeit (Performance) dieser Datenstruktur. Die Länge eines Pfades bzw. die Zugriffsgeschwindigkeit auf einen Datenblock hängt nicht vom Primärschlüsselwert ab, nach dem gesucht wird, sondern von der Anzahl an Datensätzen, die sich zur Zeit im B*-Baum befinden.

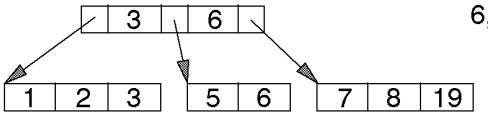
Ausgangssituation ist ein leerer B*-Baum
Es werden nacheinander Datensätze mit den key-Werten
7, 1, 3, 8 aufgenommen:



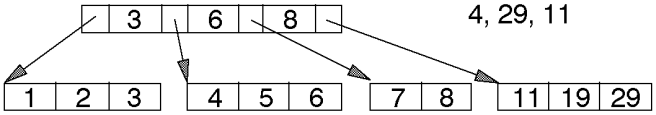
Aufnahme eines Datensatzes mit key-Wert 5



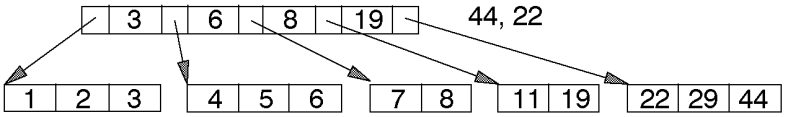
Aufnahme von Datensätzen mit key-Werten 6, 2, 19



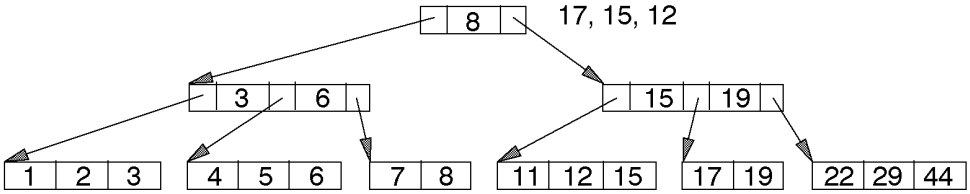
Aufnahme von Datensätzen mit key-Werten 4, 29, 11



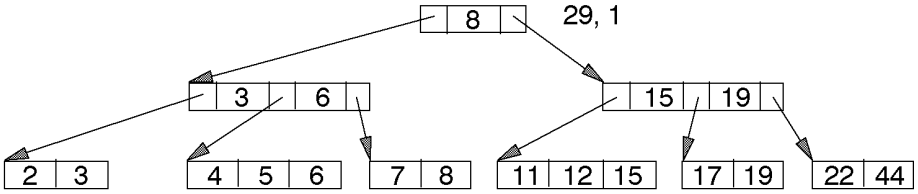
Aufnahme von Datensätzen mit key-Werten 44, 22



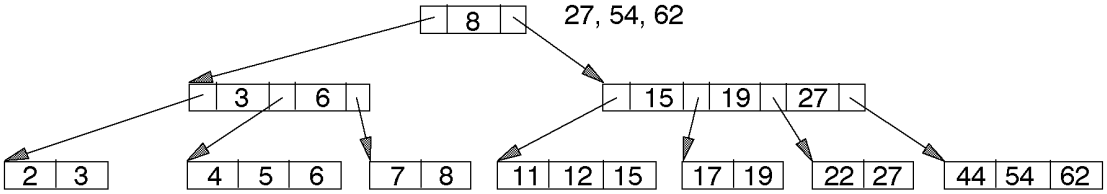
Aufnahme von Datensätzen mit key-Werten 17, 15, 12



Entfernen der Datensätze mit key-Werten 29, 1



Aufnahme von Datensätzen mit key-Werten 27, 54, 62



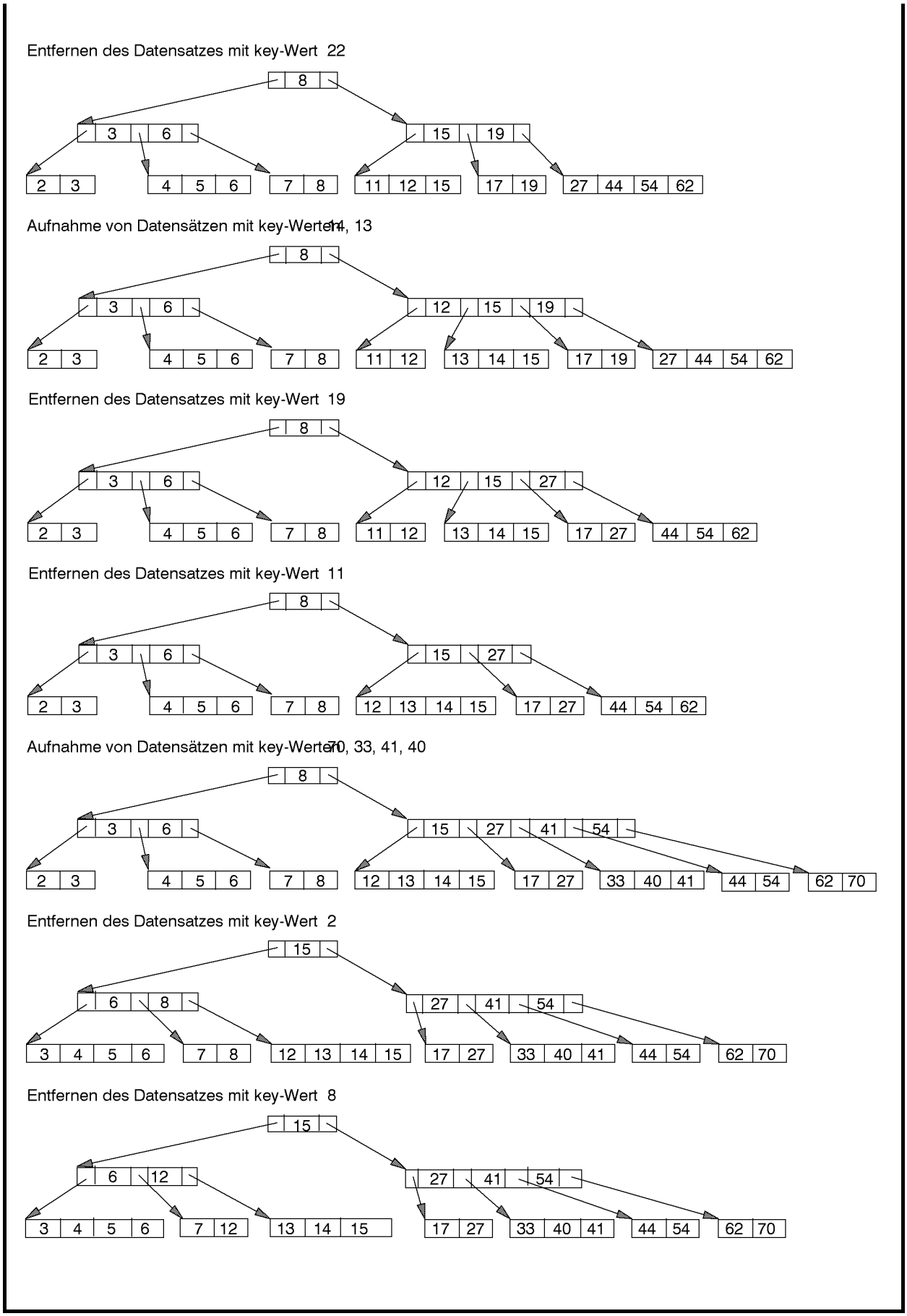


Abbildung 6.2.2-3: Beispiel eines B*-Baums

Abbildung 6.2.2-4 zeigt das **Mengengerüst eines B*-Baums**. In dieser Darstellung führen aus einem inneren Knoten nur zwei Kanten heraus, die jedoch alle aus dem Knoten herausführenden Kanten repräsentieren. Der Zahlenwert an der linken Kante gibt an, wieviele Nachfolger der Knoten mindestens hat; der Zahlenwert an der rechten Kante beschreibt die Maximalzahl an herausführenden Kanten aus diesem Knoten. Bei einem Datenblock beziehen sich die Zahlenangaben auf die Mindest- bzw. Maximalanzahl an Datensätzen des Datenblocks.

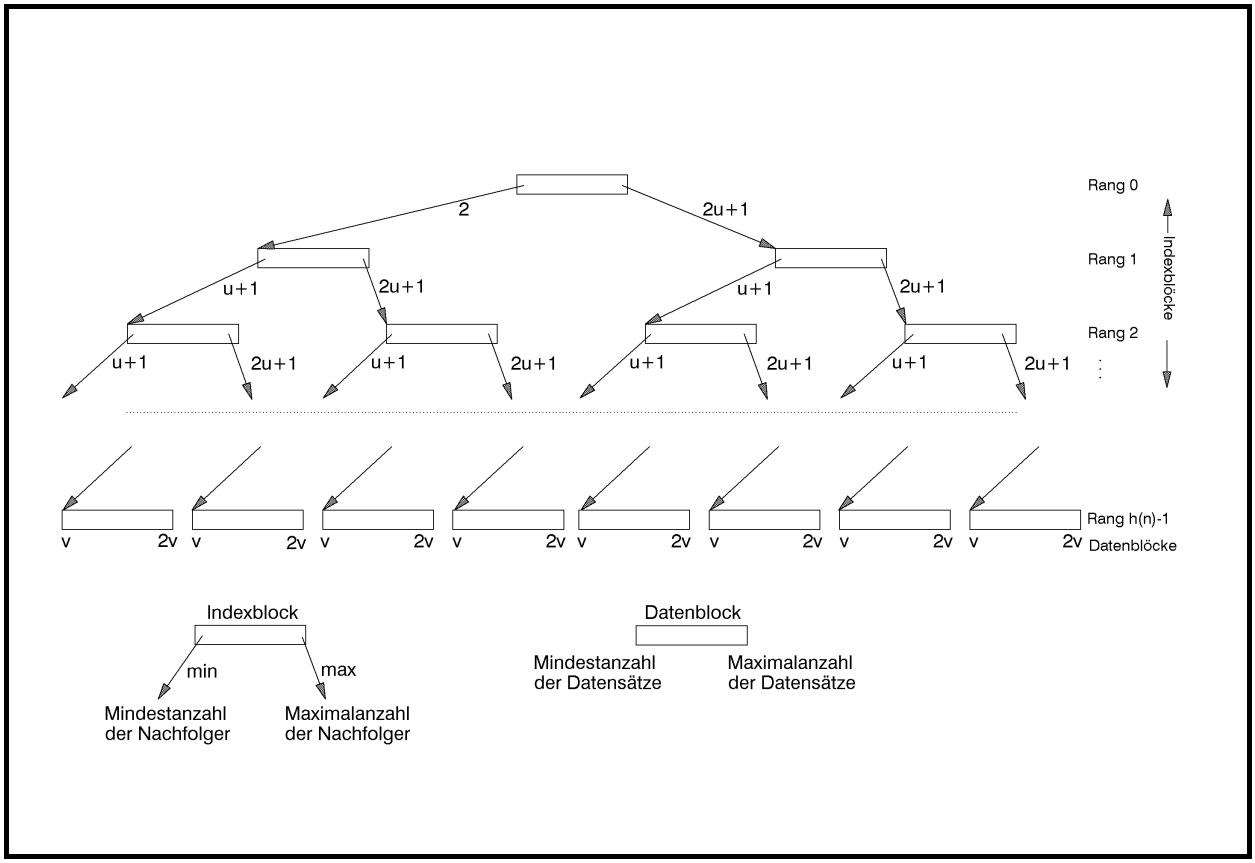


Abbildung 6.2.2-4: Mengengerüst eines B*-Baums

Der B*-Baum mit Parametern u und v enthalte n Datensätze. Mit $DB(n)$ werde die Anzahl der Datenblöcke bezeichnet, um diese Datensätze im B*-Baum zu speichern. Die Anzahl der dabei verwendeten Indexblöcke sei $IB(n)$. Die Höhe des B*-Baums sei $h(n)$. Dieser Wert beschreibt die Anzahl von Blocktransfers, die nötig sind, um an einen beliebigen Datensatz der Datei zu gelangen; der Aufwand zum Suchen *innerhalb* eines Datenblocks nach einem bezeichneten Datensatz bzw. eines Primärschlüsselwerts *innerhalb* eines Indexblocks soll hier unberücksichtigt bleiben.

Dann gilt:

Für $1 \le n \le 2v$ besteht der Baum nur aus der Wurzel, d.h. $h(n) = 1$, $DB(n) = 1$, $IB(n) = 0$.

Für $n \ge 2v+1$ werden Index- und Datenblöcke benötigt. Die Wurzel wird ein Indexblock, und der B*-Baum bekommt mindestens die Höhe 2. Die entsprechenden Werte bei Höhe 2 lauten:

$h(n) = 2$, $2v + 1 \le n \le 2v \cdot (2u + 1)$, $2 \le DB(n) \le 2u + 1$, $IB(n) = 1$.

Aus Abbildung 6.2.2-4 sieht man, dass bei $h(n) = h \ge 2$ gilt:

$$h(n) = h \ge 2,\ 2v \cdot (u+1)^{h-2} \le n \le 2v \cdot (2u+1)^{h-1},\ 2 \cdot (u+1)^{h-2} \le DB(n) \le (2u+1)^{h-1}$$

und bei $h(n) = h \ge 3$ außerdem

$$1 + 2 \cdot \sum_{i=0}^{h-3} (u+1)^i \le IB(n) \le \sum_{i=0}^{h-2} (2u+1)^i \text{, also}$$

$$1 + 2 \cdot \frac{(u+1)^{h-2} - 1}{u} \le IB(n) \le \frac{(2u+1)^{h-1} - 1}{2u}.$$

Enthält eine Datei, die nach dem hier beschriebenen Prinzip eines B*-Baums mit Parametern u und v gespeichert ist, $n \ge 2v + 1$ viele Datensätze, so gilt für die Höhe $h(n)$ des Baums:

$$\frac{\log(n) - \log(2v)}{\log(2u+1)} + 1 \le h(n) \le \frac{\log(n) - \log(2v)}{\log(u+1)} + 2.$$

Im allgemeinen belegen Primärschlüsselwerte und die Verweise weniger Speicherplatz als komplette Datensätze. Daher kann man $u \ge v$ wählen.

Belegt beispielsweise ein Datensatz 200 Bytes, wobei 20 Bytes davon für den Primärschlüssel benötigt werden, so können bei einer Blockgröße von 2.048 Bytes 10 Datensätze pro Datenblock abgelegt werden (der Platz für implementierungsabhängige Zusatzverweise ist bereits eingerechnet), d.h. $2v = 10$. Sieht man für einen Verweis innerhalb eines Indexblocks 4 Bytes vor, so ist er gefüllt, wenn er $2u = 84$ Primärschlüsselwerte und Verweise enthält; die restlichen Bytes seien für zusätzliche organisatorische Verweise reserviert. Bei $n = 15.000$ Datensätzen gilt dann für die Höhe $h(n)$ des B*-Baums und damit für die Anzahl physikalischer Blocktransfers, um einen Datensatz aufzusuchen, $2{,}64 \le h(15.000) \le 3{,}94$, also $h(15.000) = 3$. Bei $n = 150.000$ ist $3{,}164 \le h(150.000) \le 4{,}557$, also $h(150.000) = 4$. Bei einer weiteren Verzehnfachung der Anzahl an Datensätzen auf $n = 1.500.000$ wächst die Anzahl benötigter Blocktransfers, um einen Datensatz aufzusuchen, auf $h(1.500.000) = 5$, denn $4{,}556 \le h(1.500.000) \le 5{,}169$.

Die Anzahl wirklich benötigter Datenblöcke hängt vom Füllungsgrad eines Datenblockes und implizit von der Reihenfolge ab, in der Datensätze in die Datei aufgenommen bzw. aus ihr entfernt werden. Das gleiche gilt für die Anzahl benötigter Indexblöcke.

7 Allgemeine Lösungsstrategien

Im folgenden werden einige **allgemeine Problemlösungsstrategien** vorgestellt, die auch schon in den vorhergehenden Kapiteln angewandt wurden (siehe auch [OTT]). Selbstverständlich lassen sich nicht alle Lösungsansätze in diese Strategien einordnen. Sie geben jedoch häufig eine Richtschnur zum Auffinden effizienter Basis- und Anwendungsalgorithmen.

Die zu lösenden Probleme hängen im allgemeinen von Eingabeinstanzen ab, die aus n Elementen, wie zu sortierende Objekte oder Knoten und Kanten eines Graphen, bestehen. In der Theoretischen Informatik wird als **Problemgröße** zur Bestimmung der Komplexität eines Algorithmus die Anzahl an Zeichen (Bits bzw. die elementaren Zeichen eines Grundalphabets) definiert, die zur Darstellung einer Eingabeinstanz erforderlich sind. Dieser Ansatz ist zumindest dann sinnvoll, wenn die Laufzeit der aus den Problemlösungsstrategien entwickelten Algorithmen von (numerischen) Werten der Eingabeinstanz und nicht nur von der Anzahl an Elementen, die eine Eingabeinstanz ausmachen, abhängt. Eine möglichst kleine obere Schranke für die Laufzeit eines Verfahrens in Abhängigkeit von der so definierten Problemgröße gibt die **Komplexität im ungünstigsten Fall** an. In den folgenden Unterkapiteln wird häufig ein etwas vereinfachter Ansatz gewählt: es werden Schranken für die Anzahl durchlaufener Anweisungen in Abhängigkeit von der Anzahl der Teilobjekte bestimmt, aus denen eine Eingabeinstanz besteht.

7.1 Divide and Conquer

Zur Lösung eines Problems nach dem Divide-and-Conquer-Prinzip geht man wie folgt vor:

Divide-and-Conquer (Teile-und-Herrsche)

Gegeben sei ein Problem der Größe n. Man führt man die folgenden Schritte durch:

1. ***Divide:*** Man zerlegt das Problem in eine Anzahl kleinerer disjunkter Teilprobleme (vom gleichen Problemtyp).

2. ***Conquer:*** Sind die Teilprobleme klein genug, so löst man diese direkt. Andernfalls löst man jedes Teilproblem (rekursiv) nach dem Divide-and-Conquer-Prinzip, d.h. mit demselben Algorithmus.

3. ***Combine:*** Man setzt die Lösungen der Teilprobleme zu einer Gesamtlösung des ursprünglichen Problems zusammen.

Die Anwendung des Divide-and-Conquer-Prinzips setzt voraus, dass das Gesamtproblem auf geeignete Weise in mindestens zwei kleinere Teilprobleme zerlegt werden kann, die leichter zu lösen sind. Dies ist häufig dann möglich, wenn der Aufwand zur Lösung eines Problems lediglich von der Problemgröße und nicht von den Werten der Eingabegrößen einer Problemstellung abhängt. Eine weitere Voraussetzung ist die effiziente Kombinierbarkeit der Teillösungen zu einer Gesamtlösung des Problems.

Beispiele dieses Prinzips wurden mit dem Quicksort und der Binärsuche bereits in den vorhergehenden Kapiteln behandelt.

7.1.1 Eine allgemeine Rekursionsformel

Zur Abschätzung der Zeitkomplexität $T(n)$ eines Lösungsverfahrens nach dem Divide-and-Conquer-Prinzip bei Eingabe eines Problems der Größe n lassen sich häufig Rekursionsgleichungen der Form

$$T(n) = \begin{cases} a & \text{für } n = 1 \\ b \cdot T(n/c) + f(n) & \text{für } n > 1 \end{cases}$$

aufstellen. Hierbei sind die Werte a, b und c natürliche Zahlen mit $1 \le b \le c$. Es wird angenommen, dass im Divide-Schritt das Problem in b kleinere Probleme mit einer Problemgröße aufgeteilt wird, die mit dem Faktor $1/c$ kleiner als das Ausgangsproblem sind. Der Wert $f(n)$ gibt die Zeitkomplexität des Aufteilungsvorgangs bei Problemgröße n und den Aufwand an, kleinere Teillösungen zu einer Gesamtlösung zu verschmelzen.

Es sei vorausgesetzt, dass f mindestens linear wächst, d.h. dass $i \cdot f(n/i) \le f(n)$ gilt. Unter der vereinfachenden Annahme, dass $n = c^m$ ist, lässt sich $T(n)$ berechnen zu

$$\begin{aligned} T(c^m) &= b \cdot T(c^{m-1}) + f(c^m) \\ &= b \cdot \left(b \cdot T(c^{m-2}) + f(c^{m-1})\right) + f(c^m) = b^2 \cdot T(c^{m-2}) + b \cdot f(c^{m-1}) + f(c^m) \\ &= b^k \cdot T(c^{m-k}) + \sum_{i=0}^{k-1} b^i \cdot f(c^{m-i}) \end{aligned}$$

für $0 \le k \le m = \log_c(n)$.

Setzt man $k = m$ und beachtet $b^i \cdot f(c^{m-i}) = b^i \cdot f(n/c^i) \le c^i \cdot f(n/c^i) \le f(n)$ so ergibt sich

bei $b = 1$: $T(n) \le a + \log_c(n) \cdot f(n)$,

bei $b > 1$: $T(n) \le b^m \cdot a + \log_c(n) \cdot f(n) \le c^m \cdot a + \log_c(n) \cdot f(n) = n \cdot a + \log_c(n) \cdot f(n)$,

d.h. $T(n) \in O\big(n + \log(n) \cdot f(n)\big)$.

Als Folgerung ergibt sich:

Für $f(n) \in O(1)$ ist $T(n) \in O(\log(n))$ bei $b = 1$ bzw. $T(n) \in O(n)$ bei $b > 1$,
für $f(n) \in O(n)$ ist $T(n) \in O(n \cdot \log(n))$,
für $f(n) \in O\big(n \cdot \log(n)\big)$ ist $T(n) \in O(n \cdot (\log(n))^2)$.

Beispielsweise steht beim Quicksort (Kapitel 6.1.2) für den Fall, dass in jedem Divide-Schritt zwei Teilfolgen gleicher Länge entstehen, $a = 0$, $b = 2$, $c = 2$ und $f(n) = n - 1$, d.h. der Aufwand ist in diesem Fall von der Ordnung $O(n \cdot \log(n))$.
Bei der Binärsuche, wie sie in Kapitel 6.2.1 implementiert wurde, gilt $a = 1$, $b = 1$, $c = 2$ und $f(n) = 2$, also ist der Aufwand von der Ordnung $O(\log(n))$.

7.2 Greedy-Methode

Die Greedy-Methode kann häufig zur Lösung von Optimierungsproblemen eingesetzt. Man geht wie folgt vor:

Greedy-Methode (für Optimierungsaufgaben):

Eine (global) optimale Lösung wird schrittweise gefunden, indem Entscheidungen gefällt werden, die auf „lokalen“ Informationen beruhen. Für eine Entscheidung wird hierbei nur ein kleiner (lokaler) Teil der Informationen genutzt, die über das gesamte Problem zur Verfügung stehen.

Es wird die in der jeweiligen Situation optimal erscheinende Entscheidung getroffen, unabhängig von vorhergehenden und nachfolgenden Entscheidungen. Die einmal getroffenen Entscheidungen werden im Laufe des Rechenprozesses nicht mehr revidiert.

Man geht nach folgendem Prinzip vor:

1. Man setzt $T := \emptyset$. Alle Elemente, die potentiell zu einer optimalen Lösung gehören können, gelten als „nicht behandelt“.

2. Solange es noch nicht behandelte Elemente gibt, wählt man ein Element a der noch nicht behandelten Elemente so aus, dass $T \cup \{a\}$ eine optimale Lösung bzgl. $T \cup \{a\}$ darstellt, und fügt a zu T hinzu. Das Element a gilt als „behandelt“.

3. T bestimmt die optimale Lösung.

Die Greedy-Methode schließt also aus einem lokalen Optimum, nämlich dem Optimum einer Teillösung, auf das globale Optimum des Problems. Daher ist sie nicht für jedes Problem anwendbar, da aus dem Optimum einer Teillösung nicht in allen Fällen auf das globale Optimum geschlossen werden kann.

7.2.1 Wege minimalen Gewichts in gerichteten Graphen

Die Greedy-Methode soll am Beispiel des Auffindens gewichtsminimaler Wege in einem gerichteten Graphen gezeigt werden, dessen Kanten mit nichtnegativen Werten gewichtet sind:

Das Problem der Wege mit minimalem Gewicht in gerichteten Graphen

Problem-
instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht. Zur Vereinfachung werden als Gewichte natürliche Zahlen genommen, die größer als 0 sind.

Gesuchte
Lösung: Die Summe der Kantengewichte der Wege mit minimalem Gesamtgewicht vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für $i = 1, ..., n$.

Die Problemgröße, nämlich die Anzahl der Bits, um eine Probleminstanz zu notieren, ist beschränkt durch $c \cdot \max\{|V|, |E|\} \cdot A$ mit $A = \max(\{\lfloor \log_2(w(e)) \rfloor + 1 \mid e \in E\})$ und einer Konstanten $c > 0$. Die Komplexität des folgenden Verfahrens wird vereinfachend in Abhängigkeit von der Anzahl der Knoten in einer Probleminstanz bestimmt.

Zur algorithmischen Behandlung wird wieder die `UNIT Graph` aus Kapitel 5.2.3 eingesetzt, die einen Graphen sowohl in Form seiner Adjazenzmatrix als auch in Form seiner Adjazenzliste verwaltet. Der Objekttyp `TGraph` aus dieser Unit wird in seinem `PUBLIC`-Teil um eine Methode `TGraph.minimale_wege` mit folgender Benutzerschnittstelle erweitert:

Methode	Bedeutung
PROCEDURE TGraph.minimale_wege;	Die Prozedur ermittelt die minimalen Gewichte der Wege vom Knoten $v_1 = 1$ zu allen anderen Knoten des Graphen (und legt sie in einem ARRAY ab, das über einen Pointer distanz im PRIVATE-Teil der Unit angesprochen werden kann bzw. zeigt die Ergebnisse an).

Im PRIVATE-Teil des Objekttyps TGraph wird ein Pointer distanz eingefügt, der auf ein ARRAY zeigt, das die minimalen Gewichte der Wege vom Knoten $v_1 = 1$ zu allen anderen Knoten aufnimmt:

```
distanz        : PVektor;
                 { minimales Gewicht eines Weges vom
                   Knoten v1 zu allen anderen Knoten          }
```

Initialisierungscode zur Allokation eines entsprechenden Speicherbereichs bzw. Code zur Freigabe dieses Speicherbereichs werden in den Konstruktor bzw. Destruktor des Objekttyps TGraph eingefügt. Außerdem wird eine Funktion min_real im Implementierungs-Teil verwendet, die das Minimum zweier Zahlen ermittelt:

```
FUNCTION min_real (a : INTEGER;
                   b : INTEGER): INTEGER;
BEGIN { min_real }
  IF a <= b THEN min_real := a
            ELSE min_real := b
END   { min_real };
```

Die Methode TGraph.minimale_wege verwendet wieder die Komponente besucht des Objekttyps TGraph, um festzuhalten, welcher Knoten *i* während des Ablaufs bereits behandelt wurde (besucht[*i* - 1] = 'T'). Der Code der Methode TGraph.minimale_wege lautet wie folgt; anschließend werden wieder einige Erläuterungen gegeben:

```
PROCEDURE TGraph.minimale_wege;

VAR i      : Knotenbereich;
    u      : Knotenbereich;
    exist  : BOOLEAN;

   FUNCTION auswahl (VAR knoten : Knotenbereich) : BOOLEAN;
     { die Prozedur wählt unter den noch nicht behandelten
       Knoten einen Knoten mit minimalem distanz-Wert aus
      (in knoten); gibt es einen derartigen Knoten, dann ist
       auswahl = TRUE, sonst ist auswahl = FALSE              }
```

```
      VAR  i   : Knotenbereich;
           min : INTEGER;

     BEGIN { auswahl }
       auswahl := FALSE;
       min     := unendlich;

       FOR i := 1 TO n DO
         IF besucht [i - 1] = 'F'
         THEN BEGIN
                IF distanz^.v(i) < min
                THEN BEGIN
                       auswahl := TRUE;
                       knoten  := i;
                       min     := distanz^.v(i);
                     END;
              END;
     END   { auswahl };


BEGIN { TGraph.minimale_wege }

  { Gewichte zum Startknoten initialisieren und alle Knoten,
    bis auf den Startknoten, als nicht behandelt kennzeichnen: }
  besucht := StrAlloc (n + 1);
  StrCopy (besucht, leer);

  FOR i := 1 TO n DO
    BEGIN
      distanz^.setze (i, Adjazenzmatrix^.m(1, i));
      StrCat (besucht, 'F');
    END;

  distanz^. setze(1, 0);
  besucht[0] := 'T';

  { einen Knoten mit minimalem Distanzwert aus den Restknoten
    auswählen:                                                }
  WHILE auswahl (u) DO
    BEGIN
      { den gefundenen Knoten aus den noch nicht behandelten
        Knoten herausnehmen:                                  }
      besucht[u-1] := 'T';

      { für jeden noch nicht behandelten Knoten idx, der mit
        dem Knoten u verbunden ist, den distanz-Wert auf den
```

```
        neuesten Stand bringen:                                  }
      FOR i := 1 TO n DO
        IF (besucht[i - 1] = 'F')
           AND
           (Adjazenzmatrix^.m(u, i) <> unendlich)
        THEN distanz^.setze (i, min_real
                                  (distanz^.v(i),
                                   distanz^.v(u)
                                   + Adjazenzmatrix^.m(u, i)));
    END  {WHILE };

  StrDispose (besucht);

  -- verarbeite das Feld distanz^, z.B. Ausgabe der Werte

END   { TGraph.minimale_wege };
```

Die Prozedur `TGraph.minimale_wege` realisiert das allgemeine Prinzip der Greedy-Methode: Zunächst gelten alle Knoten v_i bis auf den Knoten v_1 als „nicht behandelt". Aus den nicht behandelten Knoten wird ein Knoten ausgewählt, der zu einer bisherigen Teillösung hinzugenommen wird, und zwar so, dass dadurch eine bzgl. der Teillösung, d.h. bzgl. der behandelten Knoten, optimale Lösung entsteht. Der neu hinzugenommene Knoten v_i ist ein Knoten, der unter den noch nicht behandelten Knoten einen Weg minimalen Gewichts von v_1 aus besitzt, der nur durch bereits behandelte Knoten geht. Durch die Hinzunahme des Knotens v_i ändern sich eventuell die Wege mit minimalem Gewicht von v_1 aus zu allen Nachfolgern v_k von v_i: Verläuft nämlich bisher ein Weg minimalen Gewichts von v_1 nach v_k (ausschließlich durch behandelte Knoten) nicht über v_i und ist das Gewicht des Wegs von v_1 nach v_i, das in `distanz^.v(i)` steht, zuzüglich des Gewichts der Kante von v_i nach v_k kleiner als das bisherige minimale Gewicht von v_1 nach v_k, so ist jetzt ein neuer Weg von v_1 nach v_k mit kleinerem Gewicht gefunden, nämlich über v_i, der nur durch behandelte Knoten führt. Auf diese Weise wird bei jedem Auswahlschritt Einfluss auf die Bestimmung des globalen Optimums genommen.

Formal lässt sich das Verhalten der Methode durch die Schleifeninvariante der „`WHILE auswahl (u)`"-Schleife erklären. Beim Eintritt gilt:

(1) Ist der Knoten v_i ein bereits behandelter Knoten, d.h. `besucht[i - 1] = 'T'`, dann ist `distanz^.v(i)` das minimale Gewicht eines Weges von v_1 nach v_i, der nur behandelte Knoten v_j enthält, für die also `besucht[j - 1] = 'T'` ist.

(2) Ist der Knoten v_i ein nicht behandelter Knoten, d.h. `besucht[i - 1] = 'F'`, dann ist `distanz^.v(i)` das minimale Gewicht eines Weges von v_1 nach v_i, der bis auf v_i nur behandelte Knoten v_j enthält, für die also `besucht[j - 1] = 'T'` ist.

Die Schleifeninvariante ermöglicht die Anwendung der Greedy-Methode; aus ihr folgt die Korrektheit des Algorithmus.

Ist $G = (V, E, w)$ eine Eingabeinstanz des Problems der Wege mit minimalem Gewicht in gerichteten Graphen mit $|V| = n$, dann liefert das beschriebene Verfahren mit der Prozedur `TGraph.minimale_wege` im Feld `distanz^` die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für $i = 1, ..., n$.

Bei einem Aufruf der Funktion `auswahl` werden $O(n)$ viele Anweisungen durchlaufen. Insgesamt ist daher die Anzahl durchlaufener Anweisungen von der Ordnung $O(n^2)$.

Die folgende Tabelle fasst den Ablauf der Prozedur bei Eingabe des Graphen aus Abbildung 7.2.1-1 zusammen.

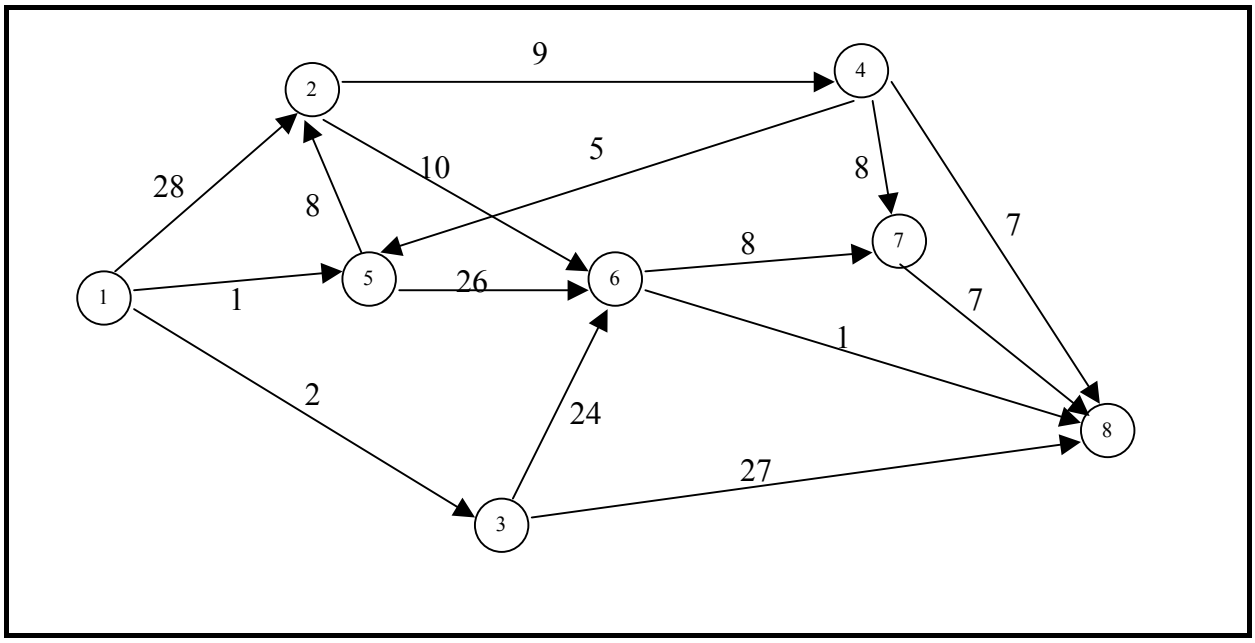


Abbildung 7.2.1-1: Beispielgraph

`u` wird jeweils beim Aufruf der Prozedur `auswahl(u)` ausgewählt:

u	nichtbehandelte Knoten	distanz [1]	distanz [2]	distanz [3]	distanz [4]	distanz [5]	distanz [6]	distanz [7]	distanz [8]
	2 3 4 5 6 7 8	---	28	2	∞	1	∞	∞	∞
5	2 3 4 6 7 8		9	2	∞	---	27	∞	∞
3	2 4 6 7 8		9	---	∞		26	∞	29
2	4 6 7 8		---		18		19	∞	∞
4	6 7 8				---		19	26	25
6	7 8						---	26	20
8	7							26	---
7	leer							---	

7.2.2 Aufspannende Bäume

Ein weiteres wichtiges Problem aus der Graphentheorie, das mittels Greedy-Methode gelöst werden kann, ist die Bestimmung eines aufspannenden Baums mit minimalem Gewicht in einem ungerichteten gewichteten Graphen. Ein **ungerichteter gewichteter Graph** $G = (V, E, w)$ wird wie ein gerichteter gewichteter Graph definiert, jedoch wird auf den Kanten keine Richtung festgelegt, d.h. mit $(v_i, v_j) \in E$ ist auch $(v_j, v_i) \in E$, und zwar mit demselben Gewicht.

Ein **Zyklus** in einem ungerichteten Graphen ist eine geschlossene Kantenfolge mit mindestens drei Knoten. Ein **(freier) Baum** ist ein zusammenhängender azyklischer ungerichteter Graph, d.h. er enthält keinen Zyklus, und jeder Knoten ist von jedem anderen Knoten über eine Kantenfolge in dem Baum erreichbar.

Ein Baum mit *n* Knoten hat genau $n-1$ viele Kanten. Fügt man zu einem Baum eine weitere Kante hinzu, so entsteht ein Zyklus.

Es sei $G = (V, E, w)$ ein ungerichteter gewichteter Graph mit *n* Knoten. Ein **aufspannender Baum** in *G* ist ein Baum, der alle Knoten von *G* enthält und dessen Kantenmenge eine Teilmenge von *E* ist. Das **Gewicht eines aufspannenden Baums** ist die Summe seiner Kantengewichte.

Eine in der Praxis häufig zu lösende Aufgabe besteht in der Bestimmung eines aufspannenden Baums mit minimalem Gewicht zu einem gegebenen ungerichteten gewichteten Graphen $G = (V, E, w)$ mit *n* Knoten.

Das Problem der Bestimmung eines aufspannenden Baums mit minimalem Gewicht in ungerichteten gewichteten Graphen

Problem-
instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht. Zur Vereinfachung werden als Gewichte natürliche Zahlen genommen, die größer als 0 sind.

Gesuchte
Lösung: Die Kantenmenge eines aufspannenden Baums mit minimalem Gesamtgewicht (unter allen aufspannenden Bäumen in G).

Die Problemgröße, nämlich die Anzahl der Bits, um eine Probleminstanz zu notieren, ist wieder beschränkt durch $c \cdot \max\{|V|, |E|\} \cdot A$ mit $A = \max(\{\lfloor \log_2(w(e)) \rfloor + 1 \mid e \in E\})$ und einer Konstanten $c > 0$. Die Komplexität des folgenden Verfahrens wird wieder wie in Kapitel 7.2.1 in Abhängigkeit von der Anzahl der Knoten bzw. der Kanten in einer Probleminstanz angegeben.

Folgender Satz legt Vorgehensweisen nach der Greedy-Methode zur Bestimmung eines aufspannenden Baums mit minimalem Gewicht nahe ([G/D]):

Es sei (U, W) eine disjunkte Zerlegung der Knotenmenge eines gewichteten ungerichteten Graphen $G = (V, E, w)$, d.h. $U \cup W = V$ und $U \cap W = \emptyset$. Die Kante $(u, w) \in E$ besitze minimales Gewicht unter allen Kanten $\{(u', w') \mid u' \in U \text{ und } w' \in W\}$. Dann gibt es einen aufspannenden Baum mit minimalem Gewicht in G, der (u, w) enthält.

Im folgenden wird vorausgesetzt, dass der Graph G zusammenhängend ist. Dann kann folgender Ansatz gewählt werden:

Zunächst wird $T := \{(u, w)\}$ gesetzt; hierbei ist $(u, w) \in E$ eine Kante mit minimalem Gewicht unter allen Kanten in E. T ist also ein aufspannender Teilbaum in G mit minimalem Gewicht. Für jeden Knoten $v \in V$, der sich nicht in T befindet, wird ein Knoten $v' \in T$ bestimmt, so dass $(v, v') \in E$ ist und diese Kante minimales Gewicht unter allen Kanten $(v, w) \in E$ mit $w \in T$ besitzt. In einem `ARRAY naechster [1..n]` im Eintrag mit der Nummer des Knotens v wird die Kante (v, v') und ihr Gewicht abgelegt. Eventuell gibt es einen derartigen Knoten $v' \in T$ nicht; in diesem Fall wird eine beliebige Knotennummer aus T genommen und

das Kantengewicht ∞ eingetragen. Für alle Knoten $w \in T$ wird im `ARRAY naechster` an der entsprechenden Position ein Dummy-Wert eingetragen, der kennzeichnet, dass $w \in T$ gilt. Im `ARRAY naechster` wird eine Kante (u, w) mit minimalem Gewicht bestimmt; dabei werden nur Kanten (u, w) betrachtet, für die $w \in T$ und $u \notin T$ ist. Diese Kante wird zu T hinzugefügt und alle Einträge im `ARRAY naechster` aktualisiert. Das Verfahren endet, wenn T $n-1$ Kanten enthält.

Zur Implementierung des Verfahrens wird wieder die `UNIT Graph` aus Kapitel 5.2.3 erweitert. Der Objekttyp `TGraph` aus dieser Unit wird in seinem `PRIVATE`-Teil um einen Pointer `spanning_tree` erweitert, der auf eine Liste zeigt, die die Kanten eines aufspannenden Baums mit minimalem Gewicht aufnehmen:

```
spanning_tree  : PListe; { Kanten eines aufspannenden Baums
                           mit minimalem Gewicht              }
```

Diese Komponente wird im Konstruktor des Objekttyps `TGraph` initialisiert bzw. entsprechend im Destruktor behandelt.

Im `PUBLIC`-Teil wird der Objekttyp um eine Methode `TGraph.span_tree` mit folgender Benutzerschnittstelle erweitert:

Methode	Bedeutung
`FUNCTION TGraph.span_tree : INTEGER;`	Die Funktion ermittelt das minimale Gewicht eines aufspannenden Baums im Graphen (und legt sie in einer Liste ab, die durch den Pointer `spanning_tree` im `PRIVATE`-Teil der Unit angesprochen werden kann bzw. zeigt die Ergebnisse an).

Der Code der Methode `TGraph.span_tree` lautet:

```
FUNCTION TGraph.span_tree : INTEGER;

VAR naechster : PVektor;
    OK        : BOOLEAN;
    idx       : Knotenbereich;
    jdx       : Knotenbereich;
    i_min     : Knotenbereich;
    j_min     : Knotenbereich;
    w_min     : INTEGER;
    p         : PKante;
    minimum   : INTEGER;

BEGIN { TGraph.span_tree }
  OK        := FALSE;
```

```
span_tree := -1;
IF spanning_tree <> NIL
THEN spanning_tree^.for_all (Alle_Kanten_entfernen)
ELSE New (spanning_tree, init);

New (naechster, init (n, OK));
IF NOT OK THEN Exit;

w_min := unendlich;

{ Kante mit minimalem Gewicht bestimmen und in T einfügen }
FOR idx := 1 TO n DO
  FOR jdx := 1 TO idx-1 DO
    IF Adjazenzmatrix^.m(idx, jdx) <= w_min
    THEN BEGIN
           w_min := Adjazenzmatrix^.m(idx, jdx);
           i_min := idx;
           j_min := jdx;
         END;
New (p, init (i_min, j_min, w_min));
spanning_tree^.insert (p);

{ Initialisierung des Vektors naechster }
FOR idx := 1 TO n DO
  IF Adjazenzmatrix^.m(idx, i_min) < Adjazenzmatrix^.m(idx, j_min)
  THEN  naechster^.setze(idx, i_min)
  ELSE  naechster^.setze(idx, j_min);

naechster^.setze(i_min, 0);
naechster^.setze(j_min, 0);

{ n-2 weitere Kanten für T bestimmen }
FOR idx := 2 TO n-1 DO
  BEGIN
    { Bestimmung eines Knotens j_min mit
      naechster.v(j_min) <> 0 und minimalem Gewicht
      der Kante (j_min, naechster.v(j_min))         }
    minimum := unendlich;
    FOR jdx := 1 TO n DO
      IF naechster^.v(jdx) <> 0
      THEN BEGIN
             IF Adjazenzmatrix^.m(jdx, naechster^.v(jdx))
                                                  < minimum
             THEN BEGIN
                    j_min   := jdx;
                    minimum
```

```
                        := Adjazenzmatrix^.m(jdx, naechster^.v(jdx));
                       END;
              END;

      { Aufnahme der neuen Kante in T }
      New (p, init (j_min, naechster^.v(j_min), minimum));
      spanning_tree^.insert (p);

      { Gewicht addieren }
      w_min := w_min + minimum;

      { Den Vektor naechster aktualisieren }
      naechster^.setze(j_min, 0);
      FOR jdx := 1 TO n DO
        IF naechster^.v(jdx) <> 0
        THEN BEGIN
               IF Adjazenzmatrix^.m(jdx, naechster^.v(jdx))
                                  > Adjazenzmatrix^.m(jdx, j_min)
               THEN naechster^.setze(jdx, j_min);
              END;
    END;

  span_tree := w_min;

  -- verarbeite das Feld spanning_tree^, z.B. Ausgabe der Werte

  Dispose (naechster, done);
END   { TGraph.span_tree };
```

Enthält der Graph n Knoten, so ist die Anzahl ausgeführter Anweisungen des Verfahrens offensichtlich von der Ordnung $O(n^2)$. Da potentiell alle Kanten des Graphen betrachtet werden müssen (denn jede Kante könnte als mögliche Kante in einem aufspannenden Baum vorkommen) und der Graph im allgemeinen $O(n^2)$ viele Kanten enthalten kann, ist das Verfahren zunächst nicht zu verbessern.

Häufig liegt jedoch ein Graph vor, der bedeutend weniger Kanten enthält. In diesem Fall ist ein Kanten-orientierter Lösungsansatz vorzuziehen, der ebenfalls nach der Greedy-Methode vorgeht und auf obigem Satz basiert. Das Verfahren ist in folgendem Pseudocode beschrieben:

```
FUNCTION TGraph.span_tree_Kanten : INTEGER;

BEGIN { TGraph.span_tree_Kanten }
```

```
    T := ∅;
    S := E { Kantenmenge von G };
    Sortiere S nach aufsteigendem Kantengewicht;

    WHILE T enthält weniger als n - 1 Kanten DO
     BEGIN
       wähle eine Kante (v_i, v_j) ∈ S mit minimalem Gewicht;
       S := S \ {(v_i, v_j)};
       IF (v_i, v_j) erzeugt keinen Zyklus in T
       THEN   T := T ∪ {(v_i, v_j)}
     END { WHILE };

  END { TGraph.span_tree_Kanten };
```

Das Verfahren baut schrittweise einen aufspannenden Baum auf, wobei im allgemeinnen zunächst isolierte Teile des Baums („Äste“) entstehen. Diese werden im Laufe des Verfahrens durch Hinzunahme weiterer Kanten zu größeren Ästen zusammengesetzt. Dabei wird darauf geachtet, dass durch die Hinzunahme einer Kante kein Zyklus entsteht. Es wird daher ein effizientes Verfahren benötigt, das festzustellen erlaubt, ob durch die Hinzunahme einer Kante ein Zyklus in T entsteht. Die Knoten eines jeden isolierten Asts in T bilden eine Teilmenge der Knotenmenge von G; verschiedene Äste stellen disjunkte Teilmengen dar. Durch die Hinzunahme einer Kante (v_i, v_j) entsteht genau dann ein Zyklus, wenn sowohl v_i als auch v_j im selben Ast liegen; kein Zyklus entsteht, wenn sie in unterschiedlichen Ästen (Teilmengen) liegen. Es bietet sich daher an, die Äste in T in einer Union-Find-Struktur (siehe Kapitel 5.1.3) zu organisieren. Die Überprüfung der Bedingung in der `IF`-Anweisung bedeutet also die Durchführung einer ***find***-Operation, genauer: kein Zyklus entsteht, wenn ***find***$(v_i) \neq$ ***find***(v_j) gilt. Nach Hinzunahme der Kante (v_i, v_j) werden dann die durch ***find***(v_i) und ***find***(v_j) bestimmten Äste miteinander vereinigt (***union***-Operation).

Der Graph enthalte m viele Kanten. Der Aufwand zur anfänglichen Sortierung der Menge S ist von der Ordnung $O(m \cdot \log(m))$. Die `WHILE`-Schleife wird höchstens m-mal durchlaufen; in jedem Durchlauf wird eine Kante aus S entfernt und eventuell in T eingefügt. Aus den Komplexitätsbetrachtungen aus Kapitel 5.1.3 ergibt sich damit ein Gesamtaufwand für die `WHILE`-Schleife von der Ordnung $O(m \cdot \log(m))$. Der Gesamtaufwand ist also von der Ordnung $O(m \cdot \log(m))$.

Enthält der Graph also deutlich weniger als die maximal mögliche Anzahl an Kanten, etwa $m = c \cdot n^{1+\varepsilon} << n^2$ viele Kanten, so ist der Aufwand dieses Verfahrens von der Ordnung $O(n^{1+\varepsilon} \cdot \log(n))$ und damit dem oben beschriebene Verfahren mit der Funktion `span_tree`

vorzuziehen. Allerdings ist die Implementierungskomplexität wegen der eingesetzten Objekttypen höher.

Die Implementierungsdetails werden als Programmierübung vorgeschlagen.

7.3 Dynamische Programmierung

Die Methode der Dynamischen Programmierung stellt eine Verallgemeinerung des Divide-and-Conquer-Prinzips dar. Es ist auch dann anwendbar, wenn die Teilprobleme bei der Zerlegung nicht disjunkt sind, sondern wiederum gemeinsame Teilprobleme enthalten.

Dynamische Programmierung (für Optimierungsaufgaben):

Wie beim Divide-and-Conquer-Prinzip wird ein Problem wird in mehrere kleinere (nun nicht notwendigerweise disjunkte) Teilprobleme aufgeteilt. Diese werden gelöst und aus deren Lösung eine Lösung für das Ausgangsproblem konstruiert. Dabei wird jedes Teilproblem nur einmal gelöst und das Ergebnis in einer Tabelle so lange aufbewahrt, wie es eventuell später noch benötigt wird. Wiederholungen von Berechnungen werden so vermieden.

Die dynamische Programmierung ist dann anwendbar, wenn für das zu lösende Problem das folgende **Optimalitätsprinzip** gilt:

Jede Teillösung einer optimalen Lösung, die Lösung eines Teilproblems ist, ist selbst eine optimale Lösung des betreffenden Teilproblems.

Man geht nach folgenden Regeln vor:

1. Das zu lösende Problem wird rekursiv beschrieben.

2. Es wird die Menge **K** der kleineren Probleme bestimmt, auf die bei Lösung des Problems direkt oder indirekt zugegriffen wird.

3. Es wird eine Reihenfolge $P_1, ..., P_r$ der Probleme in **K** festgelegt, so dass bei der Lösung von Problemen P_l $(1 \leq l \leq r)$ nur auf Probleme P_k mit Index k kleiner als l zugegriffen wird.

4. Die Lösungen für $P_1, ..., P_r$ werden in dieser Reihenfolge berechnet und gespeichert, wobei einmal berechnete Lösungen solange gespeichert werden, wie sie für später noch zu berechnende Problemen direkt oder indirekt benötigt werden.

Die Methode der dynamischen Programmierung ist durch eine mathematisch orientierten Betrachtungsweise von Algorithmen geprägt und trägt damit Züge einer formalisierten Vorgehensweise. Sie wird in den folgenden Unterkapiteln an einigen wichtigen Beispielen erläutert.

7.3.1 Problem des Handlungsreisenden 1. Lösungsansatz

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem (minimum traveling salesperson problem)

Problem-instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Eine Tour $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ durch G ist eine geschlossene Kantenfolge, in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt. Für eine Tour T durch G ist das Gewicht der Tour definiert durch $m(G,T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$.

Gesuchte Lösung: Eine Tour T^*, die minimales Gewicht (unter allen möglichen Touren durch G) besitzt, und $m(G, T^*)$.

Im folgenden wird $v_{i_1} = v_1$ gesetzt.

Um die Methode der Dynamischen Programmierung anzuwenden, wird das Problem rekursiv beschrieben (Regel 1.):

Das Problem wird allgemein mit $\Pi(v_i, S)$ bezeichnet. Dabei ist $v_i \in V$ und $S \subseteq V$ mit $v_i \notin S$ und $v_1 \notin S$. $\Pi(v_i, S)$ beschreibt das Problem der Bestimmung eines Weges mit minimalem Gewicht von v_i durch alle Knoten von S nach v_1. Das für $\Pi(v_i, S)$ ermittelte (optimale) Gewicht sei $g(v_i, S)$. Eventuell ist $g(v_i, S) = \infty$.

Gesucht wird eine Lösung von $\Pi(v_1, V - \{v_1\})$ bzw. $g(v_1, V - \{v_1\})$.

Eine rekursive Problembeschreibung ergibt sich aus der Überlegung, dass eine gewichtsoptimale Tour für das Problem $\Pi(v_i, S)$ mit einer Kante $(v_i, v_j) \in E$ mit $v_j \in S$ beginnt und dann aus einem gewichtsoptimalen Weg von v_j durch alle Knoten von $S \setminus \{v_j\}$ zum Knoten v_1 besteht. Daher gelten die Rekursionsgleichungen für $g(v_i, S)$:

$$g(v_i, S) = \min\{ w((v_i, v_j)) + g(v_j, S - \{v_j\}) \mid v_j \in S \} \text{ für } v_i \notin S \text{ und } v_1 \notin S$$
$$g(v_i, \emptyset) = w((v_i, v_1)) \text{ für } i = 2, ..., n.$$

Diese Rekursionsformeln legen die Berechnungsreihenfolge fest. Die Menge der kleineren Probleme, auf die zur Lösung von $\Pi(v_1, V - \{v_1\})$ zugegriffen wird (Regel 2.), sind die Probleme $\Pi(v_i, S)$ mit $|S| < n - 1$, und zwar sortiert nach aufsteigender Mächtigkeit von S (Regel 3.):

Teilproblem (Berechnung in aufsteigender Reihenfolge)	Optimaler Wert der Zielfunktion
$\Pi(v_i, \emptyset)$ für $i = 2, ..., n$	$g(v_i, \emptyset)$ für $i = 2, ..., n$
$\Pi(v_i, S)$ mit $\lvert S\rvert = 1$, $v_i \notin S$, $v_1 \notin S$	$g(v_i, S)$ mit $\lvert S\rvert = 1$, $v_i \notin S$, $v_1 \notin S$
$\Pi(v_i, S)$ mit $\lvert S\rvert = 2$, $v_i \notin S$, $v_1 \notin S$	$g(v_i, S)$ mit $\lvert S\rvert = 2$, $v_i \notin S$, $v_1 \notin S$
...	...
$\Pi(v_i, S)$ mit $\lvert S\rvert = n - 1$, $v_i \notin S$, $v_1 \notin S$, d.h. $\Pi(v_1, V - \{v_1\})$	$g(v_i, S)$ mit $\lvert S\rvert = n - 1$, $v_i \notin S$, $v_1 \notin S$, d.h. $g(v_1, V - \{v_1\})$

Das folgende Beispiel zeigt die Bezeichnungen, hier nur angegeben für die Zielfunktion:

Gegeben sei der folgende Graph $G = (V, E, w)$ mit $n = 4$ Knoten durch seine Adjazenzmatrix

$$A(G) = \begin{bmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{bmatrix}.$$

Das Verfahren läuft in $(n-1)+1 = 4$ Iterationen ab. Zuerst werden die Werte $g(v_i, \emptyset)$ für $i = 2, ..., n$ bestimmt (1. Iteration):

$g(v_2, \emptyset) = w((v_2, v_1)) = 5$, $\quad g(v_3, \emptyset) = w((v_3, v_1)) = 6$, $\quad g(v_4, \emptyset) = w((v_4, v_1)) = 8$.

Zur Berechnung von $g(v_i, S)$ mit $|S| = 1$, $v_i \notin S$, $v_1 \notin S$ (2. Iteration) wird nur auf die in der vorherigen Iteration berechneten Werte zurückgegriffen:

$g(v_1, \{v_2\}) = w((v_1, v_2)) + g(v_2, \emptyset) = 15$, $\quad g(v_1, \{v_3\}) = w((v_1, v_3)) + g(v_3, \emptyset) = 21$,

$g(v_1, \{v_4\}) = w((v_1, v_4)) + g(v_4, \emptyset) = 28$,

$g(v_2, \{v_3\}) = w((v_2, v_3)) + g(v_3, \emptyset) = 15$, $\quad g(v_2, \{v_4\}) = w((v_2, v_4)) + g(v_4, \emptyset) = 18$,

$g(v_3, \{v_2\}) = w((v_3, v_2)) + g(v_2, \emptyset) = 18$, $\quad g(v_3, \{v_4\}) = w((v_3, v_4)) + g(v_4, \emptyset) = 20$,

$g(v_4, \{v_2\}) = w((v_4, v_2)) + g(v_2, \emptyset) = 13$, $\quad g(v_4, \{v_3\}) = w((v_4, v_3)) + g(v_3, \emptyset) = 15$.

Falls man nur am Wert der Zielfunktion einer optimalen Tour interessiert ist, werden nach Ausführung der 2. Iteration die Werte der 1. Iteration nicht mehr benötigt. Möchte man eine optimale Tour selbst bestimmen, müssen in diesem Verfahren die berechneten Werte aller Iterationen zwischengespeichert werden (siehe unten).

Die 3. Iteration greift nur auf die berechneten Werte der 2. Iteration und nicht auf die Werte der 1. Iteration zurück:

$g(v_1, \{v_2, v_3\}) = \min\{w((v_1, v_2)) + g(v_2, \{v_3\}),\ w((v_1, v_3)) + g(v_3, \{v_2\})\} = 25$,

$g(v_1, \{v_2, v_4\}) = \min\{w((v_1, v_2)) + g(v_2, \{v_4\}),\ w((v_1, v_4)) + g(v_4, \{v_2\})\} = 28$,

$g(v_1, \{v_3, v_4\}) = \min\{w((v_1, v_3)) + g(v_3, \{v_4\}),\ w((v_1, v_4)) + g(v_4, \{v_3\})\} = 35$

$g(v_2, \{v_3, v_4\}) = \min\{w((v_2, v_3)) + g(v_3, \{v_4\}),\ w((v_2, v_4)) + g(v_4, \{v_3\})\} = 25$,

$g(v_3, \{v_2, v_4\}) = \min\{w((v_3, v_2)) + g(v_2, \{v_4\}),\ w((v_3, v_4)) + g(v_4, \{v_2\})\} = 25$,

$g(v_4, \{v_2, v_3\}) = \min\{w((v_4, v_2)) + g(v_2, \{v_3\}),\ w((v_4, v_3)) + g(v_3, \{v_2\})\} = 23$.

Analog greift die 4. Iteration nur auf die berechneten Werte der 3. Iteration und nicht auf die Werte der vorhergehenden Iterationen zurück:

$$g(v_1, \{v_2, v_3, v_4\}) = \min\left\{\begin{matrix} w((v_1, v_2)) + g(v_2, \{v_3, v_4\}),\ w((v_1, v_3)) + g(v_3, \{v_2, v_4\}), \\ w((v_1, v_4)) + g(v_4, \{v_2, v_3\}) \end{matrix}\right\} = 35.$$

Mit Hilfe Pascal-ähnlichem Pseudocode lässt sich diese Berechnung wie folgt ausdrücken:

```
BEGIN
  { 1. Iteration: }
  FOR i := 2 TO n DO
     g(v_i,∅) := w((v_i,v_1));

  { restliche Iterationen: }
  FOR k := 1 TO n−1 DO
    FOR S ⊆ {v_2,...,v_n} AND (|S| = k) DO
      FOR i := 1 TO n DO
        IF v_i ∉ S THEN g(v_i,S) = min{ w((v_i,v_j)) + g(v_j,S − {v_j}) | v_j ∈ S };
END;
```

Ein Vorteil der Methode der Dynamischen Programmierung wird deutlich: zur Berechnung der Werte einer Iteration brauchen nur die Werte der vorhergehenden Iteration gespeichert zu werden (in ihnen sind implizit die Zwischenergebnisse aus allen vorherigen Iterationen enthalten). Um jedoch eine optimale Tour aus der Berechnung von $g(v_1, V - \{v_1\})$ zu ermitteln, werden alle berechneten Werte $g(v_i, S)$ benötigt:

Eine optimale Tour mit Gewicht $g(v_1, V - \{v_1\})$ startet in $v_1 = v_{j_1}$ und geht zu demjenigen Knoten v_{j_2}, der zum Minimum bei der Ermittlung von $g(v_1, V - \{v_1\})$ führte. Das bedeutet, dass $g(v_1, V - \{v_1\}) = w((v_1, v_{j_2})) + g(v_{j_2}, V - \{v_1, v_{j_2}\})$ ist. Anschließend bestimmt man den Knoten v_{j_3}, der zum minimalen Wert in $g(v_{j_2}, V - \{v_1, v_{j_2}\})$ Anlass gab, d.h. für den $g(v_{j_2}, V - \{v_1, v_{j_2}\}) = w((v_{j_2}, v_{j_3})) + g(v_{j_3}, V - \{v_1, v_{j_2}, v_{j_3}\})$ ist. Das Verfahren wird fortgesetzt, bis man einen Knoten v_{j_n} gefunden hat, der zum minimalen Wert $g(v_{j_{n-1}}, \{v_{j_n}\}) = w((v_{j_{n-1}}, v_{j_n})) + g(v_{j_n}, \varnothing)$ führt.

Das Verfahren kann an obigen Beipiel nachvollzogen werden. Eine optimale Tour startet in $v_1 = v_{j_1}$ und führt zu $v_{j_2} = v_2$, da

$$\begin{aligned} g(v_1, \{v_2, v_3, v_4\}) &= \min \left\{ \begin{matrix} w((v_1, v_2)) + g(v_2, \{v_3, v_4\}), & w((v_1, v_3)) + g(v_3, \{v_2, v_4\}), \\ w((v_1, v_4)) + g(v_4, \{v_2, v_3\}) & \end{matrix} \right\} \\ &= w((v_1, v_2)) + g(v_2, \{v_3, v_4\}) = 35 \end{aligned}$$

ist. Sie führt weiter über v_4 und v_3 zu v_1. Im folgenden sind noch einmal die berechneten Werte $g(v_i, S)$ aufgeführt und diejenigen Werte markiert, die zu einer optimalen Tour gehören:

$$g(v_2,\varnothing)=w((v_2,v_1))=5,\qquad g(v_3,\varnothing)=w((v_3,v_1))=6,\qquad g(v_4,\varnothing)=w((v_4,v_1))=8,$$

$$g(v_1,\{v_2\})=w((v_1,v_2))+g(v_2,\varnothing)=15,\qquad g(v_1,\{v_3\})=w((v_1,v_3))+g(v_3,\varnothing)=21,$$
$$g(v_1,\{v_4\})=w((v_1,v_4))+g(v_4,\varnothing)=28,$$
$$g(v_2,\{v_3\})=w((v_2,v_3))+g(v_3,\varnothing)=15,\qquad g(v_2,\{v_4\})=w((v_2,v_4))+g(v_4,\varnothing)=18,$$
$$g(v_3,\{v_2\})=w((v_3,v_2))+g(v_2,\varnothing)=18,\qquad g(v_3,\{v_4\})=w((v_3,v_4))+g(v_4,\varnothing)=20,$$
$$g(v_4,\{v_2\})=w((v_4,v_2))+g(v_2,\varnothing)=13,\qquad g(v_4,\{v_3\})=\mathbf{w((v_4,v_3))+g(v_3,\varnothing)}=15,$$

$$g(v_1,\{v_2,v_3\})=\min\{w((v_1,v_2))+g(v_2,\{v_3\}),\ w((v_1,v_3))+g(v_3,\{v_2\})\}=25,$$
$$g(v_1,\{v_2,v_4\})=\min\{w((v_1,v_2))+g(v_2,\{v_4\}),\ w((v_1,v_4))+g(v_4,\{v_2\})\}=28,$$
$$g(v_1,\{v_3,v_4\})=\min\{w((v_1,v_3))+g(v_3,\{v_4\}),\ w((v_1,v_4))+g(v_4,\{v_3\})\}=35$$
$$g(v_2,\{v_3,v_4\})=\min\{w((v_2,v_3))+g(v_3,\{v_4\}),\ \mathbf{w((v_2,v_4))+g(v_4,\{v_3\})}\}=25,$$
$$g(v_3,\{v_2,v_4\})=\min\{w((v_3,v_2))+g(v_2,\{v_4\}),\ w((v_3,v_4))+g(v_4,\{v_2\})\}=25,$$
$$g(v_4,\{v_2,v_3\})=\min\{w((v_4,v_2))+g(v_2,\{v_3\}),\ w((v_4,v_3))+g(v_3,\{v_2\})\}=23,$$

$$g(v_1,\{v_2,v_3,v_4\})=\min\left\{\begin{array}{l}\mathbf{w((v_1,v_2))+g(v_2,\{v_3,v_4\})},\ w((v_1,v_3))+g(v_3,\{v_2,v_4\}),\\ w((v_1,v_4))+g(v_4,\{v_2,v_3\})\end{array}\right\}=35.$$

Zur Abschätzung der (worst-case-) Zeitkomplexität des Verfahrens wird zunächst die Anzahl möglicher Werte $g(v_i,S)$ für $v_i\notin S$ und $v_1\notin S$ berechnet, wenn der Graph *n* Knoten besitzt:

Für $g(v_i,\varnothing)$ mit $i=2,\ldots,n$ gibt es $n-1$ Möglichkeiten. Für $g(v_1,S)$ mit $|S|=k$, $1\le k\le n-2$ und $v_1\notin S$ gibt es $\binom{n-1}{k}$ Möglichkeiten; für $g(v_i,S)$ mit $|S|=k$, $v_i\ne v_1$, $1\le k\le n-2$, $v_i\notin S$ und $v_1\notin S$ gibt es $(n-1)\cdot\binom{n-2}{k}$ Möglichkeiten; schließlich ist noch $g(v_i,S)$ mit $|S|=n-1$, $v_i\notin S$, $v_1\notin S$, d.h. $g(v_1,V-\{v_1\})$ zu bestimmen. Insgesamt werden

$$\begin{aligned}n-1+\sum_{k=1}^{n-2}\binom{n-1}{k}+(n-1)\cdot\sum_{k=1}^{n-2}\binom{n-2}{k}+1&=n-1+\sum_{k=0}^{n-1}\binom{n-1}{k}-1+(n-1)\cdot\sum_{k=0}^{n-2}\binom{n-2}{k}-(n-1)\\&=2^{n-1}+(n-1)\cdot 2^{n-2}-1\\&=(n+1)\cdot 2^{n-2}-1\end{aligned}$$

berechnet.

Zur Bestimmung von $g(v_i,\varnothing)$ mit $i=2,\ldots,n$ sind keine Additionen oder Vergleiche erforderlich. Bei der Berechnung von $g(v_i,S)$ mit $|S|=k$, $1\le k\le n-1$, $v_i\notin S$ werden jeweils *k*

Additionen ausgeführt. Insgesamt sind dieses $\sum_{k=1}^{n-1} k \cdot \binom{n-1}{k} + (n-1) \cdot \sum_{k=1}^{n-2} k \cdot \binom{n-2}{k}$ Additionen.

Unter Verwendung der Identität $k \cdot \binom{n-1}{k} = (n-1) \cdot \frac{(n-2)!}{(k-1)! \cdot (n-1-k)!} = (n-1) \cdot \binom{n-2}{k-1}$ ergibt sich für die Anzahl ausgeführter Additionen der Wert

$$\begin{aligned}\sum_{k=1}^{n-1} k \cdot \binom{n-1}{k} + (n-1) \cdot \sum_{k=1}^{n-2} k \cdot \binom{n-2}{k} &= (n-1) \cdot \sum_{k=0}^{n-2} \binom{n-2}{k} + (n-1) \cdot (n-2) \cdot \sum_{k=0}^{n-3} \binom{n-3}{k} \\ &= (n-1) \cdot 2^{n-2} + (n-1) \cdot (n-2) \cdot 2^{n-3} \\ &= n \cdot (n-1) \cdot 2^{n-3}\end{aligned}$$

von der Ordnung $O(n^2 \cdot 2^n)$. Die Anzahl der Vergleiche zur Bestimmung der Minima in $g(v_i, S)$ ist von derselben Ordnung.

Ein Problem beim Einsatz der Methode der Dynamischen Programmierung besteht darin, eine geeignete Datenstruktur zu finden, um die berechneten Ergebnisse kleinerer Probleme so lange zwischenzuspeichern, wie sie benötigt werden, bzw. um auf diese Ergebnisse in effizienter Weise zuzugreifen. Man könnte versuchen, die Möglichkeiten zu nutzen, die die eingesetzte Programmiersprache in Form rekursiver Prozeduraufrufe bietet.

Zur Berechnung der Funktion $g(v_i, S)$ könnte man etwa folgenden (Pseudo-) Code verwenden:

```
FUNCTION G(v_i,S): REAL;

  BEGIN { G(v_i,S) }
    IF S = ∅ THEN G(v_i,S) := w((v_i,v_1))
             ELSE G(v_i,S) := min{ w((v_i,v_j)) + G(v_j, S - {v_j}) | v_j ∈ S };
  END   { G(v_i,S) };
```

Das obige Beispiel hat folgende ineinander geschachtelte Aufruffolge:

Aufruf von $G(v_1, \{v_2, v_3, v_4\})$

Aufruf von $G(v_2, \{v_3, v_4\})$

Aufruf von $G(v_3, \{v_4\})$

Aufruf von $G(v_4, \varnothing)$

Aufruf von $G(v_4, \{v_3\})$

Aufruf von $G(v_3, \varnothing)$

Aufruf von $G(v_3, \{v_2, v_4\})$

Aufruf von $G(v_2, \{v_4\})$
Aufruf von $G(v_4, \varnothing)$
Aufruf von $G(v_4, \{v_2\})$
Aufruf von $G(v_2, \varnothing)$

Aufruf von $G(v_4, \{v_2, v_3\})$
Aufruf von $G(v_2, \{v_3\})$
Aufruf von $G(v_3, \varnothing)$
Aufruf von $G(v_3, \{v_2\})$
Aufruf von $G(v_2, \varnothing)$.

Die gesamte Aufrufabfolge kann als Baum dargestellt werden, dessen Knoten mit einem Aufruf der Funktion G markiert sind. Auf Niveau 0 steht dabei der Aufruf $G(v_1, V - \{v_1\})$, der $n-1$ untergeordnete Aufrufe $G(v_j, S - \{v_j\})$ mit $v_j \in S$ auf Niveau 1 auslöst. Jeder Aufruf $G(v_j, S)$ mit $|S| = n-2$ auf Niveau 1 löst $n-2$ Aufrufe der Form $G(v_k, T)$ mit $|T| = n-3$ auf Niveau 2 aus. Niveau 2 enthält also insgesamt $(n-1)\cdot(n-2)$ Aufrufe. Allgemein werden von den $(n-1)\cdot(n-2)\cdot\ldots\cdot(n-k)$ Aufrufen der Form $G(v_j, S)$ mit $|S| = n-k-1$ Niveau k jeweils $n-k-1$ Aufrufe der Form $G(v_k, T)$ mit $|T| = n-k-2$ auf Niveau $k+1$ ausgelöst. Die Abarbeitung von $G(v_l, \varnothing)$ auf Niveau $k = n-1$ löst keine weiteren Aufrufe aus, so dass der Baum der gesamten Aufruffolge insgesamt $\sum_{k=0}^{n-1} k! \cdot \binom{n-1}{k}$ Knoten enthält. Mit jedem Knoten (Aufruf der Funktion G) mit Ausnahme der Knoten auf Niveaus 0 und $n-1$ ist eine Addition verbunden. Man sieht also, dass insgesamt $\sum_{k=1}^{n-2} k! \cdot \binom{n-1}{k}$ Additionen ausgeführt werden. Durch vollständige Induktion über n lässt sich leicht zeigen, dass dieser Wert für $n \geq 7$ größer ist als die Anzahl der Additionen bei Verwendung der Dynamischen Programmierung, nämlich größer als $n\cdot(n-1)\cdot 2^{n-3}$. Im Zeitbedarf ist also die Dynamische Programmierung der rekursiven Implementierung überlegen (wenn auch immer noch exponentiell; ein besseres Ergebnis ist jedoch wegen der NP-vollständigkeit des Handlungsreisenden-Entscheidungsproblems nicht zu erwarten).

Der Speicherplatzbedarf der rekursiven Implementierung ist proportional zur Höhe des beschriebenen Aufrufbaums, also von der Ordnung $O(n)$. Der Speicherplatzbedarf bei Dynamischer Programmierung wird bestimmt durch die maximale Anzahl zu berechnender Werte $g(v_i, S)$ in den jeweiligen Iterationen; hierbei müssen mindestens die Werte einer Iteration bis zur Bestimmung der nächsten Iteration gespeichert werden. Die Anzahl zu berechnender Wer-

te $g(v_i, S)$ einer Iteration mit $|S| = k$ beträgt $(n-1)\cdot\binom{n-2}{k}+\binom{n-1}{k}$. Dieser Wert ist maximal für $k \approx n/2$. Wegen $\binom{2n}{n} = \frac{(2n)!}{n!\cdot n!} = 2^n \cdot \frac{1\cdot 3\cdot \ldots \cdot (2n-1)}{n!} > 2^n$ ist der Speicherplatz bei Dynamischer Programmierung also mindestens von der Ordnung $O(n \cdot 2^n)$.

7.3.2 Alle Wege minimalen Gewichts in gerichteten Graphen

Die Methode der Dynamischen Programmierung führt nicht in jedem Fall auf Algorithmen mit exponentiellem Laufzeitverhalten, wie folgendes Beispiel zeigt.

Das Problem der Wege mit minimalem Gewicht in gerichteten Graphen zwischen allen Knotenpaaren

Problem-
instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}$ gibt jeder Kante $e \in E$ ein Gewicht mit der Eigenschaft, dass keine Zyklen mit negativem Gewicht entstehen.

Gesuchte
Lösung: Die minimalen Gewichte $d_{i,j}$ der Wege vom Knoten v_i zum Knoten v_j des Graphs für $i = 1, ..., n$ und $j = 1, ..., n$.

Bemerkung: Es sind jetzt auch Instanzen mit Gewichtungen zugelassen, die negativ sind. Jedoch dürfen jedoch keine Zyklen mit negativem Gewicht vorkommen.

Der folgende Algorithmus realisiert das Prinzip der Dynamischen Programmierung. Dazu wird das Problem, nämlich für jedes Knotenpaar einen Weg mit minimalem Gewicht zu bestimmen, wobei alle Knoten des Graphs als Zwischenknoten zugelassen sind, in kleinere Probleme aufgeteilt. Dazu wird $A^k(i, j)$ für $k = 1, ..., n$ als das minimale Gewicht eines Wegs vom Knoten v_i zum Knoten v_j definiert, der nur Zwischenknoten aus $\{v_1, ..., v_k\}$ enthält, also keine Zwischenknoten mit einer Nummer, die größer als k ist. Gesucht ist $A^n(i, j) = d_{i,j}$.

Das Problem der Bestimmung von $A^n(i, j)$ wird aufgeteilt in die Bestimmung der kleineren Probleme der Bestimmung von $A^k(i, j)$ für $k = 1, ..., n$ (Regel 2.), und zwar in dieser Reihenfolge (Regel 3.).

Ein Weg mit minimalem Gewicht vom Knoten v_i zum Knoten v_j definiert, der nur Zwischenknoten aus $\{v_1, ..., v_k\}$ enthält, geht entweder nicht durch den Knoten v_k (dann ist $A^k(i, j) = A^{k-1}(i, j)$) oder er führt durch v_k; in diesem Fall ist $A^k(i, j) = A^{k-1}(i,k) + A^{k-1}(k, j)$, da der Graph keine Zyklen mit negativem Gewicht enthält und damit ein Weg mit minimalem Gewicht vom Knoten v_i zum Knoten v_j den Knoten v_k nur einmal durchläuft.

Insgesamt ist

$$A^k(i, j) = \min\{A^{k-1}(i, j),\ A^{k-1}(i,k) + A^{k-1}(k, j)\} \text{ für } k = 1, ..., n$$
$$\text{mit } A^0(i, j) = w((v_i, v_j)).$$

Das Verfahren läuft in n Iterationen ab. In der k-ten Iteration wird die Matrix $[A^k(i, j)]_{(n,n)}$ bestimmt. Dabei kann für jedes k derselbe Speicherplatz verwendet werden, da zur Berechnung eines Eintrags $A^k(i, j)$ entweder der dort bereits stehende Wert $A^{k-1}(i, j)$ genommen oder auf Werte der Form $A^{k-1}(i,k)$ und $A^{k-1}(k, j)$ zugegriffen wird. Dabei kommt es nicht darauf an, ob diese Werte noch aus der vorhergehenden Iteration stammen (also in der laufenden Iteration noch nicht berechnet wurden, d.h. dort wirklich $A^{k-1}(i,k)$ bzw. $A^{k-1}(k, j)$ steht) oder ob diese Werte bereits in der laufenden Iteration neu berechnet wurden. Denn ein Weg mit minimalem Gewicht vom Knoten v_i zum Knoten v_k definiert, der nur Zwischenknoten aus $\{v_1, ..., v_{k-1}\}$ enthält, ist ein Weg mit minimalem Gewicht vom Knoten v_i zum Knoten v_k, der nur Knoten aus $\{v_1, ..., v_k\}$ enthält. Das bedeutet $A^{k-1}(i,k) = A^k(i,k)$ bzw. $A^{k-1}(k, j) = A^k(k, j)$.

Algorithmus zur Lösung des Problems der Wege mit minimalem Gewicht in gerichteten Graphen zwischen allen Knotenpaaren:

Durch Ergänzungen in der `UNIT Graph` aus Kapitel 5.2.3 wird der Algorithmus als Methode in den Objekttyp `TGraph` eingebaut. Im `PUBLIC`-Teil des Objekttyps `TGraph` wird eine Prozedur

```
PROCEDURE TGraph.alle_minimalen_Wege;
```

eingebaut:

Methode	Bedeutung
`PROCEDURE TGraph.alle_minimalen_Wege;`	Die Prozedur ermittelt die minimalen Gewichte aller Wege zwischen je zwei Knoten und zeigt diese an.

Die Prozedur legt das Ergebnis in einer dynamisch eingerichteten Matrix ab, auf die ein Verweis

```
alle_min_Wege  : PMatrix;
```

zeigt, der in den `PRIVATE`-Teil des Objekttyps hinzugefügt wird; der Konstruktor bzw. der Destruktor werden durch entsprechenden Initialisierungs- bzw. Freigabecode ergänzt. Das Verfahren setzt die obigen rekursiven Gleichungen um.

```
PROCEDURE TGraph.alle_minimalen_Wege;

VAR idx : Knotenbereich;
    jdx : Knotenbereich;
    kdx : Knotenbereich;

BEGIN { TGraph.alle_minimalen_Wege }
  { Kostenwegematrix initialisieren }
  FOR idx := 1 TO n DO
    FOR jdx := 1 TO n DO
      alle_min_Wege^.setze (idx, jdx, Adjazenzmatrix^.m(idx, jdx));

  { Schleifen auf demselben Knoten entfernen }
  FOR idx := 1 TO n DO
    alle_min_Wege^.setze (idx, idx, 0);

  FOR kdx := 1 TO n DO
    FOR idx := 1 TO n DO
      FOR jdx := 1 TO n DO
        alle_min_Wege^.setze (idx, jdx,
                              min_real(alle_min_Wege^.m(idx, jdx),
                                       alle_min_Wege^.m(idx, kdx)
                                     + alle_min_Wege^.m(kdx, jdx)));

  -- Matrixinhalt ausgeben;

END   { TGraph.alle_minimalen_Wege };
```

Ist $G = (V, E, w)$ eine Eingabeinstanz des Problems der Wege mit minimalem Gewicht zwischen allen Knotenpaaren in gerichteten Graphen mit *n* Knoten, dann liefert das beschriebene

Verfahren mit der Methode `TGraph.alle_minimalen_wege` im Feld `alle_min_Wege` die minimalen Gewichte $d_{i,j}$ der Wege vom Knoten v_i zu allen Knoten v_j des Graphs für i = 1, ..., n und j = 1, ..., n. Die (worst-case-) Zeitkomplexität und der Speicherplatzbedarf des Verfahrens sind von der Ordnung $O(n^3)$.

7.3.3 0/1-Rucksackproblem

Das nächste Beispiel zur Dynamischen Programmierung behandelt ein wichtiges Zuordnungsproblem. Das Verfahren soll hier nur informell beschrieben werden.

Das 0/1-Rucksackproblem als Maximierungsproblem
(maximum 0/1 knapsack problem)

Eingabe-
instanz: $I = (A, M)$

$A = \{a_1, ..., a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{R}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{R}_{>0}$ das Gewicht und $p_i \in \mathbf{R}_{>0}$ den Wert (Profit) des Objekts a_i.

Eine zulässige Lösung ist eine Folge $(x_1, ..., x_n)$ mit

$x_i = 0$ oder $x_i = 1$ für $i = 1, ..., n$ und $\sum_{i=1}^{n} x_i \cdot w_i \le M$.

Für eine zulässige Lösung $(x_1, ..., x_n)$ wird die Zielfunktion definiert durch

$$m(I, (x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$$

Gesuchte
Lösung: Eine zulässige Lösung $(x_1^*, ..., x_n^*)$, für die $m(I, (x_1^*, ..., x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ maximal ist unter allen zulässigen Lösungen.

Die Anzahl an Zeichen, um die Eingabeinstanz darzustellen, d.h. die Problemgröße, ist durch $2 \cdot n \cdot \lceil \log(p_{max} + 1) \rceil$ mit $p_{max} = \max\{\{ p_i \mid i = 1, ..., n \} \cup \{ w_i \mid i = 1, ..., n \}\}$ beschränkt.

Für eine Instanz $I = (A, M)$ wird im folgenden $\sum_{i=1}^{n} w_i > M$ angenommen, denn für $\sum_{i=1}^{n} w_i \le M$ kann man $x_1^* = 1, ..., x_n^* = 1$ nehmen.

Es sei die Eingabeinstanz $I = (A, M)$ mit $A = \{a_1, ..., a_n\}$ gegeben. Das Problem der Ermittlung einer optimalen Lösung für diese Instanz wird in kleinere Probleme zerlegt, die jeweils nur einen Teil der Objekte in *A* und variable Rucksackkapazitäten betrachten. Aus optimalen Lösungen dieser kleineren Probleme wird schrittweise eine optimale Lösung des ursprünglichen Problems zusammengesetzt.

Das ursprüngliche Problem der Ermittlung einer optimalen Lösung für *I* betrachtet alle *n* Elemente und die Rucksackkapazität *M*. Es soll daher mit *RUCK*(*n*, *M*) bezeichnet werden. Werden nur die Objekte $a_1, ..., a_j$ und die Rucksackkapazität *y* betrachtet, so wird dieses kleinere Problem mit *RUCK*(*j*, *y*) bezeichnet. Eine optimale Lösung für *RUCK*(*j*, *y*) habe einen Wert der Zielfunktion, der mit $f_j(y)$ bezeichnet wird.

Gesucht wird also neben $(x_1^*, ..., x_n^*) \in \mathrm{SOL}(I)$ der Wert $m^*(I) = f_n(M)$.

Das zu lösende Problem wird zunächst rekursiv beschrieben (Regel 1.), die Menge der kleineren Probleme bestimmt, auf die bei der Lösung des Problems direkt oder indirekt zugegriffen wird (Regel 2.), und die Reihenfolge festgelegt, in der diese Teilprobleme gelöst werden (Regel 3.):

Umsetzung der Regel 1.:

Zwischen den Problemen *RUCK*(*j*, *y*) mit ihren Zielfunktionswerten $f_j(y)$ besteht folgender (rekursiver) Zusammenhang:

$$\boxed{\begin{array}{l} f_j(y) = \max\{f_{j-1}(y),\ f_{j-1}(y - w_j) + p_j\} \text{ für } j = n, ..., 1, 0 \le y \le M \ \text{(„\textbf{Rückwärtsmethode}“)} \\ f_0(y) = 0 \text{ für } 0 \le y \le M \text{ und } f_0(y) = -\infty \text{ für } y < 0. \end{array}}$$

$f_j(y)$ kann als Funktion von *y* aufgefasst werden. Aus der Kenntnis des (gesamten) Funktionsverlaufs von f_n ist insbesondere $m^*(I) = f_n(M)$ ablesbar. Es zeigt sich (siehe unten), dass aus der Kenntnis des Funktionsverlaufs von f_n an der Stelle *M* auch eine optimale zulässige Lösung $(x_1^*, ..., x_n^*)$ „leicht“ ermittelt werden kann. Daher wird zunächst eine Methode zur Berechnung und Speicherung der Funktionsverläufe $f_j(y)$ beschrieben.

Umsetzung der Regeln 2. und 3.:

Teilproblem (Berechnung in aufsteigender Reihenfolge)	Optimaler Wert der Zielfunktion
$RUCK(0, y),\ \ y \le M$	$f_0(y)$ mit $y \le M$
$RUCK(1, y),\ 0 \le y \le M$	$f_1(y)$ mit $y \le M$
$RUCK(2, y),\ 0 \le y \le M$	$f_2(y)$ mit $y \le M$
...	
$RUCK(n, y),\ 0 \le y \le M$	$f_n(y)$ mit $y \le M$

Zur Berechnung von $f_j(y)$ gemäß der rekursiven Formel

$f_j(y) = \max\{f_{j-1}(y),\ f_{j-1}(y - w_j) + p_j\}$ für $j = n, ..., 1, 0 \le y \le M$

braucht wegen $y \le M$ und $y - w_j \le M$ nur die Funktion $f_{j-1}(y)$ bereitgehalten zu werden. Um jedoch aus f_n an der Stelle M eine optimale Lösung $(x_1^*, ..., x_n^*)$ zu ermitteln, werden alle Funktionen $f_j(y)$ für $j = 0, ..., n$ benötigt.

Man sieht leicht, dass wegen $x_i \in \{0, 1\}$ und $p_i \in \mathbf{R}_{>0}$ die Funktionen $f_j(y)$ Treppenfunktionen sind (Abbildung 7.3.3-1). Zur Algorithmen-gerechten Speicherung des Funktionsverlaufs von $f_j(y)$ brauchen daher nur die Sprungstellen der Funktion gespeichert zu werden.

Es sei S_j für $j = 0, ..., n$ die Menge der Sprungstellen von f_j.

Umsetzung der Regel 4.:

Die algorithmische Umsetzung des Lösungsverfahrens für das 0/1-Rucksackproblem mit Hilfe der Methode der Dynamischen Programmierung, d.h. die Erzeugung der Funktionen f_0, ..., f_n bzw. der Mengen ihrer Sprungstellen S_0, ..., S_n und die Ermittlung einer optimalen Lösung, wird im folgenden nur mit Hilfe von Pascal-ähnlichem Pseudocode angegeben. Dabei werden folgende Begriffe verwendet:

Ein Zahlenpaar (W, P) **dominiert** ein Zahlenpaar (W', P'), wenn $W \le W'$ und $P \ge P'$ gilt.

Für zwei Mengen A und B von Zahlenpaaren sei $A \otimes B$ die Vereinigungsmenge von A und B ohne die Paare aus A, die durch ein Paar aus B dominiert werden, und ohne die Paare aus B, die durch ein Paar aus A dominiert werden.

Algorithmus zur Lösung des 0/1-Rucksackproblems:

{ Berechnung der Funktionen f_j bzw. der Mengen S_j ihrer Sprungstellen für $j = 1, ..., n$ }

```
BEGIN
```
$S_0 := \{(0,0)\}$;

```
FOR j := 1 TO n DO
   BEGIN
```
$\hat{S}_j := \{(W + w_j, P + p_j) \mid (W, P) \in S_{j-1}\}$;

$S_j := S_{j-1} \otimes \hat{S}_j$;

```
   END;

END;
```

{ Die Paare in S_j seien aufsteigend nach der ersten (und damit auch nach der zweiten) Komponente geordnet, d.h.
$S_j = \{(W_{0,j}, P_{0,j}), ..., (W_{i_j,j}, P_{i_j,j})\}$ mit $W_{k,j} \le W_{k+1,j}$ für $0 \le k < i_j$.
Für $y \ge 0$ gelte $W_{k,j} \le y < W_{k+1,j}$. Dann ist $f_j(y) = P_{k,j}$. }

{ Berechnung einer optimierende Auswahl $x_1^*, ..., x_n^*$ }

Es sei (W, P) dasjenige Paar in S_n mit $f_n(M) = P$.

```
FOR k := n DOWNTO 1 DO
   BEGIN
```
IF $(W, P) \in S_{k-1}$

```
      THEN
```
$x_k^* := 0$

```
      ELSE BEGIN
```
$x_k^* := 1$;

$W := W - w_k$;

$P := P - p_k$;

```
           END;
   END;
```

Die Folge $\left(x_1^*, ..., x_n^*\right)$ und der Wert $m\left(I, \left(x_1^*, ..., x_n^*\right)\right) = \sum_{i=1}^{n} x_i^* \cdot p_i$ der Zielfunktion werden ausgegeben.

Abbildung 7.3.3-1 zeigt ein einfaches Beispiel für das Aussehen der Funktionen $f_j(y)$. Die Eingabeinstanz lautet hier

$A = \{a_1, a_2, a_3\}, ..., a_i = (w_i, p_i)$ für $i = 1, 2, 3$:

i	Gewicht w_i	Wert p_i
1	2	1
2	3	2
3	4	5

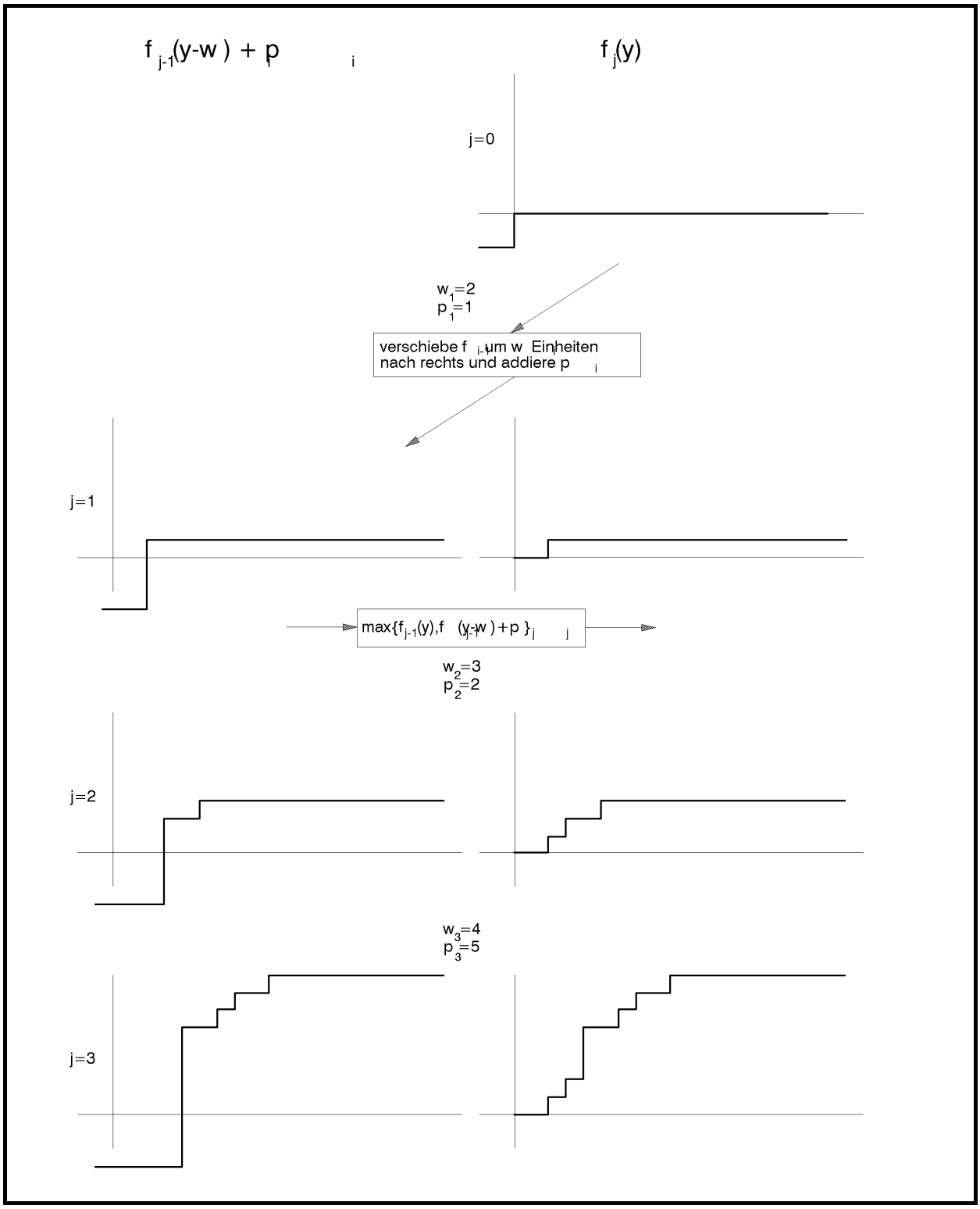


Abbildung 7.3.3-1: Beispiel für das 0/1-Rucksackproblem

Das folgende Beispiel zeigt den Ablauf des beschriebenen Algorithmus.

Eingabe $A = \{a_1, ..., a_7\}, ..., a_i = (w_i, p_i)$ für $i = 1, ..., 7$:

i	Gewicht w_i	Wert p_i
1	2	10
2	3	5
3	5	15
4	7	7
5	1	6
6	4	18
7	1	3

Rucksackkapazität $M = 15$

Berechnung der Funktionen f_j bzw. der Mengen S_j ihrer Sprungstellen für $j = 1, ..., n$:

$$S_0 = \{(0,0)\}$$

$$\hat{S}_1 = \{(2,10)\} \qquad S_1 = \{(0,0),(2,10)\}$$

$$\hat{S}_2 = \{(3,5),(5,15)\} \qquad S_2 = \{(0,0),(2,10),(5,15)\}$$

$$\hat{S}_3 = \{(5,15),(7,25),(10,30)\}$$
$$S_3 = \{(0,0),(2,10),(5,15),(7,25),(10,30)\}$$

$$\hat{S}_4 = \{(7,7),(9,17),(12,22),(14,32),(17,37)\}$$
$$S_4 = \{(0,0),(2,10),(5,15),(7,25),(10,30),(14,32),(17,37)\}$$

$$\hat{S}_5 = \{(1,6),(3,16),(6,21),(8,31),(11,36),(15,38),(18,43)\}$$
$$S_5 = \{(0,0),(1,6),(2,10),(3,16),(6,21),(7,25),(8,31),(11,36),(15,38),(18,43)\}$$

$$\hat{S}_6 = \{(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(15,54),(19,56),(22,61)\}$$

$$S_6 = \{(0,0),(1,6),(2,10),(3,16),(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(15,54),(19,56),(22,61)\}$$

$$\hat{S}_7 = \{(1,3),(2,9),(3,13),(4,19),(5,21),(6,27),(7,31),(8,37),(11,42),(12,46),(13,52),(16,57),(20,59),(23,64)\}$$
$$S_7 = \{(0,0),(1,6),(2,10),(3,16),(4,19),(5,24),(6,28),(7,34),(8,37),(10,39),$$

$$(11,43),(12,49),(13,52),(15,54),(16,57),(20,59),(22,61),(23,64)\}$$

$f_7(15) = 54.$

Berechnung einer optimierende Auswahl $x_1^*, ..., x_n^*$:

$S_0 = \{(\mathbf{0,0})\}$

$$k = 1,\ W = 2,\ P = 10, \qquad x_1^* = 1, \qquad W = 0,\ P = 0$$

$\hat{S}_1 = \{(\mathbf{2,10})\}$ $\qquad S_1 = \{(0,0),(\mathbf{2,10})\}$

$$k = 2,\ W = 10,\ P = 30, \qquad x_2^* = 1, \qquad W = 2,\ P = 10$$

$\hat{S}_2 = \{(3,5),(\mathbf{5,15})\}$ $\qquad S_2 = \{(0,0),(2,10),(\mathbf{5,15})\}$

$$k = 3,\ W = 10,\ P = 30, \qquad x_3^* = 1, \qquad W = 5,\ P = 15$$

$\hat{S}_3 = \{(5,15),(7,25),(\mathbf{10,30})\}$

$S_3 = \{(0,0),(2,10),(5,15),(7,25),(\mathbf{10,30})\}$

$$k = 4,\ W = 10,\ P = 30, \qquad x_4^* = 0$$

$\hat{S}_4 = \{(7,7),(9,17),(12,22),(14,32),(17,37)\}$

$S_4 = \{(0,0),(2,10),(5,15),(7,25),(\mathbf{10,30}),(14,32),(17,37)\}$

$$k = 5,\ W = 11,\ P = 36, \qquad x_5^* = 1, \qquad W = 10,\ P = 30$$

$\hat{S}_5 = \{(1,6),(3,16),(6,21),(8,31),(\mathbf{11,36}),(15,38),(18,43)\}$

$S_5 = \{(0,0),(1,6),(2,10),(3,16),(6,21),(7,25),(8,31),(\mathbf{11,36}),(15,38),(18,43)\}$

$$k = 6,\ W = 15,\ P = 54, \qquad x_6^* = 1, \qquad W = 11,\ P = 36$$

$\hat{S}_6 = \{(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(\mathbf{15,54}),(19,56),(22,61)\}$

$S_6 = \{(0,0),(1,6),(2,10),(3,16),(4,18),(5,24),(6,28),(7,34),(10,39),(11,43),(12,49),(\mathbf{15,54}),(19,56),(22,61)\}$

$$k = 7,\ W = 15,\ P = 54, \qquad x_7^* = 0$$

$\hat{S}_7 = \{(1,3),(2,9),(3,13),(4,19),(5,21),(6,27),(7,31),(8,37),(11,42),(12,46),(13,52),(16,57),(20,59),(23,64)\}$

$$S_7 = \{(0,0),(1,6),(2,10),(3,16),(4,19),(5,24),(6,28),(7,34),(8,37),(10,39),$$
$$(11,43),(12,49),(13,52),(15,54),(16,57),(20,59),(22,61),(23,64)\}$$

Aus der Anzahl der Mengen S_j und deren Mächtigkeiten ergibt sich folgende Komplexitätsabschätzung:

Ist $I = (A, M)$ eine Instanz des 0/1-Rucksackproblems (als Maximierungsproblem) mit n Eingabeelementen, dann liefert das beschriebene Verfahren eine optimale Lösung mit einer (worst-case-) Zeitkomplexität der Ordnung $O(n \cdot \sum_{i=1}^{n} p_i)$; hierbei ist p_i der Profit des i-ten Eingabeelements. Die Zeitkomplexität hängt also von den Werten in der Eingabeinstanz ab. Daher muss für die Berechnung der Zeitkomplexität die Anzahl elementarer Bitoperationen in Abhängigkeit von der Problemgröße $size(I)$ abgeschätzt werden. Diese ist von der Ordnung $O(n \cdot \log(p_{max}))$ mit $p_{max} = \max\{\{ p_i \mid i = 1, \ldots, n \} \cup \{ w_i \mid i = 1, \ldots, n \}\}$.

Wegen $\log\left(\sum_{i=1}^{n} p_i\right) \le n \cdot \log(p_{\max})$ ist $c_1 \cdot n \cdot \sum_{i=1}^{n} p_i \le c_1 \cdot n \cdot 2^{n \cdot \log(p_{\max})} \le c_1 \cdot n \cdot \log(p_{\max}) \cdot 2^{n \cdot \log(p_{\max})}$,

also ist die Zeitkomplexität der Ordnung $O(size(I) \cdot 2^{size(I)})$, d.h. das Verfahren hat exponentielle Laufzeit in der Größe der Eingabe. Dieselbe Größenordnung ergibt sich für den Speicherplatzbedarf.

Bis heute ist kein Verfahren bekannt, das in polynomieller Laufzeit eine optimale Lösung für das 0/1-Rucksackproblem liefert. Wahrscheinlich wird es auch kein Verfahren geben; denn das zugehörige Entscheidungsproblem ist **NP**-vollständig. In einigen Spezialfällen lassen sich jedoch schnelle Lösungsverfahren angeben.

Das folgende Beispiel (aus der Kryptologie) beschränkt die Eingabeinstanzen auf „schnell wachsende“ (super increasing) Folgen und einfache Eingabewerte:

Das 0/1-Rucksackproblem mit schnell wachsenden ganzzahligen Eingabewerten (als Entscheidungsproblem)

Eingabe-
instanz: $I = (A, M)$

$A = \{a_1, \ldots, a_n\}$ ist eine Menge von n natürlichen Zahlen mit der Eigenschaft $a_j > \sum_{i=1}^{j-1} a_i$ (super increasing) und $M \in \mathbf{N}$.

Gesuchte Lösung: Entscheidung „ja“, falls es eine Folge $x_1, ..., x_n$ von Zahlen mit $x_i = 0$ oder $x_i = 1$ für $i = 1, ..., n$ gibt, so dass $\sum_{i=1}^{n} x_i \cdot a_i = M$ gilt,

Entscheidung „nein“ sonst.

Die folgenden (Pseudocode-) Anweisungen nach der Greedy-Methode ermitteln eine Folge $x_1, ..., x_n$ mit der gewünschten Eigenschaft, falls es sie gibt:

```
TYPE Entscheidungstyp = (ja, nein);

VAR  Entscheidung : Entscheidungstyp;

summe := 0;
FOR i := n DOWNTO 1 DO
  IF a_i + summe ≤ M
  THEN BEGIN
         summe := summe + a_i;
         x_i := 1
       END
  ELSE x_i := 0;
IF summe = M THEN Entscheidung = ja
             ELSE Entscheidung = nein;
```

Ist $I = (A, M)$ eine Eingabeinstanz des 0/1-Rucksackproblems mit schnell wachsenden Eingabewerten als Entscheidungsproblem mit Problemgröße $size(I) \in O(n \cdot \log(a_n))$, dann liefert das beschriebene Verfahren eine Entscheidung mit einer (worst-case-) Zeitkomplexität der Ordnung $O(size(I))$.

Beispiel: $I = (\{a_1, ..., a_7\}, 234)$ mit $a_1 = 1, a_2 = 2, a_3 = 5, a_4 = 11, a_5 = 32, a_6 = 87, a_7 = 141$.

a_i + summe $\leq M$	x_i
$a_7 + 0 = 141 < 234$	$x_7 = 1$
summe = 141	
$a_6 + 141 = 87 + 141 = 228 < 234$	$x_6 = 1$
summe = 228	
$a_5 + 228 = 32 + 228 = 260 > 234$	$x_5 = 0$
summe = 228	
$a_4 + 228 = 11 + 228 = 239 > 234$	$x_4 = 0$
summe = 228	
$a_3 + 228 = 5 + 228 = 233 < 234$	$x_3 = 1$
summe = 233	
$a_2 + 233 = 2 + 233 = 235 > 234$	$x_2 = 0$
summe = 233	
$a_1 + 233 = 1 + 233 = 234$	$x_1 = 1$
summe = 234	
Entscheidung „ja“	

7.4 Branch and Bound

Die Branch-and-bound-Methode ist ein Spezialfall einer Backtrack-Methode:

Aufzählungsmethoden: Backtrack-Methode:

Die Backtrack-Methode ist auf Probleme anwendbar, für deren Lösung kein besseres Verfahren bekannt ist, als alle möglichen Lösungskandidaten zu inspizieren und daraufhin zu untersuchen, ob sie als Lösung in Frage kommen. Die Backtrack-Methode organisiert diese erschöpfende Suche in einem im allgemeinen sehr großen Problemraum. Dabei wird ausgenutzt, dass sich oft nur partiell erzeugt Lösungskandidaten schon als inkonsistent ausschließen lassen.

Die effiziente Einsatz der Backtrack-Methode setzt voraus, dass das zu lösende Problem folgende Struktur aufweist:

1. Die Lösung ist als Vektor ($a[1]$, $a[2]$, ...) unbestimmter, aber endlicher Länge darstellbar.

2. Jedes Element $a[i]$ ist eine Möglichkeit aus einer endlichen Menge $A[i]$.

3. Es gibt einen effizienten Test zur Erkennung von inkonsistenten Teillösungen, d.h. Kandidaten ($a[1]$, $a[2]$, ..., $a[i]$), die sich zu keiner Lösung ($a[1]$, $a[2]$, ..., $a[i]$, $a[i+1]$, ...) erweitern lassen.

Das Verfahren kann allgemein so formuliert werden:

Schritt 1: Man wählt als erste Teillösung $a[1]$ ein mögliches Element aus $A[1]$.

Schritt n: Ist eine Teillösung ($a[1]$, $a[2]$, ..., $a[i]$) noch keine Gesamtlösung, dann erweitert man sie mit dem nächsten nicht inkonsistenten Element $a[i+1]$ aus $A[i+1]$ zur neuen Teillösung ($a[1]$, $a[2]$, ..., $a[i]$, $a[i+1]$). Falls alle nicht inkonsistenten Elemente aus $A[i+1]$ bereits abgearbeitet sind, ohne dass man eine so erweiterte Teillösung gefunden wurde, geht man zurück und wählt $a[i]$ aus $A[i]$ neu (bzw. $a[i-1]$ aus $A[i-1]$ usw., wenn auch alle nicht inkonsistenten Kandidaten für $a[i]$ bereits abgearbeitet sind).

Die Branch-and-Bound-Methode ordnet sich in dieses Prinzip ein:

Aufzählungsmethoden: Branch-and-Bound-Methode (Optimierungsaufgaben):

Potentielle, aber nicht unbedingt optimale Lösungen werden systematisch erzeugt, z.B. mit der Backtrack-Methode. Diese werden in Teilmengen aufgeteilt, die sich auf Knoten eines Entscheidungsbaums abbilden lassen. Es wird eine Abschätzung für das Optimum mitgeführt und während des Verfahrensverlaufs laufend aktualisiert. Potentielle Lösungen, deren Zielfunktion einen „weit von der Abschätzung entfernten Wert" aufweisen, werden nicht weiter betrachtet, d.h. der Teilbaum, der bei einer derartigen Lösung beginnt, muss nicht weiter durchlaufen werden.

Die Branch and Bound-Methode liefert in vielen praktischen Fällen eine sehr effiziente Lösungsmethode für Probleme, deren Lösungsraum mindestens exponentiell großen Umfang aufweist. Sie wird wieder am Problem des Handlungsreisenden auf Graphen (minimum traveling salesperson problem), siehe Kapitel 7.3.1, erläutert.

7.4.1 Problem des Handlungsreisenden 2. Lösungsansatz

Eine Eingabeinstanz $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht; als Problemgröße kann hier die Anzahl n der Konten des Graphen genommen werden. Gesucht wird eine Tour T^*, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G, T^*)$.

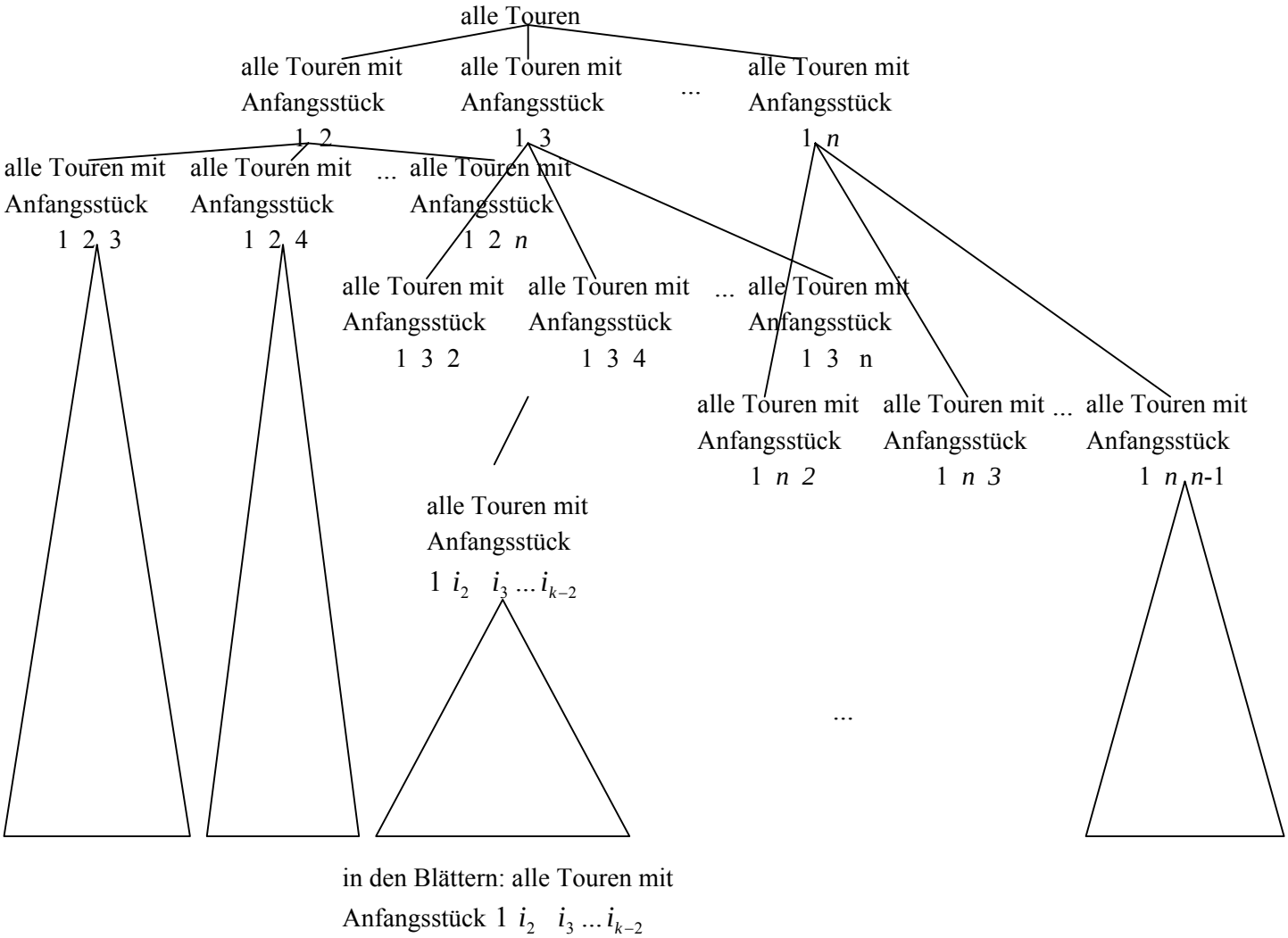
Im folgenden wird wieder angenommen, dass eine Tour bei v_1 beginnt und endet. Außerdem wird angenommen, dass der Graph **vollständig** ist, d.h. dass $w((v_i, v_j))$ für jedes $v_i \in V$ und $v_j \in V$ definiert ist (eventuell ist $w((v_i, v_j)) = \infty$).

Im folgenden wird (zur Vereinfachung der Darstellung) die Knotenmenge $V = \{v_1, ..., v_n\}$ mit der Menge der Zahlen $\{1, ..., n\}$ gleichgesetzt (dem Knoten v_i entspricht die Zahl i). Eine Tour durch G lässt sich dann als Zahlenfolge

$$\langle 1 \; i_1 \; i_2 \ldots i_{n-1} \; 1 \rangle$$

darstellen, wobei alle i_j paarweise verschieden (und verschieden von 1) sind. Alle Touren erhält man, wenn man für i_j in $\langle 1 \; i_1 \; i_2 \ldots i_{n-1} \; 1 \rangle$ alle $(n-1)!$ Permutationen der Zahlen 2, ..., n einsetzt. Manche dieser Touren haben eventuell das Gewicht ∞.

Alle Touren (Zahlenfolgen der beschriebenen Art) lassen sich als Blätter eines Baums darstellen :



Alle Touren (Zahlenfolgen der beschriebenen Art) werden systematisch erzeugt (siehe unten). Dabei wird eine obere Abschätzung *bound* für das Gewicht einer optimalen Tour (Tour mit minimalem Gewicht) mitgeführt (Anfangswert *bound* = ∞). Wird ein Anfangsstück einer Tour erzeugt, deren Gewicht größer als der Wert *bound* ist, dann müssen alle Touren, die mit diesem Anfangsstück beginnen, nicht weiter betrachtet werden. Es wird also ein ganzer Teilbaum aus dem Baum aller Touren abgeschnitten. Ist schließlich eine Tour gefunden, deren Gewicht kleiner oder gleich dem Wert dem Wert *bound* ist, so wird *bound* durch diesen neuen Wert ersetzt, und die Tour wird als temporär optimale Tour vermerkt. Die Knotennummern einer temporär optimalen Tour werden ebenfalls mitgeführt.

Ist bereits ein Anfangsstück $\langle 1\ i_1\ i_2 \ldots i_{k-1} \rangle$ einer Tour erzeugt, so sind die Zahlen $1,\ i_1,\ i_2, \ldots,\ i_{k-1}$ alle paarweise verschieden. Eine weitere Zahl i kann in die Folge aufgenommen werden, wenn

(1) i unter den Zahlen $1,\ i_1,\ i_2, \ldots,\ i_{k-1}$ nicht vorkommt und
(2) $w((v_{i_{k-1}}, v_i)) < \infty$ ist und
(3) das Gewicht der neu entstehenden Teiltour $\langle 1,\ i_1,\ i_2, \ldots,\ i_{k-1}\ i \rangle$ kleiner oder gleich dem Wert *bound* ist.

Treffen (1) oder (2) nicht zu, kann i nicht als Fortsetzung der Teiltour $1,\ i_1,\ i_2, \ldots,\ i_{k-1}$ genommen werden, denn es entsteht keine Tour. Trifft (3) nicht zu (aber (1) und (2)), dann brauchen alle Touren, die mit dem Anfangsstück $\langle 1,\ i_1,\ i_2, \ldots,\ i_{k-1}\ i \rangle$ beginnen, nicht weiter berücksichtigt zu werden.

Algorithmus zur Lösung des Problems des Handlungsreisenden nach der Branch-and-bound-Methode:

Der Objekttyp `TGraph` in der `UNIT Graph` wird wieder um entsprechende Methoden ergänzt:

Methode	Bedeutung
`FUNCTION TGraph.salesman_bab : INTEGER;`	Die Funktion ermittelt das minimale Gewicht einer Tour durch den Graphen nach der Branch-and-Bound-Methode und zeigt diese an. Gibt es keine Tour, so wird der Wert ∞ zurückgegeben.

```
FUNCTION TGraph.salesman_bab : INTEGER;

  VAR idx      : INTEGER;
      OK       : BOOLEAN;
      opttour  : PVektor;
                 { temporäre optimale Tour                    }
      teiltour : PVektor;
                 { zu ergänzendes Anfangsstück einer Tour     }
      bound    : INTEGER;
                 { Länge der temporären Tour                  }

  PROCEDURE  bab_g (gewicht  : INTEGER;
                    { Gewicht des Anfangsstücks        }
                    position : INTEGER
                    { Position, an der zu ergänzen ist });

     VAR i  : INTEGER;
```

```
        j  : INTEGER;
        OK : BOOLEAN;
        w  : INTEGER;

    BEGIN {bab_g }
      IF position = n + 1
      THEN BEGIN
             w := Adjazenzmatrix^.m(teiltour^.v(n), 1);
             IF (w < unendlich)
                AND
                (gewicht + w < bound)
             THEN BEGIN
                    teiltour^.setze (position, 1);
                    bound   := gewicht + w;
                    FOR i := 1 TO n + 1 DO
                     opttour^.setze (i, teiltour^.v(i));
                  END;
           END
      ELSE BEGIN
             FOR i := 2 TO n DO
               BEGIN
                 w  := Adjazenzmatrix^.m
                                 (teiltour^.v(position - 1), i);
                 OK := TRUE;
                 { Bedingung (2):}
                 FOR j := 2 TO position - 1 DO
                   IF teiltour^.v(j) = i
                   THEN BEGIN
                          OK := FALSE;
                          Break;
                        END;
                 IF OK
                    AND
                    (w < unendlich)
                    AND
                    (gewicht + w < bound)
                 THEN BEGIN
                        teiltour^.setze (position, i);
                        bab_g (gewicht + w, position + 1);
                      END;
               END;
           END;
    END   {bab_g };

BEGIN { TGraph.salesman_bab }
  New (opttour, init (n + 1, OK));
  IF OK THEN New (teiltour, init (n + 1, OK));
```

```
  IF OK
  THEN BEGIN
         FOR idx := 2 TO n + 1 DO
           opttour^.setze (idx, 0);
         teiltour^.setze (1, 1);
         bound := unendlich;

         bab_g(0, 2);

         { optimale Tour anzeigen: }
         FOR idx := 1 TO n + 1 DO
           -- opttour^.v(idx) anzeigen ;

         Dispose (opttour, done);
         Dispose (teiltour, done);

         salesman_bab := bound;
       END;
END   { TGraph.salesman_bab };
```

Das Verfahren erzeugt u.U. alle $(n-1)!$ Folgen $\langle 1\ i_1\ i_2 \ldots i_{n-1}\ 1 \rangle$, bis es eine optimale Tour gefunden hat. In der Praxis werden jedoch schnell Folgen mit Anfangsstücken, die ein zu großes Gewicht aufweisen, ausgeschlossen. Die (worst-case-) Zeitkomplexität des Verfahrens bleibt jedoch mindestens exponentiell in der Anzahl der Knoten des Graphen in der Eingabeinstanz.

8 Spezielle Lösungsansätze

In den folgenden Unterkapiteln werden zwei Ansätze im Überblick und an wenigen Beispielen vorgestellt, die von den klassischen Programmieransätzen der vorherigen Kapitel abweichen.

8.1 Parallele Algorithmen

In den bisherigen Kapiteln wurden Datenstrukturen und Algorithmen behandelt, die als Rechnergrundlage die klassische von-Neumann-Architektur aufweisen. Beim Einsatz massiv paralleler Rechner (symmetrische Mehrprozessorsysteme oder auch lose gekoppelte Mehrprozessorsysteme) wird versucht, Teile eines Algorithmus auf mehrere physikalisch parallel arbeitende Prozessoren zu verteilen. Dabei kann versucht werden, einen klassischen Algorithmus in möglichst unabhängige Teile zu zerlegen, so dass diese mit einem hohen Maß an Parallelität auf die Prozessoren verteilt werden können. Durch den Entwurf spezieller Verfahren können dann die Spezifika der zugrundeliegenden Hardwarestruktur genutzt werden.

Im folgenden wird eine **massiv parallele Rechnerstruktur** angenommen. Hierbei sind mehrere (häufig mehrere hundert) Rechner miteinander fest gekoppelt, oder ihre Verbindung ist konfigurierbar, so dass ein festes Netz aus einzelnen Rechnerknoten entsteht (Abbildung 8.1-1). Häufig wird dabei eine **reguläre Struktur**, z.B. hierarchische Baumstruktur, würfelförmig vernetzte Struktur, rückgekoppelte Schichten usw., verwendet. Im allgemeinen ist die Verbindungsstruktur algorithmusabhängig. Jeder dieser Rechnerknoten verfügt eventuell über einen eigenen lokalen Arbeitsspeicher. Meist besteht eine Zugriffsmöglichkeit auf einen gemeinsamen globalen Speicher. Der Befehlssatz eines Rechnerknotens umfasst einfache und damit sehr schnell ausführbare arithmetische und logische Operationen und E/A-Operationen zu verbundenen Rechnerknoten. Jede CPU wird mit *demselben* Programm geladen oder das gesamte Rechnernetz wird durch ein globales Kontrollprogramm gesteuert, so dass in jedem Rechnerknoten zu jedem Zeitpunkt Teile eines gemeinsamen globalen Programms läuft. Gegenseitige Synchronisation vermittelt den Eindruck, dass alle Rechner global getaktet an einen gemeinsamen „parallelen“ Algorithmus arbeiten; die Parallelität bezieht sich dabei auf unterschiedliche Datenströme innerhalb des Gesamtalgorithmus.

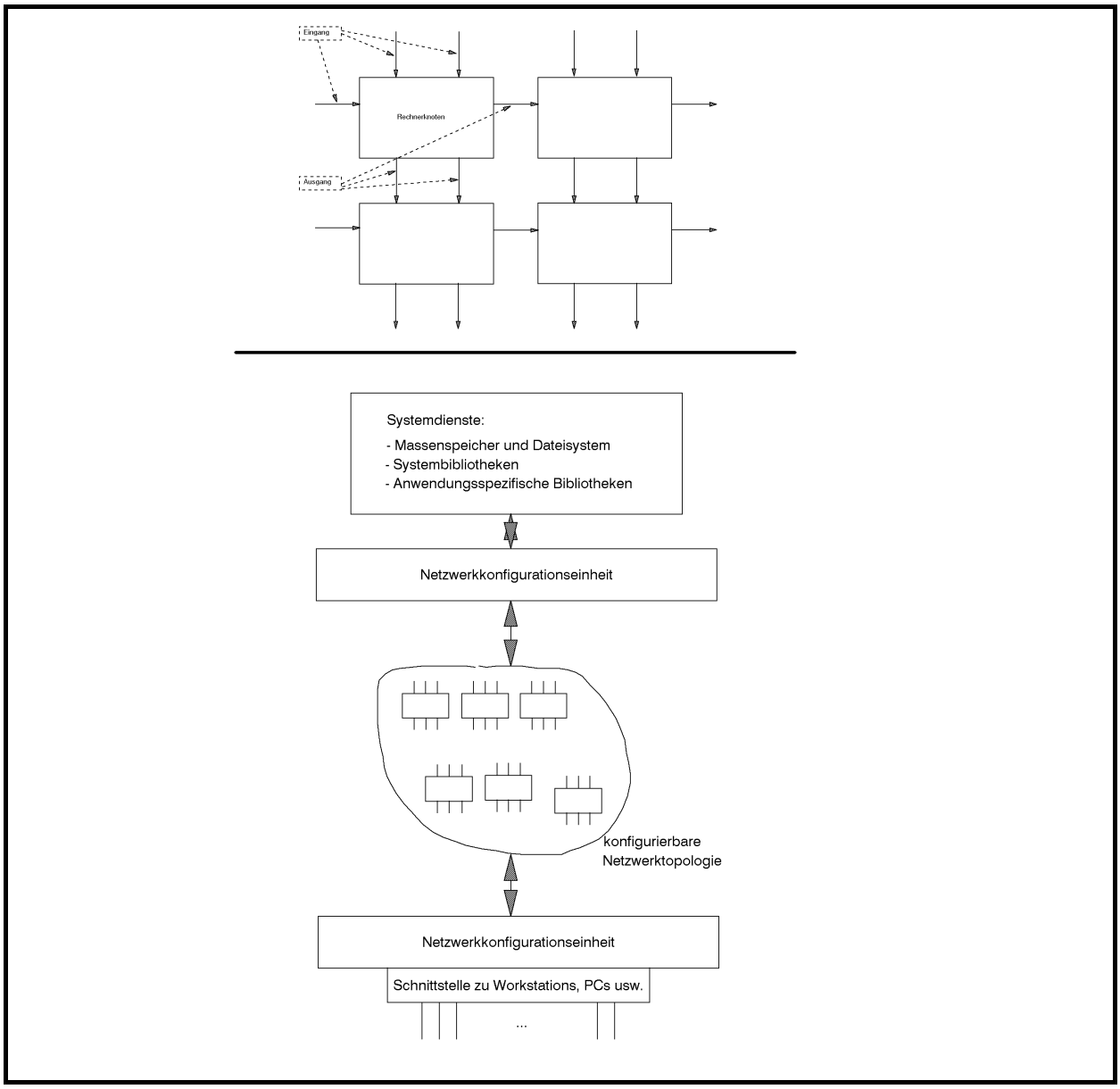


Abbildung 8.1-1: Parallele Hardwarestruktur

Die theoretische Grundlage eines parallelen Algorithmus bietet in Erweiterung der Random Access Maschine (RAM) das Modell der **Parallelen Random Access Maschine (PRAM)**: Parallel arbeitende RAM-Module greifen auf einen gemeinsamen Speicher zu. Dabei ist die Definition unterschiedlicher Type von **Restriktionen bezüglich des gleichzeitigen Zugriffs auf dieselben Speicherzellen** durch unterschiedliche RAM-Module erforderlich ([CHA]), die beim Entwurf paralleler Algorithmen zu beachten sind. Eine Auswahl dieser Restriktionen ist:

- zwei unterschiedliche RAM-Module dürfen nicht gleichzeitig auf dieselbe Speicherzelle zugreifen (**exclusive read exclusive write PRAM, EREW PRAM**)
- zwei unterschiedliche RAM-Module dürfen gleichzeitig lesend, aber nicht schreibend auf dieselbe Speicherzelle zugreifen (**concurrent read exclusive write PRAM, CREW PRAM**)

- zwei unterschiedliche RAM-Module dürfen gleichzeitig lesend und schreibend auf dieselbe Speicherzelle zugreifen (**concurrent read concurrent write PRAM, CRCW PRAM**); Konflikte beim simultanen Schreiben werden nach definierten Regeln gelöst, z.B. werden RAM-Modulen Prioritäten zugeordnet, und das RAM-Modul mit der höchsten Priorität schreibt seinen Wert in die Speicherzelle, oder simultanes Schreiben ist nur erlaubt, wenn alle RAM-Module denselben Wert in die Speicherzelle schreiben wollen.

In jedem Fall hat der Typ der Restriktionen Einfluss auf die zeitliche Komplexität des parallelen Algorithmus.

Weitere mögliche Probleme und Konflikte entstehen bei der Aufteilung des Gesamtalgorithmus in Teilaufgaben bzw. der Entwicklung neuer speziell angepasster paralleler Algorithmen. Problematisch sind weiterhin die Definition einer für alle oder zumindest für sehr viele Problemklassen gültigen **Kommunikationsstruktur** (Lösungsansätze sind Shuffle-Exchange-Netze, mehrdimensionale Gitter, Hypercube-Struktur usw.), die Synchronisation der Teilabläufe und des Gesamtalgorithmus, die Entwicklung geeigneter Betriebssysteme, die eine Algorithmusaufteilung auf die einzelnen Rechnerknoten eventuell selbständig vornehmen, die Skalierbarkeit (z.B. die Frage, ob eine Performanceverbesserung durch Vergrößerung der Parallelität erzielt wird) usw. Ein möglicher Realisierungsansatz besteht darin, dem Rechnernetz eine Netzwerkkonfigurationseinheit vorzuschalten, die in einer Initialisierungsphase das Rechnernetz zu einer den jeweiligen Algorithmen angemessenen Topologie zusammenschalten kann. Die Netzwerkkonfigurationseinheit ermöglicht auch den Zugriff für alle Rechnerknoten auf allgemeine Systemdienste und –prozeduren und auf umfangreiche Massenspeicher usw.

Es sei $T_{\mathbf{A}}(n,p)$ der Zeitaufwand, gemessen in parallelen Schritten (Takten), die ein paralleler Algorithmus **A** zur Lösung eines Problems mit Problemgröße n benötigt, wenn p parallel arbeitende Prozessoren eingesetzt werden. Die **Kosten des Algorithmus** sind in diesem Fall definiert durch

$$C_{\mathbf{A}}(n,p) = p \cdot T_{\mathbf{A}}(n,p) .$$

Es ist leicht einzusehen, dass ein paralleler Algorithmus, der p Prozessoren einsetzt und einen (parallelen) Zeitaufwand der Ordnung $O\big(T_{\mathbf{A}}(n,p)\big)$ hat, durch einen seriellen Algorithmus mit Zeitkomplexität der Ordnung $O\big(p \cdot T_{\mathbf{A}}(n,p)\big)$ simuliert werden kann. Daher sind die oben definierten Kosten eines parallelen Algorithmus ein guter Maßstab für den Vergleich des Verfahrens mit einem seriell ablaufenden Algorithmus.

Ein weiterer Maßstab zur Beurteilung eines parallelen Algorithmus **A** ist sein **Speedup**, der durch

$$S_{\mathbf{A}}(n,p) = \frac{\text{Zeitaufwand des schnellsten seriellen Algorithmus zur Lösung des Problems}}{T_{\mathbf{A}}(n,p)}$$

definiert wird.

Zur Beurteilung der Auslastung der Prozessoren in einem parallelen Algorithmus **A** wird die **Effizienz** herangezogen, die durch

$$E_{\mathbf{A}}(n,p) = \frac{S_{\mathbf{A}}(n,p)}{p} = \frac{\text{Zeitaufwand des schnellsten seriellen Algorithmus zur Lösung des Problems}}{C_{\mathbf{A}}(n,p)}$$

definiert ist. Es gilt $0 \le E_{\mathbf{A}}(n,p) \le 1$, und ein hoher Wert bedeutet, dass die Kosten des parallelen Algorithmus dicht beim optimalen Aufwand eines seriellen Algorithmus zur Lösung des Problems liegen.

In den folgenden Unterkapiteln werden Beispiele für Verfahren beschrieben, die auf spezielle massiv parallele Rechnerarchitekturen zugeschnitten sind. Die Beispiele behandeln klassische Basisprobleme, nämlich Sortier- und Mischverfahren und arithmetische Basisalgorithmen zu Matrixoperationen.

8.1.1 Paralleles Sortieren

Das erste Beispiel beschreibt ein parallel ablaufendes Verfahren für das **Sortierproblem**.

Abbildung 8.1.1-1 zeigt das **Prinzip eines Sortiernetzes zur parallelen Sortierung** von $n = 2m$ unsortierten Elemente $a_1, ..., a_n$. Die durch einen Primärschlüsselwert in jedem Element definierte Ordnungsrelation ist auf die Elemente selbst übertragbar, so dass jeweils zwei Elemente vergleichbar sind, geschrieben $a_i \le a_j$ bzw. $a_j \le a_i$ (vgl. Kapitel 6.1). Das Prinzip ist dem Divide-and-Conquer-Ansatz entlehnt. Zur Vereinfachung der Darstellung wird $n = 2^k$ angenommen.

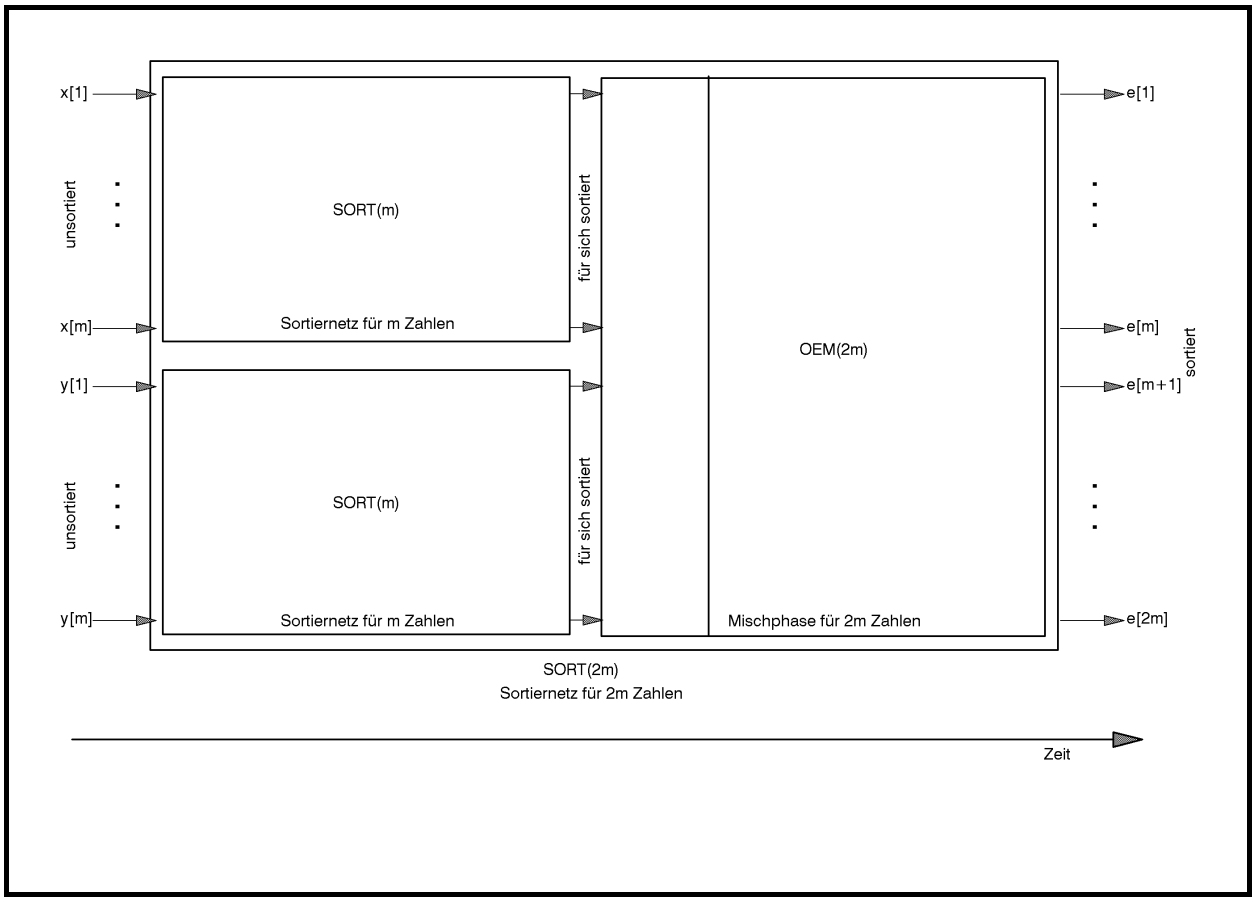


Abbildung 8.1.1-1: Sortiernetz

Die erste Hälfte der Eingabe $a_1, ..., a_{n/2}$ wird in Abbildung 8.1.1-1 mit $x[1], ..., x[m]$ bezeichnet, die zweite Hälfte $a_{n/2+1}, ..., a_n$ mit $y[1], ..., y[m]$. Der zeitliche Ablauf ist von links nach rechts gedacht. Alle Eingabewerte $x[1], ..., x[m]$ und $y[1], ..., y[m]$ werden in einem einzigen „parallelen" Schritt in das Sortiernetz eingegeben, wobei $x[1], ..., x[m]$ in das obere Teilnetz SORT(m) einfließen, $y[1], ..., y[m]$ in das untere Teilnetz SORT(m). Beide mit SORT(m) bezeichneten Teilnetze arbeiten jetzt zeitlich parallel zueinander und erzeugen nach einer gewissen Zeit gleichzeitig zwei sortierte Teilfolgen jeweils der Länge $m = n/2$. Die erste dieser für sich sortierten Teilfolgen wird mit $u[1], ..., u[m]$ bezeichnet, die zweite Teilfolge mit $v[1], ..., v[m]$. Beide Teilfolgen werden nun einer mit OEM($2m$) bezeichneten Mischphase (OEM steht für Odd-Even-Merge) unterzogen, die aus den beiden jeweils für sich sortierten Folgen der Längen m eine insgesamt sortierte Folge der Länge $2m = n$ produziert. Diese besteht aus der „zusammengemischten" Folge beider Eingabefolgen. Sie wird jetzt mit $e[1], ..., e[n]$ bezeichnet und ist die sortierte Umordnung von $a_1, ..., a_n$.

Der Ablauf der Mischphase OEM($2m$) ist in Abbildung 8.1.1-2 dargestellt. Die Teilfolgen der Elemente mit ungeradem Index aus $u[1], ..., u[m]$ und $v[1], ..., v[m]$ (das sind die Elemente $u[1], u[3], ..., u[m-1]$ und $v[1], v[3], ..., v[m-1]$) werden einer Mischphase OEM(m) unterzogen und parallel dazu die Teilfolgen der Elemente mit geradem Index aus $u[1], ..., u[m]$ und $v[1], ..., v[m]$ (das sind die Elemente $u[2], u[4], ..., u[m]$ und $v[2], v[4], ..., v[m]$). Die Ergebnisse sind zwei Folgen $c[1], ..., c[m]$ bzw. $d[1], ..., d[m]$ jeweils mit Länge m, die aus den nun

in der korrekten numerischen Reihenfolge zusammengemischten Elementen aus $u[1], ..., u[m]$ und $v[1], ..., v[m]$ mit ungeradem bzw. geradem Index bestehen. Jeweils zwei dieser Ausgabewerte aus beiden Mischphasen OEM(m) werden nun noch einmal in einem parallelen Schritt größenmäßig verglichen, so dass OEM($2m$)

$$
\begin{aligned}
e[1] &= c[1], \\
e[2i] &= \min \{ c[i+1], d[i] \}, \\
e[2i+1] &= \max \{ c[i+1], d[i] \} \text{ für } i = 1, ..., m-1, \\
e[2m] &= d[m]
\end{aligned}
$$

ausgibt. In Anschluss an Abbildung 8.1.1-3 wird gezeigt, dass in der Tat die Folge $e[1], ..., e[2m]$ aufsteigend sortiert ist.

Offensichtlich lässt sich die Mischphase OEM($2m$) durch zwei zeitlich parallel arbeitende Mischphasen OEM(m) mit derselben Struktur und zusätzlichen Parallelschaltungen eines einfachen Grundbausteins aufbauen (Abbildung 8.1.1-2). Dieser Grundbaustein wird als L/H-Element (lower/higher) bezeichnet und kann als ein eigenständiger speicherloser Prozessor angesehen werden, der zeitlich getaktet arbeitet und zwei Inputleitungen I1 und I2 besitzt. In einem Zeittakt (Schritt) werden auf die mit L bzw. H bezeichneten Leitungen gleichzeitig die Werte L = min { I1, I2 } bzw. H = max { I1, I2 } ausgegeben, sobald beide Inputs anliegen. Nach diesem Zeittakt sind die Inputs nicht mehr verfügbar (sie werden „konsumiert“). Falls noch nicht beide Inputs verfügbar sind, wartet der Grundbaustein für einen Zeittakt.

Insgesamt besteht ein OEM($2m$) aus L/H-Elementen und OEM(m)'s, die wiederum aus kleineren OEM's und L/H-Elementen aufgebaut sind. Da ein OEM(2) selbst ein einzelnes L/H-Element ist, setzt sich der gesamte Algorithmus OEM($2m$) aus L/H-Elementen zusammen. Insgesamt kann das Sortiernetz SORT($2m$) aus L/H-Elementen (für OEM($2m$)) und kleineren Sortiernetzen zusammengesetzt werden, also insgesamt aus L/H-Elementen (SORT(2) ist ein einziges L/H-Element).

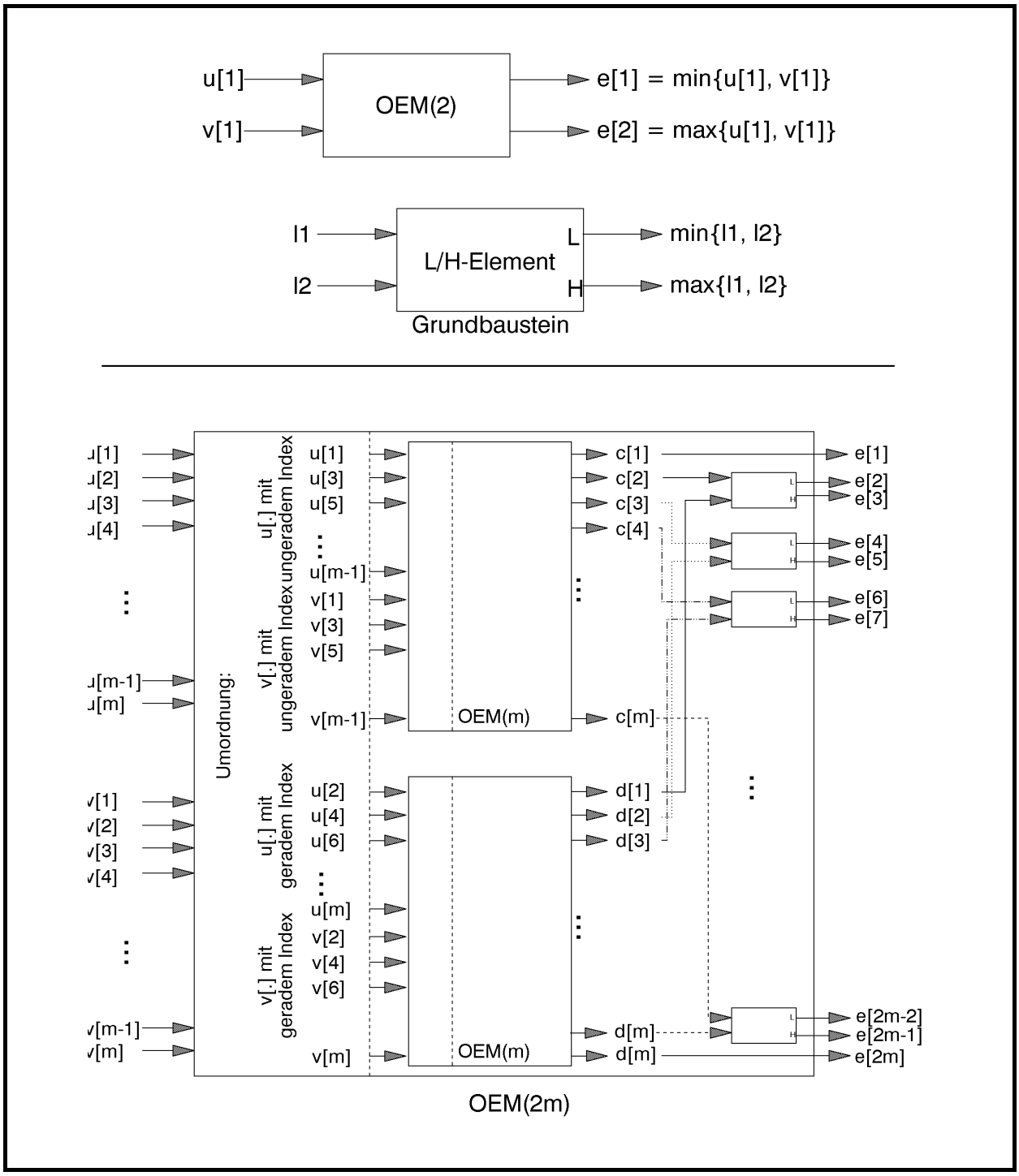


Abbildung 8.1.1-2: Odd-Even-Merge (OEM)

Zur Illustration zeigt Abbildung 8.1.1-3 ein Sortiernetz für $n = 8$ Zahlen. Jedes L/H-Element wird nur aktiv, wenn beide Eingänge mit Werten belegt sind. Man sieht, dass in einem Zeittakt (in einem parallelen Schritt) immer die übereinanderstehenden L/H-Elemente arbeiten. Damit alle 8 Zahlen schließlich sortiert auf der rechten Seite zur Verfügung stehen, sind 6 parallele Schritte erforderlich, nämlich 3 Zeittakte der beiden parallel zueinander arbeitenden SORT(4) und anschließend 3 parallele Schritte des Teilalgorithmus OEM(8).

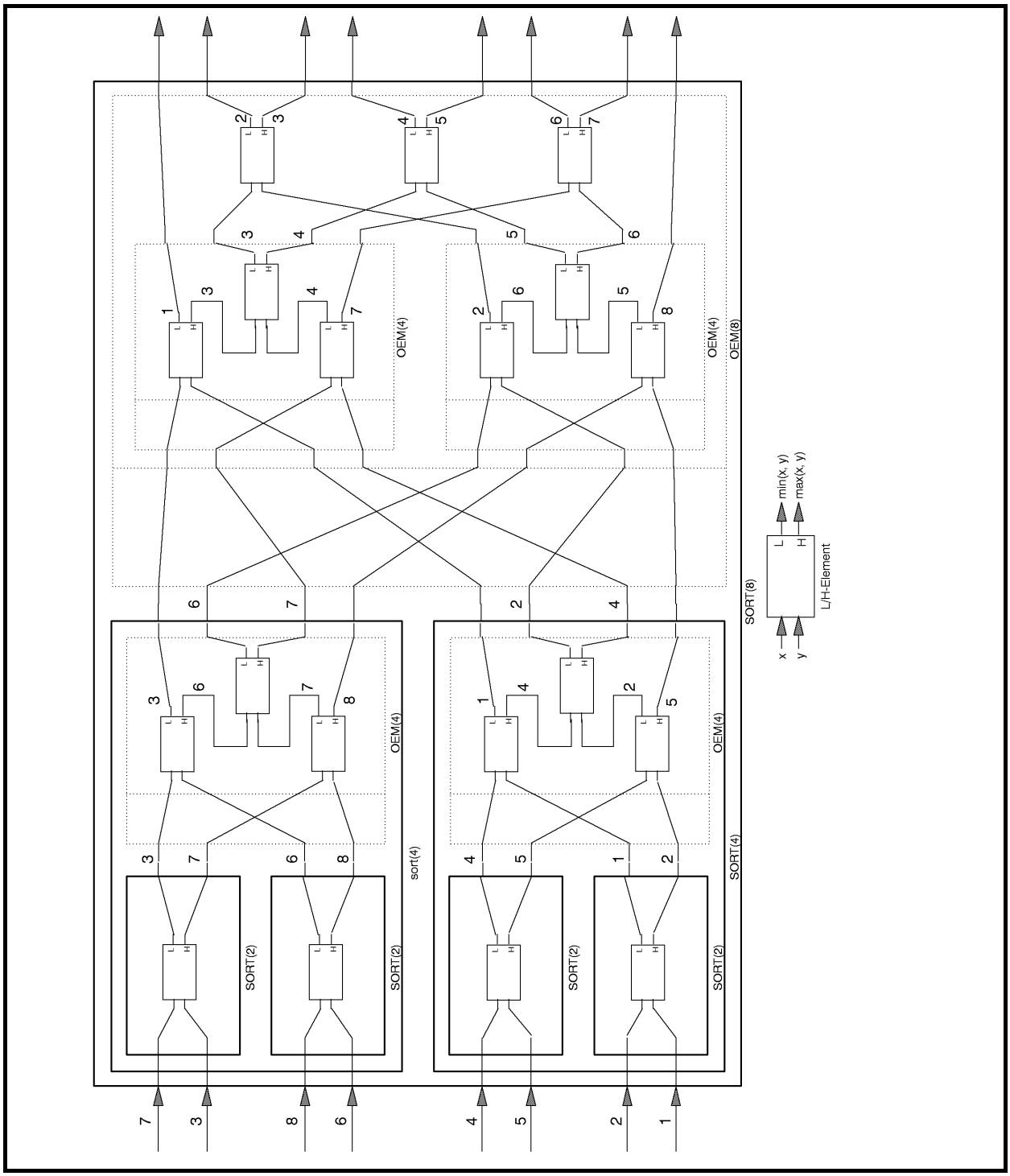


Abbildung 8.1.1-3: Sortiernetz (Beispiel)

Die Korrektheit des gesamten Verfahrens lässt sich durch den Nachweis der Korrektheit der Mischphase OEM(2*m*) zeigen. Es kann dabei angenommen werden, dass $n = 2m$ eine Zweierpotenz ist; insbesondere ist dann die Zahl *m* gerade.

Für kleine Werte von 2*m* gelingt dieser Nachweis direkt:

Für 2*m* = 2 besteht das Sortiernetz aus einem einzigen L/H-Element (siehe Abbildung 8.1.1-3 im inneren linken Teil) ohne OEM-Phase und ist offensichtlich korrekt.

Die Mischphase für $2m = 4$ zeigt Abbildung 8.1.1-4.

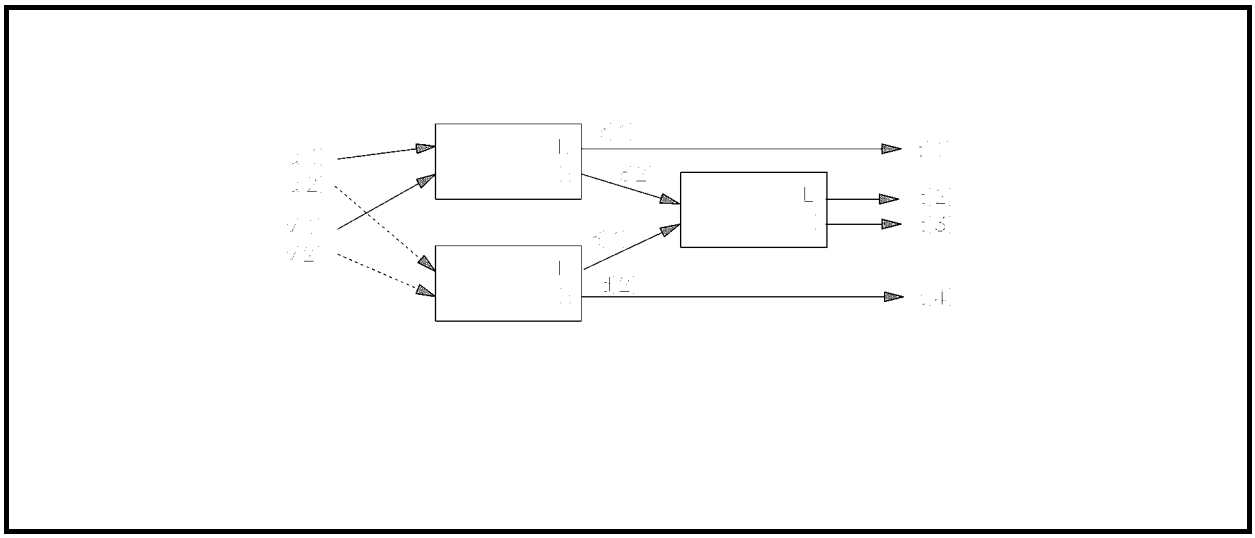


Abbildung 8.1.1-4: OEM(4)

Hier lässt sich durch Überprüfung aller möglichen Anordnungen der Werte $u[1]$, $u[2]$, $v[1]$ und $v[2]$, wobei $u[1] \le u[2]$ und $v[1] \le v[2]$ gilt, die Korrektheit nachweisen:

Anordnung	$c[1]$	$c[2]$	$d[1]$	$d[2]$	$e[1]$	$e[2]$	$e[3]$	$e[4]$
$u[1] \le u[2] \le v[1] \le v[2]$	$u[1]$	$v[1]$	$u[2]$	$v[2]$	$u[1]$	$u[2]$	$v[1]$	$v[2]$
$u[1] \le v[1] \le u[2] \le v[2]$	$u[1]$	$v[1]$	$u[2]$	$v[2]$	$u[1]$	$v[1]$	$u[2]$	$v[2]$
$v[1] \le u[1] \le u[2] \le v[2]$	$v[1]$	$u[1]$	$u[2]$	$v[2]$	$v[1]$	$u[1]$	$u[2]$	$v[2]$
$u[1] \le v[1] \le v[2] \le u[2]$	$u[1]$	$v[1]$	$v[2]$	$u[2]$	$u[1]$	$v[1]$	$v[2]$	$u[2]$
$v[1] \le u[1] \le v[2] \le u[2]$	$v[1]$	$u[1]$	$v[2]$	$u[2]$	$v[1]$	$u[1]$	$v[2]$	$u[2]$
$v[1] \le v[2] \le u[1] \le u[2]$	$v[1]$	$u[1]$	$v[2]$	$u[2]$	$v[1]$	$v[2]$	$u[1]$	$u[2]$

Für größere Werte von $2m$ argumentiert man wie folgt.

Die jeweils aufsteigend sortierten Inputs $[u[1], ..., u[m]]$ und $[v[1], ..., v[m]]$ werden in Teillisten gemäß des Indexes der Elemente zerlegt:

$$u_{ungerade} = [u[1], u[3], ..., u[m-1]],\ u_{gerade} = [u[2], u[4], ..., u[m]],$$
$$v_{ungerade} = [v[1], v[3], ..., v[m-1]],\ v_{gerade} = [v[2], v[4], ..., v[m]].$$

In Abbildung 8.1.1-2 kann man annehmen, dass die beiden kleineren Mischphasen OEM(m) jeweils korrekt arbeiten, d.h. dass die Folgen $[c[1], ..., c[m]]$ und $[d[1], ..., d[m]]$ jeweils aufsteigend sortiert sind. Die Folge $[c[1], ..., c[m]]$ besteht aus den Werten der Teillisten $u_{ungerade}$ und $v_{ungerade}$, die Folge $[d[1], ..., d[m]]$ besteht aus den Werten der Teillisten u_{gerade} und v_{gerade}.

Zu zeigen bleibt, dass der letzte Mischtakt zum korrekten Ergebnis der aufsteigend sortierten Werte der Eingabe $u[1], ..., u[m]$ und $v[1], ..., v[m]$ führt.

In der als aufsteigend sortiert betrachteten Gesamtfolge der Eingabe $[u[1], ..., u[m]]$ und $[v[1], ..., v[m]]$ bezeichne $a[j]$ das j-te Element ($j = 1, ..., 2m$).

Offensichtlich ist $a[1] = \min\{u[1], v[1]\}$ und $a[2m] = \max\{u[m], v[m]\}$. Also erzeugt OEM($2m$) korrekt das kleinste und größte Element der gesamten Eingabefolge.

Es sei i ein fester Index mit $1 \le i \le m-1$.

Man betrachte die Elemente (Werte) $c[1], c[2], ..., c[i], c[i+1]$. Das Element $c[i+1]$ ist größer oder gleich a für mindestens $i+1$ Elemente a aus den Teillisten $u_{ungerade}$ und $v_{ungerade}$. Weiterhin gibt es zu jedem dieser Elemente a (außer für $u[1]$ und $v[1]$) ein Element mit *geradem* Index, das kleiner oder gleich a ist. Daher ist $c[i+1]$ größer oder gleich b für mindestens $i+1-2 = i-1$ Elemente b mit geradem Index, d.h. aus den Teillisten u_{gerade} und v_{gerade}. Insgesamt ist $c[i+1] \ge c$ für mindestens $i+1+i-1 = 2i$ Elemente c aus der Eingabe $[u[1], ..., u[m]]$ und $[v[1], ..., v[m]]$, also

$$c[i+1] \ge a[2i].$$

Entsprechend zeigt man, dass mindestens $2i$ viele Elemente der Eingabe $[u[1], ..., u[m]]$ und $[v[1], ..., v[m]]$ kleiner oder gleich $d[i]$ sind, d.h.

$$d[i] \ge a[2i].$$

Das $(2i+1)$-te Element $a[2i+1]$ in der als aufsteigend sortiert betrachteten Gesamtfolge der Eingabe $[u[1], ..., u[m]]$ und $[v[1], ..., v[m]]$ ist größer oder gleich a für mindestens $2i+1$ viele Elemente a aus $[u[1], ..., u[m]]$ und $[v[1], ..., v[m]]$. Aus der Teileingabe $[u[1], ..., u[m]]$ seien dieses t viele, aus der Teileingabe $[v[1], ..., v[m]]$ seien es s viele. Da die Zahl $t+s = 2i+1$ ungerade ist, ist einer der Zahlen t oder s ungerade und die andere gerade. Es sei etwa t ungerade und s gerade: $t = 2k+1$ und $s = 2l$ mit entsprechenden Zahlen k und l.
Da die Teileingaben $[u[1], ..., u[m]]$ bzw. $[v[1], ..., v[m]]$ jeweils aufsteigend sortiert sind, gilt:

$$u[1] \le a[2i+1],\ \ u[2] \le a[2i+1],\ \ u[3] \le a[2i+1],\ \ u[4] \le a[2i+1],\ ...,\ \ u[2k+1] \le a[2i+1]$$

und

$$v[1] \le a[2i+1],\ \ v[2] \le a[2i+1],\ \ v[3] \le a[2i+1],\ \ v[4] \le a[2i+1],\ \ ...,\ \ v[2l] \le a[2i+1]$$

und insbesondere auf die ungeraden Indizes beschränkt (Teillisten $u_{ungerade}$ und $v_{ungerade}$)

$$u[1] \le a[2i+1],\ \ u[3] \le a[2i+1],\ ...,\ \ u[2k+1] \le a[2i+1],$$
$$v[1] \le a[2i+1],\ \ v[3] \le a[2i+1],\ ...,\ \ v[2l-1] \le a[2i+1]$$

bzw. auf die geraden Indizes beschränkt (Teillisten u_{gerade} und v_{gerade})

$$u[2] \le a[2i+1],\ \ u[4] \le a[2i+1],\ ...,\ \ u[2k] \le a[2i+1],$$
$$v[2] \le a[2i+1],\ \ v[4] \le a[2i+1],\ ...,\ \ v[2l] \le a[2i+1].$$

Aus den Teillisten $u_{ungerade}$ und $v_{ungerade}$ sind also $k+1+l$ Elemente kleiner oder gleich $a[2i+1]$, aus den Teillisten u_{gerade} und v_{gerade} sind es $k+l$ viele.

Nach Konstruktion ist $2k+1=t$ und $2l=s$, also $2(k+l)+1=t+s=2i+1$ und damit $i=k+l$. Das bedeutet:

Aus den Teillisten $u_{ungerade}$ und $v_{ungerade}$ sind $i+1$ viele Elemente kleiner oder gleich $a[2i+1]$; aus den Teillisten u_{gerade} und v_{gerade} sind i viele Elemente kleiner oder gleich $a[2i+1]$. Daher ist

$$c[i+1] \le a[2i+1]$$

und

$$d[i] \le a[2i+1].$$

Insgesamt ergibt sich

$$a[2i] \le c[i+1] \le a[2i+1] \text{ und } a[2i] \le d[i] \le a[2i+1].$$

Daher ist $\min\{c[i+1], d[i]\} = a[2i]$ und $\max\{c[i+1], d[i]\} = a[2i+1]$, und der letzte Mischtakt erzeugt für $1 \le i \le m-1$ die korrekten Werte.

Der Wert $e[1]=c[1]$ ist das kleinste Element aus den Teillisten $u_{ungerade}$ und $v_{ungerade}$, $e[2m]=d[m]$ das größte Element aus den Teillisten u_{gerade} und v_{gerade}.

Die Anzahl $T_{\mathbf{OEM}}(n,p)$ paralleler Schritte, die die Mischphase OEM(n) bei Eingabe von zwei jeweils für sich sortierter Folgen der Länge $n/2$ mit p L/H-Elementen benötigt (die Größe p wird weiter unten berechnet) ist

$$T_{\mathbf{OEM}}(n,p) = \begin{cases} 1 & \text{für } n = 2 \\ T_{\mathbf{OEM}}(n/2,p) + 1 & \text{für } n > 2 \end{cases}.$$

Mit der in Kapitel 7.1.1 hergeleiteten Formel folgt $T_{\mathbf{OEM}}(n,p) \le \log_2(n) + 1$, d.h.

$$T_{\mathbf{OEM}}(n,p) \in O(\log(n)).$$

Bezeichnet $T_{\mathbf{parSORT}}(n,p)$ den parallelen Zeitaufwand zur Sortierung von n Zahlen mit dem beschriebenen Verfahren, so ist

$$T_{\mathbf{parSORT}}(n,p) \le \begin{cases} 1 & \text{für } n = 2 \\ T_{\mathbf{parSORT}}(n/2,p) + \log_2(n) + 1 & \text{für } n > 2 \end{cases}.$$

Ähnlich wie in Kapitel 7.1.1 (die dort hergeleitete Formel ist nicht direkt anwendbar, da $f(n) = \log_2(n)$ nicht mindestens linear wächst) ergibt sich

$$T_{\mathbf{parSORT}}(n,p) \le T_{\mathbf{parSORT}}\left(n/2^l, p\right) + l + l \cdot \log_2(n) - \sum_{i=0}^{l-1} \log_2\left(2^i\right) \quad \text{für } l = 1, \ldots, k$$

und damit (unter Beachtung von $T_{\mathbf{OEM}}(1,p) = 0$ und $n = 2^k$, d.h. $k = \log_2(n)$)

$$T_{\mathbf{parSORT}}(n,p) \le k + k \cdot \log_2(n) - \frac{k(k-1)}{2}, \text{ also}$$

$$T_{\mathbf{parSORT}}(n,p) \in O\left((\log(n))^2\right).$$

Gegenüber der seriellen Sortierung ergibt sich bei der hier beschriebenen parallelen Sortierung ein Zeitgewinn, jedoch werden viele L/H-Elemente, d.h. Prozessorelemente, benötigt. Um die Anzahl $p = p_{\mathbf{parSORT}}(n)$ der eingesetzten Prozessoren zur parallelen Sortierung von n Elementen abzuschätzen, wird zunächst die Anzahl $p_{\mathbf{OEM}}(n)$ der benötigten L/H-Elemente für die Mischphase OEM(n) berechnet:

$$p_{\mathbf{OEM}}(n) = \begin{cases} 1 & \text{für } n = 2 \\ 2 \cdot p_{\mathbf{OEM}}(n/2) + n/2 - 1 & \text{für } n > 2 \end{cases}.$$

Mit der in Kapitel 7.1.1 hergeleiteten Formel folgt $p_{\mathbf{OEM}}(n) \in O(n \cdot \log(n))$.

Für die Anzahl von L/H-Elementen $p_{\mathbf{parSORT}}(n)$ im Sortierverfahren gilt:

$$p_{\mathbf{parSORT}}(n) = \begin{cases} 1 & \text{für } n = 2 \\ 2 \cdot p_{\mathbf{parSORT}}(n/2) + O(n \cdot \log(n)) & \text{für } n > 2 \end{cases}.$$

Daraus folgt (vgl. Kapitel 7.1.1) $p_{\mathbf{parSORT}}(n) \in O\left(n \cdot (\log(n))^2\right)$.

Daher sind die Kosten $C_{\mathbf{parSORT}}(n, p_{\mathbf{parSORT}}(n)) = p_{\mathbf{parSORT}}(n) \cdot T_{\mathbf{parSORT}}(n, p_{\mathbf{parSORT}}(n))$ von der Ordnung $O\left(n \cdot (\log(n))^4\right)$, d.h. von einer weit von der optimalen Größenordnung $O(n \cdot \log(n))$ entfernten Größenordnung. Offensichtlich enthält das Sortiernetz ja auch sehr viele zu jedem Zeitpunkt inaktive L/H-Elemente. Diese Tatsache zeigt sich auch in der Effizienz (vgl. 8.1)

$$E_{\mathbf{A}}(n, p) \approx \frac{c_1 \cdot n \cdot \log(n)}{c_2 \cdot n \cdot (\log(n))^4} = \frac{c_1}{c_2} \cdot \frac{1}{(\log(n))^3}$$

des Verfahrens, die für große n beliebig klein wird.

Die **effektiven Kosten des Verfahrens** sei die Summe aller durchgeführten Schritte der *aktiven* Prozessorelemente. Diejenigen Prozessorelemente, die in einem parallelen Schritt gerade inaktiv sind, tragen nichts zu den effektiven Kosten bei. Offensichtlich ist die Anzahl aktiver Prozessorelemente in jedem parallelen Schritt von der Ordnung $O(n)$, so dass die effektiven Kosten von der Ordnung $O\left(n \cdot (\log(n))^2\right)$ sind, also immer noch um den Faktor log(n) von der optimalen Größenordnung entfernt.

Ein paralleler Sortieralgorithmus, der den optimalen Kostenwert erreichen, jedoch nicht die Übersichtlichkeit des oben beschriebenen Verfahrens aufweist, soll hier kurz skizziert werden.

Das folgende Verfahren lässt sich im CREW-PRAM-Modell implementieren. Die einzelnen RAM-Module sind hierbei Prozessoren mit lokalen Speichern und Zugriffsmöglichkeit auf einen globalen Speicherbereich, der die zu sortierenden Zahlen enthält.

Es wird angenommen, dass die zunächst unsortierten Zahlen $a_1, ..., a_n$ bereits im globalen Speicher zur Verfügung stehen und dass der Algorithmus $p \le n$ viele Prozessoren $P_1, ..., P_p$ nutzen kann. Zur Vereinfachung der Darstellung wird $p = 2^l$ angenommen. Die Eingabefolge

wird in p Teillisten $A[1]$, ..., $A[p]$ eingeteilt, die jeweils $\lfloor n/p \rfloor$ bzw. $\lceil n/p \rceil$ viele Elemente enthalten (ist $j = n \bmod p$, dann werden j Teillisten mit $\lceil n/p \rceil$ Elementen und $p - j$ viele Teillisten mit $\lfloor n/p \rfloor$ gebildet). Jeder Prozessor P_i (mit $i = 1, ..., p$) sortiert seriell die Teilliste $A[i]$, etwa mit dem Heapsort. Das Ergebnis ist jeweils eine sortierte Teilliste mit $\lfloor n/p \rfloor$ bzw. $\lceil n/p \rceil$ vielen Elementen. Jeweils zwei der p Teillisten werden mit einem parallelen Misch-Algorithmus zusammengemischt, wobei hierzu jeweils zwei Prozessoren parallel eingesetzt werden können. Dadurch sind alle Prozessoren beschäftigt. Das Ergebnis sind $p/2$ viele für sich sortierte Listen mit jeweils höchstens $2 \cdot \lceil n/p \rceil$ vielen Elementen. In der nächsten Phase werden jeweils zwei dieser Listen von vier parallel einsetzbaren Prozessoren zusammengemischt usw. bis nur noch eine dann sortierte Liste übrig bleibt. Abbildung 8.1.1-5 fasst die Vorgehensweise zusammen.

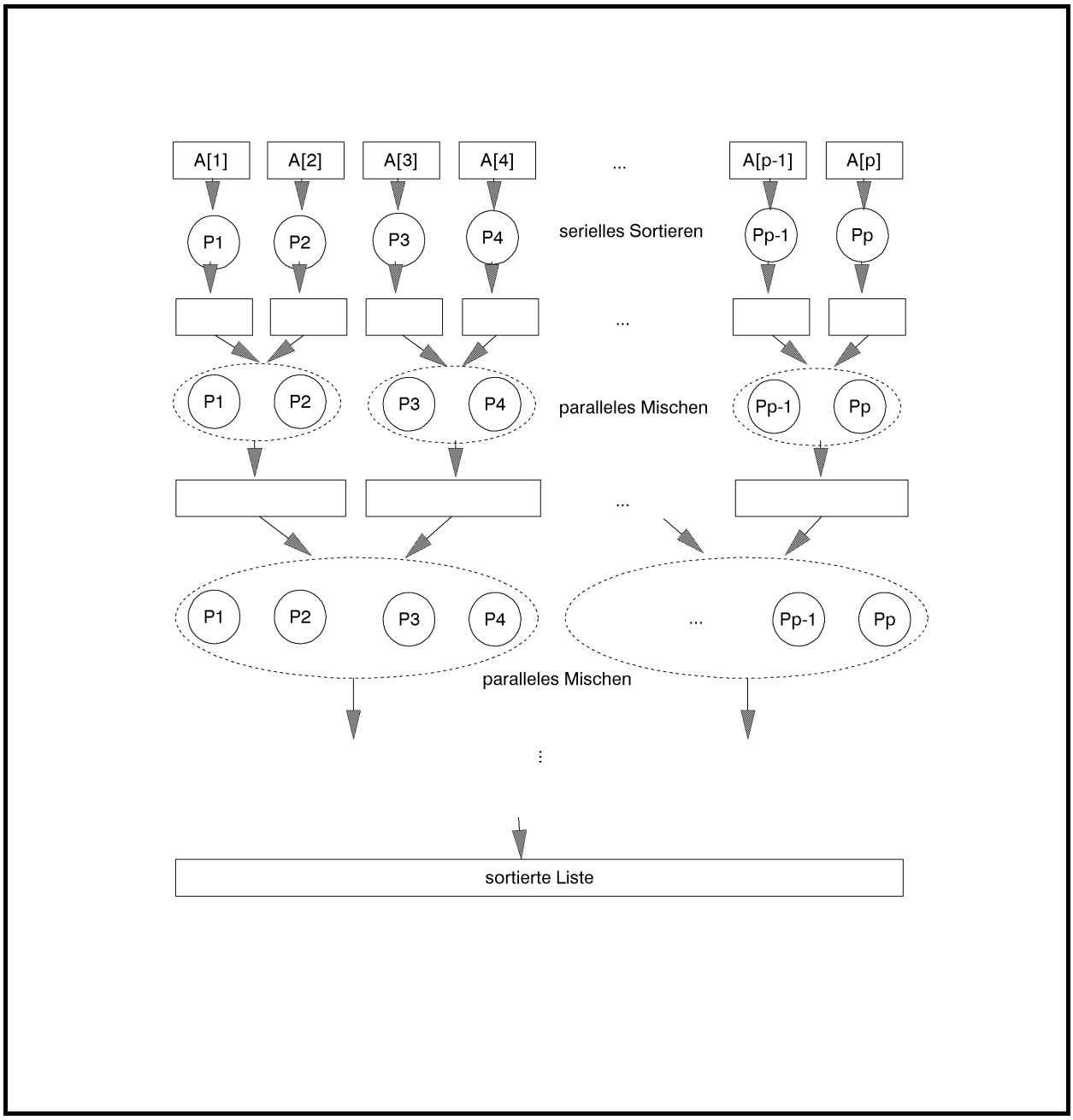


Abbildung 8.1.1-5: Verbessertes paralleles Sortierverfahren

Insgesamt werden $\log_2(p)$ viele Mischphasen ausgeführt. In jeder Mischphase ist der Quotient aus der Anzahl zu mischender Elemente und der eingesetzten Prozessoren $\leq \lceil n/p \rceil$. Es gibt parallele Mischverfahren für das CREW-PRAM-Modell, die zum parallelen Mischen zweier jeweils sortierter Listen mit k_1 und k_2 vielen Elementen mit $k_1 \leq k_2$ unter Einsatz von $q \leq k_2$ vielen Prozessoren einen parallelen Zeitaufwand der Ordnung $O(k_2/q + \log(k_2))$ verursachen (siehe z.B. [CHA], am Ende von Kapitel 8.1.1 wird das Verfahren überblicksmäßig beschrieben). Im vorliegenden Fall ist das Verhältnis $k_2/q \leq \lceil n/p \rceil$ und $k_2 \leq n$, so dass der parallele Aufwand aller Mischphasen zusammen von der Ordnung $O(\log(p) \cdot (n/p + \log(n)))$ ist. Zusammen mit den vorgeschalteten parallel ausgeführten Sortierungen der Teillisten entsteht so ein zeitlicher Aufwand der Ordnung

$O((n/p)\cdot\log(n/p)+\log(p)\cdot(n/p+\log(n)))$,

d.h. von der Ordnung $O((n/p)\cdot\log(n)+\log(p)\cdot\log(n))$.

Die Kosten des Verfahrens sind daher von der Ordnung $O((n)\cdot\log(n)+p\cdot\log(p)\cdot\log(n))$. Diese Größenordnung ist für $p \le n/\log(n)$ optimal. Die Effizienz ist in diesem Fall gleich 1: alle Prozessorelemente werden in jedem parallelen Schritt eingesetzt.

Die folgende Beschreibung gibt einen Überblick über ein paralleles Mischverfahren (im CREW-PRAM-Modell), das bei Eingabe zweier jeweils sortierter Listen mit k_1 und k_2 vielen Elementen mit $k_1 \le k_2$ unter Einsatz von $q \le k_2$ vielen Prozessoren einen parallelen Zeitaufwand der Ordnung $O(k_2/q+\log(k_2))$ verursacht ([CHA]).

Der Algorithmus erhält als Eingabe zwei jeweils aufsteigend sortierte Folgen $a_1,...,a_{k_1}$ und $b_1,...,b_{k_2}$ mit $k_1 \le k_2$ und mischt diese zu einer aufsteigend sortierten Ausgabefolge zusammen. Das Verfahren verläuft in zwei Phasen: die erste Phase bereitet die beiden Folgen so auf, dass sie in der zweiten Phase effizient gemischt werden können.

In der ersten Phase werden beide Eingabefolgen $a_1,...,a_{k_1}$ und $b_1,...,b_{k_2}$ in q Intervalle etwa gleicher Länge aufgeteilt, indem in einem parallelen Schritt aus beiden Folgen jeweils $q-1$ Elemente bestimmt werden. Diese Elemente seien $a'_1,...,a'_{q-1}$ aus $a_1,...,a_{k_1}$ bzw. $b'_1,...,b'_{q-1}$ aus $b_1,...,b_{k_2}$: das erste Intervall in $a_1,...,a_{k_1}$ beginnt mit a_1, das zweite Intervall in $a_1,...,a_{k_1}$ mit a'_1, das dritte Intervall in $a_1,...,a_{k_1}$ mit a'_2 usw. bis zum q-ten Intervall, das mit a'_{q-1} beginnt. Entsprechend stellen die Elemente $b_1, b'_1,...,b'_{q-1}$ jeweils das erste Element eines Intervalls in $b_1,...,b_{k_2}$ dar. Die Auswahl durch den Prozessor P_i für $i = 1, ..., q-1$ erfolgt nach der Regel

$$a'_i := a_{i\cdot\lceil k_1/q\rceil};$$
$$b'_i := b_{i\cdot\lceil k_2/q\rceil}$$

Diese beiden Teilfolgen $a'_1,...,a'_{q-1}$ und $b'_1,...,b'_{q-1}$ werden zu einer Folge $l_1,...,l_{2q-2}$ zusammengemischt. Dabei wird für jedes l_j festgehalten, aus welcher der Teilfolgen $a'_1,...,a'_{q-1}$ oder $b'_1,...,b'_{q-1}$ das Element stammt und welchen Index es in der jeweiligen Teilfolge hatte. Hierzu werden parallel $q-1$ Prozessoren eingesetzt. Prozessor P_i mit $i = 1, ..., q-1$ führt dabei aus:

finde in $b'_1, ..., b'_{q-1}$ den kleinsten Index j mit $a'_i < b'_j$;

```
IF (b'_j existiert) THEN BEGIN
                        l_{i+j-1} := a'_i ;
                        vermerke: l_{i+j-1} kommt aus a'_1, ..., a'_{q-1} und dort von Position i;
                    END
                ELSE BEGIN
                        l_{i+q-1} := a'_i ;
                        vermerke: l_{i+q-1} kommt aus a'_1, ..., a'_{q-1} und dort von Position i;
                    END;
```

finde in $a'_1, ..., a'_{q-1}$ den kleinsten Index j mit $b'_i < a'_j$;

```
IF (a'_j existiert) THEN BEGIN
                        l_{i+j-1} := b'_i ;
                        vermerke: l_{i+j-1} kommt aus b'_1, ..., b'_{q-1} und dort von Position i;
                    END
                ELSE BEGIN
                        l_{i+q-1} := b'_i ;
                        vermerke: l_{i+q-1} kommt aus b'_1, ..., b'_{q-1} und dort von Position i;
                    END;
```

Der Prozessor P_i durchsucht in diesem Programmabschnitt die Listen $a'_1, ..., a'_{q-1}$ und $b'_1, ..., b'_{q-1}$ der Länge $q - 1$ und führt dann noch eine konstante Anzahl von Schritten aus. Da beide Teillisten aufsteigend sortiert sind, kann zum Durchsuchen Binärsuche (Kapitel 6.2.1) eingesetzt werden, so dass dieser Programmabschnitt in jedem Prozessor einen Aufwand der Ordnung $O(\log(q))$ erfordert.

Damit ist die erste Phase beendet.

Die zweite Phase bekommt als Eingabe die Folgen $a_1, ..., a_{k_1}$, $b_1, ..., b_{k_2}$ und $l_1, ..., l_{2q-2}$. In der ersten Phase waren die beiden Folgen $a_1, ..., a_{k_1}$ und $b_1, ..., b_{k_2}$ in Teilfolgen (Abschnitte) jeweils etwa gleicher Länge aufgeteilt worden; in $l_1, ..., l_{2q-2}$ ist festgehalten, bei welchen Elementen die einzelnen Abschnitte beginnen und wie die einzelnen Abschnitte beider Folgen zueinander liegen (Abbildung 8.1.1-6).

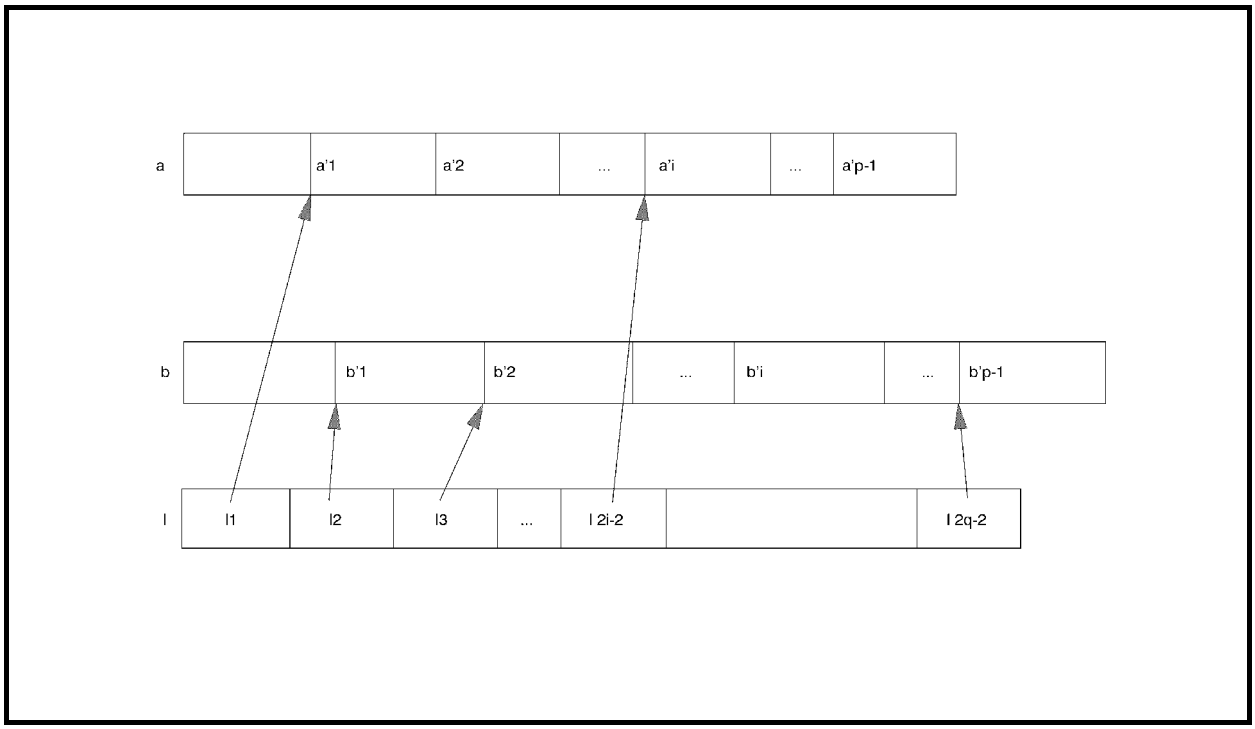


Abbildung 8.1.1-6: Teilverfahren beim parallelen Mischen im CREW-PRAM-Modell

Jeder Prozessor P_i für i = 1, ..., q ist zuständig für das serielle Mischen einer Teilfolge aus $a_1, ..., a_{k_1}$ mit einer Teilfolge aus $b_1, ..., b_{k_2}$; das Ergebnis fügt er in die Ausgabefolge ein.

Prozessor P_1 mischt die beiden Teilfolgen, die in $a_1, ..., a_{k_1}$ mit a_1 und in $b_1, ..., b_{k_2}$ mit b_1 beginnen und kleiner als l_2 sind.

Prozessor P_q mischt die beiden Teilfolgen aus $a_1, ..., a_{k_1}$ bzw. $b_1, ..., b_{k_2}$, die größer als l_{2q-2} sind.

Jeder Prozessor P_i mit i = 2, ..., q stellt fest, aus welcher der Listen $a_1, ..., a_{k_1}$ und $b_1, ..., b_{k_2}$ das Element l_{2i-2} stammt; dieses Element ist der eine Startwert für das Mischen. Kommt das Element aus $a_1, ..., a_{k_1}$ (bzw. aus $b_1, ..., b_{k_2}$), so wird in $b_1, ..., b_{k_2}$ (bzw. in $a_1, ..., a_{k_1}$) das kleinste Element gesucht, das größer als dieses Element ist. Damit sind die Startelemente für P_i gefunden. Er mischt nun die Elemente, die zwischen dem $(2i-2)$-ten und dem $(2i)$-ten Element aus $l_1, ..., l_{2q-2}$ liegen.

Die zweite Phase lautet (in Pseudocode-Notation) für Prozessor P_i mit i = 1, ..., q – 1:

```
{ Finden die Startwerte der zu mischenden Teilfolgen: }
IF i = 1 THEN BEGIN
```

der Startwert in $a_1, ..., a_{k_1}$ steht an Position $s_a = 1$;

der Startwert in $b_1, ..., b_{k_2}$ steht an Position $s_b = 1$;

```
            END
        ELSE BEGIN
                IF (l_{2i-2} kommt aus a_1,...,a_{k_1})
                THEN BEGIN
                        finde den kleinsten Index j mit l_{2i-2} < b_j;
                        der Startwert in a_1,...,a_{k_1} steht an Position
                                s_a = (die bei l_{2i-2} vermerkte Position)·⌈k_1/q⌉;
                        der Startwert in b_1,...,b_{k_2} steht an Position s_b = j;
                      END
                ELSE BEGIN
                        finde den kleinsten Index j mit l_{2i-2} < a_j;
                        der Startwert in a_1,...,a_{k_1} steht an Position s_a = j;
                        der Startwert in b_1,...,b_{k_2} steht an Position
                                s_b = (die bei l_{2i-2} vermerkte Position)·⌈k_2/q⌉;
                      END;
            END;

{ Mischen der Teilfolgen, die an den jeweiligen
  Startwerten beginnen:                          }
IF i < q
THEN BEGIN
        mische die Teilliste aus a_1,...,a_{k_1} mit der Teilliste aus b_1,...,b_{k_2}, die bei den
        jeweils ermittelten Positionen s_a bzw. s_b beginnen, bis ein Element ≥ l_{2i}
        erreicht ist, und füge das Ergebnis in die Ausgabefolge ab Position s_a + s_b - 1 ein;
     END
ELSE BEGIN
        mische die Teilliste aus a_1,...,a_{k_1} mit der Teilliste aus b_1,...,b_{k_2}, die bei den
        jeweils ermittelten Positionen s_a bzw. s_b beginnen, bis kein Element in a_1,...,a_{k_1}
        und b_1,...,b_{k_2} übrig bleibt, und füge das Ergebnis in die Ausgabefolge
        ab Position s_a + s_b - 1 ein;
     END;
```

Das folgende Beispiel erläutert das Verfahren. Die Eingabefolgen seien

$A = [5,\ 8,\ 9,\ 12,\ 16,\ 17,\ 19,\ 22]$ und
$B = [1,\ 3,\ 7,\ 10,\ 11,\ 14,\ 18,\ 21,\ 24,\ 25,\ 28]$,

d.h. $k_1 = 8$ und $k_2 = 11$. Die Anzahl der Prozessoren sei $q = 3$.

Im ersten Schritt der ersten Phase werden die jeweils 3 Intervalle der Eingabefolgen bestimmt. Dazu werden aus jeder Folge 2 Elemente entnommen, die die ersten Elemente eines jeweiligen Intervalls sind.

$$a_1' = a_{1\cdot\lceil k_1/q\rceil} = a_3 = 9\,, \qquad a_2' = a_{2\cdot\lceil k_1/q\rceil} = a_6 = 17\,, \quad A' = [a_1', a_2'] = [9, 17]$$
$$b_1' = b_{1\cdot\lceil k_2/q\rceil} = b_4 = 10\,, \qquad b_2' = b_{2\cdot\lceil k_2/q\rceil} = b_8 = 21\,, \quad B' = [b_1', b_2'] = [10, 21].$$

Die hieraus zusammengemischte Folge lautet (die erste Komponente ist das Element, die zweite Komponente die Angabe der Ursprungsfolge, die dritte Komponente die Position innerhalb $[a_1', a_2']$ bzw. $[b_1', b_2']$):

$$[l_1, l_2, l_3, l_4] = [(9, A', 1), (10, B', 1), (17, A', 2), (21, B', 2)].$$

Zur Erzeugung dieser Folge arbeiten die Prozessoren P_1, und P_2 parallel wie folgt:

Prozessor P_1: sucht in B' den kleinsten Index j mit $a_1' = 9 < b_j'$, also $j = 1$

$l_{1+j-1} = l_1 = (9, A', 1)$;

sucht in A' den kleinsten Index j mit $b_1' = 10 < a_j'$, also $j = 2$

$l_{1+j-1} = l_2 = (10, B', 1)$.

Prozessor P_2: sucht in B' den kleinsten Index j mit $a_2' = 17 < b_j'$, also $j = 2$

$l_{2+j-1} = l_3 = (17, A', 2)$;

sucht in A' den kleinsten Index j mit $b_2' = 21 < a_j'$; dieser existiert nicht, also

$l_{2+q-1} = l_4 = (21, B', 2)$.

Damit beginnt die zweite Phase.

$A = [5,\ 8,\ \boxed{9,\ 12,\ 16},\ 17,\ 19,\ 22]$

$B = [1,\ 3,\ 7,\ \boxed{10,\ 11,\ 14,\ 18,}\ 21,\ 24,\ 25,\ 28]$

$[(9, A', 1), (10, B', 1), (17, A', 2), (21, B', 2)]$

Prozessor P_1: Startwerte $s_a = 1$ und $s_b = 1$; es werden die Teillisten aus *A* bzw. *B* gemischt, die bei s_a bzw. s_b beginnen und deren Werte kleiner als die erste Komponente 10 in l_2 sind; die Ausgabe wird ab Position 1 eingefügt:
$[1,\ 3,\ 5,\ 7,\ 8,\ 9]$

Prozessor P_2: l_2 kommt aus B', d.h. aus *B*, also sucht P_2 den kleinsten Index *j*, so dass die erste Komponente in l_2 kleiner als a_j ist; hier: Startwert in *A* bei $s_a = 4$ (Beginn in *A* bei $a_4 = 12$); Startwert in *B* bei $s_b = 1 \cdot \lceil k_2/q \rceil = 4$ (Beginn in *B* bei $b_4 = 10$) ; es werden die Teillisten aus *A* bzw. *B* gemischt, die bei s_a bzw. s_b beginnen und deren Werte kleiner als die erste Komponente 21 in l_4 sind; die Ausgabe wird ab Position 7 eingefügt:
$[., .,\ .,\ .,\ .,\ .,\ 10,\ 11,\ 12,\ 14,\ 16,\ 17,\ 18,\ 19]$

Prozessor P_3: l_4 kommt aus B', d.h. aus *B*, also sucht P_3 den kleinsten Index *j*, so dass die erste Komponente in l_4 kleiner als a_j ist; hier: Startwert in *A* bei $s_a = 8$ (Beginn in *A* bei $a_8 = 22$); Startwert in *B* bei $s_b = 2 \cdot \lceil k_2/q \rceil = 8$ (Beginn in *B* bei $b_8 = 21$) ; es werden die restlichen Teillisten aus *A* bzw. *B* gemischt, die bei s_a bzw. s_b beginnen; die Ausgabe wird ab Position 15 eingefügt:
$[., .,\ .,\ .,\ .,\ .,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ \_,\ 21,\ 22,\ 24,\ 25,\ 28]$

Bei Einsatz von Binärsuche ist der Aufwand für die Ermittlung der Positionen s_a und s_b der Startwerte des Mischvorgangs in jedem Prozessor von der Ordnung $O(\log(k_2))$. Die Folge $l_1, ..., l_{2q-2}$ teilt die schließlich entstehende Ausgabefolge in $2q-1$ viele Teilfolgen, deren Längen durch $\lceil k_1/q \rceil + \lceil k_2/q \rceil$ beschränkt sind. Der anschließende serielle Mischvorgang ist daher von der Ordnung $O(k_2/q)$.

Insgesamt ergibt sich ein paralleler Zeitaufwand der Ordnung $O(k_2/q + \log(k_2))$ mit q Prozessoren. Die entstehenden Kosten sind von der Ordnung $O(k_2 + q \cdot \log(k_2))$; dieser Wert ist für $q \le k_2/\log(k_2)$ optimal.

8.1.2 Parallele Matrixoperationen

In numerischen Anwendungen kommen häufig **Matrix- und Vektoroperationen** wie Multiplikation von Matrizen, Invertierung einer Matrix, Determinatenberechnung oder Multiplikation von Matrizen und Vektoren vor. Spezielle Systemarchitekturen (Vektorrechner, systolische Felder) und spezielle (Erweiterungen von) Programmiersprachen tragen diesen Anwendungen Rechnung. In den folgenden Beispielen werden parallele Algorithmen und Architekturen für die Matrixmultiplikation und die Multiplikation einer Matrix mit einem Vektor vorgestellt. Die Komplexität der Algorithmen für die wichtigsten übrigen Matrixoperationen wird im wesentlichen durch die Komplexität der Matrixmultiplikation bestimmt ([AHU]). Des weiteren erfolgt eine Beschränkung auf numerische Matrizen; spezielle Verfahren beispielsweise für Boolesche Matrizen werden in der angegebenen Literatur beschrieben ([AHU], [CHA]).

Für eine **Matrix** $\mathbf{A}_{(m,n)} = [a_{i,j}]$ bezeichne m die Anzahl der Zeilen und n die Anzahl der Spalten. Das Element $a_{i,j}$ ist das Element im Schnittpunkt der i-ten Zeile mit der j-ten Spalte. Numerierung von Zeilen und Spalten erfolgen jeweils beginnend bei 1.

Das Produkt zweier Matrizen $\mathbf{A}_{(m,n)} = [a_{r,i}]$ und $\mathbf{B}_{(n,k)} = [b_{i,s}]$ ist durch die Matrix $\mathbf{C}_{(m,k)} = [c_{r,s}] = \mathbf{A} \cdot \mathbf{B}$ mit

$$c_{r,s} = \sum_{i=1}^{n} a_{r,i} \cdot b_{i,s} \quad \text{für } r = 1, ..., m,\ s = 1, ..., n$$

definiert.

Im folgenden werden zur Vereinfachung der Darstellung nur **quadratische Matrizen** mit n Zeilen und n Spalten betrachtet.

Die Berechnung des Produkts $\mathbf{A} \cdot \mathbf{B}$ zweier Matrizen $\mathbf{A}_{(n,n)}$ und $\mathbf{B}_{(n,n)}$ erfordert die Berechnung von n^2 vielen Matrixelementen. Die durch die Definition $c_{r,s} = \sum_{i=1}^{n} a_{r,i} \cdot b_{i,s}$ des Matrixproduktes bestimmte Formel legt einen sequentiellen Basisalgorithmus nahe, der insgesamt $n^2 \cdot n$ viele Multiplikationen und $n^2 \cdot (n-1)$ viele Additionen, d.h. eine Anzahl arithmetischer

Operationen der Ordnung $O(n^3)$ erfordert. Der aus der Literatur bekannte Strassenalgorithmus ([AHU]) kommt sogar mit nur $O(n^{\log_2(7)})$ aus.

Die Aufgabe besteht im folgenden in der Berechnung des Matrixproduktes $\mathbf{C} = \mathbf{C}_{(n,n)} = [c_{r,s}] = \mathbf{A} \cdot \mathbf{B}$ mit zwei Eingabematrizen $\mathbf{A} = \mathbf{A}_{(n,n)} = [a_{i,j}]$ und $\mathbf{B} = \mathbf{B}_{(n,n)} = [b_{i,j}]$.

Unter Einsatz von n^2 vielen Prozessoren $P_1, ..., P_{n^2}$ kann für das CREW-PRAM-Modell folgender paralleler Algorithmus entworfen werden, der eine Anzahl paralleler Schritte der Ordnung $O(n)$ ausführt und damit bezüglich der parallelen Kosten mit dem üblichen seriellen Basisalgorithmus übereinstimmt:

Die Prozessoren werden durch zwei Indizes i und j mit $i = 1, ..., n$ und $j = 1, ..., n$ indiziert: der Prozessor $P_{i,j}$ ist der $((i-1) \cdot n + j)$-te Prozessor in der Aufzählung $P_1, ..., P_{n^2}$.

Der Prozessor $P_{i,j}$ mit $i = 1, ..., n$ und $j = 1, ..., n$ berechnet das Matrixelement $c_{i,j}$ mit $2n + 1$ vielen arithmetischen Operationen:

```
c_{i,j} := 0 ;
FOR k := 1 TO n DO
   c_{i,j} := c_{i,j} + a_{i,k} * b_{k,j} ;
```

Offensichtlich wird in den einzelnen parallelen Schritten von unterschiedlichen Prozessoren lesend auf dieselben Elemente der Ausgangsmatrizen zugegriffen. Dieses ist im CREW-PRAM-Modell zulässig. Die Matrixelemente müssen in einem gemeinsam zugreifbaren Speicher abgelegt und für jeden Prozessor in jedem Schritt über ihren Index erreichbar sein.

Das Verfahren soll exemplarisch an der Berechnung des Produkts zweier Matrizen **A** und **B** vom Typ (3, 3) gezeigt werden ($n = 3$). Dazu werden die Zwischenergebnisse der parallelen Schritte angegeben.

Im ersten parallelen Schritt ($k = 1$) nach der Initialisierung mit 0 berechnet Prozessor $P_{i,j}$ mit $i = 1, 2, 3$ und $j = 1, 2, 3$ das Produkt aus dem Element in der i-ten Zeile und der 1. Spalte von **A** und der 1. Zeile und j-ten Spalte von **B**:

$$[c_{1,1} \quad c_{1,2} \quad c_{1,3}] = a_{1,1} \cdot [b_{1,1} \quad b_{1,2} \quad b_{1,3}]$$
$$[c_{2,1} \quad c_{2,2} \quad c_{2,3}] = a_{2,1} \cdot [b_{1,1} \quad b_{1,2} \quad b_{1,3}]$$
$$[c_{3,1} \quad c_{3,2} \quad c_{3,3}] = a_{3,1} \cdot [b_{1,1} \quad b_{1,2} \quad b_{1,3}]$$

Im zweiten parallelen Schritt ($k = 2$) wird in Prozessor $P_{i,j}$ das Produkt aus dem Element in der i-ten Zeile und der 2. Spalte von **A** und der 2. Zeile und j-ten Spalte von **B** hinzuaddiert:

$$\begin{aligned}
\begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \end{bmatrix} &= a_{1,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix} + a_{1,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} \\
\begin{bmatrix} c_{2,1} & c_{2,2} & c_{2,3} \end{bmatrix} &= a_{2,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix} + a_{2,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} \\
\begin{bmatrix} c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} &= a_{3,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix} + a_{3,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix}
\end{aligned}$$

Im dritten parallelen Schritt ($k = 3$) wird in Prozessor $P_{i,j}$ das Produkt aus dem Element in der i-ten Zeile und der 3. Spalte von **A** und der 3. Zeile und j-ten Spalte von **B** hinzuaddiert:

$$\begin{aligned}
\begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \end{bmatrix} &= a_{1,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix} + a_{1,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} + a_{1,3} \cdot \begin{bmatrix} b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \\
\begin{bmatrix} c_{2,1} & c_{2,2} & c_{2,3} \end{bmatrix} &= a_{2,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix} + a_{2,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} + a_{2,3} \cdot \begin{bmatrix} b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \\
\begin{bmatrix} c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} &= a_{3,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix} + a_{3,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} + a_{3,3} \cdot \begin{bmatrix} b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}
\end{aligned}$$

Durch Umordnung der Einzelschritte des gesamten parallelen Algorithmus lässt sich erreichen, dass unterschiedliche Prozessoren in jedem parallelen auf unterschiedliche Elemente der Ausgangsmatrizen zugreifen. Dadurch entsteht ein paralleler Algorithmus für das restriktivere EREW-PRAM-Modell. Zusätzlich kann erreicht werden, dass die Elemente der Matrizen **A** und **B** in einem einzigen parallelen Schritt in die Prozessoren eingelesen werden und vor jedem weiteren parallelen Schritt jedem Prozessor diejenigen Matrixelemente (von anderen Prozessoren) zur Verfügung gestellt werden, die er dann gerade benötigt.

Das Verfahren soll zunächst an obigem Beispiel erläutert werden, nachdem alle Prozessoren $c_{i,j}$ mit 0 initialisiert haben.

1. paraller Schritt:

$c_{1,1}$:= Operation des Prozessors $P_{1,1}$ aus dem 1. Schritt mit Ergebnis $a_{1,1} \cdot b_{1,1}$

$c_{2,1}$:= Operation des Prozessors $P_{2,1}$ aus dem 2. Schritt mit Ergebnis $a_{2,2} \cdot b_{2,1}$

$c_{3,1}$:= Operation des Prozessors $P_{3,1}$ aus dem 3. Schritt mit Ergebnis $a_{3,3} \cdot b_{3,1}$

$c_{1,2}$:= Operation des Prozessors $P_{1,2}$ aus dem *2*. Schritt mit Ergebnis $a_{1,2} \cdot b_{2,2}$

$c_{2,2}$:= Operation des Prozessors $P_{2,2}$ aus dem *3*. Schritt mit Ergebnis $a_{2,3} \cdot b_{3,2}$

$c_{3,2}$:= Operation des Prozessors $P_{3,2}$ aus dem *1*. Schritt mit Ergebnis $a_{3,1} \cdot b_{1,2}$

$c_{1,3}$:= Operation des Prozessors $P_{1,3}$ aus dem 3. Schritt mit Ergebnis $a_{1,3} \cdot b_{3,3}$

$c_{2,3}$:= Operation des Prozessors $P_{2,3}$ aus dem 1. Schritt mit Ergebnis $a_{2,1} \cdot b_{1,3}$

$c_{3,3}$:= Operation des Prozessors $P_{3,3}$ aus dem 2. Schritt mit Ergebnis $a_{3,2} \cdot b_{2,3}$

2. paraller Schritt:

$c_{1,1}$:= Operation des Prozessors $P_{1,1}$ aus dem 3. Schritt mit Ergebnis $a_{1,1} \cdot b_{1,1} + a_{1,3} \cdot b_{3,1}$

$c_{2,1}$:= Operation des Prozessors $P_{2,1}$ aus dem 1. Schritt mit Ergebnis $a_{2,2} \cdot b_{2,1} + a_{2,1} \cdot b_{1,1}$

$c_{3,1}$:= Operation des Prozessors $P_{3,1}$ aus dem 2. Schritt mit Ergebnis $a_{3,3} \cdot b_{3,1} + a_{3,2} \cdot b_{2,1}$

$c_{1,2}$:= Operation des Prozessors $P_{1,2}$ aus dem *1*. Schritt mit Ergebnis $a_{1,2} \cdot b_{2,2} + a_{1,1} \cdot b_{1,2}$

$c_{2,2}$:= Operation des Prozessors $P_{2,2}$ aus dem *2*. Schritt mit Ergebnis $a_{2,3} \cdot b_{3,2} + a_{2,2} \cdot b_{2,2}$

$c_{3,2}$:= Operation des Prozessors $P_{3,2}$ aus dem *3*. Schritt mit Ergebnis $a_{3,1} \cdot b_{1,2} + a_{3,3} \cdot b_{3,2}$

$c_{1,3}$:= Operation des Prozessors $P_{1,3}$ aus dem 2. Schritt mit Ergebnis $a_{1,3} \cdot b_{3,3} + a_{1,2} \cdot b_{2,3}$

$c_{2,3}$:= Operation des Prozessors $P_{2,3}$ aus dem 3. Schritt mit Ergebnis $a_{2,1} \cdot b_{1,3} + a_{2,3} \cdot b_{3,3}$

$c_{3,3}$:= Operation des Prozessors $P_{3,3}$ aus dem 1. Schritt mit Ergebnis $a_{3,2} \cdot b_{2,3} + a_{3,1} \cdot b_{1,3}$

3. paraller Schritt:

$c_{1,1}$:= Operation des Prozessors $P_{1,1}$ aus dem 2. Schritt mit Ergebnis

$a_{1,1} \cdot b_{1,1} + a_{1,3} \cdot b_{3,1} + a_{1,2} \cdot b_{2,1}$

$c_{2,1}$:= Operation des Prozessors $P_{2,1}$ aus dem 3. Schritt mit Ergebnis

$a_{2,2} \cdot b_{2,1} + a_{2,1} \cdot b_{1,1} + a_{2,3} \cdot b_{3,1}$

$c_{3,1}$:= Operation des Prozessors $P_{3,1}$ aus dem 1. Schritt mit Ergebnis

$a_{3,3} \cdot b_{3,1} + a_{3,2} \cdot b_{2,1} + a_{3,1} \cdot b_{1,1}$

$c_{1,2}$:= Operation des Prozessors $P_{1,2}$ aus dem *3*. Schritt mit Ergebnis

$a_{1,2} \cdot b_{2,2} + a_{1,1} \cdot b_{1,2} + a_{1,3} \cdot b_{3,2}$

$c_{2,2}$:= Operation des Prozessors $P_{2,2}$ aus dem *1*. Schritt mit Ergebnis

$a_{2,3} \cdot b_{3,2} + a_{2,2} \cdot b_{2,2} + a_{2,1} \cdot b_{1,2}$

$c_{3,2}$:= Operation des Prozessors $P_{3,2}$ aus dem *2*. Schritt mit Ergebnis

$a_{3,1} \cdot b_{1,2} + a_{3,3} \cdot b_{3,2} + a_{3,2} \cdot b_{2,2}$

$c_{1,3}$:= Operation des Prozessors $P_{1,3}$ aus dem 1. Schritt mit Ergebnis

$a_{1,3} \cdot b_{3,3} + a_{1,2} \cdot b_{2,3} + a_{1,1} \cdot b_{1,3}$

$c_{2,3}$:= Operation des Prozessors $P_{2,3}$ aus dem 2. Schritt mit Ergebnis

$a_{2,1} \cdot b_{1,3} + a_{2,3} \cdot b_{3,3} + a_{2,2} \cdot b_{2,3}$

$c_{3,3}$:= Operation des Prozessors $P_{3,3}$ aus dem 3. Schritt mit Ergebnis

$a_{3,2} \cdot b_{2,3} + a_{3,1} \cdot b_{1,3} + a_{3,3} \cdot b_{3,3}$

Vor dem ersten und anschließend in jedem parallelen Schritt werden also die ursprünglichen parallelen Schritte der einzelnen Prozessoren zyklisch vertauscht.

Zur genaueren Beschreibung des Verfahrens im allgemeinen Fall wird angenommen, dass jeder Prozessor $P_{i,j}$ mit $i = 1, ..., n$ und $j = 1, ..., n$ über drei Register $AREG_{i,j}$, $BREG_{i,j}$ und $CREG_{i,j}$ verfügt (die Indizes an den Registernamen sind nur zur Verdeutlichung der Zugehörigkeit zu einem Prozessor angebracht). Zusätzlich verfügt jeder Prozessor über weitere lokale Variablen (Speicherelemente). Die Prozessoren sind in einem Netzwerk angeordnet, in dem jeder Prozessor mit vier direkten Nachbarn verbunden ist, so dass zwischen Nachbarn Werte der jeweiligen Register *AREG*, *BREG* und *CREG* ausgetauscht werden können; insbesondere kann ein Prozessor die Registerinhalte der Nachbarn, mit denen er verbunden ist, lesen. Dabei wird angenommen, dass ein lesender Prozessor den Inhalt liest, der zu Beginn eines Taktes im Register des Nachbarprozessors steht, so dass der Nachbarprozessor im gleichen Takt einen neuen Wert in das Register schreiben kann. Auf die weiteren lokalen Variablen eines Prozessors hat ein mit dem Prozessor verbundener Nachbar keinen Zugriff.

Die Prozessoren sind in einem quadratischen Schema der Größe n^2 angeordnet (Abbildung 8.1.2-1):

Zeilenweise ist $P_{i,j}$ für $i = 1, ..., n$ und $j = 1, ..., n-1$ mit $P_{i,j+1}$ verbunden, und $P_{i,n}$ ist mit $P_{i,1}$ verbunden.
Spaltenweise ist $P_{i,j}$ für $i = 1, ..., n-1$ und $j = 1, ..., n$ mit $P_{i+1,j}$ verbunden, und $P_{n,j}$ ist mit $P_{1,j}$ verbunden.

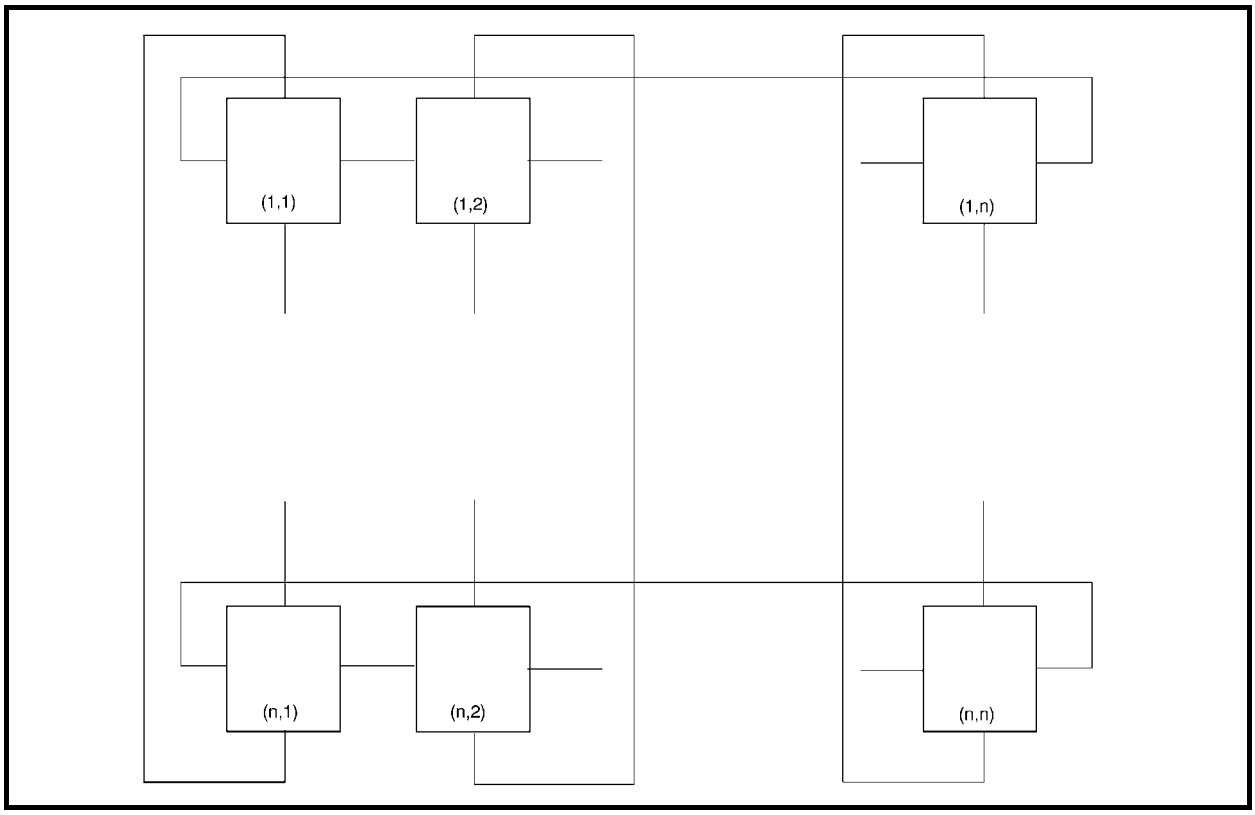


Abbildung 8.1.2-1: Prozessoranordnung zur Matrixmultiplikation

Die Register $AREG_{i,j}$ bzw. $BREG_{i,j}$ des Prozessors $P_{i,j}$ mit $i = 1, ..., n$ und $j = 1, ..., n$ werden mit den Elementen $a_{i,j}$ bzw. $b_{i,j}$ der Matrizen **A** bzw. **B** initialisiert (Abbildung 8.1.2-2 auf der linken Seite). Anschließend werden die Elemente der Zeilen von **A** und der Spalten von **B** (über die Verbindungen der Prozessoren) so verschoben, dass jeder Prozessor $P_{i,j}$ im ersten parallelen Schritt in seinen Registern $AREG_{i,j}$ und $BREG_{i,j}$ über diejenigen Matrixelemente verfügt, die er benötigt (Abbildung 8.1.2-2 auf der rechten Seite). Dazu wird für $l = 1, ...n$ die l-te Zeile von **A** um $l - 1$ Positionen zyklisch nach links und für $k = 1, ...n$ die k-te Spalte von **B** um $k - 1$ Positionen zyklisch nach oben verschoben.

Registerbelegung von $P_{i,j}$ in der Reihenfolge
$AREG_{i,j}$
$BREG_{i,j}$
$CREG_{i,j}$

Initialisierung:

$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$b_{1,1}$	$b_{1,2}$	$b_{1,3}$	$b_{1,1}$	$b_{2,2}$	$b_{3,3}$
0	0	0	0	0	0
$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,2}$	$a_{2,3}$	$a_{2,1}$
$b_{2,1}$	$b_{2,2}$	$b_{2,3}$	$b_{2,1}$	$b_{3,2}$	$b_{1,3}$
0	0	0	0	0	0
$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,3}$	$a_{3,1}$	$a_{3,2}$
$b_{3,1}$	$b_{3,2}$	$b_{3,3}$	$b_{3,1}$	$b_{1,2}$	$b_{2,3}$
0	0	0	0	0	0

Abbildung 8.1.2-2: Prozessorinitialisierung zur Matrixmultiplikation (Beispiel)

Alle Prozessoren arbeiten getaktet parallel. Prozessor $P_{i,j}$ mit $i = 1, ..., n$ und $j = 1, ..., n$ durchläuft dabei folgendes Programmstück (in Pseudocode-Notation):

```
{ Deklaration lokaler Variablen in P_{i,j} :}
VAR links  : INTEGER;
    rechts : INTEGER;
    oben   : INTEGER;
    unten  : INTEGER;
    l      : INTEGER;
    k      : INTEGER;

{ Initialisierung: }
CREG_{i,j}  := 0;
AREG_{i,j}  := a_{i,j};
BREG_{i,j}  := b_{i,j};

{ Verschiebung der l-ten Zeile von A um l - 1 Positionen
  nach links und
  Verschiebung der k-ten Spalte von B um k - 1 Positionen
  nach oben:                                              }
IF j = 1 THEN links := n ELSE links := j - 1;
IF j = n THEN rechts := 1 ELSE rechts := j + 1;
IF i = 1 THEN oben := n ELSE oben := i - 1;
IF i = n THEN unten := 1 ELSE unten := i + 1;
```

```
FOR l := 1 TO i - 1 DO
   AREG_{i,j} := AREG_{i,rechts};
FOR k := 1 TO j - 1 DO
   BREG_{i,j} := BREG_{unten,i};

{ Multiplikation der Matrixelemente, Kumulierung in CREG_{i,j}
  und (für den nächsten Schritt) Verschiebung der Zeilen von A
  um eine Position nach rechts und der Spalten von B
  um eine Position nach unten:                                 }

FOR k := 1 TO n DO
  BEGIN
    CREG_{i,j} := CREG_{i,j} + AREG_{i,j} * BREG_{i,j};
    AREG_{i,j} := AREG_{i,links};
    BREG_{i,j} := BREG_{oben,j};
   END;
```

Der Ablauf für $n = 3$ ist in Abbildung 8.1.2-3 dargestellt.

Initialisierung:

$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$b_{1,1}$	$b_{1,2}$	$b_{1,3}$	$b_{1,1}$	$b_{2,2}$	$b_{3,3}$
0	0	0	0	0	0
$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,2}$	$a_{2,3}$	$a_{2,1}$
$b_{2,1}$	$b_{2,2}$	$b_{2,3}$	$b_{2,1}$	$b_{3,2}$	$b_{1,3}$
0	0	0	0	0	0
$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,3}$	$a_{3,1}$	$a_{3,2}$
$b_{3,1}$	$b_{3,2}$	$b_{3,3}$	$b_{3,1}$	$b_{1,2}$	$b_{2,3}$
0	0	0	0	0	0

Registerbelegung am Ende des ersten parallelen Schritts (k = 1):

$a_{1,3}$	$a_{1,1}$	$a_{1,2}$
$b_{3,1}$	$b_{1,2}$	$b_{2,3}$
$a_{1,1} \cdot b_{1,1}$	$a_{1,2} \cdot b_{2,2}$	$a_{1,3} \cdot b_{3,3}$
$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$b_{1,1}$	$b_{2,2}$	$b_{3,3}$
$a_{2,2} \cdot b_{2,1}$	$a_{2,3} \cdot b_{3,2}$	$a_{2,1} \cdot b_{1,3}$
$a_{3,2}$	$a_{3,3}$	$a_{3,1}$
$b_{2,1}$	$b_{3,2}$	$b_{1,3}$
$a_{3,3} \cdot b_{3,1}$	$a_{3,1} \cdot b_{1,2}$	$a_{3,2} \cdot b_{2,3}$

Registerbelegung am Ende des zweiten parallelen Schritts (k = 2):

$a_{1,2}$	$a_{1,3}$	$a_{1,1}$
$b_{2,1}$	$b_{3,2}$	$b_{1,3}$
$a_{1,1} \cdot b_{1,1} + a_{1,3} \cdot b_{3,1}$	$a_{1,2} \cdot b_{2,2} + a_{1,1} \cdot b_{1,2}$	$a_{1,3} \cdot b_{3,3} + a_{1,2} \cdot b_{2,3}$
$a_{2,3}$	$a_{2,1}$	$a_{2,2}$
$b_{3,1}$	$b_{1,2}$	$b_{2,3}$
$a_{2,2} \cdot b_{2,1} + a_{2,1} \cdot b_{1,1}$	$a_{2,3} \cdot b_{3,2} + a_{2,2} \cdot b_{2,2}$	$a_{2,1} \cdot b_{1,3} + a_{2,3} \cdot b_{3,3}$
$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
$b_{1,1}$	$b_{2,2}$	$b_{3,3}$
$a_{3,3} \cdot b_{3,1} + a_{3,2} \cdot b_{2,1}$	$a_{3,1} \cdot b_{1,2} + a_{3,3} \cdot b_{3,2}$	$a_{3,2} \cdot b_{2,3} + a_{3,1} \cdot b_{1,3}$

Registerbelegung am Ende des dritten parallelen Schritts (k = 3):

$a_{1,2}$	$a_{1,3}$	$a_{1,1}$
$b_{2,1}$	$b_{3,2}$	$b_{1,3}$
$a_{1,1} \cdot b_{1,1} + a_{1,3} \cdot b_{3,1} + a_{1,2} \cdot b_{2,1}$	$a_{1,2} \cdot b_{2,2} + a_{1,1} \cdot b_{1,2} + a_{1,3} \cdot b_{3,2}$	$a_{1,3} \cdot b_{3,3} + a_{1,2} \cdot b_{2,3} + a_{1,1} \cdot b_{1,3}$
$a_{2,3}$	$a_{2,1}$	$a_{2,2}$
$b_{3,1}$	$b_{1,2}$	$b_{2,3}$
$a_{2,2} \cdot b_{2,1} + a_{2,1} \cdot b_{1,1} + a_{2,3} \cdot b_{3,1}$	$a_{2,3} \cdot b_{3,2} + a_{2,2} \cdot b_{2,2} + a_{2,1} \cdot b_{1,2}$	$a_{2,1} \cdot b_{1,3} + a_{2,3} \cdot b_{3,3} + a_{2,2} \cdot b_{2,3}$
$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
$b_{1,1}$	$b_{2,2}$	$b_{3,3}$
$a_{3,3} \cdot b_{3,1} + a_{3,2} \cdot b_{2,1} + a_{3,1} \cdot b_{1,1}$	$a_{3,1} \cdot b_{1,2} + a_{3,3} \cdot b_{3,2} + a_{3,2} \cdot b_{2,2}$	$a_{3,2} \cdot b_{2,3} + a_{3,1} \cdot b_{1,3} + a_{3,3} \cdot b_{3,3}$

Abbildung 8.1.2-3: Parallele Matrixmultiplikation (Fortsetzung des Beispiels aus Abbildung 8.1.2-2)

Die Anzahl durchgeführter paralleler Schritte ist offensichtlich von der Ordnung $O(n)$. Da n^2 viele Prozessoren eingesetzt werden, sind die parallelen Kosten von der Ordnung $O(n^3)$.

Es gibt eine Reihe weiterer Verfahren zur parallelen Matrixoperation, die weniger parallele Schritte ausführen, aber dabei mehr Prozessoren einsetzen. Ein Beispiel ist in folgendem Ansatz für das CREW-PRAM-Modell beschrieben.

Es werden n^3 viele Prozessoren $P_1, ..., P_{n^3}$ eingesetzt, die durch drei Indizes i, j und k mit $i = 1, ..., n$, $j = 1, ..., n$ und $k = 1, ..., n$ indiziert werden. Jeder Prozessor $P_{i,j,k}$ verfügt wieder über drei Register $AREG_{i,j,k}$, $BREG_{i,j,k}$ und $CREG_{i,j,k}$ (die Indizes an den Registernamen sind wieder nur zur Verdeutlichung der Zugehörigkeit zu einem Prozessor angebracht) und ist mit anderen Prozessoren verknüpft, so dass verbundene Prozessoren Werte dieser Register austauschen können. Die Vernetzungstopologie wird weiter unten beschrieben. Zusätzlich verfügt er über weitere lokale Variablen, die anderen Prozessoren nicht zugänglich gemacht werden.

Die Prozessoren mit einem festen Index k bilden die k-te (Prozessor-) Schicht (oberer Teil von Abbildung 8.1.2-4).

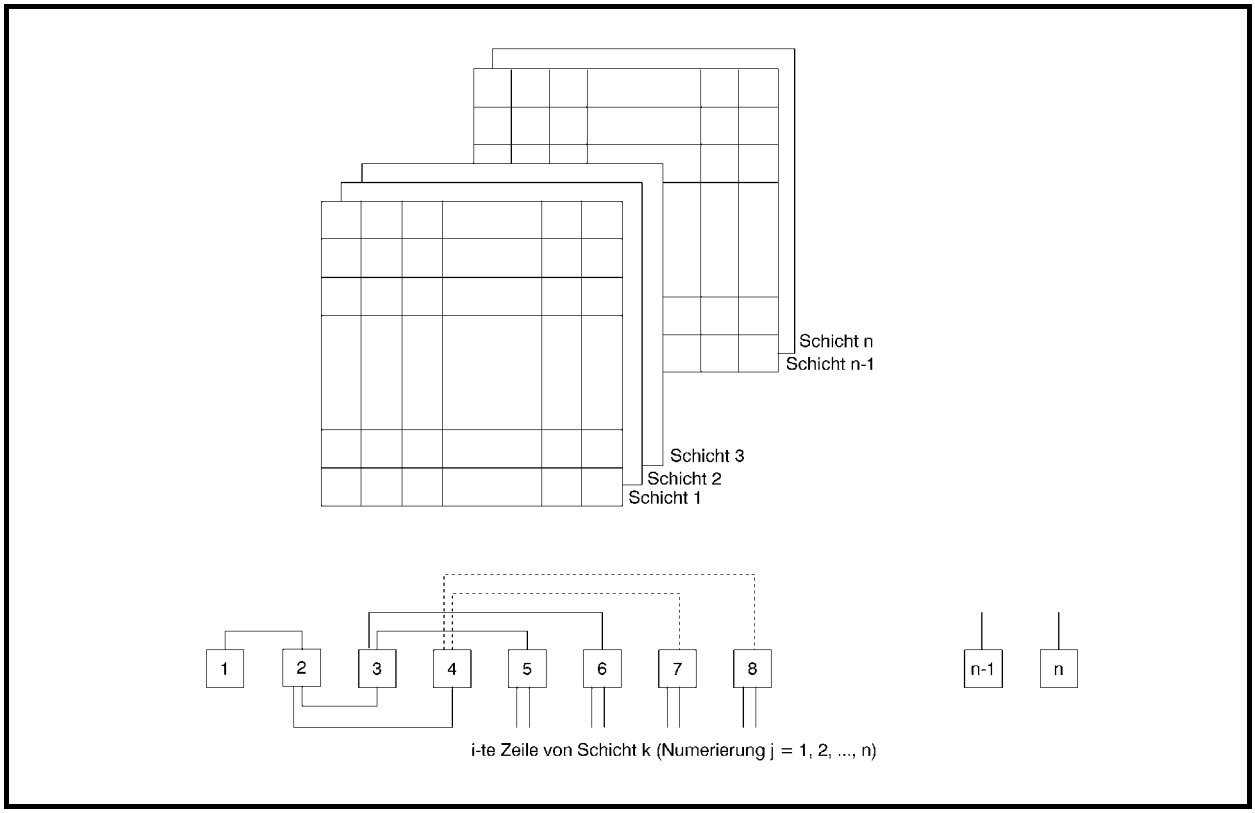


Abbildung 8.1.2-4: Schichtenmodell zur schnellen parallelen Matrixmultiplikation

Die i-te „Zeile“ in der k-ten Schicht, das sind die Prozessoren $P_{i,1,k}, ..., P_{i,n,k}$, sind miteinander in folgender Weise verknüpft (unterer Teil von Abbildung 8.1.2-4):

$P_{i,1,k}$ ist verbunden mit $P_{i,2,k}$,
$P_{i,j,k}$ ist verbunden mit $P_{i,2j-1,k}$ und $P_{i,2j,k}$ für $j = 2, ..., \lfloor n/2 \rfloor$,
für ungerades n ist $P_{i,(n+1)/2,k}$ verbunden mit $P_{i,n,k}$.

In parallelen Initialisierungsschritten wird in die Prozessoren der k-ten Schicht für $k = 1, ..., n$ die Matrix **A** und in jede „Zeile" der k-ten Schicht die k-te Spalte der Matrix **B** geladen. Anschließend werden parallele Schritte zur Bildung der einzelnen Summen ausgeführt.

Auf diese Weise ermittelt die k-te Schicht alle Elemente der Matrix **C**, die von der k-ten Spalte in **B** abhängen, d.h. die k-te Spalte in **C**:

$$\begin{bmatrix} c_{1,k} \\ . \\ . \\ . \\ c_{n,k} \end{bmatrix} = \mathbf{A} \cdot \begin{bmatrix} b_{1,k} \\ . \\ . \\ . \\ b_{n,k} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} a_{1,j} \cdot b_{j,k} \\ . \\ . \\ . \\ \sum_{j=1}^{n} a_{n,j} \cdot b_{j,k} \end{bmatrix}$$

Der Prozessor $P_{i,j,k}$ für $i = 1, ..., n$, $j = 1, ..., n$ und $k = 1, ..., n$ durchläuft den Pseudocode

```
{ Deklaration lokaler Variablen in P_{i,j,k} :}
VAR idx      : INTEGER;
    s        : INTEGER;
    t        : INTEGER;
    ungerade : BOOLEAN;

{ Initialisierung: }
t := ⌈log_2(n)⌉;
s := n;
IF (s mod 2) = 0
THEN BEGIN
       s        := s SHR 1;
       ungerade := FALSE;
     END
ELSE BEGIN
       s        := (s SHR 1) + 1;
       ungerade := TRUE;
     END;

AREG_{i,j,k} := a_{i,j};
```

```
BREG_{i,j,k}  :=  b_{j,k} ;
CREG_{i,j,k}  :=  AREG_{i,j,k}  *  BREG_{i,j,k} ;

{ Kumulation der Matrixelement-Produkte: }
FOR idx := 1 TO t DO
  BEGIN
     IF j <= s
     THEN BEGIN
            AREG_{i,j,k}  :=  CREG_{i,2j-1,k} ;
            IF (j = s) AND ungerade
            THEN BREG_{i,j,k}  := 0
            ELSE BREG_{i,j,k}  :=  CREG_{i,2j,k} ;
            CREG_{i,j,k}  :=  AREG_{i,j,k}  +  BREG_{i,j,k} ;
            IF (s mod 2) = 0
            THEN BEGIN
                   s        := s SHR 1;
                   ungerade := FALSE;
                 END
            ELSE BEGIN
                   s        := (s SHR 1) + 1;
                   ungerade := TRUE;
                 END;
          END;
  END;
```

Die Werte der lokalen Variablen und Registerbelegungen während des Ablaufs in den Prozessoren $P_{i,j,k}$ der i-ten Zeile in der k-ten Schicht mit $j = 1, ..., n$ ist für $n = 14$ in Abbildung 8.1.2-5 dargestellt.

Registerbelegung von $P_{i,j,k}$ in der Reihenfolge
$AREG_{i,j,k}$
$BREG_{i,j,k}$
$CREG_{i,j,k}$
In allen Prozessoren ist `t` = 4, die Variable `ungerade` ist durch `u` abgekürzt, `T` steht für `TRUE`, `F` steht für `FALSE`.

Initialisierung:

$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	$a_{i,4}$	$a_{i,5}$	$a_{i,6}$	$a_{i,7}$	$a_{i,8}$	$a_{i,9}$	$a_{i,10}$	$a_{i,11}$	$a_{i,12}$	$a_{i,13}$	$a_{i,14}$
$b_{1,k}$	$b_{2,k}$	$b_{3,k}$	$b_{4,k}$	$b_{5,k}$	$b_{6,k}$	$b_{7,k}$	$b_{8,k}$	$b_{9,k}$	$b_{10,k}$	$b_{11,k}$	$b_{12,k}$	$b_{13,k}$	$b_{14,k}$
$a_{i,1} \cdot b_{1,k}$	$a_{i,2} \cdot b_{2,k}$	$a_{i,3} \cdot b_{3,k}$	$a_{i,4} \cdot b_{4,k}$	$a_{i,5} \cdot b_{5,k}$	$a_{i,6} \cdot b_{6,k}$	$a_{i,7} \cdot b_{7,k}$	$a_{i,8} \cdot b_{8,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,14} \cdot b_{14,k}$
s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7	s = 7
u = F	u = F	u = F	u = F	u = F	u = F	u = F	u = F	u = F	u = F	u = F	u = F	u = F	u = F

Werte am Ende des ersten Schleifendurchlaufs (`idx` = 1):

$a_{i,1} \cdot b_{1,k}$	$a_{i,3} \cdot b_{3,k}$	$a_{i,5} \cdot b_{5,k}$	$a_{i,7} \cdot b_{7,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,8}$	$a_{i,9}$	$a_{i,10}$	$a_{i,11}$	$a_{i,12}$	$a_{i,13}$	$a_{i,14}$
$a_{i,2} \cdot b_{2,k}$	$a_{i,4} \cdot b_{4,k}$	$a_{i,6} \cdot b_{6,k}$	$a_{i,8} \cdot b_{8,k}$	$a_{i,10} \cdot b_{10,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,14} \cdot b_{14,k}$	$b_{8,k}$	$b_{9,k}$	$b_{10,k}$	$b_{11,k}$	$b_{12,k}$	$b_{13,k}$	$b_{14,k}$
$a_{i,1} \cdot b_{1,k}$ $+ a_{i,2} \cdot {}_{2,k}$	$a_{i,3} \cdot b_{3,k}$ $+a_{i,4} \cdot b_{4,k}$	$a_{i,5} \cdot b_{5,k}$ $+a_{i,6} \cdot b_{6,k}$	$a_{i,7} \cdot b_{7,k}$ $+a_{i,8} \cdot b_{8,k}$	$a_{i,9} \cdot b_{9,k}$ $+$ $a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$ $+$ $a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,8} \cdot b_{8,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,14} \cdot b_{14,k}$
s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F

Werte am Ende des zweiten Schleifendurchlaufs (`idx` = 2):

$a_{i,1} \cdot b_{1,k}$ $+ a_{i,2} \cdot {}_{2,k}$	$a_{i,5} \cdot b_{5,k}$ $+a_{i,6} \cdot b_{6,k}$	$a_{i,9} \cdot b_{9,k}$ $+$ $a_{i,10} \cdot b_{10,k}$	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,8}$	$a_{i,9}$	$a_{i,10}$	$a_{i,11}$	$a_{i,12}$	$a_{i,13}$	$a_{i,14}$
$a_{i,3} \cdot b_{3,k}$ $+a_{i,4} \cdot b_{4,k}$	$a_{i,7} \cdot b_{7,k}$ $+a_{i,8} \cdot b_{8,k}$	$a_{i,11} \cdot b_{11,k}$ $+$ $a_{i,12} \cdot b_{12,k}$	0	$a_{i,10} \cdot b_{10,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,14} \cdot b_{14,k}$	$b_{8,k}$	$b_{9,k}$	$b_{10,k}$	$b_{11,k}$	$b_{12,k}$	$b_{13,k}$	$b_{14,k}$
$a_{i,1} \cdot b_{1,k}$ $+ a_{i,2} \cdot {}_{2,k}$ $+a_{i,3} \cdot b_{3,k}$ $+a_{i,4} \cdot b_{4,k}$	${}_{5} \cdot b_{5,k}$ $+a_{i,6} \cdot b_{6,k}$ $+a_{i,7} \cdot b_{7,k}$ $+a_{i,8} \cdot b_{8,k}$	$a_{i,9} \cdot b_{9,k}$ $+$ $a_{i,10} \cdot b_{10,k}$ $+$ $a_{i,11} \cdot b_{11,k}$ $+$ $a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,9} \cdot b_{9,k}$ $+$ $a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$ $+$ $a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,8} \cdot b_{8,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,14} \cdot b_{14,k}$
s = 2 u = F	s = 2 u = F	s = 2 u = F	s = 2 u = F	s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F

Werte am Ende des dritten Schleifendurchlaufs (`idx` = 3):

$a_{i,j} \cdot b_{j,k}$ sum-miert für j = 1,...,4	$a_{i,j} \cdot b_{j,k}$ sum-miert für j = 9,...,12	$a_{i,9} \cdot b_{9,k}$ $+$ $a_{i,10} \cdot b_{10,k}$	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,8}$	$a_{i,9}$	$a_{i,10}$	$a_{i,11}$	$a_{i,12}$	$a_{i,13}$	$a_{i,14}$
$a_{i,j} \cdot b_{j,k}$ sum-miert für j = 5,...,8	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,11} \cdot b_{11,k}$ $+$ $a_{i,12} \cdot b_{12,k}$	0	$a_{i,10} \cdot b_{10,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,14} \cdot b_{14,k}$	$b_{8,k}$	$b_{9,k}$	$b_{10,k}$	$b_{11,k}$	$b_{12,k}$	$b_{13,k}$	$b_{14,k}$
$a_{i,j} \cdot b_{j,k}$ sum-miert für j = 1,...,8	$a_{i,j} \cdot b_{j,k}$ sum-miert für j = 9,...,14	$a_{i,9} \cdot b_{9,k}$ $+$ $a_{i,10} \cdot b_{10,k}$ $+$ $a_{i,11} \cdot b_{11,k}$ $+$ $a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,9} \cdot b_{9,k}$ $+$ $a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$ $+$ $a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$ $+$ $a_{i,14} \cdot b_{14,k}$	$a_{i,8} \cdot b_{8,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,14} \cdot b_{14,k}$
s = 1 u = F	s = 1 u = F	s = 2 u = F	s = 2 u = F	s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F

Werte am Ende des vierten Schleifendurchlaufs (`idx` = 4):													
$a_{i,j} \cdot b_{j,k}$ summiert für j = 1,...,8	$a_{i,j} \cdot b_{j,k}$ summiert für j = 9,...,12	$a_{i,9} \cdot b_{9,k}$ + $a_{i,10} \cdot b_{10,k}$	$a_{i,13} \cdot b_{13,k}$ + $a_{i,14} \cdot b_{14,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,8}$	$a_{i,9}$	$a_{i,10}$	$a_{i,11}$	$a_{i,12}$	$a_{i,13}$	$a_{i,14}$
$a_{i,j} \cdot b_{j,k}$ summiert für j = 9,...,14	$a_{i,13} \cdot b_{13,k}$ + $a_{i,14} \cdot b_{14,k}$	$a_{i,11} \cdot b_{11,k}$ + $a_{i,12} \cdot b_{12,k}$	0	$a_{i,10} \cdot b_{10,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,14} \cdot b_{14,k}$	$b_{8,k}$	$b_{9,k}$	$b_{10,k}$	$b_{11,k}$	$b_{12,k}$	$b_{13,k}$	$b_{14,k}$
$a_{i,j} \cdot b_{j,k}$ summiert für j = 1,...,14	$a_{i,j} \cdot b_{j,k}$ summiert für j = 9,...,14	$a_{i,9} \cdot b_{9,k}$ + $a_{i,10} \cdot b_{10,k}$ + $a_{i,11} \cdot b_{11,k}$ + $a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$ + $a_{i,14} \cdot b_{14,k}$	$a_{i,9} \cdot b_{9,k}$ + $a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$ + $a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$ + $a_{i,14} \cdot b_{14,k}$	$a_{i,8} \cdot b_{8,k}$	$a_{i,9} \cdot b_{9,k}$	$a_{i,10} \cdot b_{10,k}$	$a_{i,11} \cdot b_{11,k}$	$a_{i,12} \cdot b_{12,k}$	$a_{i,13} \cdot b_{13,k}$	$a_{i,14} \cdot b_{14,k}$
s = 0 u = T	s = 1 u = F	s = 2 u = F	s = 2 u = F	s = 4 u = T	s = 4 u = T	s = 4 u = T	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F	s = 7 u = F

Abbildung 8.1.2-5: Schnelle parallele Matrixmultiplikation (Beispiel)

Die Werte der Produktmatrix finden sich am Ende des Ablaufs in Prozessor $P_{i,1,k}$, und zwar enthält Register $CREG_{i,1,k}$ den Wert $\sum_{j=1}^{n} a_{i,j} \cdot b_{j,k}$. Die Anzahl durchgeführter paralleler Schritte ist offensichtlich von der Ordnung $O(\log(n))$, also gegenüber dem sequentiellen Verfahren deutlich besser, jedoch haben die parallelen Kosten wegen der n^3 vielen eingesetzten Prozessoren die Größenordnung $O\left(n^3 \cdot \log(n)\right)$.

Zu beobachten ist, dass nach der Initialisierungsphase, in der alle n^3 Prozessoren aktiv sind, in den nachfolgenden parallelen Schritten ein Prozessor $P_{i,j,k}$ der *i*-ten Zeile der Schicht *k* mit $j > \mathtt{s}$ wegen der Anweisung

```
IF j <= s
     THEN BEGIN
              ...
          END;
```

lediglich feststellt, dass er keine arithmetischen Operationen durchzuführen und auch keine Registerinhalte anderen Prozessoren zur Verfügung zu stellen hat. Sobald also ein Prozessor feststellt, dass er im gegenwärtigen parallelen Schritt und den folgenden parallelen Schritten keine Zwischenergebnisse zur Berechnung beiträgt, kann für ihn die Schleife

```
FOR idx := 1 TO t DO
  BEGIN
    ...
  END;
```

abgebrochen werden, und er kann als inaktiv betrachtet werden. Dazu wird der obige Pseudocode folgendermaßen modifiziert:

```
FOR idx := 1 TO t DO
  BEGIN
    IF j <= s
     THEN BEGIN
            ...
          END
     ELSE Break;
  END;
```

Im ersten parallelen Schritt nach der Initialisierungsphase sind in Zeile i der Schicht k noch alle n Prozessoren aktiv (von denen etwa die Hälfte lediglich in die `Break`-Anweisung läuft), im zweiten parallelen Schritt nur noch $\lceil n/2 \rceil$ viele, im dritten nur noch $\lceil n/4 \rceil$ viele usw. Im l-ten parallelen Schritt sind noch $\lceil n/2^{l-1} \rceil$ viele Prozessoren aktiv. Die effektiven Kosten des Verfahrens, d.h. die Summe aller durchgeführten Schritte der *aktiven* Prozessoren, lassen sich für die Initialisierungsphase durch eine Größe der Ordnung $O(n^3)$ abschätzen (alle Prozessoren führen jeweils konstant viele Operationen aus). Für die i-te Zeile von Schicht k ergibt sich ein Anteil an den effektiven Kosten der Form

$$c \cdot \sum_{l=1}^{\lceil \log_2(n) \rceil} \lceil n/2^{l-1} \rceil = c \cdot \sum_{l=0}^{\lceil \log_2(n) \rceil - 1} \lceil n/2^{l} \rceil \leq c \cdot \sum_{l=0}^{\lceil \log_2(n) \rceil - 1} \left(n/2^{l} + 1 \right) \leq c \cdot \left(\log_2(n) + 2n + 1 \right)$$

mit einer Konstanten $c > 0$. Die Anzahl der Zeilen in Schicht k ist n, und es gibt n viele Schichten, so dass die effektiven Kosten durch einen Wert der Ordnung $O\left(n^3 + n^2 \cdot (\log(n) + n)\right)$, also durch einen Wert der Ordnung $O(n^3)$ abschätzen lassen.

Die Aufgabe bestehe nun in der Berechnung des Produkts

$\vec{y}_{(n,1)} := \mathbf{A}_{(n,m)} \cdot \vec{x}_{(m,1)}$

mit einer Matrix $\mathbf{A}_{(n,m)} = \lfloor a_{i,j} \rfloor$ mit einem Spaltenvektor $\vec{x}_{(m,1)}$. Die Komponenten von $\vec{x}_{(m,1)}$ seien x_j für $j = 1, .., m$. Dann berechnen sich die Komponenten y_i für $i = 1, ..., n$ des Vektors $\vec{y}_{(n,1)}$ zu

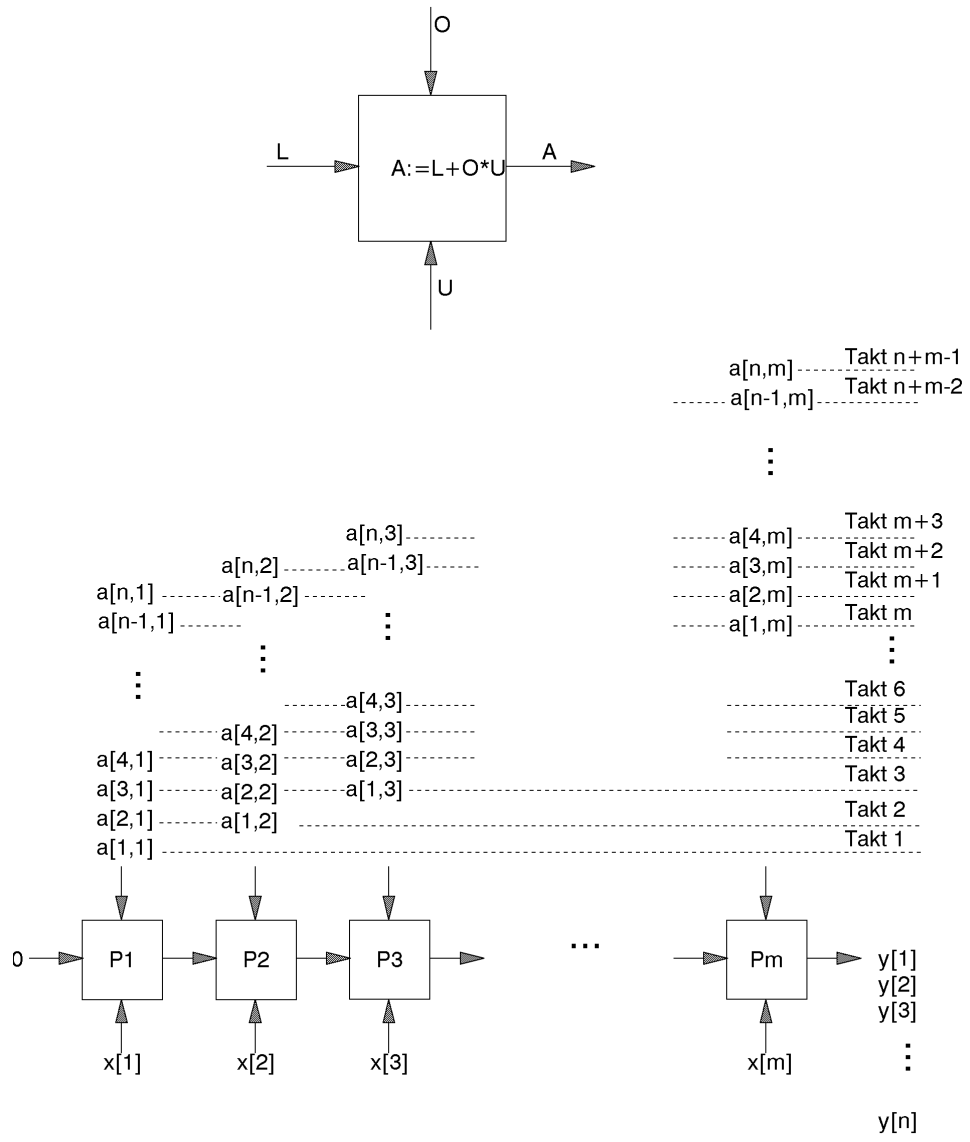
$y_i = \sum_{j=1}^{m} a_{i,j} \cdot x_j$ für $i = 1, ..., n$.

Bei serieller Ausführung sind also $O(nm)$ viele arithmetische Operationen notwendig.

Eine parallele Architektur setzt die im oberen Teil der Abbildung 8.1.2-6 gezeigten Prozessorelemente ein. Jedes Prozessorelement besitzt drei mit L, O bzw. U bezeichnete Eingänge und einen mit A bezeichneten Ausgang. Die Prozessorelemente sind zusammengeschaltet - das zeigt der untere Teil der Abbildung - und können über die gezeigten Pfeile Daten an ihre Nachbarn weitergeben bzw. von dort empfangen. Außerdem können Daten von außen in ein Prozessorelement eingegeben werden; die entsprechenden Eingänge sind durch einen auf das Prozessorelement weisenden Pfeil ohne Anfangsknoten gekennzeichnet. Ein Prozessorelement gibt Daten nach außen (über einen nicht mit einem weiteren Prozessorelement verbundenen Pfeil, in der Abbildung nur das Prozessorelement am rechten Rand). Alle Prozessorelemente arbeiten über einen globalen Takt synchronisiert. Ein Prozessorelement kann nur eine sehr einfache Operationen ausführen, nämlich es berechnet in einem Takt

```
a := l + (o * u)
```

wobei `l`, `o` bzw. `u` die auf den Eingängen L, O bzw. U zu Beginn des Takts liegenden Werte sind, und gibt den Wert `a` auf den Ausgang A, so dass dieser Wert im nächsten Takt von dem mit A verbundenen Nachbarn gelesen werden kann bzw. aus dem Netz ausgegeben wird. Ein Prozessorelement wird in jedem Takt aktiv.



Zur besseren Lesbarkeit werden hier anstelle tiefgestellter Indizes in eckige Klammern gesetzte Indizes verwendet.

Abbildung 8.1.2-1: Multiplikation einer Matrix mit einem Vektor

Es werden m Prozessorelemente P_1, ..., P_m eingesetzt. Die Komponenten des Vektors $\vec{x}_{(m,1)}$ werden jeweils auf die Eingänge U gelegt und bleiben dort in jedem Takt erhalten (alternativ könnten die Vektorkomponenten im ersten Schritt in das jeweilige Prozessorelement eingelesen und dort lokal gespeichert werden). Das Prozessorelement P_1 erhält in jedem Takt den Wert 0 auf seinem Eingang L. Die Komponenten der Matrix $\mathbf{A}_{(n,m)}$ werden nacheinander zeitlich versetzt folgendermaßen über die Eingänge O in die Prozessorelemente eingegeben. Im ersten Takt wird $a_{1,1}$ in P_1 eingegeben; alle anderen Prozessorelemente erhalten über den Ein-

gang O den Wert 0. Im zweiten Takt liegen $a_{2,1}$ an P_1 und $a_{1,2}$ an P_2 an, und alle übrigen Prozessorelemente erhalten über den Eingang O wieder den Wert 0. Im dritten Takt liegen $a_{3,1}$ an P_1, $a_{2,2}$ an P_2 und $a_{1,3}$ an P_3; alle anderen Prozessorelement empfangen wieder den Wert 0 usw. Allgemein erhält das Prozessorelement P_j am Eingang O im Takt t (t = 1, ..., n+m) den Wert

$$\begin{cases} a_{t-j+1,j} & \text{für } 1 \le t-j+1 \le n \\ 0 & \text{sonst} \end{cases}$$

Am Ende der Takte m, m + 1, ..., m + n – 1 liefert der Ausgang A des Prozessorelements P_m nacheinander die Werte y_1, y_2, ..., y_n. Offensichtlich werden bei diesem Prinzip $O(n+m)$ viele parallele Schritte mit m Prozessoren ausgeführt, so dass die parallelen Kosten von der Größenordnung $O(nm+m^2) = O(n^2)$ (bei $m \le n$) und damit kostenoptimal sind.

8.2 Stochastische Verfahren

Die bisher beschriebenen Lösungsverfahren waren rein deterministisch. Eine Ausnahme bildet das Sortieren gemäß Quicksort, der das Teilungselement jeweils zufällig auswählt. Dieser Ansatz, nämlich Zufallsexperimente in das Problemlösungsverfahren einzubauen, hat sich häufig bewährt und in der Praxis zu sehr effizienten Problemlösungsverfahren geführt. Die Theoretische Informatik hat hierzu (wie lange bereits zu den deterministischen Verfahren) das theoretische Fundament geliefert ([A/.]).

Im ansonsten deterministischen Algorithmus werden als zulässige Elementaroperationen Zufallsexperimente zugelassen. Ein derartiges Zufallsexperiment kann beispielsweise mit Hilfe eines Zufallszahlengenerators ausgeführt werden. So wird beispielsweise auf diese Weise entschieden, in welchem Teil einer Programmverzweigung der Algorithmus während seines Ablaufs fortgesetzt wird. Andere Möglichkeiten zum Einsatz eines Zufallsexperiments bestehen bei Entscheidungen zur Auswahl möglicher Elemente, die im weiteren Ablauf des Algorithmus als nächstes untersucht werden sollen.

Man nennt derartige Algorithmen **randomisierte Algorithmen**. Eine Einführung in Methoden zum Entwurf derartiger Verfahren findet man beispielsweise in [HRO].

Grundsätzlich gibt es zwei Klassen randomisierter Algorithmen: **Las-Vegas-Verfahren**, die stets – wie von deterministischen Algorithmen gewohnt – ein korrektes Ergebnis berechnen. Daneben gibt es **Monte-Carlo-Verfahren**[14], die ein korrektes Ergebnis nur mit einer gewis-

[14] Die Typbezeichnung „Monte Carlo“ steht für *mostly correct*.

sen Fehlerwahrscheinlichkeit, aber in jedem Fall effizient, bestimmen. Häufig findet man bei randomisierten Algorithmen einen Trade-off zwischen Korrektheit und Effizienz.

Beispiele für randomisierte Algorithmen vom Las-Vegas-Typ sind:

- Einfügen von Primärschlüsselwerten in einen binären Suchbaum: Statt die Schlüssel $S_1, ..., S_n$ in dieser Reihenfolge sequentiell in den binären Suchbaum einzufügen, wird jeweils der nächste einzufügende Schlüssel aus den restlichen, d.h. noch nicht eingefügten Schlüsseln zufällig ausgewählt und in den binären Suchbaum eingefügt. Das Ergebnis ist eine mittlere Baumhöhe der Ordnung $O(\log(n))$
- `Quicksort` zum Sortieren Elementen, auf denen eine lineare Ordnung definiert ist.

Größere praktische Bedeutung haben randomisierte Algorithmen vom Monte-Carlo-Typ. Im folgenden Unterkapitel wird exemplarisch die Primzahlsuche nach dem Monte-Carlo-Prinzip beschrieben. Auch wenn man heute über (theoretisch) effiziente deterministische Algorithmen zur Primzahlsuche verfügt, hat sich das beschriebeneVerfahren als äußerst praktikabel erwiesen.

8.2.1 Primzahlsuche

Beispiele für randomisierte Algorithmen vom Monte-Carlo-Typ sind die heute üblichen Primzahltests, von denen ein Verfahren hier informell beschrieben werden soll (Details findet man in der angegebenen Literatur, z.B. in [YAN]).

Einer der wichtigsten Sätze der Zahlentheorie sagt über die Anzahl der Primzahlen unterhalb einer vorgegebenen Grenze x aus:

(i) Es gilt $\lim_{x\to\infty} \frac{\pi(x)\cdot\ln(x)}{x} = 1$, d.h. $\pi(x) \sim \frac{x}{\ln(x)}$ (für große x).

(ii) Für $x \geq 67$ ist $\ln(x) - \frac{3}{2} < \frac{x}{\pi(x)} < \ln(x) - \frac{1}{2}$.

(iii) Für $n \geq 20$ ist $n\cdot\left(\ln(n) + \ln(\ln(n)) - \frac{3}{2}\right) < p_n < n\cdot\left(\ln(n) + \ln(\ln(n)) - \frac{1}{2}\right)$.

Auf der Grundlage dieses Satzes lässt sich ein effizientes Verfahren zur Erzeugung von (großen) Zahlen angeben, die mit beliebig großer Wahrscheinlichkeit Primzahlen sind. Dabei wird in Kauf genommen, dass das Verfahren eine Zahl eventuell als Primzahl einstuft, die keine

Primzahl ist. Die Fehlerwahrscheinlichkeit dieser falschen Entscheidung kann jedoch auf einfache Weise beliebig klein gehalten werden.

Um eine Primzahl mit *m* Dezimalstellen zu erzeugen, kann man etwa folgendes Verfahren (Pseudocode) verwenden. Zur Darstellung ganzer Zahlen mit betragsmäßig beliebig großer Genauigkeit werde hier wieder der Datentyp `INTEGER` genommen. Das Verfahren verwendet die beiden Funktionen `is_prime` und `zufallszahl`:

Die Funktion

```
FUNCTION is_prime (p : INTEGER) : BOOLEAN;
```

liefert den Wert `TRUE`, wenn *p* eine Primzahl ist, ansonsten den Wert `FALSE`. Zur Implementierung dieser Funktion werden weiter unten Überlegungen angestellt.

Die Funktion

```
FUNCTION zufallszahl (m : INTEGER) : INTEGER;
```

liefert eine ungerade zufällig ausgewählte Zahl mit *m* Dezimalstellen. Die Implementierung kann beispielsweise so erfolgen, dass man nacheinander *m* Ziffern aus $\{0,1,2,...,9\}$ zufällig zieht und diese Folge zu einer Dezimalzahl zusammensetzt. Dabei darf die führende Ziffer nicht 0 und die niedrigstwertige Ziffer nicht gerade sein.

```
VAR p : INTEGER;
    m : INTEGER;

m := ...;
p := 1;
WHILE NOT is_prime(p) DO p := zufallszahl(m);
```

Für $m > 1$ gibt es $\frac{1}{2} \cdot \left(10^m - 10^{m-1}\right)$ viele ungerade Zahlen mit *m* Dezimalstellen. Die Anzahl der *m*-stelligen Primzahlen ist $\pi\left(10^m - 1\right) - \pi\left(10^{m-1} - 1\right)$. Die Wahrscheinlichkeit, dass eine erzeugte ungerade *m*-stellige Zufallszahl eine Primzahl ist, beträgt daher

$$W(m) = \frac{2 \cdot \left(\pi\left(10^m - 1\right) - \pi\left(10^{m-1} - 1\right)\right)}{10^m - 10^{m-1}} .$$

Es sei $m > 2$, so dass im obigen Satz Teil (ii) angewendet werden kann. Für $x \geq 67$ ist $\frac{x}{\pi(x)} = \ln(x) - \varepsilon$ mit $\frac{1}{2} < \varepsilon < \frac{3}{2}$. Daher ist

$$\pi\left(10^m - 1\right) - \pi\left(10^{m-1} - 1\right) = \pi\left(10^m\right) - \pi\left(10^{m-1}\right) = \frac{10^m}{m \cdot \ln(10) - \varepsilon_1} - \frac{10^{m-1}}{(m-1) \cdot \ln(10) - \varepsilon_2}$$

mit $\frac{1}{2} < \varepsilon_1, \varepsilon_2 < \frac{3}{2}$.

Der Ausdruck ist am kleinsten, wenn ε_1 möglichst klein und ε_2 möglichst groß ist. Entsprechend ist er am größten, wenn ε_1 möglichst groß und ε_2 möglichst klein ist. Setzt man die jeweiligen Grenzen $1/2$ und $3/2$ für ε_1 und ε_2 ein, so erhält man die Abschätzung

$$\frac{2}{9}\cdot\left(\frac{10}{m\cdot\ln(10)-1/2}-\frac{1}{(m-1)\cdot\ln(10)-3/2}\right)<W(m)<\frac{2}{9}\cdot\left(\frac{10}{m\cdot\ln(10)-3/2}-\frac{1}{(m-1)\cdot\ln(10)-1/2}\right).$$

Beispielsweise ergibt sich für $m=200$: 0,0043442 < $W(200)$ < 0,0043558. Im Mittel erhält man daher eine 200-stellige Primzahl nach $1/W(200)$ Schleifendurchläufen. Hier ist $229{,}581<1/W(200)<230{,}192$, also ist im Mittel eine Primzahl nach 230 Schleifendurchläufen zu erwarten. Da diese Aussage jedoch nur „im Mittel“ gilt, ist ein Test auf Primzahleigenschaft unabdingbar.

Um eine natürliche Zahl $n>2$ auf Primzahleigenschaft zu testen, könnte man alle Primzahlen von 2 bis $\lfloor\sqrt{n}\rfloor$ daraufhin untersuchen, ob es eine von ihnen gibt, die n teilt. Wenn die Zahl n nämlich zusammengesetzt ist, d.h. keine Primzahl ist, hat sie einen Primteiler p mit $p\le\sqrt{n}$. Umgekehrt, falls alle Primzahlen p mit $p\le\sqrt{n}$ keine Teiler von n sind, dann ist n selbst eine Primzahl. Die Primzahlen könnte man etwa systematisch erzeugen (in der Literatur unter dem Stichwort „Sieb des Eratosthenes“ zu finden) oder man könnte sie einer Primzahltabelle (falls vorhanden) entnehmen. Allerdings ist dieser Ansatz für sehr große Werte von n nicht praktikabel und benötigt exponentiellen Rechenaufwand (in der Anzahl der Stellen von n): Die Anzahl der Stellen $\beta(n)$ einer Zahl $n\ge 1$ im Zahlensystem zur Basis B ist $\beta(n)=\lfloor\log_B(n)\rfloor+1=\lceil\log_B(n+1)\rceil$, d.h. $\beta(n)\in O(\log(n))$. Die Anzahl der Primzahlen unterhalb $\lfloor\sqrt{n}\rfloor$ beträgt für große n nach dem Primzahlsatz der Zahlentheorie $\pi(\sqrt{n})\sim\frac{n^{1/2}}{\ln(n^{1/2})}$, also ein Wert der Ordnung $O\left(\frac{2^{\beta(n)/2}}{\beta(n)}\right)$. Jede in Frage kommende Primzahl muss daraufhin untersucht werden, ob sie n teilt. Dazu sind mindestens $O((\beta(n))^2)$ viele Bitoperationen erforderlich (siehe Kapitel 4.5). Daher ist der Gesamtaufwand mindestens von der Ordnung $O(\beta(n)\cdot 2^{\beta(n)/2})$.

Durch Anwendung zahlentheoretischer Erkenntnisse hat man versucht, effiziente Primzahltests zu entwickeln. Im Jahr 2002 wurde ein deterministisches Verfahren veröffentlicht, das eine Zahl n daraufhin testet, ob sie eine Primzahl ist[15]. Die Laufzeit dieses Verfahrens ist von der Ordnung $O((\log(n))^{12})$ und damit trotz der polynomiellen Laufzeit für große n nicht praktikalbel. Der bis dahin bekannte schnellste Algorithmus zur Überprüfung einer Zahl n auf

15 Agrawal, M.; Kayal, N.; Saxena, N.: PRIMES is in P, preprint, http://www.cse.iitk.ac.in/news/primality.ps, Aug. 8, 2002.

Primzahleigenschaft, der APRCL-Test, hat eine Laufzeit der Ordnung $O\left((\log(n))^{c(\log(\log(\log(n))))}\right)$ mit einer Konstanten $c > 0$. Das ist jedoch keine polynomielle Laufzeit.

Als effiziente Primzahltest haben sich **probabilistische Algorithmen** vom Monte-Carlo-Typ erwiesen.

Um zu testen, ob eine Zahl *n* eine Primzahl ist oder nicht, versucht man, einen **Zeugen (witness) für die Primzahleigenschaft von** *n* zu finden. Ein Zeuge ist dabei eine Zahl *a* mit $1 \le a \le n-1$, der eine bestimmte Eigenschaft zukommt, aus der man *vermuten* kann, dass *n* Primzahl ist. Dabei muss diese Eigenschaft bei Vorgabe von *a* einfach zu überprüfen sein, und nach Auffinden einiger weniger Zeugen für die Primzahleigenschaft von *n* muss der Schluss gültig sein, dass *n* mit hoher Wahrscheinlichkeit eine Primzahl ist. Die Eigenschaft „die Primzahl *p* mit $2 \le p \le \lfloor \sqrt{n} \rfloor$ ist kein Teiler von *n*“ ist dabei nicht geeignet, da man im allgemeinen zu viele derartige Zeugen zwischen 2 und $\lfloor \sqrt{n} \rfloor$ bemühen müsste, um sicher auf die Primzahleigenschaft schließen zu können.

Es sei $E(a)$ eine (noch genauer zu definierende) Eigenschaft, die einer Zahl *a* mit $1 \le a \le n-1$ zukommen kann und für die gilt:

(i) $E(a)$ ist algorithmisch mit geringem Aufwand zu überprüfen

(ii) falls *n* Primzahl ist, dann trifft $E(a)$ für alle Zahlen *a* mit $1 \le a \le n-1$ zu

(iii) falls *n* keine Primzahl ist, dann trifft $E(a)$ für weniger als die Hälfte aller Zahlen *a* mit $1 \le a \le n-1$ zu.

Falls $E(a)$ gilt, dann heißt *a* **Zeuge (witness) für die Primzahleigenschaft von** *n*. Dann lässt sich folgender randomisierte Primzahltest definieren:

Eingabe: $n \in \mathbf{N}$, *n* ist ungerade, $m \in \mathbf{N}$

Verfahren: Aufruf der Funktion `random_is_prime (n : INTEGER;`
`m : INTEGER): BOOLEAN;`

Ausgabe: *n* wird als Primzahl angesehen, wenn `random_is_prime (n, m) = TRUE` ist, ansonsten wird *n* nicht als Primzahl angesehen.

```
FUNCTION random_is_prime (n  : INTEGER;
                          m  : INTEGER): BOOLEAN;

   VAR idx      : INTEGER;
       a        : INTEGER;
       is_prime : BOOLEAN;



   BEGIN { random_is_prime }
     is_prime := TRUE;

     FOR idx := 1 TO m DO
       BEGIN
         -- wähle eine Zufallszahl a zwischen 1 und n-1;
         IF (E(a) trifft nicht zu)
         THEN BEGIN
                is_prime := FALSE;
                Break;
              END;
       END;

     random_is_prim := is_prime;

   END   { random_is_prime };
```

Der Algorithmus versucht also, m Zeugen für die Primzahleigenschaft von n zu finden. Wird dabei zufällig eine Zahl a mit $1 \le a \le n-1$ erzeugt, für die $E(a)$ nicht zutrifft, dann wird wegen (ii) die korrekte Antwort gegeben. Ist n Primzahl, dann gibt der Algorithmus ebenfalls wegen (ii) die korrekte Antwort. Wurden m Zeugen für die Primzahleigenschaft von n festgestellt, kann es trotzdem sein, dass n keine Primzahl ist, obwohl der Algorithmus angibt, n sei Primzahl. Die Wahrscheinlichkeit, bei einer Zahl n, die nicht Primzahl ist, m Zeugen zu finden, ist wegen (iii) kleiner als $(1/2)^m$, d.h. die Wahrscheinlichkeit einer fehlerhaften Entscheidung ist in diesem Fall kleiner als $(1/2)^m$. Insgesamt ist die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus größer als $1-(1/2)^m$. Da wegen (i) die Eigenschaft $E(a)$ leicht zu überprüfen ist, ist hiermit ein effizientes Verfahren beschrieben, das mit beliebig hoher Wahrscheinlichkeit eine korrekte Antwort liefert. Diese ist beispielsweise für $m = 20$ größer als 0,999999.

Die Frage stellt sich, ob es geeignete Zeugeneigenschaften $E(a)$ gibt. Einen ersten Versuch legt der Satz von Fermat nahe:

Ist $n \in \mathbf{N}$ eine Primzahl und $a \in \mathbf{N}$ eine Zahl mit $ggT(a,n)=1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$.

Für die Primzahl $n=13$ zeigt folgende Tabelle für alle a mit $1 \le a \le n-1$ die Werte $a^i \pmod{13}$ mit i = 1, ..., 12 :

$a \bmod 13$	$a^i \bmod 13$ für $i=1,\ldots,12$
1	1 für $i=1,\ldots,12$
2	2, 4, 8, $16 \equiv 3$, 6, 12, $24 \equiv 11$, $22 \equiv 9$, $18 \equiv 5$, 10, $20 \equiv 7$, $14 \equiv 1$
3	3, 9, $27 \equiv 1$, 3, 9, $27 \equiv 1$, 3, 9, $27 \equiv 1$, 3, 9, $27 \equiv 1$
4	4, $16 \equiv 3$, 12, $48 \equiv 9$, $36 \equiv 10$, $40 \equiv 1$, 4, $16 \equiv 3$, 12, $48 \equiv 9$, $36 \equiv 10$, $40 \equiv 1$
5	5, $25 \equiv 12$, $60 \equiv 8$, $40 \equiv 1$, 5, $25 \equiv 12$, $60 \equiv 8$, $40 \equiv 1$, 5, $25 \equiv 12$, $60 \equiv 8$, $40 \equiv 1$
6	6, $36 \equiv 10$, $60 \equiv 8$, $48 \equiv 9$, $54 \equiv 2$, 12, $72 \equiv 7$, $42 \equiv 3$, $18 \equiv 5$, $30 \equiv 4$, $24 \equiv 11$, $66 \equiv 1$
7	7, $49 \equiv 10$, $70 \equiv 5$, $35 \equiv 9$, $63 \equiv 11$, $77 \equiv 12$, $84 \equiv 6$, $42 \equiv 3$, $21 \equiv 8$, $56 \equiv 4$, $28 \equiv 2$, $14 \equiv 1$
8	8, $64 \equiv 12$, $96 \equiv 5$, $40 \equiv 1$, 8, $64 \equiv 12$, $96 \equiv 5$, $40 \equiv 1$, 8, $64 \equiv 12$, $96 \equiv 5$, $40 \equiv 1$
9	9, $81 \equiv 3$, $27 \equiv 1$, 9, $81 \equiv 3$, $27 \equiv 1$, 9, $81 \equiv 3$, $27 \equiv 1$, 9, $81 \equiv 3$, $27 \equiv 1$
10	10, $100 \equiv 9$, $90 \equiv 12$, $120 \equiv 3$, $30 \equiv 4$, $40 \equiv 1$, 10, $100 \equiv 9$, $90 \equiv 12$, $120 \equiv 3$, $30 \equiv 4$, $40 \equiv 1$
11	11, $121 \equiv 4$, $44 \equiv 5$, $55 \equiv 3$, $33 \equiv 7$, $77 \equiv 12$, $132 \equiv 2$, $22 \equiv 9$, $99 \equiv 8$, $88 \equiv 10$, $110 \equiv 6$, $66 \equiv 1$
12	12, $144 \equiv 1$, 12, $144 \equiv 1$, 12, $144 \equiv 1$, 12, $144 \equiv 1$, 12, $144 \equiv 1$, 12, $144 \equiv 1$

Definiert man $E(a)=$ „$ggT(a,n)=1$ und $a^{n-1} \equiv 1 \pmod{n}$“, dann sieht man, dass (i) und (ii) gelten. Leider gilt (iii) für diese Eigenschaft $E(a)$ nicht. Es gibt nämlich unendlich viele Zahlen *n*, die **Carmichael-Zahlen**, die folgende Eigenschaft besitzen: *n* ist *keine* Primzahl, und es ist $a^{n-1} \equiv 1 \pmod{n}$ für alle *a* mit $ggT(a,n)=1$. Die ersten zehn Carmichael-Zahlen sind 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341.

Ist *n* also eine Carmichael-Zahl (wobei man einige Carmichael-Zahlen wie 561, 1105, 2465 oder 10585 leicht als Nichtprimzahlen erkennt) und *a* eine Zahl mit $1 \le a \le n-1$ und $ggT(a,n)=1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$. In diesem Fall gilt also für *alle* Zahlen *a* mit $1 \le a \le n-1$ und $ggT(a,n)=1$ die Eigenschaft $E(a)$, und das sind mehr als die Hälfte aller Zahlen *a* mit $1 \le a \le n-1$.

Trotzdem führt die Eigenschaft $E(a)=$ „$ggT(a,n)=1$ und $a^{n-1}\equiv 1 \pmod n$“ schon auf einen brauchbaren randomisierten Primzahltest, der jedoch bei Eingabe einer Carmichael-Zahl versagt. Das Beispiel der Nichtprimzahl n = 15 belegt, dass (iii) doch gelten kann:

$a \bmod 15$	$a^i \bmod 15$ für $i=1,\ldots,14$
1	1 für $i=1,\ldots,14$
2	$2, 4, 8, 16\equiv 1, 2, 4, 8, 16\equiv 1, 2, 4, 8, 16\equiv 1, 2, 4$
3	$3,\ 9,\ 27\equiv 12,\ 36\equiv 6,\ 18\equiv 3,\ 9,\ 27\equiv 12,\ 36\equiv 6,\ 18\equiv 3,\ 9,\ 27\equiv 12,\ 36\equiv 6,\ 18\equiv 3, 9$
4	$4, 16\equiv 1, 4, 16\equiv 1, 4, 16\equiv 1, 4, 16\equiv 1, 4, 16\equiv 1, 4, 16\equiv 1, 4, 16\equiv 1$ ***(falscher) Zeuge dafür, dass 15 Primzahl ist***
5	$5,\ 25\equiv 10,\ 50\equiv 5,\ 25\equiv 10,\ 50\equiv 5,\ 25\equiv 10,\ 50\equiv 5,\ 25\equiv 10,\ 50\equiv 5,\ 25\equiv 10,\ 50\equiv 5,\ 25\equiv 10,\ 50\equiv 5,\ 25\equiv 10$
6	$6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6,\ 36\equiv 6$
7	$7,\ 49\equiv 4,\ 28\equiv 13,\ 91\equiv 1,\ 7,\ 49\equiv 4,\ 28\equiv 13,\ 91\equiv 1,\ 7,\ 49\equiv 4,\ 28\equiv 13,\ 91\equiv 1,\ 7,\ 49\equiv 4$
8	$8,\ 64\equiv 4,\ 32\equiv 2,\ 16\equiv 1,\ 8,\ 64\equiv 4,\ 32\equiv 2,\ 16\equiv 1,\ 8,\ 64\equiv 4,\ 32\equiv 2,\ 16\equiv 1,\ 8,\ 64\equiv 4$
9	$9,\ 81\equiv 6,\ 54\equiv 9,\ 81\equiv 6,\ 54\equiv 9,\ 81\equiv 6,\ 54\equiv 9,\ 81\equiv 6,\ 54\equiv 9,\ 81\equiv 6,\ 54\equiv 9,\ 81\equiv 6,\ 54\equiv 9,\ 81\equiv 6$
10	$10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10,\ 100\equiv 10$
11	$11,\ 121\equiv 1,\ 11,\ 121\equiv 1,\ 11,\ 121\equiv 1,\ 11,\ 121\equiv 1,\ 11,\ 121\equiv 1,\ 11,\ 121\equiv 1, 11, 121\equiv 1$ ***(falscher) Zeuge dafür, dass 15 Primzahl ist***
12	$12,\ 144\equiv 9,\ 108\equiv 3,\ 36\equiv 6,\ 72\equiv 12,\ 144\equiv 9,\ 108\equiv 3,\ 36\equiv 6,\ 72\equiv 12,\ 144\equiv 9,\ 108\equiv 3,\ 36\equiv 6,\ 72\equiv 12,\ 144\equiv 9$
13	$13,\ 169\equiv 4,\ 52\equiv 7,\ 91\equiv 1,\ 13,\ 169\equiv 4,\ 52\equiv 7,\ 91\equiv 1,\ 13,\ 169\equiv 4,\ 52\equiv 7,\ 91\equiv 1, 13, 169\equiv 4$
14	$14,\ 196\equiv 1,\ 14,\ 196\equiv 1,\ 14,\ 196\equiv 1,\ 14,\ 196\equiv 1,\ 14,\ 196\equiv 1,\ 14,\ 196\equiv 1,\ 14,\ 196\equiv 1$ ***(falscher) Zeuge dafür, dass 15 Primzahl ist***

Es gilt folgender Satz:

Entweder gilt für alle Zahlen a mit $1\le a\le n-1$ und $ggT(a,n)=1$ die Eigenschaft $a^{n-1}\equiv 1 \pmod n$ oder für höchstens die Hälfte.

Anders formuliert:

Entweder sind mit der Zeugeneigenschaft $E(a)=$ „$ggT(a,n)=1$ und $a^{n-1}\equiv 1 \pmod n$“ alle Zahlen a mit $1\le a\le n-1$ Zeugen für die Primzahleigenschaft für n oder höchstens die Hälfte.

Ein zweiter Versuch zur geeigneten Definition einer Zeugeneigenschaften $E(a)$ erweitert den ersten Versuch und wird durch die folgenden mathematischen Sätze aus der Zahlentheorie begründet:

Es sei n eine Primzahl. Dann gilt $x^2\equiv 1 \pmod n$ genau dann, wenn $x\equiv 1 \pmod n$ oder $x\equiv -1\equiv n-1 \pmod n$ ist.

Es sei n eine Primzahl mit $n>2$. Dann ist n ungerade, d.h. $n=1+2^j\cdot r$ mit ungeradem r und $j>0$ bzw. $n-1=2^j\cdot r$. Ist a eine Zahl mit $1\le a\le n-1$, dann ist $ggT(a,n)=1$ und folglich $a^{n-1}\equiv 1 \pmod n$. Wegen $a^{n-1}=a^{(2^{j-1}\cdot r)2}=\left(a^{2^{j-1}\cdot r}\right)^2\equiv 1 \pmod n$ folgt (im vorhergehenden Satz wird $x=a^{2^{j-1}\cdot r}$ gesetzt): $a^{2^{j-1}\cdot r}\equiv 1 \pmod n$ oder $a^{2^{j-1}\cdot r}\equiv -1 \pmod n$. Ist hierbei $j-1>0$ und $a^{2^{j-1}\cdot r}\equiv 1 \pmod n$, dann kann man den Vorgang des Wurzelziehens wiederholen: dazu wird $x=a^{2^{j-2}\cdot r}$ gesetzt usw. Der Vorgang ist spätestens dann beendet, wenn $a^{2^{j-j}\cdot r}=a^r$ erreicht ist.

Es gilt daher der folgende Satz:

Es sei n eine ungerade Primzahl, $n=1+2^j\cdot r$ mit ungeradem r und $j>0$. Dann gilt für jedes a mit $1\le a\le n-1$:

Die Folge
$\left(a^r \pmod n, a^{2\cdot r} \pmod n, a^{4\cdot r} \pmod n, \ldots, a^{2^{j-1}\cdot r} \pmod n, a^{2^j\cdot r} \pmod n\right)$
der Länge $j+1$ hat eine der Formen

$(1,1,1,\ldots,1,1)$ oder

$(*,*,\ldots,*,-1,1,\ldots,1,1)$.

Hierbei steht das Zeichen „*“ für eine Zahl, die verschieden von 1 oder -1 ist.

Wenn die Folge eine der drei Formen

$(*,*,\ldots,*,1,1,\ldots,1,1)$,

$(*,*,\ldots,*,-1)$ oder

$(*,*,\ldots,*,*,\ldots,*)$

aufweist, dann ist n mit Sicherheit keine Primzahl. Andererseits ist es nicht ausgeschlossen, dass für eine ungerade zusammengesetzte Zahl n und eine Zahl a mit $1\le a\le n-1$ die Folge $\left(a^r \pmod n, a^{2\cdot r} \pmod n, a^{4\cdot r} \pmod n, \ldots, a^{2^{j-1}\cdot r} \pmod n, a^{2^j\cdot r} \pmod n\right)$ eine der beiden Formen

$(1,1,1,...,1,1)$ oder $(*,*,...,*,-1,1,...,1,1)$ hat. In diesem Fall heißt *n* **streng pseudoprim zur Basis** *a*.

Beispielsweise lauten für die zusammengesetzte Zahl $n = 2047 = 23 \cdot 89$ mit $a = 2$ wegen $n-1 = 2046 = 2^1 \cdot 1023$ die Werte $j = 1$ und $r = 1023$, und die Folge
$\left(a^r \pmod n, a^{2\cdot r} \pmod n, a^{4\cdot r} \pmod n, ..., a^{2^{j-1}\cdot r} \pmod n, a^{2^j\cdot r} \pmod n\right)$ ist
$\left(2^{1023} \pmod{2047}, 2^{2046} \pmod{2047}\right) = (1,1)$.

Es gilt jedoch folgender Satz:

Es sei *n* eine ungerade zusammengesetzte Zahl. Dann ist *n* streng pseudoprim für höchtens ein Viertel aller Basen *a* mit $1 \le a \le n-1$.

Eine geeignete Zeugeneigenschaften $E(a)$ für die Funktion

```
random_is_prime (n : INTEGER;
                 m : INTEGER): BOOLEAN;
```

ist daher die folgende Bedingung:

$E(a) =$ „$ggT(a,n) = 1$ und
$a^{n-1} \equiv 1 \pmod n$ und
die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1}\cdot r}, a^{2^j\cdot r}\right) \bmod n$ hat eine der Formen
$(1,1,1,...,1,1)$ oder $(*,*,...,*,-1,1,...,1,1)$“.

Die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus mit dieser Zeugeneigenschaft ist gemäß vorigem Satz größer als $1-(1/4)^m$.

Um für praktische Belange die Güte des Verfahrens abzuschätzen, werden in folgender Tabelle einige Werte für $(1/4)^m$ abgeschätzt:

m	10	25	30	50	100	168	1000
$(1/4)^m$	$<10^{-6}$	$<10^{-15}$	$<10^{-18}$	$<10^{-30}$	$<10^{-60}$	$<10^{-101}$	$<10^{-602}$

Ist *n* also eine zusammengesetzte Zahl und werden $m = 100$ zufällig erzeugte Zahlen *a* ausgewählt und auf Zeugeneigenschaft untersucht, so ist die Wahrscheinlichkeit, dass alle diese ausgewählten Zahlen „falsche Zeugen“ sind, d.h. die Primzahleigenschaft von *n* fälschlicherweise bezeugen, kleiner als 10^{-60}.

Es ist übrigens nicht notwendig, für eine große Anzahl von Zahlen *a* mit $1 \le a \le n-1$ die Zeugeneigenschaft zu überprüfen um sicher zu gehen, dass *n* eine Primzahl ist: Nur eine zu-

sammengesetzte Zahl $n < 2{,}5 \cdot 10^{10}$, nämlich n = 3.215.031.751, ist streng pseudoprim zu den vier Basen a = 2, 3, 5 und 7. Für praktische Belange ist daher das Verfahren ein effizienter Primzahltest.

Untersucht man große Zahlen, die spezielle Formen aufweisen, etwa Mersenne-Zahlen, auf Primzahleigenschaft bieten sich speziell angepasste Testverfahren an. Schließlich gibt es eine Reihe von Testverfahren, die andere zahlentheoretische Eigenschaften nutzen.

8.2.2 Stochastische Approximation des Binpacking-Problems

Ein möglicher Ansatz, der probabilistische Eigenschaften der Eingabeinstanzen ausnutzt, besteht im Entwurf eines Problemlösungsverfahrens, das im deterministischen Fall eventuell unbefriedigende Lösungen (d.h. eventuell nicht korrekte bzw. bei Optimierungsaufgaben nicht-optimale Lösungen) erzeugt, sich im Mittel aber „sehr gut“ verhält. Das folgende Beispiel erläutert diesen Ansatz.

Bei einem **Optimierungsproblem** besteht die Aufgabe im Auffinden eines Algorithmus, der bei Eingabe einer Instanz I unter allen für I zulässigen Lösungen y eine optimale Lösung y^* findet. Die Optimalität ist durch das numerische Maß $m(I, y)$ festgelegt, nämlich

bei einem Minimierungsproblem: $m(I, y^*) \leq m(I, y)$ für alle für I zulässigen Lösungen y,
bei einem Maximierungsproblem: $m(I, y^*) \geq m(I, y)$ für alle für I zulässigen Lösungen y.

Häufig ist die Komplexität eines Optimierungsalgorithmus exponentiell in der Größe der Eingabe und damit in den meisten Fällen nicht praktikabel, so dass man nach einem **Approximationsalgorithmus A** sucht, der bei Eingabe von I eine für I zulässige Lösung $\mathbf{A}(I)$ findet, deren Maß $m(I, \mathbf{A}(I))$ sich dem Maß $m(I, y^*)$ einer optimalen Lösung y^* nähert, die zu finden aber vertretbaren, d.h. meist polynomiellem Zeitaufwand erfordert. Als Gütekriterium des Approximationsalgorithmus wird häufig die relative Approximationsgüte bzw. deren Erwartungswert, d.h. die mittlere relative Approximationsgüte, genommen:

Die **relative Approximationsgüte** eines Algorithmus **A** ist
bei einem Minimierungsproblem definiert durch $R_{\mathbf{A}}(I) = \dfrac{m(I, \mathbf{A}(I))}{m(I, y^*)}$,
bei einem Maximierungsproblem definiert durch $R_{\mathbf{A}}(I) = \dfrac{m(I, y^*)}{m(I, \mathbf{A}(I))}$.

Wegen $R_{\mathbf{A}}(I) \geq 1$ ist der Algorithmus **A** „besonders gut“, wenn $R_{\mathbf{A}}(I)$ möglichst dicht bei 1 liegt.

Bei der **mittleren relativen Approximationsgüte** unterscheidet man noch folgende Zielgrößen:

- erwartetes Approximationsergebnis bei Eingaben der Größe n: $\mathbf{E}\left[m(I,\mathbf{A}(x)) \mid |I| = n\right]$
- erwartete relative Approximationsgüte bei Eingaben der Größe n:
  bei einem Minimierungsproblem $\mathbf{E}\left[m(I,\mathbf{A}(I))/m^*(I) \mid |I| = n\right]$
  bei einem Maximierungsproblem $\mathbf{E}\left[m^*(I)/m(I,\mathbf{A}(I)) \mid |I| = n\right]$
- erwartete Approximationsgüte bei „großen“ bzw. fast allen Eingabegrößen: $\lim\limits_{n\to\infty} \mathbf{E}\left[m(I,\mathbf{A}(I) \mid I = n\right]$
- erwartete relative Approximationsgüte bei „großen“ bzw. fast allen Eingabengrößen:
  bei einem Minimierungsproblem $\lim\limits_{n\to\infty} \mathbf{E}\left[m(I,\mathbf{A}(I))/m^*(I) \mid |I| = n\right]$
  bei einem Maximierungsproblem $\lim\limits_{n\to\infty} \mathbf{E}\left[m^*(I)/m(I,\mathbf{A}(I)) \mid |I| = n\right]$.

Ein typisches Zuordnungsproblem ist das Binpacking-Optimierungsproblem:

Binpacking-Minimierungsproblem:

Problem-
instanz: $[a_1, \ldots, a_n, b]$
$a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \ldots, n$, $b \in \mathbf{N}$ („Behältergröße“).

Gesuchte
Lösung: Eine Zuordnung der Objekte auf möglichst wenige Behälter mit Behältergröße b, so dass die Summe der Objekte in einem Behälter nicht größer als b ist (kein Behälter „läuft über“).
Formal:
Eine zulässige Lösung ist eine Abbildung $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$, so dass für alle $j \in \{1, \ldots, k\}$ gilt: $\sum\limits_{f(i)=j} a_i \le b$ („diejenigen Objekte, die in den j-ten Behälter gelegt werden, überschreiten die Behältergröße nicht“, $f(i)$ ist die Nummer des Behälters, in den a_i gelegt wird). Das Maß einer zulässigen Lösung ist k. Eine optimale Lösung (Anzahl benötigter Behälter) ist eine zulässige Lösung mit minimalem Wert k.

Die Problemgröße ist beschränkt durch $n \cdot A$ mit
$A = \max\left(\left\{\lfloor \log_2(a_i) \rfloor + 1 \mid i = 1, \ldots, n\right\} \cup \left\{\lfloor \log_2(b) \rfloor + 1\right\}\right)$.

Das zum Binpacking-Minimierungsproblem zugehörige Entscheidungsproblem ist **NP**-vollständig, so dass kein Optimierungsalgorithmus mit polynomieller Laufzeit zu erwarten ist. Es sind daher zahlreiche Approximationsalgorithmen entwickelt worden ([A/.]). Das folgende sogenannte zeitbeschränkte Approximationsschema ist ein online Approximationsalgorithmus für das Binpacking-Minimierungsproblem, d.h. jedes Objekt nur einmal inspiziert und eine Packungsentscheidung trifft, ohne die nachfolgenden Objekte zu kennen. Es ist genau auf die Voraussetzung der Gleichverteilung der Objekt in einer Eingabeinstanz zugeschnitten. Daher kann es vorkommen, dass es sich bei nicht Erfüllung dieser Voraussetzung „schlecht" verhält. Außerdem kann es auf allgemeinere Verteilungen der Objekt in der Eingabeinstanz verallgemeinert werden, nämlich auf stetige Verteilungen, deren Dichtefunktion monoton fällt, insbesondere für die kleine Objekte häufiger vorkommen als große.

Zur Vereinfachung wird das Verfahren so formuliert, dass die Behältergröße b gleich 1 ist und die Objekte a_i in der Eingabeinstanz rationale Zahlen mit $0 < a_i \leq 1$ sind.

Online Approximationsschema für das Binpacking-Minimierungsproblem (online Faltungsalgorithmus)

Eingabe: $I = [a_1, \ldots, a_n]$, $s \in \mathbf{N}$ mit $s \geq 4$ und $s \equiv 0 \bmod(2)$

$a_1, \ldots, a_n$ („Objekte") sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren:

```
VAR idx : INTEGER;
    k   : INTEGER;
    I   : ARRAY [1..s] OF Intervall ⊆ ]0,1];

BEGIN
  FOR idx := 1 TO s DO
    I[idx] := ]( s-idx)/s , (s-idx+1)/s ];

  FOR k := 1 TO n DO
    BEGIN
      idx := s + 1 - ⌈s·a_k⌉; { Bestimmung eines Intervalls
                                I[idx] mit a_k ∈ I[idx]     }
      IF (idx >= 2)
         AND
         (es gibt einen Behälter B, der ein Objekt b ∈ I[s-(idx-2)]
          enthält)
      THEN BEGIN
             B := B ∪ {a_k};
```

```
                kennzeichne B als gefüllt;
            END
        ELSE BEGIN
                plaziere a_k in einen neuen Behälter;
                { für idx = 1 ist dieser Behälter gefüllt }
            END;
    END;
```

Ausgabe: $\mathbf{OFD}(I) =$ benötigte nichtleere Behälter $B_1, ..., B_k$.

Offensichtlich hat das Verfahren polynomielle Laufzeit (in der Größe der Eingabeinstanz). Es lässt sich zeigen[16]:

Bezeichnet $OPT(I_n)$ das Maß einer minimalen Lösung bei einer Eingabeinstanz $I_n = [a_1, \dots, a_n]$ für das Binpacking-Minimierungsproblem und sind die Objekte in einer Eingabeinstanz im Intervall $]0,1]$ gleichverteilt, so gilt
$1 \le \lim_{n\to\infty} \mathbf{E}\left[OPT(I_n)/m(I_n, \mathbf{OFD}(I_n))\right] \le 1 + 1/s$.

Das Verfahren nutzt also eine bekannte probabilistische Eigenschaft der Eingabeinstanz.

16 Hoffmann, U.: A class of simple stochastic bin packing algorithms, Computing 29, 227 – 239, 1982.