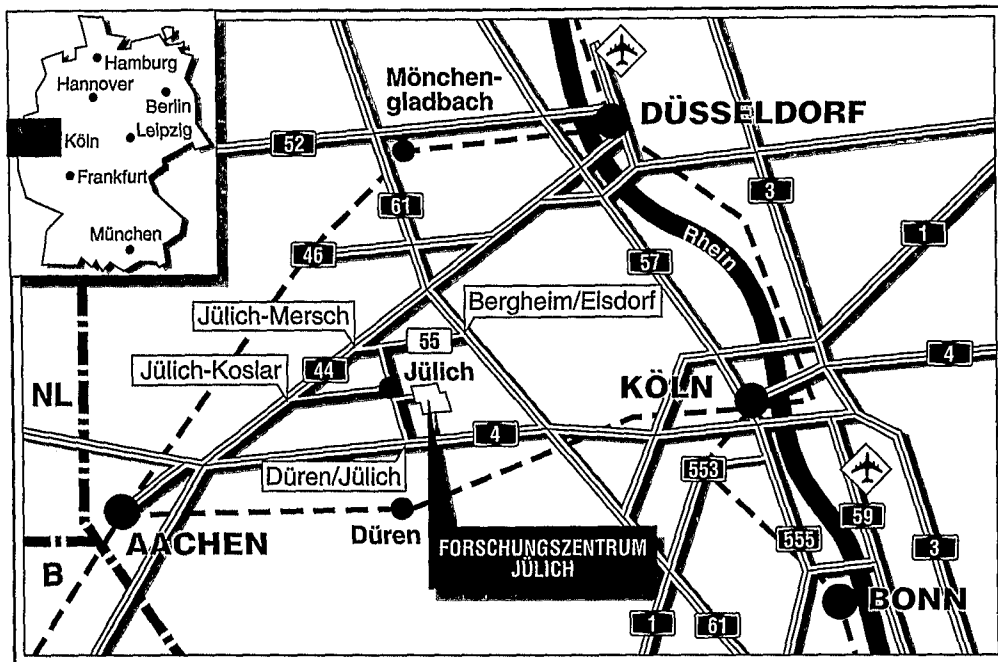


Zentralinstitut für Angewandte Mathematik

Algorithmen und Strukturen

Wolfgang E. Nagel · Renate Knecht



Berichte des Forschungszentrums Jülich ; 2527

ISSN 0366-0885

Zentralinstitut für Angewandte Mathematik Jül-2527

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek

Postfach 1913 · D-5170 Jülich · Bundesrepublik Deutschland

Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Algorithmen und Strukturen

Wolfgang E. Nagel · Renate Knecht

Vorwort

Der Kurs '*Algorithmen und Strukturen*' ist der zweite von derzeit 4 Informatikkursen im Rahmen der Ausbildung von Mathematisch-Technischen Assistentinnen und Assistenten (MTAs) im Forschungszentrum Jülich. Eines der Hauptmerkmale dieses Berufsbildes ist die Praxisbezogenheit des Unterrichts, d.h. die Auszubildenden sollen frühzeitig in der Lage sein, an praktischen Problemlösungen in den Instituten mitzuwirken und ihre Kenntnisse einzubringen. Die hier vorliegenden Materialien stützen sich z.T. stark auf die im Literaturverzeichnis angegebenen Informatikbücher, die sich an einen anderen Leserkreis wenden. Hier wurde der Versuch gemacht, anhand einer selektiven Auswahl aus der Standardliteratur den Stoff zu extrahieren, der im Hinblick auf die Ausbildung und das Berufsbild des Mathematisch-Technischen Assistenten ausreichende praktische Relevanz besitzt und trotzdem in ca. 50 Unterrichtsstunden vermittelbar bleibt.

Die Einführung in die Datenstrukturen basiert auf dem formalen Ansatz des *abstrakten Datentyps*, der sich in den letzten Jahren für den Zugang zu den Datenstrukturen als vorteilhaft erwiesen hat. Da sehr formale Methoden jedoch für eine Einführung nur bedingt geeignet sind, wurde versucht, anhand von Beispielen den Grundgedanken der abstrakten Datentypen und die damit verbundenen Vorteile einer *formalen Spezifikation* von Strukturen zu vermitteln.

Die Datenstrukturen werden mit Hilfe einer *Spezifikation* eingeführt, die eine Beschreibung der Datenelemente, der Struktur zwischen den Datenelementen und der zulässigen Operationen auf diesen Datentypen umfaßt. Die Operationen werden bei dieser Vorgehensweise mit sog. *pre-* und *post-conditions* beschrieben, die die Einwirkungen der Operationen auf das Umfeld vollständig spezifizieren. Ein zentraler Punkt ist die implementationsunabhängige Spezifikation der Datentypen. Anhand von Beispielen wird gezeigt, wie unterschiedlich die Implementationen für eine Spezifikation einer Datenstruktur ausfallen können. Durch diesen Ansatz ist nicht unmittelbar vorherbestimmt, in welcher Programmiersprache die Datenstruktur implementiert werden wird. Ebenfalls können sich bei unterschiedlichen Implementationen u.U. signifikante Unterschiede im Verbrauch von Systemressourcen ergeben. Das Material basiert wesentlich auf dem Buch '*Data Structures with Abstract Data Types and Pascal*' von D.F. Stubbs und N.W. Webre [10], wobei jedoch deutliche Modifikationen und Kürzungen vorgenommen wurden. Leider konnte das Konzept des *Moduls* und des *Packages* nicht adäquat berücksichtigt werden, da die Implementationen in den Standard-Programmiersprachen Pascal und FORTRAN vorgenommen werden sollten. Weitere wichtige Quellen für dieses Kapitel waren z.B. die Bücher '*Intermediate Problem Solving and Data Structures: Walls and Mirrors*' von P. Helman und R. Veroff [4] und '*Data Structures and Algorithms*' von A.V. Aho, J.E. Hopcroft und J.D. Ullman [2], daneben gab das Manuskript '*Datenstrukturen*' von R.H. Güting [3] einige Anregungen.

Ein wesentlicher Teil der im Kapitel 4 beschriebenen Algorithmen basiert auf Unterlagen von U. Detert, der diesen Kurs bereits 1984/85 im Rahmen der Ausbildung für MTAs durchgeführt hat. Aufgrund der wesentlich erhöhten Wochenstunden, die heute für diesen Kurs zur Verfügung stehen, konnten jedoch auch in diesem Bereich Erweiterungen vorgenommen werden. So ist z.B. ein Kapitel über Graphenalgorithmen zugefügt worden. Als Basisliteratur wurde dort neben den oben genannten Literaturstellen besonders auf die Bücher '*Algorithms*' von R. Sedgewick [11] und '*Sorting and Searching: The Art of Computer Programming Vol.3*' von D.E. Knuth [7] zurückgegriffen.

Die erste Version dieses Manuskripts basiert auf einer Mitschrift des Kurses *Algorithmen und Strukturen*, der im Sommer 1988 von W.E. Nagel durchgeführt wurde. Die Mitschrift wurde von den Kursteilnehmern angefertigt. Die Integration der einzelnen Teile sowie ein wesentlicher Teil der Nachbearbeitung wurde in mehrmonatiger Tätigkeit von Herrn W. Frings durchgeführt. Nur durch seinen unermüdlichen Einsatz bei der Erstellung von zusätzlichen grafischen Darstellungen, bei der Umwandlung von Programm-Code in Nassi-Shneiderman-Diagramme und bei der Beseitigung von Fehlern konnte das Manuskript in der jetzt vorliegenden Form verfügbar gemacht werden. Für seine Bemühungen danken wir Herrn Frings sehr herzlich.

August 1991

Die Autoren

Inhaltsverzeichnis

1. Einleitung	1
2. Einführung der Grundbegriffe	3
2.1 Algorithmen als Lösungsvorschrift	3
2.1.1 Effizienz von Algorithmen	4
2.1.2 Komplexität von Algorithmen als Maß für die Effizienz	5
2.2 Datenstrukturen als Basis für effiziente Algorithmen	10
2.2.1 Datentypen und Datenstrukturen	10
2.2.2 Abstraktionsniveaus bei Datentypen	12
2.2.3 Physikalische Speicherung von Daten	13
2.2.4 Spezifikation von Basisoperationen auf Datenstrukturen	15
3. Datenstrukturen	17
3.1 Einfache Datenstrukturen	18
3.1.1 Felder (arrays)	19
3.1.2 Sätze (records)	22
3.1.3 Zeiger (pointer)	25
3.1.4 Mengen (sets)	28
3.1.5 Sequenzen (sequences)	31
3.2 Lineare Datenstrukturen	39
3.2.1 Einfach verkettete Listen (linked lists)	39
3.2.1.1 Realisierung über Pointer	45
3.2.1.2 Realisierung mit Hilfe von Feldern	51
3.2.1.3 Realisierung durch ein statisches Feld ohne Pointer	53
3.2.2 Mehrfach verkettete Listen	55
3.2.3 Stapel (stacks)	60
3.2.4 Warteschlangen (queues)	69
3.3 Graphen	74
3.4 Bäume	80
3.4.1 Binärbäume	81
3.4.2 Binäre Suchbäume als Anwendung von Binärbäumen	95
3.4.3 B-Bäume	108
3.5 Haufen (heaps)	112
4. Ausgewählte Algorithmen	119
4.1 Suchverfahren	119
4.1.1 Lineares Suchen (Sequential Search)	120
4.1.2 Binäres Suchen (Binary Search)	120
4.2 Sortierv Verfahren	123
4.2.1 Allgemeine Voraussetzungen für Sortierv Verfahren	124
4.2.2 Sortieren durch Auswahl (Selection Sort)	125

4.2.3 Sortieren durch Einfügen (Insertion Sort)	128
4.2.4 Sortieren durch Vertauschen (Bubble Sort, Sinking Sort)	130
4.2.5 Heapsort	132
4.2.6 Quicksort	134
4.2.7 Sortieren durch Mischen (Mergesort)	140
4.3 Hash-Verfahren	146
4.3.1 Offene Hash-Verfahren	148
4.3.1.1 Der Divisions-Hash	150
4.3.1.2 Der Fibonacci-Hash	153
4.3.1.3 Zugriffszeiten beim Suchen und Speichern in Hash-Tabellen	156
4.3.2 Hash in Teillisten	159
4.4 Algorithmen auf Graphen	161
4.4.1 Das Durchlaufen von gerichteten Graphen	161
4.4.2 Kürzeste Wege von einer Quelle	165
4.4.3 Kürzeste Wege zwischen allen Knotenpaaren	168
4.4.4 Transitiver Abschluß	170
Anhang A. Die Turingmaschine	173
Anhang B. Literatur	177
Index	179

Verzeichnis der Abbildungen

Abb. 1.	Zeitverhalten der einzelnen Algorithmen	9
Abb. 2.	Zerlegung der Klasse der Datentypen	12
Abb. 3.	Abstraktionsniveaus bei Datentypen	13
Abb. 4.	Spezifikation eines strukturierten abstrakten Datentyps	14
Abb. 5.	Elementare Strukturrelationen	17
Abb. 6.	Beispiel für Postleitzahlen	19
Abb. 7.	Strukturbaum und Speicherdarstellung der Datenstruktur Mitarbeiter	25
Abb. 8.	Beispielprogramm mit Pointern	27
Abb. 9.	Beispiel für die Darstellung einer einfach verketteten Liste	40
Abb. 10.	Liste vor und nach der Ausführung von insert_after	44
Abb. 11.	Liste mit drei Feldern	50
Abb. 12.	Beispiel für die Darstellung einer doppelt verketteten Liste	56
Abb. 13.	Beispielhafte Darstellung eines Stacks	61
Abb. 14.	Darstellung der Operation push auf dem Datentyp Stack	64
Abb. 15.	Visualisierung der Feldimplementation eines Stacks	64
Abb. 16.	Mittelbarer rekursiver Aufruf	66
Abb. 17.	Queue (übliche Darstellung)	71
Abb. 18.	Queue, realisiert mit einem Feld bei unterschiedlichem Füllungsgrad	72
Abb. 19.	Visualisierung einer zirkulären Queue	72
Abb. 20.	Beispiel für einen gerichteten Graph	75
Abb. 21.	Übergang von symmetrisch gerichtetem zu einem ungerichteten Graph	76
Abb. 22.	Königsberger Brückenproblem	76
Abb. 23.	Graphentheoretische Problemformulierung	77
Abb. 24.	Auswahl von Wegen in einem Graphen G	78
Abb. 25.	Durchlaufstrategien eines Baumes	83
Abb. 26.	Speicherung des Infix-Ausdruck $(a + b) * c + 7$	84
Abb. 27.	Speicherung des Infix-Ausdruck $(a + b) * (c + 7)$	84
Abb. 28.	Beispielbäume	85
Abb. 29.	Baumdarstellung mit Pointer	88
Abb. 30.	Baumdarstellung mit statischen Feldern	89
Abb. 31.	Beispiele für binäre Suchbäume	96
Abb. 32.	Unterschiedliche Suchbäume mit gleichem Informationsgehalt	106
Abb. 33.	Binärer Suchbaum und Suchpfad zu Element 27	123
Abb. 34.	Speicherreferenzdiagramm für den Quicksort-Algorithmus	137
Abb. 35.	Mischen von sortierten Listen A und B zu einer sortierten Liste C	141
Abb. 36.	address calculator	147
Abb. 37.	Darstellung der Swierczkowsky-Verteilung für $N=7$	154
Abb. 38.	Qualitativer Verlauf von $S(n,k)$ für ein festes n	157
Abb. 39.	Kollisionen beim Fibonacci-Hash	158
Abb. 40.	Turingmaschine	174

1. Einleitung

Analog zu der Vorgehensweise in den Ingenieurwissenschaften sind auch in der Datenverarbeitung bei der Entwicklung von Problemlösungen mehrere Schritte notwendig, die nacheinander vollzogen werden müssen, um gute Resultate und damit effiziente Realisierungen zu erzielen. Nach der Problemerkennung und -formulierung steht bei Aufgabenstellungen in der Informatik zunächst die Modellbildung im Vordergrund; durch die Beschreibung des Modells werden bereits Randbedingungen festgelegt, die sich in späteren Schritten überhaupt nicht oder aber unter Umständen nur mit sehr hohem Aufwand modifizieren lassen. In einem zweiten Schritt kann dann in dem zuvor festgelegten Modell ein Lösungsentwurf entwickelt werden, der durch die Angabe eines Algorithmus fixiert wird und damit die Handlungsanweisungen zur Lösung des Problems bestimmt. Anschließend muß dieser Algorithmus in einer geeigneten Programmiersprache auf einem Rechner implementiert und getestet werden. Die Qualität der in allen Schritten notwendigen und begleitend vorzunehmenden Dokumentation bestimmt dabei maßgeblich die langfristige Verfügbarkeit der dann fertiggestellten Software.

Die vorgestellten Algorithmen kennzeichnen jeweils Lösungsvorschriften für ein gegebenes Problem, die Präsentation kann auf sehr unterschiedliche Arten erfolgen. Eine Möglichkeit besteht darin, in einer verbalen Umschreibung die Vorgehensweise anzugeben. Weiterhin kann man, neben der Verwendung einer höheren Programmiersprache wie Pascal, auch eine Darstellung in Pseudocode wählen, die aus einer Kombination von Programmstrukturen einer Programmiersprache und informellen Anweisungen besteht. Eine dritte Möglichkeit zur Beschreibung eines Algorithmus sind grafische Darstellungen wie beispielsweise die Nassi-Shneiderman-Diagramme, die in der vorliegenden Arbeit für die Vorstellung der Algorithmen verwendet werden.

Nach einer Einführung in die Grundbegriffe werden im ersten Teil dieses Manuskripts an Hand von fundamentalen mathematischen Strukturen wie z.B. Feldern, Mengen, Listen, Graphen, Bäumen etc. zunächst Möglichkeiten vorgestellt, in welcher Weise diese Strukturen auf Rechner abgebildet werden können, so daß die auf diesen Strukturen definierten Operationen (wie z.B. Zugriffsoperationen in Bäumen, Sortieren von Listenelementen) effizient durchgeführt werden können. Die daraus resultierenden Datenstrukturen beschreiben sowohl die Methode der Speicherung als auch die eindeutig festgelegten Basisoperationen auf den Daten. In diesem Teil werden durch Vorstellung und Implementation von Datenstrukturen und den zugehörigen Basisoperationen Programmkerne zur Verfügung gestellt, mit deren Hilfe dann größere Problemlösungen für komplexere Aufgabenstellungen kombiniert werden können. Neben der Präsentation der Operationen stehen Aufwandsbetrachtungen im Vordergrund, die den Vergleich der unterschiedlichen Algorithmen im Hinblick auf Zeitbedarf und benötigten Speicherplatz ermöglichen.

Der zweite Teil des Manuskripts beschäftigt sich mit einigen wichtigen Algorithmen für unterschiedliche Aufgabenstellungen (z.B. Suchen und Sortieren von Listenelementen, effiziente Speichermethoden, Graphenalgorithmen), die universell verwendbar implementiert und dann sehr häufig als Bausteine in Lösungen für komplexe Probleme integriert werden können. Für die unterschiedlichen Bereiche werden exemplarisch Algorithmen vorgestellt und in Bezug auf den Berechnungsaufwand analysiert.

2. Einführung der Grundbegriffe

Eine Lösung für ein Problem besteht generell aus zwei Komponenten: dem Algorithmus und den zugehörigen Datenstrukturen. Algorithmen arbeiten auf Datenstrukturen, und Datenstrukturen enthalten Algorithmen als Komponenten. Da es für viele der verwendeten Begriffe keine eindeutigen und gleichzeitig allgemein akzeptierten Definitionen gibt, werden in diesem Abschnitt die verwendeten Begriffe eingeführt und wichtige Zusammenhänge und Vorgehensweisen erläutert.

2.1 Algorithmen als Lösungsvorschrift

Hat man ein geeignetes mathematisches Modell für das gestellte Problem gefunden, kann im Rahmen der durch das Modell vorgegebenen Möglichkeiten eine Lösung in Form eines *Algorithmus* entwickelt werden. Ein Algorithmus kann als eine "Menge von Regeln" interpretiert werden, die eine Folge von Operationen beschreibt und als Lösungsvorschrift verwendet werden kann. Es gibt eine Reihe von Eigenschaften, die die Qualität eines Algorithmus maßgeblich beeinflussen. Zunächst einmal sollte ein Algorithmus notwendigerweise korrekt arbeiten, d.h. zu einer gegebenen Eingabe eine korrekte Ausgabe erzeugen. Daneben gibt es eine Reihe von wünschenswerten Eigenschaften wie z.B. einfache Implementierbarkeit sowie effizienter Zeit- und Speicherbedarf; auch sollte der Algorithmus leicht zu verstehen sein. Jede dieser Eigenschaften ist wünschenswert, unter Umständen lassen sich jedoch nicht alle Punkte gleichzeitig und jeweils optimal erfüllen. Da die Bewertung auch von subjektiven Empfindungen geleitet wird, sind formale Methoden notwendig, um Vergleichs- und Beurteilungsmaßstäbe für unterschiedliche Algorithmen einer Problemklasse zu erhalten.

Formale Anforderungen an einen Algorithmus

- **Endlichkeit:** Der Algorithmus stoppt nach einer endlichen Anzahl von Operationen (Schritten).
- **Definitheit:** Die Regeln sind eindeutig und präzise (Determinismus versus Non-Determinismus), die Reihenfolge der durchzuführenden Operationen ist (ggf. in Abhängigkeit von den Eingabedaten) eindeutig festgelegt.
- **Anfangszustand:** Der Algorithmus beginnt mit einem eindeutig definierten Anfangszustand, z.B. haben die Eingabedaten einen wohldefinierten Wert; der Startwert des

Algorithmus ist damit festgelegt. Daten, die im Algorithmus verarbeitet werden, haben definierte Werte.

- **Endzustand:** Es ist eindeutig festgelegt, unter welchen Bedingungen der Algorithmus stoppt; die vom Algorithmus erzeugten Ausgabedaten sind wohldefiniert.

Über diese sehr allgemein formulierten Anforderungen hinaus stehen zunehmend Effizienzbetrachtungen bei der Beurteilung von Algorithmen im Vordergrund.

2.1.1 Effizienz von Algorithmen

Zur Lösung eines Problems kann es sehr viele verschiedene Algorithmen mit gleichem Ein-Ausgabeverhalten, aber u.U. **sehr** unterschiedlichem Zeit- und Speicherbedarf geben.

Ziel: Möglichst effiziente Algorithmen

Für das "Messen" der Effizienz eines Algorithmus im Hinblick auf die Laufzeit gibt es mehrere Möglichkeiten:

- "Programmieren des Algorithmus" in einer künstlichen Programmiersprache und Zählen und Gewichten der Operationen, die nötig sind, um eine bestimmte Problemgröße zu lösen.
Nachteile: Dies ist eine aufwendige Möglichkeit und eine Übertragbarkeit auf reale Rechner ist nicht unbedingt gegeben.
- Programmieren des Algorithmus auf einem realen Rechner und Messen des Zeitverbrauchs.
Nachteile: Die Resultate sind einerseits rechnerabhängig und hängen andererseits sowohl von den Fähigkeiten der Programmiersprache als auch der Effizienz der Implementation dieser Sprache auf dem entsprechenden Rechner ab; auch spielt die Fertigkeit des Programmierers in der jeweiligen Programmiersprache eine gewisse Rolle.
- Zählen der Operationen auf einem sehr hohen Niveau, z.B. Anzahl der "zu betrachtenden" Elemente beim Suchen in einer Liste oder Anzahl der zu vertauschenden Elementepaare beim Sortieren einer Liste.
Nachteile: Es ist nur eine grobe Abschätzung des tatsächlichen Rechenaufwands möglich, und man erhält nur asymptotische Aussagen.

Als Resultat ergibt sich: Man braucht ein aussagekräftiges Maß für die Effizienz eines Algorithmus.

2.1.2 Komplexität von Algorithmen als Maß für die Effizienz

Bei der Beurteilung der Effizienz von Algorithmen muß man die Komplexität des gestellten Problems mitberücksichtigen, d.h. die Frage lautet: Wie *schwierig* ist das Problem zu lösen, und welche Ressourcen benötigt der zu untersuchende Algorithmus für diese Aufgabe? Da Problemstellungen häufig nur bedingt zu formalisieren sind, wird als abstrakte Beschreibung der Begriff der *berechenbaren Funktion* eingeführt.

Definition 2.1: Berechenbare Funktion

Eine Funktion $f : M \rightarrow N$ heißt berechenbar, wenn es einen Algorithmus gibt, der für jeden Eingabewert $m \in M$, für den $f(m)$ definiert ist, nach endlich vielen Schritten anhält und als Ergebnis $f(m)$ liefert; in allen Fällen, in denen $f(m)$ nicht definiert ist, bricht der Algorithmus nicht ab.

Ein Problem ist lösbar, wenn die zugehörige Funktion berechenbar ist. Die Komplexität einer berechenbaren Funktion ist der zu ihrer Berechnung erforderliche Aufwand an Betriebsmitteln (Speicherplatz, Rechenzeit, benötigte Geräte, usw.), und die Untersuchung des Rechenaufwandes sollte unabhängig von speziellen Rechnern sein.

Ein formales Berechnungsmodell ist z.B. die Turingmaschine, wie sie im Anhang beschrieben ist. Mit Hilfe dieses Modells lassen sich viele theoretische Fragestellungen aus dem Bereich der Berechenbarkeit von Funktionen sowohl übersichtlich als auch elegant untersuchen.

Bei der Analyse von Algorithmen, die derartige Funktionen realisieren, wird sowohl das **Zeitverhalten** als auch der **Speicherplatzbedarf** untersucht, und beides bestimmt die Effizienz einer Berechnung. Die folgenden Überlegungen konzentrieren sich auf die Untersuchung des Zeitverhalten.

Man unterscheidet zunächst zwischen der Zeitkomplexität einer Funktion und der Zeitkomplexität eines Algorithmus.

- Die *Zeitkomplexität einer Funktion* gibt eine untere Schranke für den Aufwand des bestmöglichen Algorithmus aus der Menge aller Algorithmen, die die Funktion berechnen.
- Die *Zeitkomplexität eines Algorithmus* bezeichnet eine obere Schranke für den Rechenaufwand, der im Rechenmodell bei einer konkreten Realisierung des Algorithmus erforderlich ist.

Sei A ein Algorithmus, der die Funktion f berechnet. Dann ist die Komplexität von A eine obere Schranke für die Komplexität von f ; umgekehrt ist die Komplexität von f eine

untere Schranke für die Komplexität von A. Fallen untere und obere Schranke zusammen, so ist der entsprechende Algorithmus asymptotisch optimal für die vorgegebene Funktion und damit für das gestellte Problem.

Die Laufzeit eines Algorithmus (oder auch der Rechenaufwand) ist die Anzahl der Rechenschritte, die bei seiner Durchführung für eine Eingabe gemacht werden. Sei n die Problemgröße und A ein Algorithmus, dann läßt sich der Zeitbedarf eines Algorithmus darstellen als eine Zeitfunktion $T_A(n)$ mit $T_A : \mathbb{N} \rightarrow \mathbb{N}$. Dadurch erhält man eine Abschätzung des asymptotischen Zeitverhaltens, aber keine exakte Angabe der Zeitfunktion.

Definition 2.2:

Sei $f : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion. Dann bezeichnet $O(f)$ die folgende Menge von Funktionen:

$$O(f) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R} \text{ mit } g(n) \leq cf(n) \forall n \geq n_0\}.$$

Bemerkung:

- $O(f)$ ist eine Funktionenklasse, nämlich die Menge der Funktionen, die asymptotisch höchstens so schnell wachsen wie cf .
- $g \in O(f)$ heißt nicht, daß $g(n) \leq f(n)$ gilt.

Definition 2.3:

Ein Algorithmus A , der die Funktion f berechnet, hat die Komplexität $O(f)$, wenn die Zeitfunktion $T_A \in O(f)$ ist.

Bemerkung:

- Falls $f(n) \in O(s(n))$ und $g(n) \in O(r(n))$, dann gilt:
 $f(n) + g(n) \in O(s(n) + r(n)).$
- Falls $f(n) \in O(s(n))$ und $g(n) \in O(r(n))$, dann gilt:
 $f(n) \bullet g(n) \in O(s(n) \bullet r(n)).$

Beispiel 2.1: O-Notation

1. Sei $T_{A1}(n) := \frac{n(n-1)}{2}$; also: $T_{A1}(n) \in O(n^2)$, denn:

$$\frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) \leq n^2 - n \leq n^2$$

(für $n \geq 1$, mit $n_0 = 1$, $c = 1$ erfüllt $T_{A1}(n)$ die Definition).

2. Sei $T_{A2}(n) := n^2 + 2n$; also: $T_{A2}(n) \in O(n^2)$, denn:

$$n^2 + 2n = n^2 + n + n \leq n^2 + n^2 + n^2 = 3n^2$$

(mit $n_0 = 1$, $c = 3$ ist die Definition erfüllt).

3. Sei $T_{A3}(n) \in O(n)$ und $T_{A4}(n) \in O(n \log n)$

Damit gilt: $T_{A3}(n) + T_{A4}(n) \in O(n \log n)$

Typische Zeitschranken sind: $\log n$, n , $n \log n$, n^2 , n^3 , 2^n , n^n .

Es gilt:

$$O(\log n) \subset O(n) \subset O(n \log n) \subset \dots \subset O(n^n),$$

d.h. die rechts stehenden Funktionen wachsen asymptotisch schneller als die links stehenden.

Bemerkung:

- $\log$ bezeichnet der Einfachheit halber im gesamten Text immer den Zweierlogarithmus.

Man unterscheidet den Zeitbedarf im schlechtesten Fall (worst-case), im Mittel (average-case) und im besten Fall (best-case). Man interessiert sich im allgemeinen nicht für eine exakte Angabe der Zeitfunktion, sondern begnügt sich mit einer Abschätzung des asymptotischen Zeitverhaltens nach oben. Falls im Manuskript nicht anders vermerkt, gilt für die Angabe der Zeitkomplexitäten von Algorithmen generell der worst-case-Fall.

Beispiel 2.2: Zeitkomplexität und Zeitverhalten von Algorithmen

Es seien 5 Algorithmen A1, ..., A5 zur Lösung eines Problems der Größe n gegeben (Sortieren von n Zahlen nach ihrer Größe, Suchen eines Elementes in einer Liste mit n Elementen, Matrixmultiplikation o.ä.). Die Laufzeiten der Algorithmen hängen in der folgenden Weise von den Problemgrößen ab:

Algorithmus	A1	A2	A3	A4	A5
Laufzeit(in ms)	$1000\ n$	$200\ n \log n$	$10\ n^2$	$30\ n^3$	2^n

Annahmen:

Es stehe eine Stunde Rechenzeit zur Verfügung. Mit den Algorithmen A1, ..., A5 können dann die folgenden Problemgrößen gelöst werden.

Algorithmus	A1	A2	A3	A4	A5
n für 1h	3600	1500	600	49	21

Zwei Auswege zur Steigerung der maximal möglichen Problemgröße:

1. Schnelleren Rechner kaufen (z.B. Faktor 10 schneller), damit ergeben sich folgende in einer Stunde maximal lösbare Problemgrößen:

Algorithmus	A1	A2	A3	A4	A5
n für 1h mit Faktor 10	36000	13500	1897	106	25

2. Benutzung des jeweils schnellsten Algorithmus für ein vorgegebenes n :

Algorithmus	A1	A2	A3	A4	A5
am schnellsten für	$n \geq 101$	nie	$10 \leq n \leq 100$	nie	$1 \leq n \leq 9$

Folgerung:

Analysen des asymptotischen Zeitverhaltens von Algorithmen (für großes n) sind zur Beurteilung der Effizienz von Algorithmen von zentraler Bedeutung. Bei kleineren Problemgrößen müssen andere Überlegungen hinzukommen.

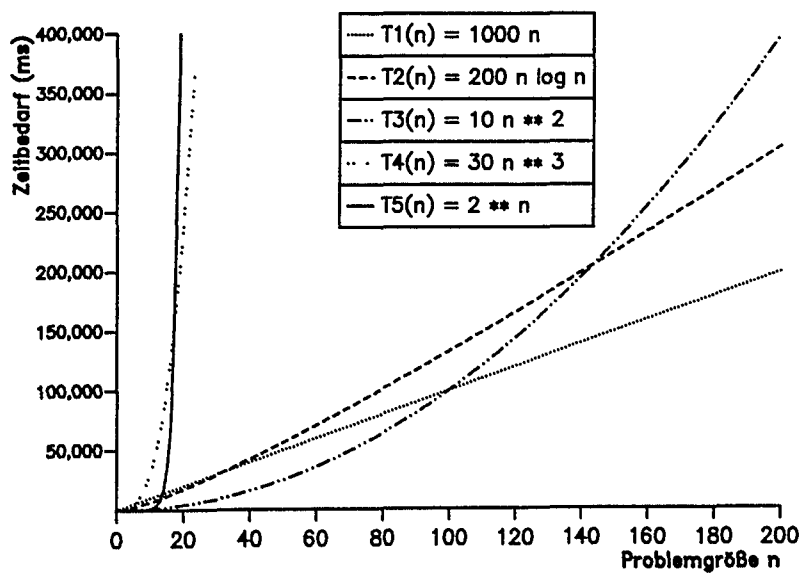
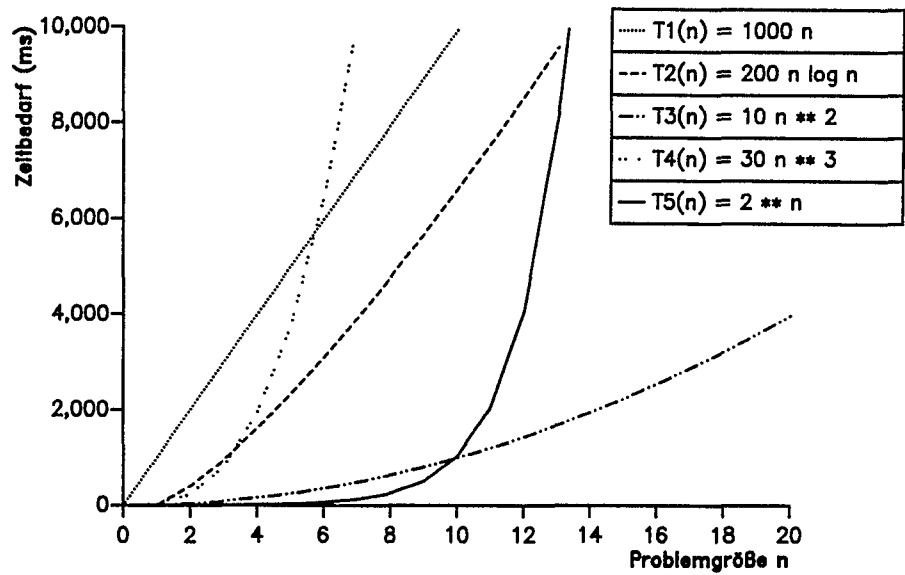


Abb. 1. Zeitverhalten der einzelnen Algorithmen

2.2 Datenstrukturen als Basis für effiziente Algorithmen

Bisher standen Effizienzbetrachtungen für Algorithmen im Vordergrund, im folgenden soll zunächst der Bereich der Datenstrukturen zusammen mit den darauf auszuführenden Operationen betrachtet werden. Gerade bei der Implementation von grundlegenden Datenstrukturen steht die effiziente Implementation der Basisoperationen auf diesen Datenstrukturen im Vordergrund.

2.2.1 Datentypen und Datenstrukturen

Objekte sind Elemente der realen Welt; im Regelfall können sie in ihre Bestandteile zerlegt werden. Bei **atomaren Objekten** geht das nicht, sie sind nicht weiter zerlegbar.

Atomare Datenobjekte sind in der Datenverarbeitung z.B. die Integerzahl '-1', der Character 'A', das Bit '0', etc.

Ein **Datentyp** ist semantisch gleichzusetzen mit einer Menge von Datenobjekten. Dabei wird versucht, Eigenschaften zu definieren, die diese Menge von Datenobjekten von anderen Datenobjekten unterscheidet. Diese Eigenschaften legen dann die Operationen fest, die auf diesem Datentyp zulässig sind.

Definition 2.4: Datentyp

Ein Datentyp ist durch zwei Angaben festgelegt:

1. Eine Menge von Datenobjekten (Werte)
2. Eine Menge von Operationen auf diesen Datenobjekten

In der Programmiersprache definiert der Datentyp einer Variablen die Menge der Werte, die diese Variable annehmen kann. Die Zulässigkeit von Operationen auf diesem Datentyp wird dann leider nur zum Teil automatisch vom Compiler überprüft.

Beispiel 2.3: Der atomare Datentyp INTEGER

- Werte: ..., -2, -1, 0, 1, 2, ...
- Operationen
 1. Addition
 2. Subtraktion
 3. Multiplikation

4. Division
5. Absolutbetrag
6. ...

Datentypen, deren Werte aus Komponenten anderer Datentypen zusammengesetzt sind und deren Elemente durch eine Strukturdefinition in Beziehung stehen, heißen **strukturierte Datentypen** oder **Datenstrukturen**.

Definition 2.5: Datenstruktur

Eine Datenstruktur ist ein Datentyp mit den folgenden Eigenschaften:

1. Sie kann in eine Menge von zusammenhängenden Datenelementen zerlegt werden, wobei jedes dieser Elemente ein atomarer Datentyp oder wieder eine eigene Datenstruktur ist.
2. Sie setzt die Elemente durch eine Menge von Regeln (eine Struktur) in Beziehung.

Die Operationen auf Datenstrukturen können von komplexer Natur sein, d.h. sie können sich auf vollständige Komponenten von Datenstrukturen beziehen.

Beispiel 2.4: Addition auf Komponenten (realisiert z.B. in PL/1, nicht in Pascal)

TYPE Beispiel = ARRAY [1..3] OF REAL;

VAR a,b,c : Beispiel;

...

a := b + c;

...

a		b		c
[1] 7.4		[1] 5.3		[1] 2.1
[2] 2.3	←	[2] 2.2	+	[2] 0.1
[3] 9.5		[3] 3.2		[3] 6.3

Die Klasse der Datentypen zerfällt durch die beiden obigen Definitionen in zwei Teile, die *atomaren Datentypen* und die *Datenstrukturen*.

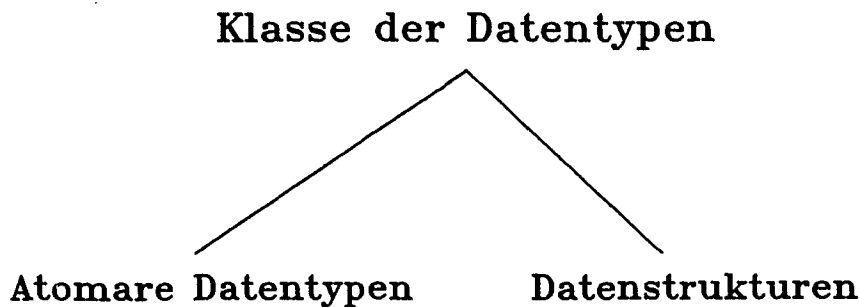


Abb. 2. Zerlegung der Klasse der Datentypen

2.2.2 Abstraktionsniveaus bei Datentypen

Eine weitere Unterscheidung der Klasse der Datentypen ergibt sich, wenn man die verschiedenen Niveaus anschaut, auf denen die Datentypen betrachtet werden können:

- **Abstrakter Datentyp:** Datentyp, der in der Vorstellung des Programmentwicklers entsteht und die wichtigsten Eigenschaften dieses Datentyps angibt, ohne Einzelheiten oder Nebenbedingungen der Realisierung näher zu spezifizieren.
- **Virtueller Datentyp:** Datentyp in einer höheren Programmiersprache.
- **Physikalischer Datentyp:** Datentyp, der auf einem physikalischen Rechner existiert.

Kennzeichen eines **abstrakten Datentyps** ist, daß bei der Konstruktion ('design-phase' bei einem Systementwurf) die spätere Realisierung dieses Datentyps in der Programmiersprache X auf dem Rechner Y **nicht** gesondert berücksichtigt wird. Diese Implementation, d.h. die Speicherverwaltung der Daten und die Bereitstellung der Operationen, findet zu einem späteren Zeitpunkt statt.

Die Realisierung der abstrakten Datentypen wird häufig in höheren Programmiersprachen vorgenommen.

Die Datenstrukturen werden dann durch Komposition von atomaren Datentypen dieser Programmiersprache zusammengesetzt. Dabei übernimmt z.B. Pascal als höhere Programmiersprache die Funktion einer virtuellen Maschine. Die Implementation der Programmiersprache durch den Programmierer beschäftigt sich nur mit der Umsetzung in die Sprache der virtuellen Maschine.

Die Umsetzung der **virtuellen Datentypen** (atomare Datentypen der virtuellen Maschine) in **physikalische Datentypen** (atomare Datentypen der physikalischen Maschine) wird automatisch durch Compiler und Betriebssystem vorgenommen.

Die Umsetzung in die physikalische Realisierung definiert dabei das 3. Abstraktionsniveau bei Datentypen. Es ist leicht einsehbar, daß auch bei sehr komplexen Datenstrukturen die zugrundeliegende Struktur zur Laufzeit des Programms in ein Feld

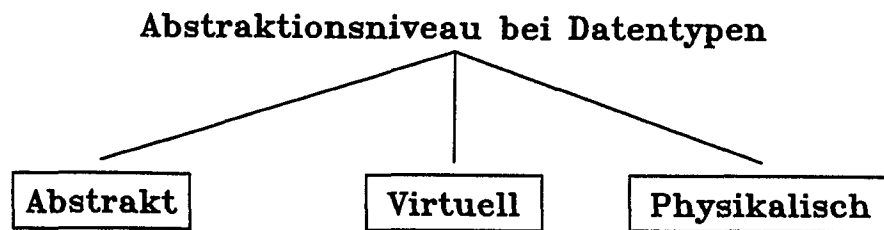


Abb. 3. Abstraktionsniveaus bei Datentypen

von Bytes (Worte), den Speicher und zugehörige Maschinenbefehle abgebildet werden muß.

<i>Datentyp</i>	<i>Abstraktionsniveau</i>		
	abstrakt	virtuell	physikalisch
strukturiert	Liste von Temperaturen	Pascal: REAL	Mehrere Gleitpunktzahlen hintereinander im Speicher
atomar	Anzahl von Leuten	Pascal: INTEGER	Festpunktzahl

Tab. 1. : Beispiele für Abstraktionsniveaus bei verschiedenen Datentypen

Die Realisierung der Spezifikation einer solchen Datenstruktur in einer Programmiersprache setzt sich aus zwei Schritten zusammen (siehe auch Abb. 4).

Repräsentation Die Repräsentation spezifiziert die virtuellen Datentypen.

Implementation Die Implementation legt, basierend auf der Repräsentation, die Anweisungen fest, die zur Lösung des Problems notwendig sind.

2.2.3 Physikalische Speicherung von Daten

Einflüsse der Hardware auf die Datenstruktur resultieren aus den Eigenschaften der Datenspeicherung:

1. *Arbeitsspeicher* (Hauptspeicher)

- a. Speicher mit wahlfreiem Zugriff (random access).
- b. Zugriff auf Daten über Adressen.
- c. Wort- oder Byte-orientierte direkte Adressierung.
- d. Im Vergleich zu externen Speichern relativ kurze Zugriffszeit in der Größenordnung von 100 ns.
- e. Aus Kostengründen und im Vergleich zu externen Speichern relativ geringe Speicherkapazität in der Größenordnung von MBytes.

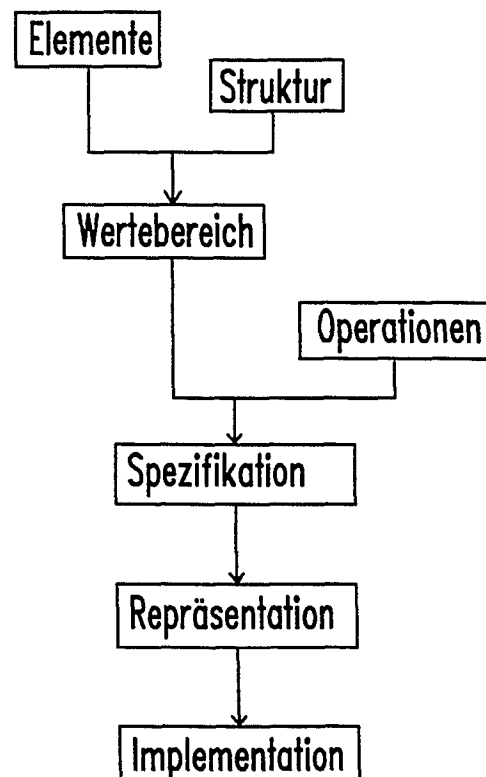


Abb. 4. Spezifikation eines strukturierten abstrakten Datentyps

2. Externe Speicher

a. Magnetbänder, Kassetten:

- 1) Daten werden sequentiell gespeichert; d.h. nur sequentieller Zugriff und in der Reihenfolge der Speicherung.
- 2) Im Vergleich zum Hauptspeicher große Speicherkapazität, für ein einzelnes Band in der Größenordnung von bis zu 1 GByte.
- 3) Langsamer Zugriff, zum Teil manueller Eingriff notwendig.

b. Magnetplatte, Trommel

- 1) Daten sind blockweise gespeichert, Blöcke wahlfrei zugreifbar.
- 2) Große Speicherkapazität (bis zu mehreren GBytes).
- 3) Im Vergleich zum Arbeitsspeicher lange Zugriffszeiten in der Größenordnung von einigen 10 Millisekunden.

Die lange Zugriffszeit zu Daten auf externen Speichern hat einen entscheidenden Einfluß auf die Datenstrukturen, da es bei der Bearbeitung von Daten wesentlich auf eine geringe Anzahl von Externspeicherzugriffen ankommt.

Bei der Behandlung der *Algorithmen und Strukturen* wird in diesem Manuskript (wenn nicht anders vermerkt) als **technische Grundlage** generell ein 'Speicher mit wahlfreiem Zugriff' (also der Hauptspeicher) vorausgesetzt.

2.2.4 Spezifikation von Basisoperationen auf Datenstrukturen

Neben der Darstellung von Algorithmen durch eine verbale Umschreibung, durch eine höhere Programmiersprache (inkl. Pseudocode) oder durch Nassi-Shneiderman-Diagramme o.ä. grafische Darstellungen kann man einen Algorithmus auch durch sog. Pre-Post-Conditions beschreiben. Bei dieser Vorgehensweise wird in **pre-condition** (im folgenden durch *pre* bezeichnet) und **post-condition** (im folgenden durch *post* bezeichnet) unterschieden.

Die *pre-condition* ist eine Bedingung, die erfüllt sein muß, damit die spezifizierte Operation korrekt ausführbar ist. Ist diese Bedingung nicht erfüllt, ist das Ergebnis der Operation undefiniert, was je nach Implementation z.B. zu einer Fehlermeldung, einem unmittelbaren Ende des Programms oder auch zu inkorrekten Werten führen kann. Aus diesem Grund muß **vor der Ausführung der Operationen sichergestellt werden**, daß die *pre-condition* erfüllt ist.

Die *post-condition* beschreibt das System nach Ausführung der Operation. Dabei werden nur die Teile beschrieben, die durch die Operation u.U. manipuliert werden. Es werden keine Angaben gemacht, wie die Zustandstransformation durchgeführt wird, es wird nur beschrieben, was erreicht werden muß.

Mit dieser Spezifikationsmethode wird im folgenden Beispiel ein abstrakter Datentyp *Farbe* definiert, dessen Werte Farben sind. Der Wertebereich enthält die Primärfarben (rot, gelb, blau) und die Farben, die man durch Mischen erhält (grün, orange, violett). Die Operationen sollen nur für die Veranschaulichung dieser Spezifikation dienen.

Beispiel 2.5: Abstrakter atomarer Datentyp Farbe

1. Wertebereich

{ rot, gelb, blau, grün, orange, violett }

2. Operationen

a. *mix* (*c1*, *c2* : *Farbe*) : *Farbe*

pre *c1* und *c2* sind Primärfarben

post *mix* ist die Farbe, die durch Mischen von *c1* und *c2* entsteht

b. *primary* (*c* : *Farbe*) : *BOOLEAN*

post

IF *c* ist Primärfarbe
THEN *primary* ist true
ELSE *primary* ist false

c. *form* (*c* : *Farbe*; var *c1*, *c2* : *Farbe*)

pre *c* ist keine Primärfarbe

post *c1* und *c2* sind die beiden Primärfarben, aus denen *c* besteht

d. *assign* (*c1* : *Farbe*; var *c2* : *Farbe*)

post *c2* hat den Wert *c1*

3. Datenstrukturen

Eine der wichtigen Eigenschaften bei der Spezifikation von Datenstrukturen ist die Beschreibung der Struktur zwischen den Elementen. Es gibt vier wesentliche Struktureigenschaften für Datenstrukturen (siehe auch Abb. 5), die in diesem Manuskript behandelt werden. Eine Struktur, bei der die Elemente nur insoweit in Beziehung stehen, als sie zu einer Menge gehören, kann durch den mathematischen Begriff der Menge (Set) beschrieben werden. Falls die Elemente jeweils zu einem anderen in Beziehung stehen (one-to-one Relation), handelt es sich um eine lineare Struktur. Falls ein Element zu mehreren anderen Elementen in Beziehung steht (one-to-many Relation), handelt es sich um eine hierarchische Struktur oder auch Baumstruktur (tree). Strukturen, bei denen viele Elemente mit vielen anderen Elementen in Beziehung stehen (many-to-many Relation), werden Graph- oder Netzwerkstrukturen genannt. Diese Begriffe werden in den folgenden Abschnitten formal definiert und eingeführt.

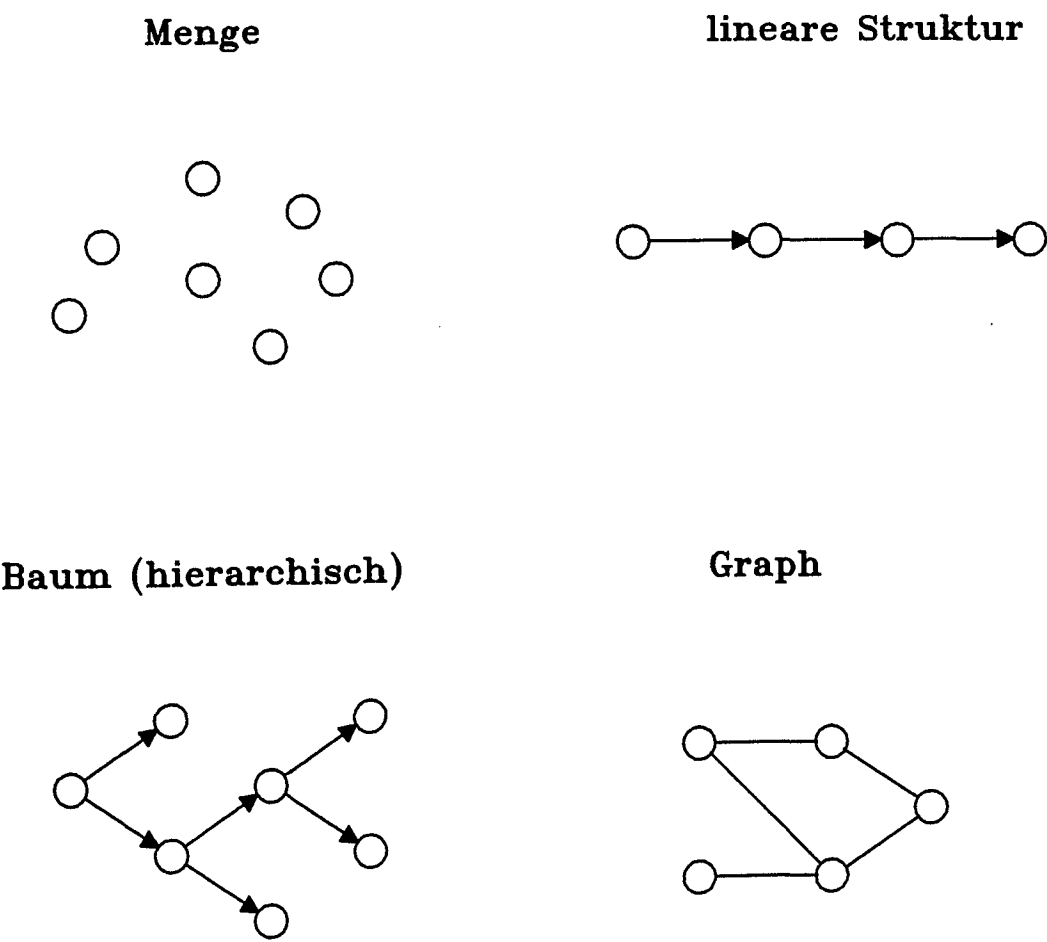


Abb. 5. Elementare Strukturrelationen

3.1 Einfache Datenstrukturen

In diesem Abschnitt werden die fundamentalen Datenstrukturen vorgestellt, die später die Grundlage für die meisten komplexen Datenstrukturen und die darauf aufbauenden Algorithmen bilden. Der atomare Datentyp Pointer nimmt dabei eine Sonderstellung ein, da er aufgrund der fehlenden Struktur nicht direkt zu den Datenstrukturen gezählt werden kann. Pointer geben jedoch Adressen von Datenstrukturen an, so daß die Vorstellung in diesem Abschnitt notwendig ist.

Definition 3.1: Homogene und heterogene Datenstrukturen

In einer homogenen Datenstruktur haben alle Komponenten den gleichen Datentyp. In einer heterogenen Datenstruktur haben die Komponenten unterschiedliche Datentypen.

Definition 3.2: Schlüsselmenge

Eine Schlüsselmenge ist eine linear geordnete endliche Menge $S = \{s_1, \dots, s_n\}$.

Bemerkung:

- Schlüssel dienen der eindeutigen *Identifikation* von Daten und werden i.allg. zusammen mit den Daten abgespeichert oder sind Bestandteil der Daten.
- Schlüssel *müssen nicht* Zahlen sein. Auch Namen können z.B. als Schlüssel verwendet werden (lexikographische Sortierung).

Beispiel 3.1: Beispiel für (hoffentlich eindeutige) Schlüssel

- Kontonummer
- Nummer eines Versicherungsvertrages
- Personalnummer
- *Postleitzahlen*

Bei Postleitzahlen dient der Schlüssel nicht nur der eindeutigen Identifikation, sondern auch der Klassifikation (schnelleres Suchen, Sortieren).

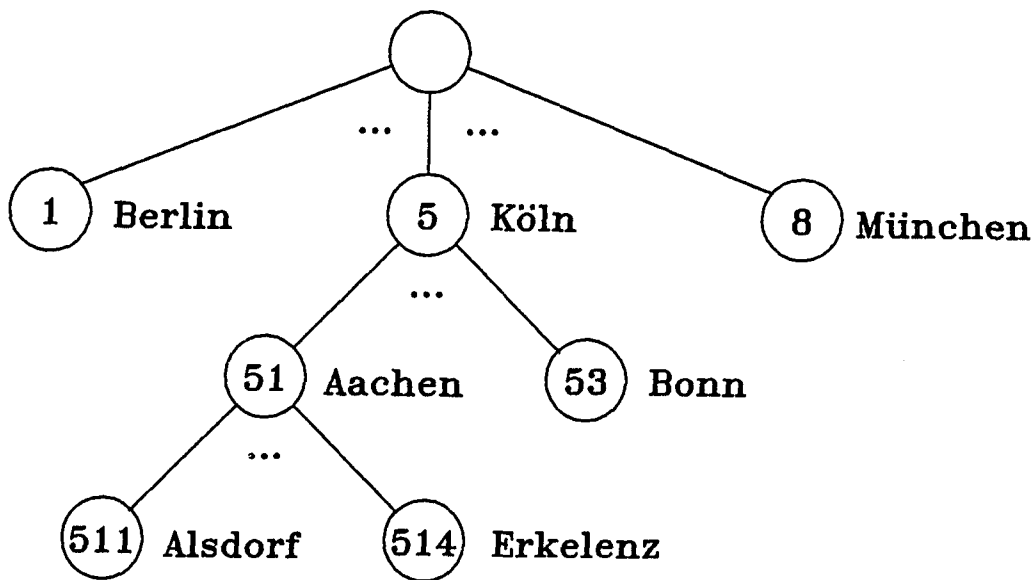


Abb. 6. Beispiel für Postleitzahlen

3.1.1 Felder (arrays)

Bei den Feldern handelt es sich um die gebräuchlichste Datenstruktur, die in den meisten Programmiersprachen (z.B. FORTRAN, Pascal, PL/1, ...) verwendet werden kann. Die Elemente eines Feldes können entweder atomare Datentypen oder wieder Datenstrukturen sein. Allerdings haben alle Elemente eines Feldes den gleichen Typ. Ein Feldelement wird durch die Spezifikation eines Wertes des Indexbereiches angesprochen. Dieser Wert kann eine Konstante, eine Variable oder ein Ausdruck sein und wird u. U. erst zur Laufzeit festgelegt.

Definition 3.3: Abstrakter Datentyp *Feld*

Ein Feld ist eine Kette von Elementen.

1. Elemente

Durch Festlegung eines Komponententyps ist der Typ des Elementes festgelegt. Da alle Elemente den gleichen Typ haben, ist das Feld eine homogene Datenstruktur.

2. Struktur

Ein Indextyp, der linear sein muß, legt den Wertebereich der Indexwerte fest. Die Indexwerte stehen in einer 1-1 Relation zu den Komponentenwerten, jeder Indexwert identifiziert genau einen Komponentenwert.

3. Operationen

a. *retrieve* ($S, C, i_1, \dots, i_n$)

gibt der Variablen C den Wert von S , der mit dem Index $i_1, \dots, i_n$ ($n \geq 1$) korrespondiert.

in Pascal: $C := S[i_1, \dots, i_n];$

b. *update* ($S, C, i_1, \dots, i_n$)

speichert den Wert der Variablen C auf die Komponente von S , die mit dem Index $i_1, \dots, i_n$ ($n \geq 1$) korrespondiert.

in Pascal: $S[i_1, \dots, i_n] := C;$

Mögliche Fehlerquellen bei Operationen auf dem Datentyp Feld:

- *retrieve*:
 - Index nicht vom zulässigen Typ.
 - Undefinierte Werte in der Feldkomponente.
 - Komponententyp und Typ von C stimmen nicht überein.
- *update*
 - Index nicht vom zulässigen Typ.
 - Undefinierter Wert in der Zuweisungsvariablen.
 - Komponententyp und Typ von C stimmen nicht überein.
 - Index liegt außerhalb des zugehörigen Typs, eine Zuweisung auf Feldelemente außerhalb der zulässigen Grenzen wird z.B. in FORTRAN im Regelfall nicht überprüft.

Die Speicherplatzzuordnung ist durch eine injektive, d.h. ein-eindeutige Abbildung

$$P: \text{Feldindex} \rightarrow \text{Speicherzelle}$$

gegeben (sogenanntes **Adreßpolynom**).

Beispiel 3.2: Speichern von Feldern

1. Eindimensionales Feld:

- $A: \text{ARRAY}[1..5] \text{ OF INTEGER}$ (jedes Element 4 Byte lang);
- Gebe *element_length*(A) die Länge eines Elementes von A als Wert zurück

- Enthalte A die Werte 16,7,5,8,10

...	16	7	5	8	10	...
-----	----	---	---	---	----	-----

- Sei $addr(A)$ die Anfangsadresse des Feldes A
- Unter den oben angegebenen Randbedingungen korrespondiert $A[3]$ mit dem Wert 5 und der Adresse $addr(A) + (2 \bullet element_length(A))$
Das bedeutet: $P: i \rightarrow addr(A) + ((i-1) \bullet element_length(A))$

2. Zweidimensionales Feld:

- B: ARRAY [1..3,1..2] OF INTEGER (jedes Element 4 Byte lang);
- Enthalte B die folgenden Werte

B(i,j)		j	
		1	2
i	1	16	7
	2	5	8
	3	10	10

- Es gibt zwei typische Methoden, die Elemente von B in aufeinanderfolgenden Speicherzellen abzulegen:

a. Spaltenweise Speicherung (FORTRAN)

...	B(1,1)	B(2,1)	B(3,1)	B(1,2)	B(2,2)	B(3,2)	...
-----	--------	--------	--------	--------	--------	--------	-----

Adreßberechnung bei spaltenweiser Speicherung:

Das zweidimensionale Feld B sei dimensioniert als $B[i_1, i_2]$ (i_1 Zeilen, i_2 Spalten).

Die Adresse ergibt sich aus den Feldindizes i ($1 \leq i \leq i_1$) und j ($1 \leq j \leq i_2$) gemäß der folgenden Abbildung P:

$$P: i, j \rightarrow addr(B) + (((j-1) \bullet i_1 + i - 1) \bullet element_length(B)) = addr(B(i, j))$$

z.B.

$$\begin{aligned} P: 3, 1 &\rightarrow addr(B) + ((0 \bullet 3 + 3 - 1) \bullet element_length(B)) \\ &= addr(B) + 2 \bullet element_length(B) = addr(B(3, 1)) \end{aligned}$$

b. Zeilenweise Speicherung (Pascal, PL/1)

...	B(1,1)	B(1,2)	B(2,1)	B(2,2)	B(3,1)	B(3,2)	...
-----	--------	--------	--------	--------	--------	--------	-----

3.1.2 Sätze (records)

Eine wichtige Eigenschaft von Records ist ihre Fähigkeit, unterschiedliche Datenelemente mit verschiedenen Datentypen zu einer einzigen Einheit zu gruppieren. Die Information kann dann

1. als Einheit betrachtet werden (z.B. bei Zuweisungen, Speicherung, etc.);
2. einzeln betrachtet werden (z.B. bei Zuweisung, Speicherung, Abfragen einzelner Elemente eines Records, etc.).

Die Datentypen der Elemente brauchen nicht atomar zu sein, Datenstrukturen sind zulässig. Jede dieser Record-Komponenten kann nur über eine Komponentenidentifikation angesprochen werden. Diese Komponentenidentifikation muß schon zur Übersetzungszeit festgelegt werden und kann nicht, wie bei Feldern, durch spezielle Indizes zur Laufzeit durch Auswerten von Ausdrücken berechnet werden.

Definition 3.4: Abstrakter Datentyp *Record*

Ein Record ist eine Gruppierung von unterschiedlichen Datenelementen zu einer Einheit.

1. Elemente

Jede Komponente benötigt die Angabe eines eigenen Datentyps. Da die Komponenten unterschiedliche Datentypen haben können, ist der Record eine heterogene Datenstruktur.

2. Struktur

Eine Liste von Namen (identifiziert) legt in einer direkten Abbildung zu den Komponenten die Struktur des Records fest.

3. Operationen

a. *retrieve* (R, V, id)

gibt der Variablen V den Wert der Komponente von R , die mit id spezifiziert ist. Gleiche Typen von V und der entsprechenden Record-Komponente von R werden hier vorausgesetzt.

in Pascal: $V := R.id;$

b. *update* (R, V, id)

speichert den Wert der Variablen V auf die Komponente von R , die durch id bezeichnet ist.

in Pascal: $R.id := V;$

Besonderheiten:

1. Wenn R1 und R2 Records vom gleichen Typ sind, kann durch $R1 := R2$ der Wert von R2 vollständig auf R1 zugewiesen werden.
2. Die Namen der Komponenten (z.B. id) müssen eindeutig zuzuordnen sein. In verschiedenen Records dürfen jedoch gleiche Namen als Komponentenidentifikation verwendet werden.

Einschränkung: Der voll qualifizierte Name (z.B. R.id) muß natürlich eindeutig sein.

3. Pascal: WITH-Statement

Mit dem WITH-Statement können die Pfadangaben spezifiziert und danach für diesen Pfad weggelassen werden.

Beispiel 3.3: Pascal Programm-Skelett mit Record-Typvereinbarungen

```
TYPE    Hersteller = (BMW,Ford,VW);
        Farbe      = (Blau,Grün,Rot);
        Kfz        = RECORD
                        Marke      : Hersteller;
                        Lackierung : Farbe;
                    END; (* Kfz *)

VAR      Auto1,Auto2      : Kfz;
        Bestelltes__Auto  : Kfz;

FUNCTION Gleiches__Auto (Auto1,Auto2 : Kfz) : BOOLEAN;
BEGIN
    ...
END;

(* im Programmtext des Hauptprogramms: *)
a)      Bestelltes__Auto.Marke := Ford;
        Bestelltes__Auto.Lackierung := Grün;
b)      Auto1.Marke := Ford;
        Auto1.Lackierung := Blau;
c)      Auto2 := Auto1;
        Auto2.Lackierung := Rot;
```

```

d)  IF (Gleiches_Auto(Bestelltes_Auto,Auto1))
    THEN BEGIN
        (* Auslieferung von Auto1 *)
    ELSE BEGIN
        IF (Gleiches_Auto(Bestelltes_Auto,Auto2))
        THEN BEGIN
            ...
        END
        ELSE BEGIN
            (* Produktion von Bestelltes_Auto *)
        END
    END;

```

Beispiel 3.4: Pascal Programm-Skelett für die Datenstruktur 'Mitarbeiter'

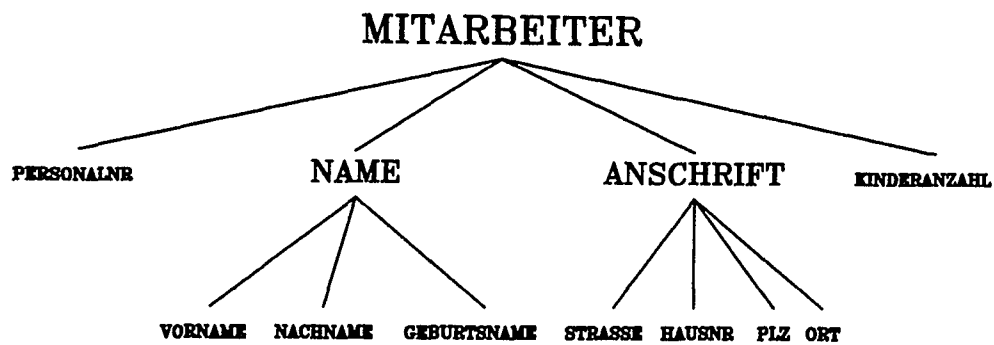
```

TYPE char_array = PACKED ARRAY [1..10] OF CHAR;
Mitarbeiter = RECORD
    Personalnr : INTEGER;
    Name       : RECORD
        Vorname   : char_array;
        Nachname  : char_array;
        Geburtsname : char_array;
    END; (* Name *)
    Anschrift  : RECORD
        Strasse    : char_array;
        Hausnr     : INTEGER;
        Plz        : 1000..9999;
        Ort        : char_array;
    END; (* Anschrift *)
    Kinderanzahl : 0..25;
END; (* Mitarbeiter *)

VAR    Einzustellende_Person Mitarbeiter;
        ... (* im Programmtext: *)
Einzustellende_Person.Name.Vorname := 'Birgit';
Einzustellende_Person.Name.Nachname := 'Müller';

```

a) Strukturbaum



b) Speicherdarstellung

PERSONALNR	VORNAME	NACHNAME	GEBURTSNAME	STRASSE	HAUSNR	PLZ	ORT	KINDERANZAHL
------------	---------	----------	-------------	---------	--------	-----	-----	--------------

Abb. 7. Strukturbaum und Speicherdarstellung der Datenstruktur Mitarbeiter

3.1.3 Zeiger (pointer)

Problemlösungen, die nur die bisher bekannten Datentypen verwenden, können nur auf Probleme angewandt werden, bei denen die Problemgröße statisch zur Übersetzungszeit bereits bekannt ist. Eine dynamische Anpassung an die aktuelle Problemgröße ist z.B. mit Hilfe des Datentyps Pointer möglich.

Definition 3.5: Abstrakter Datentyp *Pointer*

Ein Pointer ist ein Datentyp, dessen Werte die Adressen von Repräsentationen anderer Datentypen sind.

1. Elemente

Als Elemente sind die Werte aller Speicheradressen einschließlich des Wertes NIL (undefiniert) zulässig.

2. Struktur

keine

3. Operationen (in der Programmiersprache Pascal)

- Assignment $':='$
- Relational $'='$ und $'<>'$
- Dynamic $'New'$ und $'Dispose'$

Beispiel 3.5: Vorstellung der Pointer

```
TYPE Ptr      = ↑INTEGER;1
VAR P,Q       : Ptr;
    X         : INTEGER;
```

Die Variablen P und Q sind Integer-Pointer, weil sie auf eine Speicherstelle im Speicher zeigen können, die eine Integerzahl enthält; X ist z.B. eine Speicherstelle, die eine Integerzahl enthält.

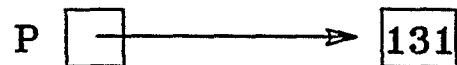
Mit der Anweisung VAR P : Ptr; wird eine Speicherstelle angelegt, die in der Lage ist, eine Referenz (Adresse) zu einer anderen Speicherstelle aufzunehmen, in der eine Integerzahl abgelegt werden kann.

Speicheradressen	32600	32604	32606	32607	...	32616	32620
Speicherinhalt	32615	'x'	50			131	



Zuweisung:

$P^{\uparrow} := 131;$



Unzulässig ist jedoch in Pascal die Zuweisung:

$P := 131;$

da dann die Pointer-Variable P mit einem Integerwert belegt würde. Dieser Fehler wird wegen der strengen Typprüfung in der Programmiersprache Pascal erkannt!

Dynamische Basisoperationen in Pascal:

1. **New(P)** ist eine Standardprozedur. Der Aufruf von new(P) stellt eine Speicherstelle (in diesem Beispiel vom Typ Integer) bereit, und P wird als Referenz auf diese Speicherstelle zurückgegeben.
2. **Dispose(P)** ist ebenfalls eine Standardprozedur. Der Aufruf von dispose(P) gibt die Speicherstelle, die durch P bezeichnet wird wieder frei. Die Referenz P bleibt unverändert und zeigt auf einen Speicherplatz, der vom Laufzeitsystem unter Umständen

¹ Das Zeichen @ entspricht auf IBM-Terminals dem Zeichen ↑.

```

program POINTER (input,output);
type PTR = ↑integer;
var P, Q : PTR;
begin

    new(P);

    P↑ := 1;

    new(Q);

    Q↑ := 2;

    P↑ := Q↑ + 3;

    P := Q;

    P↑ := 7;

    new(P);

    Q := NIL;

    dispose(P);

    P := Nil;

end.

```

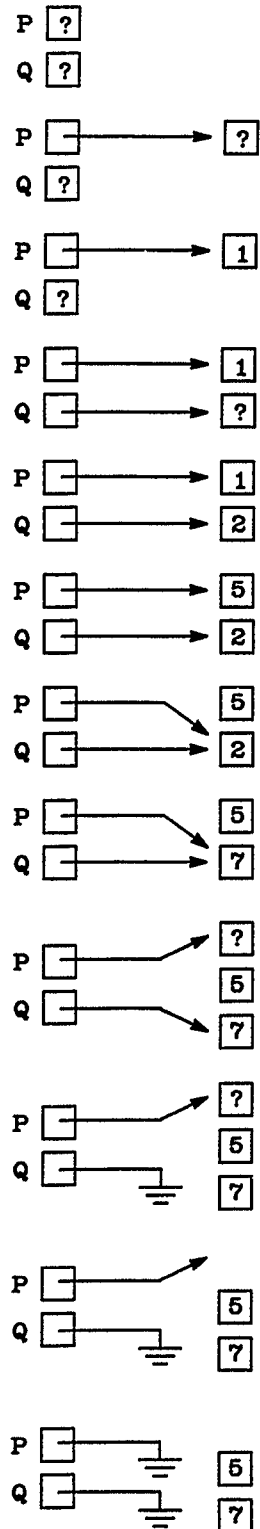


Abb. 8. Beispielprogramm mit Pointern

wieder (in anderem Kontext) vergeben wird. Nach einer Freigabe von P sollte der Pointer wieder auf Nil gesetzt werden.

Besonderheiten

- In der Programmiersprache Pascal sind Pointer streng typisiert, d.h. der Pointer P hängt streng am Datentyp der Speicherstelle, auf die P zeigt.
- $R : \uparrow\text{CHAR};$ und $P := R$ sind nicht zulässig.

3.1.4 Mengen (sets)

Bei einer Menge handelt es sich um einen Datentyp, bei dem es keine Reihenfolge der Elemente gibt und die Elemente keine Beziehungen untereinander (Vorgänger, Nachfolger, Vater, Sohn) besitzen. Es gibt kein erstes oder letztes Element. Ein Element ist entweder in der Menge enthalten oder nicht enthalten. Die Elemente sind atomar und durch Aufzählung angebbbar.

Bemerkung:

Der Pascal-Typ REAL ist atomar, aber nicht aufzählbar. Er ist daher für Sets nicht erlaubt.

Definition 3.6: Abstrakter Datentyp *Set*

Ein Set ist eine Sammlung von Elementen, die alle denselben Datentyp haben.

1. Elemente

Die Elemente sind von einem Aufzählungstyp und atomar.

2. Struktur

Die Elemente sind Mitglieder einer Menge.

3. Operationen

a. *assign* ($S1 : \text{set}; \text{var } S2 : \text{set}$)

post S2 hat die gleichen Elemente wie S1.

in Pascal: $S2 := S1;$

b. *in* (*S1* : *set*; *e* : *element*) : *BOOLEAN*

post

IF *e* Mitglied von *S1*
THEN *in* ist true
ELSE *in* ist false

in Pascal: *in* := (*e* IN *S1*);

c. *intersection* (*S1,S2* : *set*; *var S3* : *set*)

post *S3* enthält den Durchschnitt von *S1* und *S2*.

in Pascal: *S3* := *S1* * *S2*;

d. *union* (*S1,S2* : *set*; *var S3* : *set*)

post *S3* enthält die Vereinigung von *S1* und *S2*.

in Pascal: *S3* := *S1* + *S2*;

e. *difference* (*S1,S2* : *set*; *var S3* : *set*).

post *S3* enthält *S1* ohne *S2*.

in Pascal: *S3* := *S1* - *S2*;

f. *equal* (*S1,S2* : *set*) : *BOOLEAN*

post

IF *S1* und *S2* haben die gleichen Elemente
THEN *equal* ist true
ELSE *equal* ist false

in Pascal: *equal* := (*S1* = *S2*);

g. *subset* (*S1,S2* : *set*) : *BOOLEAN*

post

IF jedes Element von *S1* auch in *S2*
THEN *subset* ist true
ELSE *subset* ist false

in Pascal: *subset* := ((*S1* * *S2*) = *S1*);

h. *mkset* (*e* : *element*; *var S1* : *set*)

post *S1* enthält als einziges Element *e*.

in Pascal: *S1* := [*e*];

i. *create* (*var S1* : *set*)

post *S1* existiert und ist leer.

in Pascal: *S1* := [];

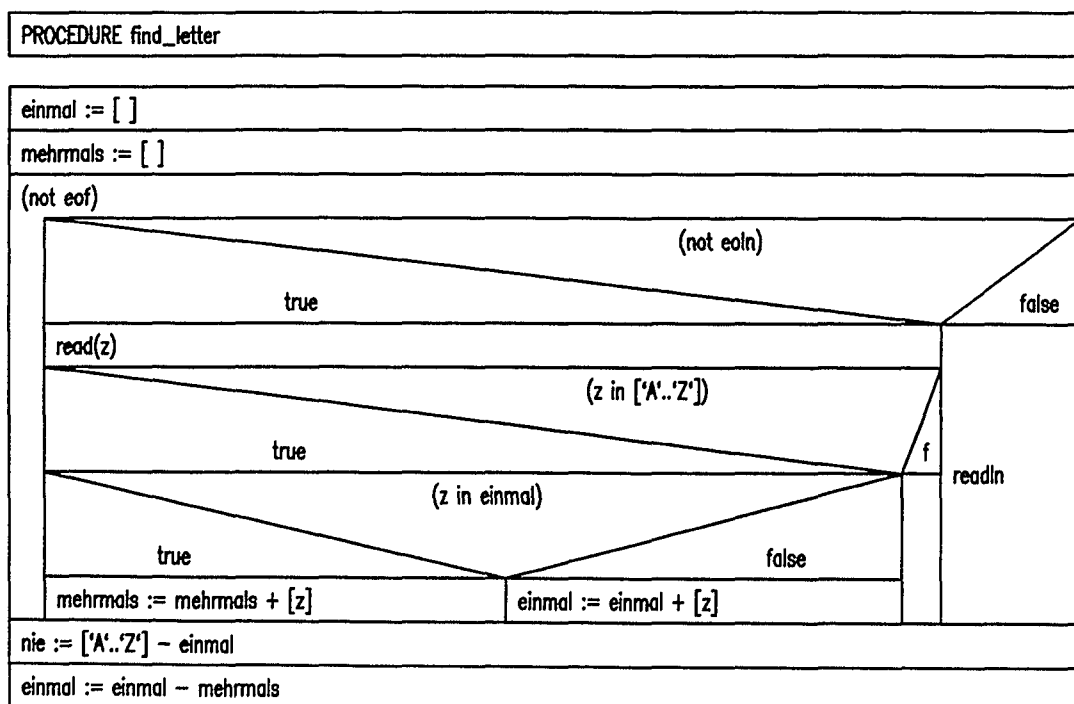
Beispiel 3.6: Dieses Programm bestimmt in einem Text die Buchstaben, die nie, einmal oder mehrmals vorkommen

1. in Pascal unter Nutzung von Sets

```

TYPE  Buchstaben = SET OF 'A'..'Z';
VAR   nie, einmal, mehrmals : Buchstaben;
       z : CHAR;

```

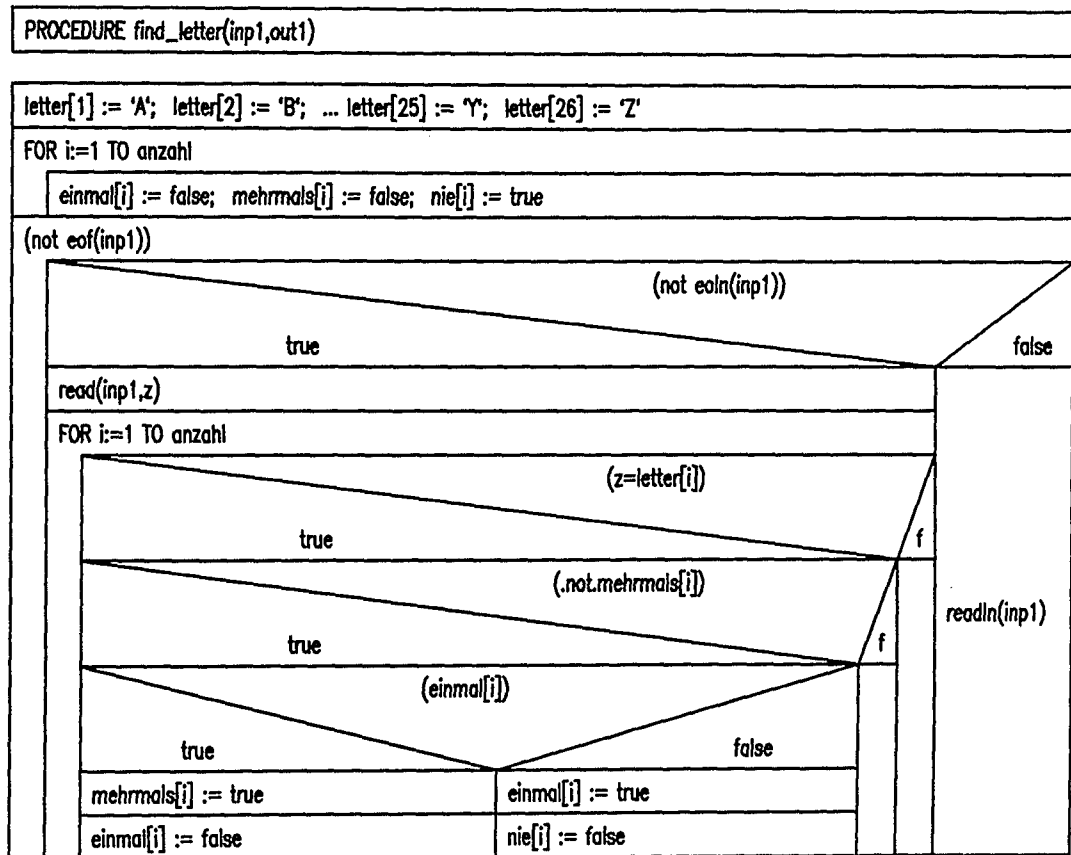


2. in Pascal als Array-Implementierung

```

CONST anzahl = 26;
VAR   nie, einmal, mehrmals : ARRAY [1..anzahl] OF BOOLEAN;
      letter                 : ARRAY [1..anzahl] OF CHAR;
      z                     : CHAR;
      i                     : INTEGER;

```



3.1.5 Sequenzen (sequences)

Im folgenden werden die Datenkomponenten der Datenstrukturen grundsätzlich mit *stdelement* (standard-element) bezeichnet.

```

TYPE stdelement = RECORD
    key : keytype; (* eindeutige Identifizierung *)
    data : datatype; (* weitere Information über das Element *)
END; (* stdelement *)

```

Ein Datentyp mit linearer Struktur ist die Sequenz, eine endliche Folge von Objekten eines gegebenen Datentyps. Im Unterschied zu Mengen ist auf Sequenzen eine Ordnung definiert, es gibt also ein erstes, zweites, ... Element und zu jedem Element (außer dem ersten bzw. letzten) einen Vorgänger und Nachfolger.

Definition 3.7: Sequenz

Eine Sequenz ist eine Folge aus keinem oder aus endlich vielen Elementen eines gegebenen Datentyps, wobei auf die Elemente nur sequentiell zugegriffen werden kann.

1. Elemente

Die Elemente sind vom Typ `stdelement`.

2. Struktur

Die Struktur zwischen den Elementen ist linear.

3. Operationen

Die Operation `create` muß vor jeder anderen Operation ausgeführt werden. Alle Operationen (bis auf `create`) haben als zusätzliche Pre-condition, daß die Sequenz existiert.

a. *find_first*

pre	Die Sequenz ist nicht leer.
post	Das erste Element ist das <code>current_element</code> .

b. *find_next*

pre	Die Sequenz ist nicht leer und <code>current_element</code> ist nicht das letzte Element.
post	Der Nachfolger des <code>current_element</code> ist das neue <code>current_element</code> .

c. *append (e : stdelement)*

post	Die Sequenz enthält <code>e</code> als letztes Element, und dieses Element ist das <code>current_element</code> .
------	---

d. *retrieve* (*var e : stdelement*)

pre Die Sequenz ist nicht leer.
post e enthält das *current_element*.

e. *empty* : *boolean*

post
 IF Sequenz enthält kein Element
 THEN *empty* ist true
 ELSE *empty* ist false

f. *last* : *boolean*

post
 IF letztes Element ist *current_element*
 THEN *last* ist true
 ELSE *last* ist false

g. *create*

post Die Sequenz existiert und ist leer.

h. *clear*

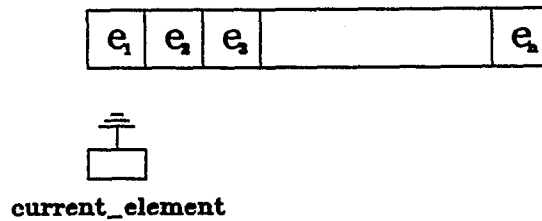
post Die Sequenz ist leer.

Bemerkung:

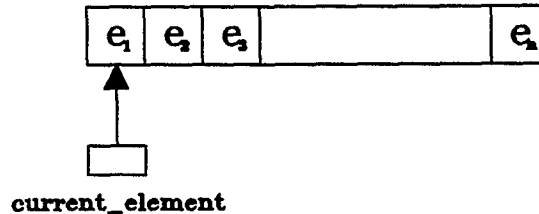
Da in einem Programm unter Umständen nicht nur eine Sequenz realisiert werden soll, ist es sinnvoll, bei einer Implementation der oben beschriebenen Operationen zusätzlich eine Variable an jede Funktion oder Prozedur zu übergeben, die die zu bearbeitende Sequenz eindeutig festlegt (siehe z.B. unten bei der Realisierung von sequentiellen Dateien in Pascal).

Veranschaulichung einiger Operationen:

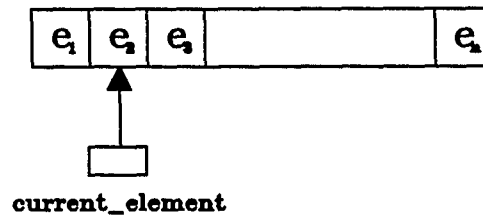
Die Elemente $e_1, \dots, e_n$
bilden eine Sequenz



nach `find_first`

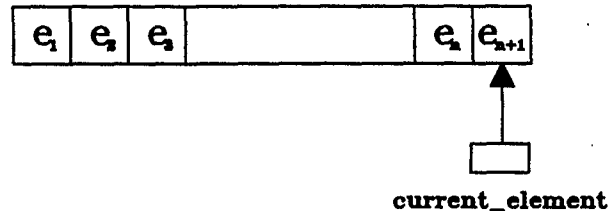


nach `find_next`



...

nach `append(  $e_{n+1}$  )`



Sequentielle Dateien in Pascal

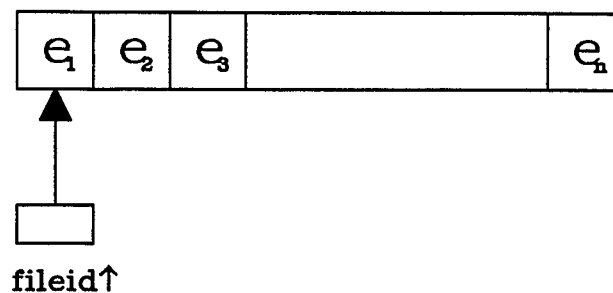
In einigen Programmiersprachen ist die Sequenz als Datentyp enthalten. Pascal-Dateien sind eine Form der Sequenz mit einigen der oben genannten Operationen. In der Praxis tritt häufig der Fall ein, daß der Programmierer größere Datenmengen zu bearbeiten und zu verwalten hat und sich diese Daten aus Speicherplatzgründen beispielsweise nicht mehr als Elemente eines Arrays darstellen lassen. Desweiteren besteht das Problem, Daten über die Dauer des Programmablaufs zu erhalten und sie in geeigneter Weise abzuliegen, um später bei Bedarf wieder auf sie zuzugreifen. Aus diesem Grund erlaubt Pascal die Definition einer *sequentiellen Datei*, dessen Komponentenzahl sich je nach Bedarf während des Programmablaufs dynamisch verändern läßt.

Sei VAR fileid : FILE OF stdelement;

Die Übergabe der Dateivariablen *fileid* ermöglicht die Anwendung der Operationen auf unterschiedliche Dateien. Weiterhin gibt es eine Boolesche Funktion EOF(*fileid*), die einen Test auf der Datei-Variablen *fileid* durchführt; die Operationen sind wie folgt definiert:

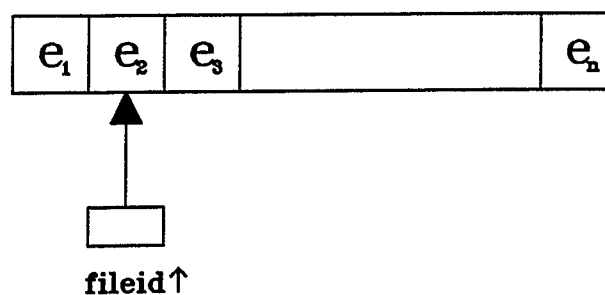
- RESET(*fileid*) (entspricht *find_first*)

Wenn die Datei *fileid* nicht leer ist, also mindestens ein Element enthält, bekommt *fileid*↑ den Wert des ersten Elementes und EOF(*fileid*) liefert den Wert false, andernfalls bleibt *fileid*↑ undefiniert, und EOF(*fileid*) liefert true.
(*fileid*↑ ist das *current_element*).



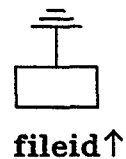
- GET(*fileid*) (entspricht *find_next*)

Die Variable *fileid*↑ bekommt den Wert der nächsten Komponente. Falls diese nicht existiert, liefert EOF(*fileid*) den Wert true, und *fileid*↑ bleibt undefiniert. GET(*fileid*) ist nur definiert, falls vor der Ausführung EOF(*fileid*) false geliefert hat.



- REWRITE(*fileid*) (entspricht *create*)

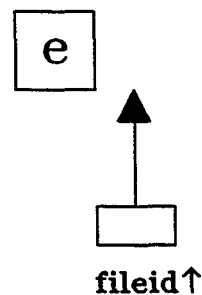
Der Inhalt der Datei *fileid* wird gelöscht, enthält also nach dem Aufruf keine Elemente mehr. EOF(*fileid*) liefert den Wert true, und *fileid*↑ bleibt undefiniert. Diese Operation initialisiert also den Prozeß des Erstellens einer neuen Sequenz.



- PUT(fileid) (entspricht *append(e)*)

Der Wert, den die Variable **fileid↑** besitzt, wird an die Datei *fileid* angehängt. Die Prozedur ist nur dann definiert, wenn vor ihrer Ausführung EOF(fileid) true geliefert hat. Nach Ausführung liefert EOF(fileid) immer noch true, und **fileid↑** ist undefiniert.

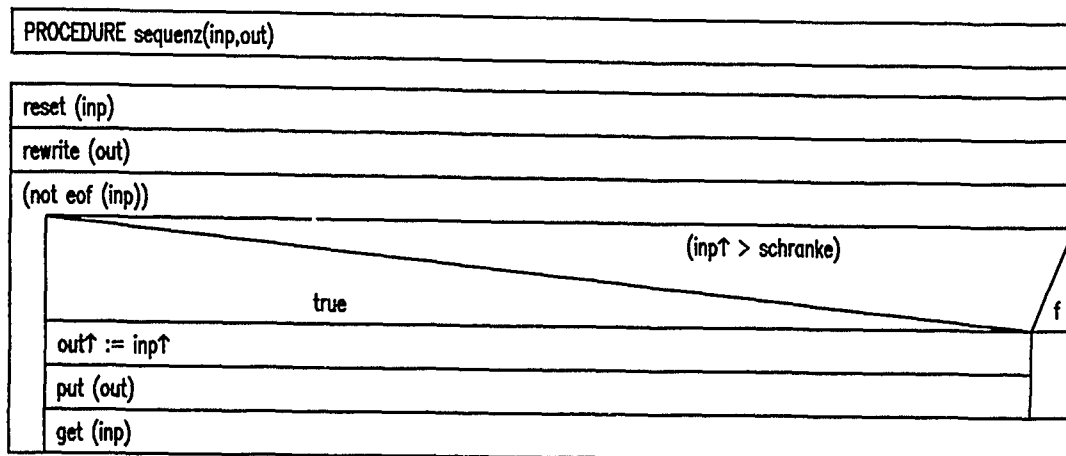
```
fileid↑ := e;
PUT(fileid)
```



Beispiel 3.7: Sequenz in Pascal

Auf einer Datei *inp* stehen nur ganzzahlige Werte. Die Datei *inp* soll gelesen werden und alle Zahlen, die eine vorgegebene Schranke übertreffen, sollen auf die Datei *out* geschrieben werden.

```
CONST    schranke = 0;
VAR      inp, out : FILE OF INTEGER;
```



Im Prinzip ist es möglich, alle Datei-Operationen durch die vier Grundoperationen auszudrücken. In der Praxis hat es sich jedoch als nützlich erwiesen, die Operation des Weitersetzens der Datei-Position (GET und PUT) mit dem Zugriff zu Puffervariablen zu verknüpfen. So gibt es zwei weitere Prozeduren, die sich durch die Grundoperationen beschreiben lassen.

Vereinbarung: Sei *special* eine Datei vom Typ *stdelement*, *y* eine Variable und *e* ein Ausdruck vom Typ *stdelement*.

Zusätzliche Operationen:

- READ(special,y): $y := \text{special}\uparrow$;
 GET(special);

- WRITE(special,e): $\text{special}\uparrow := e$;
 PUT(special);

Spezialfall: Dateien vom Typ Text

Dateien mit Komponenten vom Typ CHAR spielen eine besonders wichtige Rolle in der Datenverarbeitung. Diesem Datentyp, der in Pascal vordefiniert ist, wird daher der Standardname *text* gegeben.

```

TYPE    text  = FILE OF CHAR;
VAR     fileid :  text;
```

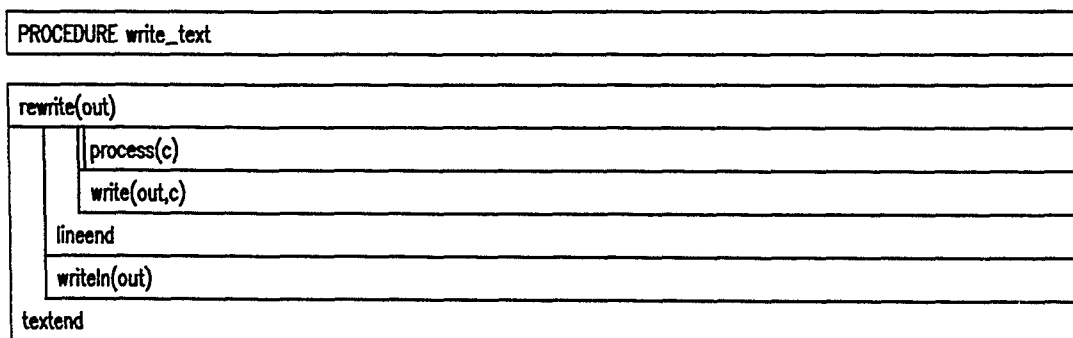
Texte sind typische Beispiele von Sequenzen, die eine Unterstruktur aufweisen, in der Regel Zeilen. Zur Darstellung dieser Unterstruktur wird häufig ein Trennzeichen verwendet (control character), das das Ende einer Zeile festlegt. Für Dateien vom Typ Text

sind daher noch zusätzliche Operationen definiert, die sich auf die Zeilenstruktur beziehen.

- **WRITELN(fileid):**
beendet die aktuelle Zeile des Textfiles *fileid*.
- **READLN(fileid):**
Übergang zur nächsten Zeile, *fileid*↑ bekommt den Wert des ersten Zeichens der nächsten Zeile.
- **EOLN(fileid):**
Boolesche Funktion, die angibt, ob das Ende der aktuellen Zeile des Textfiles *fileid* erreicht wurde.

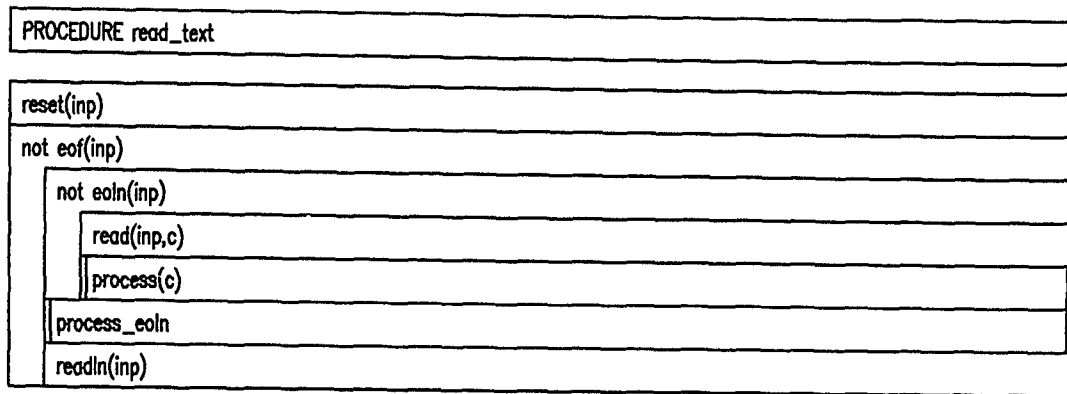
Beispiel 3.8: Textdateien

1. Ein Text wird auf die Datei *out* geschrieben. Die Prozedur *process* liefert ein Zeichen und weist es der Variablen *c* zu. Wenn die aktuelle Zeile beendet ist, liefert eine Boolesche Funktion *lineend* den Wert true; wenn der Text beendet ist, liefert *textend* den Wert true.



Prozedur write_text

2. Lesen eines Textes von der Datei *inp*. Die Prozedur *process(c)* bearbeitet das Zeichen *c*, und *process_eoln* bezeichnet eine Aktion, die bei Erreichen des Zeilenendes ausgeführt wird.



Prozedur read_text

Einschränkungen des Pascal-Typs FILE

1. Es ist nicht direkt möglich den Wert eines Elementes in einer Datei zu aktualisieren. Die Operation PUT kann beim Lesen, d.h. im Input-Mode, nicht ausgeführt werden.
2. Es ist nicht direkt möglich, neue Elemente an eine existierende Datei anzuhängen. Um in den Output-Mode zu gelangen, muß die Operation REWRITE ausgeführt werden, wodurch alle Elemente gelöscht werden.

3.2 Lineare Datenstrukturen

Listen werden häufig als Basis für sehr komplexe Datenstrukturen verwendet. Aufgrund dieser Tatsache werden in diesem Abschnitt verschiedene Typen und Realisierungen von Listen vorgestellt, die sich zum Teil wesentlich durch die Zugriffsoperationen, die zulässig sind, voneinander unterscheiden.

3.2.1 Einfach verkettete Listen (linked lists)

Bei einfach verketteten Listen handelt es sich um eine spezielle Realisierung einer Liste, in der jedes Listenelement neben Datenwerten auch die Adresse enthält, bei der das nächste Element zu finden ist. Dies ist in Pascal durch eine kombinierte Nutzung von Records und Pointern möglich.

Ausgezeichnete Pointer

head (bzw. *Anker*) zeigt auf das erste Element der Liste.

current_node zeigt auf das aktuell bearbeitete Element.

Eigenschaften

Der Pointer des letzten Elementes zeigt auf einen leeren Knoten (bei Pointern : NIL). Ist die Liste leer, so zeigt head auf einen leeren Knoten. Die physikalische Speicherung der Elemente ist im Regelfall nicht zusammenhängend.

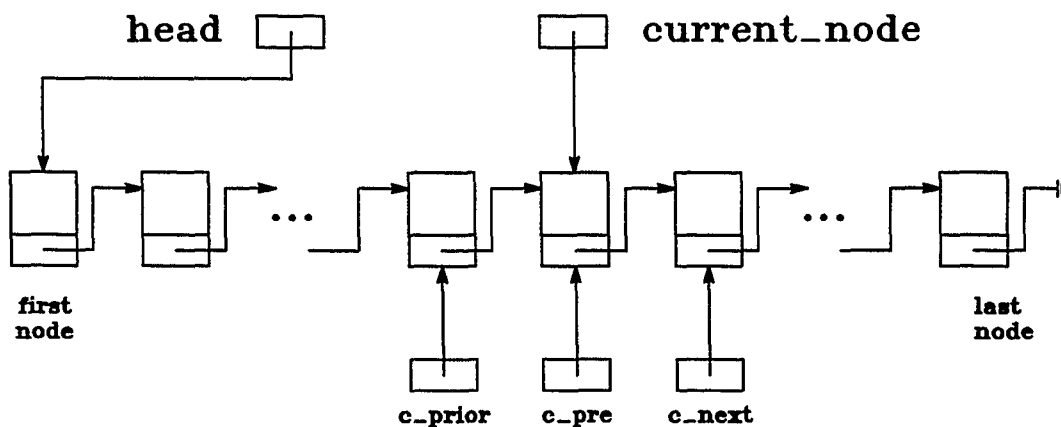


Abb. 9. Beispiel für die Darstellung einer einfach verketteten Liste

Definition 3.8: Abstrakter Datentyp *Einfach verkettete Liste*

Eine einfach verkettete Liste ist eine Folge von null oder endlich vielen Elementen eines gegebenen Datentyps, bei der der Nachfolger eines Elementes durch einen Pointer erreicht werden kann.

1. Elemente

Die Elemente heißen Knoten. Jeder Knoten enthält eine Datenkomponente und einen Pointer auf einen Knoten.

2. Struktur

Die Struktur zwischen den Knoten ist linear.

3. Operationen

Konvention: Der **current_node** zeigt auf den aktuell bearbeiteten Knoten in der Liste. Manchmal ist es bei der Formulierung der post-condition notwendig, sich auf den **current_node** zu beziehen, der vor der Ausführung der Operation ak-

tuell war. Dies geschieht durch die Bezeichnungen **c_pre**. Dabei wird zwischen dem Knoten und dem Pointer, der auf den Knoten zeigt, nicht weiter unterschieden. Dies ist für die programmtechnische Formulierung natürlich erforderlich (siehe Realisierung). Der Vorgänger bzw. Nachfolger von **c_pre** kann über **c_prior** bzw. **c_next** (siehe Abb. 9) angesprochen werden. Falls auf den Zustand der gesamten Liste vor der Ausführung der Operation Bezug genommen werden muß, wird die Bezeichnung **ℓ_pre** verwendet.

Die Operation **create** muß vor jeder anderen Operation ausgeführt werden. Alle Operationen (bis auf **create**) haben als zusätzliche Pre-condition, daß die Liste existiert.

a. Basis-Operationen

1) *insert_after* (*e : stdelement*)

```
pre    Die Liste ist nicht voll.
post   Ein neuer Knoten mit dem Datenelement e ist in der Liste ent-
       halten, dieser Knoten ist auch der neue current_node.
IF      ℓ_pre war nicht leer
THEN    current_node ist Nachfolger von c_pre und c_next ist
       Nachfolger vom current_node.
```

2) *insert_before* (*e : stdelement*)

```
pre    Die Liste ist nicht voll.
post   Ein neuer Knoten mit dem Datenelement e ist in der Liste ent-
       halten, dieser Knoten ist auch der neue current_node.
IF      ℓ_pre war nicht leer
THEN
      IF      c_pre war erstes Element in der Liste
      THEN    c_pre ist Nachfolger vom current_node.
      ELSE    current_node ist Nachfolger von c_prior und
              c_pre ist Nachfolger vom current_node.
```

3) *delete*

```
pre    Die Liste ist nicht leer.
post   c_pre ist nicht mehr in der Liste.
IF      c_pre gleich erster Knoten
THEN    c_next ist erster Knoten, falls er existiert.
ELSE    Nachfolger von c_prior ist c_next.
```

```

IF      Liste nicht leer
THEN   current_node zeigt auf ersten Knoten
ELSE   current_node zeigt auf leeren Knoten.

```

4) *retrieve (var e : stdelement)*

```

pre     Die Liste ist nicht leer.
post    e enthält das Datenelement von c_pre.

```

5) *update (e : stdelement)*

```

pre     Die Liste ist nicht leer.
post    c_pre enthält e als Datenelement.

```

b. Manipulations-Operationen

1) *find_first*

```

pre     Die Liste ist nicht leer.
post    current_node zeigt auf ersten Knoten.

```

2) *find_next (var fail : BOOLEAN)*

```

pre     Die Liste ist nicht leer.
post
    IF      c_pre zeigt nicht auf den letzten Knoten der Liste
    THEN   current_node zeigt auf c_next; fail ist false
    ELSE   fail ist true.

```

3) *find_prior (var fail : BOOLEAN)*

```

pre     Die Liste ist nicht leer.
post
    IF      c_pre zeigt nicht auf den ersten Knoten der Liste
    THEN   current_node gleich c_prior; fail ist false
    ELSE   fail ist true.

```

4) *locate* (*e* : *stdelement*; var *fail* : *BOOLEAN*)

```
pre    Die Liste ist nicht leer.
post
      IF      e ist in der Liste
      THEN fail ist false und current_node zeigt auf den Knoten,
            der e als Datenelement enthält
      ELSE fail ist true und current_node zeigt auf den letzten
            Knoten der Liste.
```

c. Precondition-Test-Operationen

1) *empty* : *BOOLEAN*

```
post
      IF      Liste nicht leer
      THEN empty ist false
      ELSE empty ist true.
```

2) *full* : *BOOLEAN*

```
post
      IF      Liste ist voll
      THEN full ist true
      ELSE full ist false.
```

d. Hilfs-Operation

create

```
post    Eine leere verkettete Liste existiert.
```

In Abb. 10 kann die Operation *insert_after* an einem Beispiel nachvollzogen werden.

Bemerkung:

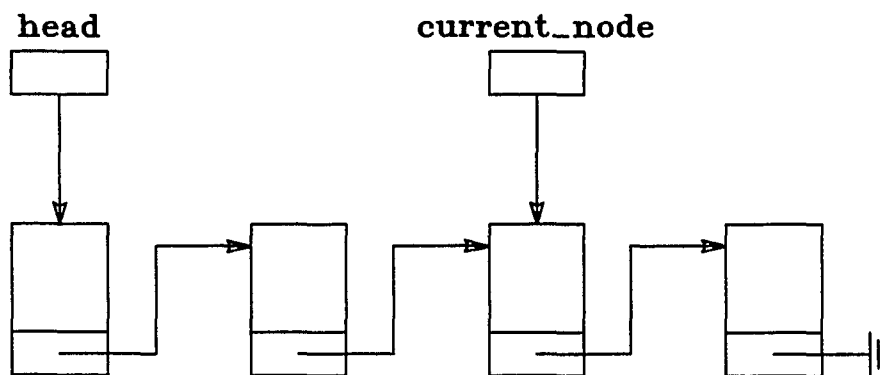
Da in einem Programm unter Umständen mehrere Datenstrukturen verwendet werden, der Name *stdelement* jedoch eindeutig benutzt werden muß, sollte dem Typ *stdelement* entsprechend der Datenstruktur ein geeigneter Präfix vorangestellt werden.

Beispiel 3.9: Namenskonvention für *stdelement*

seq_stdelement für das *stdelement* einer Sequenz, oder

linked_stdelement für das *stdelement* einer einfach verketteten Liste.

Vorher:



Nachher:

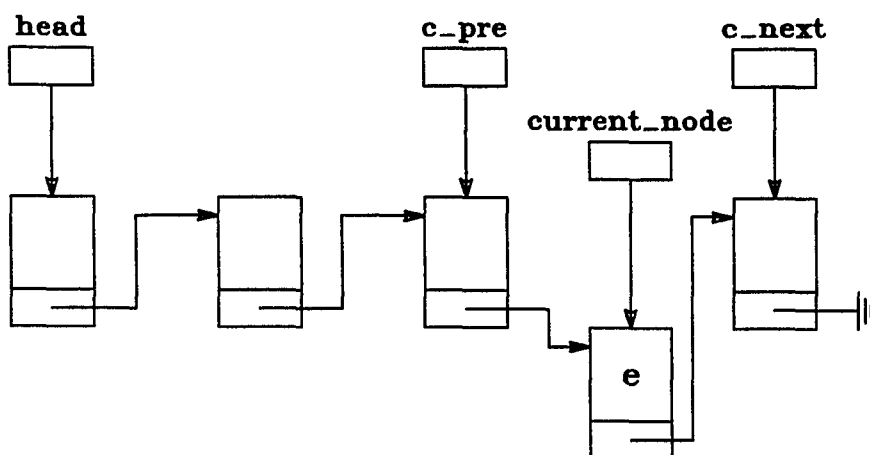


Abb. 10. Liste vor und nach der Ausführung von `insert_after`

3.2.1.1 Realisierung über Pointer

1. Repräsentation

```

TYPE    listpointer    = ↑node;
        node           = RECORD
                                element : stdelement;
                                next    : listpointer;
                                END; (* node *)

VAR     head,p          : listpointer;
        current_node    : listpointer;
        fail            : BOOLEAN;
        element         : stdelement;

```

2. Implementation

PROCEDURE create
head:=nil
current_node:=nil

Prozedur create

PROCEDURE insert_after(e:stdelement)	
new(p)	
p↑.element:=e	
head<>nil	
true	false
p↑.next:=current_node↑.next	p↑.next:=nil
current_node↑.next:=p	head:=p
current_node:=p	

Prozedur insert_after

PROCEDURE insert_before(e:stdelement)	
current_node=head	
true	false
new(p)	p:=head
p↑.element:=e	p↑.next<>current_node
p↑.next:=head	p:=p↑.next
head:=p	current_node:=p
current_node:=p	insert_after(e)

Prozedur insert_before

PROCEDURE delete	
current_node<>head	
true	false
p:=head	head:=head↑.next
p↑.next<>current_node	
p:=p↑.next	
p↑.next:=current_node↑.next	
dispose(current_node)	
current_node:=head	

Prozedur delete

PROCEDURE retrieve(var e:stdelement)
e:=current_node↑.element

Prozedur retrieve

PROCEDURE update(e:stdelement)
current_node↑.element:=e

Prozedur update

```
PROCEDURE find_first
current_node:=head
```

Prozedur find_first

```
PROCEDURE find_prior(var fail:boolean)
fail:=false
current_node<>head
true
p:=head
p↑.next<>current_node
p:=p↑.next
current_node:=p
false
fail:=true
```

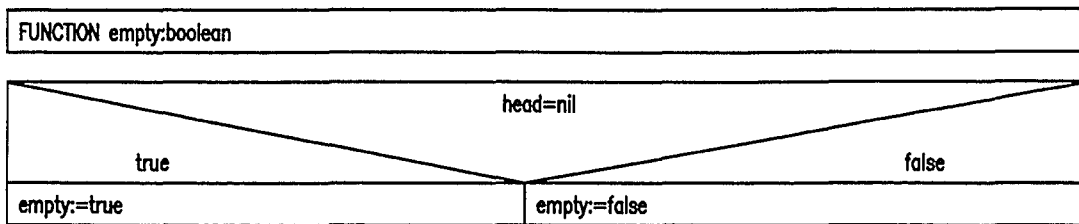
Prozedur find_prior

```
PROCEDURE find_next(var fail:boolean)
fail:=false
current_node↑.next<>nil
true
current_node:=current_node↑.next
false
fail:=true
```

Prozedur find_next

```
PROCEDURE locate(e:stdelement; var fail:boolean)
fail:=false
find_first
retrieve(element)
((not fail) and (element<>e))
find_next(fail)
retrieve(element)
```

Prozedur locate



Funktion empty

Beurteilung der elementaren Listenoperationen

in Bezug auf den erforderlichen Zeitbedarf, der in Abhängigkeit von der Anzahl n der bereits gespeicherten Elemente zur Ausführung notwendig ist:

create Schnell, da konstante Anzahl von Operationen; d.h. $O(1)$.

insert_after Schnell, da direkter Zugriff über den Pointer (konstante Anzahl von Operationen) erfolgt.

$$T_{\text{insert_after}}(n) \leq c \bullet 1$$

$$\text{Also: } T_{\text{insert_after}}(n) \in O(1)$$

insert_before Langsame Operation, da der Vorgänger gesucht werden muß, bevor das Element eingeordnet werden kann. Falls alle Positionen, an denen eingefügt werden kann, mit gleicher Wahrscheinlichkeit vorkommen, gilt:

$$\begin{aligned}
 T_{\text{insert_before}}^{\text{av}} &= \frac{1}{n} \bullet 0 + \frac{1}{n} \bullet 1 + \dots + \frac{1}{n} (n-1) \\
 &= \frac{1}{n} \sum_{i=1}^{n-1} i \\
 &= \frac{1}{n} \frac{(n-1)n}{2} \\
 &= \frac{n-1}{2} \in O(n)
 \end{aligned}$$

delete Langsam, da der Vorgänger gesucht werden muß, bevor das Element gelöscht werden kann. Falls alle Positionen, an denen gelöscht werden kann, mit gleicher Wahrscheinlichkeit vorkommen, gilt:

$$T_{\text{delete}}^{\text{av}}(n) = \frac{1}{n} \sum_{i=1}^{n-1} i = \frac{n-1}{2} \in O(n)$$

retrieve	Schnell, da unmittelbarer Zugriff über den Pointer <code>current_node</code> ; d.h. $O(1)$.
update	Schnell, da unmittelbarer Zugriff über den Pointer <code>current_node</code> ; d.h. $O(1)$.
find_first	Schnell, da unmittelbarer Zugriff auf ersten Knoten vorhanden; d.h. $O(1)$.
find_next	Schnell, da unmittelbarer Zugriff auf nächsten Knoten vorhanden; d.h. $O(1)$.
find_prior	Langsam, da Liste von Anfang an durchsucht werden muß.

$$\begin{aligned}
 T_{\text{find_prior}}^{\text{av}}(n) &= \frac{1}{n} \sum_{i=1}^{n-1} i \\
 &= \frac{n-1}{2} \in O(n)
 \end{aligned}$$

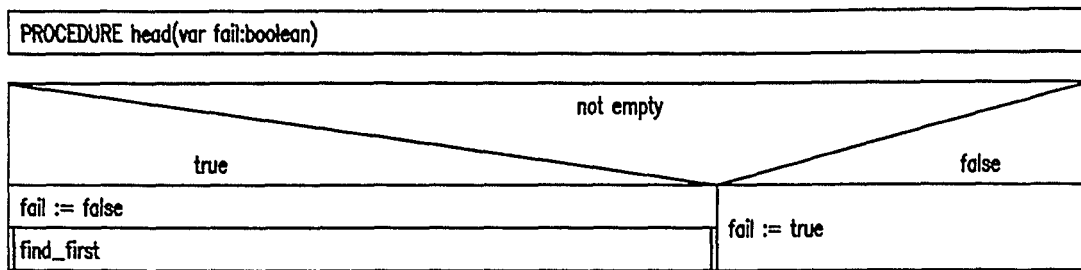
locate	Langsam, da Liste von Anfang an durchsucht werden muß.
--------	--

$$\begin{aligned}
 T_{\text{locate}}^{\text{av}}(n) &= \frac{1}{n} \sum_{i=1}^n i = n \frac{(n+1)}{2n} \\
 &= \frac{n+1}{2} \in O(n)
 \end{aligned}$$

empty	Schnell, da Test auf die Existenz eines Knotens in einem Schritt erfolgt; d.h. $O(1)$.
-------	---

Bemerkung:

Mit Hilfe der vorgestellten Basisoperationen lassen sich geeignete neue Prozeduren konstruieren, die für die Verwendung in einer Problemlösung besser geeignet sind. Als Beispiel kann die Prozedur *head* dienen, die die Basisoperation *find_first* realisiert, jedoch vorab implizit einen Test durchführt, ob die Liste leer ist.



Prozedur head

Implementation einer verketteten Liste mit Hilfe von Arrays (statische Länge)

Bei Implementation mit Hilfe von endlichen, statisch festgelegten Datenstrukturen, z.B. in der Programmiersprache FORTRAN, benötigt man eine neue Funktion **full** (vgl. auch Definition 3.8). Diese Funktion überprüft, ob noch ein Element eingefügt werden kann. Dieses war bei dynamischen Datenstrukturen unwichtig, da nur der Adreßbereich eine Grenze setzt.

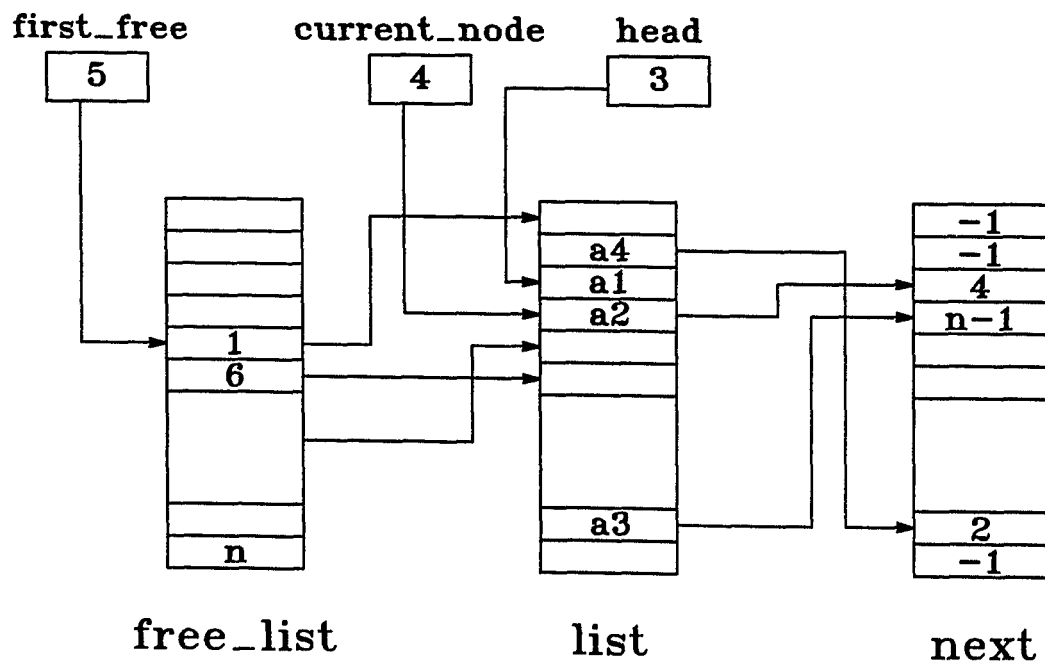


Abb. 11. Liste mit drei Feldern

3.2.1.2 Realisierung mit Hilfe von Feldern

1. Repräsentation

```
TYPE listpointer      = -1..maxlist + 1;
   listpointer_array  = ARRAY [1..maxlist] OF listpointer;
VAR list              : ARRAY [1..maxlist] OF stdelement;
   next, free_list    : listpointer_array;
   head, current_node : listpointer;
   first_free         : listpointer;
   i, p               : listpointer; (* Hilfsvariablen *)
   e                  : stdelement;
   fail               : BOOLEAN;
```

2. Implementation

PROCEDURE create
head:=0
current_node:=0
FOR i:=1 TO maxlist
free_list[i]:=i
next[i]:=-1
first_free:=1

Prozedur create

PROCEDURE insert_after(e:stdelement)	
p:=free_list[first_free]	
first_free:=first_free+1	
list[p]:=e	
current_node>0	
true	false
next[p]:=next[current_node]	next[p]:=-1
next[current_node]:=p	head:=p
current_node:=p	

Prozedur insert_after

PROCEDURE insert_before(e:stdelement)	
current_node<>head	
true	false
p:=head	p:=free_list[first_free]
next[p]<>current_node	first_free:=first_free+1
p:=next[p]	list[p]:=e
current_node:=p	next[p]:=head
insert_after(e)	head:=p
	current_node:=p

Prozedur insert_before

PROCEDURE delete	
current_node<>head	
true	false
p:=head	head:= next[head]
next[p]<>current_node	
p:=next[p]	
next[p]:=next[current_node]	
first_free:=first_free-1	
free_list[first_free]:=current_node	
current_node:=head	

Prozedur delete

FUNCTION full: boolean
full:=(first_free=(maxlist+1))

Funktion full

Beurteilung der elementaren Listenoperationen bei einer Array-Implementation mit drei Feldern

create $O(n)$, da die ganze Liste durchlaufen wird.

full Schnell, da Test auf volle Liste in einem Schritt erfolgen kann; d.h. $O(1)$.

Die übrigen Operationen insert_after, insert_before, delete, retrieve, update, find_first, find_next, locate und empty entsprechen den Operationen bei der Realisierung der einfach verketteten Liste über Pointer.

3.2.1.3 Realisierung durch ein statisches Feld ohne Pointer

Sonderfall

Die Implementation einer linearen Liste mit Hilfe eines Arrays, bei der die Verkettung nicht durch explizite Angabe von Pointern, sondern durch die gegebene Ordnung auf dem Speicherplatz gegeben ist (statisches Feld). Das Element $a[i]$ ist der Vorgänger von $a[i+1]$, falls list[i] das Datenelement $a[i]$ und list[i+1] das Datenelement $a[i+1]$ enthält.

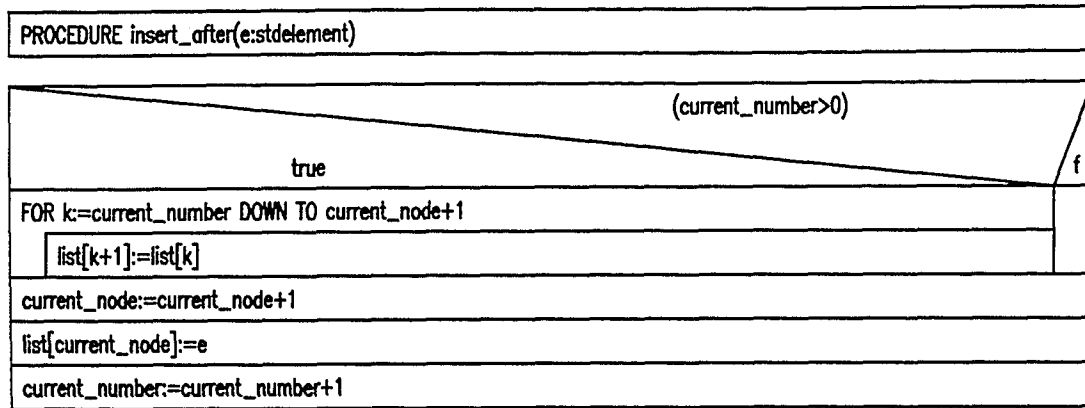
1. Repräsentation

```
TYPE    list_index      = 0..maxlist;
VAR     list             : ARRAY [1..maxlist] OF stdelement;
        current_number   : list_index; (* Anzahl Knoten in der Liste *)
        current_node,k   : list_index;
        e                 : stdelement;
        fail              : BOOLEAN;
```

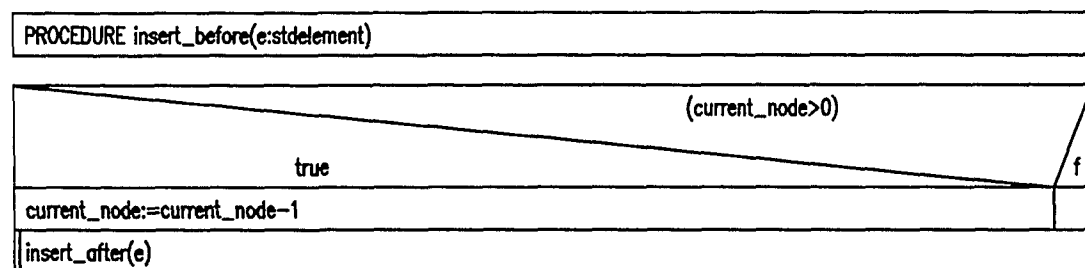
2. Implementation

PROCEDURE create
current_number:=0
current_node :=0

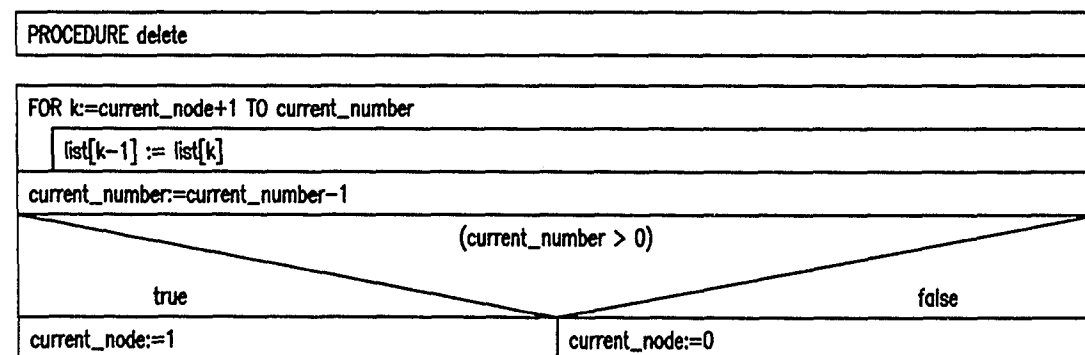
Prozedur create



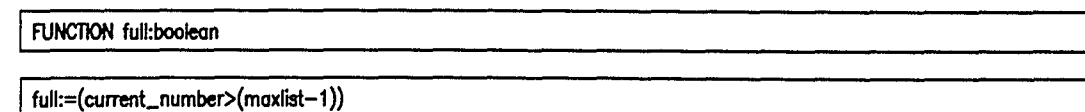
Prozedur insert_after



Prozedur insert_before



Prozedur delete



Funktion full

Beurteilung der elementaren Listenoperationen bei einer Feldimplementation ohne Pointer

insert_after $O(n)$, da im Durchschnitt $\frac{n-1}{2}$ Shift-Operationen nötig sind.

$$\begin{aligned} T_{\text{insert_after}}^{\text{av}}(n) &= \frac{1}{n} (0 + 1 + \dots + (n-1)) \\ &= \frac{1}{n} \sum_{i=1}^{n-1} i \\ &= \frac{1}{n} \frac{n(n-1)}{2} \\ &= \frac{n-1}{2} \in O(n) \end{aligned}$$

full $O(1)$, da unmittelbarer Zugriff.

Die übrigen Operationen create, insert_before, delete, retrieve, update, find_next, find_prior und empty entsprechen den Operationen bei der Realisierung der einfach verketteten Liste über Pointer.

3.2.2 Mehrfach verkettete Listen

Definition 3.9: n-fach verkettete Liste

Eine n-fach verkettete Liste ist eine Folge von null oder endlich vielen Elementen eines gegebenen Datentyps, bei denen die Nachfolger eines Elementes durch Pointer erreicht werden.

1. Elemente

Die Elemente heißen Knoten. Jeder Knoten enthält eine Datenkomponente und n Knotenpointer.

2. Struktur

Die Struktur zwischen diesen Knoten ist linear.

3. Operationen

Die Operationen können direkt aus Definition 3.8 übernommen werden.

Begründung für die Verwendung von mehrfach verketteten Listen

Ziel ist, daß Listenelemente nur einmal gespeichert werden. Auf diesen Elementen sind jedoch u.U. mehrere verschiedene Ordnungen (Reihenfolgen) definiert, z.B. kann ein Mitarbeiter-Record nach folgenden Kriterien sortiert sein:

- Name
- Geburtsdatum
- Kinderanzahl
- Postleitzahl

Wenn die Elemente alphabetisch nach dem Namen sortiert sind, muß bei der Suche nach einer bestimmten Altersklasse der gesamte Datenbestand durchsucht werden. Sind die Elemente zusätzlich nach dem Geburtsdatum sortiert, entfällt diese zeitaufwendige Operation.

Ein allgemeines Dilemma bei dem Entwurf von Algorithmen besteht darin, schnellere Operationen zu implementieren, die jedoch mehr Speicherplatz benötigen, oder langsamere Operationen zu akzeptieren, die weniger Speicherplatz erfordern. Um die zeitaufwendigen Operationen `insert_before` und `delete` bei der Datenstruktur **einfach verkettete Liste** zu beschleunigen, werden häufig **doppelt verkettete Listen** verwendet. In doppelt verketteten Listen hat ein Knoten nicht nur einen Pointer auf seinen Nachfolger, sondern auch einen Pointer auf seinen Vorgänger. Damit ist die doppelt verkettete Liste ein Sonderfall der mehrfach verketteten Liste der Ordnung zwei, bei der ein zweiter Pointer verwendet werden kann, um eine beliebige andere Ordnung festzulegen.

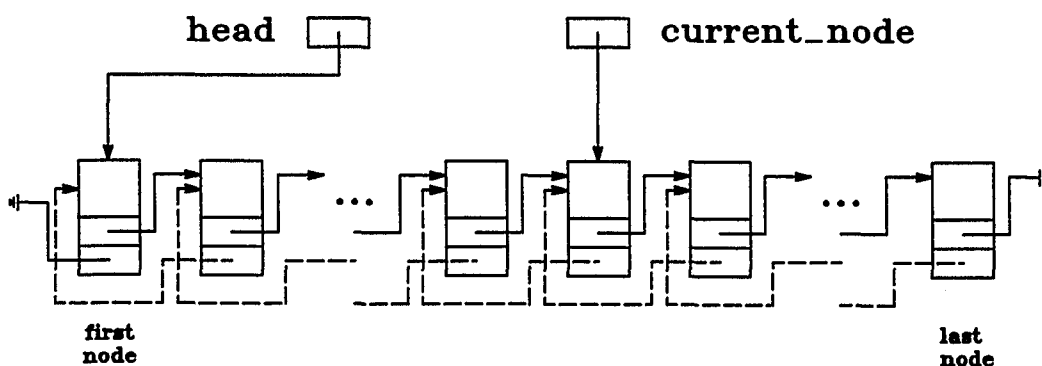


Abb. 12. Beispiel für die Darstellung einer doppelt verketteten Liste

Realisierung einer doppelt verketteten Liste

1. Repräsentation

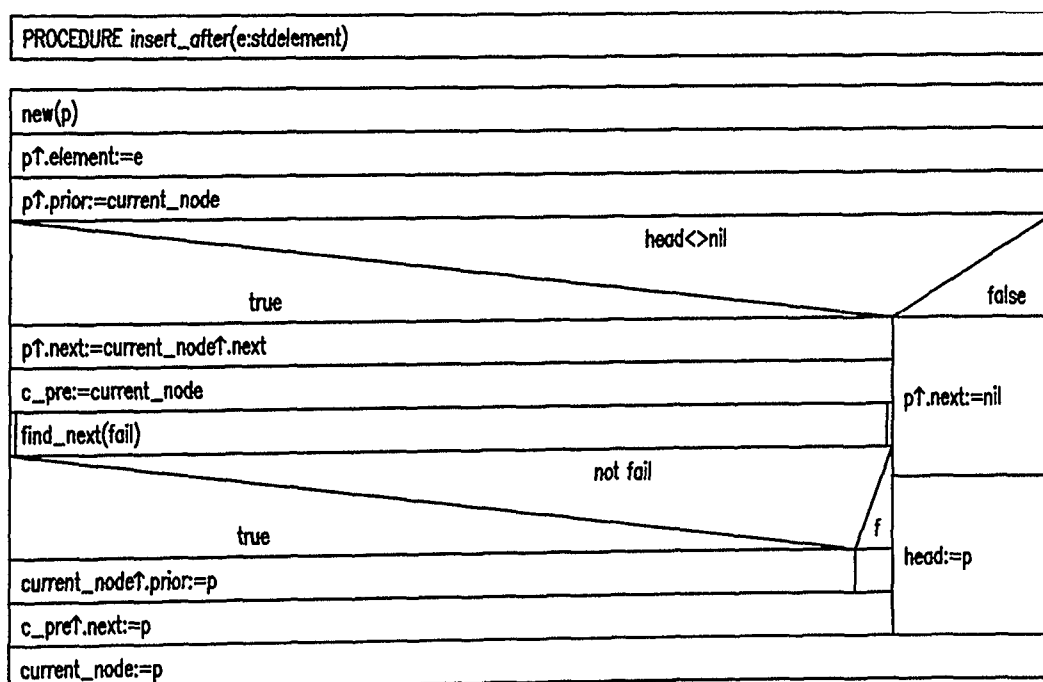
```

TYPE      listpointer      = ↑node;
          node              = RECORD
                                element      : stdelement;
                                next         : listpointer;
                                prior       : listpointer;
                                END; (* node *)

VAR       head,current_node,p : listpointer;
          c_pre                : listpointer;
          fail                 : BOOLEAN;
          element              : stdelement;
  
```

2. Implementation

Die Veränderungen der Implementation bei den meisten Operationen sind geringfügig und können in [10] nachgelesen werden. Besondere Beachtung gilt den Operationen `insert_after`, `insert_before` und `delete`.



Prozedur `insert_after`

PROCEDURE insert_before(e:stdelement)		
current_node<>head		
true	false	
find_prior(fail)	new(p)	
	p↑.element:=e	
	p↑.prior:=nil	
	p↑.next:=head	
insert_after(e)	head<>nil	
	true	
	head↑.prior:=p	
	head:=p	
	current_node:=p	

Prozedur insert_before

PROCEDURE delete		
current_node<>head		
true		false
c_pre:=current_node		head:=head↑.next
find_prior(fail)		dispose(current_node)
current_node↑.next:=c_pre↑.next		
dispose(c_pre)		not empty
c_pre:=current_node		
find_next(fail)		
not fail		
true		f
current_node↑.prior:=c_pre		find_first
current_node:=head		current_node↑.prior:=nil

Prozedur delete

Beurteilung der elementaren Listenoperationen bei doppelt verketteten Listen (über Pointer)

insert_before Schnell, da Zugriff auf Vorgänger über einen Pointer erfolgt.

$$T_{\text{insert_before}}(n) \leq c \cdot 1; \quad \text{Also: } T_{\text{insert_before}}(n) \in O(1)$$

delete Schnell, da der Vorgänger über einen Pointer erreicht werden kann.
das bedeutet eine konstante Anzahl von Operationen

$$T_{\text{delete}}(n) \leq c \cdot 1; \quad \text{Also: } T_{\text{delete}}(n) \in O(1)$$

find_prior Schnell, da unmittelbarer Zugriff auf Vorgänger vorhanden; d.h. $O(1)$.

Die übrigen Operationen create, insert_after, retrieve, update, find_first, find_next, locate und empty entsprechen weitgehend den Operationen bei der Realisierung der einfach verketteten Liste über Pointer.

Operationen	einfach verkettete Liste						doppelt verkettete Liste	
	dynamische Pointer-Realisierung		statisches Feld mit Pointern		statisches Feld ohne Pointer			
insert_after	c	O(1)	c	O(1)	$\frac{(n-1)}{2}$	O(n)	c	O(1)
insert_before	$\frac{(n-1)}{2}$	O(n)	$\frac{(n-1)}{2}$	O(n)	$\frac{(n+1)}{2}$	O(n)	c	O(1)
delete	$\frac{(n-1)}{2}$	O(n)	$\frac{(n-1)}{2}$	O(n)	$\frac{(n-1)}{2}$	O(n)	c	O(1)
retrieve	1	O(1)	1	O(1)	1	O(1)	1	O(1)
update	1	O(1)	1	O(1)	1	O(1)	1	O(1)
find_first	1	O(1)	1	O(1)	1	O(1)	1	O(1)
find_next	c	O(1)	c	O(1)	c	O(1)	c	O(1)
find_prior	$\frac{(n-1)}{2}$	O(n)	$\frac{(n-1)}{2}$	O(n)	c	O(1)	c	O(1)
locate	$\frac{(n+1)}{2}$	O(n)	$\frac{(n+1)}{2}$	O(n)	$\frac{(n+1)}{2}$	O(n)	$\frac{(n+1)}{2}$	O(n)
create	c	O(1)	2n	O(n)	c	O(1)	c	O(1)
empty	c	O(1)	c	O(1)	c	O(1)	c	O(1)
full	c	O(1)	c	O(1)	c	O(1)	c	O(1)

Tab. 2. : Zeitkomplexitäten der einzelnen Basisoperationen in verschiedenen Listenrealisierungen.

Zusammenfassung der bisherigen Ergebnisse für Listen

1. Verkettete Listen erlauben eine flexible Handhabung beim Einfügen und Löschen von Elementen an vorgegebenen Positionen (kein Shiften von Elementen).
2. Dem steht ein erhöhter Speicherplatz für die Pointer gegenüber.
3. Verkettete Listen sollten nur dort verwendet werden, wo häufiges Einfügen oder Löschen wichtig ist.
4. Bei den verketteten Listen folgen logisch aufeinanderfolgende Elemente u.U. nicht unmittelbar im Speicher aufeinander. Die Liste kann physikalisch auch Lücken haben.
5. Für Anwendungen, bei denen es auf einen schnellen Zugriff (oder schnelles Suchen) ankommt, sind ARRAY-Realisierungen für Listen wesentlich vorteilhafter. Auf diesen Aspekt werden wir im Kapitel 'Ausgewählte Algorithmen' noch einmal zurückkommen.

3.2.3 Stapel (stacks)

Beim Stapel (auch Stack, Push-Down-Stack, Kellerspeicher, LIFO-Liste genannt) handelt es sich um eine sehr wichtige Datenstruktur, die auf der bisher vorgestellten Datenstruktur 'einfach verkettete Liste' basiert, aber nur Operationen erlaubt, die das zuletzt eingefügte Element der Liste manipulieren. Dieses Element ist z.B. das einzige Element, das aus einem Stack entfernt werden kann (**Last-In-First-Out**).

Einsatzgebiete:

- Tablettspender in einem Selbstbedienungsrestaurant (z.B. im Seekasino)
- Auswertung von arithmetischen Ausdrücken, z.B. umgekehrt polnische Notation zur Aufnahme der Operanden arithmetischer Ausdrücke
- Rücksprungadressen bei geschachtelten Unterprogrammen (Stack dann Hard- oder Software-mäßig realisiert)

Definition 3.10: Abstrakter Datentyp *Stack*

Bei einem Stack handelt es sich um eine lineare Datenstruktur mit einem LIFO-Verhalten.

1. Elemente

Die Elemente sind von einem beliebigen Datentyp (stdelement).

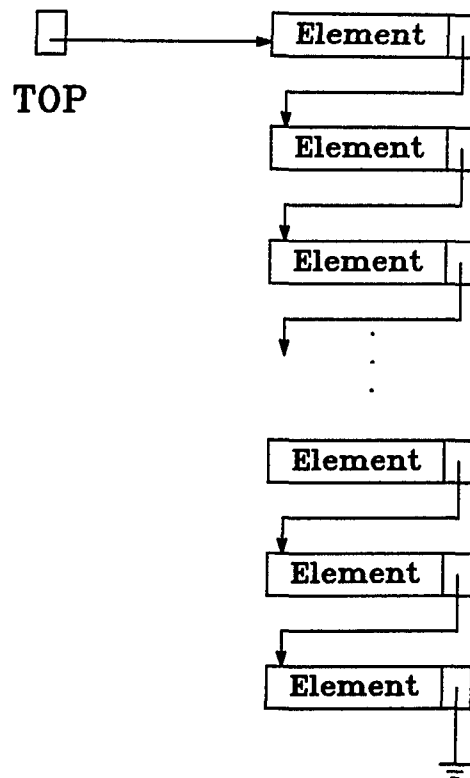


Abb. 13. Beispielhafte Darstellung eines Stacks

2. Struktur

Die Struktur setzt die Elemente so in Beziehung, daß die Ankunftsreihenfolge zu jedem Zeitpunkt feststellbar bleibt (z.B. Zeitmarke).

3. Operationen

Die Operation create muß vor jeder anderen Operation ausgeführt werden. Als zusätzliche Pre-Condition für die anderen Operationen kommt hinzu, daß der Stack existiert.

a. *push* (*e :stdelement*)

pre Der Stack ist nicht voll.

post Der Stack enthält e als das zuletzt angekommene Element.

b. *pop* (*var e :stdelement*)

pre Der Stack ist nicht leer.

post Ist e das zuletzt angekommene Element, dann enthält der Stack nicht mehr das Element e.

c. *empty* : *BOOLEAN*

```
post
    IF      Der Stack enthält kein Element
    THEN    empty ist true
    ELSE    empty ist false
```

d. *full* : *BOOLEAN*

```
post
    IF      Der Stack ist voll
    THEN    full ist true
    ELSE    full ist false
```

e. *create*

```
post    Der Stack existiert und ist leer.
```

Häufig wird auch eine Operation *top* als Test auf das oberste Element definiert. Dies ist bei der obigen Definition von *pop* nicht notwendig. Wird *pop* jedoch nur als Entfernen des obersten Elements definiert, so wird *top* notwendig.

Realisierung mit Hilfe einer verketteten Struktur in Pascal

1. Repräsentation

```
TYPE    stack_pointer = ↑stack_element;
        stack_element = RECORD
                                element    : stdelement;
                                next       : stack_pointer
                            END; (* stack_element *)

VAR      top,p          : stack_pointer;
        e               : stdelement;
```

2. Implementation

PROCEDURE create

top:=nil

Prozedur create

PROCEDURE push(e:stdelement)

new(p)

p↑.element:=e

p↑.next:=top

top:=p

Prozedur push

PROCEDURE pop(var e:stdelement)

e:=top↑.element

p:=top

top:=top↑.next

dispose(p)

Prozedur pop

FUNCTION empty:boolean

empty:=(top=nil)

Funktion empty

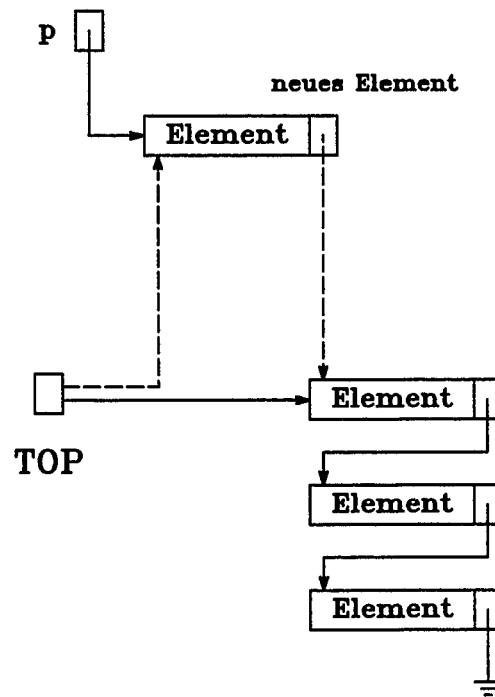


Abb. 14. Darstellung der Operation push auf dem Datentyp Stack

Realisierung eines Stacks mit Hilfe eines Feldes

1. Repräsentation

```

TYPE      stack_type  = ARRAY [1..max_stack] OF stdelement;
          stackbereich = 0..max_stack;

VAR       top         : stackbereich;
          stack        : stack_type;
  
```

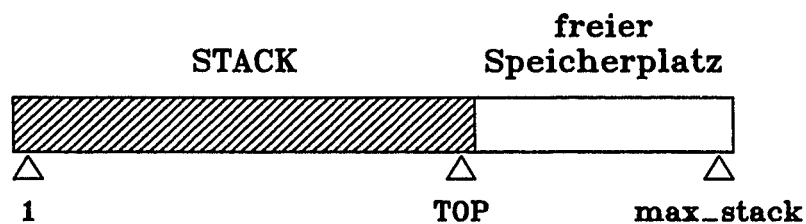


Abb. 15. Visualisierung der Feldimplementierung eines Stacks

2. Implementation

```
PROCEDURE create
```

```
top:=0
```

Prozedur create

```
PROCEDURE push(e:stdelement)
```

```
top:=top+1
```

```
stack[top]:=e
```

Prozedur push

```
PROCEDURE pop(var e:stdelement)
```

```
e:=stack[top]
```

```
top:=top-1
```

Prozedur pop

```
FUNCTION empty:boolean
```

```
empty:=(top=0)
```

Funktion empty

```
FUNCTION full:boolean
```

```
full:=(top=max_stack)
```

Funktion full

In der nächsten Tabelle finden sich die Zeitkomplexitäten in den verschiedenen Realisierungen.

Operation	verkettete Liste	Feldimplementation
push	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
empty	$O(1)$	$O(1)$
full	$O(1)$	$O(1)$
create	$O(1)$	$O(1)$

Tab. 3. : Zeitkomplexitäten für die Operationen auf der Datenstruktur Stack

Stacks werden in Programmiersprachen (z.B. in Pascal) verwendet, um die dynamische Zuordnung von lokalen Datenbereichen zu realen Adressen zu realisieren. Dies ist z.B. bei rekursiven Prozeduren notwendig.

Definition 3.11: Rekursive Prozedur

Eine rekursive Prozedur ist eine Prozedur (Funktion), die unmittelbar durch sich selbst oder mittelbar über andere Prozeduren wieder aufgerufen wird.

Bemerkung:

In Pascal sind rekursive Prozeduraufrufe erlaubt, in FORTRAN sind rekursive Programmaufrufe verboten.

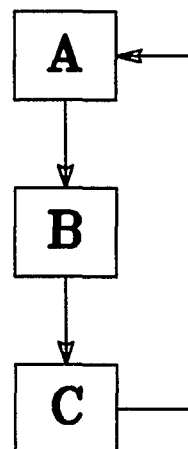


Abb. 16. Mittelbarer rekursiver Aufruf

Beispiel 3.10: Rekursive Prozeduren durch unmittelbaren Aufruf

```
FUNCTION fakultaet(n:INTEGER):INTEGER;
BEGIN
  IF (n = 0)
  THEN fakultaet: = 1
  ELSE fakultaet: = fakultaet(n-1)*n
END;
```

Beispiel 3.11: LIFO-Stack zur Realisierung rekursiver Prozeduraufrufe

$$\begin{aligned} n! &= n \cdot (n-1)! \\ &= n \cdot (n-1) \cdot (n-2)! \\ &= n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1 \end{aligned}$$

Der rekursive Funktionsaufruf führt die Berechnung von $n!$ auf das reduzierte Problem von $(n-1)!$ zurück. Die Abbruchbedingung ist erfüllt, wenn $n=0$ erreicht ist.

Aufruf:

```
...
m: = fakultaet(3);
...
```

Programm-Code im Speicher:

```
IF (n = 0)
THEN BEGIN
  fakultaet: = 1;
  return
END
ELSE BEGIN
  call fakultaet(n-1);
  fakultaet: = n*fakultaet;
  return
END
```

Stackbelegung bei der rekursiven Funktion fakultaet:

	Operation	1. Rücksprung	2. Rücksprung	3. Rücksprung
1.Stack	Rücksprungadresse zum Hauptprogramm			
	n = 3			
	fakultaet = ?			fakultaet = 6
2.Stack	Rücksprungadresse nach fakultaet(3)			
	n = 2			
	fakultaet = ?		fakultaet = 2	
3.Stack	Rücksprungadresse nach fakultaet(2)			
	n = 1			
	fakultaet = ?	fakultaet = 1		
4.Stack	Rücksprungadresse nach fakultaet(1)			
	n = 0			
	fakultaet = 1			

Bemerkung:

1. Die oben beschriebene Methode der Stackverwaltung heißt dynamische Speicherverwaltung.
2. Jeder Unterprogramm- oder Funktionsaufruf bewirkt die Bereitstellung eines neuen Speicherbereichs, der als Datenstack für die lokalen Variablen dergerufenen Prozedur verwendet wird.
3. Der Anfangswert des Pointers auf den Stack wird vom Betriebssystem bereitgestellt.
4. Rekursive Programmaufrufe verbrauchen u.U. eine beträchtliche Menge Speicherplatz und viel Zeit. Sie erlauben jedoch u.U. mit sehr wenig Programmieraufwand die Lösung extrem komplexer Aufgabenstellungen. Das Abwägen zwischen der Effizienz der Programmierung und dem Laufzeitverhalten ist manchmal jedoch notwendig.
5. FORTRAN hat eine statische Speicherverwaltung, d.h. rekursive Programmlösungen sind nicht möglich.
6. Pascal hat grundsätzlich dynamische Speicherverwaltung.
7. Bei rekursiven Programmen werden der Programm-Code und die veränderlichen Daten strikt getrennt, und damit wird das Programm 'reentrant'; 'reentrant' Programme können mehrfach, quasi gleichzeitig ausgeführt werden, noch bevor

die erste Programmausführung beendet ist, wobei jedem Benutzer des 'reentrant'-Programms sein eigener Stack zugeordnet ist.

3.2.4 Warteschlangen (queues)

Eine Warteschlange (auch Queue, FIFO-Liste, FIFO-Stack genannt) ist eine Liste, bei der nur am Listeneende Elemente hinzugefügt und nur am Listenanfang Elemente entfernt werden können.

Bemerkung:

1. FIFO steht für **First-In-First-Out**.
2. Der Anfang einer Queue wird i.allg. mit Head, das Ende mit Tail bezeichnet.
3. Das Einfügen von Elementen in die Queue wird PUT (auch Enqueue), das Löschen mit gleichzeitigem Lesen wird GET (Dequeue) genannt.

Definition 3.12: Abstrakter Datentyp *Queue*

Eine Queue ist ein Datentyp mit einem FIFO-Verhalten (First-In-First-Out).

1. Elemente
Die Elemente sind von einem beliebigen Datentyp (stdelement).
2. Struktur
Die Struktur setzt die Elemente so in Beziehung, daß ihre Ankunftsreihenfolge erhalten bleibt.
3. Operationen
Die Operation create muß vor jeder anderen Operation ausgeführt werden. Alle Operationen (bis auf create) haben als zusätzliche Pre-condition, daß die Queue existiert. **q_pre** bezeichnet den Zustand der Queue zur Pre-condition Zeit.

a. *put (e:stdelement)*

pre Die Queue ist nicht voll.

post Die Queue enthält e als das zuletzt angekommene Element.

b. *get* (*var e:stdelement*)

pre Die Queue ist nicht leer.

post e ist das Element, das am längsten in q_pre ist. Das Element e ist
 nicht länger in der Queue.

c. *empty* : *BOOLEAN*

post

IF Die Queue ist leer

THEN *empty* = true

ELSE *empty* = false

d. *full* : *BOOLEAN*

post

IF Die Queue ist voll

THEN *full* = true

ELSE *full* = false

e. *length*

post *length* gibt die Anzahl der Elemente in der Queue an.

f. *create*

post Die Queue existiert und ist leer.

Operationen: PUT am tail
 GET am head

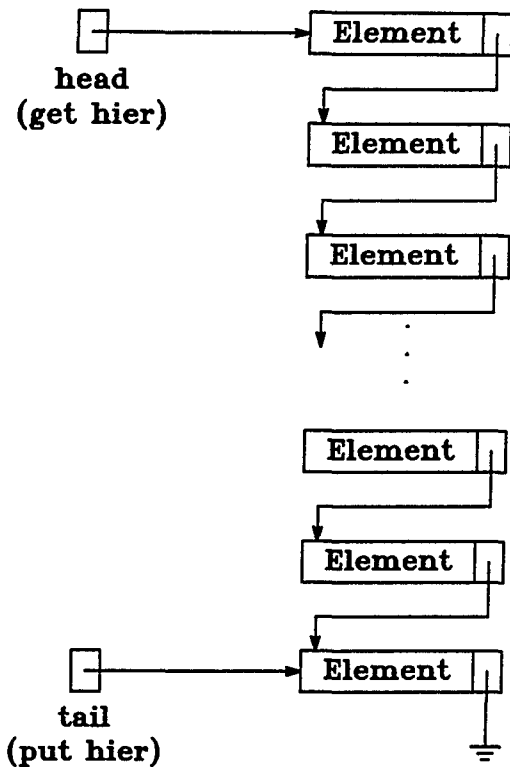


Abb. 17. Queue (übliche Darstellung)

Einsatzgebiete von Queues

- im täglichen Leben
 - Anstellen am Schalter der Post.
- Betriebssysteme
 - Zuteilung von Ressourcen
 - CPU-Verwaltung
 - I/O Request (zirkuläre Queues)
 - Disk-Verwaltung

Realisierung einer Queue mit Hilfe eines Feldes

1. Repräsentation

```

CONST    max_queue = ...
TYPE     stdelement = ...
VAR      queue      : ARRAY [1..max_queue] OF stdelement;
         head,tail   : 0..max_queue;
  
```

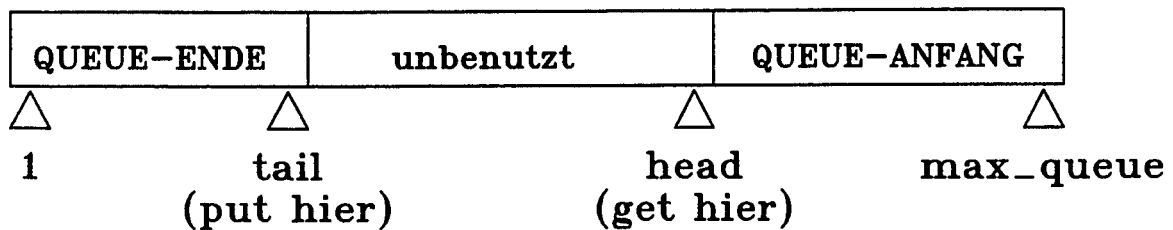
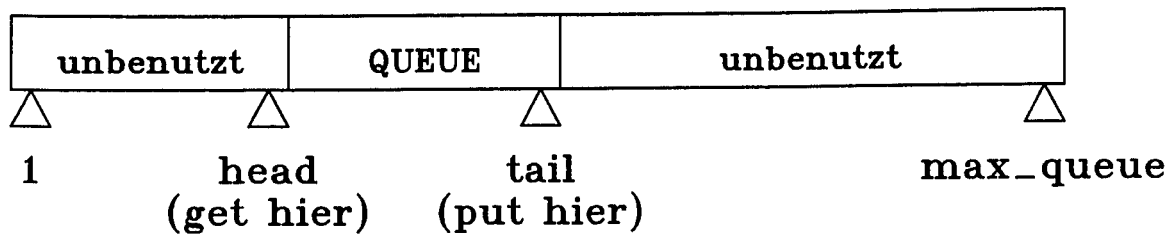


Abb. 18. Queue, realisiert mit einem Feld bei unterschiedlichem Füllungsgrad

Eine Queue kann also in einem quasi zyklischen (als zyklisch aufgefaßten) Feld abgespeichert werden.

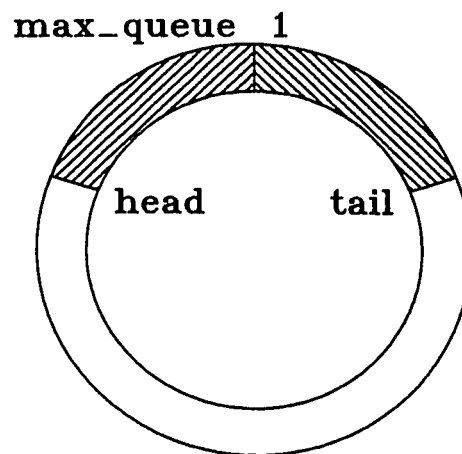
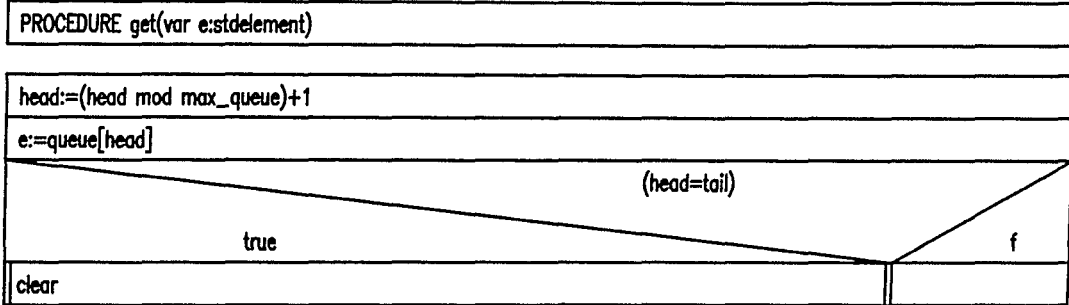


Abb. 19. Visualisierung einer zirkulären Queue

2. Implementation

```
PROCEDURE put(e:stdelement)
tail:=(tail mod max_queue)+1
queue[tail]:=e
```

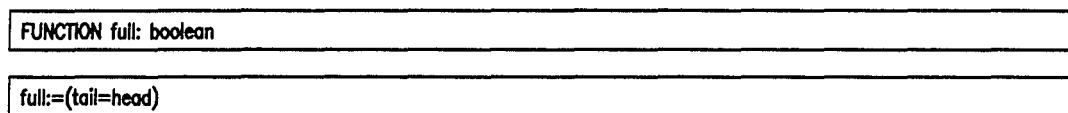
Prozedur put



Prozedur get



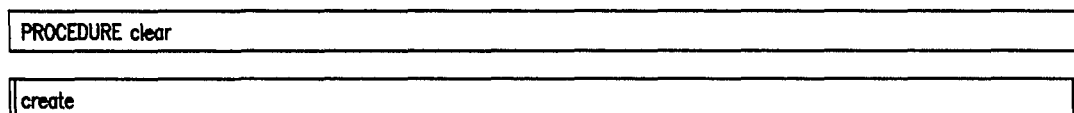
Funktion empty



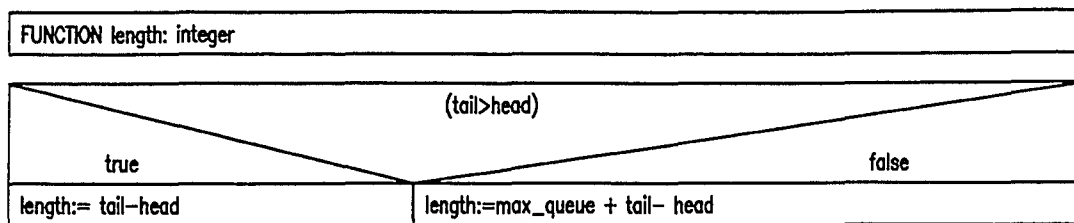
Funktion full



Prozedur create



Prozedur clear



Funktion length

Für die Implementierung mit Pointern sehen die Operationen ähnlich aus wie bei der Stack-Implementierung. Zusätzlich wird ein tail-Pointer für das Einfügen von Elementen gebraucht sowie eine Zählvariable, die die aktuelle Zahl von Elementen in der Queue enthält.

Zeitkomplexität:

Alle Operationen bei der Feldimplementierung haben die Zeitkomplexität $O(1)$. Bei der verketteten Implementation ist der Aufwand der Operation clear $O(n)$.

3.3 Graphen

Im folgenden werden Graphen eingeführt, die als Hilfsmittel zur Darstellung von zweistelligen Relationen dienen. Dabei muß berücksichtigt werden, daß die Relationen symmetrisch oder nicht symmetrisch sein können. Daraus ergibt sich eine Unterteilung in ungerichtete und gerichtete Graphen.

Definition 3.13: Gerichteter Graph

Ein gerichteter Graph $G=(V,E)$ (auch Digraph genannt) besteht aus einer endlichen, nicht leeren Menge $V=\{v_1, \dots, v_n\}$ von Knoten (vertices, nodes) und einer Menge E ($E \subset V \times V$) von geordneten Paaren $e=(u,v)$ (Kanten, edges). Jede Kante $e \in E$ hat einen Anfangsknoten u und einen Endknoten v und damit eine Richtung von u nach v ($u=v$ möglich).

Beispiel 3.12:

Sei $V = \{1,2,3,4,5,6\}$ und $E = \{(1,2), (1,4), (2,3), (3,5), (5,1)\}$

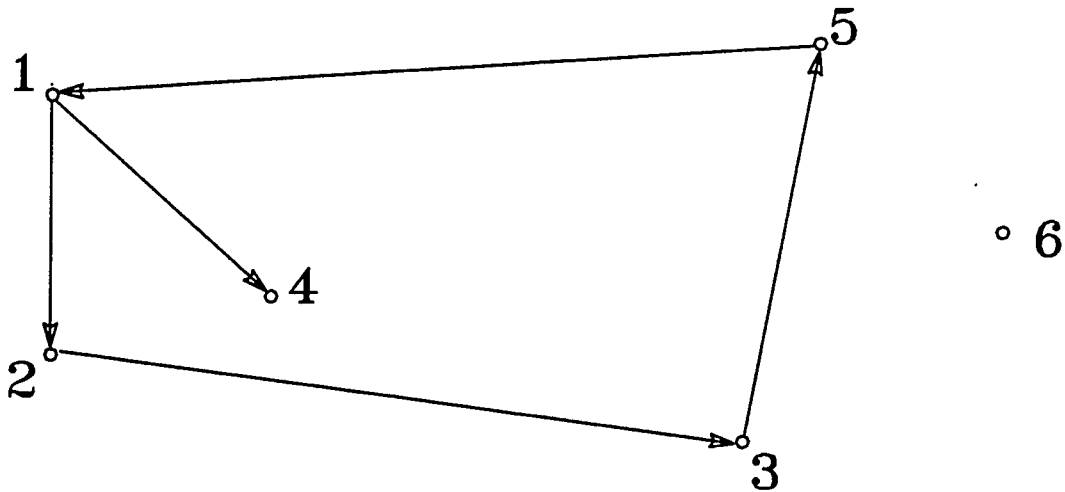


Abb. 20. Beispiel für einen gerichteten Graph

Definition 3.14: Ungerichteter Graph

Ein ungerichteter Graph $G=(V,E)$ besteht aus einer endlichen nicht leeren Menge V von Knoten und einer Menge E von ungeordneten Paaren $e=(u,v)$ von Kanten von G ($u,v \in V$).

Bemerkung:

- Wenn bei praktischen Anwendungen der Graphentheorie die Relation zwischen den Objekten, die durch den Graphen dargestellt wird, symmetrisch ist, kann man ungerichtete Graphen verwenden (d.h. sowohl $(a,b) \in E$ als auch $(b,a) \in E$ existieren).
- Man läßt bei der Beschreibung ungerichteter Graphen üblicherweise eine der beiden Kanten (a,b) bzw. (b,a) weg und faßt die Kanten als ungerichtet auf.

Beispiel 3.13: Darstellung eines ungerichteten Graphen

$$V = \{1,2,3,4,5\}$$

$$E = \{(1,2), (2,3), (3,4), (3,5), (4,5)\}$$

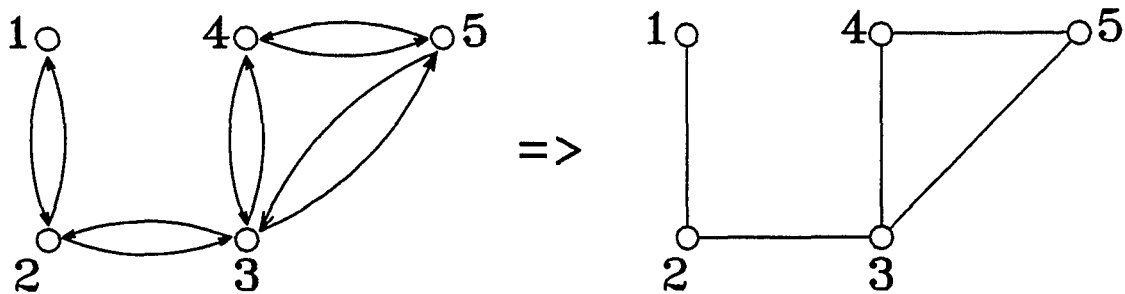


Abb. 21. Übergang von symmetrisch gerichtetem zu einem ungerichteten Graph

Beispiel 3.14: Königsberger Brückenproblem

Euler löste im Jahre 1736 folgendes Problem, das am Anfang der Graphentheorie steht: Im 18. Jahrhundert führten über die Flußläufe des Pregel in der Innenstadt von Königsberg sieben Brücken. Gibt es einen Weg vom Ausgangspunkt, bei dem man jede Brücke genau einmal überquert und schließlich zum Ausgangsort zurückkehrt?

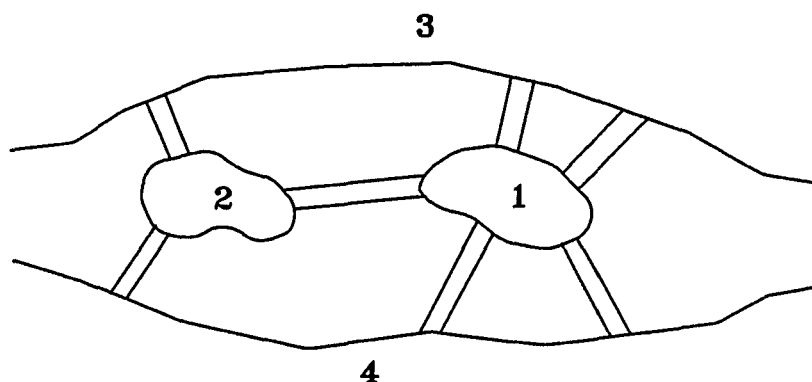


Abb. 22. Königsberger Brückenproblem

Euler zeigte, daß es keinen solchen Weg gibt, indem er das Problem auf ein Graphenproblem zurückführte. Er ersetzte jede Landmasse durch einen Knoten im Graphen und jede Brücke zwischen zwei Gebieten durch eine Kante zwischen den zugehörigen Knoten.

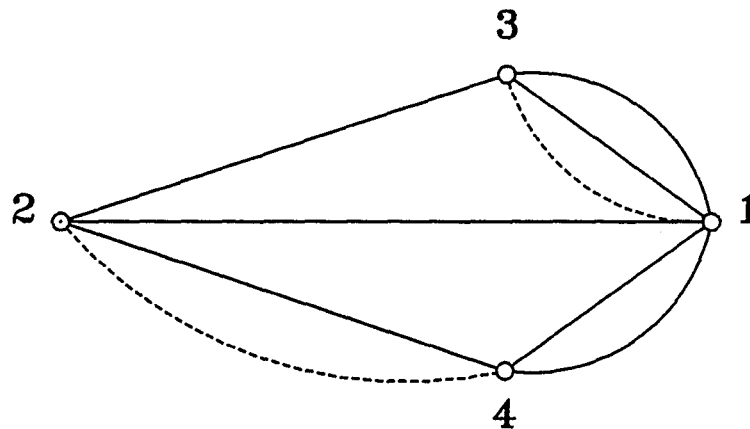


Abb. 23. Graphentheoretische Problemformulierung

Auf Graphen übertragen lautet das Königsberger Brückenproblem: Gibt es zu jedem Knoten einen Weg, der jede Kante genau einmal durchläuft und am Ende wieder am Ausgangsknoten ankommt? (→ Eulerscher Kreis)

Euler bewies für beliebige Graphen, daß ein Eulerscher Kreis dann und nur dann existiert, wenn der Grad jedes Knotens eine gerade Zahl ist, d.h. wenn an jedem Knoten eine gerade Anzahl von Kanten anliegt.

Der Graph erfüllt diese Bedingung nicht. Es folgt also, daß es den gesuchten Rundweg über die Brücken des Pregel nicht gibt. Durch Hinzufügen der gestrichelten zusätzlichen Kanten, wäre diese Situation gegeben.

Bemerkung:

In diesem Beispiel kommen Mehrfachkanten vor, z.B. die Kante zwischen Knoten 1 und Knoten 3. In der Praxis gibt es Fälle, die diese allgemeine Definition benötigen, aber im folgenden werden nur Graphen betrachtet, die solche Kanten nicht zulassen.

Definition 3.15: Weg

Sei $G=(V,E)$ ein gerichteter oder ungerichteter Graph.

Eine Folge $W=(v_1, v_2, v_3, \dots, v_n)$ heißt Weg in G , falls

$$\forall i = 1 \dots n : v_i \in V$$

und

$$\forall i = 1 \dots n - 1 : (v_i, v_{i+1}) \in E \text{ gilt.}$$

$v_1 = \alpha(W)$ heißt *Anfangsknoten* des Weges W , $v_n = \omega(W)$ heißt *Endknoten* des Weges W . (v_i) heißt *trivialer Weg* und ist stets ein Weg in $G \quad \forall v_i \in V$.
Die Länge eines Weges ist $\ell(W) = n-1$, falls n Knoten auf diesem Weg besucht werden.

Beispiel 3.15: Wege in einem Graphen

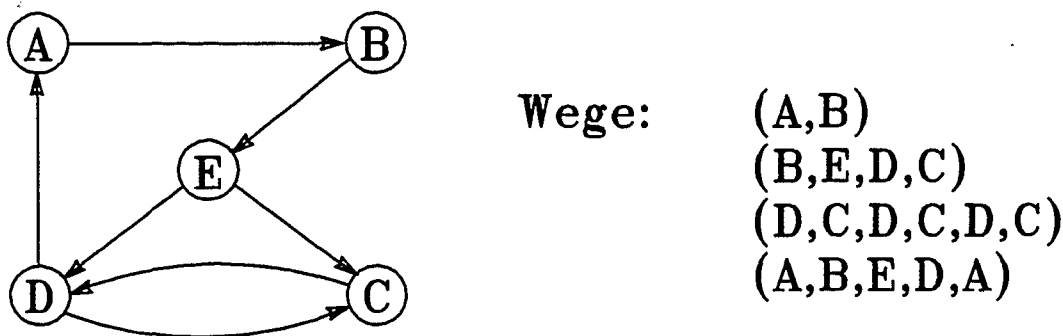


Abb. 24. Auswahl von Wegen in einem Graphen G

Weitere Begriffe aus der Graphentheorie:

- Ein Graph $G' = (V', E')$ heißt *Teilgraph* von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt.
- Ein Weg heißt *einfacher Weg*, wenn jeder Knoten auf dem Weg genau einmal vorkommt.
- Ein *Kreis* (cycle) ist ein einfacher Weg mit der Ausnahme $\alpha(W) = \omega(W)$.
- Zwei Knoten heißen *adjacent*, wenn es eine Kante gibt, die sie verbindet.
- Ein Graph heißt *gewichtet* (bewertet), wenn jeder Kante ein Wert als Gewicht zugeordnet ist (z.B. Transportkosten, Entfernung, etc.).

Speicherung von Graphen in Rechnern:

1. Adjazenzmatrix

Sei $G = (V, E)$ ein ungerichteter oder gerichteter Graph mit $V = \{v_1, \dots, v_n\}$. Die Adjazenzmatrix eines Graphen ist eine $n \times n$ Matrix $A = (a_{ij})_{i,j=1 \dots n}$ mit

$$a_{ij} = \begin{cases} 1 & \text{falls } v_j \text{ adjazent zu } v_i \text{ ist, d.h. falls } (v_i, v_j) \in E \text{ existiert} \\ 0 & \text{sonst} \end{cases}$$

Im allgemeinen wird die Adjazenzmatrix dünn besetzt sein.

Beispiel 3.16: Adjazenzmatrix für den Graphen aus Beispiel 3.15:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	0	1
C	0	0	0	1	0
D	1	0	1	0	0
E	0	0	1	1	0

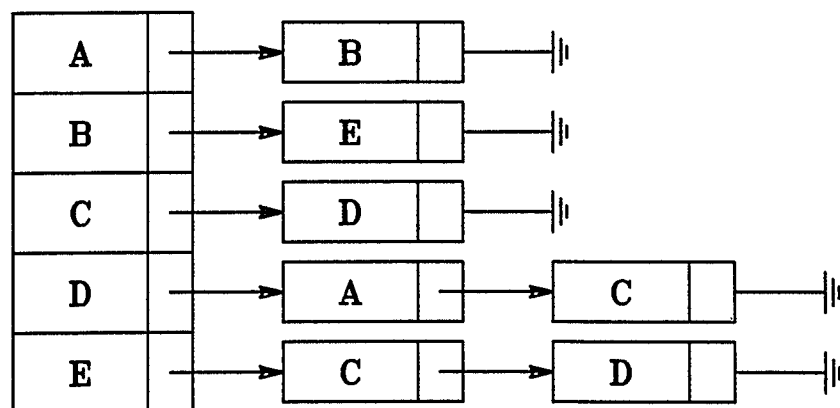
Bemerkung:

- Bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch.
- Der Speicherbedarf beträgt $|V|^2$ Speicherplätze. Bei ungerichteten Graphen ist eine Verringerung des Speicherplatzes möglich, wenn die Symmetrie ausgenutzt wird.
- Für verschiedene Graphenprobleme reicht die Information, ob eine Kante in einem Graphen enthalten ist, nicht aus. Für gewichtete Graphen können deshalb die Gewichte der Kanten in eine *bewertete Adjazenzmatrix* gebracht werden. Anstelle der Einsen werden dann die Gewichte der Kanten in die Matrix eingetragen.

2. Adjazenzlisten (Nachbarschaftslisten)

Bei der Speicherung von Graphen in Adjazenzlisten wird eine Liste aller Knoten verwendet, deren Elemente auf den Anfang einer verketteten Liste zeigen, in der alle Knoten stehen, die zu dem entsprechenden Knoten adjazent sind.

Beispiel 3.17: Adjazenzlisten eines Graphen (vgl. Beispiel 3.15)



Bemerkung:

Der Speicherplatzbedarf ist proportional zu $|V| + |E|$. Wegen $|E| \ll |V|^2$ wird der Speicherplatz der Adjazenzlisten damit im allgemeinen sehr viel geringer sein als der der Adjazenzmatrix.

Aber: Der Aufbau der Liste und der Zugriff auf die Listenelemente benötigt wesentlich mehr CPU-Zeit!

3.4 Bäume

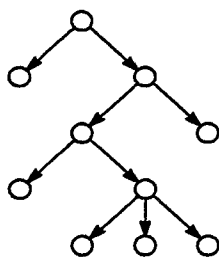
Bei Bäumen handelt es sich um Graphen mit speziellen Eigenschaften.

Definition 3.16: Baum

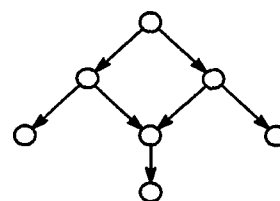
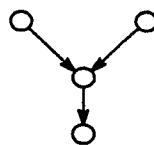
Ein gerichteter Baum ist ein gerichteter Graph $G = (V, E)$ mit folgenden Eigenschaften:

Es existiert genau ein Knoten $v \in V$ (Wurzel des Baumes, root) derart, daß für alle Knoten v_i genau ein Weg von v nach $v_i \in V$ existiert.

Beispiel 3.18: Unterscheidung von Bäumen und Graphen



a) Baum



b) Allgemeine Graphen, keine Bäume

Folgerung:

In einem Baum existieren keine Kreise (cycle).

Bemerkung:

- Bei einem Baum handelt es sich um eine hierarchische Datenstruktur.
- Auf den Knoten eines Baumes ist eine partielle Ordnung definiert, da alle Wege von der Wurzel ausgehen.
 - A heißt *Vorgänger* von B, wenn A auf einem Weg von der Wurzel zu B liegt. B heißt entsprechend *Nachfolger* von A.
 - A heißt *Vater* von B, wenn $(A,B) \in E$, entsprechend heißt B *Sohn* von A.
 - Haben B und C denselben Vater, so heißen sie *Brüder*.
 - Knoten, die keinen Nachfolger haben, heißen *Blätter* (leaf).
 - Knoten, die einen oder mehrere Nachfolger haben, heißen *innere Knoten*.
- Ein Knoten S mit allen seinen Nachfolgern wird *Teilbaum* eines Baumes T genannt.

Definition 3.17: Level eines Knotens

Jeder Knoten in einem Baum liegt auf einem bestimmten Level (Ebene).

1. Das Level der Wurzel ist 0.
2. Das Level eines Knotens ist gleich dem um 1 inkrementierten Level seines Vaters.

Definition 3.18: Höhe eines Baumes

Die Höhe eines Baumes ist gleich dem maximalen Level, welches mit einem beliebigen Knoten des Baumes assoziiert ist.

3.4.1 Binärbäume

Von besonderer Bedeutung sind geordnete Bäume, bei denen jeder Knoten maximal zwei Söhne hat. Diese Bäume werden als Binärbäume bezeichnet.

Definition 3.19: Binärbäume

Bäume, bei denen jeder Knoten höchstens zwei Söhne hat, heißen Binärbäume. Hat jeder Knoten entweder keinen oder zwei Söhne und die Blätter liegen auf dem gleichen Level, so heißt der Baum *voller Binärbäum*. Ist die Reihenfolge der Söhne durch die Indizes eindeutig festgelegt (T_1 bezeichnet linken Sohn, linken Teilbaum;

T_2 bezeichnet rechten Sohn, rechten Teilbaum), so handelt es sich um einen *geordneten Binärbaum*.

Bemerkung:

Wird die Wurzel als ausgezeichneter Knoten des Baumes eindeutig festgelegt, so kann man von einem gerichteten Graphen zu einem ungerichteten Graphen übergehen.

Im folgenden werden nur noch geordnete Binärbäume (als ungerichtete Graphen) betrachtet.

Durchlaufen von Bäumen

1. Es gibt $n!$ Möglichkeiten, eine Struktur mit n Elementen zu durchlaufen.
2. Für einen binären Baum gilt: Es gibt $3! = 6$ Möglichkeiten, einen solchen Baum (bestehend aus der Wurzel sowie dem rechten und linken Teilbaum) **geordnet zu durchlaufen**. (W = Wurzel, L = Linker Teilbaum, R = Rechter Teilbaum)
 - a. W L R $\leftarrow$ Preorder
 - b. W R L
 - c. R W L
 - d. R L W
 - e. L W R $\leftarrow$ Inorder
 - f. L R W $\leftarrow$ Postorder
3. Mögliche Reihenfolgen (der Klammerbegriff gibt den Zeitpunkt an, in dem der Knoten bearbeitet wird):
 - a. Der Knoten wird betrachtet. (Preorder)
 - b. Alle Knoten in seinem linken Teilbaum werden betrachtet.
 - c. Der Knoten wird beim zweiten Besuch betrachtet. (Inorder)
 - d. Alle Knoten in seinem rechten Teilbaum werden betrachtet.
 - e. Der Knoten wird beim dritten Besuch betrachtet. (Postorder)

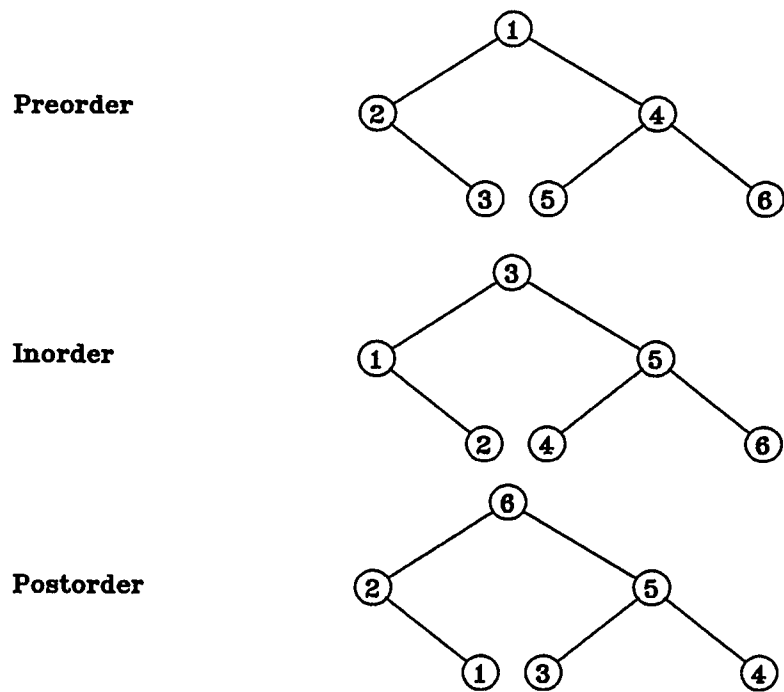


Abb. 25. Durchlaufstrategien eines Baumes: Die Numerierung gibt die Reihenfolge der Bearbeitung an

Beispiel 3.19:

Speicherung eines arithmetischen Ausdrucks in einem Baum, wobei die inneren Knoten den Operatoren und die Blätter den Operanden entsprechen (siehe Abb. 26 und Abb. 27).

Bemerkung:

Bei der üblichen Infix-Notation sind im allgemeinen Klammern erforderlich, um die Baumstruktur eindeutig wiedergeben zu können.

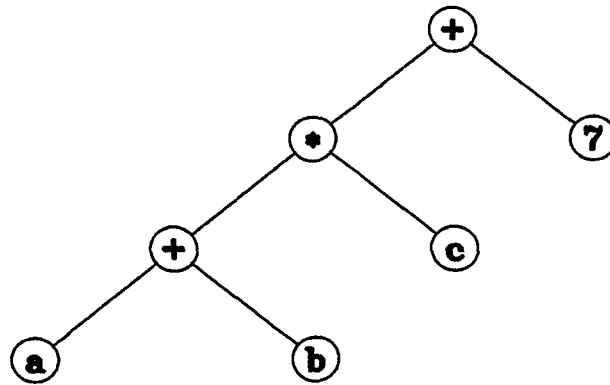


Abb. 26. Speicherung des Infix-Ausdruck $(a + b) * c + 7$:
 Prefix-Notation für den Baum: $+ * + a b c 7$

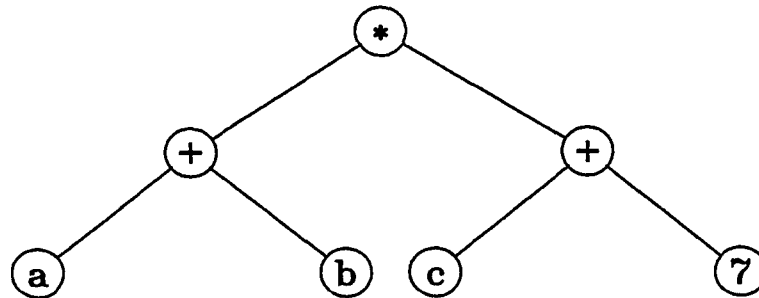


Abb. 27. Speicherung des Infix-Ausdruck $(a + b) * (c + 7)$:
 Prefix-Notation für den Baum: $* + a b + c 7$

Folgende Datentypen werden für den Binärbaum vereinbart:

```

TYPE order      = (preorder, inorder, postorder);
   relative     = (rootv, leftchild, rightchild, parent);
   status       = RECORD
                       size                : INTEGER; (* number of nodes *)
                       height              : INTEGER;
                       average_path_length REAL;
   END; (* status *)
  
```

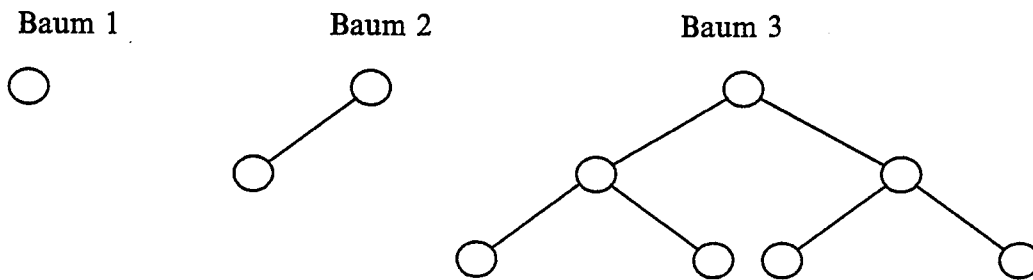


Abb. 28. Beispielbäume

	Baum 1	Baum 2	Baum 3
size	1	2	7
height	0	1	2
average_path_length	0	$\frac{0+1}{2} = \frac{1}{2}$	$\frac{(0+1+1+2+2+2+2)}{7} = \frac{10}{7}$

Tab. 4. Statuswerte für die Bäume aus Abb. 28

Definition 3.20: Abstrakter Datentyp *Binärbaum*

1. Elemente

Die Elemente eines Binärbaumes heißen Knoten, und jeder Knoten enthält eine Datenkomponente *e*.

2. Struktur

Ein Binärbaum ist entweder leer oder ein Knoten (Wurzel) zusammen mit zwei Binärbäumen. Diese zwei Binärbäume sind disjunkt voneinander sowie von der Wurzel und werden linker bzw. rechter Teilbaum der Wurzel genannt.

Ein Binärbaum hat eine hierarchische Struktur. Immer dann, wenn der Baum nicht leer ist, gibt es eine eindeutige Wurzel. Jeder Knoten (außer der Wurzel) hat genau einen Vorgänger und keinen, einen oder zwei Nachfolger. Ein Sohn ist entweder ein rechter oder ein linker Sohn eines Knotens.

3. Operationen

Konvention: Alle Operationen (außer *create*) setzen die Existenz eines Binärbaumes voraus. Der *current_node* bezeichnet den aktuell bearbeiteten Knoten im Baum. Manchmal ist es bei der Formulierung der Post-condition notwendig, sich auf den *current_node* oder den gesamten Baum zu beziehen, die vor der Ausführung aktuell waren. Dies geschieht durch die Bezeichnung *c_pre* und *t_pre*.

rootv wird verwendet, um den Knoten Wurzel von dem Pointer **root** auf die Wurzel zu unterscheiden.

a. *traverse* (*traverse_order* : *order*; *procedure process* (*p* : *node*))

pre Der Baum ist nicht leer.

post Jeder Knoten des Baumes wurde genau einmal bearbeitet. Die Reihenfolge, in der die Knoten bearbeitet werden, ist durch den Parameter *traverse_order* festgelegt.

case *traverse_order* of

preorder: Jeder Knoten wird vor der Verarbeitung seiner Teilbäume bearbeitet.

inorder: Jeder Knoten wird nach der Verarbeitung seines linken Teilbaums und vor der Verarbeitung seines rechten Teilbaums bearbeitet.

postorder: Jeder Knoten wird nach der Verarbeitung seiner Teilbäumen bearbeitet.

Die Art der Verarbeitung eines Knotens (bzw. seiner Daten) wird durch die Prozedur *process* festgelegt.

b. *insert* (*e* : *stdelement*; *rel* : *relative*; *var fail* : *BOOLEAN*)

pre Entweder ist *rel* gleich **rootv**, und der Baum ist leer, oder *rel* ist ungleich **rootv**, und der Baum ist nicht leer.

post Ein neuer Knoten mit dem Datenelement *e* kann in Abhängigkeit vom Wert *rel* im Baum enthalten sein. Falls der Knoten im Baum enthalten ist, ist er der *current_node*.

case *rel* of

rootv: *e* ist Element der Wurzel des Baumes, und *fail* ist false.

leftchild:

IF *c_pre* hat keinen linken Sohn
THEN *c_pre* hat einen linken Sohn, dieser Knoten enthält *e* als Datenelement, und *fail* ist false

ELSE *fail* ist true.

rightchild:

IF *c_pre* hat keinen rechten Sohn
THEN *c_pre* hat einen rechten Sohn, dieser Knoten enthält *e* als Datenelement, und *fail* ist false

ELSE *fail* ist true.

parent: fail ist true.

c. *delete_sub*

pre Der Baum ist nicht leer.

post Der Teilbaum von t_pre, dessen Wurzel c_pre ist, ist aus dem Baum entfernt. Der neue current_node ist die Wurzel des Baumes.

d. *retrieve (var e : stdelement)*

pre Der Baum ist nicht leer.

post e enthält das Datenelement von c_pre.

e. *update (e : stdelement)*

pre Der Baum ist nicht leer.

post c_pre enthält e als sein Datenelement.

f. *find(rel :relative; var fail: BOOLEAN)*

pre Der Baum ist nicht leer.

post current_node ist in Abhängigkeit von rel wie folgt definiert:

case rel of

rootv: current_node ist die Wurzel, und fail ist false.

leftchild:

IF c_pre hat linken Sohn

THEN current_node ist der linke Sohn, fail ist false

ELSE fail ist true.

rightchild:

IF c_pre hat rechten Sohn

THEN current_node ist der rechte Sohn, fail ist false

ELSE fail ist true.

parent:

IF c_pre hat Vater

THEN current_node ist der Vater, fail ist false

ELSE fail ist true.

g. *characteristics (var st : status)*

post st enthält die Anzahl der Knoten, die Höhe des Baumes und die durchschnittliche Pfadlänge von der Wurzel zu einem Blatt.

h. *empty* : BOOLEAN

post

IF	Baum nicht leer
THEN	<i>empty</i> ist false.
ELSE	<i>empty</i> ist true.

i. *create*

post Ein leerer binärer Baum existiert.

Darstellung von Binärbäumen im Rechner

1. Inhalt des Knotens
2. Pointer auf den linken Sohn
3. Pointer auf den rechten Sohn
4. Ein Pointer *root* zeigt auf die Wurzel des Baumes.

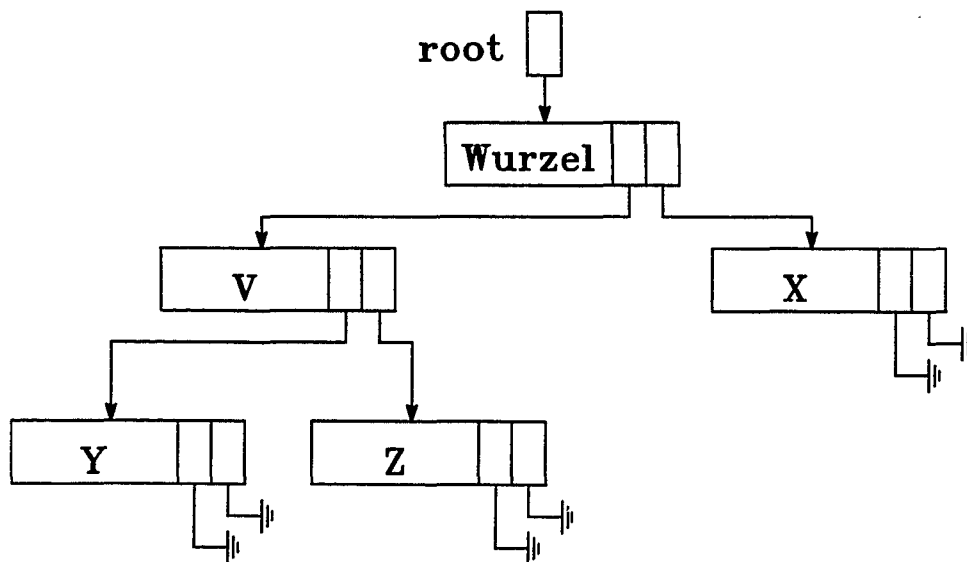


Abb. 29. Baumdarstellung mit Pointer

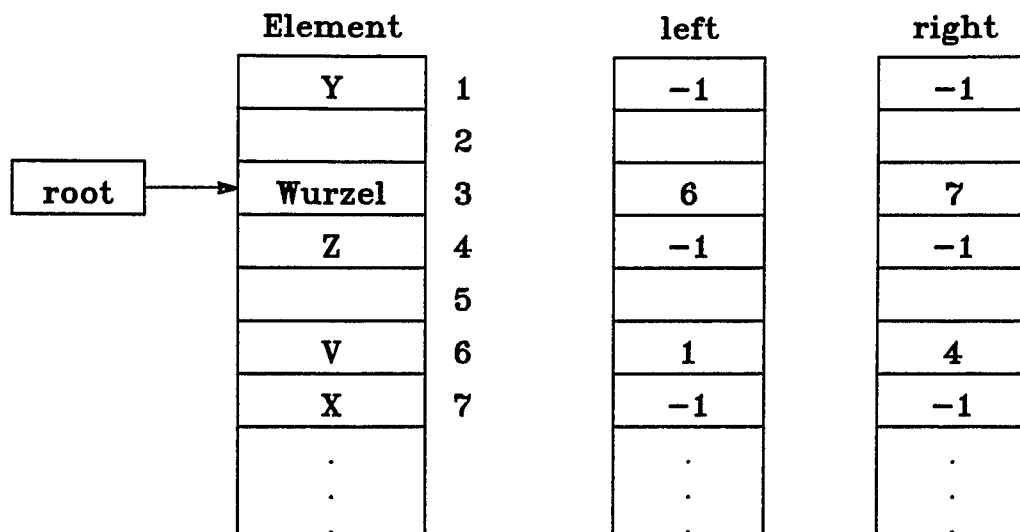


Abb. 30. Baumdarstellung mit statischen Feldern

Realisation eines Binärbaumes

1. Repräsentation

```

TYPE relative      = (rootv, leftchild, rightchild, parent);
   node_pointer = ↑node;
   node        = RECORD
                       element      : stdelement;
                       left         : node_pointer;
                       right        : node_pointer;
                   END; (* node *)

   order        = (preorder,inorder,postorder);
   status       = RECORD
                       size         : INTEGER; (* number of nodes *)
                       height       : INTEGER;
                       average_path_length REAL;
                   END; (* status *)

VAR root        : node_pointer;
    current_node node_pointer;

```

2. Implementation

External Procedures

- a. aus der Stack-Realisierung, siehe Kapitel „3.2.3 Stapel (stacks)“:

```
PROCEDURE push ( p : node_pointer);
PROCEDURE pop (VAR p : node_pointer);
```

- b. eine Prozedur zur Bearbeitung der Knoten:

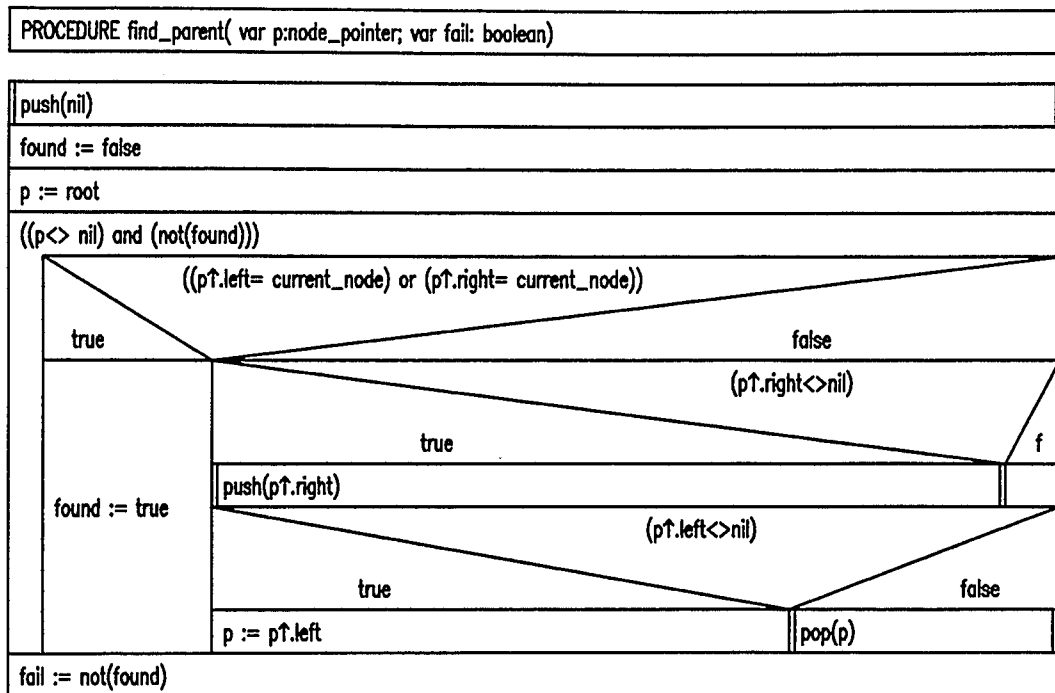
```
PROCEDURE process (VAR p : node_pointer; level : INTEGER);
```

PROCEDURE newnode(var p:node_pointer)
new(p)
p↑.left := nil
p↑.right := nil

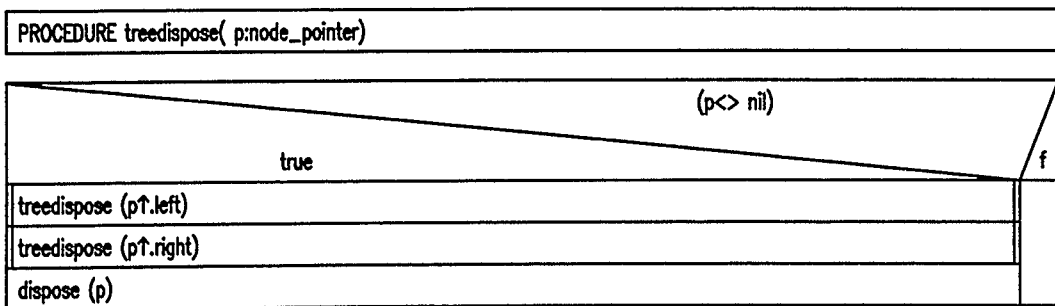
Prozedur newnode

PROCEDURE insert(e: stdelement; rel:relative; var fail:boolean)										
fail := false										
newnode(p)										
p↑.element := e										
root v	rel									
root :=p	leftchild			rightchild			parent	oth		
	current_node↑.left = nil			current_node↑.right= nil			fail := true			
	true	false		true	false					
	current_node↑.left:= p		fail := true	current_node↑.right:= p		fail := true				
			fail							
true				false						
dispose(p)				current_node := p						

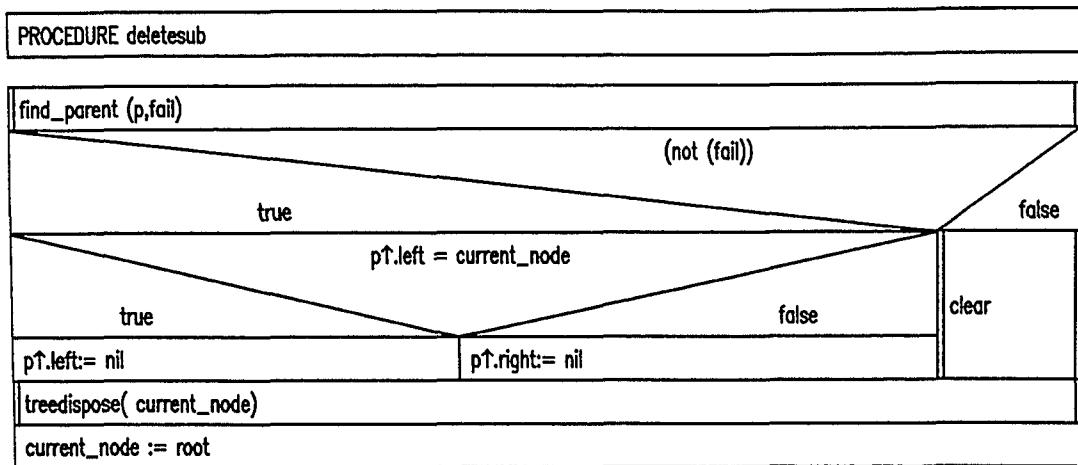
Prozedur insert



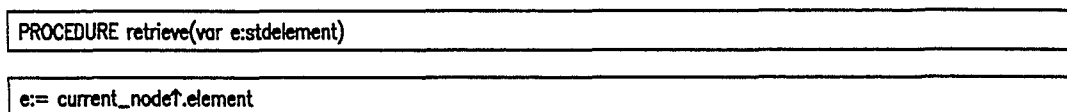
Prozedur find_parent



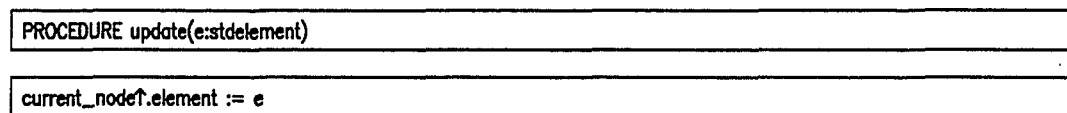
Prozedur treedispense



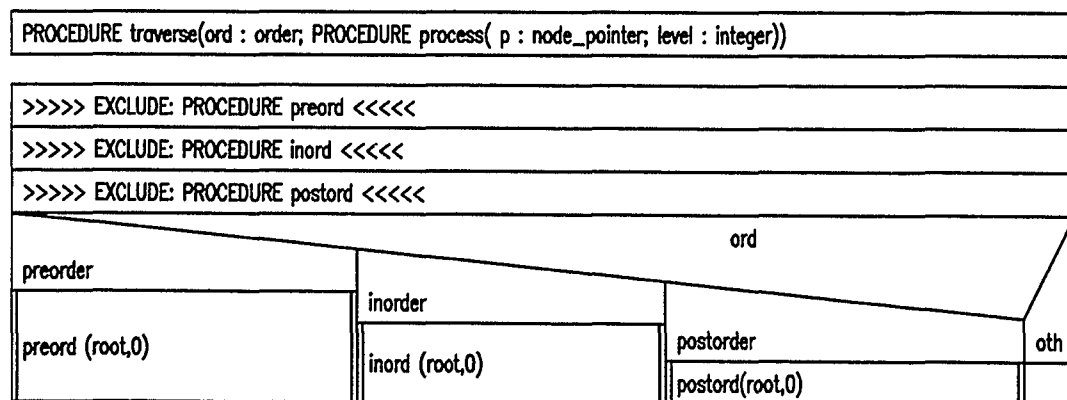
Prozedur deletesub



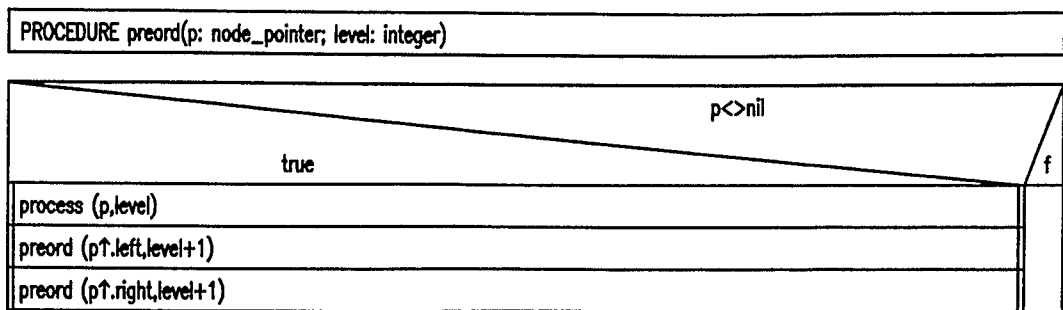
Prozedur retrieve



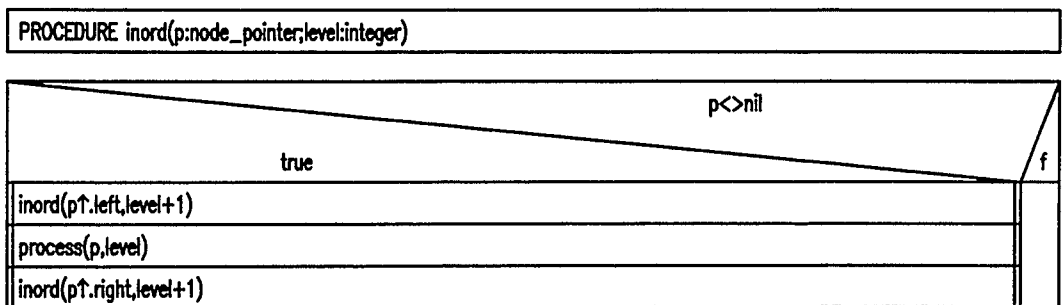
Prozedur update



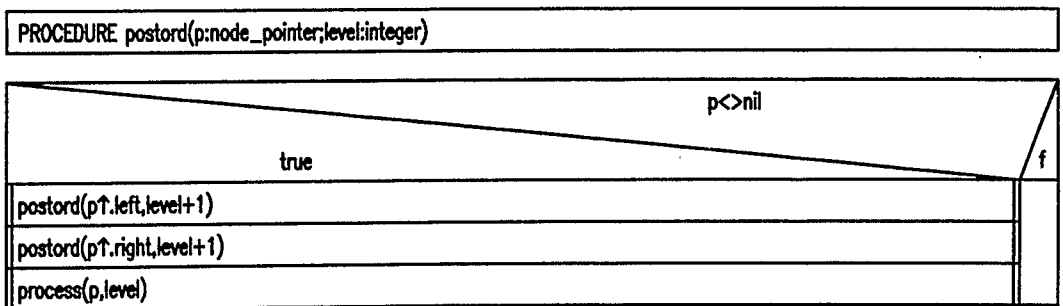
Prozedur traverse



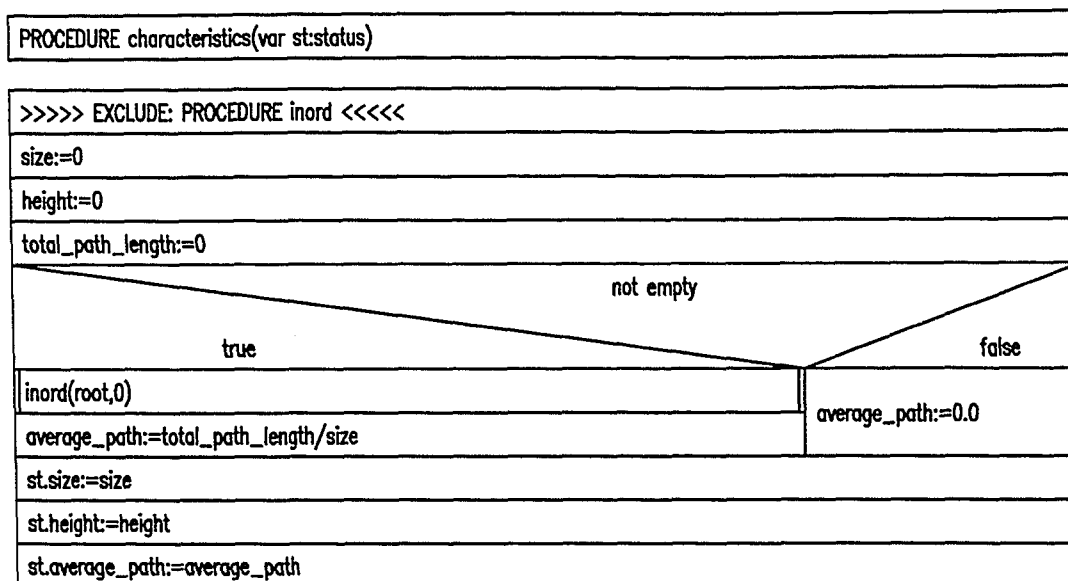
Prozedur preord



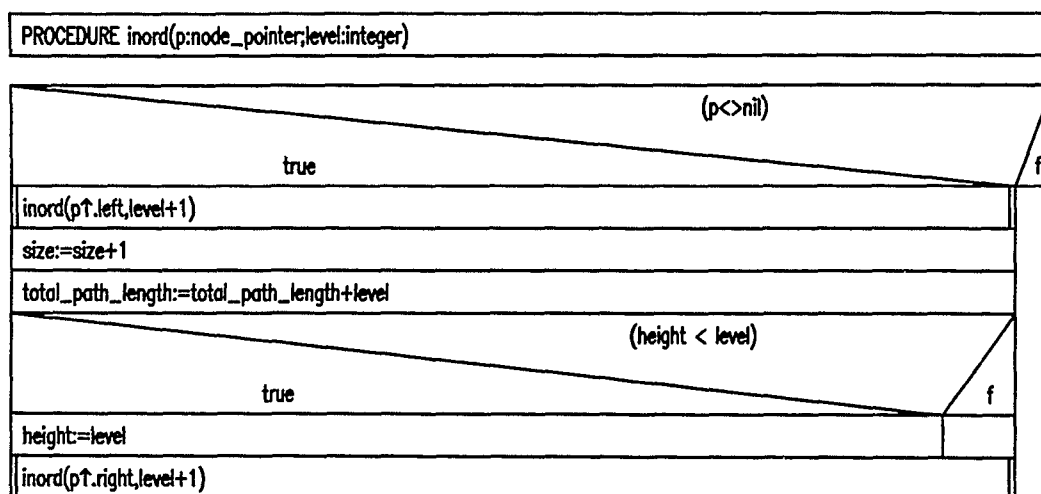
Prozedur inord



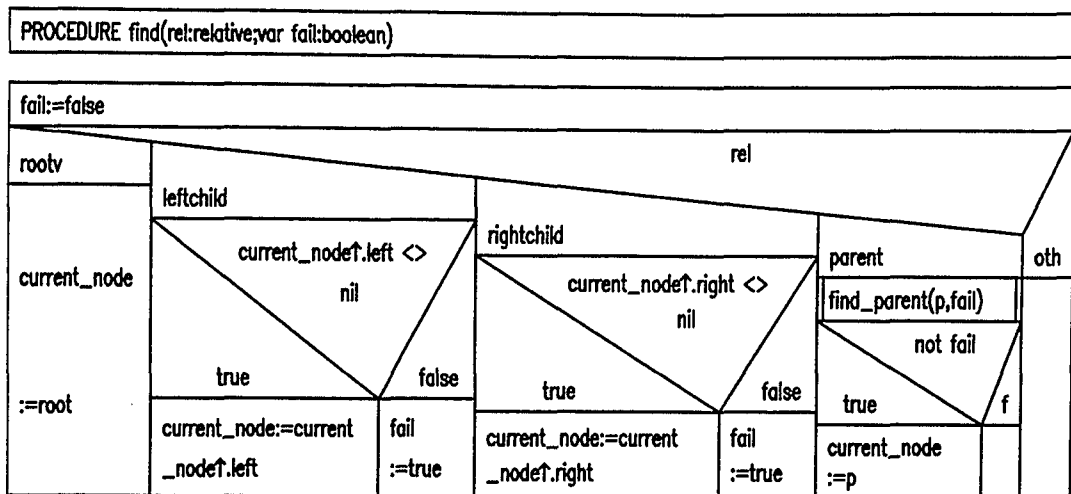
Prozedur postord



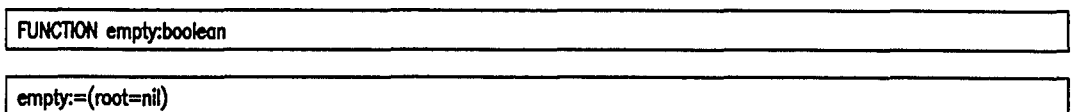
Prozedur characteristics



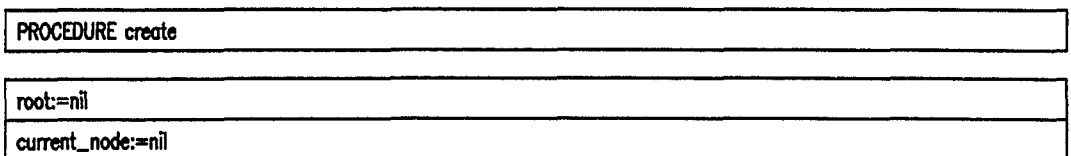
Prozedur inord (lokal zu characteristics)



Prozedur find



Funktion empty



Prozedur create

3.4.2 Binäre Suchbäume als Anwendung von Binärbäumen

Ein binärer Suchbaum ist ein Binärbaum mit der Eigenschaft, daß für jeden Knoten N die folgenden Aussagen wahr sind:

1. Falls sich der Knoten L im linken Teilbaum vom Knoten N befindet, so ist der Wert des Schlüssels von Knoten L kleiner als der Wert des Schlüssels von Knoten N.
2. Falls sich der Knoten R im rechten Teilbaum vom Knoten N befindet, so ist der Wert des Schlüssels von Knoten R größer als der Wert des Schlüssels von Knoten N.

Vorteil: Man braucht, falls beim Einfügen eine spezielle Ordnung eingehalten wird, nicht alle Elemente zu durchsuchen, sondern kann sich auf eine Teilmenge des Baumes beschränken.

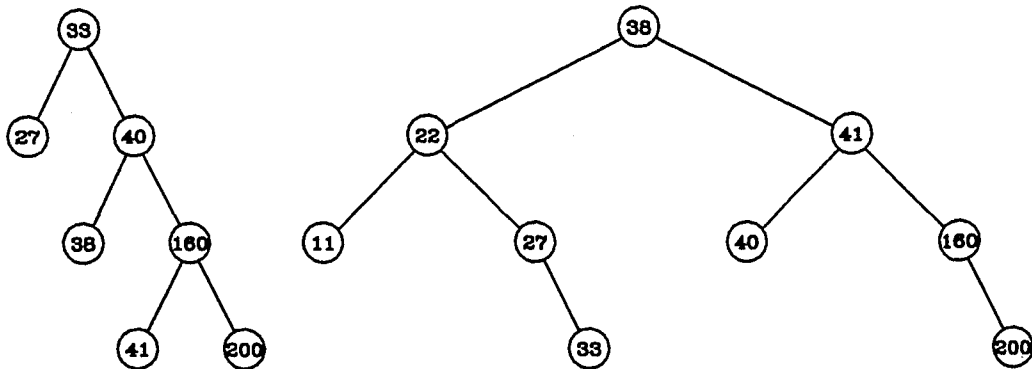


Abb. 31. Beispiele für binäre Suchbäume

Definition 3.21: Abstrakter Datentyp *binärer Suchbaum*

1. Elemente

Die Elemente eines binären Suchbaumes heißen Knoten; sie sind durch einen Schlüssel eindeutig identifiziert.

2. Struktur

Ein binärer Suchbaum ist ein Binärbaum mit der zusätzlichen Eigenschaft:

- Für alle Knoten im linken Teilbaum eines Knotens N gilt, daß sie bezüglich des Schlüssels kleiner sind als der Knoten N .
- Für alle Knoten im rechten Teilbaum eines Knotens N gilt, daß sie bezüglich des Schlüssels größer sind als der Knoten N .

3. Operationen

Hier werden vier wichtige Operationen spezifiziert. Die restlichen noch notwendigen Operationen entnehmen Sie bitte der Implementation der Definition 3.20.

a. *insert* (*e* : *stdelement*; var *fail* : *BOOLEAN*);

```
pre    Keine
post
      IF      t_pre enthält e nicht
      THEN    e ist Element des Baumes, und fail ist false
      ELSE    fail ist true
```

b. *delete_key* (*e* : *stdelement*; var *fail* : *BOOLEAN*)

```
post
      IF      t_pre enthält das Element e
      THEN    dieses Element ist nicht mehr im Baum enthalten, und
              fail ist false
      ELSE    fail ist true
```

c. *update*(*e* : *stdelement*)

```
pre    Der Baum ist nicht leer.
post    current_node enthält das Element e, und der Baum ist weiterhin
        ein binärer Suchbaum.
```

d. *findkey* (var *e* : *stdelement*; var *fail* : *BOOLEAN*)

```
pre    Der Baum ist nicht leer.
post
      IF      der Baum enthält einen Knoten mit dem Inhalt e
      THEN    dieser Knoten ist der current_node, und fail ist false
      ELSE    der current_node ist der Knoten, an dem ein Knoten
              mit dem Inhalt e als Sohn angehängt wäre, wenn e im
              Baum enthalten wäre, und fail ist true
```

Realisation eines binären Suchbaumes

1. Repräsentation

```

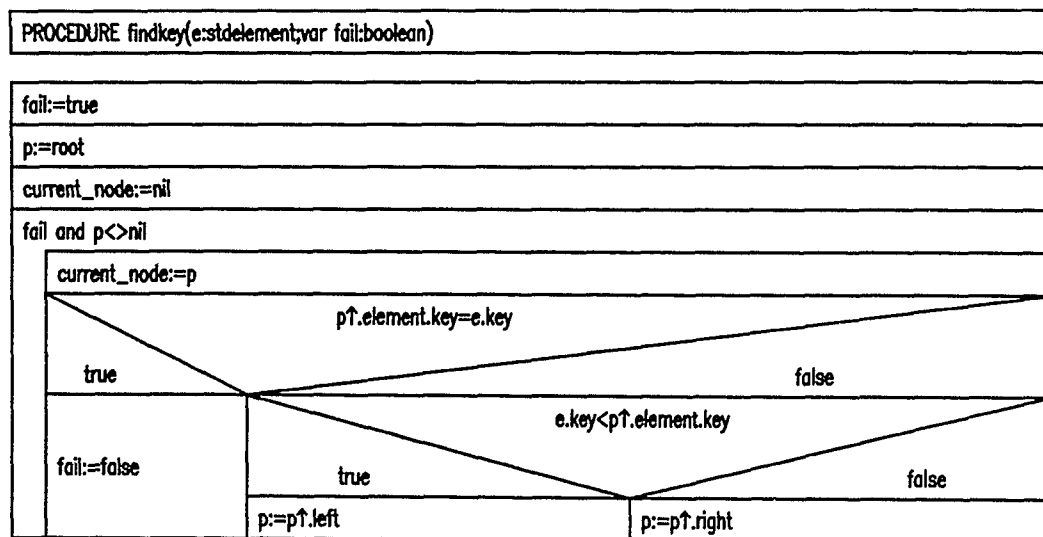
TYPE  relative      = (rootv, leftchild, rightchild, parent);
      node_pointer   = ↑node;
      node           = RECORD
                          element      : stdelement;
                          left         : node_pointer;
                          right        : node_pointer;
                      END; (* node *)

      order          = (preorder,inorder,postorder);

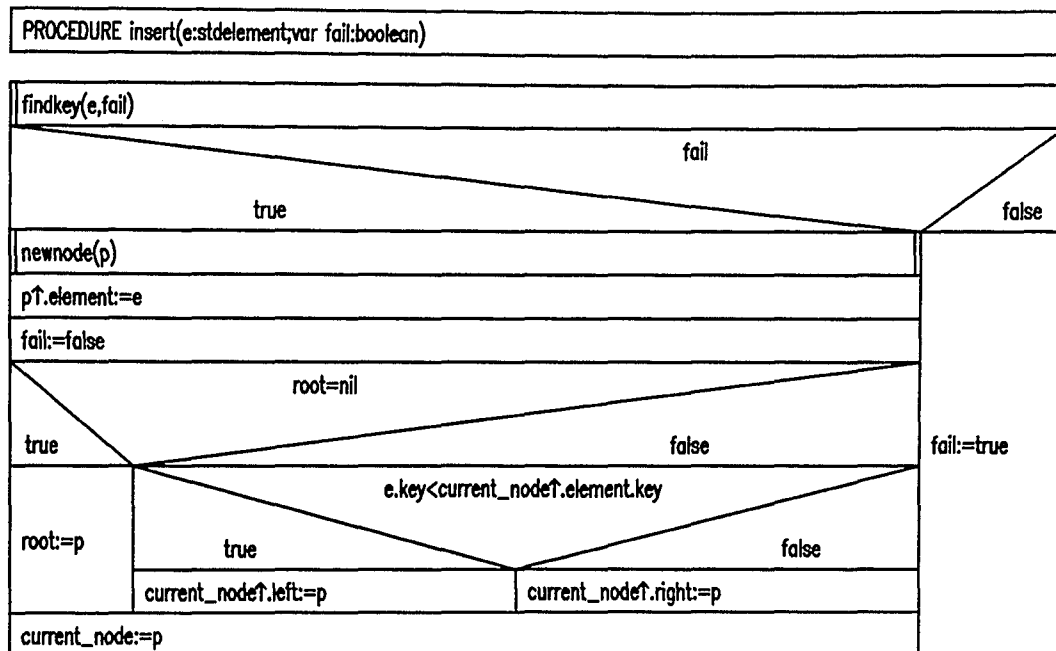
VAR   root          : node_pointer;
      current_node   : node_pointer;
  
```

2. Implementation

Hier werden nur die drei wichtigen Operationen findkey, insert und delete_key besprochen.

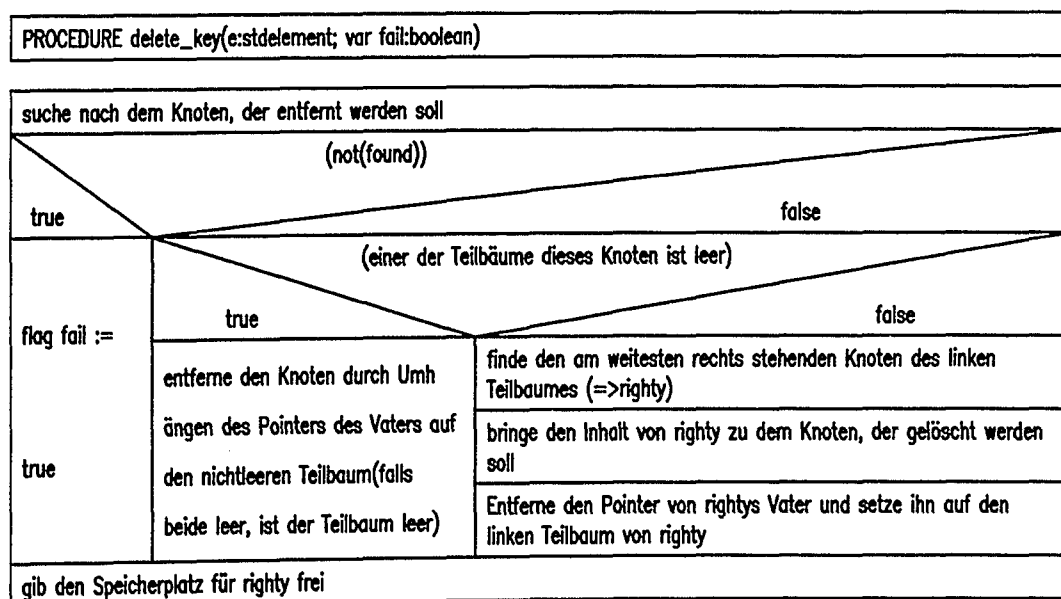


Prozedur findkey



Prozedur insert

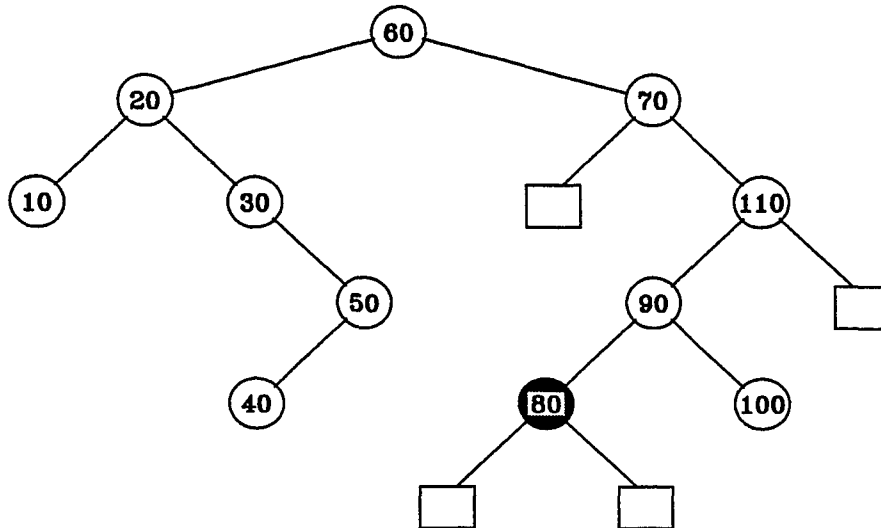
Die Operation delete_key ist eine Prozedur mit komplexer Aufgabenstellung und soll deshalb zunächst durch eine Prozedurskizze veranschaulicht werden.



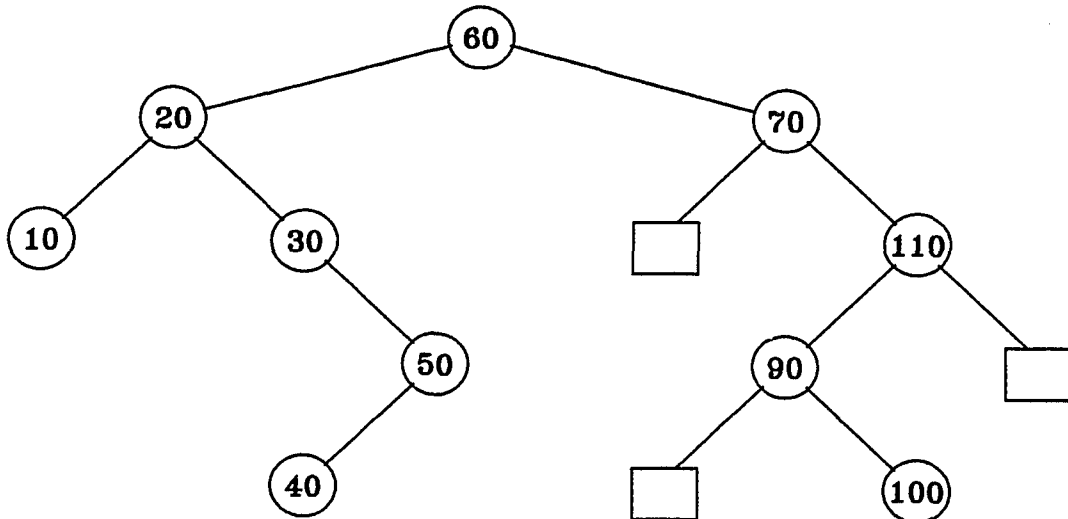
Prozedurskizze für die Operation delete_key

Beispiel 3.20: Löschen in einem binären Suchbaum

1. Das erste Beispiel zeigt das Entfernen des Knotens 80, der zwei leere Teilbäume hat. Der linke Pointer des Vaters wird entweder durch den linken oder rechten Pointer des zu löschenden Knotens ersetzt.

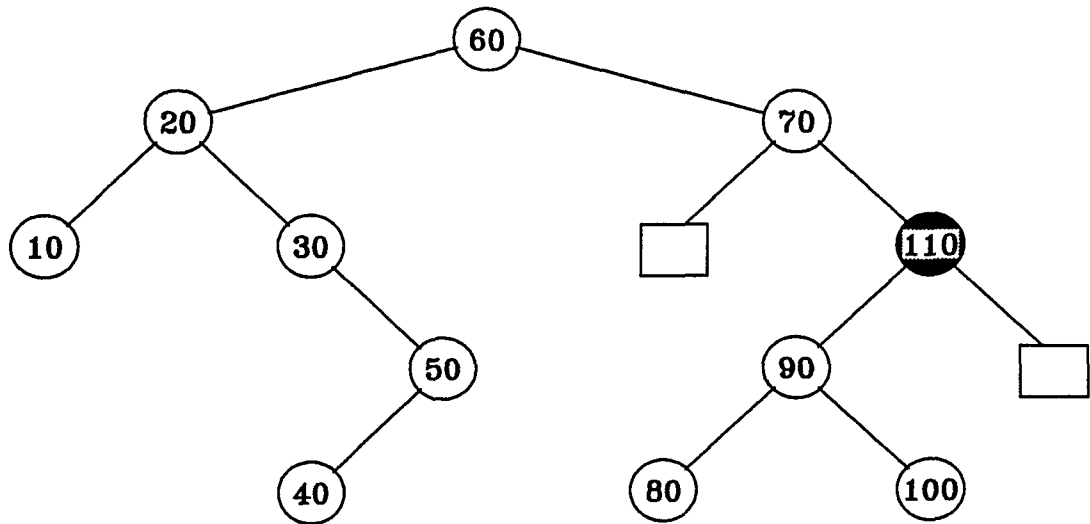


Darstellung 1a: Baum vor dem Löschen von 80

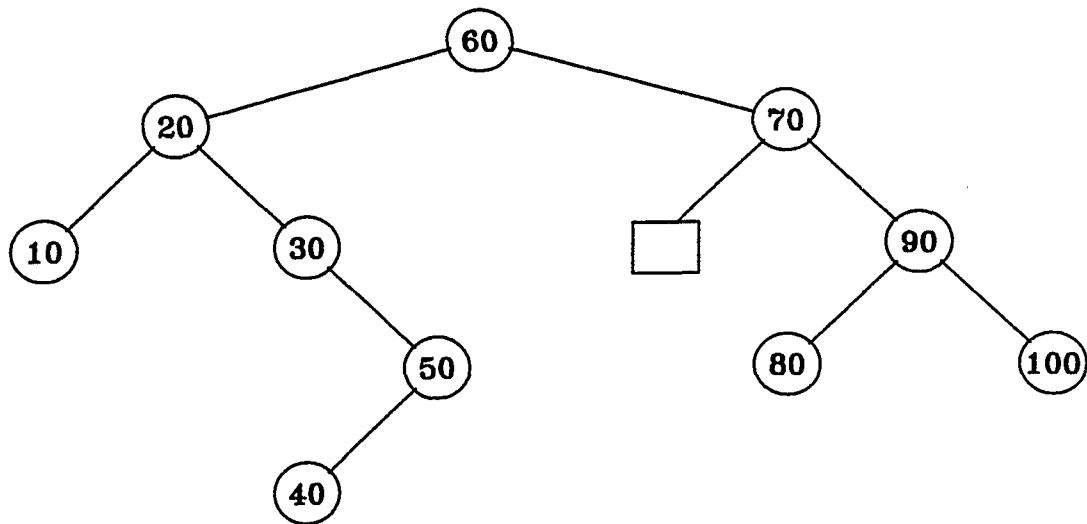


Darstellung 1b: Baum nach dem Löschen von 80

2. Das zweite Beispiel zeigt das Entfernen des Knotens 110 mit einem leeren rechten Teilbaum. Der Pointer des Vaters, d.h. der rechte Pointer des Knotens 70 wird durch den linken Pointer des zu löschenden Knotens 110 ersetzt.



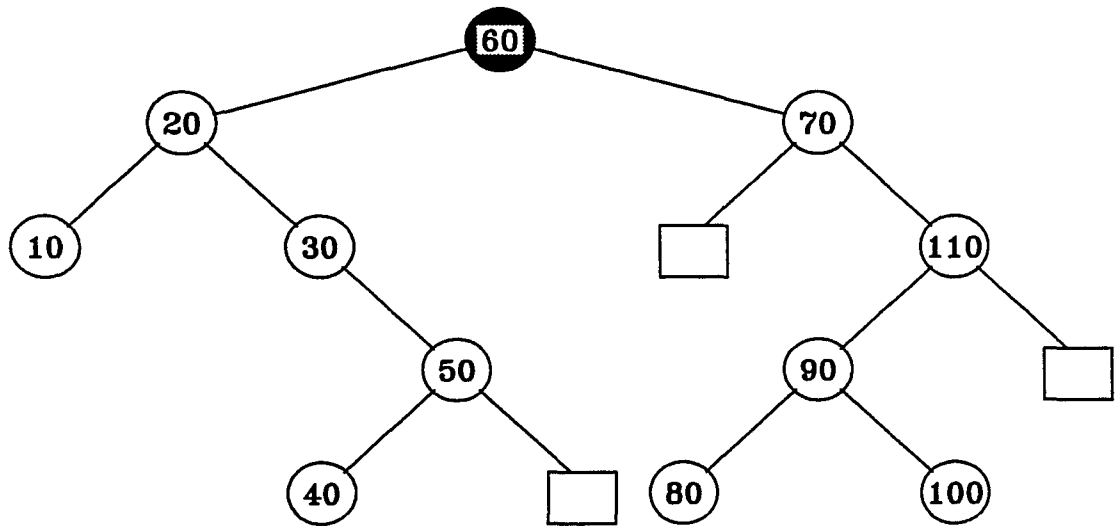
Darstellung 2a: Baum vor dem Löschen von 110



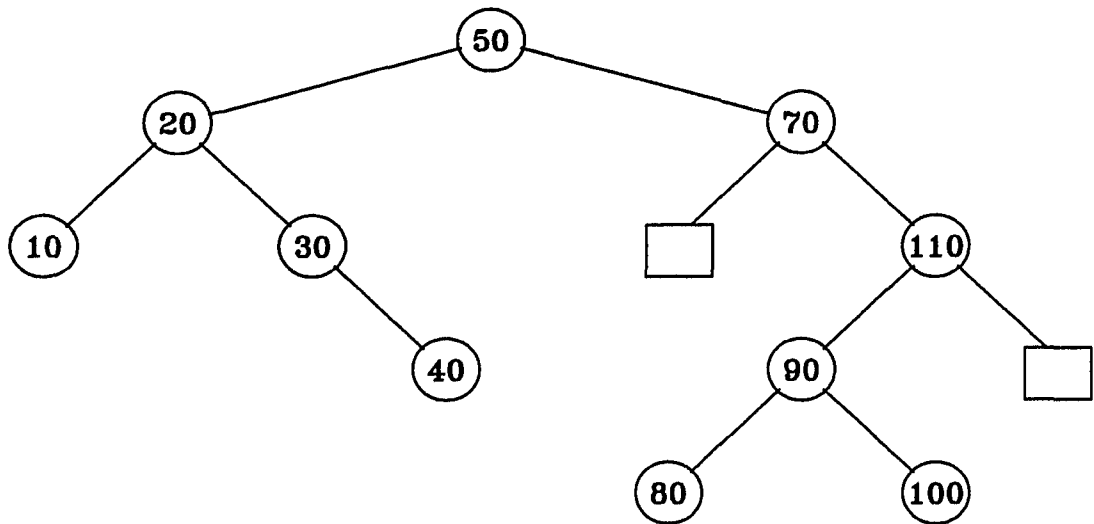
Darstellung 2b: Baum nach dem Löschen von 110

3. Das dritte Beispiel zeigt das Entfernen eines Knotens, der keinen leeren Teilbaum hat. Zwei verschiedene Möglichkeiten werden beschrieben.

- a. In dieser Realisierung wird das größte Element im linken Teilbaum des zu löschenden Knotens verwendet. Dieser Knoten ist der ganz rechts stehende Knoten im linken Teilbaum. Der zu löschende Knoten (60) wird mit diesem Knoten (50) vertauscht. Knoten 60 wird entfernt wie unter 2. beschrieben. Diese Möglichkeit wird in der Prozedur `delete_key` verwendet.

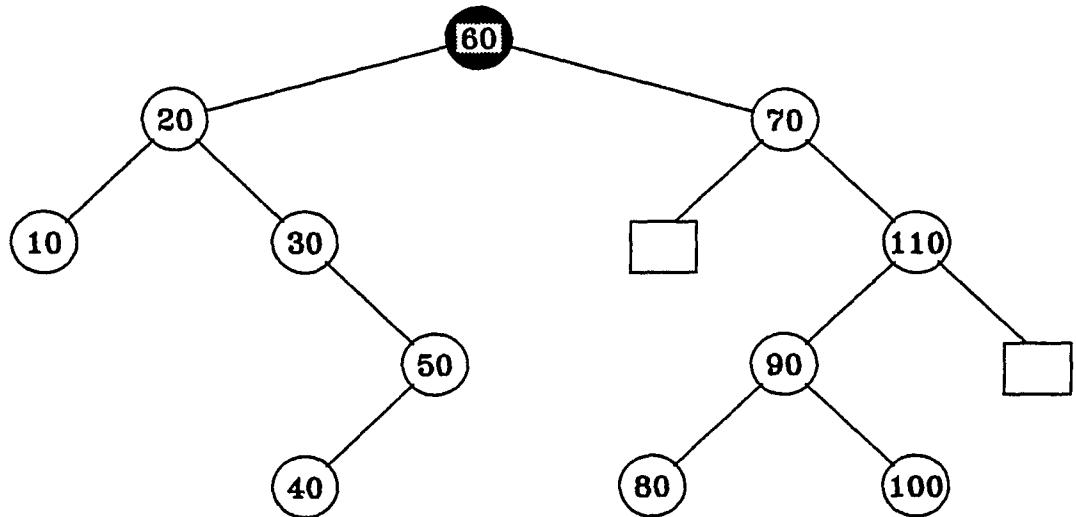


Darstellung 3a: Baum vor dem Löschen von 60

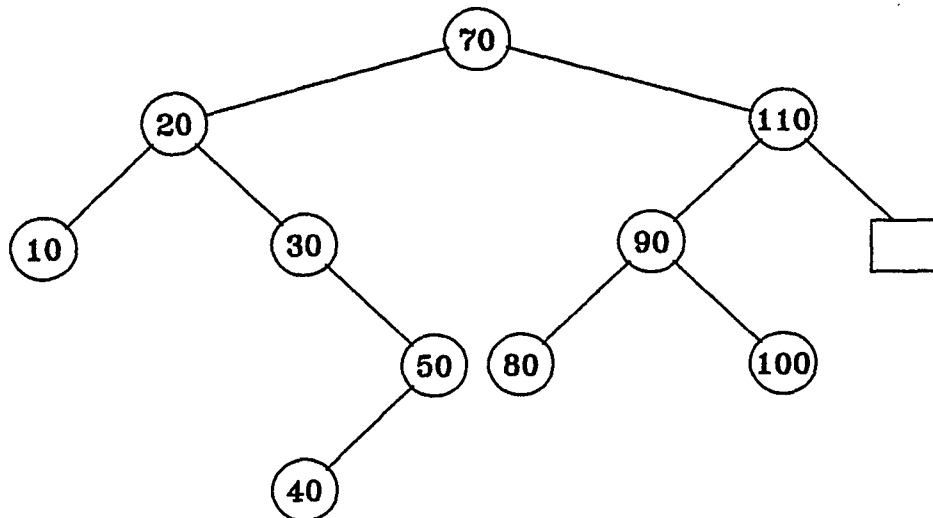


Darstellung 3b: Baum nach dem Löschen von 60

- b. Die zuletzt gezeigten Darstellungen skizzieren das Entfernen durch Austauschen des zu löschenden Knotens mit dem größten Element im linken Teilbaum. Daneben gibt es eine weitere Möglichkeit für das Entfernen eines Knotens. Der zu löschende Knoten (60) wird mit dem kleinsten Element im rechten Teilbaum (70) vertauscht.

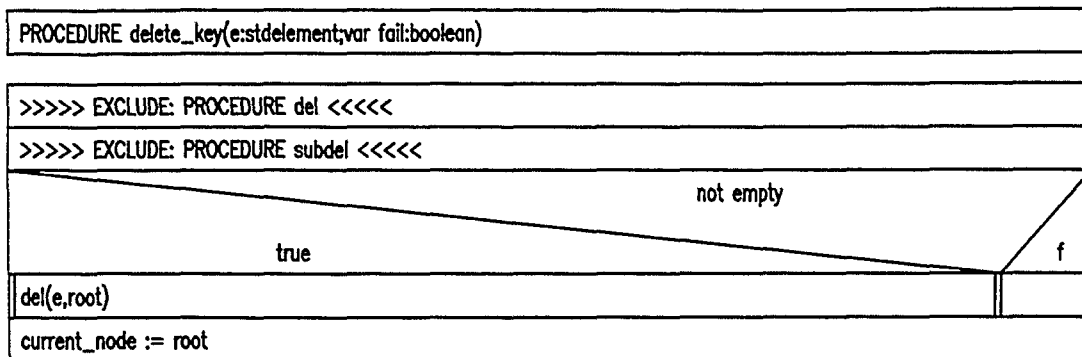


Darstellung 3c: Baum vor dem Löschen von 60

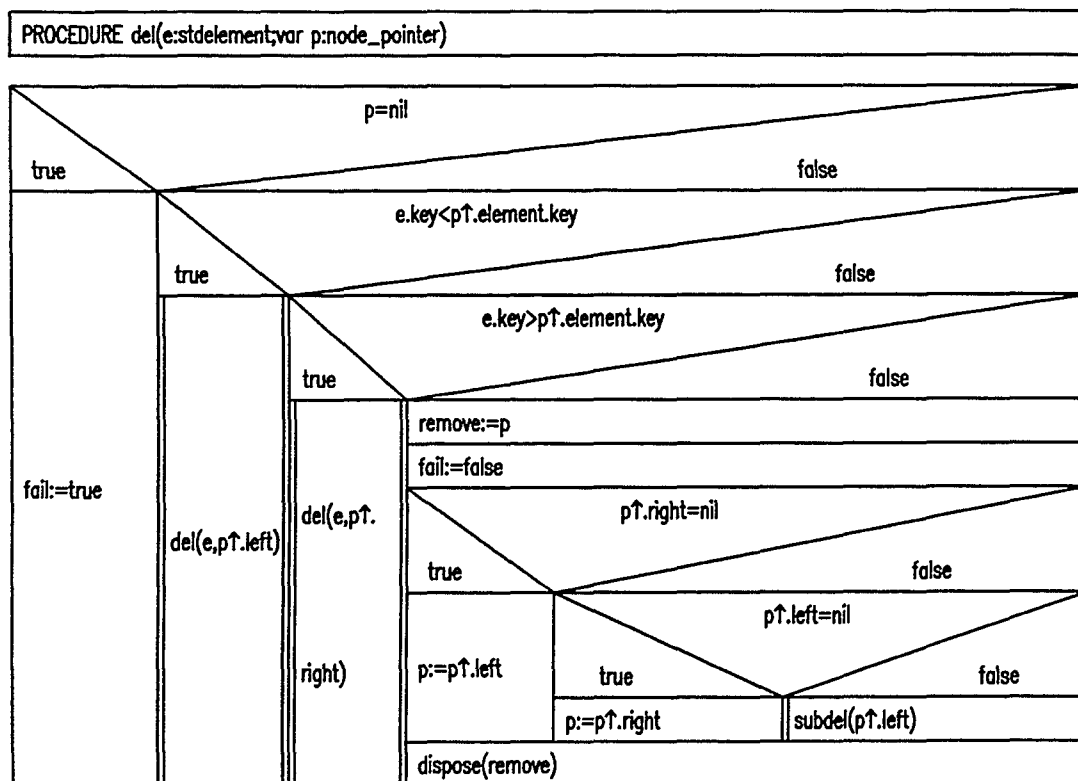


Darstellung 3d: Baum nach dem Löschen von 60

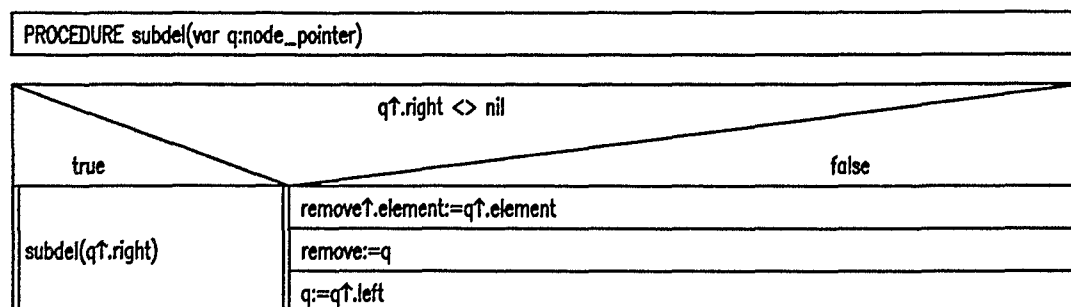
Im folgenden wird die vollständige Implementierung für die Operation `delete_key` angegeben.



Prozedur delete_key



Prozedur del (lokal zu der Prozedur delete_key)



Prozedur subdel (lokal zu der Prozedur delete_key)

Das Aktualisieren eines Datenelementes wird, wegen der einzuhaltenden Ordnung, durch das Löschen des entsprechenden Elementes und anschließendem Einfügen des neuen Elementes erreicht.

PROCEDURE update(e:stdelement)

delete_key(current_nodef.element,fail)
--

insert(e,fail)

Prozedur update

Beispiel 3.21: Einfügen in einen binären Suchbaum

1. create
2. insert (10)
3. insert (20)
4. insert (30)
5. insert (40)
6. insert (50)
7. insert (60)

Es entsteht ein degenerierter Baum (siehe linker Baum in Abb. 32).

1. create
2. insert (40)
3. insert (20)
4. insert (50)
5. insert (10)
6. insert (30)
7. insert (60)

(siehe rechter Baum in Abb. 32).

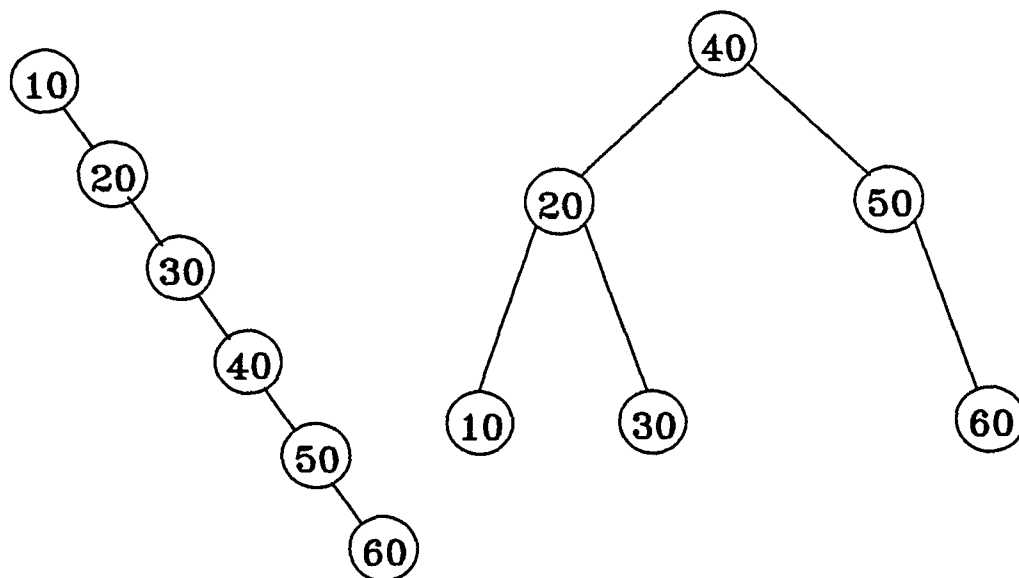


Abb. 32. Unterschiedliche Suchbäume mit gleichem Informationsgehalt

Probleme bei binären Suchbäumen:

Die Implementation von insert garantiert leider keine balancierte Verteilung der Knoten. In einem vollen Binärbaum der Höhe k sind $2^{k+1}-1$ Knoten enthalten. Dies ist nicht praxisgerecht für binäre Suchbäume, da eine beliebige, sich verändernde Anzahl von Knoten gespeichert werden muß. Gesucht sind binäre Bäume minimaler Höhe.

Definition 3.22: Minimaler Binärbaum

Ein Binärbaum heißt minimal genau dann, wenn kein Binärbaum existiert, der die gleiche Anzahl von Knoten, aber eine niedrigere Höhe hat.

Definition 3.23: Vollständiger Binärbaum

Ein vollständiger Binärbaum ist ein minimaler Binärbaum, in dem die Knoten auf dem untersten Level so weit wie möglich links stehen.

Bei einem vollständigen Binärbaum sind alle Ebenen bis auf die letzte voll besetzt. Dagegen ist bei einem vollen Binärbaum auch die unterste Ebene voll besetzt, d.h. ist der Baum voll, gilt: $n = 2^{k+1} - 1 \Rightarrow k = \log(n + 1) - 1$.

Sei die Suchlänge zu einem Knoten die Anzahl von Knoten, die bei der Suche nach diesem Knoten in einem binären Suchbaum inspiziert werden muß, bis der Knoten gefunden ist:

1. Suchen in einem vollen binären Suchbaum

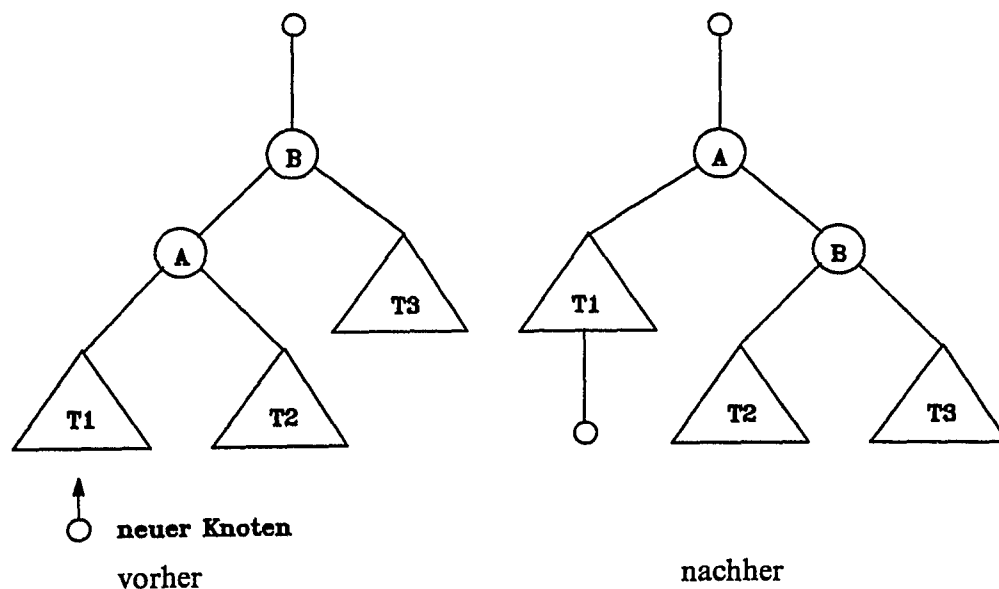
- worst-case $\log(n+1)$ entspricht $O(\log n)$
- average-case $\log(n+1) - 1$ entspricht $O(\log n)$

2. Ist der Baum degeneriert, d.h. auf jedem Level ist nur ein Knoten

- worst-case n entspricht $O(n)$
- average-case $\frac{n+1}{2}$ entspricht $O(n)$

Aus diesem Grund sorgt man dafür, daß Bäume höhenbalanciert sind. Dies geschieht durch geeignete Umstrukturierungsmaßnahmen, wie sie in Beispiel 3.22 skizziert sind.

Beispiel 3.22: Einfügen in einen höhenbalancierten Suchbaum



Bekannte Varianten von höhenbalancierten Bäumen sind die AVL-Bäume, die 2-3-Bäume und die B-Bäume. In diesem Manuskript werden die B-Bäume genauer vorgestellt.

3.4.3 B-Bäume

Ein B-Baum ist ein spezieller Baum, der günstige Eigenschaften zum Speichern und Suchen von Informationen besitzt. Der längste Weg zwischen der Wurzel und einem Blatt ist $O(\log_{d+1} n)$. Jeder Knoten kann mehrere Elemente enthalten und mehrere Söhne haben. Da B-Bäume höhenbalanciert sind, hat man sehr kurze Weglängen um eine Vielzahl von Elementen anzusprechen.

Definition 3.24: B-Baum

Jeder Knoten in einem B-Baum enthält d bis $2d$ Elemente, wobei d die Ordnung des Baumes angibt. Die Wurzel bildet die einzige Ausnahme, sie kann 1 bis $2d$ Elemente enthalten. Die Anzahl der Söhne in einem B-Baum ist entweder 0 oder um eins größer als die Anzahl der Elemente, die der Knoten enthält. Die Elemente in einem Knoten sind aufsteigend sortiert.

Jeder Knoten des B-Baumes kann aufgefaßt werden als $(p_0, k_1, p_1, k_2, p_2, \dots, k_n, p_n)$, wobei p_i ein Pointer auf den i -ten Sohn des Knotens ($0 \leq i \leq 2d$) und k_i ein Schlüssel ($1 \leq i \leq 2d$) ist. Für die Schlüssel innerhalb eines Knoten gilt $k_1 < k_2 < \dots < k_n$. Alle Schlüssel in dem Teilbaum, auf den p_0 zeigt, sind kleiner als k_1 . Für $1 \leq i < 2d$ liegen alle Schlüssel k des Teilbaumes, auf den p_i zeigt, im Bereich $k_i < k < k_{i+1}$. Alle Schlüssel in dem Teilbaum, auf den p_n zeigt, sind größer als k_n .

Definition 3.25: Abstrakter Datentyp *B-Baum*

1. Elemente

Die Elemente eines B-Baumes heißen Knoten; sie sind durch einen Schlüssel eindeutig identifiziert.

2. Struktur

Ein B-Baum ist ein geordneter Baum. Jeder Knoten, außer der Wurzel, hat einen eindeutigen Vater. Jeder Knoten, außer den Blättern, hat einen Sohn mehr als er Elemente enthält. Die Elemente in einem Knoten sind nach den Schlüsseln aufsteigend sortiert.

3. Operationen

Die Operationen sind denen eines Binärbaumes bzw. eines binären Suchbaumes sehr ähnlich.

a. *empty* und *characteristics* bleiben gleich.

- b. Die Operationen für einen binären Suchbaum *findkey*, *update*, *insert* und *delete_key* bleiben erhalten.
- c. Für *create* wird Binärbaum durch B-Baum ersetzt.
- d. *retrieve*: Das Element, das geholt werden soll, muß aus $2d$ Möglichkeiten spezifiziert werden.
- e. *delete_sub* ist nicht erlaubt.
- f. *find*: Es wird nicht nur der linke oder rechte Sohn angegeben, sondern es gibt $2d + 1$ mögliche Söhne.
- g. *traverse*: inorder ist nicht erlaubt.

Realisierung eines B-Baumes

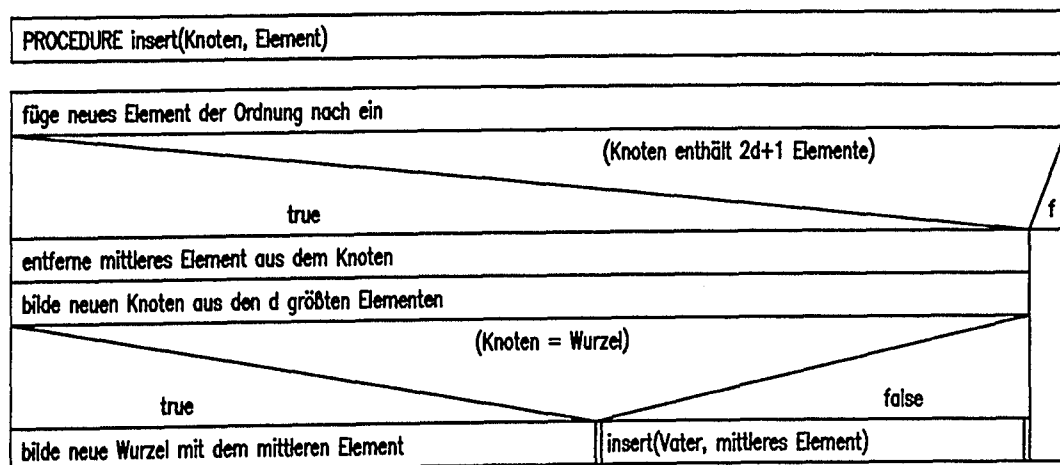
1. Repräsentation

```
CONST order      = ...;    (* Ordnung des B-Baumes *)
      maxnode    = 2 * order; (* max. Anzahl Elemente in einem Knoten *)
```

```
TYPE node_pointer = ↑node;
      node = RECORD
          parent : node_pointer;
          number_of_elements : 0..maxnode;
          elements : ARRAY [1..maxnode] OF stdelement;
          child    : ARRAY [1..maxnode] OF node_pointer;
      END; (* node *)
```

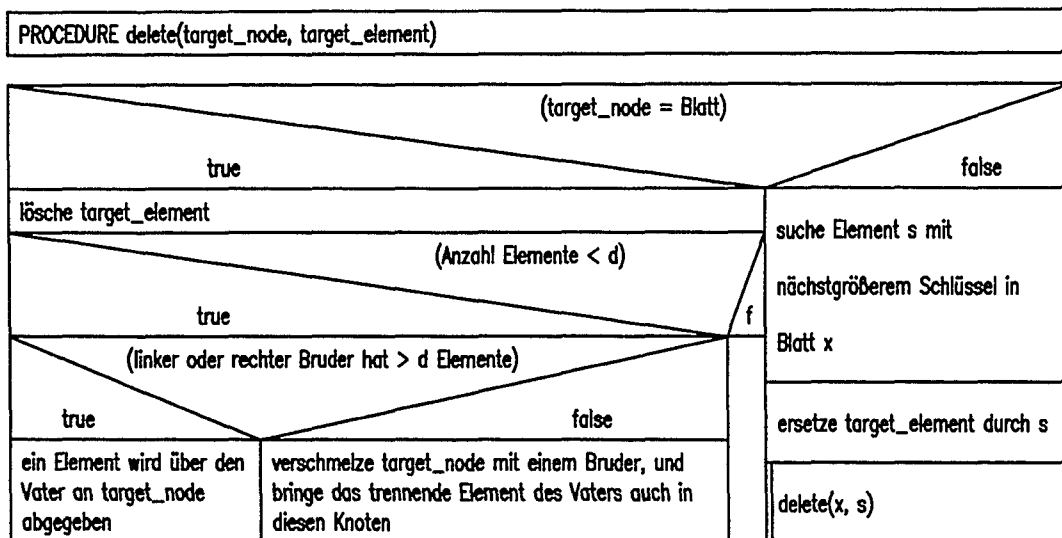
2. Implementation

Die genaue Implementation eines B-Baumes kann in [12] nachgelesen werden. Hier sollen nur die beiden wichtigen Operationen insert und delete veranschaulicht werden.



Prozedur insert

In der Prozedur delete bezeichnet *target_node* den Knoten, in dem das Element *target_element* gelöscht werden soll.

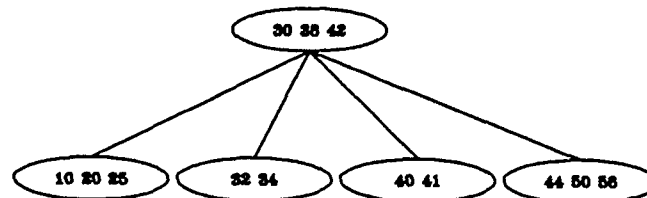


Prozedur delete

Beispiel 3.23: Einfügen und Löschen in einem B-Baum der Ordnung 2

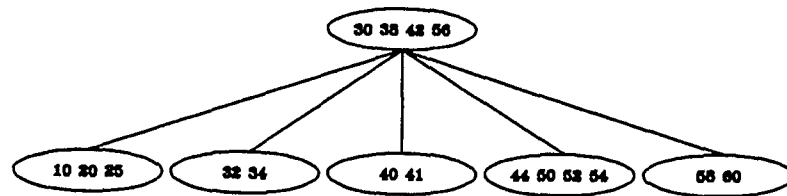
Dieses Beispiel soll die Operationen Einfügen und Löschen von Elementen in einem B-Baum veranschaulichen.

1. Zunächst sollen mehrere Elemente in den folgenden B-Baum eingefügt werden.



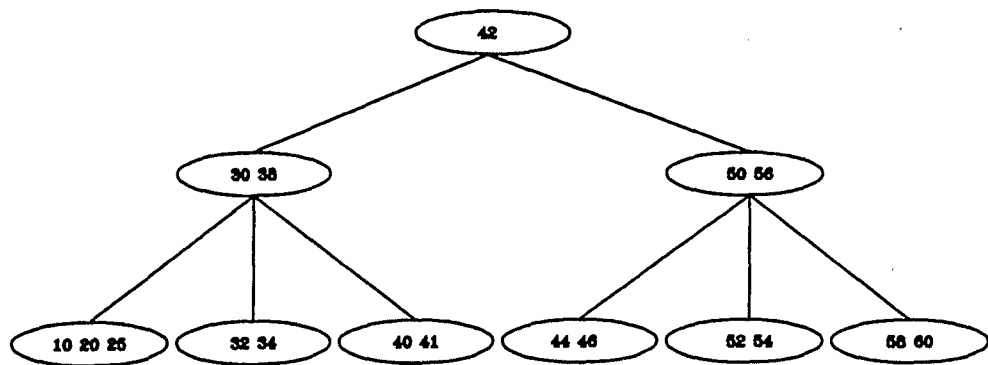
Einfügen von 58, 60, 52 und 54:

Element 58 kann ohne weiteres in das rechte Blatt eingefügt werden. Da dieses Blatt dann die maximale Anzahl Elemente enthält, muß zum Einfügen von 60 ein neuer Knoten aus den beiden größten Elementen (58, 60) gebildet werden und das mittlere Element (56) wird an den Vaterknoten abgegeben. Die Elemente 52 und 54 können ohne weiteres eingefügt werden.



Einfügen von 46:

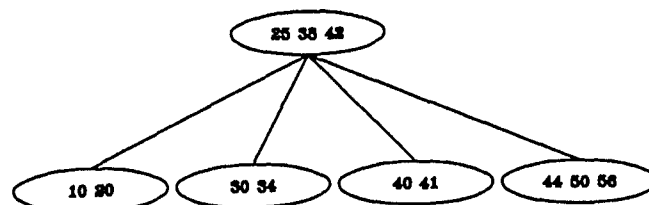
Wenn nun ein Element zwischen 42 und 56 eingefügt werden soll, muß der Knoten aufgeteilt und das mittlere Element aus dem vierten Blatt an den Vaterknoten abgegeben werden. Dies ist beispielsweise beim Einfügen von 46 das Element 50. Da der Vaterknoten, in diesem Fall die Wurzel des Baumes, die maximale Anzahl Elemente enthält, muß er aufgeteilt werden, und das mittlere Element (42) bildet die neue Wurzel.



2. Das zweite Beispiel zeigt das Löschen von Elementen ausgehend von dem B-Baum wie in 1.

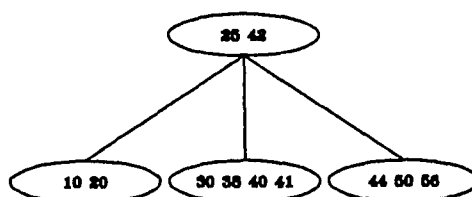
Löschen von 32:

Soll nun Element 32 gelöscht werden, muß ein Element des linken Bruders abgegeben werden (25). Dieses Element wandert in den Vaterknoten als das neue trennende Element der beiden linken Blätter und 30 ersetzt Element 32.



Löschen von 34:

Für das Löschen von Element 34 kann kein Element von einem Bruder abgegeben werden, da beide die minimale Anzahl Elemente enthalten. Daher wird dieser Knoten mit dem rechten Bruder verschmolzen, und das trennende Element aus dem Vaterknoten (38) wandert auch in diesen Knoten.



3.5 Haufen (heaps)

In diesem Abschnitt wird eine Datenstruktur beschrieben, die es erlaubt, auf effiziente Weise Elemente in eine Menge einzufügen und das kleinste Element zu finden. Eine Datenstruktur, die diese beiden Möglichkeiten bietet, nennt man eine Prioritätswarteschlange. Viele Algorithmen benötigen Prioritätswarteschlangen, daher ist eine effiziente Implementierung der zugehörigen Operationen dieser Datenstruktur sehr nützlich.

Definition 3.26: Heap

Ein Heap ist ein vollständiger Binärbaum, der aus einer Folge von Schlüsseln $a[1], a[2], \dots, a[n]$ besteht. Es gelten die Relationen

$$\begin{aligned} a[i] &\leq a[2 \bullet i] \\ a[i] &\leq a[2 \bullet i + 1] \end{aligned} \quad \text{für alle zugehörigen Indizes.}$$

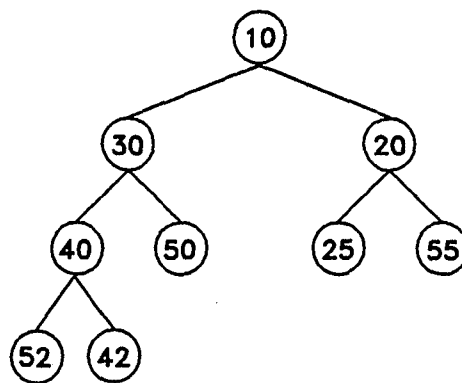
Aus dieser Definition folgt, daß sich das kleinste Element an der Wurzel des Heap befindet.

Beispiel 3.24: Heap

Die folgenden drei Sequenzen von Elementen bilden jeweils einen Heap:

	Heap a	Heap b	Heap c
[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

Binärbaum konstruiert aus (b):



Definition 3.27: Abstrakter Datentyp *Heap*

1. Elemente

Die Elemente sind von einem beliebigen Datentyp (*stdelement*).

2. Struktur

Jedes Element enthält einen Wert aus einer geordneten Menge. Die Elemente bilden einen vollständigen Binärbaum, so daß jeder Wert des Vaterknotens kleiner oder gleich als die Werte seiner Söhne ist.

3. Operationen

a. *heapcreate* (*n* : integer)

pre vollständiger Binärbaum mit *n* Elementen

post Heap mit *n* Elementen

b. *siftdown* (*q*, *n* : integer)

pre *a*[*q* + 1], ..., *a*[*n*] erfüllt die Heap-Bedingung.

post *a*[*q*] ist so eingefügt worden, daß *a*[*q*], ..., *a*[*n*] die Heap-Bedingung erfüllt.

Realisierung eines Heap

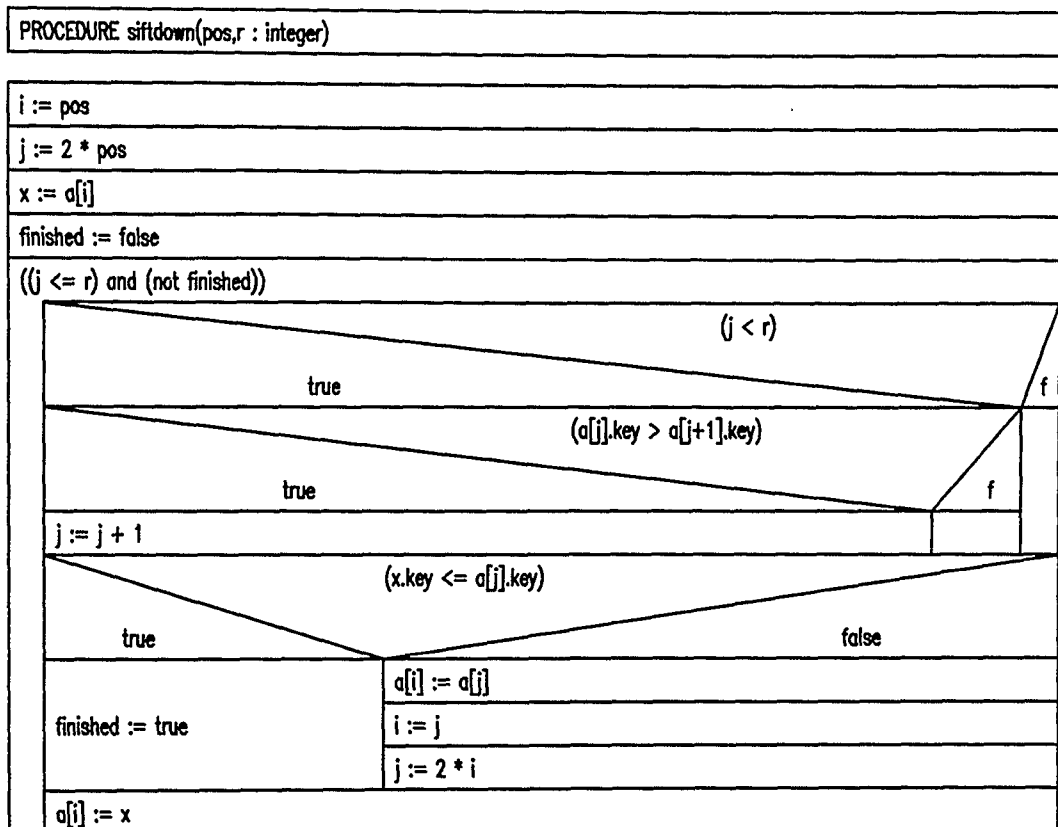
1. Repräsentation

```
TYPE stdelement    = RECORD
                                key : keytype;
                                data : datatype;
                                END; (* stdelement *)
VAR    a                : ARRAY [1..n] OF stdelement;
       x                : stdelement;
```

2. Implementation:

```
PROCEDURE heapcreate(k : integer)
    insert := (k div 2) + 1
    (insert > 1)
        insert := insert - 1
        siftdown (insert,k)
```

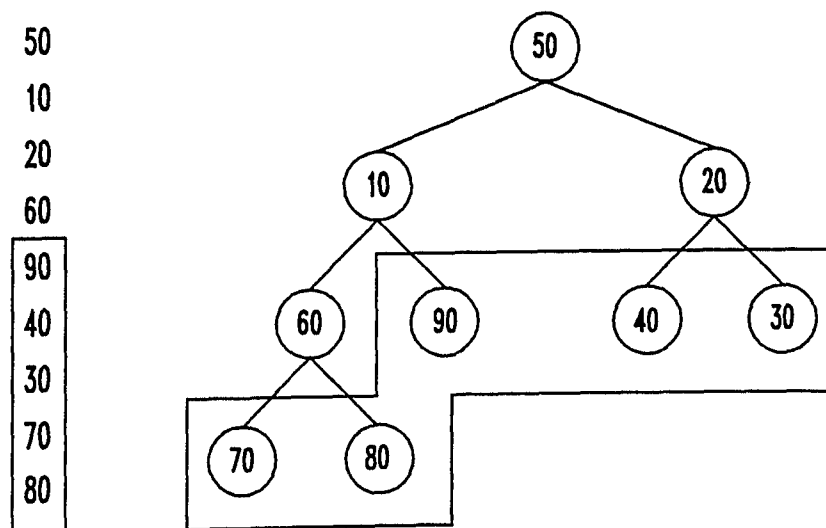
Prozedur heapcreate



Prozedur siftdown

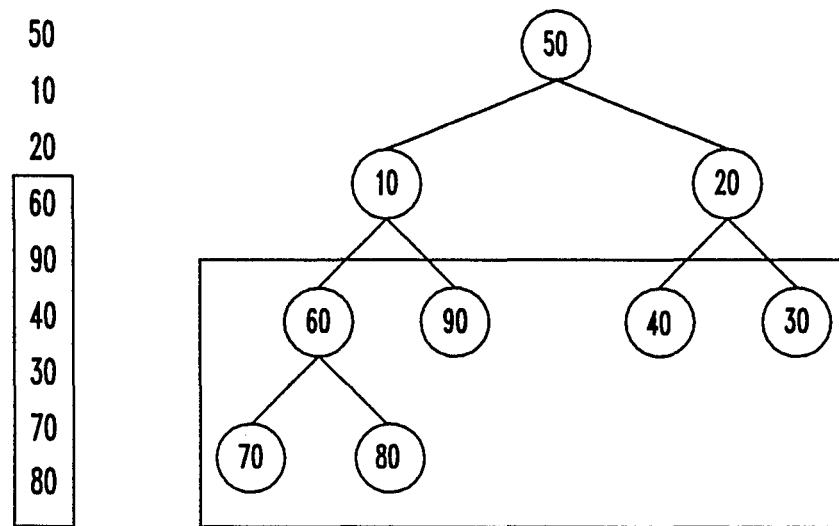
Beispiel 3.25: Bilden eines Heap

Beide Darstellungen eines Heap zeigen dieselben Daten. Die eingerahmten Knoten haben keine Söhne und daher erfüllen die Elemente dieser Knoten die Heap-Bedingung.



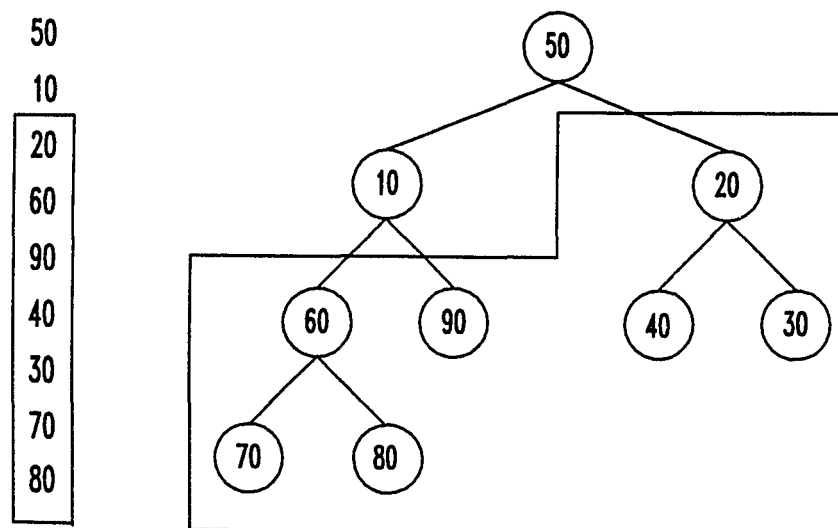
1. Operation: Einfügen von 60

Da das einzufügende Element 60 nur zwei Nachfolger hat, wird es mit dem kleineren Sohn vertauscht.



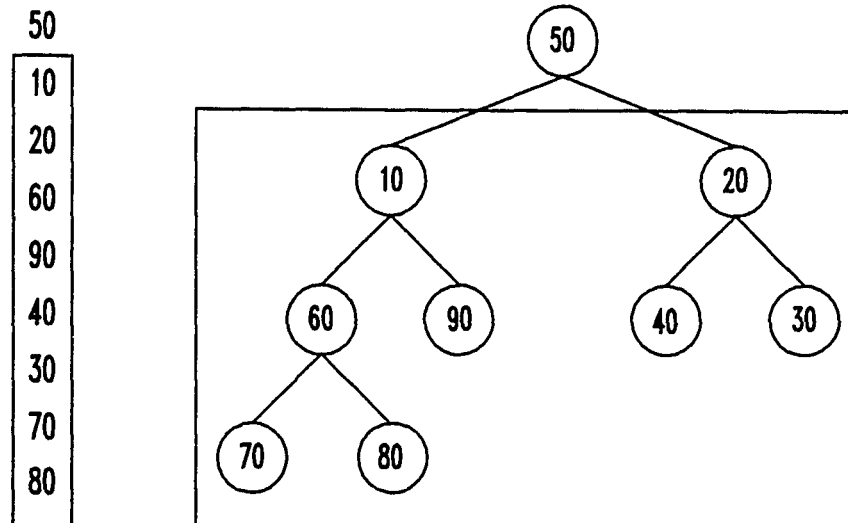
2. Operation: Einfügen von 20

Für das Einfügen von 20 ist kein Austausch nötig, da die Heap-Bedingung bereits erfüllt ist.



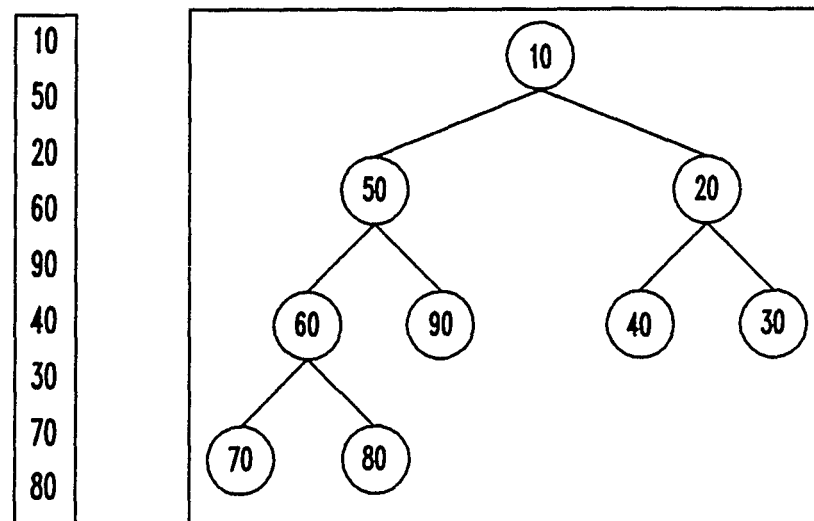
3. Operation: Einfügen von 80

Für das Einfügen von 80 werden nur die Elemente in den beiden Teilbäumen von 80 betrachtet. Es werden jeweils die Kanten vom Vaterknoten zum kleineren Sohn betrachtet und die Elemente vertauscht, bis ein Knoten keinen kleineren Sohn mehr hat.



4. Operation: Einfügen von 50

Für den letzten Schritt, das Einfügen von 50, wird 50 mit dem kleineren Sohn (10) vertauscht. Da die beiden nächsten Söhne größer sind, werden keine weiteren Elemente mehr ausgetauscht und die Konstruktion des Heap ist beendet.



Fertig konstruierter Heap

Zeitkomplexität:

Die Prozedur siftdown folgt einem einfachen Weg von der Wurzel eines vollständigen Binärbaumes zu einem seiner Blätter, d.h. es werden höchstens $\log (n + 1)$ Elemente untersucht.

Die Prozedur heapcreate ruft siftdown für $\frac{n}{2}$ Elemente auf.

Also: $T_{\text{heapcreate}}^{\text{av}}(n) = T_{\text{heapcreate}}^{\text{w}}(n) = \frac{n}{2} \log (n + 1) \in O(n \log n)$

4. Ausgewählte Algorithmen

Es gibt einige wichtige Problembereiche, die in leicht modifizierter Form in nahezu allen komplexen Aufgabenstellungen als Teilprobleme vorkommen. Effiziente Algorithmen und Implementationen für derartige Probleme lassen sich deshalb sehr häufig als Bausteine für die Lösung komplexer Probleme verwenden. In den nun folgenden Abschnitten werden für die Bereiche

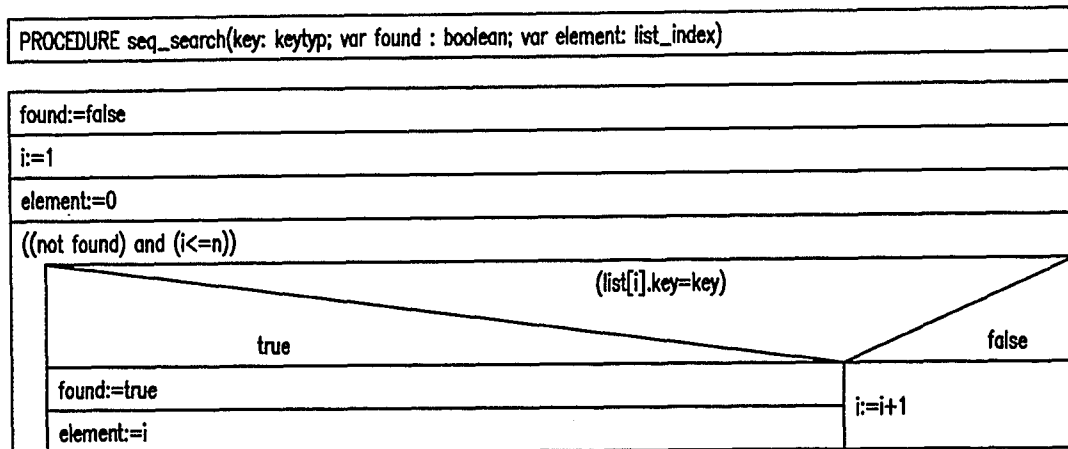
- Suchen von Elementen,
- Sortieren von Elementen,
- effiziente Speicherungs- und Zugriffsverfahren sowie
- Graphenalgorithmen

Lösungsverfahren vorgestellt, Aufwandsbetrachtungen durchgeführt und gegebenenfalls unterschiedliche Implementationen des gleichen Algorithmus gegeneinander abgegrenzt. Wenn im Text nicht anderes vermerkt, gilt in diesem Kapitel als Voraussetzung, daß die Listen generell als Feldimplementierung (mit eindimensionalen Feldern) verfügbar sind, da wahlfreie Adressierung erforderlich ist. Als Vereinbarung gilt, daß die Listenlänge mit n gegeben sei.

4.1 Suchverfahren

Bei der Suche nach Elementen in einer Liste wird üblicherweise nach Schlüsseln (sog. Kennzeichen eines Datensatzes, z.B. der Personalnummer) gesucht.

4.1.1 Lineares Suchen (Sequential Search)



Prozedur seq_search

Vorteile des Algorithmus *lineares Suchen*:

- Einfach zu programmieren;
- Schnell für *kurze* Listen;
- Liste braucht nicht sortiert zu sein.

Nachteile des Algorithmus *lineares Suchen*:

- Hoher Zeitbedarf für lange Listen;
- Im Mittel müssen $\frac{n+1}{2}$, d.h. $O(n)$ Elemente betrachtet werden.

4.1.2 Binäres Suchen (Binary Search)

Definition 4.28: Sortierte Liste

Eine Liste mit n Elementen heißt (nach Schlüsseln) sortiert, falls gilt:

$\forall 1 \leq i \leq n-1: \text{Feld}[i].\text{Schlüssel} \leq \text{Feld}[i+1].\text{Schlüssel}.$

Sortiervverfahren, die eine solche Bedingung auf Listen sicherstellen, werden im nächsten Abschnitt behandelt. Das Suchen von Elementen kann auf zuvor sortierten Listen wesentlich beschleunigt werden.

Beispiel 4.1: Binäres Suchen (Suche nach dem Element mit dem Schlüssel 27)

Listenindex	1	2	3	4	5	6	7	8	9
Schlüssel	11	22	27	33	38	40	41	160	200

Prinzip:

- Greife zuerst auf das mittlere Element zu.
- Prüfe, ob das gesuchte Element
 - a. kleiner
 - b. gleich
 - c. größerist als das betrachtete Element;
im Fall b. ist man fertig, im Fall a. bzw. c. mit linker bzw. rechter Teilliste weitermachen.

Es seien u und o Indizes, die den Beginn bzw. das Ende einer Teilliste festlegen. Mit Hilfe von u und o kann der Index des jeweils mittleren Elementes einer Teilliste berechnet werden.

1. $u = 1$
 $o = 9$

$$\left\lfloor \frac{u + o}{2} \right\rfloor = 5$$

Feld[5].Schlüssel = 38 → linkes Teilfeld weiter betrachten

2. $u = 1$
 $o = 4$

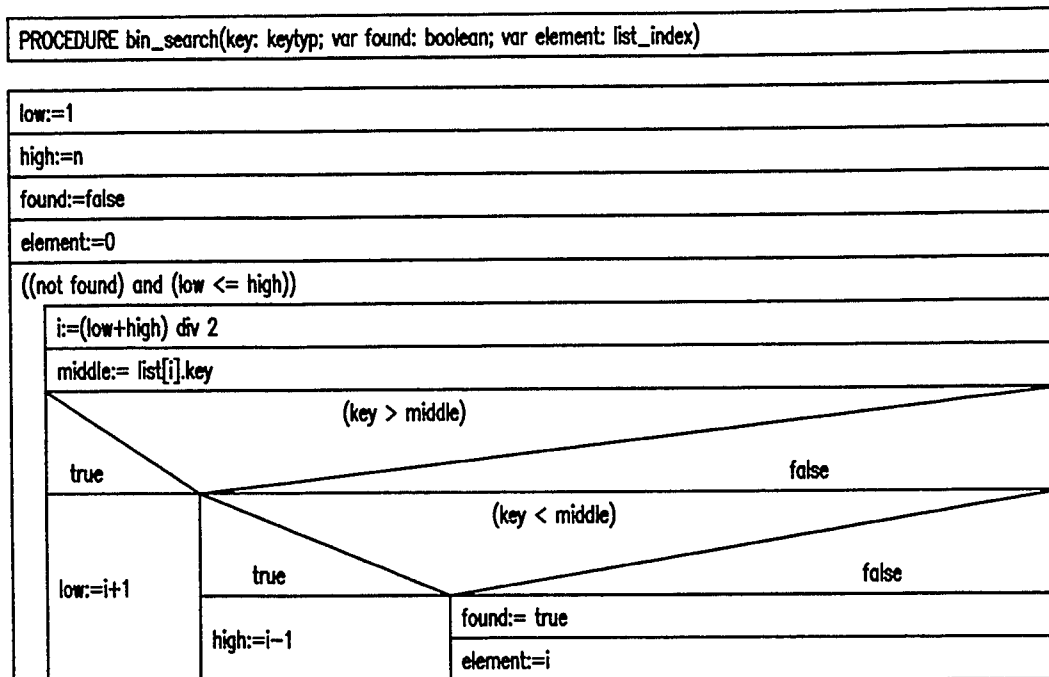
$$\left\lfloor \frac{u + o}{2} \right\rfloor = 2$$

Feld[2].Schlüssel = 22 → rechtes Teilfeld betrachten

3. $u = 3$
 $o = 4$

$$\left\lfloor \frac{u + o}{2} \right\rfloor = 3$$

Feld[3].Schlüssel = 27 → gesuchtes Element gefunden.



Prozedur bin_search

Zeitkomplexität:

Bei jedem Suchschritt halbiert sich die Anzahl der für die Suche relevanten Elemente. Also sind im schlechtesten Fall $\lceil \log(n+1) \rceil$ Suchschritte erforderlich, d.h.

$$T_{\text{bin_search}}^w(n) \in O(\log n).$$

Das Suchverfahren kann durch einen binären Suchbaum veranschaulicht werden (siehe Abb. 33).

Mit k Suchschritten sind $2^k - 1$ Elemente erreichbar. Nach $\lceil \log(n+1) \rceil - 1$ Schritten ist ungefähr die Hälfte aller Elemente erreichbar, d.h. der Suchaufwand im Mittel wird nicht wesentlich geringer sein als der im schlechtesten Fall.

Vorteil des Algorithmus *binäres Suchen*:

- Zeitkomplexität $O(\log n)$

Nachteil des Algorithmus *binäres Suchen*:

- Liste muß sortiert sein, d.h. zum Suchaufwand kommt der Sortieraufwand hinzu. Der Algorithmus ist günstig, falls der Datenbestand statisch ist, d.h. selten oder nie Elemente eingefügt oder gelöscht werden müssen. Beim Einfügen und Löschen

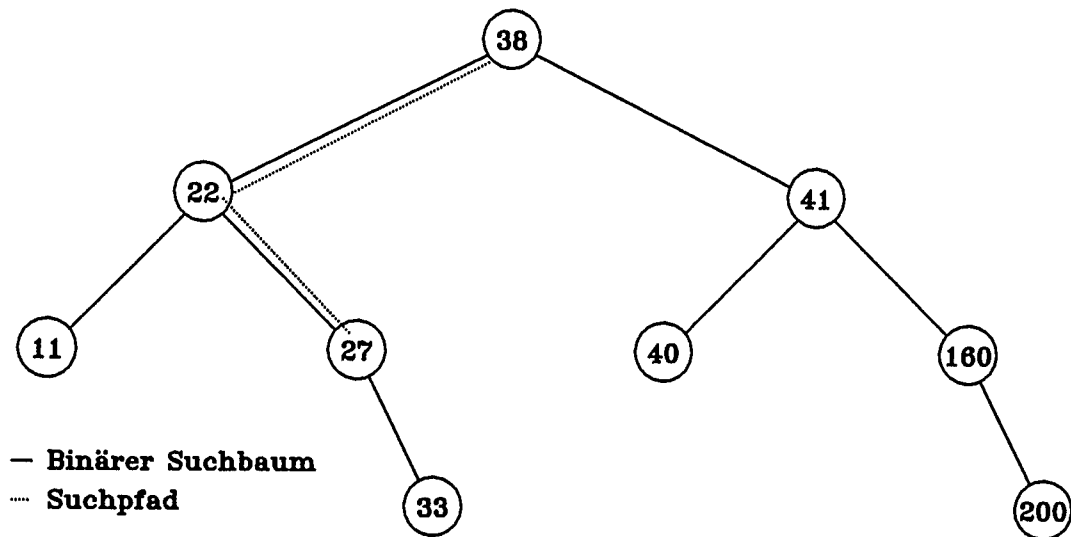


Abb. 33. Binärer Suchbaum und Suchpfad zu Element 27

sind sonst u.U. spezielle Einfügeoperationen möglich, die die Eigenschaft 'sortiert' erhalten.

4.2 Sortierverfahren

Unter Sortieren versteht man allgemein den Prozeß des Anordnens einer gegebenen Menge von Objekten in einer bestimmten Ordnung. Der Sinn des Sortierens liegt in der Vereinfachung der späteren Suche nach Elementen in der gegebenen Menge.

Die Problemstellung 'Sortieren' ist ein gutes Beispiel, um mit einer Vielzahl von Algorithmen die Notwendigkeit der Leistungsanalyse darzulegen, denn durch die Verwendung von effizienten Algorithmen können signifikante Leistungszuwächse erzielt werden.

Unterschieden werden bei den Sortierverfahren *interne* und *externe Sortierverfahren*. Bei einem internen Sortierverfahren liegen die zu sortierenden Elemente im Hauptspeicher. Dagegen müssen bei einem externen Sortierverfahren die zu sortierenden Elemente nicht vollständig im Hauptspeicher liegen. Mit Ausnahme des Sortierens durch Mischen gehören alle hier besprochenen Sortierverfahren zu den internen Sortierverfahren.

4.2.1 Allgemeine Voraussetzungen für Sortierverfahren

Im folgenden seien die Daten mit den zugehörigen Schlüsseln in einem Feld gespeichert.

Das Sortieren kann auf zwei Arten geschehen:

1. Die Daten (inkl. Schlüssel) werden im Speicher auf die richtige Position gebracht (vertauscht).
2. Die Daten bleiben im Speicher an ihrer ursprünglichen Position. Der Zugriff zu den Daten erfolgt über eine "Sortierpermutation" der Indizes.

Definition 4.29: Sortierpermutation

Eine Permutation σ über $(1..n)$ heißt eine Sortierpermutation der Liste A , falls gilt:

$$\forall i, j \quad 1 \leq i < j \leq n : A[\sigma(i)].key \leq A[\sigma(j)].key$$

Bemerkung:

Das Sortieren mit Hilfe einer Sortierpermutation ist u.U. dann sinnvoll, wenn das Umspeichern der Datenrecords sehr aufwendig ist, z.B. bei langen Datenrecords, oder wenn eine Liste nach mehreren unterschiedlichen Schlüsseln sortiert werden soll.

Beispiel 4.2: Sortieren mit Sortierpermutation

Vorher:

Index i	1	2	3	4	5	6
Sortierpermutation $\sigma(i)$	1	2	3	4	5	6
$a[i].key = a[\sigma(i)].key$	3	16	4	1	11	2

Nachher:

Index i	1	2	3	4	5	6
Sortierpermutation $\sigma(i)$	4	6	1	3	5	2
Sortierschlüssel $a[\sigma(i)]$	1	2	3	4	11	16

Bemerkung:

- Die Sortierpermutation kann durch ein eindimensionales Integerfeld der Länge n realisiert werden (z.B. $\text{IPERM}[1..N]$).
- Die Adressierung des Feldes A geschieht immer über $A[\text{IPERM}[I]]$.
- Beim Sortieren werden die Komponenten von IPERM vertauscht, die Komponenten von A (mit u.U. sehr langen Datenrecords) können an der entsprechenden Stelle stehenbleiben.
- Zu Beginn ist $\text{IPERM}[I] = I \forall I = 1, \dots, N$.

Definition 4.30: Stabiles Sortierv Verfahren

Eine Sortiermethode heißt stabil, wenn die relative Ordnung der Elemente mit gleichen Schlüsseln durch den Sortierprozeß unverändert bleibt.

Stabilität beim Sortieren ist oft erwünscht, wenn die Elemente bereits nach einem zweitrangigen Schlüssel geordnet sind, d.h. nach Eigenschaften, die nicht durch den Hauptschlüssel selbst ausgedrückt werden.

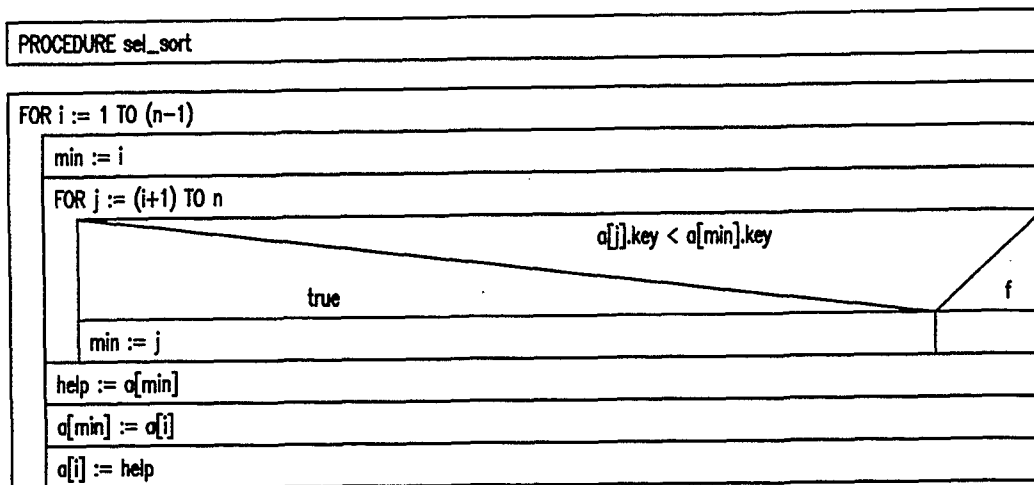
Von den beschriebenen Sortierv Verfahren sind nur das Sortieren durch Einfügen und der Bubblesort-Algorithmus stabile Verfahren.

4.2.2 Sortieren durch Auswahl (Selection Sort)

Prinzipielle Vorgehensweise:

FOR $i := 1$ TO $n-1$ führe durch

1. Suche 'kleinstes' Element von $a[i]$ bis $a[n]$ und weise den Index der Variablen min zu
2. Vertausche $a[i]$ und $a[min]$



Prozedur sel_sort

Beispiel 4.3: Sortieren durch Auswahl

Liste				Variablen	
a[1]	a[2]	a[3]	a[4]	Minimum	i
<5>	<1>	8	3	2	1
1	<5>	8	<3>	4	2
1	3	<8>	<5>	4	3
1	3	5	8		

Mit < > werden die in einer Iteration jeweils zu vertauschenden Elemente gekennzeichnet.

Zeitkomplexität des Algorithmus:

Im i-ten Schritt des Algorithmus ist das Minimum aus (n-i+1) Elementen zu suchen, wobei i von 1 bis (n-1) läuft, d.h. im ersten Schritt sind es (n-1) Vergleiche.

$$\begin{aligned}
 T_{\text{sel_sort}}^{\text{av}}(n) &= T_{\text{sel_sort}}^{\text{w}}(n) = (n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n-1} i \\
 &= \frac{n(n-1)}{2} \in O(n^2).
 \end{aligned}$$

Bemerkung:

Vertauscht werden maximal n-1 Elemente, da das Vertauschen in der äußeren Schleife stattfindet.

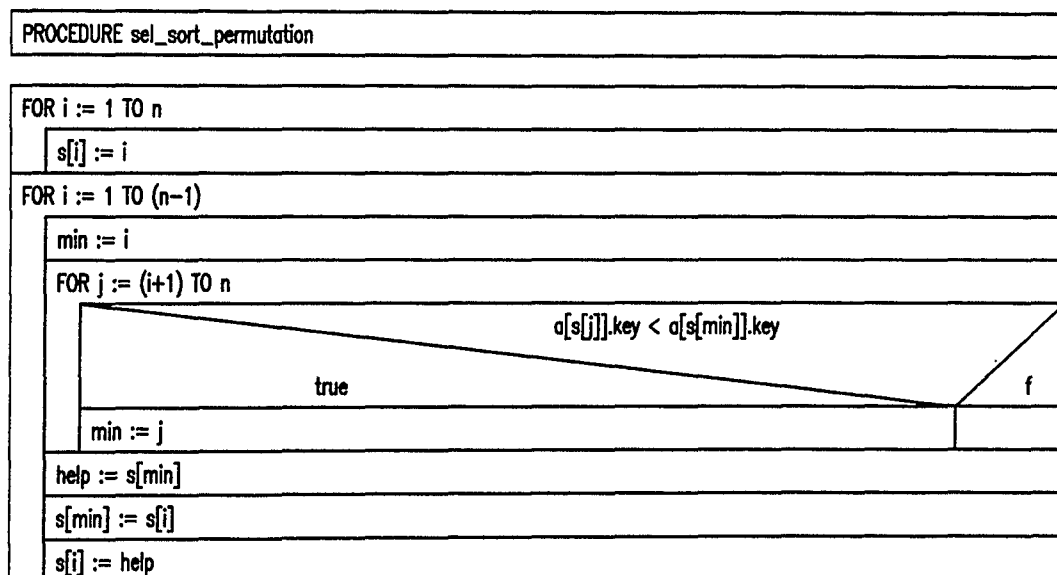
Vorteile des Algorithmus *Sortieren durch Auswahl*:

- Einfach zu programmieren.
- Schnell für sehr kleine Werte von n (z.B. $n < 20$).

Nachteile des Algorithmus *Sortieren durch Auswahl*:

- Hohe Zeitkomplexität von $O(n^2)$.
- Der Algorithmus erkennt nicht, ob eine Liste vorsortiert ist. Bei total sortierter Eingabe ist der gleiche Rechenaufwand nötig.

Im folgenden wird der Algorithmus unter Verwendung einer Sortierpermutation vorgestellt:



Prozedur sel_sort_permutation

4.2.3 Sortieren durch Einfügen (Insertion Sort)

Die Basisoperation bei diesem Algorithmus ist das Einfügen eines einzelnen Elementes in eine bereits sortierte Liste an der entsprechenden Stelle.

Prinzipielle Vorgehensweise:

FOR $i := 2$ TO n führe durch

1. $x := a[i]$
2. Füge x am entsprechenden Platz in $a[1] \dots a[i]$ ein

PROCEDURE ins_sort
FOR $i := 2$ TO n
$x := a[i]$
$a[0] := x$
$j := i - 1$
$(x.key < a[j].key)$
$a[j+1] := a[j]$
$j := j - 1$
$a[j+1] := x$

Prozedur ins_sort

Zum Bestimmen des entsprechenden Platzes ist es angebracht, zwischen Vergleichen und Bewegungen abzuwechseln, d.h. x "nach links wandern zu lassen", mit dem nächsten Element $a[j]$ zu vergleichen und entweder x einzufügen oder $a[j]$ nach rechts zu schieben und nach links fortzufahren.

Es gibt zwei Möglichkeiten, durch die der Prozeß des "nach links Wanderns" beendet werden kann:

1. Ein Element $a[j]$ mit kleinerem Schlüssel als x wird gefunden.
2. Das linke Ende der Ziel-Sequenz ist erreicht.

Für diese Wiederholung mit zwei Abbruchbedingungen wird eine Marke $a[0] = x$ verwendet.

(Beachte: Indexbereich von a muß auf $0..n$ erweitert werden.)

Beispiel 4.4: Sortieren durch Einfügen

Liste				Variablen		
a[1]	a[2]	a[3]	a[4]	i	j	a[0]
<5>	<1>	8	3	2	1	1
1	<5>	<8>	3	3	2	8
1	5	<8>	<3>	4	3	3
1	<5>	<3>	8	4	2	3
<1>	<3>	5	8	4	1	3

Zeitkomplexität des Algorithmus:

Die Zahl der Vergleiche von Schlüsseln beim i-ten Durchlauf ist höchstens i-1 und mindestens 1 und somit, unter der Annahme, daß alle Permutationen gleich wahrscheinlich sind, im Mittel $\frac{i}{2}$.

$$\begin{aligned}
 T_{\text{ins_sort}}^{\text{av}}(n) &= \sum_{i=2}^n \frac{i}{2} = \frac{1}{2} \left(\frac{n(n+1)}{2} - 1 \right) = \frac{n(n+1) - 2}{4} \\
 &= \frac{n^2 + n - 2}{4} \in O(n^2)
 \end{aligned}$$

Der beste Fall ist, wenn die Elemente von Anfang an geordnet sind; der schlimmste Fall tritt ein, wenn die Elemente zu Beginn in umgekehrter Reihenfolge angeordnet sind. In diesem Sinn zeigt Sortieren durch Einfügen ein natürliches Verhalten.

Sortieren durch Auswahl ist gewissermaßen das Gegenstück zum Sortieren durch Einfügen. Sortieren durch Einfügen betrachtet bei jedem Schritt nur das eine nächste Element der Quellen-Sequenz und alle Elemente des Ziel-Array, um die Einschubstelle zu finden, während Sortieren durch Auswahl alle Elemente der Quellen-Sequenz betrachtet, um das Element mit dem kleinsten Schlüssel zu finden und es als das eine nächste Element der Ziel-Sequenz anzufügen.

Der Algorithmus des Sortieren durch Einfügen läßt sich ohne weiteres verbessern, indem die bereits vorhandene Ordnung der Ziel-Sequenz $a[1] \dots a[i-1]$, in die das neue Element einzufügen ist, ausgenutzt wird. Dazu kann man die *binäre Suche* verwenden $\Rightarrow$ *Binäres Einfügen*.

Vorteile des Algorithmus *Sortieren durch Einfügen*:

- Einfach zu programmieren.
- Schnell für sehr kleine Werte von n .
- Vorsortierung wird ausgenutzt.

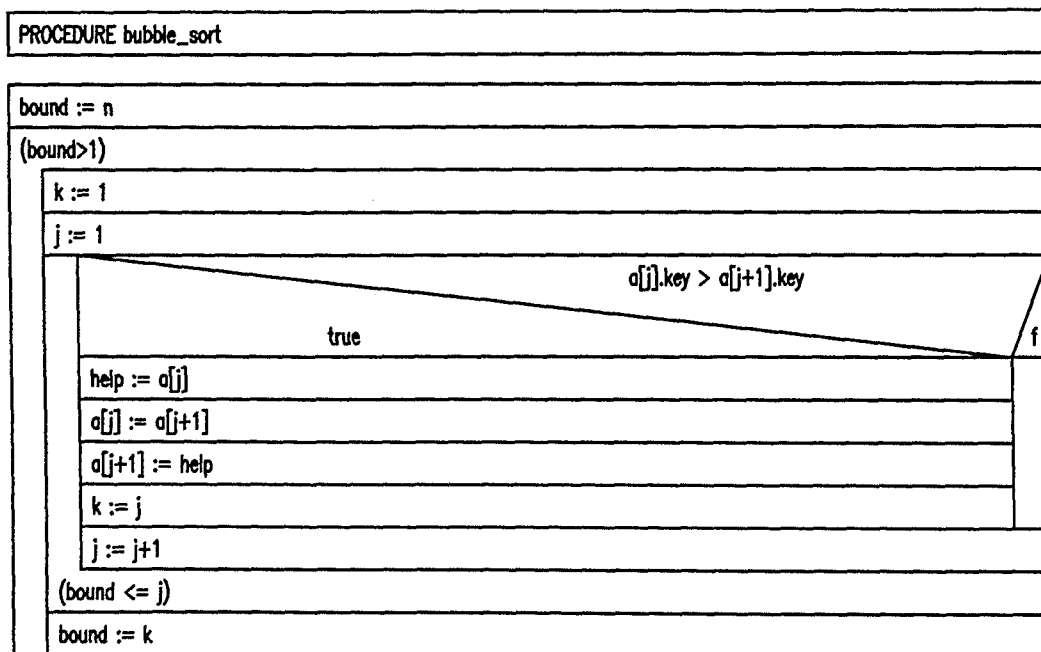
Nachteile des Algorithmus *Sortieren durch Einfügen*:

- Hohe Zeitkomplexität von $O(n^2)$.

4.2.4 Sortieren durch Vertauschen (Bubble Sort, Sinking Sort)

Prinzip:

- Jeweils zwei aufeinanderfolgende $a[i]$ und $a[i+1]$ ($i=1, \dots, (n-1)$) werden miteinander verglichen und vertauscht, wenn $a[i] > a[i+1]$ gilt. Auf diese Weise wandert im ersten Durchgang zumindest das größte Element ganz nach rechts an Position n .
- Es werden weitere Durchgänge für $i=1, \dots, k$ gemacht, wobei k die Position ist, ab der bei dem vorigen Durchgang keine Vertauschungen mehr aufgetreten sind.



Prozedur bubble_sort

Beispiel 4.5: Sortieren durch Vertauschen (Bubblesort)

Liste					Variablen		
a[1]	a[2]	a[3]	a[4]	a[5]	j	k	bound
<5>	<3>	1	2	9	1	1	5
3	<5>	<1>	2	9	2	1	5
3	1	<5>	<2>	9	3	2	5
3	1	2	5	9	4	3	5
<3>	<1>	2	5	9	1	1	3
1	<3>	<2>	5	9	2	1	3
1	2	3	5	9	3	2	3
1	2	3	5	9	1	1	2
1	2	3	5	9	1	1	1

Zeitkomplexität:

Im ersten Durchgang werden (n-1) Elementpaare miteinander verglichen und ggf. miteinander vertauscht. Allgemein werden im i-ten Durchgang im ungünstigsten Fall (n-i) Elementpaare verglichen und vertauscht ($i = 1, \dots, (n-1)$), d.h. der Aufwand beträgt:

$$T_{\text{bubble_sort}}^w(n) = (n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2},$$

d.h. diese Anzahl Elementpaare wird im worst-case vertauscht.

Also: $T_{\text{bubble_sort}}^w(n) \in O(n^2)$.

Im günstigsten Fall, d.h. wenn die Liste schon vorsortiert ist, sind (n-1) Vergleiche erforderlich. Dieser Fall ist jedoch sehr unwahrscheinlich, eine von n! Möglichkeiten. Der Aufwand beträgt dann:

$$T_{\text{bubble_sort}}^b(n) \in O(n).$$

Vorteile des Algorithmus *Sortieren durch Vertauschen*:

- Schnell für sehr kleines n.
- Vorsortierung wird ausgenutzt.

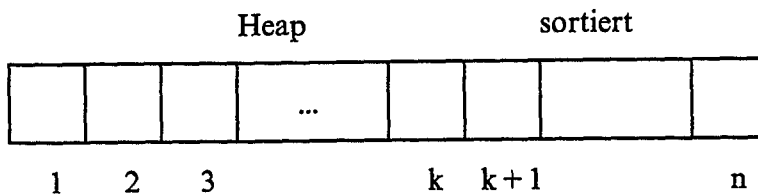
Nachteile des Algorithmus *Sortieren durch Vertauschen*:

- Viele Vertauschungen in der innersten Schleife.
- Hohe Komplexitätsordnung.

4.2.5 Heapsort

Der Algorithmus besteht aus zwei Schritten: Zuerst werden die Elemente in einen Heap gebracht, und dann werden die Elemente sortiert aus dem Heap entnommen.

Der Heap kann analog zur Beschreibung in Kapitel 3.5 gebildet werden. Man beachte, daß das kleinste Element das erste Element ist. Die Elemente sind in $a[1]$, $a[2]$, ..., $a[n]$ gespeichert.



Im ersten Schritt werden $a[1]$ und $a[n]$ ausgetauscht. Für $a[1]$, ..., $a[n-1]$ wird der Heap wieder aufgebaut. Das Ergebnis des ersten Schritts ist, daß $a[1]$, ..., $a[n-1]$ einen Heap bilden und $a[n]$ eine sortierte Liste.

Im zweiten Schritt werden $a[1]$ und $a[n-1]$ ausgetauscht, und für $a[1]$, ..., $a[n-2]$ wird der Heap wieder aufgebaut, d.h. $a[1]$, ..., $a[n-2]$ erfüllt die Heapbedingung und $a[n-1]$, $a[n]$ bildet eine sortierte Liste, usw.

PROCEDURE heapsort

heapcreate(n)

k := n

(k > 1)

temp := a[1]

a[1] := a[k]

a[k] := temp

k := k - 1

siftdown(1,k)

Prozedur heapsort

Beispiel 4.6: Heapsort

390 205 182 45 235

Bilde Heap:

45	205	182	390	235
Heap				

Vertausche a[1] und a[5] und bilde Heap neu:

182	205	235	390	45
Heap				sortiert

Vertausche a[1] und a[4] und bilde Heap neu:

205	390	235	182	45
Heap			sortiert	

Vertausche a[1] und a[3] und bilde Heap neu:

235	390	205	182	45
Heap		sortiert		

Vertausche a[1] und a[2]:

390	235	205	182	45
sortiert				

Bemerkung:

Das Beispiel zeigt, daß die resultierende Ordnung der Elemente absteigend ist. Die Ordnung läßt sich leicht in die umgekehrte Sortierreihenfolge ändern, indem man die Ordnungsrelation in der Prozedur für das Einfügen (siftdown) ändert.

Zeitkomplexität des Algorithmus:

Für das Bilden des Heap (heapcreate) wird für $\frac{n}{2}$ Elemente die Prozedur siftdown ausgeführt. Maximal werden $\log(n+1)$ Elemente verglichen.

Für Schritt 1 gilt:

$$T_{\text{heapcreate}}^w(n) = \frac{n}{2} \log(n+1).$$

Für das Sortieren der Elemente wird siftdown für $n-1$ Elemente ausgeführt, d.h. für Schritt 2 gilt: $T(n) = (n-1) \log(n+1)$.

Für Heapsort insgesamt gilt dann:

$$T_{\text{heapsort}}^w(n) = \frac{3}{2} n \log(n+1) \in O(n \log n)$$

Bemerkung:

Heapsort ist verwandt mit dem Sortieren durch Auswahl, benutzt aber eine effizientere Datenstruktur. Für wenige Elemente ist jedoch Sortieren durch Auswahl schneller.

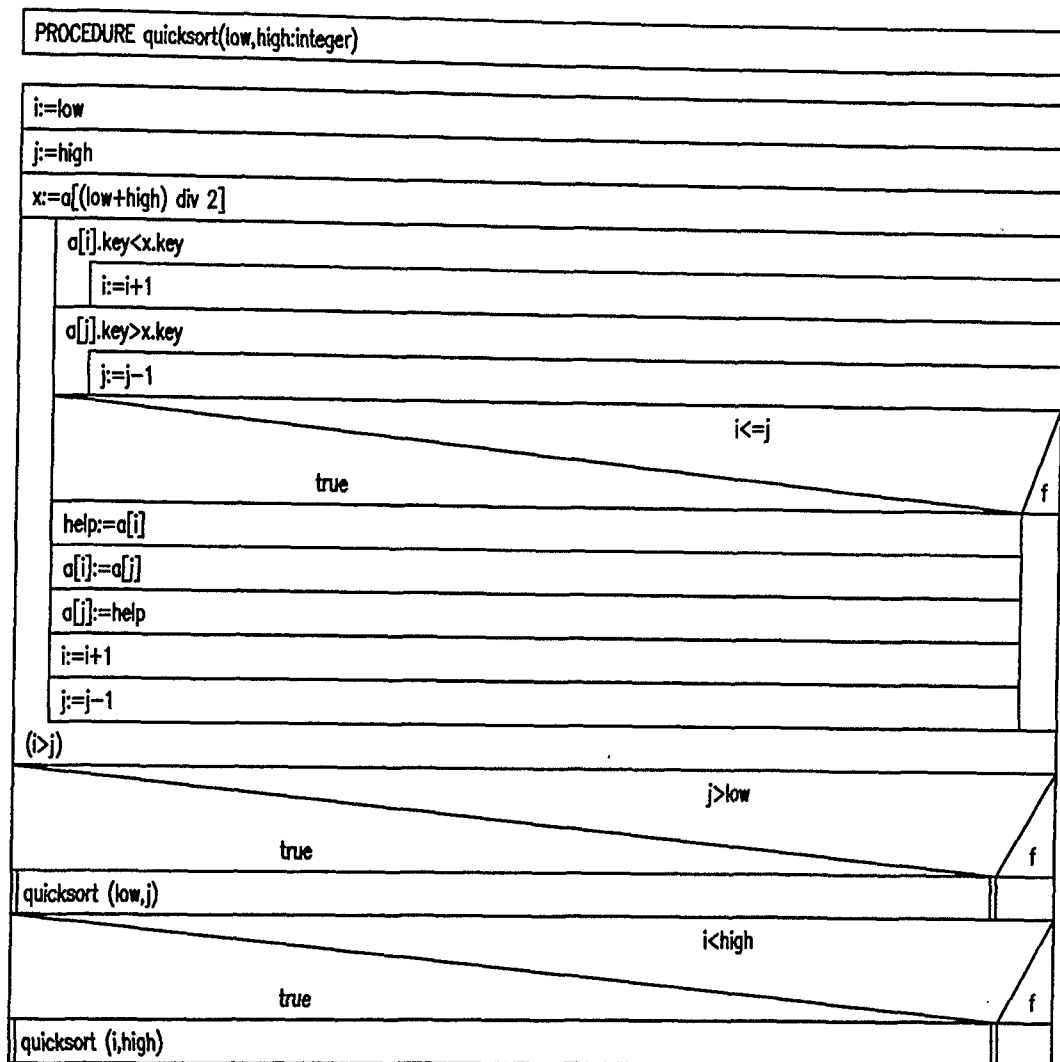
4.2.6 Quicksort

Das Quicksort-Verfahren arbeitet nach dem Prinzip *divide-and-conquer* ('teile' und 'herrsche').

Vorgehensweise:

1. Die zu sortierende Liste wird in zwei Teile aufgeteilt (untere Teilliste, obere Teilliste).
2. Die Aufteilung geschieht so, daß in der unteren Teilliste nur Elemente $\leq$ einem Referenzelement, in der oberen Teilliste nur Elemente $\geq$ diesem Referenzelement vorkommen (ggf. müssen dabei Elemente vertauscht werden).
3. Die so entstandenen (kleineren) Teillisten werden nach dem gleichen Prinzip "sortiert" (rekursive Vorgehensweise). Das Verfahren endet, wenn die Teillisten weniger als zwei Elemente enthalten.

$\leq X$	X	$\geq X$
untere Teilliste	Referenzelement	obere Teilliste



Prozedur quicksort (rekursive Lösung)

Korrektheit des Algorithmus:

Nach Ende der Partitionierung in zwei Teillisten (Ende der Repeat-Schleife) gilt:

1. i zeigt auf ein Element $\geq x.key$
2. j zeigt auf ein Element $\leq x.key$
3. $i > j$
4. Alle Elemente links von j sind $\leq x.key$, denn es gilt: $i > j$.
5. Alle Elemente rechts von i sind $\geq x.key$, denn es gilt: $i > j$.

Also ist (low,j) eine Teilliste mit Elementen $\leq x.key$ und (i,high) eine Teilliste mit Elementen $\geq x.key$. Zwei Fälle sind möglich:

1. $j+1=i$: O.K., da vollständige Zerlegung in zwei Teillisten.
2. $j+2=i$: In diesem Fall gilt unmittelbar vor Ende der Repeat-Schleife $i=j$ und $a[i]=a[j]=x$. Dann folgt $i=i+1$ und $j=j-1$, und damit endet die Repeat-Schleife.
Also: (low,j) $\times$ $(i,high)$, d.h. x steht schon an der korrekten Position, (low,j) und $(i,high)$ sind noch zu sortieren.

Beispiel 4.7: Veranschaulichung der Vorgehensweise des Quicksort Algorithmus

Liste (Werte von $a[i].key$)									Variablen			
$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$	low	high	i	j
1	3	9	7	<2>	5	8	10	4	1	9	1	9
1	<2>	9	7	3	5	8	10	4	1	9	2	5
Anmerkung: neuer Aufruf von Quicksort(1,2)												
<1>	2								1	2	1	2
Anmerkung: neuer Aufruf von Quicksort(3,9)												
		9	7	3	<5>	8	10	4	3	9	3	9
		4	7	3	<5>	8	10	9	3	9	3	9
		4	<5>	3	7	8	10	9	3	9	4	6
Anmerkung: neuer Aufruf von Quicksort(3,5)												
		4	<5>	3					3	5	3	5
		4	3	<5>					3	5	4	5
Anmerkung: neuer Aufruf von Quicksort(3,4)												
		<4>	3						3	4	3	4
		3	<4>						3	4	3	4
Anmerkung: neuer Aufruf von Quicksort(6,9)												
					7	<8>	10	9	6	9	6	9
Anmerkung: neuer Aufruf von Quicksort(8,9)												
							<10>	9	8	9	8	9
							9	<10>	8	9	8	9
Anmerkung: Verlassen der Prozedur Quicksort												
1	2	3	4	5	7	8	9	10				

Bemerkung:

In Abb. 34 ist der Zugriff auf die einzelnen Elemente eines zu sortierenden Feldes (mit Zufallszahlen initialisiert) mit Hilfe von einem Speicherreferenzdiagramm visualisiert. An diesem Beispiel wird die rekursive Vorgehensweise des divide-and-conquer-Prinzips durch die Zerlegung in Teillisten verdeutlicht.

Bemerkung:

Die iterative Realisierung von Quicksort unterscheidet sich in den folgenden Punkten vom rekursiven Algorithmus:

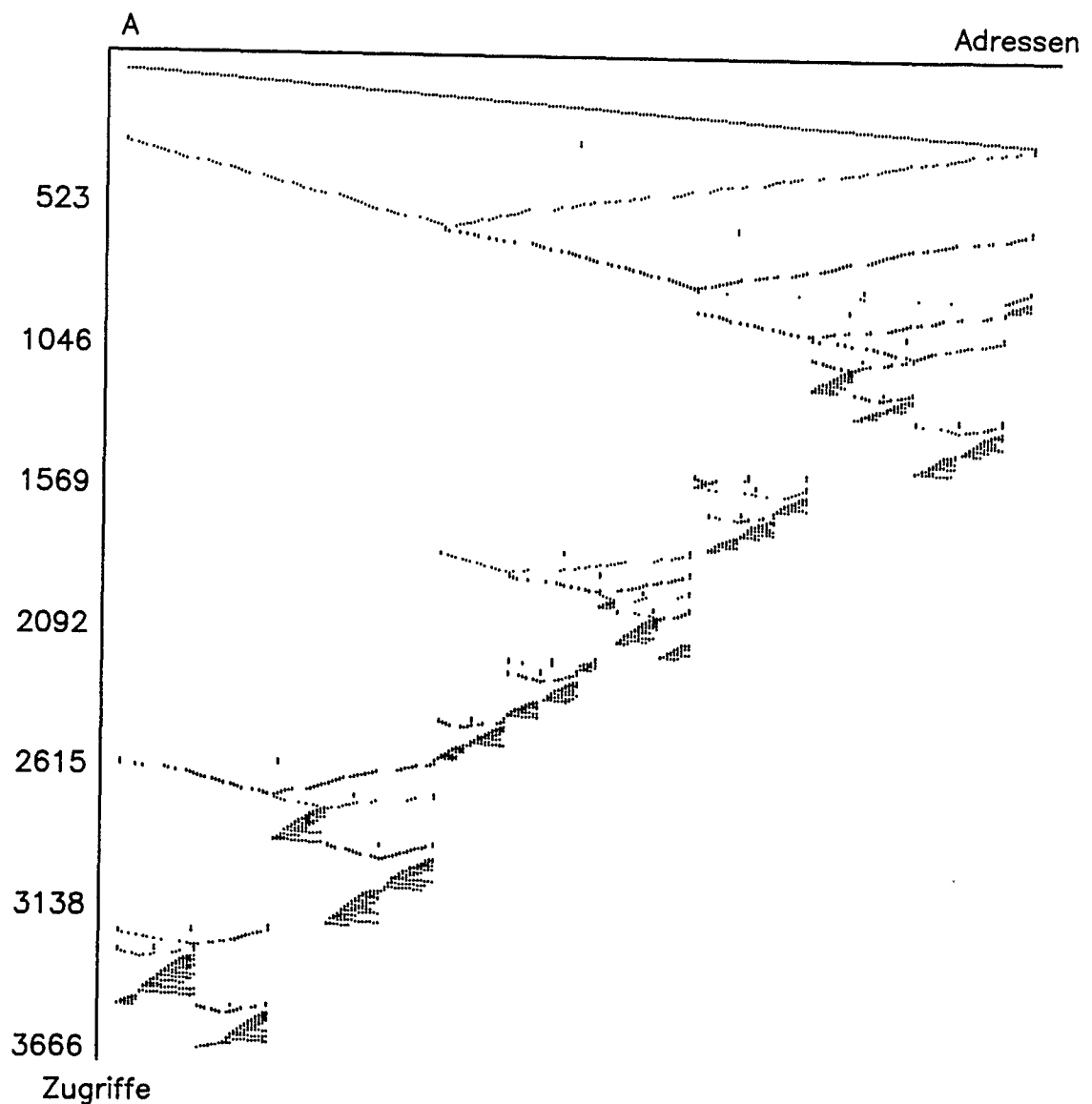
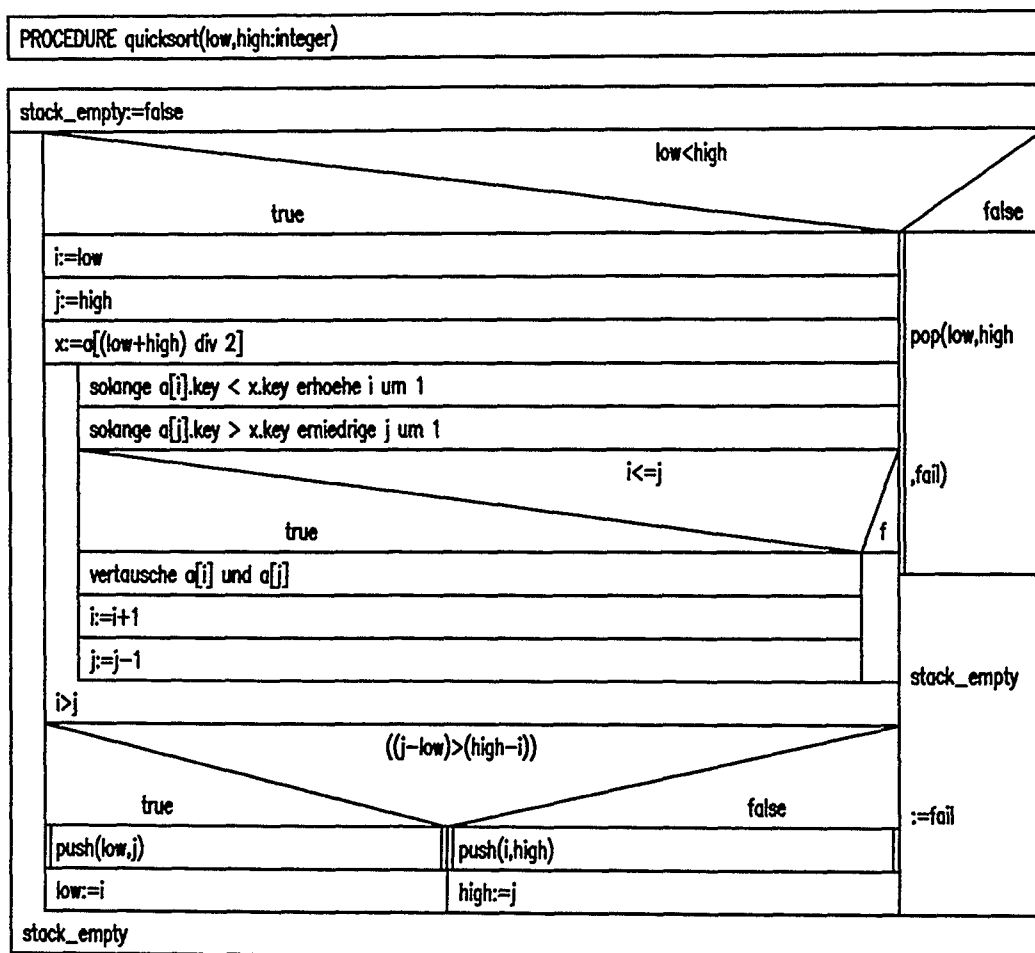


Abb. 34. Speicherreferenzdiagramm für den Quicksort-Algorithmus

1. Nach der Partitionierung (Ende der inneren Repeat-Schleife) wird der größere Bereich auf den Stack geschrieben. Der kleinere Bereich wird sofort weiterbehandelt, d.h. nicht auf den Stack geschrieben. Auf diese Weise wird gewährleistet, daß der Stack nicht mehr als $O(\log n)$ Elemente aufnehmen muß. (Bem.: Der jeweils bearbeitete Teilbereich ist höchstens halb so groß wie der vorige, damit endet diese "Rekursion" nach spätestens $\log n$ Schritten).
2. Beim rekursiven Algorithmus muß der (interne) Stack dagegen im worst-case $O(n)$ Elemente aufnehmen. Dies kann u.U. zu Speicherplatzmangel führen.

3. Die Abfrage $low < high$ ist an den Anfang der Prozedur gestellt worden.



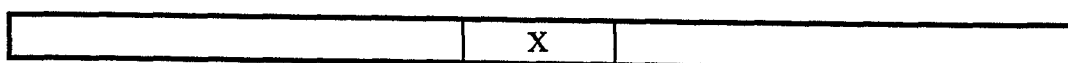
Prozedur quicksort (iterative Lösung)

Zeitkomplexität von Quicksort:

Aufwand im günstigsten Fall:

Die Teillisten werden stets halbiert.

Stufe 1



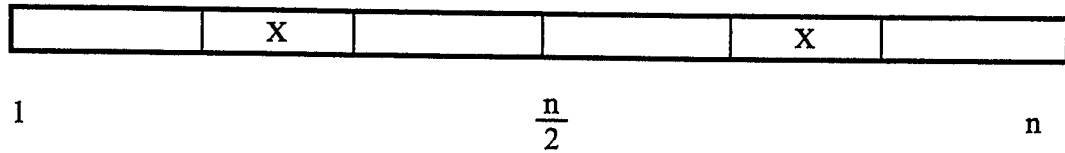
1

n

In jeder Teilliste der Stufe 1 werden $\frac{n}{2}$ Elemente betrachtet und ggf. vertauscht.

Also insgesamt in der Stufe 1: $\frac{n}{2} + \frac{n}{2} = n$.

Stufe 2



In jeder Teilliste werden $\frac{n}{4}$ Elemente betrachtet, also insgesamt $4 \frac{n}{4} = n$, da vier Teillisten existieren. Allgemein gilt, daß in jeder Stufe n Elemente betrachtet werden müssen. Wegen der fortgesetzten Halbierung der Teillisten und Abbruch bei Teillisten der Länge $n < 2$ existieren $\lceil \log n \rceil$ Stufen.

Also: $T_{\text{quicksort}}^b(n) \in O(n \log n)$.

Aufwand im worst-case:

In jeder Stufe wird als Referenzelement X gerade das größte oder kleinste Element der Teilliste ausgewählt.

Dann gilt:

Die Länge der längsten Teilliste ist $(n-1)$ bei Stufe 1, $(n-2)$ bei Stufe 2, etc.

Allgemein: $(n-i)$ bei Stufe i

In diesem Fall existieren $(n-1)$ Stufen. In jeder Stufe werden n Elemente betrachtet.

Also: $T_{\text{quicksort}}^w(n) \in O(n^2)$.

Aufwand im Mittel:

Die genaue Analyse ist sehr aufwendig (siehe [7]), als Resultat ergibt sich:

$T_{\text{quicksort}}^{\text{av}}(n) \in O(n \log n)$.

Verbesserungsmöglichkeiten für Quicksort:

1. End-Rekursion vermeiden. Die End-Abfrage erfolgt nicht nach dem letzten (rekursiven) Aufruf, sondern vor dem Aufruf. Hierdurch werden die Rekursionen der letzten Stufe vermieden.
2. Kleine Teillisten (mit weniger als 20 Elementen) sollten mit *Insertion Sort* oder *Bubblesort* sortiert werden.

Vorteil: Dies verhindert die sehr häufigen Rekursionen am Ende.

3. *Median-of-three-Methode* zum Auswählen des Referenzelementes:

Prinzip:

Es werden drei Elemente als Referenzelemente, z.B. vom Listenanfang, vom Listenende und aus der Mitte gewählt. Das Element mit dem Median-Wert (mittlerer Wert) als Schlüssel wird als Referenzelement gewählt. So wird vermieden, das größte (bzw. das kleinste) Element als Referenzelement auszuwählen, und die Wahrscheinlichkeit für ein 'mittleres' Element erhöht sich.

Vorteile des Algorithmus *Quicksort*:

- Niedrige Komplexitätsordnung im average-case.
- Kleine Konstanten in der Zeitfunktion.

Nachteile des Algorithmus *Quicksort*:

- Zusätzlicher Zeit- und Speicherplatzbedarf für den Stack.
- Vorsortierung wird nicht ausgenutzt.
- Schlechte Zeitkomplexität im worst-case-Fall.

4.2.7 Sortieren durch Mischen (Mergesort)

Die bisher beschriebenen Sortieralgorithmen sind leider nicht anwendbar, wenn die zu sortierende Menge der Daten nicht in den Hauptspeicher paßt, sondern z.B. in einem peripheren und sequentiellen Speichermedium, wie einem Band, enthalten ist. In diesem Fall beschreiben wir die Daten als (sequentielle) Datei mit der Eigenschaft, daß zu jeder Zeit genau eine Komponente direkt zugreifbar ist. Dies ist eine starke Einschränkung im Vergleich zu Array-Strukturen und es sind andere Techniken des Sortierens zu verwenden. Die Bedeutendste ist *Sortieren durch Mischen* (Mergesort).

Mischen bedeutet das Zusammenfassen von zwei (oder mehreren) geordneten Sequenzen zu einer einzigen geordneten Sequenz durch fortgesetzte Auswahl aus den gerade zugreifbaren Komponenten.

Beispiel 4.8:

Eine Liste (Datei, File) A habe Schlüssel $a[i].key \in S$ (S Schlüsselmenge) und eine Liste B habe Schlüssel $b[i].key \in S$. Aus beiden Listen soll eine neue sortierte Liste C erstellt werden, die die Elemente von A und B in sich vereinigt.

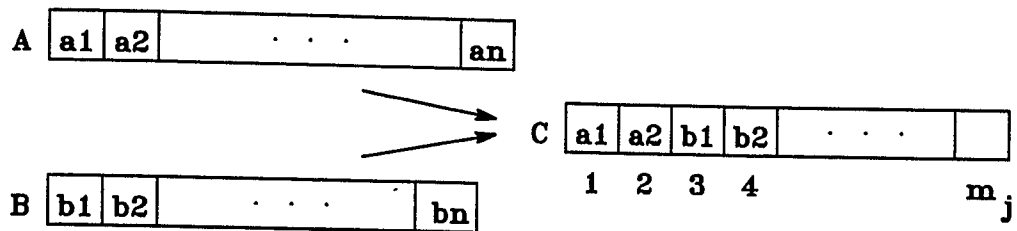


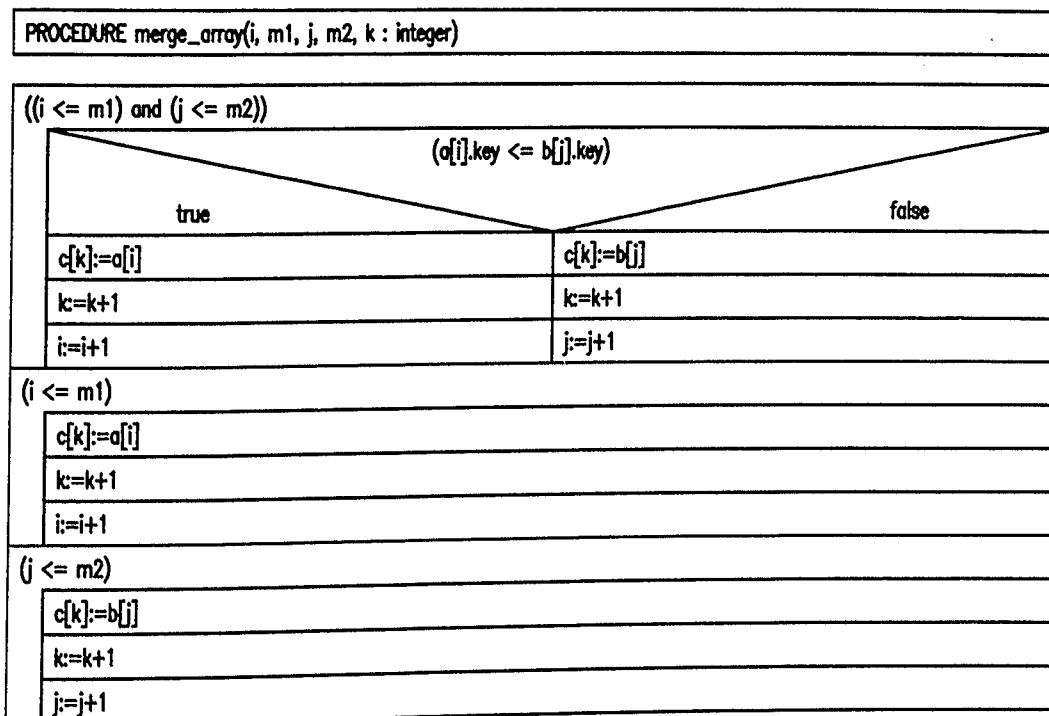
Abb. 35. Mischen von sortierten Listen A und B zu einer sortierten Liste C

Zunächst gelte:

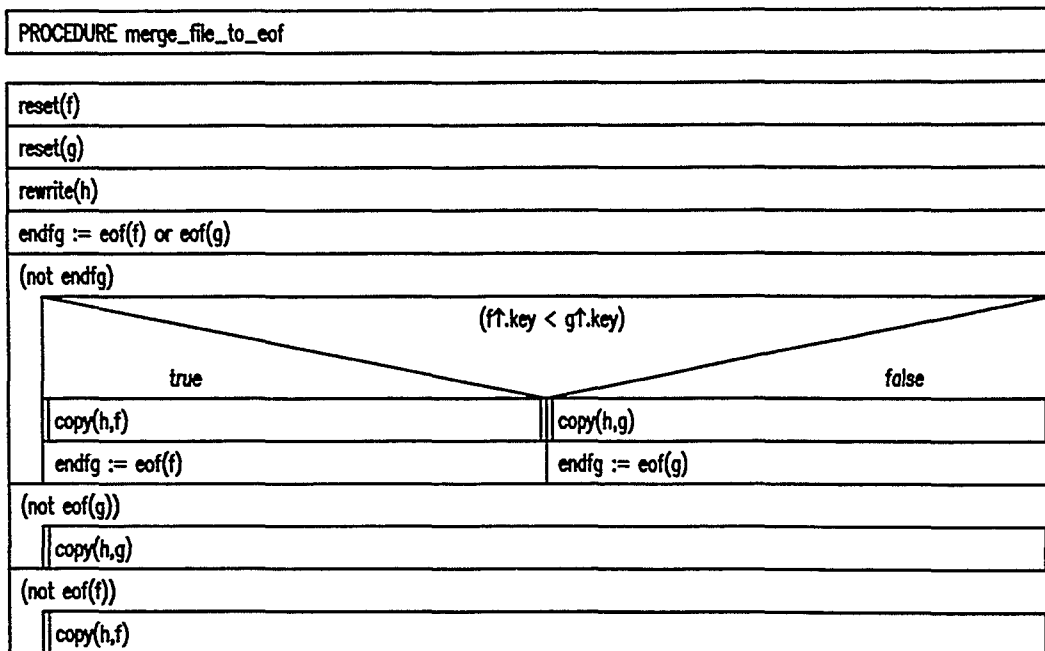
A,B,C : Listen, die Listen A und B seien vorsortiert
i,m1 : untere/obere Grenze von A
j,m2 : untere/obere Grenze von B
k : untere Grenze von C

In der Prozedur merge_array sind die Daten in zwei sortierten Arrays gespeichert, die aber streng sequentiell durchlaufen werden.

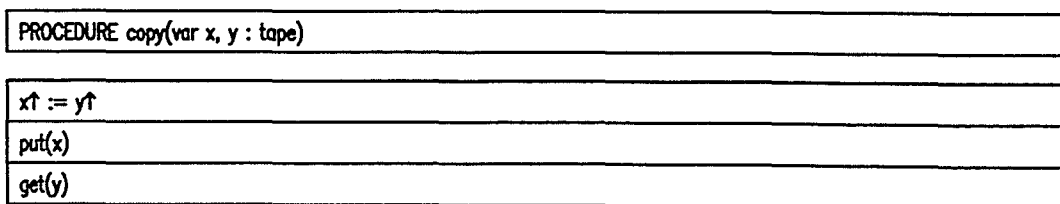
Die Prozedur merge_file_to_eof mischt zwei sequentielle sortierte Dateien unter Verwendung der Prozedur copy.



Prozedur merge_array

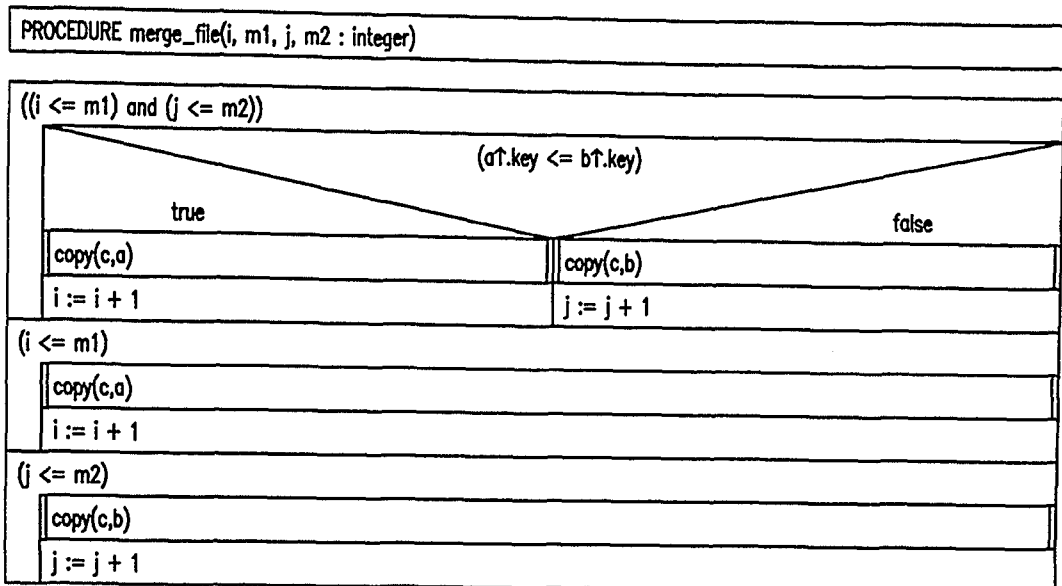


Prozedur merge_file_to_eof



Prozedur copy

Die Prozedur merge_file_to_eof soll nun verändert werden, so daß auch eine bestimmte Anzahl von Elementen gemischt werden kann. Bisher wurden zwei Dateien bis zum Ende abgearbeitet. Die neue Prozedur merge_file entspricht dann der Prozedur merge_array.



Prozedur merge_file

Vorgehensweise des Mergesort-Algorithmus:

1. Zerlege die Liste C in zwei Hälften A und B.
2. Mische A und B durch Kombination einzelner Elemente zu geordneten Paaren.
3. Bezeichne die gemischte Liste mit C und wiederhole Schritt 1. und 2., wobei dieses Mal die geordneten Paare zu geordneten Quadrupeln zusammenzufassen sind.
4. Wiederhole die voranstehenden Schritte durch Mischen der Quadrupel zu Octupel, und fahre damit so lange fort, indem jedesmal die Längen der gemischten Listen verdoppelt werden, bis die ganze Liste geordnet ist.

Spezialfall: Listenlänge $n = 2^k, k > 0$

Beispiel 4.9: Sortieralgorithmus *Mergesort*

$$n = 2^3$$

C : 5 1 8 7 10 3 12 2

1. Verteilung der Liste C auf zwei Listen A und B

A:	5		1		8		7
B:	10		3		12		2

2. 4 Mischvorgänge der Länge 1 zu geordneten Paaren

C: 5 10 | 1 3 | 8 12 | 2 7

3. C verteilen auf A und B

A : 5 10 | 1 3

B : 8 12 | 2 7

4. 2 Mischvorgänge der Länge 2 zu geordneten Quadrupeln

C: 5 8 10 12 | 1 2 3 7

5. C verteilen auf A und B

A: 5 8 10 12

B: 1 2 3 7

6. Ein Mischvorgang der Länge 4 liefert dann die geordnete Liste

C: 1 2 3 5 7 8 10 12

Nun soll das Sortieren durch Mischen einmal unter Verwendung von Arrays (mergesort_array) und einmal unter Verwendung von Dateien (mergesort_file) vollständig implementiert werden.

PROCEDURE mergesort_array
n1 := n div 2
k := log2 (n)
FOR i := 1 TO k
distribute_array
length := 2 ** (i-1)
lmax := n div (2 ** i)
FOR l := 1 TO lmax
anfx := (l-1) * length + 1
endx := (l * length)
anfxc := (l-1) * 2**i + 1
merge_array(anfx, endx, anfx, endx, anfxc)

Prozedur mergesort_array

PROCEDURE distribute_array
l := 1
FOR j := 1 TO n1
a[j] := c[l]
l := l + 1
FOR j := 1 TO n1
b[j] := c[l]
l := l + 1

Prozedur distribute_array

PROCEDURE mergesort_file
n1 := n div 2
k := log2 (n)
FOR i := 1 TO k
distribute_file
rewrite(c)
reset(a)
reset(b)
length := 2 ** (i-1)
lmax := n div (2 ** i)
FOR l := 1 TO lmax
anfx := (l-1) * length + 1
endx := (l * length)
merge_file(anfx,endx,anfx,endx)

Prozedur mergesort_file

PROCEDURE distribute_file
reset(c)
rewrite(a)
rewrite(b)
FOR j := 1 TO n1
copy(a,c)
FOR j := 1 TO n1
copy(b,c)

Prozedur distribute_file

Zeitkomplexität:

Die Dateien A, B und C werden $k = (\log n)$ -mal verteilt und gemischt. Das Verteilen und Mischen fordert $O(n)$ Operationen.

Damit ergibt sich:

$$T_{\text{mergesort}}^{\text{av}}(n) = T_{\text{mergesort}}^{\text{w}}(n) \in O(n \log n).$$

Vorteile des Algorithmus *Mergesort*:

- Niedrige Komplexitätsordnung (auch im worst-case).
- Der Mergesort-Algorithmus sortiert (auch) extern.

Nachteile des Algorithmus *Mergesort*:

- Wird das Verfahren als internes Sortierverfahren verwendet, so ist je nach Implementation zusätzlicher Speicherplatz erforderlich.
- Programmierung für beliebiges n ist aufwendig.
- die Konstanten bezüglich der O-Notation sind u.U. sehr groß, d.h. im average-Fall ist der Algorithmus z.B. nicht so schnell wie Quicksort.

4.3 Hash-Verfahren

Motivation und generelle Idee

Suchbäume in der höhenbalancierten Version (z.B. B-Baum) erlauben sehr effiziente Zugriffe zu Elementen großer Mengen. Wenn eine Tabelle $n = 10.000$ Elemente enthält, so ist in $\sim \log_2 n \simeq 13$ Schritten jedes beliebige Element gefunden. Um das Zugriffsverhalten zu Elementen einer Menge im average-case günstiger als $\log n$ zu gestalten, bedarf es einer radikal veränderten Strategie zur Speicherung von Schlüsseln.

Annahme:

In einem Feld a vom Typ *hasharray* (*hasharray* = *array* [1.. n] of *stdelement*) sollen Elemente einer Menge gespeichert werden. Darüberhinaus existiere eine 'black box' mit Namen *address calculator*. Wir wollen zunächst den inneren Aufbau dieser black box nicht weiter beschreiben, aber wir gehen davon aus, daß dieser *address calculator* die folgende Aufgabe erfüllt.

Wann immer ein neues Element in die Tabelle eingefügt werden soll, geben wir dem *address calculator* den Schlüssel X dieses Elementes, und er liefert den Feldindex i , in den dieses Element eingefügt werden kann.

Dieses Vorgehen soll auch bei Retrieve- und Locate-Operationen anwendbar sein.

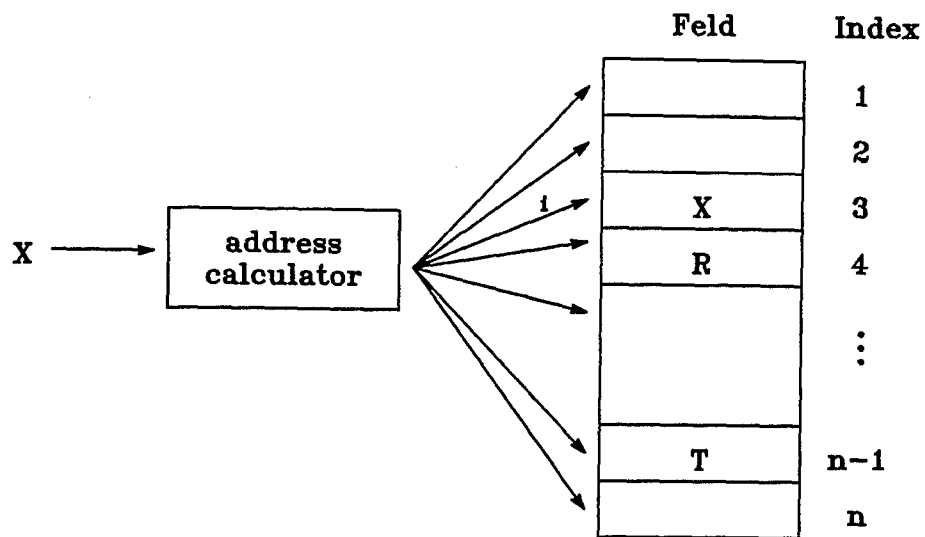


Abb. 36. address calculator

Eine Insert-Operation unter Zuhilfenahme des *address calculators* ist damit eine triviale Aufgabe.

```
PROCEDURE insert(var a: hasharray; newitem: stdelement)
```

```
  i := address_calculator(newitem.key)
```

```
  a[i] := newitem
```

Prozedur insert

Für Retrieve- und Delete-Operationen kann der address calculator ebenso verwendet werden.

Das Zeitverhalten von insert und locate wird bei einem solchen Vorgehen maßgeblich von der Anzahl der Schritte bestimmt, die der address calculator benötigt, um aus dem Schlüssel eine Adresse im Feld zu berechnen.

Dieses beschriebene Schema ist eine Idealisierung des *Hashing*. Für die Implementierung eines solchen Schemas muß ein address calculator konstruiert werden, der mit wenig Aufwand die Adresse eines Elementes liefert. So ein address calculator wird gewöhnlich als Hash-Funktion bezeichnet.

Definition 4.31: Hash-Funktion

Sei S eine Schlüsselmenge und I eine Adreßmenge im weitesten Sinn.

Dann heißt $h : S \rightarrow I$ Hash-Funktion.

Die Bildmenge $h(S) \subseteq I$ bezeichnet die Menge der *Hash-Indizes*.

Bemerkung:

Die Schlüsselmenge ist i.allg. *sehr viel* größer als die verfügbare Adreßmenge, d.h. die Hash-Funktion wird i.allg. nicht injektiv sein (d.h. mehrere Schlüssel haben dieselbe Adresse).

Definition 4.32: Kollision

Sei S Schlüsselmenge, h Hash-Funktion.

Ist für $s_1 \neq s_2$ (mit $s_i \in S$) $h(s_1) = h(s_2)$, so spricht man von einer Kollision.

Für die nachfolgenden Hash-Verfahren setzen wir voraus, daß die Daten mit wahlfreiem Zugriff gespeichert sind.

Für den Fall einer Kollision werden je nach Verfahren unterschiedliche Strategien verfolgt werden, um das Element dennoch abzuspeichern bzw. zu finden.

4.3.1 Offene Hash-Verfahren

Die prinzipielle Vorgehensweise bei offenen Hash-Verfahren kann wie folgt beschrieben werden:

1. Aus dem Schlüssel eines Datensatzes wird mit Hilfe einer geeigneten Hash-Funktion (diese muß der Art und dem Umfang der Schlüsselmenge, sowie den verfügbaren Speicheradressen angepaßt sein) eine Speicheradresse errechnet. Ist der so berechnete Speicherplatz frei, so wird der Datensatz inklusive Schlüssel dort gespeichert.
2. Ist der errechnete Speicherplatz durch einen anderen Datensatz belegt, so liegt eine Kollision vor. In diesem Fall wird eine Ersatzadresse berechnet und der Speicher-versuch wiederholt. Ist der berechnete Speicherplatz wiederum belegt, so wird so lange eine neue Ersatzadresse berechnet, bis ein freier Platz gefunden wird oder der verfügbare Speicher ganz durchlaufen wurde.

3. Das Lesen (Suchen) von Elementen erfolgt analog, bis das gesuchte Element gefunden ist.

Eine Voraussetzung für eine gute Hash-Funktion ist eine möglichst gleichmäßige Verteilung der Schlüssel auf den Bereich der Indexwerte. Neben dieser Bedingung ist die Verteilung an kein Muster gebunden, und es ist wünschenswert, wenn sie völlig zufällig zu sein scheint.

Beispiel 4.10:

Eine Firma hat zur eindeutigen Identifikation ihrer Mitarbeiter Personalnummern im Bereich 1...9999 vergeben. Sie hat zur Zeit nur 60 Mitarbeiter und hat zur Speicherung der Personaldaten deshalb nur 100 Speicherplätze für Datensätze bereitgestellt.

Der Zugriff zu den Daten erfolgt anhand der Personalnummer mit Hilfe eines Hash-Verfahrens.

Schlüsselmenge : $S = \{1..9999\}$
 Hash-Indizes : $I = \{1..100\}$
 Hash-Funktion : $h : S \rightarrow I$ mit $h(s_k) = s_k \bmod 100 + 1$.

Bei einer Kollision wird der berechnete Hash-Index solange um 1 erhöht, bis ein freier bzw. der gesuchte Datenrecord gefunden ist.

S	Name	h(s)	
1233	Müller	34	
2034	Meier	35	
9539	Schulze	40	
3433	Schmidt	34	⇒ Kollision ⇒ 1.Ersatzadresse 35 ⇒ Kollision ⇒ 2.Ersatzadresse 36

Wiederfinden von Datensätzen:

S	Name	h(s)	
2034	Meier	35	⇒ Datensatz gefunden
3433	Schmidt	34	⇒ Datensatz nicht gefunden ⇒ 1.Ersatzadresse 35 ⇒ Datensatz nicht gefunden ⇒ 2.Ersatzadresse 36 ⇒ Datensatz gefunden
7334	Johannes	35	⇒ Datensatz nicht gefunden ⇒ 1.Ersatzadresse 36 ⇒ Datensatz nicht gefunden ⇒ 2.Ersatzadresse 37 ⇒ Speicherplatz leer ⇒ Datensatz nicht gefunden.

Bemerkung:

Das Berechnen der Ersatzadresse heißt *Sondieren* oder *Probing*.

4.3.1.1 Der Divisions-Hash**Definition 4.33:** Divisions-Hash mit linearem Sondieren

Ein Hash-Verfahren, das die Hash-Indizes aus dem Schlüssel mit Hilfe der Modulo-Funktion (Divisionsrest) berechnet, heißt Divisions-Hash.

Wird die Ersatzadresse bei jeder Kollision durch Erhöhen der alten Adresse um 1 berechnet, so spricht man von linearem Sondieren (linearem Probing). Die i -te Ersatzadresse für einen Schlüssel s mit Hash-Index $h(s)$ wird also wie folgt berechnet:

$$e_i(s) = (h(s) + i) \bmod n$$

$(e_0(s) = h(s))$ ist der Hash-Index der Hash-Funktion selbst)

Bemerkung:

Ein Nachteil des Divisions-Hash mit *linearem Sondieren* ist, daß sich an Stellen mit häufigen Kollisionen leicht Ketten bilden, so daß sich die Wahrscheinlichkeit für weitere Kollisionen in diesem Bereich noch erhöht.

Beispiel 4.11: Speichern von aufeinanderfolgenden Schlüsselns bei linearem Sondieren

S	Name	h(s)	
1234	Schmitt	35	
2034	Huber	35	⇒ Kollision ⇒ Ersatzadresse 36
2134	Meyer	35	⇒ Kollision ⇒ Ersatzadresse 36 ⇒ Kollision ⇒ Ersatzadresse 37
2135	Hansen	36	⇒ Kollision ⇒ Ersatzadresse 37 ⇒ Kollision ⇒ Ersatzadresse 38
2136	Braun	37	⇒ Kollision ⇒ Ersatzadresse 38 ⇒ Kollision ⇒ Ersatzadresse 39
2137	Reitz	38	⇒ Kollision ⇒ Ersatzadresse 39 ⇒ Kollision ⇒ Ersatzadresse 40

→ Kette: Es ergeben sich 9 Kollisionen (statt zwei).

Bemerkung:

Im folgenden sei $I = \{0, \dots, n-1\}$ die Menge der Hash-Indizes. Der Nachteil des linearen Sondierens ist, daß es die Schlüssel nach Kollision nicht wirklich verstreut, sondern dicht am Ort der Kollision beläßt (bzw. speichert). Besser ist eine möglichst breite, "zufällige" Streuung.

Aufgrund der häufigen Kollisionen ist das Verfahren des linearen Sondierens nicht optimal, bessere Ergebnisse liefern im Normalfall andere Varianten des Sondierens, die *Quadratische Sondiervverfahren* genannt werden.

Definition 4.34: Quadratisches Sondieren

Die Strategie, bei der die Funktion $e_i(s) = (h(s) + i^2) \bmod n$ zur Berechnung der Ersatzadresse gewählt wird, heißt quadratisches Sondieren.

Beispiel 4.12:

Sei $n = 11$ (11 Speicheradressen)

i	0	1	2	3	4	5	6	7	8	9	10
i^2	0	1	4	9	16	25	36	49	64	81	100
$i^2 \bmod n$	0	1	4	9	5	3	3	5	9	4	1

Bemerkung:

Ein Nachteil ist offensichtlich, daß $e_i(s) = e_{n-i}(s)$ gilt, d.h. nicht alle zur Verfügung stehenden Adressen werden erreicht.

Der *vollständige Hash* nutzt den gesamten Speicherplatz aus. Es gelten die folgenden zwei Sätze.

Satz 4.1: (ohne Beweis)

Ist n eine Primzahl, so sind die Zahlen $i^2 \bmod n$ für $0 \leq i \leq \frac{n}{2}$ paarweise verschieden. Hiermit läßt sich also bei geeigneter Wahl der Tabellengröße immerhin der halbe Speicherplatz überdecken.

Weiter gilt:

Satz 4.2: (ohne Beweis)

Ist $n \equiv 3 \pmod{4}$ (sprich: n kongruent 3 modulo 4, d.h. n läßt bei Division durch 4 den Rest 3) und n Primzahl, so ist die Menge der Reste modulo n von

i^2 für i ungerade

$-i^2$ für i gerade (inkl. $i=0$)

ein volles Restsystem modulo n , d.h. alle möglichen Reste kommen genau einmal vor.

Als Resultat ergibt sich: Die Funktion

$$e_i(s) := (h(s) + (-1)^{i+1} i^2) \pmod{n}$$

mit $n \equiv 3 \pmod{4}$ und n Primzahl stellt n verschiedene Ersatzadressen bereit.

Beispiel 4.13: Vollständiger Hash mit $n=11$

$n=11$ erfüllt die oben genannte Voraussetzung, wegen $(11 \div 4) = 2$ mit einem Rest von 3.

Für $i=0, \dots, 10$ ergeben sich die folgenden Ersatzadressen e_i

i	0	1	2	3	4	5	6	7	8	9	10
$(-1)^{i+1} i^2 \pmod{n}$	0	1	7	9	6	3	8	5	2	4	10

$$i=4: \Rightarrow (-1)^5 = -1, \quad i^2 = 16 \Rightarrow -i^2 \pmod{11} = -5 \equiv 6$$

Bemerkung:

- Ist $(-1)^{i+1} i^2 \pmod{n}$ negativ, so muß der Wert um n bzw. Vielfache von n erhöht werden.
- Laufen die Adressen nicht von 0 bis $n-1$, sondern von 1 bis n , so ist die Ersatzadresse entsprechend um 1 zu erhöhen, andere Index-Bereiche sollten vorab auf den Adreßbereich 0 bis $n-1$ transformiert werden.

4.3.1.2 Der Fibonacci-Hash

Nachteil des Divisions-Hash ist, daß aufeinanderfolgende Schlüssel auch auf aufeinanderfolgende Adressen gespeichert werden. Eine bessere Verteilung liefert die folgende Hash-Funktion:

Definition 4.35: Fibonacci-Hash

Ein Hash-Verfahren, das die Hash-Indizes aus dem Schlüssel mit Hilfe der Formel

$$h(s) = \lfloor (n \bullet s \bullet \Theta) \bmod n \rfloor$$

berechnet, heißt Fibonacci-Hash.

Dabei muß für Θ der Wert $\frac{1}{2}(\sqrt{5} - 1) \approx 0.618034$ eingesetzt werden.

Bemerkung:

Die Zahl Θ bestimmt den goldenen Schnitt.

Goldener Schnitt: $\frac{b}{a} = \frac{a+b}{b}$, mit $\frac{a}{b} = \Theta$ gilt:

$$\Theta^2 + \Theta - 1 = 0 \Rightarrow \frac{1}{\Theta} = \frac{\Theta}{1 - \Theta}$$

Definition 4.36: Fibonacci-Folge

Sei $f(0) = 0$ und $f(1) = 1$, dann heißt

$$f(n) = f(n-1) + f(n-2)$$

die n-te Fibonacci-Zahl.

n	0	1	2	3	4	5	6	7	8	9	10	11
f(n)	0	1	1	2	3	5	8	13	21	34	55	89

Man kann zeigen, daß die Fibonacci-Zahlen durch die geschlossene Formel

$$f(n) = \left[\frac{1}{\sqrt{5}} \frac{1}{\Theta^n} \right] \quad \forall n \in \mathbb{N}$$

berechnet werden können.

Satz 4.3: (Swierczkowski, 1958)

Sei n die Größe des Speichers und Θ irgendeine irrationale Zahl. Betrachte für $s=0,1,\dots,N$ (N beliebig) die Vielfachen:

$$g(s) = (n \bullet s \bullet \Theta) \bmod n$$

Dann gilt: Die $N+1$ Teilbereiche (Segmente) haben höchstens 3 verschiedene Längen, und jede neue Zahl s_i teilt einen größten Teilbereich von $[0,n]$.

Beispiel 4.14: Swierczkowski-Verteilung

Sei $n=5$, $\Theta = \frac{1}{2}(\sqrt{5}-1)$, $N=15$

σ	$5 \bullet \Theta \bullet \sigma$	$g(\sigma)$	$[g(\sigma)]$
0	0.00	0.00	0
1	3.09	3.09	3
2	6.18	1.18	1
3	9.27	4.27	4
4	12.36	2.36	2
5	15.45	0.45	0
6	18.54	3.54	4
7	21.63	1.63	2
8	24.72	4.72	5
9	27.81	2.81	3
10	30.90	0.90	1
11	33.99	3.99	4
12	37.08	2.08	2
13	40.17	0.17	0
14	43.26	3.26	3
15	46.35	1.35	1

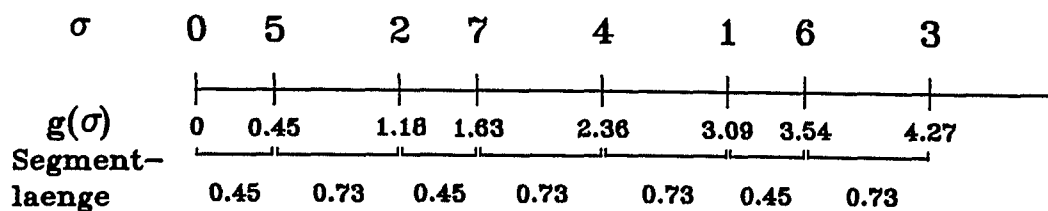


Abb. 37. Darstellung der Swierczkowski-Verteilung für $N=7$

Wenn alle 16 Adressen gespeichert sind, so treten 3 verschiedene Längen auf. Man kann (vereinfacht) sagen, daß $g(\sigma)$ eine reelle Distanz-Maximierungsfunktion

approximiert. Durch die Funktion $g(\sigma)$ werden die Schlüssel $\sigma = 0, \dots, N$ (N beliebig), für einen gegebenen Modul n auf das Intervall $[0, n]$ verteilt. Die Güte dieser Verteilung bzw. Streuung hängt von der Wahl der Konstanten Θ ab. Es läßt sich zeigen, daß $\Theta = \frac{1}{2}(\sqrt{5} - 1)$ zu einer optimalen Streuung führt (Beweis: Knuth[7], S. 688 ff.).

Naive Begründung:

Man würde es selbstverständlich als optimale Situation ansehen, wenn alle Schlüssel äquidistant zueinander liegen. Durch die Zerstörung dieser Situation bei Einfügung eines neuen Schlüssels zeigt sich die dynamische Instabilität einer solchen Situation. Das hier beschriebene Verfahren ist dagegen dynamisch stabil.

Sei c_1 die kleinste, c_2 die mittlere und c_3 die größte Länge aus Satz 4.3. Die Güte des Verfahrens kann man durch die beiden Quotienten $\frac{c_1}{c_2}$ und $\frac{c_2}{c_3}$ beschreiben.

Die Gleichverteilung der Schlüssel im Hash-Feld ist um so besser, je weniger die Quotienten $\frac{c_1}{c_2}$ und $\frac{c_2}{c_3}$ voneinander abweichen.

Für $\Theta = \frac{1}{2}(\sqrt{5} - 1)$ gilt:

$$\frac{c_1}{c_2} = \frac{c_2}{c_3} = 0.61\dots$$

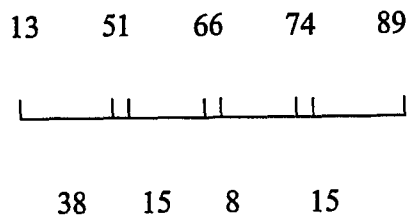
Beispiel 4.15: (Vergleiche Beispiel 4.10)

$S = \{1..9999\}$, $n = 100$ (Tabellengröße)

Speichern aufeinanderfolgender Schlüssel:

S	Name	h(s)
1234	Schmitt	$\lfloor 100 \bullet 1234 \bullet 0.618034 \bmod 100 \rfloor + 1 = 66$
2134	Meyer	89
2135	Hansen	51
2136	Braun	13
2137	Reitz	74

Verteilung im Feld:



Bemerkung:

Aufeinanderfolgende Schlüssel werden bei diesem Verfahren über den ganzen Adreßbereich verteilt.

4.3.1.3 Zugriffszeiten beim Suchen und Speichern in Hash-Tabellen

Definition 4.37: Schrittzahl

Unter der Schrittzahl $S(s)$ versteht man die Anzahl der zu berechnenden Hash-Indizes und Ersatzadressen, die nötig sind, um den Datensatz mit Schlüssel s zu speichern bzw. wiederzufinden.

Definition 4.38: Füllungsgrad

Der Füllungsgrad einer Hash-Tabelle ist der Quotient $\alpha = \frac{k}{n}$, wobei n die Anzahl der Tabellenplätze (Adressen) und k die Anzahl der belegten Tabellenplätze ist.

Bemerkung:

Für ein ideales Hash-Verfahren gilt:

1. Zu Beginn eines Speicherversuches sind alle möglichen Speicherbelegungen gleich wahrscheinlich.
2. Die Anzahl der Speicherversuche ist unabhängig davon, welche Adressen bereits belegt sind.

Bei einem solchen (idealen) Hash-Verfahren ist der Erwartungswert für die mittlere Schrittzahl $S(n,k)$ bei der Speicherung eines Datensatzes in einer Tabelle mit n Plätzen, von denen k bereits belegt sind, gegeben durch:

$$S(n,k) = \frac{n+1}{n-k+1}$$

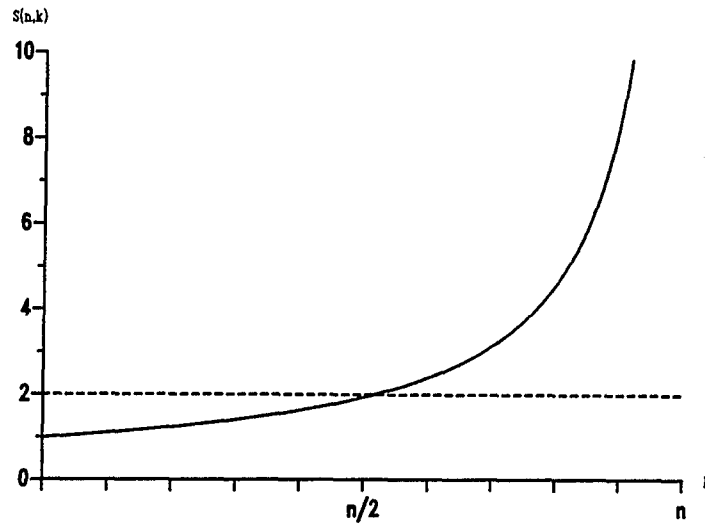


Abb. 38. Qualitativer Verlauf von $S(n,k)$ für ein festes n

Beispiel 4.16: Anzahl der Kollisionen beim Speichern eines neuen Datenrecords

Sei $n=991$. Die Records werden in der Reihenfolge $k=1..\max n$ eingetragen (k bezeichnet die laufende Nummer des Records, $\max n$ entspricht $n-1=990$).

Auf der x-Achse wird die Anzahl der schon gespeicherten Records-1 aufgetragen, auf der y-Achse die Anzahl der Kollisionen beim Speichern des Records (gemittelt über 200 verschiedene Hash-Tabellen).

Die Verteilung aller Schlüssel ist zufällig (Zufallszahlen; alle Schlüssel sind gleichwahrscheinlich).

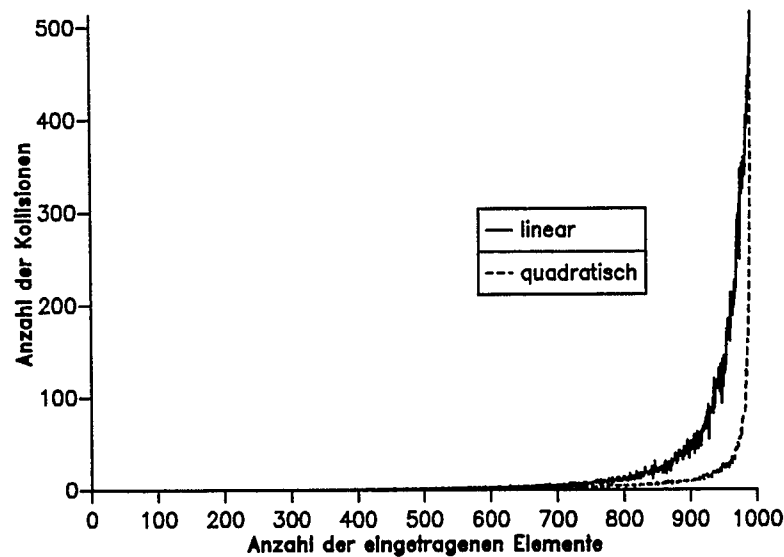


Abb. 39. Kollisionen beim Fibonacci-Hash (mit linearem bzw. quadratischen Sondieren)

Zusammenfassung der Eigenschaften des offenen Hash-Verfahrens:

- Zugrundeliegende Datenstruktur: Speicher mit wahlfreiem Zugriff (z.B. Array, Array of Records, etc.).
- Schlüssel werden zusammen mit den Daten gespeichert. Schlüssel sind eindeutig.
- Es muß erkennbar sein, ob ein Speicherplatz belegt ist oder nicht (z.B. Schlüsselfeld leer, extra Bit, etc.).
- Beim Füllen der Tabelle müssen die auftretenden Schlüssel möglichst gleichmäßig auf die verfügbaren Adressen verteilt werden (keine Häufung).
- Beim Berechnen der Ersatzadressen müssen alle möglichen Adressen (genau einmal) durchlaufen werden.
- Bei festem Füllungsgrad ist die mittlere Schrittzahl beim Suchen niedriger als die beim Speichern.
- Die verwendete Methode des Sondierens hat häufig größeren Einfluß auf die Qualität des Hash-Verfahrens als die Wahl der Hash-Funktion.
- Die Tabellengröße kann nicht dynamisch verändert werden. Beim quadratischen Sondieren muß sie zusätzlich eine Primzahl $n \equiv 3 \pmod{4}$ sein.
- Das Entfernen von Datenobjekten ist nicht ohne weiteres möglich. Eine Möglichkeit besteht darin, das Datenobjekt zwar zu entfernen, die Speicherzelle aber nur als gelöscht zu markieren, so daß auch Datenobjekte mit Ersatzadressen noch erreicht werden.

4.3.2 Hash in Teillisten

Prinzip: Die Hash-Tabelle besteht nicht wie beim offenen Hash-Verfahren aus einer Liste mit n Plätzen, sondern aus n Teillisten mit jeweils k Plätzen.

1	2		k
2			
.			
.			
.			
.			
.			
.			
.			
n			

Speicherung:

1. Wie beim offenen Hash-Verfahren wird zunächst mit Hilfe der offenen Hash-Funktion aus dem Schlüssel der Hashindex ($1 \leq h(s) \leq n$) berechnet. Dieser gibt an, in welcher Teilliste der Datensatz gespeichert wird.
2. Innerhalb der Teillisten wird sequentiell gespeichert.

Bemerkung:

- Die Speicherung in der (sequentiellen) Teilliste kann effizient vorgenommen werden, wenn zu jeder Teilliste ein Pointer existiert, der jeweils auf den ersten freien Listenplatz zeigt.
- Das Suchen in der Teilliste bleibt sequentiell.

Realisierung eines Hash-Algorithmus in Teillisten

1. Repräsentation

```
TYPE    daten    = RECORD
        key      : INTEGER;
        name     : PACKED ARRAY [1..20] OF CHAR;
        ort      : PACKED ARRAY [1..20] OF CHAR
END; (* daten *)
```

```
CONST n      = 1000;  
      k      = 10;
```

```
VAR  tabelle : ARRAY [1..n, 1..k] OF daten;  
     pointer : ARRAY [1..n] OF 1..k+1;
```

Bemerkung:

Das Feld Pointer enthält die Indizes des jeweils ersten freien Platzes in jeder Teilliste und dient gleichzeitig dazu, um festzustellen, ob eine Teilliste voll ist.

Die Teillisten können als verkettete Liste implementiert werden. Beim Speichern eines Datensatzes ist dann Einfügen am Anfang sinnvoll.

Schrittzahl beim Suchen:

Sei n die Anzahl der Teillisten und m die Anzahl der gespeicherten Records. Bei idealer Verteilung der Schlüssel entfallen $\frac{m}{n}$ Elemente auf jede Teilliste. Damit dauert das Suchen $O(\frac{m}{n})$ Schritte ($m \geq n$).

Ist $m < n$, so ist mindestens ein Suchschritt nötig.

Folgerung:

Um schnell suchen zu können, müssen größenordnungsmäßig so viele Teillisten wie Records vorhanden sein.

Vergleich: Offene Hash-Verfahren - Hash in Teillisten

- Die ersten vier Punkte der Zusammenfassung der Eigenschaften offener Hash-Verfahren gelten auch für den Hash in Teillisten.
- Der Speicherplatzbedarf ist bei Hash in Teillisten im allgemeinen höher, denn die Hash-Tabelle ist schon voll, wenn eine Teiltabelle voll ist (unter Umständen ist nur $\frac{1}{n}$ der Tabelle belegt).
- Die Zugriffszeit beim Hash in Teillisten ist nur dann günstig, wenn die Teillisten kurz sind. Diese entspricht einer Situation mit wenigen Kollisionen beim offenen Hash.
- Löschen von Elementen ist beim Hash in Teillisten ohne Probleme möglich!

4.4 Algorithmen auf Graphen

Nachdem in dem Kapitel über die Datenstrukturen die Graphen zur Modellierung von Beziehungen zwischen Objekten eingeführt worden sind und kurz auf die Entstehung der Graphentheorie eingegangen worden ist, sollen nun Algorithmen für das Durchlaufen von Graphen sowie zur Bestimmung kürzester Wege in Graphen beschrieben werden. Im folgenden sei $n = |V|$.

4.4.1 Das Durchlaufen von gerichteten Graphen

Um Probleme mit gerichteten Graphen effizient lösen zu können, ist es häufig notwendig, die Knoten und Kanten eines Graphen in einer systematischen Art und Weise zu durchlaufen. Hier werden die beiden Durchlaufstrategien Depth-First-Search und Breadth-First-Search beschrieben. Beide Durchlaufstrategien können als 'Skelett' für viele Algorithmen angesehen werden, die verschiedene Probleme sehr effizient lösen (z.B. Bestimmung der Zusammenhangskomponenten eines Graphen).

4.4.1.1 Der Algorithmus Depth-First-Search

Der Algorithmus *Depth-First-Search* (DFS), eine Verallgemeinerung des Preorder-Algorithmus auf Bäumen, ist eine sehr hilfreiche Methode für diese Aufgabe. DFS kann als 'Skelett' für viele Algorithmen angesehen werden, die verschiedene Probleme sehr effizient löst. Die prinzipielle Vorgehensweise des Algorithmus lautet:

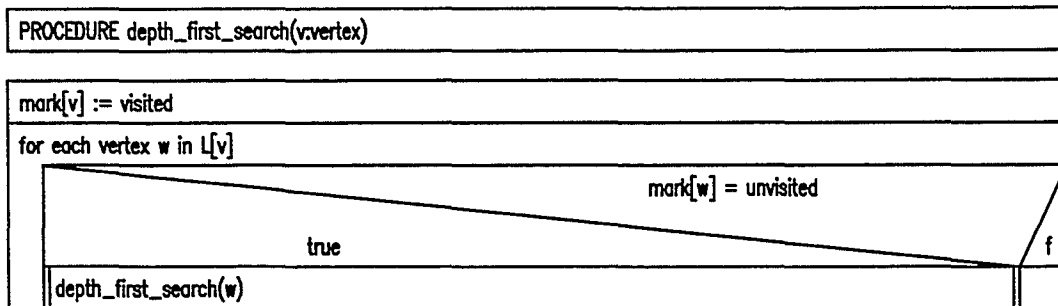
1. Markiere alle Knoten eines Graphen zu Beginn als nicht besucht.
2. Wähle einen beliebigen Knoten v als Startknoten.
3. Dieser Knoten wird als besucht markiert.
4. Jeder Knoten, der adjazent zu v und noch nicht als besucht markiert ist, wird als nächster Knoten durch rekursiven Aufruf von DFS besucht.
5. Wenn alle Knoten, die von v erreichbar sind, besucht worden sind, ist die Suche für v beendet.
6. Wenn noch Knoten existieren, die als nicht besucht markiert sind, wird einer dieser Knoten als v ausgewählt, und man fährt mit Schritt 3 des Algorithmus fort.

Diese Vorgehensweise wird DFS genannt, weil mit der Suche in Vorwärtsrichtung (tiefer, deeper) fortgefahren wird, solange dies möglich ist. Es gilt:

$L[v]$: Adjazenzliste, enthält alle Knoten, die adjazent zu v sind.

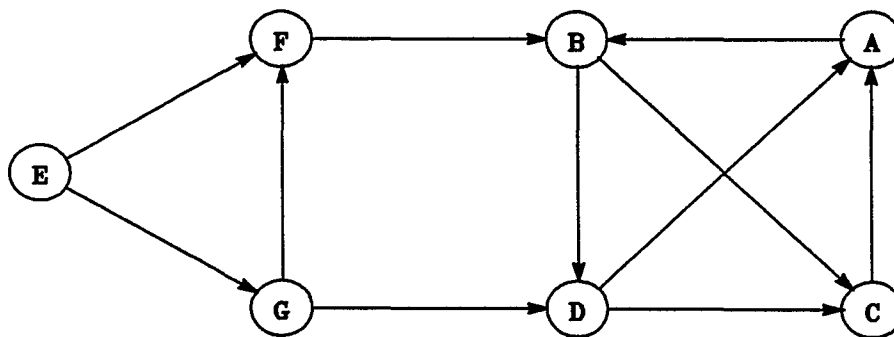
mark: Feld, dessen Elemente (Knoten) 'visited' oder 'unvisited' sein können. Alle Knoten sind zu Beginn unvisited.

DFS wird für jeden Knoten aufgerufen, der noch nicht markiert ist, bis alle Knoten des Graphen besucht worden sind.



Prozedur depth_first_search

Beispiel 4.17: Depth-First-Search



Wende die Prozedur DFS auf den Graphen mit $v = A$ an.

Adjazenzlisten:

A	:	B
B	:	C, D
C	:	A
D	:	A, C
E	:	F, G
F	:	B
G	:	D, F

Der Algorithmus markiert A als besucht und wählt B aus der Adjazenzliste von A aus. Da B nicht markiert ist, wird die Prozedur rekursiv mit $v = B$ aufgerufen. B wird markiert.

Die Adjazenzliste von B enthält die Knoten C und D. Sei C der erste Knoten in der Liste, dann wird die Prozedur mit $v=C$ aufgerufen. C wird markiert. Der einzige Knoten in der Adjazenzliste von C ist A. Da dieser Knoten schon markiert ist, wird mit der Suche bei B fortgefahren und die Prozedur mit $v=D$ aufgerufen. Die Knoten in der Adjazenzliste von D sind bereits markiert, daher geht die Suche zurück zu B und dann zu A. Damit ist die Suche von A aus beendet. Es gibt aber noch Knoten E, F und G, die noch nicht markiert sind. Um die Suche zu vervollständigen, kann DFS mit $v=E$ aufgerufen werden.

Zeitkomplexität des Depth-First-Search-Algorithmus:

Die Zeitkomplexität des Depth-First-Search-Algorithmus bei Graphen mit $|V|$ Knoten und $|E|$ Kanten beträgt $O(|E|)$. DFS wird nur für Knoten aufgerufen, die noch nicht als besucht markiert sind. Für jeden dieser Knoten werden alle von diesem Knoten ausgehenden Kanten betrachtet, d.h. die Anzahl der Durchläufe in der inneren FOR-Schleife ist genau die Summe aller Längen von Teillisten für alle Kanten. Damit entspricht die Zeitkomplexität bis auf einen konstanten Faktor genau der Zeit, die benötigt wird, um jede Kante des Graphen 'anzusehen'.

$$T_{\text{depth_first_search}}^{\text{av}}(|E|) = T_{\text{depth_first_search}}^{\text{w}}(|E|) = T_{\text{depth_first_search}}^{\text{b}}(|E|) \in O(|E|).$$

Falls noch Knoten als nicht besucht markiert sind, wird DFS erneut aufgerufen. Damit erhöht sich die Zeitkomplexität für das Durchlaufen des gesamten Graphen auf:

$$\begin{aligned} T_{\text{depth_first_search}}^{\text{av}}(|V|, |E|) &= T_{\text{depth_first_search}}^{\text{w}}(|V|, |E|) \\ &= T_{\text{depth_first_search}}^{\text{b}}(|V|, |E|) \\ &\in O(|V| + |E|) = O(\max\{|V|, |E|\}). \end{aligned}$$

Bemerkung:

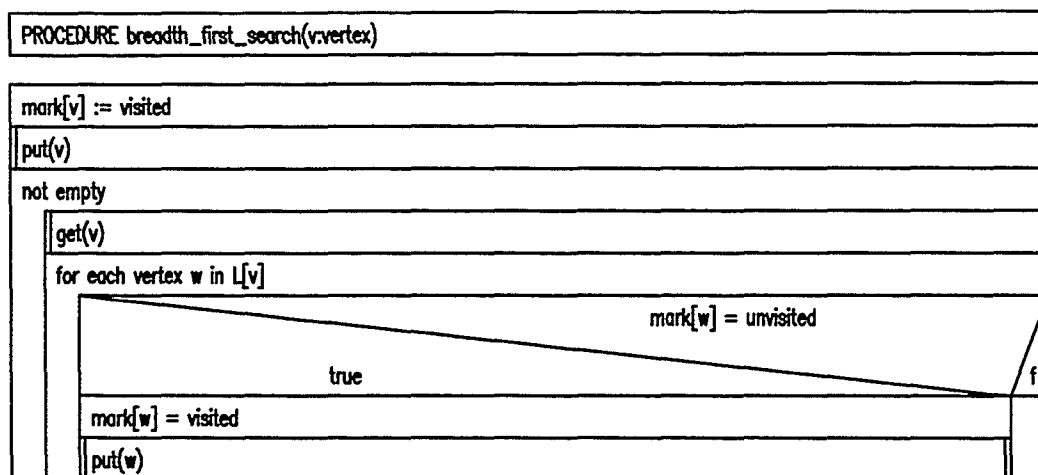
Falls der Algorithmus auf einer Adjazenzmatrix ausgeführt wird, ergibt sich als Zeitkomplexität $O(|V|^2)$.

4.4.1.2 Der Algorithmus Breadth-First-Search

Eine andere Möglichkeit, einen Graphen systematisch zu durchlaufen, ist der Algorithmus *Breadth-First-Search* (BFS). Bei dieser Methode wird die Suche so lange wie möglich 'in die Breite' fortgeführt, d.h. von einem Knoten v aus werden zunächst alle adjazenten Knoten zu v besucht. Die prinzipielle Vorgehensweise des Algorithmus lautet:

1. Markiere alle Knoten eines Graphen zu Beginn als nicht besucht.
2. Wähle einen beliebigen Knoten v als Startknoten.
3. Dieser Knoten wird als besucht markiert.
4. Jeder Knoten, der adjazent zu v und noch nicht als besucht markiert ist, wird markiert und in einer Queue gespeichert.
5. Entferne Knoten v aus der Queue und fahre bei 4. fort, bis die Queue leer ist.
6. Wenn noch Knoten existieren, die als nicht besucht markiert sind, wird einer dieser Knoten als v ausgewählt, und man fährt mit Schritt 3 des Algorithmus fort.

Zusätzlich zu den Datenstrukturen aus dem DFS-Algorithmus wird für BFS eine Queue mit den zugehörigen Operationen verwendet (siehe Kapitel 3.2.4).



Prozedur breadth_first_search

Beispiel 4.18: Breadth-First-Search

Wende die Prozedur BFS auf den Graphen aus Kapitel 4.4.1.1 mit $v = A$ an.

Der Algorithmus markiert A als besucht und wählt B aus der Adjazenzliste von A aus. B wird markiert und in der Queue gespeichert. Da kein weiterer Knoten in der Adjazenzliste von A enthalten ist, wird B aus der Queue entfernt. Die adjazenten Knoten C und D werden nacheinander markiert und in der Queue gespeichert. Dann wird der Knoten C aus der Queue entfernt. Der einzige Knoten in der Adjazenzliste von C ist A. Da dieser Knoten schon markiert ist, wird als nächstes der Knoten D aus der Queue entfernt. Die Knoten in der Adjazenzliste von D sind bereits markiert. Da die Queue nun leer ist, wird die Suche von A aus beendet. Um auch hier die Suche zu vervollständigen, kann BFS mit $v = E$ aufgerufen werden.

Zeitkomplexität des Breadth-First-Search-Algorithmus:

Die Zeitkomplexität des Breadth-First-Search-Algorithmus ist wie bei Depth-First-Search $O(\max \{|V|, |E|\})$. Jeder Knoten wird einmal in die Queue geschrieben, so daß die while-Schleife für jeden Knoten genau einmal ausgeführt wird. Jede Kante wird genau einmal untersucht.

4.4.2 Kürzeste Wege von einer Quelle

Dieses Teilkapitel behandelt die Bestimmung der kürzesten Wege von einem festen Knoten (Quelle) zu allen anderen Knoten in einem Graphen (single-source shortest-path).

Gegeben sei ein gerichteter oder ungerichteter Graph $G = (V, E)$, bei dem jeder Kante $e = (v_i, v_j)$ eine positive Bewertung $C(v_i, v_j)$ (auch Kosten genannt) zugeordnet ist.

Bemerkung:

- Ist eine der Bewertungen negativ, so addiere den Betrag der betragsmäßig größten negativen Bewertung zu allen Bewertungen. Es ergibt sich ein Folgeproblem, bei dem alle Kantenbewertungen positiv sind.
- Existiert zwischen zwei Knoten v_i und v_j keine Kante, so sind die Kosten $C(v_i, v_j) = \infty$. Für die Diagonalelemente $C(v_i, v_i)$ gilt: Kosten $C(v_i, v_i) = 0$.

Ferner sei ein Knoten als Quelle markiert. Das Problem besteht darin, die minimalen Kosten von der Quelle zu allen anderen Knoten zu bestimmen, unter Umständen über Zwischenknoten (Kantenbewertungen können Preise, Zeit, o.ä. sein).

Man kann sich z.B. einen Flugplan vorstellen, in dem jeder Knoten eine Stadt und jede Kantenbewertung einer Kante $e = (v_i, v_j)$ die Flugzeit von Stadt v_i nach Stadt v_j repräsentiert. Die Lösung des Problems ergibt dann die minimale Reisezeit von einer gegebenen Stadt zu allen anderen Städten.

Um dieses Problem zu lösen, werden wir den Dijkstra-Algorithmus benutzen, ein Beispiel für einen *greedy*-Algorithmus. Bei einem greedy-Algorithmus werden aufgrund von lokalen Informationen Entscheidungen getroffen, und diese lokal optimalen Entscheidungen führen dann zu einem globalen Optimum.

4.4.2.1 Der Dijkstra-Algorithmus

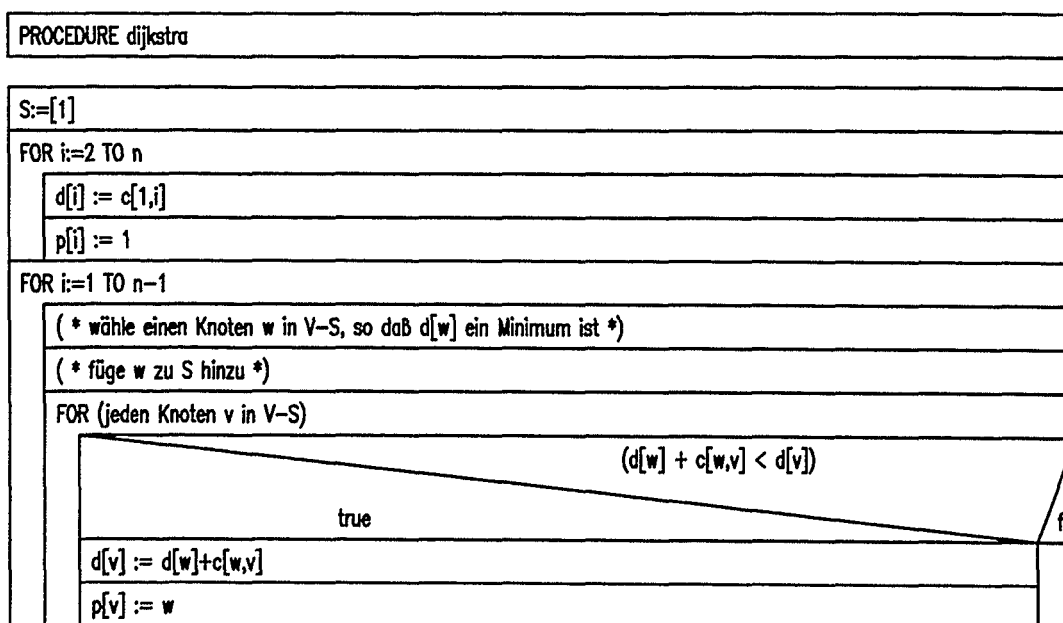
Der Algorithmus verwendet eine Menge S der Knoten, deren kürzeste Entfernungen zur Quelle bereits bekannt sind.

Voraussetzung: $C(u,v) \geq 0 \forall u,v \in V$.

1. Zu Beginn der Ausführung enthält die Menge S nur die Quelle.
2. Der Algorithmus beginnt an der Quelle und berechnet die direkten Wege zu allen erreichbaren Nachbarknoten (analog zum BFS-Algorithmus). Der Knoten mit der kürzesten Entfernung wird zu S hinzugenommen.
3. Dann wird ein Knoten ausgewählt, der noch nicht in S ist und die kürzeste Entfernung zur Quelle hat. Anschließend werden die Entfernungen von diesem Knoten zu allen Knoten, die nicht in S sind, berechnet.

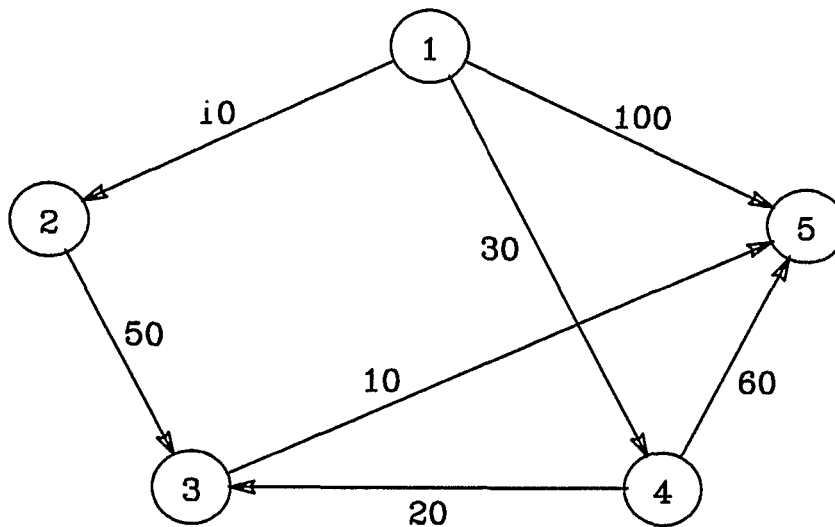
Dieser Schritt wird so lange wiederholt, bis alle Knoten in S enthalten sind.

Das Feldelement $D[v]$ enthält die jeweils aktuelle kürzeste Entfernung des Knotens v von der Quelle. Wenn man zusätzlich zu der kürzesten Entfernung auch den kürzesten Weg zu jedem Knoten haben möchte, kann man ein Feld P verwenden, so daß $P[v]$ den direkten Vorgänger von v auf dem kürzesten Weg enthält. $P[v]$ wird mit der Quelle initialisiert (für alle $v \neq$ Quelle). Am Ende des Algorithmus kann der kürzeste Weg zu jedem Knoten mit Hilfe des Feldes P bestimmt werden.



Prozedur dijkstra

Beispiel 4.19: Veranschaulichung des Dijkstra-Algorithmus



Iteration	S	w	D[2]	D[3]	D[4]	D[5]	P[2]	P[3]	P[4]	P[5]
initial	{1}	-	10	∞	30	100	1	1	1	1
1	{1,2}	2	10	60	30	100	1	2	1	1
2	{1,2,4}	4	10	50	30	90	1	4	1	4
3	{1,2,3,4}	3	10	50	30	60	1	4	1	3
4	{1,2,3,4,5}	5	10	50	30	60	1	4	1	3

Um z.B. die Knoten des kürzesten Weges von 1 nach 5 zu bestimmen, betrachtet man den Vorgänger von 5, d.h. Knoten 3. Der Vorgänger von 3 ist Knoten 4 und der Vorgänger von 4 ist Knoten 1. Damit erhält man den kürzesten Weg von 1 über 4 und 3 nach 5 mit der Länge 60.

Laufzeit des Dijkstra-Algorithmus

1. Verwendet man eine Adjazenzmatrix als Repräsentation des Graphen, so braucht die innere FOR-Schleife $O(|V|)$ Zeit.

Daraus ergibt sich die Zeitkomplexität des Algorithmus:

$$T_{\text{dijkstra}}^{\text{av}}(|V|) = T_{\text{dijkstra}}^{\text{w}}(|V|) = T_{\text{dijkstra}}^{\text{b}}(|V|) \in O(|V|^2).$$

2. Bei der Verwendung von Adjazenzlisten kann die Zeitkomplexität auf $O(\max\{|V|, |E|\})$ reduziert werden, falls $|E| \ll |V|^2$ (siehe [2], S. 207 ff).

4.4.3 Kürzeste Wege zwischen allen Knotenpaaren

Wir wollen nun die Bestimmung der kürzesten Wege zwischen allen Knotenpaaren in einem Graphen behandeln (all-pairs shortest-path). Bei Interpretation des Graphen $G=(V,E)$ als Flugplan kann das allgemeine Problem der kürzesten Wege interpretiert werden als Aufgabenstellung, die verlangt, daß eine Tabelle aller Städte die kürzesten Flugverbindungen zu allen Städten enthält.

Gegeben sei: $G=(V,E)$ und für jede Kante (v,w) eine nicht-negative Bewertung $C(v,w)$.

Aufgabenstellung: Bestimme für alle geordneten Paare (v,w) den kürzesten Weg von v nach w .

Lösungsmöglichkeit:

1. Wende den *Dijkstra*-Algorithmus für alle Knoten v als Quelle an, daraus ergibt sich die Zeitkomplexität $O(n^3)$.
2. Verwende den *Floyd*-Algorithmus.

4.4.3.1 Der Floyd-Algorithmus

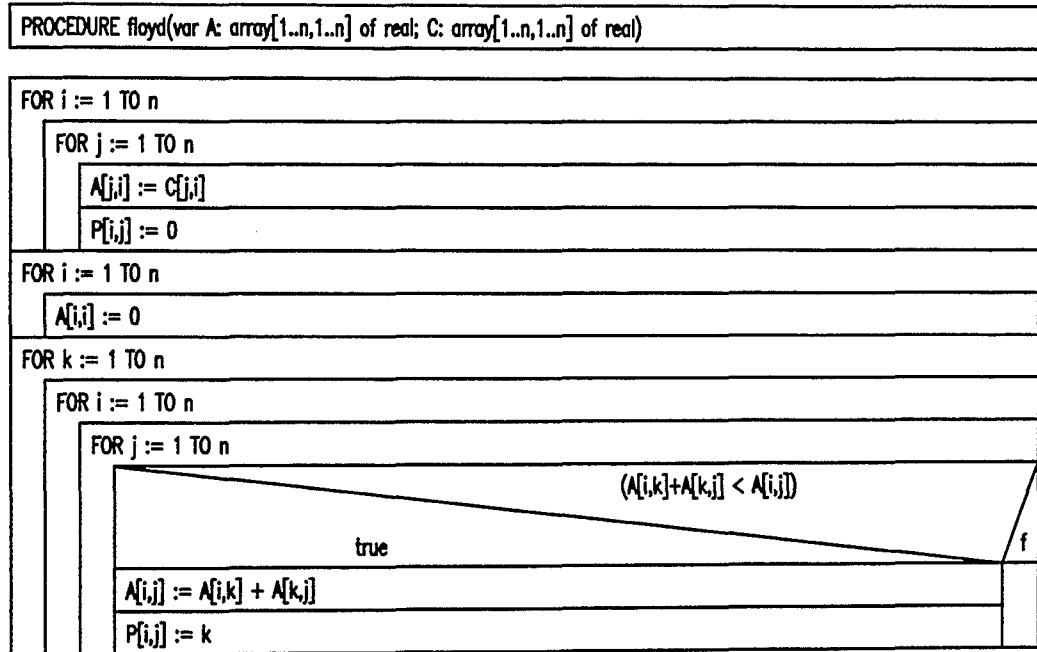
Das Verfahren ist sehr einfach. Im ersten Schritt vergleicht man alle Wege, die von einem beliebigen Knoten i zu einem anderen Knoten j führen, mit dem Umweg über Knoten 1. Ist der Umweg kürzer, so wird der alte Weg durch den Umweg ersetzt. Im zweiten Schritt werden alle Umwege über den Knoten 2 gebildet, im k -ten Schritt über Knoten k usw.

1. Der *Floyd*-Algorithmus nutzt eine $n \times n$ Matrix, um die Längen der kürzesten Wege zu berechnen.
2. Setze zu Beginn $A_0[i,j] = C[i,j]$ für alle i, j .
3. Es werden n Iterationen über die Matrix A gemacht.
In der k -ten Iteration wird die folgende Formel zur Berechnung der neuen $A[i,j]$ benutzt:

$$A_k[i,j] = \min (A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])$$

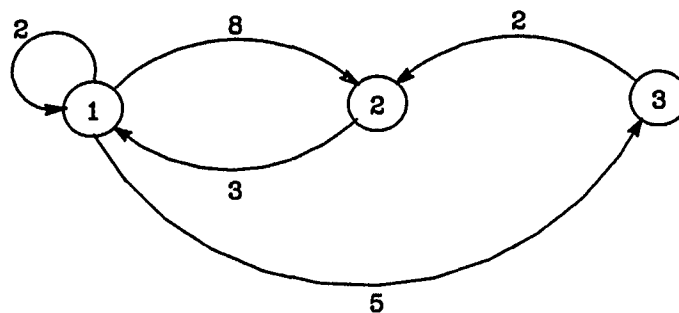
Zum Schluß enthält $A_n[i,j]$ den kürzesten Weg von i nach j , denn nach der k -ten Iteration ist $A_k[i,j]$ die kürzeste Entfernung von i nach j mit Zwischenknoten aus der Menge $\{1, \dots, k\}$.

Um die Zwischenknoten eines kürzesten Weges von i nach j festzuhalten, kann eine Matrix P verwendet werden, wobei $P[i,j]$ den Knoten k enthält, wodurch der kleinste Wert $A[i,j]$ gefunden wurde. Falls $P[i,j]=0$ ist, dann ist die kürzeste Entfernung die direkte Verbindung.



Prozedur floyd

Beispiel 4.20: Veranschaulichung des Floyd-Algorithmus



$A_0[i,j]$	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_1[i,j]$	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_2[i,j]$	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_3[i,j]$	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

Die Matrix P für das Beispiel ergibt sich folgendermaßen:

$P[i,j]$	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

Aus dieser Matrix können dann die Zwischenknoten eines kürzesten Weges von i nach j bestimmt werden.

Zeitkomplexität des Floyd-Algorithmus:

Die Laufzeit des Floyd-Algorithmus ist offensichtlich:

$$T_{\text{floyd}}^{\text{av}}(|V|) = T_{\text{floyd}}^{\text{w}}(|V|) = T_{\text{floyd}}^{\text{b}}(|V|) \in O(|V|^3).$$

Er besitzt damit die gleiche Zeitkomplexität wie die n-fache Lösung mit dem Dijkstra-Algorithmus. Für praktische Probleme wird der Floyd-Algorithmus aufgrund seiner einfachen Struktur jedoch wesentlich schneller sein als die Problemlösung bei Nutzung des Dijkstra-Algorithmus.

4.4.4 Transitiver Abschluß

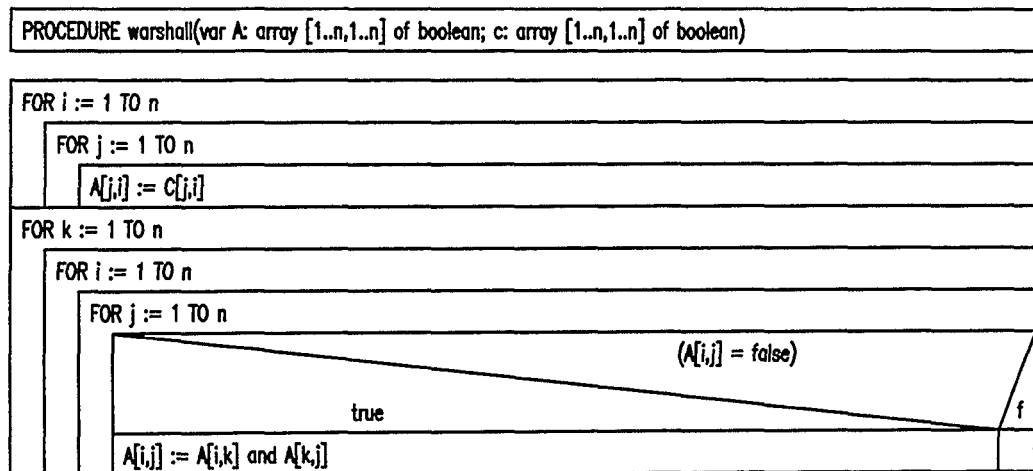
In speziellen Fällen ist man lediglich daran interessiert, ob ein Weg einer beliebigen Länge von Knoten i zu Knoten j existiert. Der Floyd-Algorithmus kann dann so modifiziert werden, daß er dieses Problem löst; der entsprechende Algorithmus heißt *Warshall-Algorithmus*.

Angenommen, die Kostenmatrix C sei gerade die Adjazenzmatrix des Graphen G , d.h. $C[i,j] = 1$, falls eine Kante von Knoten i nach Knoten j existiert, sonst 0. Wir wollen die Matrix A so berechnen, daß $A[i,j] = 1$ genau dann gilt, wenn ein nicht trivialer Weg von i nach j existiert, d.h. der Weg hat eine Länge ≥ 1 . A wird *transitiver Abschluß der Adjazenzmatrix* genannt.

Die Formel, die dem Algorithmus zugrunde liegt lautet:

$$A_k[i,j] = A_{k-1}[i,j] \text{ or } (A_{k-1}[i,k] \text{ and } A_{k-1}[k,j])$$

Daraus ergibt sich der folgende Algorithmus:



Prozedur warshall

Beispiel 4.21: Veranschaulichung des Warshall-Algorithmus

Zu der Adjazenzmatrix A des gerichteten Graphen aus Beispiel 4.20 erhält man den transitiven Abschluß A^* wie folgt:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$A^* = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Die Zeitkomplexität ist gegenüber dem Floyd-Algorithmus unverändert, lediglich die Formel ist modifiziert:

$$T_{\text{warshall}}^{\text{av}}(|V|) = T_{\text{warshall}}^{\text{w}}(|V|) = T_{\text{warshall}}^{\text{b}}(|V|) \in O(|V|^3).$$

Zusammenfassung für die Algorithmen aus der Graphentheorie:

- Die Einschränkung auf gerichtete Graphen ist nicht substantiell. Entsprechende Lösungen können auch für ungerichtete Graphen angegeben werden.
- Teile dieser Algorithmen finden sich als 'Skelett' in vielen interessanten Lösungen zu Problemen aus dem wirklichen Leben (z.B. Netzplantechnik, Compiler (Vektorisierung), das Problem des Handlungsreisenden, aber auch Produktionssteuerung, etc.).
- Darüberhinaus gibt es eine beliebige Anzahl weiterer Probleme und interessanter Algorithmen (z.B. Traveling Salesman Problem), die hier leider nicht weiter behandelt werden können.

Anhang A. Die Turingmaschine

Bei der Turingmaschine handelt es sich um ein leistungsfähiges mathematisches Modell eines Rechners, die trotz der Einfachheit die Fähigkeiten eines General Purpose-Rechners umfaßt. Die Turingmaschine [5], 1936 von A.M. Turing vorgeschlagen, ist das erste mathematische Modell einer idealisierten Rechenmaschine. Das Basismodell einer Turingmaschine (siehe Abb. 40) besteht aus

- einer endlichen Kontrolle,
- einem Eingabeband, unterteilt in Felder und
- einem Lese-Schreib-Kopf.

Das Eingabeband ist nach rechts unendlich lang. Jedes Feld des Eingabebandes kann genau ein Zeichen des Bandalphabetes enthalten.

Die folgenden Begriffe sind für das Verständnis unerlässlich:

Zeichen - z.B. Buchstabe, Ziffer

Alphabet - nicht leere endliche Menge von Zeichen

Wort - endliche Folge von Zeichen aus einem Alphabet
z.B. abcb
Spezialfall: leeres Wort ε

Sprache - Menge von Worten aus Zeichen eines Alphabetes Σ
z.B. $\Sigma = \{a,b\}$,
 $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
(Σ^* bezeichnet die Menge aller Worte über dem Alphabet Σ)

Funktionsbeschreibung der Turingmaschine:

Die n linken Felder ($n \geq 0$) enthalten die Eingabe, d.h. ein Wort aus Zeichen des Eingabealphabets. Die restlichen unendlich vielen Felder enthalten jeweils das Blank, das ein spezielles Zeichen des Bandalphabetes und kein Eingabezeichen ist.

Die Turingmaschine verarbeitet zeichenweise die auf dem Band gespeicherte Information. Ein Übergangsschritt der Turingmaschine sieht wie folgt aus:

In Abhängigkeit von dem durch den Lese-Schreib-Kopf gelesenen Zeichen und von dem Zustand der endlichen Kontrolle werden folgende Operationen ausgeführt:

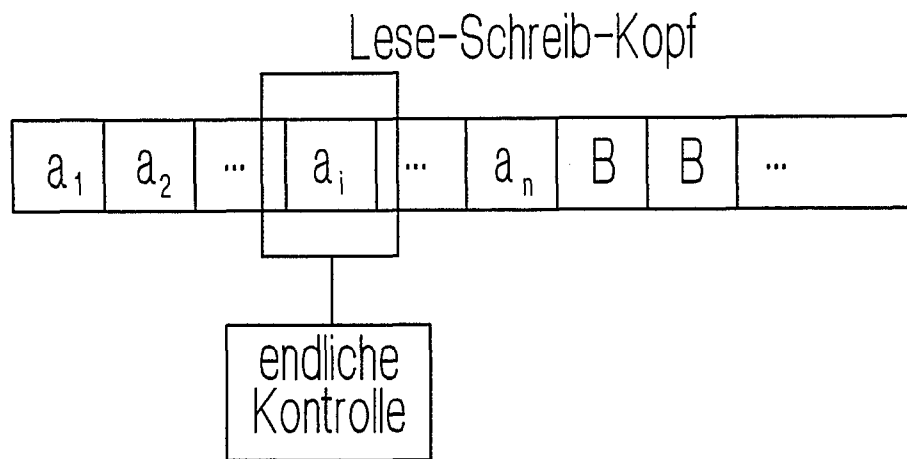


Abb. 40. Turingmaschine

1. Ein neuer Zustand wird angenommen.
2. Ein neues Zeichen wird auf das Arbeitsfeld, d.h. auf das Feld, wo der Lese-Schreib-Kopf steht, geschrieben. Das Zeichen, das auf das Arbeitsfeld geschrieben wird, kann u. U. das gleiche Zeichen sein, das vorher dort stand.
3. Der Lese-Schreib-Kopf wird ein Feld nach rechts oder links bewegt.

Definition B.1: Turingmaschine

Eine Turingmaschine M wird durch die Angabe von sieben Komponenten $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ beschrieben, wobei die einzelnen Komponenten wie folgt definiert sind:

Q *Zustandsmenge*

Γ *Bandalphabet*

B *Blank*

$\Sigma \subseteq \Gamma \setminus \{B\}$ *Eingabealphabet*

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ *Übergangsfunktion*
 (L bzw. R bedeutet einen Übergang nach links
 bzw. nach rechts)

$q_0 \in Q$	<i>Anfangszustand</i>
$F \subseteq Q$	<i>Menge der Endzustände</i>

Definition B.2: Konfiguration einer Turingmaschine

$\alpha_1 q \alpha_2$ heißt Konfiguration einer Turingmaschine, wobei q der momentane Zustand der endlichen Kontrolle ist und $\alpha_1 \alpha_2 \in \Gamma^*$.

Definition B.3: Übergangsschritt einer Turingmaschine

Sei $X_1 X_2 \dots X_{i-1} q X_i \dots X_n$ eine Konfiguration ($i > 1$).

- Falls $\delta(q, X_i) = (p, Y, L)$, dann gilt:

$$X_1 X_2 \dots X_{i-1} q X_i \dots X_n \vdash X_1 X_2 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_n$$

- Falls $\delta(q, X_i) = (p, Y, R)$, dann gilt:

$$X_1 X_2 \dots X_{i-1} q X_i \dots X_n \vdash X_1 X_2 \dots X_{i-1} Y p X_{i+1} \dots X_n$$

Bemerkung:

Wenn zwei Konfigurationen durch $\vdash$ in Verbindung stehen, so sagt man, die zweite Konfiguration entsteht aus der ersten durch einen Übergangsschritt. Wenn der Übergang durch eine endliche Anzahl von Schritten (einschl. 0 Schritten) erfolgt, wird $\vdash^*$ verwendet.

Ausgehend von einer Anfangskonfiguration $(q_0, w) \in Q \times \Gamma^*$ werden mittels δ die Zeichen des Eingabewortes w schrittweise verarbeitet, bis die Turingmaschine mit einer Endkonfiguration $(q_e, w') \in F \times \Gamma^*$ anhält. Auf diese Weise wird zu einem vorgegebenen Argument durch eine Turingmaschine ein Funktionswert $f(w)$ definiert. Interessiert das Ergebnis w' einer Berechnung nicht, sondern nur, ob ein ausgezeichneter Endzustand aus F erreicht wird, so spricht man davon, daß das Wort w akzeptiert wird.

Definition B.4: Akzeptierte Sprache

Die Sprache $L(M)$, die durch eine Turingmaschine M akzeptiert wird, ist die Menge von Worten aus Σ^* , bei deren Eingabe M einen Endzustand erreicht.

Formal: $L(M) = \{w \mid w \in \Sigma^*, q_0 w \vdash^* \alpha_1 p \alpha_2 \text{ für } p \in F, \alpha_1 \alpha_2 \in \Gamma^*\}$.

Beispiel B.1: Turingmaschine, die $L = \{0^n 1^n \mid n \geq 1\}$ akzeptiert:

Sei $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, X, Y, B\}$, $F = \{q_4\}$ und

δ durch die folgende Tabelle definiert.

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

Eine Berechnung von M für $w = 0011$ sieht dann wie folgt aus:

$$\begin{array}{lclclcl}
 q_0 0011 & \vdash & Xq_1 011 & \vdash & X0q_1 11 & \vdash & Xq_2 0Y1 & \vdash \\
 q_2 X0Y1 & \vdash & Xq_0 0Y1 & \vdash & XXq_1 Y1 & \vdash & XXYq_1 1 & \vdash \\
 XXq_2 YY & \vdash & Xq_2 XYY & \vdash & XXq_0 YY & \vdash & XXYq_3 Y & \vdash \\
 XXYq_3 & \vdash & XXYq_4 & & & & &
 \end{array}$$

Bei einer Turingmaschine entspricht ein Rechenschritt einem Übergangsschritt, d.h. einer Anwendung der Übergangsfunktion δ . Der Speicherplatzbedarf wird nach der Zahl der benötigten Speicherzellen festgelegt. Eine Speicherzelle entspricht bei einer Turingmaschine einer Zelle des Bandes. Da die Eingabe selbst schon auf dem Turingband steht, bezieht man in diesem Berechnungsmodell nur die zusätzlich zur Eingabe benötigten Speicherzellen in die Platzbedarfsrechnung ein.

Anhang B. Literatur

- [1] Aho, A. V., Hopcroft, J. E. and Ullman, J. D.
The Design and Analysis of Computer Algorithms
Addison-Wesley Publishing Company, 1974
- [2] Aho, A. V., Hopcroft, J. E. and Ullman, J. D.
Data Structures and Algorithms
Addison-Wesley Publishing Company, 1987
- [3] Güting, R.H.
Datenstrukturen
Manuskript zum Kurs der Fernuniversität Hagen, 1990
- [4] Helman, P. and Veroff, R.
Intermediate Problem Solving and Data Structures: Walls and Mirrors
The Benjamin/Cummings Publishing Company, Inc., 1986
- [5] Hopcroft, J.E. and Ullman, J.D.
Introduction to Automata Theory, Languages, and Computation
Addison-Wesley Publishing Company, 1979
- [6] Horowitz, E. and Sahni, S.
Algorithmen
Springer-Verlag, 1981
- [7] Knuth, D. E.
Sorting and Searching
The Art of Computer Programming Vol.3
Addison-Wesley Publishing Company, 1974
- [8] Lange, O. und Stegemann, G.
Datenstrukturen und Speichertechniken
Friedr. Vieweg & Sohn, 1985
- [9] Manber, U.
Introduction to Algorithms
Addison-Wesley Publishing Company, 1989
- [10] Stubbs, D. F. and Webre, N. W.
Data Structures with Abstract Data Types and Pascal
Brooks/Cole Publishing Company, 1984
- [11] Sedgewick, R.
Algorithms
Addison-Wesley Publishing Company, 1988
- [12] Wirth, N.
Algorithmen und Datenstrukturen
Teubner Studienbücher Informatik, 1979

A

- Abstrakter Datentyp 12
- Abstraktionsebenen 12
- Adjazenzmatrix 78
- Akzeptierte Sprache einer Turingmaschine 176
- Algorithmen
 - Ausgewählte Algorithmen 119
 - Effizienzbeurteilung 4
 - Spezifikation durch Pre- und Post-Conditions 15
 - Zeit- und Speicherkomplexität 5
- Arrays
 - siehe Felder (arrays)
- Atomare Objekte 10
- Aufwandsbetrachtungen 7
- Aufzählungstyp 23
- Ausgewählte Algorithmen 119-172
 - Graphenalgorithmen 161
 - Hash-Verfahren 146
 - Sortierverfahren 123
 - Suchverfahren 119
- average-case
 - siehe Aufwandsbetrachtungen

B

- B-Bäume 108-112
- Bäume 80-112
 - Binärbäume 81
 - Binäre Suchbäume 95, 122
 - Haufen (heaps) 112
 - Höhenbalancierte Bäume 107
 - B-Bäume 108
- Begriffe bei Bäumen
 - Blatt 81
 - Bruder eines Knoten 81

- Ebene eines Knoten (level) 81
- Höhe eines Baumes 81
- Innerer Knoten 81
- Nachfolger in einem Baum 81
- Sohn eines Knoten 81
- Teilbaum 81
- Vater eines Knoten 81
- Vorgänger in einem Baum 81
- Berechenbare Funktion 5
- best-case
 - siehe Aufwandsbetrachtungen
- BFS
 - siehe Breadth-First-Search-Algorithmus
- Binärbäume 81-95
 - Geordneter Binärbaum 82
 - Minimaler Binärbaum 106
 - Voller Binärbaum 81
 - Vollständiger Binärbaum 106
- Binäre Suchbäume 95-107
- Binäres Einfügen 129
- Binäres Suchen 120
- Binary Search
 - siehe Binäres Suchen
- Breadth-First-Search-Algorithmus 163
- Bubblesort
 - siehe Sortierverfahren

D

- Datenstrukturen
 - Bäume 80
 - Einfache Datenstrukturen 18
 - Einführung der Grundbegriffe 3
 - Graphen 74
 - Haufen (heaps) 112
 - Heterogene Datenstruktur 18
 - Homogene Datenstruktur 18

- Lineare Datenstrukturen 39
- Datentypen
 - abstrakt 12
 - atomar 10
 - physikalisch 12
 - strukturiert 11
 - virtuell 12
- Depth-First-Search-Algorithmus 161
- DFS
 - siehe Depth-First-Search-Algorithmus
- Dijkstra-Algorithmus 166
- Dispose 26
- Divisions-Hash 150
- Doppelt verkettete Liste 56
- Durchlaufen von gerichteten Graphen
 - Breadth-First-Search 163
 - Depth-First-Search 161
 - Kürzeste Wege 165
- Durchlaufstrategien in Bäumen
 - Inorder 82, 86, 93
 - Postorder 82, 86, 93
 - Preorder 82, 86, 93
- Dynamische Datenstrukturen 25, 34
 - Pascal-Operation Dispose 26
 - Pascal-Operation New 26

E

- Effizienz von Algorithmen 4
- Einfach verkettete Liste 39-55
 - Abstrakter Datentyp 40
 - Realisierung durch drei Felder 51
 - Realisierung durch ein Feld 53
 - Realisierung über Pointer 45
- Einfache Datenstrukturen 18-39
 - Mengen (sets) 28
 - Sätze (records) 22
 - Sequenzen (sequences) 31
 - Zeiger (pointer) 25
- Einfügen

- in binären Suchbaum 96, 99
- in eine lineare Liste 41, 45
- in einen B-Baum 109
- in einen Heap 115
- Sortieren durch 128
- Externe Sortiervverfahren 123

F

- Felder (arrays) 19
- Fibonacci-Hash 153
 - Fibonacci-Folge 153
 - Swierczkowsky-Verteilung 154
 - Zugriffszeiten 158
- FIFO (First-In-First-Out) 69
- Files
 - siehe Pascal-Dateien
- Floyd-Algorithmus 168

G

- Gerichtete Graphen 74
- Get 35
- Gewichtete Graphen 78
- Graphen 74-80
 - siehe auch Graphentheoretische Begriffe
- Graphenalgorithmen 161-172
 - Breadth-First-Search-Algorithmus 163
 - Depth-First-Search-Algorithmus 161
 - Dijkstra-Algorithmus 166
 - Floyd-Algorithmus 168
 - Warshall-Algorithmus 170
- Graphentheoretische Begriffe
 - Adjazenz 78
 - Einfacher Weg 78
 - Gerichteter Graph 74
 - Gewichteter Graph 78
 - Kantenmenge E 74

Knotenmenge V 74
 Kreis 78
 Teilgraph 78
 Ungerichteter Graph 75
 Weg 77
 Greedy-Algorithmen 165
 Groß-O-Notation 6

H

Hash-Funktion 148
 Hash-Verfahren 146-160
 Divisions-Hash 150
 Fibonacci-Hash 153, 158
 Füllungsgrad 156
 Hash in Teillisten 159
 Ideales Hashing 156
 Kollisionen beim Hashing 148, 157
 Lineares Sondieren (linear probing) 150
 Offene Hash-Verfahren 148, 158
 Quadratisches Sondieren (quadratic probing) 151
 Schrittzahl 156
 Vollständiger Hash 151
 Haufen (heaps) 112-118, 132
 Heap
 siehe Haufen (heaps)
 Heapsort 132
 Höhenbalancierte Bäume 107-112, 146
 B-Bäume 108

I

Inorder
 siehe Durchlaufstrategien in Bäumen
 Insertion Sort
 siehe Sortiervverfahren
 Interne Sortiervverfahren 123

K

Knoten
 Bruder eines 81
 Ebene eines 81
 Innerer 81
 Sohn eines 81
 Vater eines 81
 Kollisionsstrategien
 Lineares Sondieren (linear probing) 150
 Quadratisches Sondieren (quadratic probing) 151
 Komplexität eines Algorithmus
 Groß-O-Notation 6
 Königsberger Brückenproblem 76
 Kürzeste Wege in Graphen
 Dijkstra-Algorithmus 166
 Floyd-Algorithmus 168

L

LIFO (Last-In-First-Out) 60
 Linear probing
 siehe Lineares Sondieren
 Lineare Datenstrukturen 39-74
 Einfach verkettete Liste
 Abstrakter Datentyp 40
 Realisierung durch drei Felder 51
 Realisierung durch ein Feld 53
 Realisierung über Pointer 45
 Mehrfach verkettete Liste 55
 Stapel (stack) 60
 Warteschlange (queue) 69
 Lineares Sondieren 150
 Lineares Suchen 120
 Linked list
 siehe Einfach verkettete Liste

Löschen
 in einem B-Baum 110
 in einem binären Suchbaum 97
 in einer linearen Liste 41, 46

M

Matrix
 Adjazenz 78
Median-of-three-Methode 140
Mehrfach verkettete Liste 55
Mengen (sets) 28
Mergesort 140

N

New 26
Nil pointer 25

O

O-Notation 6
Offene Hash-Verfahren 148
 Divisions-Hash 150
 Fibonacci-Hash 153, 158
 Zusammenfassung der
 Eigenschaften 158

P

Pascal Datentypen
 array 20
 file 35
 pointer 26
 record 22
 set 28
 text 37
Pascal Text-Dateien 37

Pascal-Dateien 34
Physikalischer Datentyp 12
Pointer
 siehe Zeiger (pointer)
Postorder
 siehe Durchlaufstrategien in Bäumen
Pre- und Post-Conditions 15
Preorder
 siehe Durchlaufstrategien in Bäumen
Put 36

Q

Quadratic probing
 siehe Quadratisches Sondieren
Quadratisches Sondieren 151
Queues
 siehe Warteschlange (queue)
Quicksort 134
 Median-of-three-Methode 140

R

Records
 siehe Sätze (records)
Rekursion 66
 Rekursive Prozedur 66
Reset 35
Rewrite 35

S

Sätze (records) 22
Schlange (queue) 69
Schlüssel 18
Schlüsselmenge 18
 Felder (arrays) 19
Selection Sort

- siehe Sortiervverfahren
- Sequences
 - siehe Sequenzen (sequences)
- Sequential Search
 - siehe Lineares Suchen
- Sequenzen (sequences) 31-39
- Sets
 - siehe Mengen (sets)
- Sinking Sort
 - siehe Sortiervverfahren
- Sortierpermutation 124
- Sortierte Liste 120
- Sortiervverfahren 123-146
 - Externe 123
 - Heapsort 132
 - Interne 123
 - Quicksort 134
 - Sortieren durch Auswahl (selection sort) 125
 - Sortieren durch Einfügen (insertion sort) 128
 - Sortieren durch Mischen (Mergesort) 140
 - Sortieren durch Vertauschen (Bubblesort, sinking sort) 130
 - Stabile 125
- Speichermedien
 - Externe Speicher 14, 140
 - Hauptspeicher 13
 - Magnetbänder 14, 140
 - Magnetplatte 14
 - Trommel 14
- Speicherung von Graphen
 - Adjazenzliste 79, 167
 - Adjazenzmatrix 78, 167
- Stabile Sortiervverfahren 125
- Stack
 - siehe Stapel (stack)
- Stapel 60
- Stapel (stack) 60-69
- stdelement 31, 43

- Struktureigenschaften
 - Baumstrukturen 17
 - Graphstrukturen 17
 - Hierarchische Strukturen 17
 - Lineare Strukturen 17
 - Menge 17
 - Netzwerkstrukturen 17
- Suchbaum 95, 122
- Suchverfahren 119-123
 - Binäres Suchen 120
 - Breadth-First-Search 163
 - Depth-First-Search 161
 - Lineares Suchen 120

T

- Transitiver Abschluß
 - Warshall-Algorithmus 170
- Turingmaschine 173
 - Konfiguration 175
 - Übergangsschritt 175

U

- Ungerichtete Graphen 75

V

- Verkettete Listen 39, 55, 56
- Virtueller Datentyp 12

W

- Warshall-Algorithmus 170
- Warteschlange (queue) 69-74
- Wege (in Graphen) 78
- With-Anweisung in Pascal 23

worst-case

siehe Aufwandsbetrachtungen

Z

Zeiger (pointer) 25, 26, 35, 39

Zeitkomplexität eines Algorithmus 5

Jül-2527
September 1991

ISSN 0366-0885