

Algorithmen und Strukturen in C

Günter Egerer Renate Knecht Wolfgang E. Nagel

Berichte des Forschungszentrums Jülich ; 3587
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3587

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
52425 Jülich · Bundesrepublik Deutschland
☎ 02461/61-5220 · Telefax: 02461/61-6103 · e-mail: zb-publikation@fz-juelich.de

Algorithmen und Strukturen in C

Dieses Manuskript dient als Basis für eine Vorlesung über Algorithmen und Strukturen im Studiengang TechnoMathematik an der Fachhochschule Aachen, Abteilung Jülich und im Rahmen der Ausbildung zum Mathematisch-technischen Assistenten im Forschungszentrum Jülich.

Nach einer Einführung in die Grundbegriffe Algorithmus, Komplexität und Effizienz eines Algorithmus sowie O-Notation werden Datenstrukturen formal auf verschiedenen Abstraktionsniveaus definiert. Anschließend werden an Hand von 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 des Manuskripts werden mit den beschriebenen Datenstrukturen und den zugehörigen Basisoperationen Programmkerne in C zur Verfügung gestellt, die dann für komplexere Aufgabenstellungen zu größeren Problemlösungen 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 sind und 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.

Algorithms and Data Structures in C

This manuscript is the basis of a lecture on algorithms and data structures. It starts with the definition of the terms algorithm, complexity, efficiency, and the asymptotic behavior of an algorithm. Data structures are then defined on various abstraction levels and it is shown how structures like arrays, sets, lists, graphs, trees, etc. can be mapped on a computer in order to achieve efficient operations on these structures as, e.g. accessing tree elements or sorting list elements. The resulting data structures describe the storage method as well as the basic operations on these data. Program kernels based on the described data structures are presented in C which can be combined for the solution of more complex problems. Complexity investigations are performed for the operations presented which allow a comparison of different algorithms with respect to time and space requirements.

In the second part algorithms for various tasks, e.g. searching and sorting of list elements, efficient storage methods, and graph algorithms are described. These can be used as building blocks for more complex problems. Examples from different areas are presented and analyzed with respect to their asymptotic behavior.

Inhaltsverzeichnis

1	Einleitung	3
2	Einführung der Grundbegriffe	5
2.1	Algorithmen als Lösungsvorschrift	5
2.1.1	Effizienz von Algorithmen	6
2.1.2	Komplexität von Algorithmen als Maß für die Effizienz	6
2.2	Datenstrukturen als Basis für effiziente Algorithmen	10
2.2.1	Datentypen und Datenstrukturen	10
2.2.2	Abstraktionsniveaus bei Datentypen	11
2.2.3	Physikalische Speicherung von Daten	13
2.2.4	Spezifikation von Basisoperationen auf Datenstrukturen	13
3	Datenstrukturen	15
3.1	Einfache Datenstrukturen	16
3.1.1	Felder (arrays)	17
3.1.2	Sätze (records)	19
3.1.3	Zeiger (pointer)	21
3.1.4	Mengen (sets)	23
3.1.5	Sequenzen (sequences)	25
3.2	Lineare Datenstrukturen	30
3.2.1	Einfach verkettete Listen (linked lists)	30
3.2.2	Mehrfach verkettete Listen	43
3.2.3	Stapel (stacks)	48
3.2.4	Warteschlangen (queues)	57
3.3	Graphen	62
3.4	Bäume	66
3.4.1	Binärbäume	66
3.4.2	Binäre Suchbäume als Anwendung von Binärbäumen	79
3.4.3	B-Bäume	90
3.5	Haufen (heaps)	95
4	Ausgewählte Algorithmen	101
4.1	Suchverfahren	101
4.1.1	Lineares Suchen (Sequential Search)	101
4.1.2	Binäres Suchen (Binary Search)	102
4.2	Sortierverfahren	105
4.2.1	Allgemeine Voraussetzungen für Sortierverfahren	105
4.2.2	Sortieren durch Auswahl (Selection Sort)	106
4.2.3	Sortieren durch Einfügen (Insertion Sort)	107
4.2.4	Sortieren durch Vertauschen (Bubble Sort, Sinking Sort)	109
4.2.5	Heapsort	111

4.2.6	Quicksort	113
4.2.7	Sortieren durch Mischen (Mergesort)	119
4.3	Hash-Verfahren	126
4.3.1	Offene Hash-Verfahren	127
4.3.2	Hash in Teillisten	135
4.4	Algorithmen auf Graphen	137
4.4.1	Das Durchlaufen von gerichteten Graphen	137
4.4.2	Kürzeste Wege von einer Quelle	141
4.4.3	Kürzeste Wege zwischen allen Knotenpaaren	143
4.4.4	Transitiver Abschluß	145
A	Die Turingmaschine	147
B	Beispiel: Realisierung von Mengen (sets)	151
C	Literatur	161
	Index	162

Abbildungsverzeichnis

2.1	Zeitverhalten der einzelnen Algorithmen	9
2.2	Zerlegung der Klasse der Datentypen	11
2.3	Abstraktionsniveaus bei Datentypen	11
2.4	Spezifikation eines strukturierten abstrakten Datentyps	12
3.1	Elementare Strukturrelationen	15
3.2	Beispiel für Postleitzahlen	16
3.3	Strukturbaum und Speicherdarstellung der Datenstruktur Mitarbeiter	21
3.4	Beispiel für die Darstellung einer einfach verketteten Liste	30
3.5	Liste vor und nach der Ausführung von insert_after	33
3.6	Liste mit drei Feldern	38
3.7	Beispiel für die Darstellung einer doppelt verketteten Liste	43
3.8	Beispielhafte Darstellung eines Stacks	48
3.9	Darstellung der Operation push auf dem Datentyp Stack	51
3.10	Visualisierung der Feldimplementation eines Stacks	51
3.11	Mittelbarer rekursiver Aufruf	53
3.12	Queue (übliche Darstellung)	58
3.13	Queue, realisiert mit einem Feld bei unterschiedlichem Füllungsgrad	59
3.14	Visualisierung einer zirkulären Queue	60
3.15	Beispiel für einen gerichteten Graph	62
3.16	Übergang von symmetrisch gerichtetem zu einem ungerichteten Graph	62
3.17	Königsberger Brückenproblem	63
3.18	Graphentheoretische Problemformulierung	63
3.19	Auswahl von Wegen in einem Graphen G	64
3.20	Durchlaufstrategien eines Baumes (Numerierung gibt die Reihenfolge der Bearbeitung an)	67
3.21	Speicherung des Infix-Ausdrucks $(a + b) * c + 7$	68
3.22	Speicherung des Infix-Ausdrucks $(a + b) * (c + 7)$	68
3.23	Beispielbäume	68
3.24	Baumdarstellung mit Pointer	71
3.25	Baumdarstellung mit statischen Feldern	71
3.26	Beispiele für binäre Suchbäume	79
3.27	Unterschiedliche Suchbäume mit gleichem Informationsgehalt	89
4.1	Binärer Suchbaum und Suchpfad zu Element 27	104
4.2	Speicherreferenzdiagramm für den Quicksort-Algorithmus	116
4.3	Mischen von sortierten Listen A und B zu einer sortierten Liste C	119
4.4	address calculator	126
4.5	Darstellung der Swierczkowsky-Verteilung für $N=7$	131
4.6	Qualitativer Verlauf von $S(n,k)$ für ein festes n	133
4.7	Kollisionen beim Fibonacci-Hash (mit linearem bzw. quadratischen Sondieren)	134

A.1 Turingmaschine	147
------------------------------	-----

Vorwort

Der Kurs '*Algorithmen und Strukturen*' ist der zweite von derzeit 5 Informatikkursen im Rahmen der Ausbildung von Mathematisch-Technischen Assistentinnen und Assistenten (MTAs) im Forschungszentrum Jülich sowie des Studiums der TechnoMathematik an der FH Aachen, Abteilung Jülich. Eines der Hauptmerkmale dieses Berufsbildes wie auch des FH-Studiums ist die Praxisbezogenheit des Unterrichts. So sollen die Auszubildenden 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 sogenannten *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 ursprünglich in den Standard-Programmiersprachen Pascal und Fortran77 vorgenommen wurden. Der Wechsel zur Programmiersprache C stellt in dieser Hinsicht auch keine wesentliche Verbesserung dar, da C modulares Programmieren zwar ermöglicht, jedoch auch nicht wirklich unterstützt. 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 Zahl von 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 Graphenalgorithmien 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. Für seinen unermüdlichen Einsatz bei der Erstellung von zusätzlichen grafischen Darstellungen, bei der Umwandlung von Programm-Code in Nassi-Shneiderman-Diagramme, an der auch Frau M. Baetzen beteiligt war, und bei der Beseitigung von Fehlern danken wir Herrn Frings sehr herzlich. Die Nassi-Shneiderman-Diagramme für die vorliegende Version wurden mit Hilfe von *nassi*, einem am Forschungszentrum Jülich entwickelten Programm (siehe <http://www.kfa-juelich.de/zam/nassi>), erzeugt. Der Text liegt als L^AT_EX-Dokument vor.

September 1998

Die Autoren

Kapitel 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 C, 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.

Kapitel 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.

2.1.0.1 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 Zeitverhaltens. 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:

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

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:
 $f(n) \cdot g(n) \in O(s(n) \cdot r(n))$.

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 oder ähnliches). 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.

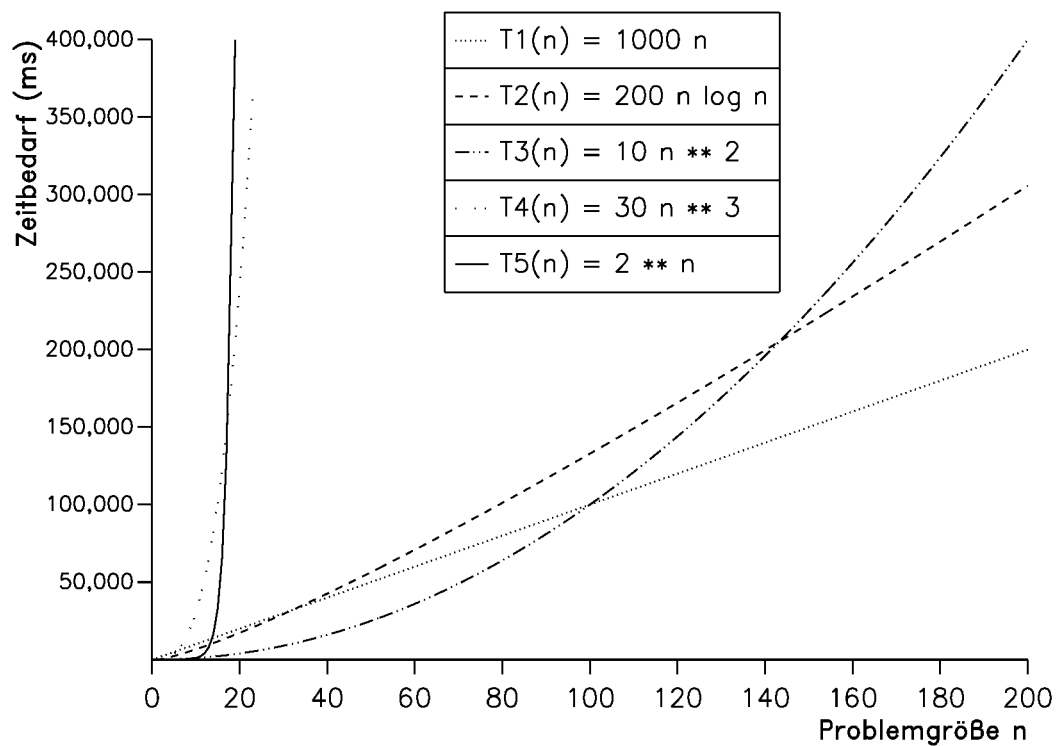
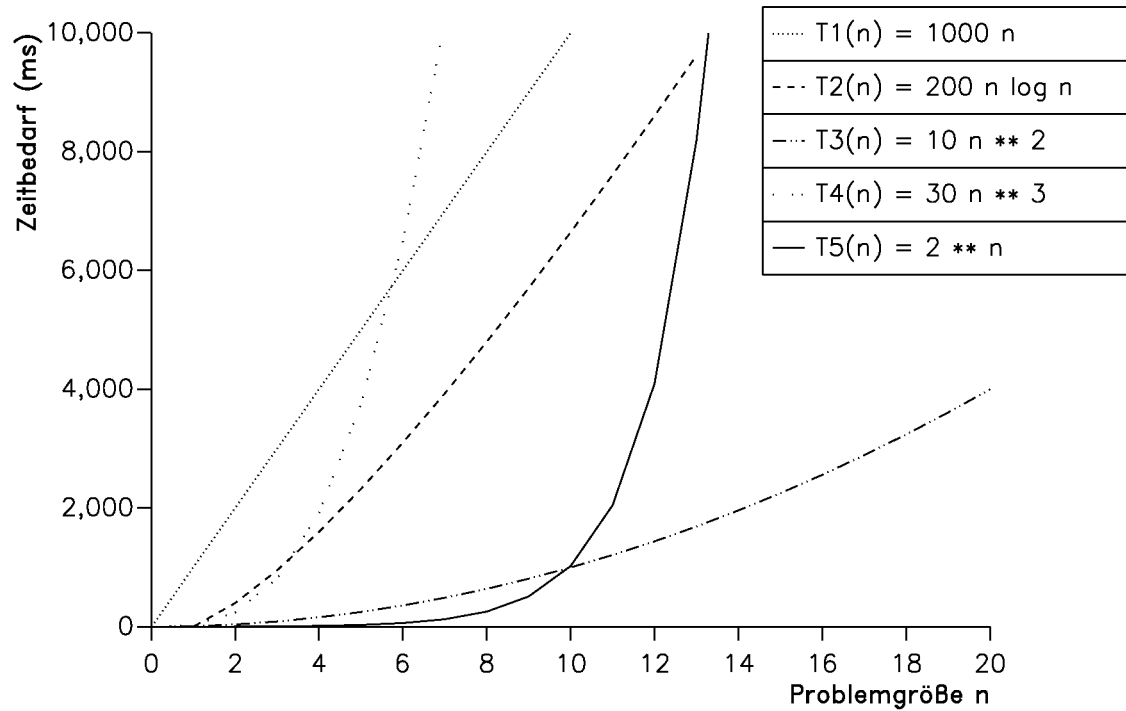


Abbildung 2.1: Zeitverhalten der einzelnen Algorithmen

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 int*

- 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 Fortran 90, nicht in C)

```
REAL, DIMENSION(3) :: a, b, c
...
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*.

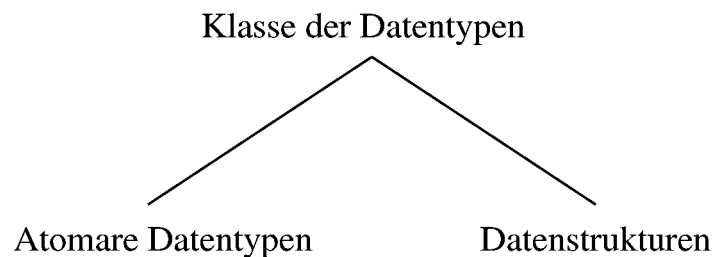


Abbildung 2.2: Zerlegung der Klasse der Datentypen

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:

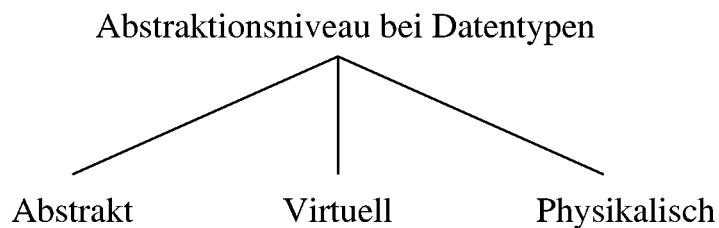


Abbildung 2.3: Abstraktionsniveaus bei Datentypen

- **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. C als höhere Programmiersprache die Funktion einer virtuellen Maschine. Die Implementation der Datenstrukturen durch den Programmierer

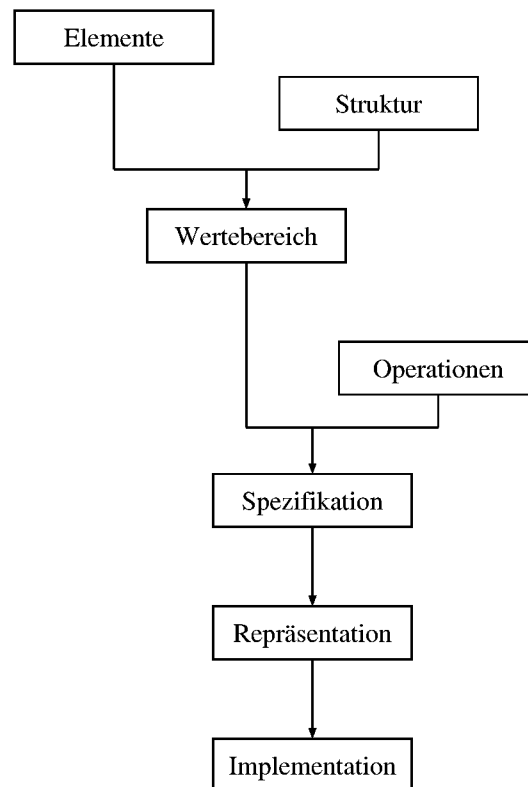


Abbildung 2.4: Spezifikation eines strukturierten abstrakten Datentyps

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

Datentyp	Abstraktionsniveau		
	abstrakt	virtuell	physikalisch
strukturiert	Liste von Temperaturen	C: float []	Mehrere Gleitpunktzahlen hintereinander im Speicher
atomar	Anzahl von Leuten	C: int	Festpunktzahl

Tabelle 2.1: 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. 2.4).

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.

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 oder ähnliche grafische Darstellungen kann man einen Algorithmus auch durch sogenannte 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. *Farbe mix(const Farbe c1, const Farbe c2)*
pre c1 und c2 sind Primärfarben
post *mix* ist die Farbe, die durch Mischen von c1 und c2 entsteht
 - b. *bool primary(const Farbe c)*
post if c ist Primärfarbe
 then *primary* ist true
 else *primary* ist false
 - c. *void form(const Farbe c, Farbe c1, Farbe c2)*
pre c ist keine Primärfarbe
post c1 und c2 sind die beiden Primärfarben, aus denen c besteht.
 - d. *void assign(const Farbe c1, Farbe c2)*
post c2 hat den Wert c1.

Kapitel 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. 3.1), 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.

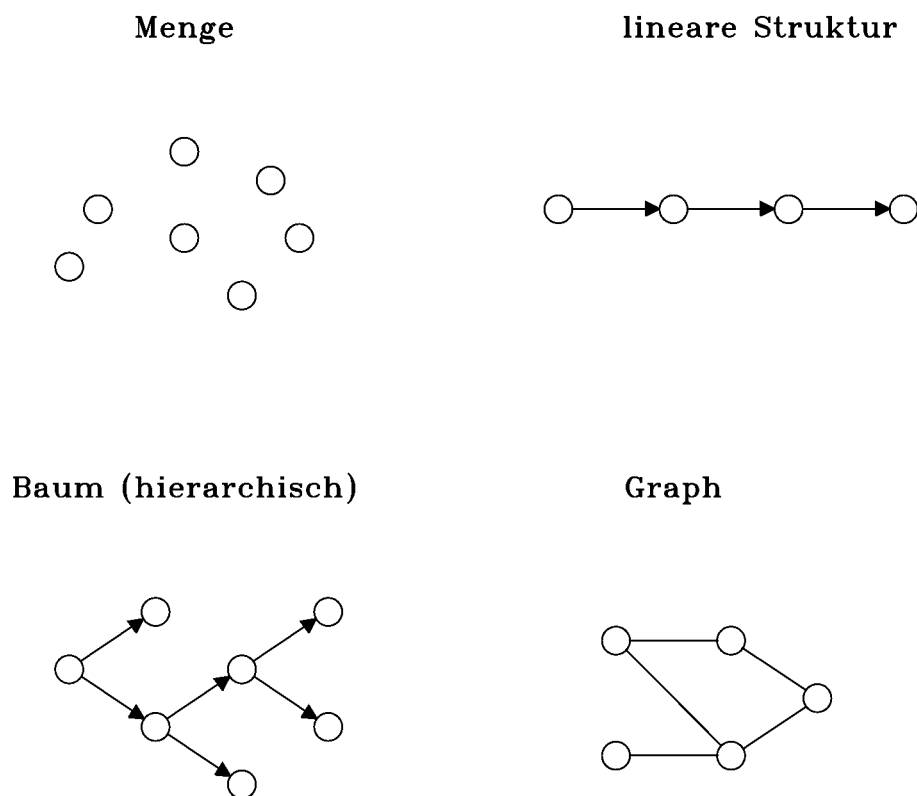


Abbildung 3.1: Elementare Strukturrelationen

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 im allgemeinen 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).

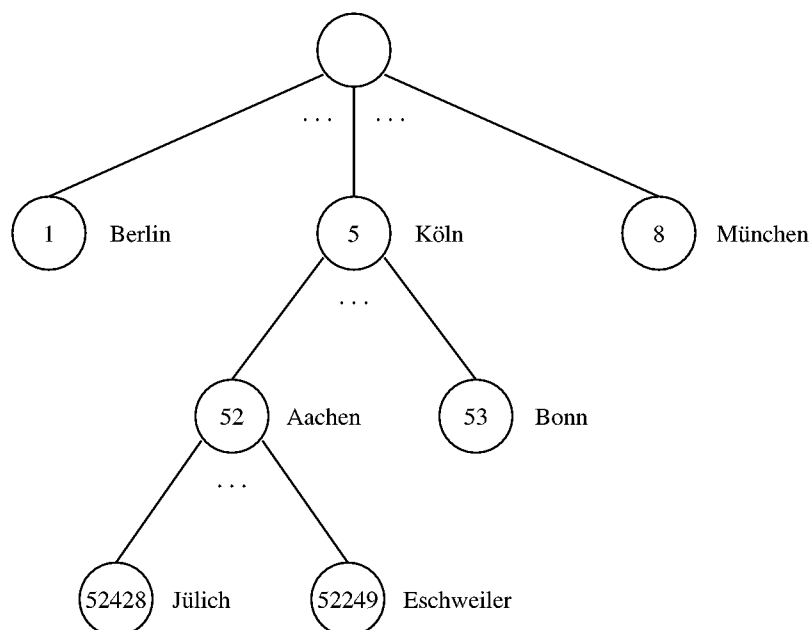


Abbildung 3.2: Beispiel für Postleitzahlen

3.1.1 Felder (arrays)

Bei den Feldern handelt es sich um die gebräuchlichste Datenstruktur, die in den meisten Programmiersprachen (z.B. Fortran, Pascal, C, ...) 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 C : $C = S[i_1] \dots [i_n]$;

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

3.1.1.1 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 (Vektor):

- `int A[5]`; (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[2]` mit dem Wert 5 und der Adresse $\text{addr}(A) + (2 \bullet \text{element_length}(A))$
Das bedeutet: $P: i \rightarrow \text{addr}(A) + (i \bullet \text{element_length}(A))$

2. Zweidimensionales Feld (Matrix):

- `int B[3][2]`; (jedes Element 4 Byte lang)
- Enthalte B die folgenden Werte

B[i][j]		j	
		0	1
i	0	16	7
	1	5	8
	2	10	10

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

a. Spaltenweise Speicherung (Fortran)

...	B[0][0]	B[1][0]	B[2][0]	B[0][1]	B[1][1]	B[2][1]	...
-----	---------	---------	---------	---------	---------	---------	-----

Adreßberechnung bei spaltenweiser Speicherung:

Das zweidimensionale Feld B sei dimensioniert als `B[i1][i2]` (*i₁* Zeilen, *i₂* Spalten).

Die Adresse ergibt sich aus den Feldindizes *i* ($0 \leq i < i_1$) und *j* ($0 \leq j < i_2$) gemäß der folgenden Abbildung P:

$$P: i, j \rightarrow \text{addr}(B) + ((j \bullet i_1 + i) \bullet \text{element_length}(B)) = \text{addr}(B[i][j])$$

z.B.

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

In C ist die untere Grenze für den Bereich der Feldindizes immer 0. Andere Sprachen erlauben dem Programmierer auch andere Werte als untere Grenze zu vereinbaren. Z.B. erhält man in Fortran durch die Vereinbarung

`INTEGER, DIMENSION(l1:u1, l2:u2) :: C`

ein zweidimensionales Feld C mit $u_1 - l_1 + 1$ Zeilen und $u_2 - l_2 + 1$ Spalten. Die angegebenen Grenzen legen die Wertebereiche für die Feldindizes *i* und *j* fest: $l_1 \leq i \leq u_1$ und $l_2 \leq j \leq u_2$. Die zugehörige Abbildung P sieht dann wie folgt aus:

$$P: i, j \rightarrow \text{addr}(C) + (((j - l_2) \bullet (u_1 - l_1 + 1) + i - l_1) \bullet \text{element_length}(C))$$

b. Zeilenweise Speicherung (C, Pascal)

...	B[0][0]	B[0][1]	B[1][0]	B[1][1]	B[2][0]	B[2][1]	...
-----	---------	---------	---------	---------	---------	---------	-----

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 (identifier) 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 C: $V = R.id;$
 - b. *update* (R, V, id)
speichert den Wert der Variablen V auf die Komponente von R , die durch id bezeichnet ist.
in C: $R.id = V;$

3.1.2.1 Besonderheiten:

1. Wenn R_1 und R_2 Records vom gleichen Typ sind, kann durch $R_1 = R_2$ der Wert von R_2 vollständig auf R_1 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.

Beispiel 3.3 C Programm-Skelett mit Record-Typvereinbarungen

```
typedef enum { BMW, Ford, VW }    Hersteller;
typedef enum { Blau, Gruen, Rot }  Farbe;
typedef enum { false, true }      bool;

typedef struct {
    Hersteller Marke;
    Farbe      Lackierung;
} Kfz;
```

```

bool Gleiches_Auto( Kfz Auto1, Kfz Auto2 ) {
    return ( (Auto1.Marke == Auto2.Marke)
            && (Auto1.Lackierung == Auto2.Lackierung) );
}

/* im Programmtext der Funktion main: */
Kfz Auto1, Auto2;
Kfz Bestelltes_Auto;

a)  Bestelltes_Auto.Marke = Ford;
    Bestelltes_Auto.Lackierung = Gruen;

b)  Auto1.Marke = Ford;
    Auto1.Lackierung = Blau;

c)  Auto2 = Auto1;
    Auto2.Lackierung = Rot;

d)  if ( Gleiches_Auto(Bestelltes_Auto, Auto1) )
    {
        /* Auslieferung von Auto1 */
    }
    else
    {
        if ( Gleiches_Auto(Bestelltes_Auto, Auto2) )
        {
            /* Auslieferung von Auto2 */
        }
        else
        {
            /* Produktion von Bestelltes_Auto */
        }
    }
}

```

Beispiel 3.4 C Programm-Skelett für die Datenstruktur 'Mitarbeiter'

```

#define MAXSTRING 10

typedef char string[ MAXSTRING + 1 ];
typedef unsigned char byte;

typedef struct {
    int Personalnr;
    struct {
        string Vorname;
        string Nachname;
        string Geburtsname;
    } Name;
    struct {
        string Strasse;
        short int Hausnr;
        int Plz;
        string Ort;
    } Anschrift;
    byte Kinderanzahl;
} Mitarbeiter;

```

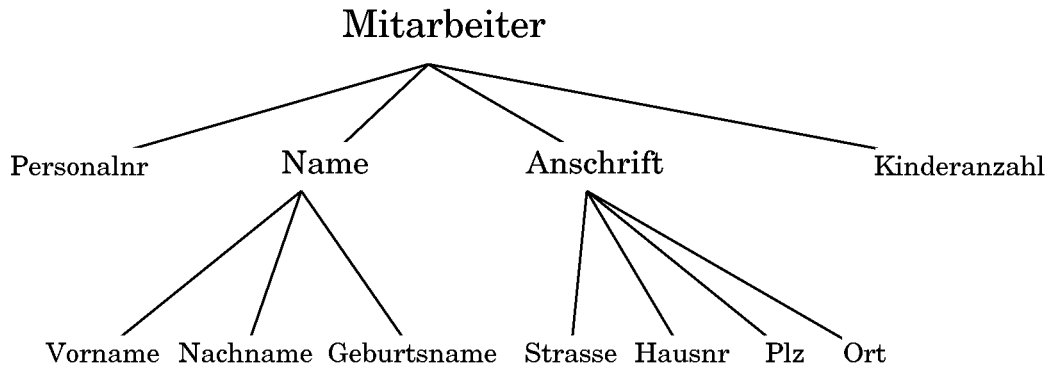
```

... /* im Programmtext: */
Mitarbeiter Einzustellende_Person;

strcpy(Einzustellende_Person.Name.Vorname, "Birgit");
strcpy(Einzustellende_Person.Name.Nachname, "Mueller");

```

a) Strukturbaum



b) Speicherdarstellung

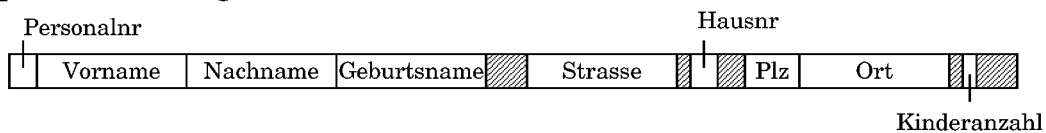


Abbildung 3.3: Strukturbaum und Speicherdarstellung der Datenstruktur Mitarbeiter

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 NULL (undefiniert) zulässig.
2. Struktur

keine
3. Operationen (in der Programmiersprache C)
 - a. Assignment (Zuweisung) '='
 - b. Relational (Vergleich) '==' und '!='
 - c. Dynamic (dynamische Bereitstellung von Speicherplatz)

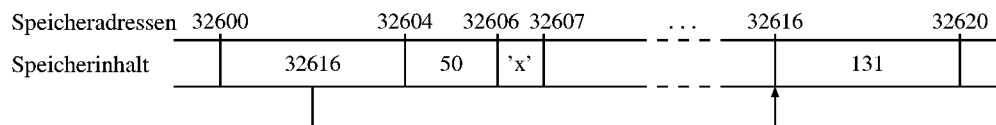
'malloc' (sowie 'calloc' und 'realloc') und 'free'

(C++: 'new', 'delete', 'new []' und 'delete []')
 - d. Dereferenzierung '*' (sowie '->' und '[]')

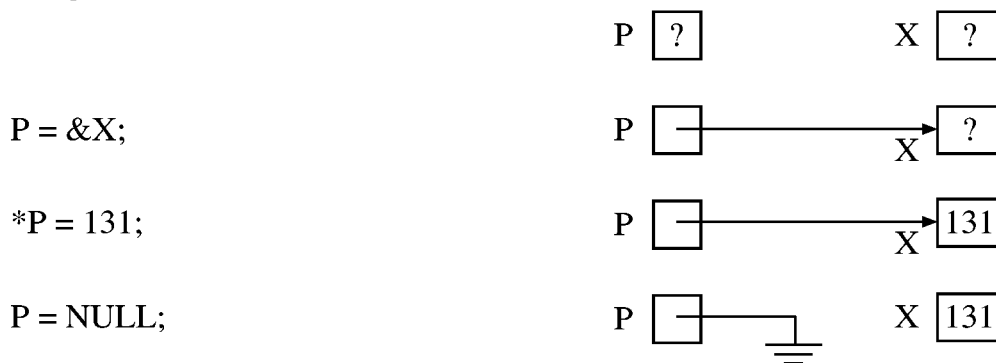
Beispiel 3.5 *Vorstellung der Pointer*

```
typedef int *ptr_t;
ptr_t P, Q;
int X;
```

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 `ptr_t P;` wird eine Speicherstelle angelegt, die in der Lage ist, eine Referenz (Adresse) zu einer anderen Speicherstelle aufzunehmen, in der eine Integerzahl abgelegt werden kann.



Zuweisung:

**3.1.3.1 Dynamische Basisoperationen in C:**

1. **malloc** ist eine Standardfunktion. Der Aufruf `P = malloc(sizeof(int))` stellt eine Speicherstelle (in diesem Beispiel vom Typ Integer) bereit, und gibt eine Referenz auf diese Speicherstelle zurück.
2. **free** ist ebenfalls eine Standardfunktion. Der Aufruf `free(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 wieder (in anderem Kontext) vergeben wird. Nach einer Freigabe von P sollte der Pointer wieder auf NULL gesetzt werden.

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 C-Typ `double` 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 (set s1, const set s2)*

post s1 hat die gleichen Elemente wie s2.

b. *bool in (const set s1, element e)*

post if e Mitglied von s1
 then in ist true
 else in ist false

c. *set intersection (const set s1, const set s2)*

post intersection ist eine Menge, die den Durchschnitt von s1 und s2 enthält.

d. *set union (const set s1, const set s2)*

post union ist eine Menge, die die Vereinigung von s1 und s2 enthält.

e. *set difference (const set s1, const set s2)*

post difference ist eine Menge, die s1 ohne s2 enthält.

f. *bool equal (const set s1, const set s2)*

post if s1 und s2 haben die gleichen Elemente
 then equal ist true
 else equal ist false

g. *bool subset (const set s1, const set s2)*

post if jedes Element von s1 auch in s2
 then subset ist true
 else subset ist false

h. *mkset (set s, const element elem_vec[], int n)*

post s enthält ausschließlich elem_vec[0]... elem_vec[n-1] als Elemente.

i. *set create (element lbound, element ubound, id_type basetype_id)*

post create ist eine leere Menge. Diese kann Elemente aufnehmen, die größer oder gleich lbound, jedoch nicht größer als ubound sind, und ist typkompatibel zu Mengen, die mit der gleichen basetype_id erzeugt wurden.

j. *void delete (set s)*

post s existiert nicht mehr.

Eine vollständige Realisierung des abstrakten Datentyps Set in der Programmiersprache C findet sich als Beispiel im Anhang B.

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

1. in C unter Nutzung (der im Anhang B angegebenen Realisierung) von Sets

```
enum Buchstaben = 0;
```

```
/* lokale Variablen: */
set *alphabet, *temp;
int c;
```

void find_letter(set *einmal, set *mehrmals, set *nie)		
mkset(einmal, "", 0, 1)		
mkset(mehrmals, "", 0, 1)		
temp = create('A', 'Z', Buchstaben)		
alphabet = mkset(create('A', 'Z', Buchstaben), "ABCDEFGHIJKLMNOPQRSTUVWXYZ ", 26, 1)		
(c = toupper(getchar())) != EOF		
in(alphabet, c) T		F
mkset(temp, &c, 1, sizeof(c))		
in(einmal, c) TRUE		
FALSE		
assign(mehrmals, union(mehrmals, temp))	assign(einmal, union(einmal, temp))	
assign(nie, difference(alphabet, einmal))		
assign(einmal, difference(einmal, mehrmals))		
delete(temp), delete(alphabet)		

2. in C als Array-Implementierung

```
typedef enum { false, true } bool;
#define ANZAHL 26
```

```
/* lokale Variablen: */
const char *letter, *p;
int i, c;
```

```
void find_letter( bool einmal[], bool mehrmals[], bool nie[] )
```

```
letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
```

```
for (i = 0; i < ANZAHL; i++)
```

```
    einmal[i] = mehrmals[i] = false
```

```
    nie[i] = true
```

```
(c = toupper(getchar())) != EOF
```

```
(p = strchr(letter, c)) != NULL
```

T

F

```
i = p - letter
```

```
!mehrmals[i]
```

T

F

```
einmal[i]
```

TRUE

FALSE

```
mehrmals[i] = true
```

```
einmal[i] = true
```

```
einmal[i] = false
```

```
nie[i] = false
```

3.1.5 Sequenzen (sequences)

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

```
typedef struct {
    keytype key; /* eindeutige Identifizierung */
    datatype data; /* weitere Information ueber das Element */
} stdelement;
```

Außerdem wird davon ausgegangen, daß der Typ *bool* durch die Vereinbarung

```
typedef enum false, true bool;
```

gegeben ist.

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 `current_element`.

b. *find_next*

pre Die Sequenz ist nicht leer und `current_element` ist nicht das letzte Element.

post Der Nachfolger des `current_element` ist das neue `current_element`.

c. *append (const stdelement e)*

post Die Sequenz enthält `e` als letztes Element, und dieses Element ist das `current_element`.

d. *stdelement retrieve*

pre Die Sequenz ist nicht leer.

post *retrieve* ist eine Kopie des `current_element`.

e. *bool empty*

post if Sequenz enthält kein Element
then *empty* ist true
else *empty* ist false

f. *bool last*

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 zu übergeben, die die zu bearbeitende Sequenz eindeutig festlegt (siehe z.B. unten bei der Realisierung von sequentiellen Dateien in C).

3.1.5.1 Veranschaulichung einiger Operationen:

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



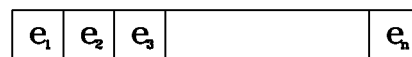
current_element

nach find_first



current_element

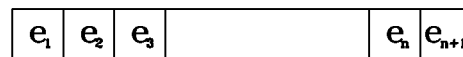
nach find_next



current_element

...

nach append(e_{n+1})



current_element

3.1.5.2 Sequentielle Dateien in C

In einigen Programmiersprachen ist die Sequenz als Datentyp enthalten. C-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 abzulegen, um später bei Bedarf wieder auf sie zuzugreifen. Aus diesem Grund erlaubt C die Definition einer *sequentiellen Datei*, dessen Komponentenzahl sich je nach Bedarf während des Programmablaufs dynamisch verändern läßt.

Sei *stream* eine Variable vom Typ FILE *.

Die Übergabe des Dateizeigers *stream* ermöglicht die Anwendung der Operationen auf unterschiedliche Dateien. Die Operationen, die den oben spezifizierten Operationen für eine Sequenz entsprechen, sind:

Operation

find_first

Realisierung in C

rewind(stream) ;

oder für den Fall, daß die Datei zwischenzeitlich abgeschlossen und noch nicht wieder eröffnet wurde:

stream = fopen(Dateiname, "rb+");

```

append(e)      fseek(stream, 0, SEEK_END);
                fwrite(&e, sizeof(stdelement), 1, stream);
                fflush(stream);
create          stream = fopen(Dateiname, "wb+");
                oder falls die Datei nicht über die Dauer des Programmablaufs hinaus benötigt
                wird:
                stream = tmpfile();
clear          freopen(Dateiname, "wb+", stream);

```

find_next und *retrieve* sind nur in der Kombination

```

e = retrieve();
find_next();

```

möglich. Die entsprechende C-Anweisung lautet:

```
fread(&e, sizeof(stdelement), 1, stream);
```

empty und *last* haben keine unmittelbare Entsprechung, jedoch gibt es die Boolesche Funktion `feof(stream)`, mit deren Hilfe das Erreichen des Dateiendes festgestellt werden kann. Diese Funktion liefert aber erst dann den Wert `true`, nachdem bereits versucht wurde, den Nachfolger des letzten Elementes (durch *retrieve*) anzusprechen.

Beispiel 3.7 Sequenz in C

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

```

#define SCHRANKE 0
#define INPUT " . . . "
#define OUTPUT " . . . "

typedef int stdelement;

FILE *inp, *out;
stdelement e;

```

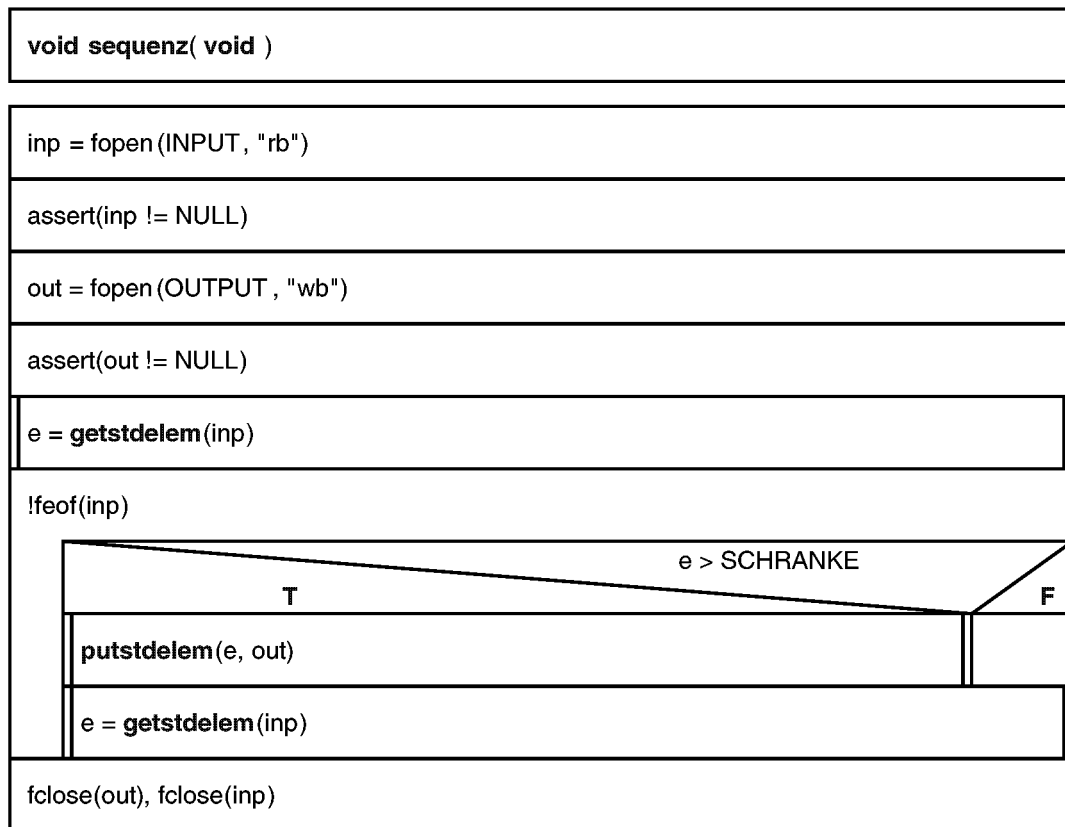
```
stdelement getstdelem( FILE *istream )
```

```
fread(&e, sizeof(stdelement), 1, istream)
```

```
return e
```

```
void putstdelem( stdelement e, FILE *ostream )
```

```
fwrite(&e, sizeof(stdelement), 1, ostream)
```

**Spezialfall:** Textdateien

Dateien mit Komponenten vom Typ `char` spielen eine besonders wichtige Rolle in der Datenverarbeitung. Dateien dieses Typs heißen in C Textdateien, sofern sie zusätzlich eine Zeilenstruktur (s.u.) beinhalten. Außerdem sind bei Ein-/Ausgabe auf Textdateien bestimmte Datenkonvertierungen erlaubt (z.B. darf das gelesene Element den Typ *float* haben).

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, in C das Zeichen `'\n'`), das das Ende einer Zeile festlegt.

Bemerkung:

Genaugenommen wird in C jede Datei als Folge von `char`-Elementen betrachtet. Da Elemente eines anderen Datentyps in Vektoren mit Komponenten des Typs `char` abgebildet werden können, ist es möglich, beliebige Datenelemente in einer Datei abzulegen. Insbesondere kann eine C-Datei auch Elemente unterschiedlichen Typs enthalten.

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 C durch eine kombinierte Nutzung von Records (Strukturen) und Pointern möglich.

3.2.1.1 Ausgezeichnete Pointer

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

3.2.1.2 Eigenschaften

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

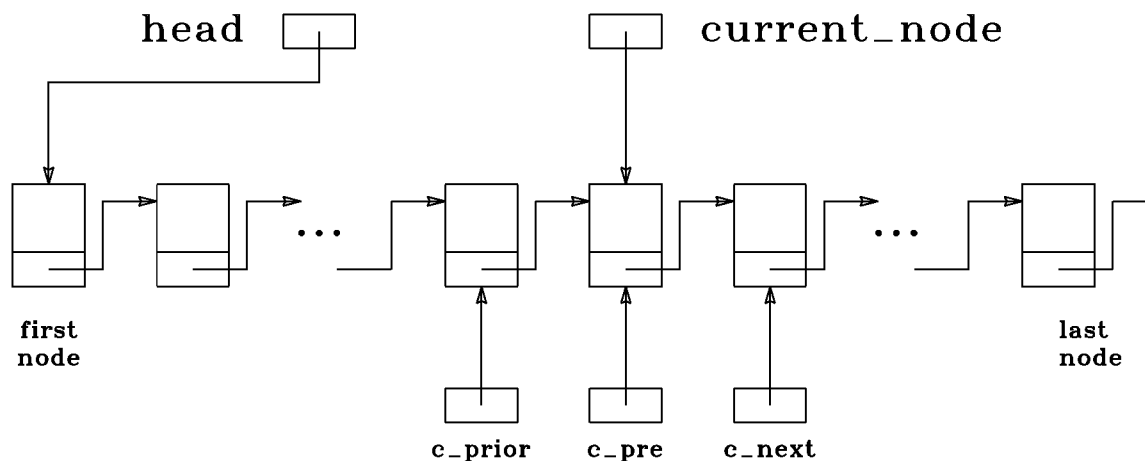


Abbildung 3.4: Beispiel für die Darstellung einer einfach verketteten Liste

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 aktuell 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. 3.4 auf der vorherigen Seite) 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.

*Basis-Operationen*a. *insert_after (const stdelement e)*

pre Die Liste ist nicht voll.
 post Ein neuer Knoten mit dem Datenelement **e** ist in der Liste enthalten, 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**.

b. *insert_before (const stdelement e)*

pre Die Liste ist nicht voll.
 post Ein neuer Knoten mit dem Datenelement **e** ist in der Liste enthalten, 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**.

c. *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.

d. *stdelement retrieve*

pre Die Liste ist nicht leer.
 post *retrieve* ist eine Kopie des Datenelements von **c_pre**.

e. *update (const stdelement e)*

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

*Manipulations-Operationen*a. *find_first*

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

b. *bool find_next*

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*; *find_next* ist true
 else *find_next* ist false.

c. *bool find_prior*

pre Die Liste ist nicht leer.
 post if *c_pre* zeigt nicht auf den ersten Knoten der Liste
 then *current_node* gleich *c_prior*; *find_prior* ist true
 else *find_prior* ist false.

d. *bool locate (const stdelement e)*

pre Die Liste ist nicht leer.
 post if *e* ist in der Liste
 then *locate* ist true und *current_node* zeigt auf den Knoten, der *e* als Datenelement enthält
 else *locate* ist false und *current_node* zeigt auf den letzten Knoten der Liste.

*Precondition-Test-Operationen*a. *bool empty*

post if Liste nicht leer
 then *empty* ist false
 else *empty* ist true.

b. *bool full*

post if Liste ist voll
 then *full* ist true
 else *full* ist false.

*Hilfs-Operation**create*

post Eine leere verkettete Liste existiert.

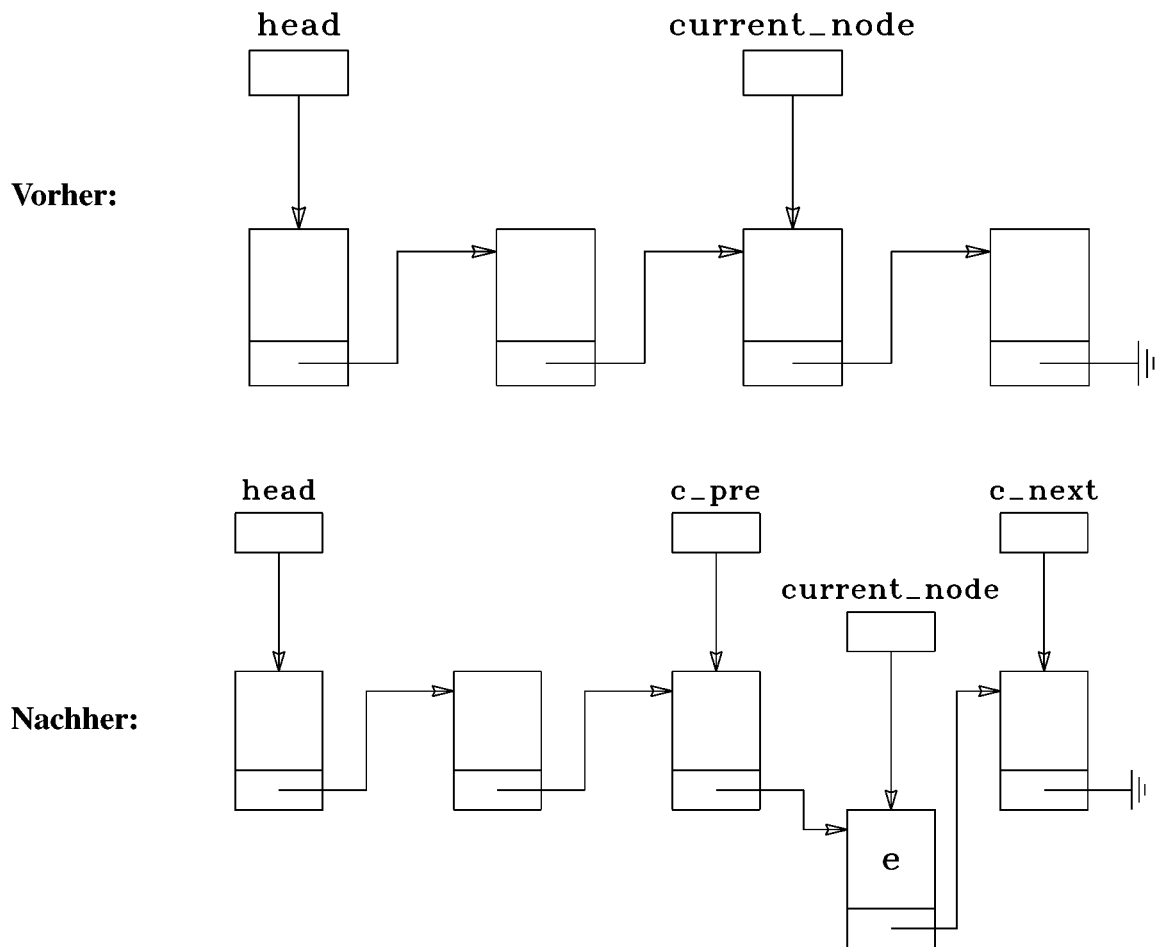
In Abb. 3.5 auf der nächsten Seite kann die Operation *insert_after* an einem Beispiel nachvollzogen werden.

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 3.8 Namenskonvention für *stdelement*

seq_stdelement für das *stdelement* einer Sequenz, oder **linked_stdelement** für das *stdelement* einer einfach verketteten Liste.

Abbildung 3.5: Liste vor und nach der Ausführung von `insert_after`**3.2.1.3 Realisierung über Pointer**

1. Repräsentation

```
typedef struct node *listpointer;

struct node {
    stdelement element;
    listpointer next;
};

/* globale Variablen: */
static listpointer head, current_node;

/* lokale Variablen: */
listpointer p;
bool found, not_failed;
```

2. Implementation

```
void create( void )
```

```
head = NULL
```

```
current_node = NULL
```

Funktion create

```
void insert_after( stdelement e )
```

```
p = (listpointer) malloc(sizeof(struct node))
```

```
assert(p != NULL)
```

```
p->element = e
```

```
head != NULL
```

T**F**

```
p->next = current_node->next
```

```
p->next = NULL
```

```
current_node->next = p
```

```
head = p
```

```
current_node = p
```

Funktion insert_after

```
void insert_before( stdelement e )
```

```
current_node == head
```

TRUE**FALSE**

```
p = (listpointer) malloc(sizeof(struct node))
```

```
p = head
```

```
assert(p != NULL)
```

```
p->next != current_node
```

```
p->element = e
```

```
p = p->next
```

```
p->next = head
```

```
current_node = p
```

```
head = p
```

```
current_node = p
```

```
insert_after(e)
```

Funktion insert_before

void delete(void)	
current_node != head	
T	F
p = head	head = head->next
p->next != current_node	
p = p->next	
p->next = current_node->next	
free(current_node)	
current_node = head	

Funktion delete

stdelement retrieve(void)
return (current_node->element)

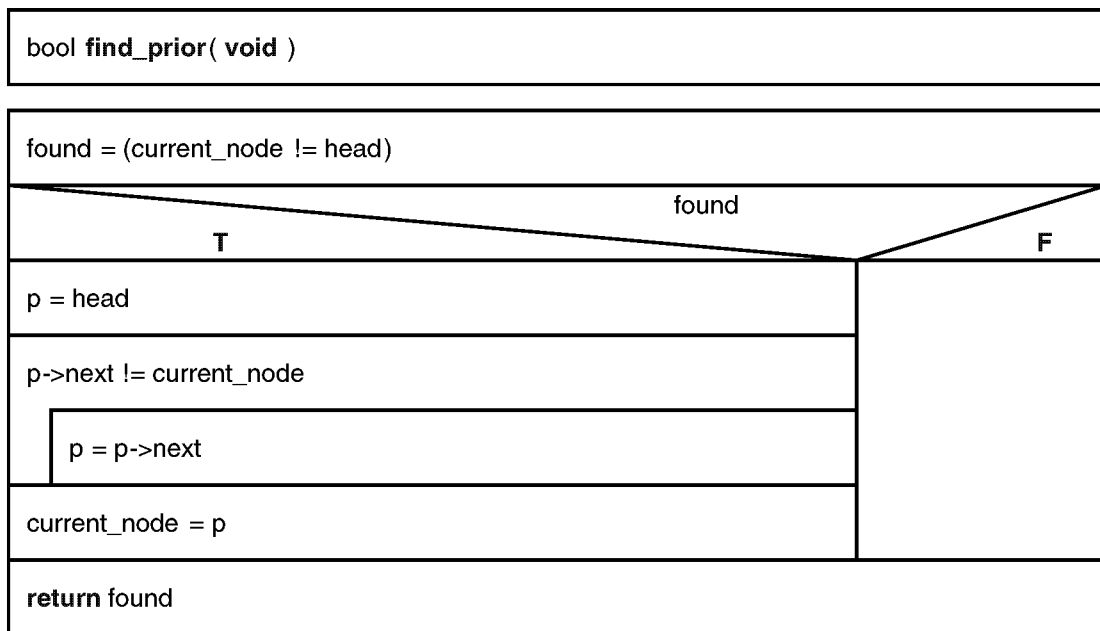
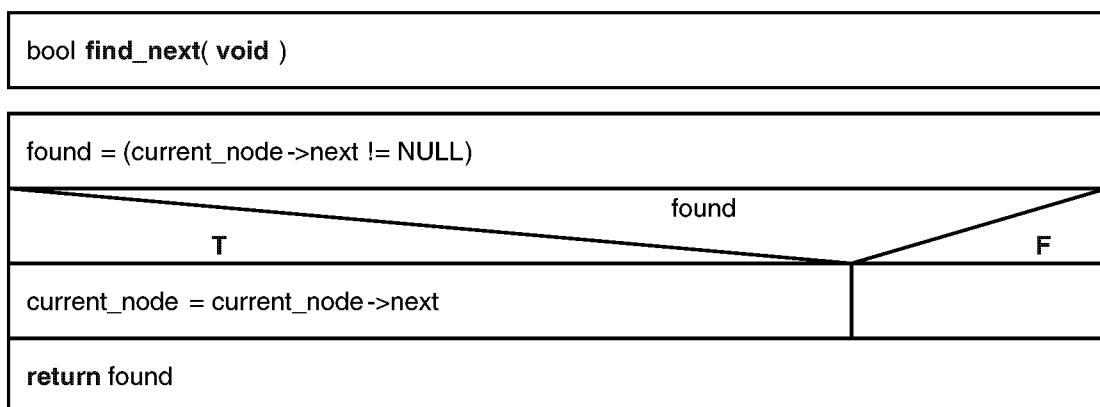
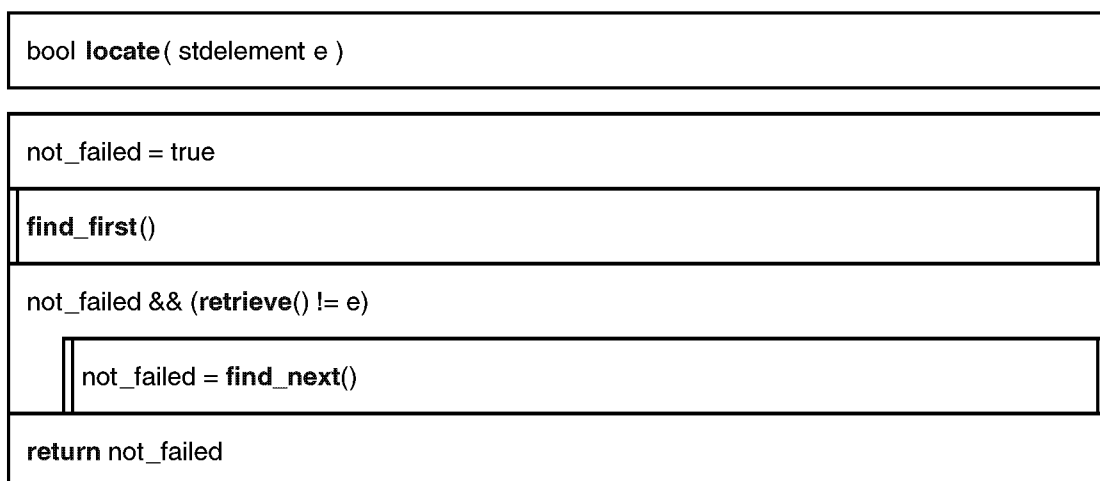
Funktion retrieve

void update(stdelement e)
current_node->element = e

Funktion update

void find_first(void)
current_node = head

Funktion find_first

**Funktion find_prior****Funktion find_next****Funktion locate**

```
bool empty( void )
```

```
return (head == NULL)
```

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.

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

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

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:

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

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:

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

retrieve Schnell, da unmittelbarer Zugriff über den Pointer current_node; d.h. $O(1)$.

update Schnell, da unmittelbarer Zugriff über den Pointer current_node; 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.

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

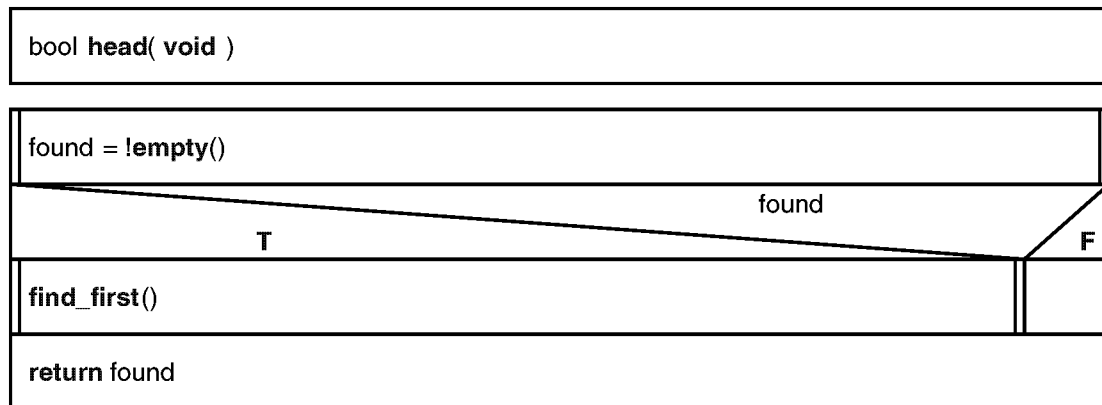
locate Langsam, da Liste von Anfang an durchsucht werden muß.

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

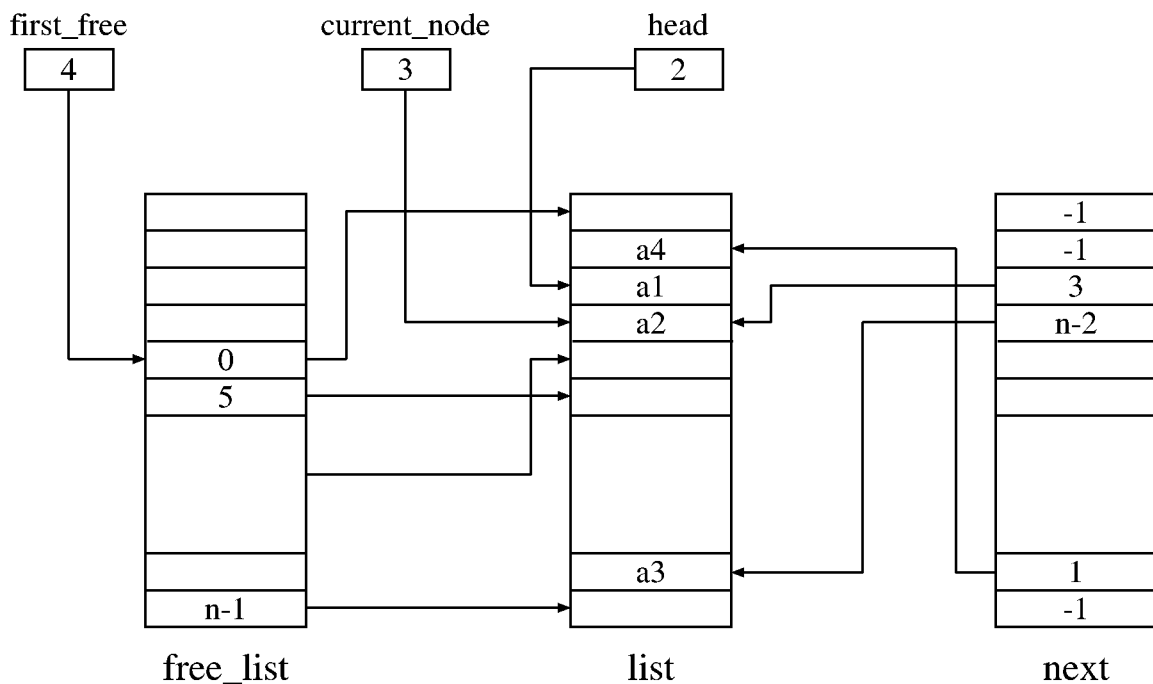
empty Schnell, da Test auf die Existenz eines Knotens in einem Schritt erfolgt; d.h. $O(1)$.

Bemerkung:

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

**Funktion head****3.2.1.4 Implementation einer verketteten Liste mit Hilfe von Arrays (statische Länge)**

Bei Implementation mit Hilfe von endlichen, statisch festgelegten Datenstrukturen benötigt man eine neue Funktion **full** (vgl. auch Definition 3.8 auf Seite 30). Diese Funktion überprüft, ob noch ein Element eingefügt werden kann. Dieses war bei dynamischen Datenstrukturen unwichtig, da nur der Adreßbereich eine Grenze setzt.

**Abbildung 3.6: Liste mit drei Feldern**

3.2.1.5 Realisierung mit Hilfe von Feldern

1. Repräsentation

```
typedef int listpointer; /* benoetigter Wertebereich: -1..MAXLIST */
typedef listpointer listpointer_array[MAXLIST];
```

```
/* globale Variablen: */
static stdelement list[MAXLIST];
static listpointer_array next, free_list;
static listpointer head, current_node;
static listpointer first_free;
```

```
/* lokale Variablen: */
```

```
listpointer i, p; /* Hilfsvariablen */
bool found, not_failed;
```

2. Implementation

```
void create( void )
```

```
head = -1
```

```
current_node = -1
```

```
for (i = 0; i < MAXLIST; i++)
```

```
    free_list[i] = i
```

```
    next[i] = -1
```

```
first_free = 0
```

Funktion create

```
void insert_after( stdelement e )
```

```
p = free_list[first_free++]
```

```
list[p] = e
```

```
current_node >= 0
```

T

F

```
next[p] = next[current_node]
```

```
next[p] = -1
```

```
next[current_node] = p
```

```
head = p
```

```
current_node = p
```

Funktion insert_after

void insert_before(stdelement e)	
current_node != head	
TRUE	FALSE
p = head	p = free_list[first_free++]
next[p] != current_node	list[p] = e
p = next[p]	next[p] = head
current_node = p	head = p
insert_after(e)	current_node = p

Funktion insert_before

void delete(void)	
current_node != head	
T	F
p = head	head = next[head]
next[p] != current_node	
p = next[p]	
next[p] = next[current_node]	
free_list[--first_free] = current_node	
current_node = head	

Funktion delete

bool full(void)
return (first_free == MAXLIST)

Funktion full**3.2.1.6 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.

3.2.1.7 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

```
typedef int list_index; /* benoetigter Wertebereich: -1..MAXLIST */

/* globale Variablen: */
static stdelement list[MAXLIST];
static list_index last_node; /* == Anzahl Knoten in der Liste - 1 */
static list_index current_node;

/* lokale Variablen: */
list_index k;
bool found, not_failed;
```

2. Implementation

```
void create( void )
```

```
last_node = -1
```

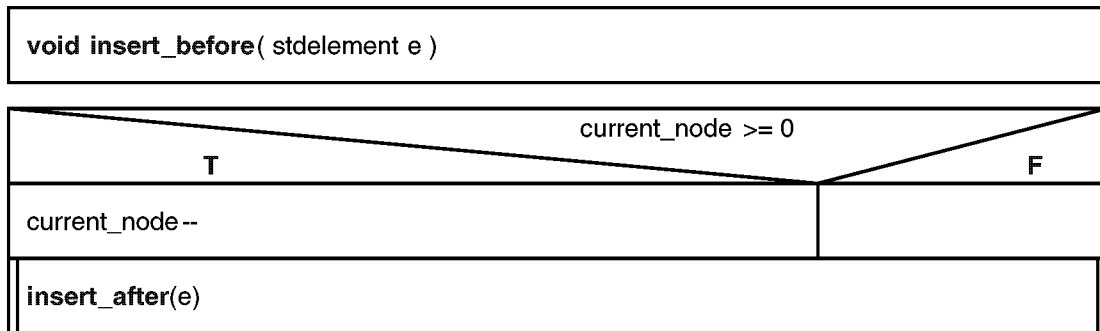
```
current_node = -1
```

Funktion create

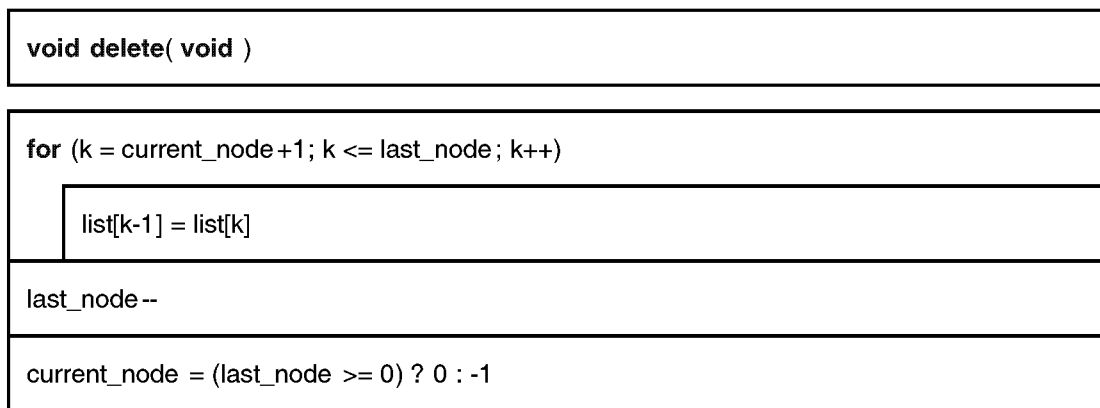
```
void insert_after( stdelement e )
```

last_node >= 0	
T	F
for (k = last_node; k > current_node; k--)	
list[k+1] = list[k]	
list[++current_node] = e	
last_node++	

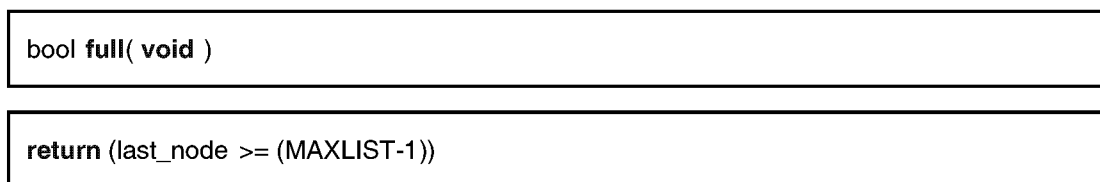
Funktion insert_after



Funktion insert_before



Funktion delete



Funktion full

3.2.1.8 Beurteilung der elementaren Listenoperationen bei einer Feldimplementation ohne Pointer

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

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

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

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

Die übrigen Operationen create, insert_before, delete, retrieve, update, find_next 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 auf Seite 30 übernommen werden.

3.2.2.1 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.

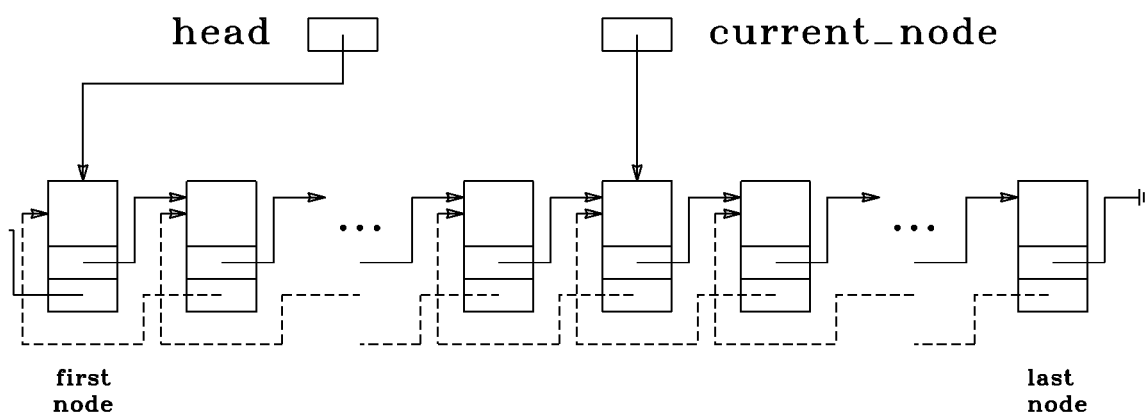


Abbildung 3.7: Beispiel für die Darstellung einer doppelt verketteten Liste

3.2.2.2 Realisierung einer doppelt verketteten Liste

1. Repräsentation

```
typedef struct node *listpointer;

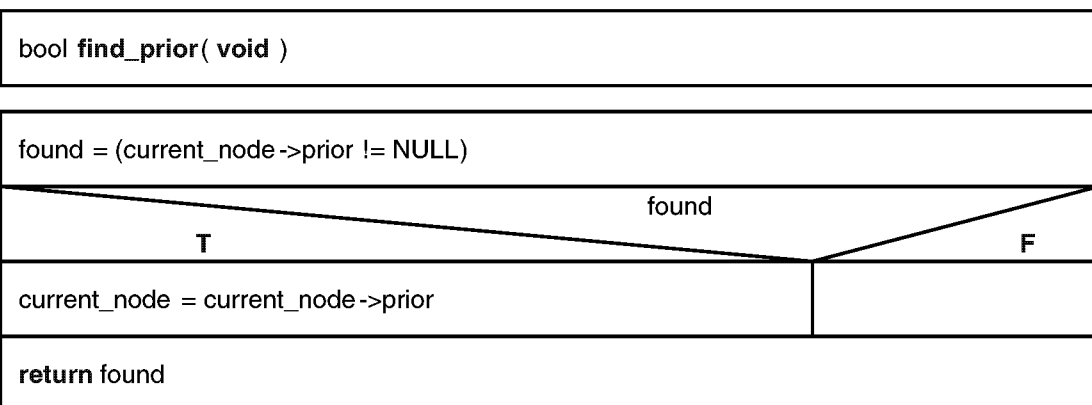
struct node {
    stdelement element;
    listpointer next;
    listpointer prior;
};

/* globale Variablen: */
static listpointer head, current_node;

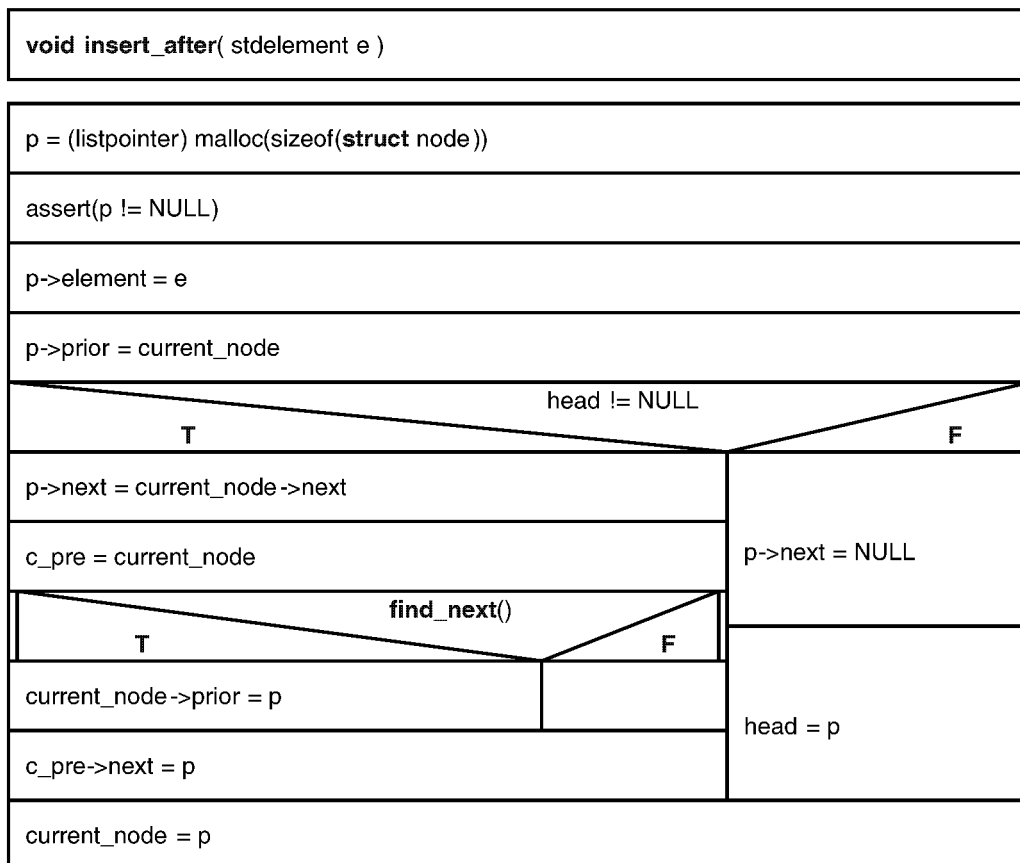
/* lokale Variablen: */
listpointer p, c_pre;
bool found, not_failed;
```

2. Implementation

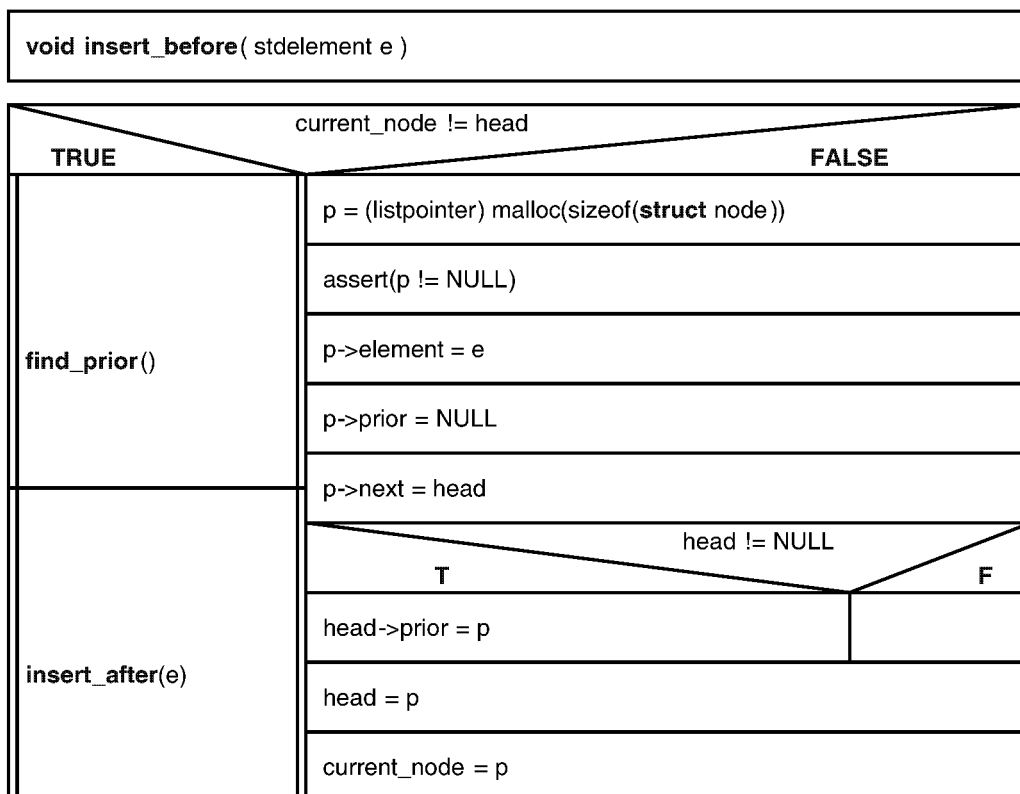
Die Veränderungen der Implementation bei den meisten Operationen sind geringfügig und können in [10] nachgelesen werden. Besondere Beachtung gilt den Operationen find_prior, insert_after, insert_before und delete.



Funktion find_prior



Funktion insert_after



Funktion insert_before

void delete(void)			
current_node != head			
TRUE		FALSE	
c_pre = current_node		head = head->next	
find_prior()			
current_node->next = c_pre->next		free(current_node)	
free(c_pre)		!empty()	
c_pre = current_node		T F	
find_next()		find_first()	
T F			
current_node->prior = c_pre		current_node->prior = NULL	
current_node = head			

Funktion delete

3.2.2.3 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)

Tabelle 3.1: Zeitkomplexitäten der einzelnen Basisoperationen in verschiedenen Listenrealisierungen

3.2.2.4 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**).

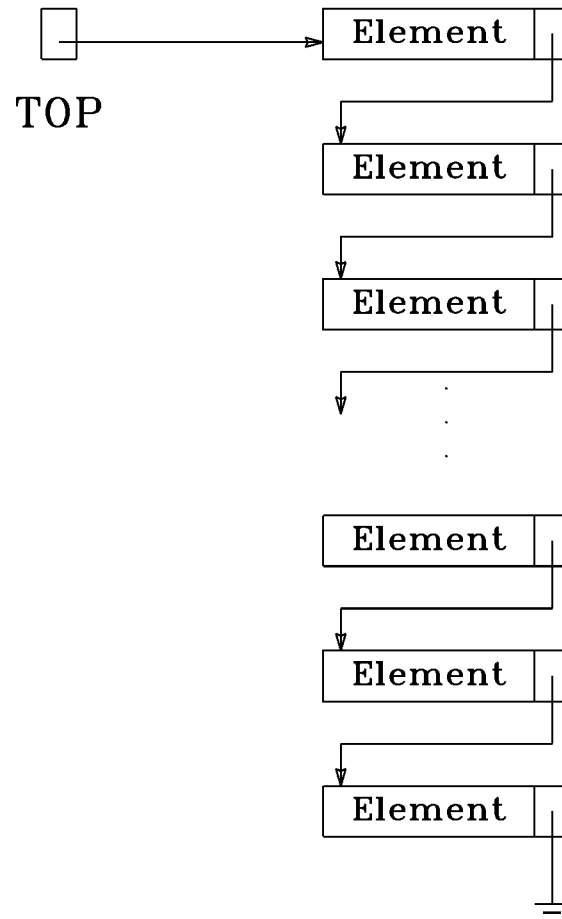


Abbildung 3.8: Beispielhafte Darstellung eines Stacks

3.2.3.1 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 Softwaremäßig 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 (*sdelement*).

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 (const sdelement e)*

pre Der Stack ist nicht voll.

post Der Stack enthält *e* als das zuletzt angekommene Element.

b. *sdelement pop*

pre Der Stack ist nicht leer.

post *pop* ist das zuletzt angekommene Element. Der Stack enthält dieses Element nicht mehr.

c. *bool empty*

post if Der Stack enthält kein Element
 then *empty* ist true
 else *empty* ist false

d. *bool full*

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.

3.2.3.2 Realisierung mit Hilfe einer verketteten Struktur in C

1. Repräsentation

```
typedef struct stack_element *stack_pointer;

struct stack_element {
    sdelement    element;
    stack_pointer next;
};

/* globale Variablen: */
static stack_pointer top;

/* lokale Variablen: */
stack_pointer p;
sdelement e;
```

2. Implementation

```
void create( void )
```

```
top = NULL
```

Funktion create

```
void push( stdelement e )
```

```
p = (stack_pointer) malloc(sizeof(struct stack_element))
```

```
assert(p != NULL)
```

```
p->element = e
```

```
p->next = top
```

```
top = p
```

Funktion push

```
stdelement pop( void )
```

```
e = top->element
```

```
p = top
```

```
top = top->next
```

```
free(p)
```

```
return e
```

Funktion pop

```
bool empty( void )
```

```
return (top == NULL)
```

Funktion empty

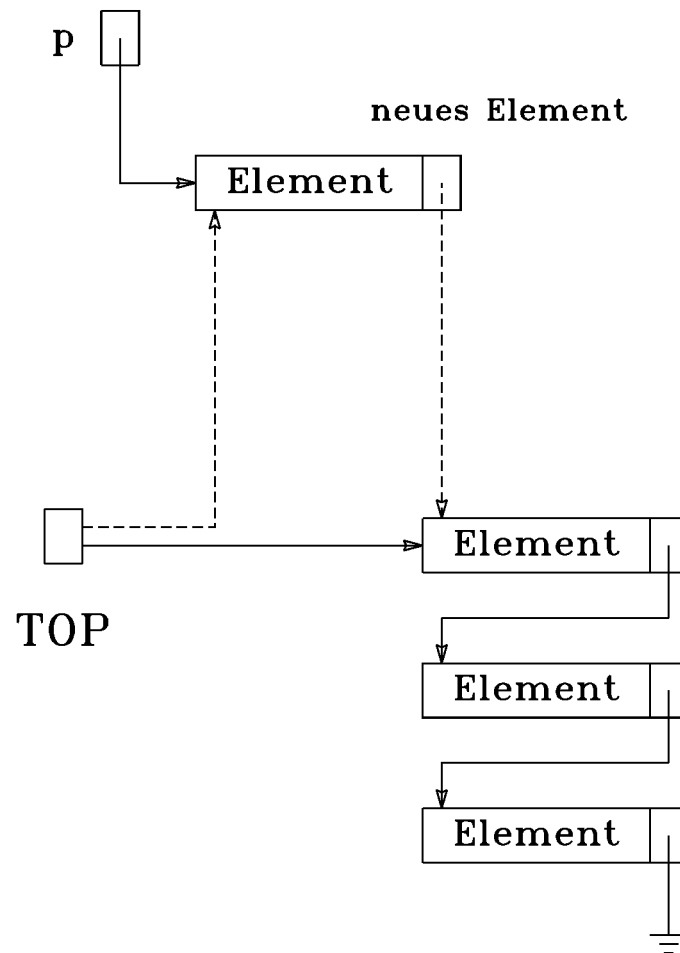


Abbildung 3.9: Darstellung der Operation push auf dem Datentyp Stack

3.2.3.3 Realisierung eines Stacks mit Hilfe eines Feldes

1. Repräsentation

```
typedef int stackbereich; /* benoetigter Wertebereich: */
                        /* -1..MAX_STACK-1 */
typedef stdelement stack_type[MAX_STACK];

/* globale Variablen: */
static stackbereich top;
static stack_type stack;
```

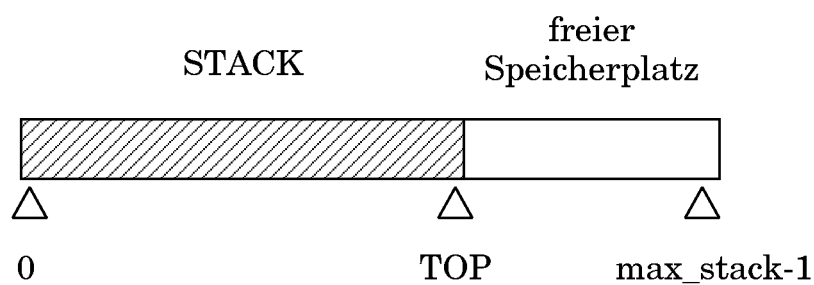


Abbildung 3.10: Visualisierung der Feldimplementierung eines Stacks

2. Implementation

```
void create( void )
```

```
top = -1
```

Funktion create

```
void push( stdelement e )
```

```
stack[++top] = e
```

Funktion push

```
stdelement pop( void )
```

```
return (stack[top--])
```

Funktion pop

```
bool empty( void )
```

```
return (top == -1)
```

Funktion empty

```
bool full( void )
```

```
return (top == (MAX_STACK-1))
```

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)

Tabelle 3.2: Zeitkomplexitäten für die Operationen auf der Datenstruktur Stack

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

Definition 3.11 *Rekursive Funktion*

Eine rekursive Funktion ist eine Funktion, die unmittelbar durch sich selbst oder mittelbar über andere Funktionen wieder aufgerufen wird.

Bemerkung:

In C und Fortran 90 sind rekursive Funktionsaufrufe erlaubt, in FORTRAN 77 waren rekursive Programmaufrufe jedoch verboten.

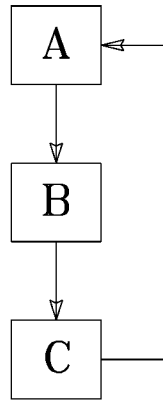


Abbildung 3.11: Mittelbarer rekursiver Aufruf

Beispiel 3.9 *Rekursive Funktionen durch unmittelbaren Aufruf*

```

int fakultaet( int n )
{
    return ( n == 0 ) ? 1 : fakultaet(n-1)*n;
}
  
```

Beispiel 3.10 *LIFO-Stack zur Realisierung rekursiver Funktionsaufrufe*

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

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 )
{
    result = 1;
    return result;
}
else
{
    result = fakultaet(n-1);
    result = n * result;
    return result;
}
  
```

Stackbelegung bei der rekursiven Funktion fakultaet:

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

Bemerkung:

1. Die oben beschriebene Methode der Stackverwaltung heißt dynamische Speicherverwaltung.
2. Jeder Unterprogramm- oder Funktionsaufruf bewirkt die Bereitstellung eines neuen Speicherbereichs auf dem Datenstack, der für die lokalen Variablen der gerufenen Funktion 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 (siehe z.B. das Backtracking-Verfahren). Das Abwägen zwischen der Effizienz der Programmierung und dem Laufzeitverhalten ist manchmal jedoch notwendig.
5. FORTRAN 77 hatte eine statische Speicherverwaltung, d.h. rekursive Programmlösungen waren nicht möglich.
6. C 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.

Backtracking:

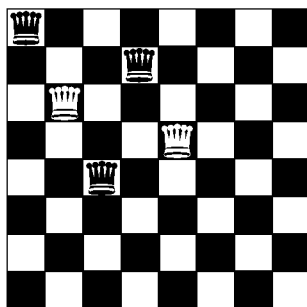
Ein Beispiel für ein einfach zu programmierendes rekursives Verfahren zur Lösung sehr komplexer Probleme ist das *Backtracking-Verfahren*. Es eignet sich zur Lösung von Problemen, die sich in eine Reihe von Teilproblemen zerlegen lassen, von denen jedes einzelne Teilpro-

blem mehrere mögliche Lösungen haben kann und die Entscheidung für eine bestimmte Alternative zur Lösung eines Teilproblems Auswirkungen darauf hat, wie mögliche Lösungen für spätere Teilprobleme aussehen. Beim Backtracking-Verfahren wird versucht eine Lösung für das Gesamtproblem zu finden, indem systematisch die Alternativen zur Lösung jedes Teilproblems ausprobiert und zur Gesamtlösung kombiniert werden. Dieses Prinzip des *Versuchens und Nachprüfens* (trial and error) führt bei nicht erfolgreichen Versuchen in eine Sackgasse. Dann war irgendeine vorher getroffene Entscheidung zur Lösung eines Teilproblems falsch und führte dazu, daß man sich von der Gesamtlösung entfernt hat, anstatt sich an sie anzunähern. In diesem Fall muß durch das Zurücknehmen der letzten Entscheidung bzw. auch mehrerer der letzten Entscheidungen zur letzten möglichen Alternative zurückgekehrt werden (daher der Name Backtracking) und dort eine andere Alternative gewählt werden.

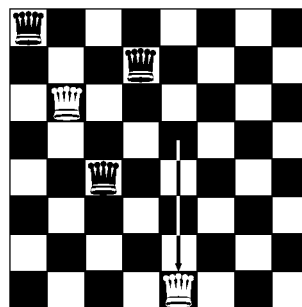
Beispiel 3.11 Acht-Damen-Problem

Ein Problem, das sich mit Hilfe des Backtracking-Verfahrens lösen läßt, ist z.B. das Acht-Damen-Problem, bei dem auf einem Schachbrett 8 Damen so aufgestellt werden sollen, daß keine Dame von einer anderen geschlagen werden kann. Dies bedeutet, daß in keiner Zeile, Spalte oder Diagonale mehr als eine Dame aufgestellt werden darf.

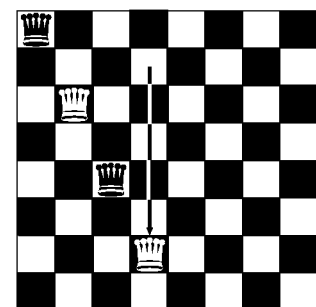
Geht man beim Aufstellen der Damen spaltenweise vor, dann kann das Plazieren der i -ten Dame innerhalb der i -ten Spalte unter Berücksichtigung der bereits auf dem Schachbrett aufgestellten $i-1$ Damen als das i -te Teilproblem angesehen werden. Durchläuft man die Spalten beim Lösen der Teilprobleme von links nach rechts und die möglichen Lösungen eines Teilproblems immer von oben nach unten, dann ergibt sich als erste Sackgasse die unten links abgebildete Situation, in der in Spalte 6 keine weitere Dame mehr aufgestellt werden kann. Im nächsten Schritt kehrt man zur Spalte 5 zurück und verschiebt dort die Dame auf die nächstmögliche Position. Da immer noch keine weitere Dame in Spalte 6 hinzugenommen werden kann und bereits alle Möglichkeiten eine Dame in Spalte 5 aufzustellen ausprobiert sind, wird weiter zurückgegangen und die Dame in Spalte 4 verschoben. Anschließend wird dann wieder versucht, die nachfolgenden Spalten zu besetzen, usw.



1. Sackgasse

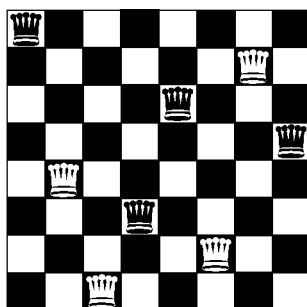


2. Sackgasse



Rückkehr zur Spalte 4

Die erste von insgesamt 92 Lösungen, die man auf diese Weise erhält, ist:

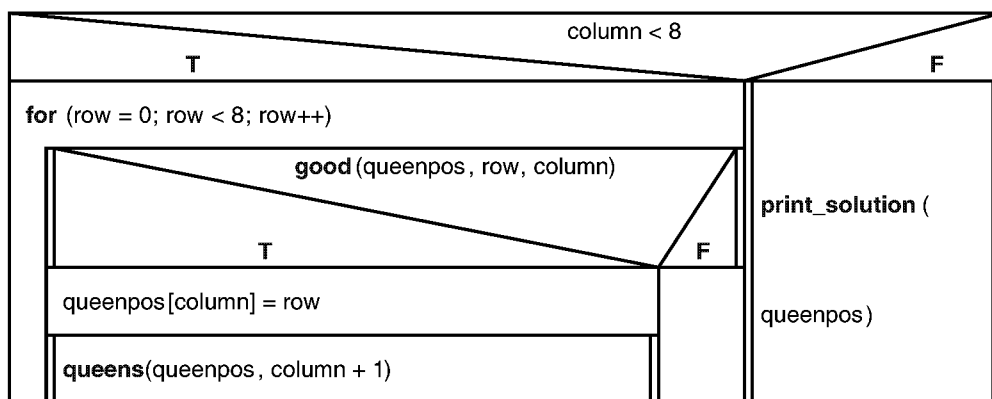


Die folgende rekursive Realisierung des beschriebenen Lösungsverfahrens verwendet das eindimensionale Feld `queenpos[8]`, um die Positionen der auf dem Schachbrett aufgestellten Damen abzuspeichern. `queenpos[i]` enthält dazu den Index der Zeile, in der die Dame in Spalte `i` aufgestellt ist.

```
void solve_queens_problem( void )
```

```
    queens(queenpos, 0)
```

```
void queens(int queenpos[], int column)
```



```
int good(int queenpos[], int row, int column)
```

```
    retval = 1
```

```
    for (i = 0; (i < column) && retval; i++)
```

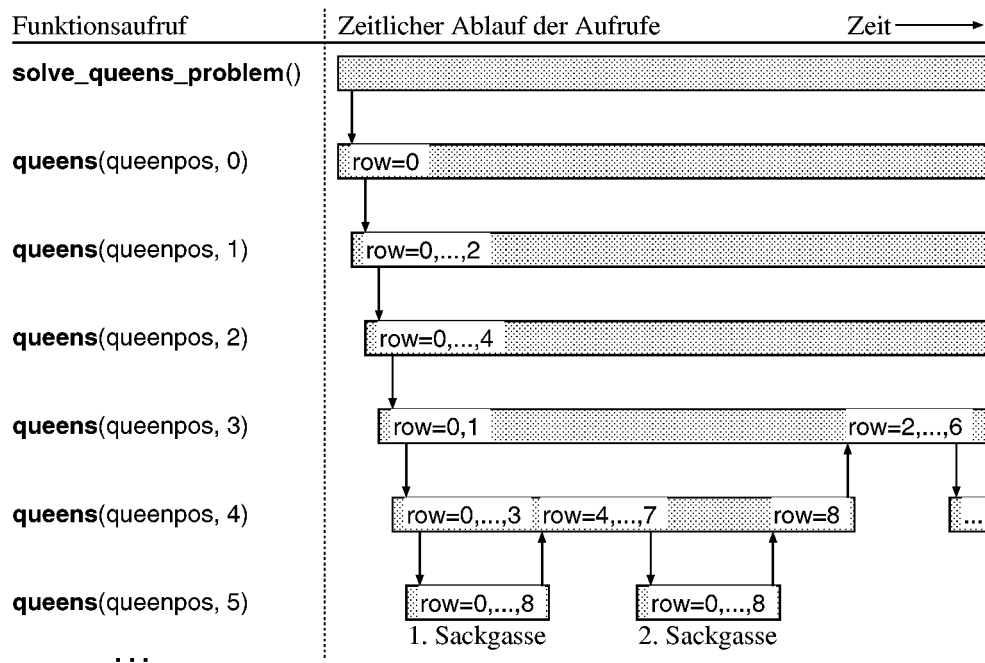
```
        die Dame in Spalte i kann die Dame an Position row/ column schlagen
```

```
        retval = 0
```

```
    return retval
```

Bemerkung:

Das folgende Diagramm veranschaulicht die Abfolge der durch einen Aufruf von `solve_queens_problem` ausgelösten Funktionsaufrufe bis zur Rückkehr aus der zweiten Sackgasse.



Backtracking führt sehr schnell (exponentielles Wachstum) zu laufzeitintensiven Programmen, so daß es in der Praxis häufig notwendig wird, Zusatzbedingungen zu finden, durch die möglichst viele Sackgassen von vorne herein ausgeschlossen werden können.

3.2.4 Warteschlangen (queues)

Eine Warteschlange (auch Queue, FIFO-Liste, FIFO-Stack genannt) ist eine Liste, bei der nur am Listenende 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 im allgemeinen 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 (const stdelement e)*

```

pre    Die Queue ist nicht voll.
post   Die Queue enthält e als das zuletzt angekommene Element.

```

- b. *stdelement get*
 pre Die Queue ist nicht leer.
 post *get* ist das Element, das am längsten in *q_pre* ist. Dieses Element ist nicht länger in der Queue enthalten.
- c. *bool empty*
 post if Die Queue ist leer
 then *empty* == true
 else *empty* == false
- d. *bool full*
 post if Die Queue ist voll
 then *full* == true
 else *full* == false
- e. *int length*
 post *length* gibt die Anzahl der Elemente in der Queue an.
- f. *create*
 post Die Queue existiert und ist leer.
- g. *clear*
 post Die Queue ist leer.

Operationen: PUT am tail
 GET am head

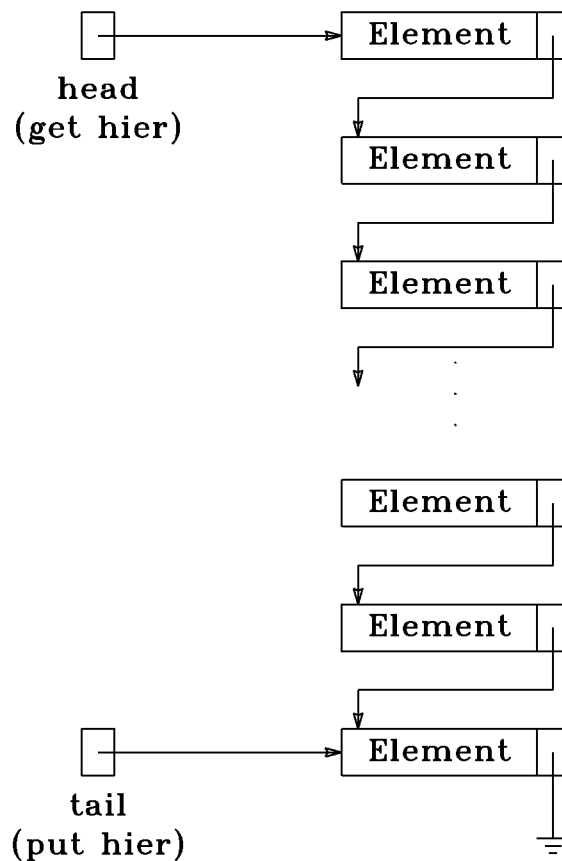


Abbildung 3.12: Queue (übliche Darstellung)

Einsatzgebiete von Queues

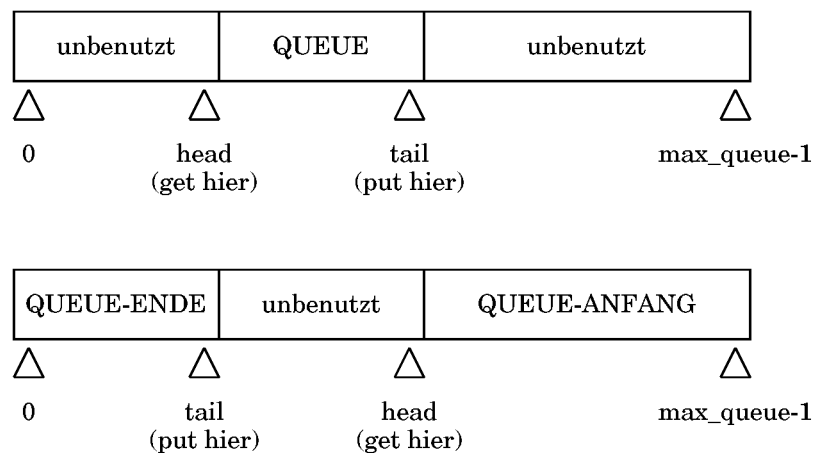
- im täglichen Leben
 - Anstellen am Schalter der Post.
- Betriebssysteme
 - Zuteilung von Ressourcen
 - * CPU-Verwaltung
 - * I/O Request (zirkuläre Queues)
 - * Disk-Verwaltung

Realisierung einer Queue mit Hilfe eines Feldes**1. Repräsentation**

```
#define MAX_QUEUE ...

/* globale Variablen: */
static stdelement queue[MAX_QUEUE];
static int head, tail;
    /* benoetigter Wertebereich: -1..MAX_QUEUE-1*/

/* lokale Variablen: */
int h, len;
```

**Abbildung 3.13:** Queue, realisiert mit einem Feld bei unterschiedlichem Füllungsgrad

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

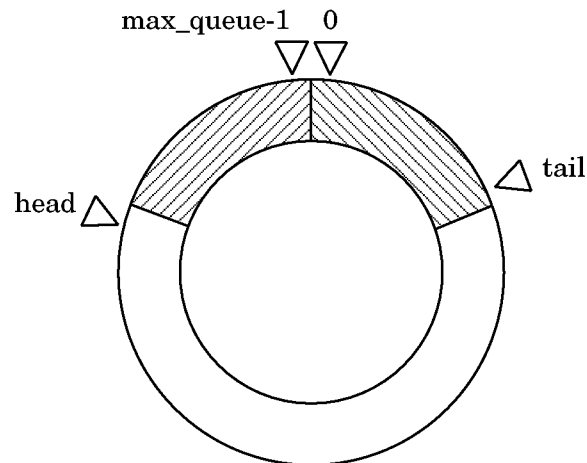


Abbildung 3.14: Visualisierung einer zirkulären Queue

2. Implementation

```
void create( void )
```

```
tail = -1
```

```
head = MAX_QUEUE -1
```

Funktion create

```
void clear( void )
```

```
create()
```

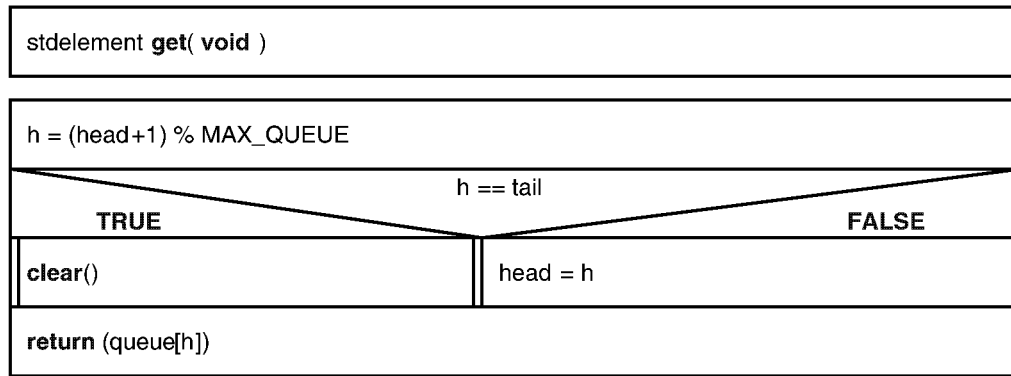
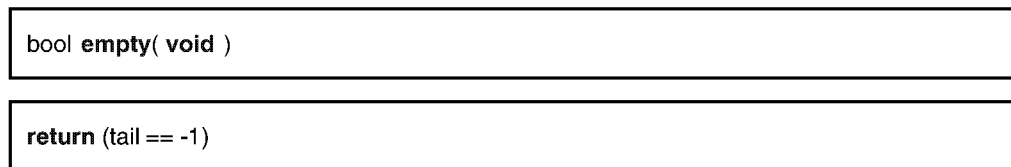
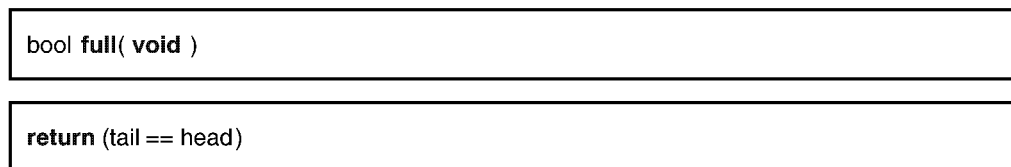
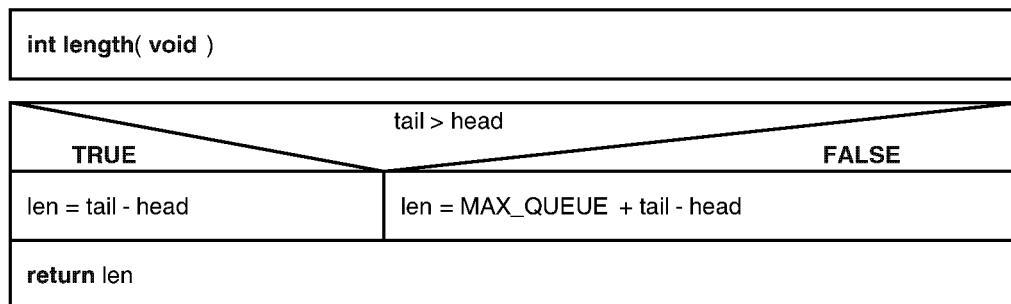
Funktion clear

```
void put( stdelement e )
```

```
tail = (tail+1) % MAX_QUEUE
```

```
queue[tail] = e
```

Funktion put

**Funktion get****Funktion empty****Funktion full****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 $E = \{(1,2), (1,4), (2,3), (3,5), (5,1)\}$

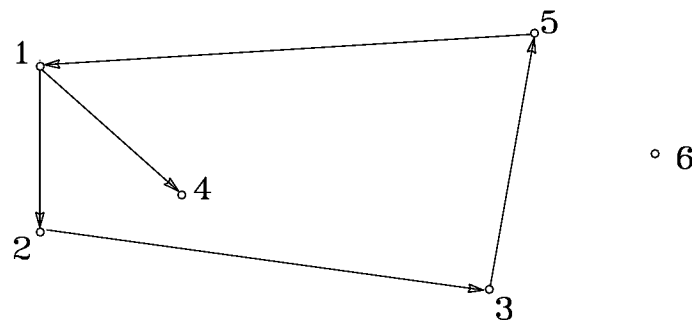


Abbildung 3.15: Beispiel für einen gerichteten Graph

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*

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

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

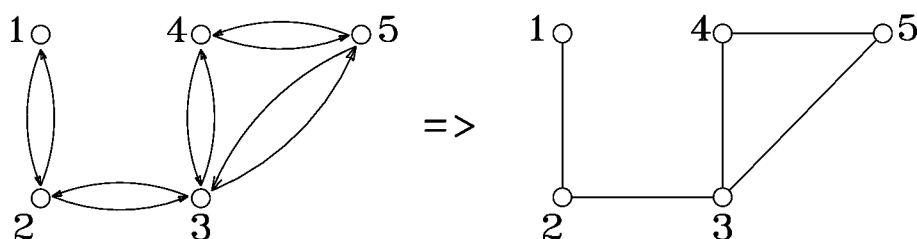
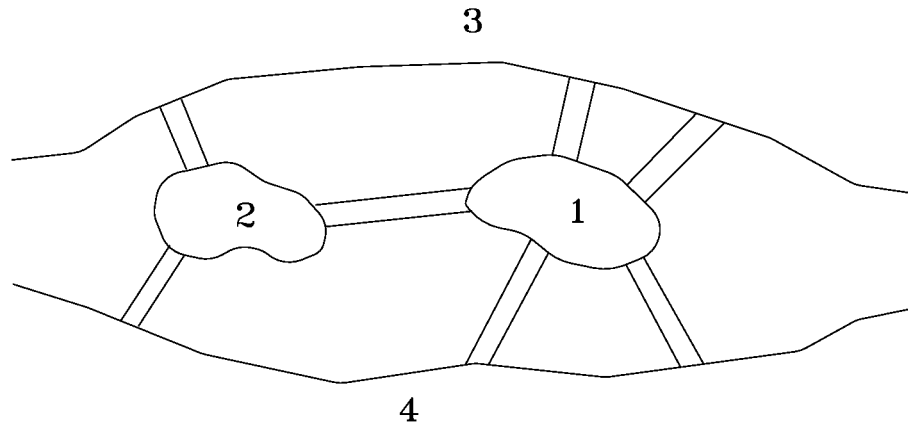


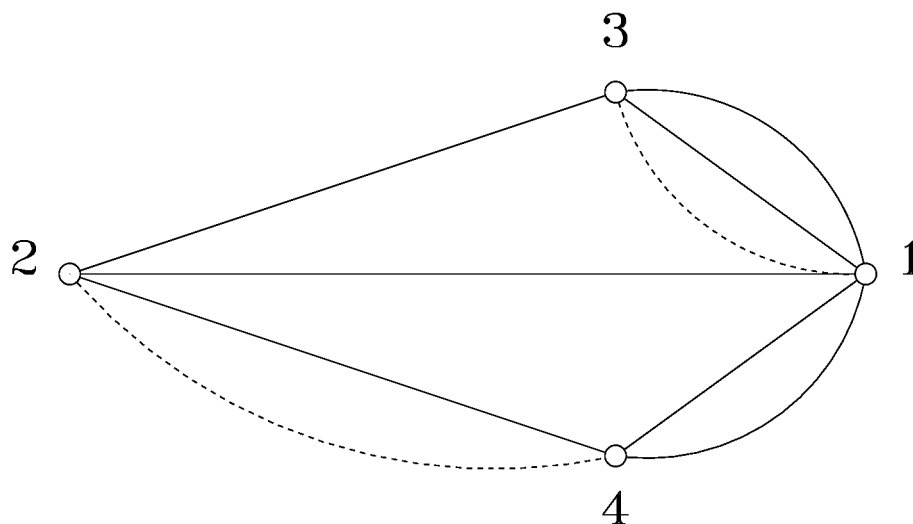
Abbildung 3.16: Übergang von symmetrisch gerichtetem zu einem ungerichteten Graph

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? Euler zeigte, daß es keinen solchen Weg

**Abbildung 3.17:** Königsberger Brückenproblem

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.

**Abbildung 3.18:** 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*

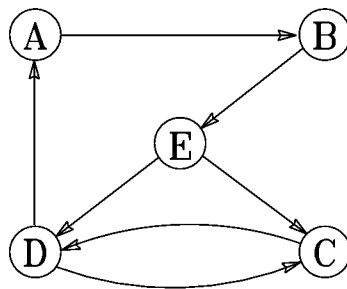
Sei $G=(V, E)$ ein gerichteter oder ungerichteter Graph.

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

$v_1 = \alpha(W)$ heißt *Anfangsknoten* des Weges W , $v_n = \omega(W)$ heißt *Endknoten* des Weges W . ($v_i = v_j$) heißt *trivialer* Weg und ist stets ein Weg in $G \forall v_i \in V$.

Die Länge eines Weges ist $l(W) = n-1$, falls n Knoten auf diesem Weg besucht werden.

Beispiel 3.15 *Wege in einem Graphen*

Wege:

- (A,B)
- (B,E,D,C)
- (D,C,D,C,D,C)
- (A,B,E,D,A)

Abbildung 3.19: 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 auf der vorherigen Seite:

	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

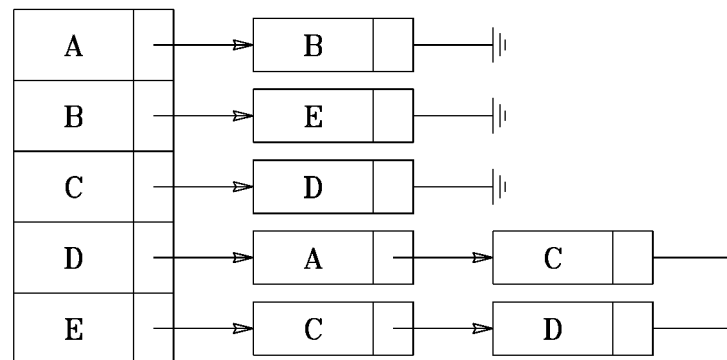
Bemerkung:

- 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 auf der vorherigen Seite)



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!

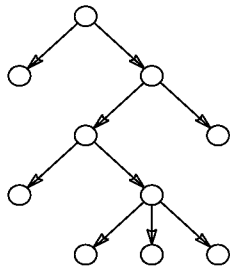
3.4 Bäume

Bei Bäumen handelt es sich um Graphen mit speziellen Eigenschaften.

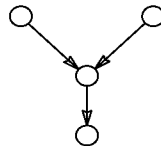
Definition 3.16 *Baum*

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 $v_i \in V$ 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

Bemerkung:

In einem Baum existieren keine Kreise (cycle).

- 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 Binärbaum*. 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 ausgezeichnete 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)

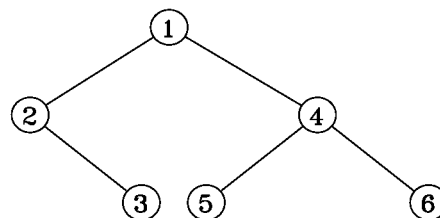
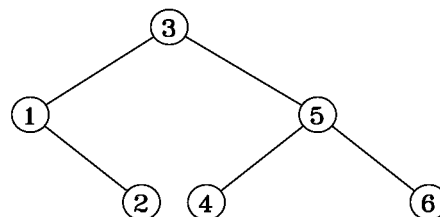
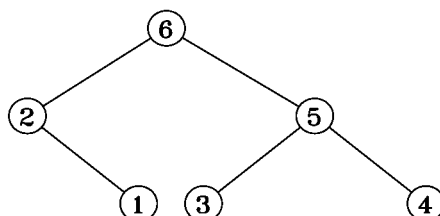
Preorder**Inorder****Postorder**

Abbildung 3.20: Durchlaufstrategien eines Baumes (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. 3.21 und Abb. 3.22).

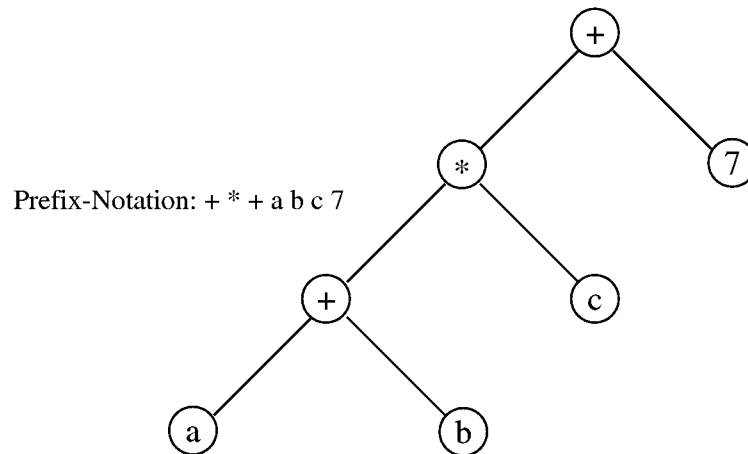


Abbildung 3.21: Speicherung des Infix-Ausdrucks $(a + b) * c + 7$

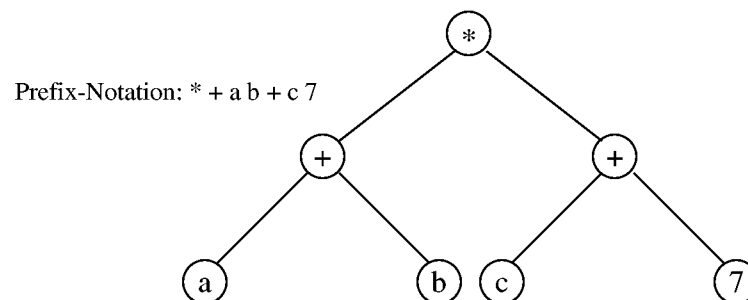


Abbildung 3.22: Speicherung des Infix-Ausdrucks $(a + b) * (c + 7)$

Bemerkung:

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

Folgende Datentypen werden für den Binärbaum vereinbart:

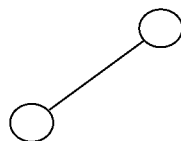
```
typedef enum {preorder, inorder, postorder} order;
typedef enum {rootv, leftchild, rightchild, parent} relative;
```

```
struct status {
    int    size; /* number of nodes */
    int    height;
    double average_path_length;
};
```

Baum 1



Baum 2



Baum 3

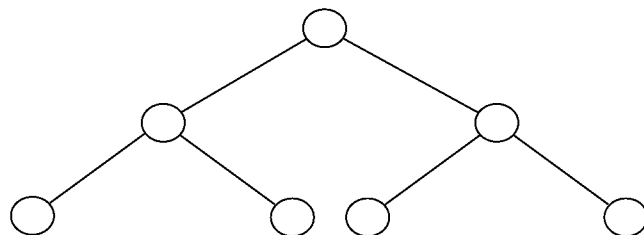


Abbildung 3.23: Beispielbäume

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

Tabelle 3.3: Statuswerte für die Bäume aus Abb. 3.23 auf der vorherigen Seite

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 Vater und keinen, einen oder zwei Söhne. 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 (order traverse_order,*

*void process (stdelement *e_ptr, int level, void *data_ptr),*

*void *data_ptr)*

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. switch (*traverse_order*)

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 Teilbäume bearbeitet.

Die Art der Verarbeitung eines Knotens (bzw. seiner Daten) wird durch die Funktion *process* festgelegt. Der Datenzeiger *data_ptr* wird an die Funktion *process* bei jedem Aufruf übergeben und kann auf einen Datenbereich verweisen, der für die Verarbeitung der Knoten benötigt wird.

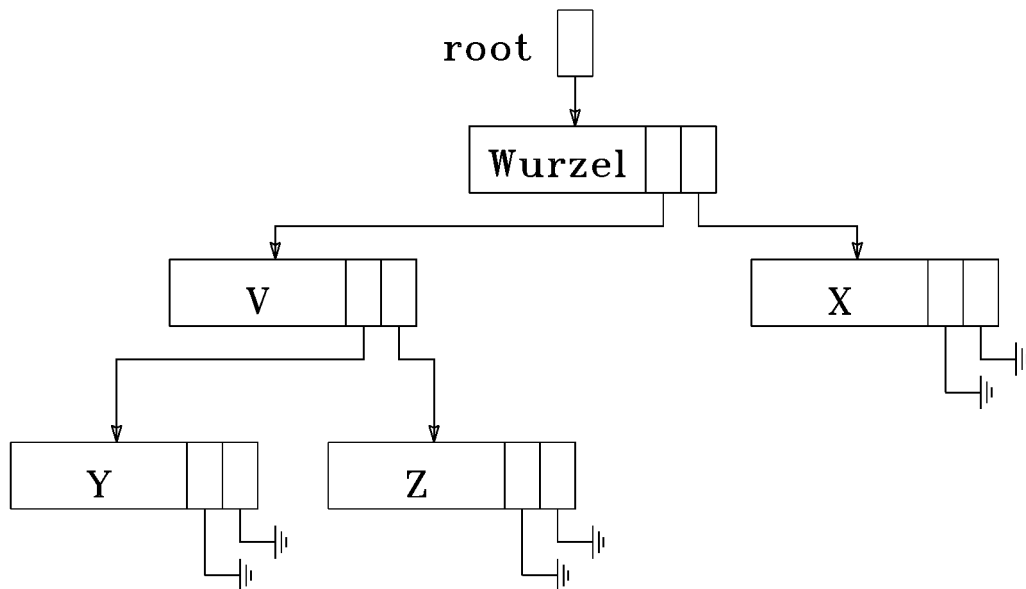
b. *bool insert (const stdelement e, relative rel)*

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*. switch (*rel*)
- rootv: e ist Element der Wurzel des Baumes, und *insert* ist true.
- leftchild: if c_pre hat keinen linken Sohn
 then c_pre hat einen linken Sohn, dieser Knoten enthält *e* als Datenelement, und *insert* ist true
 else *insert* ist false.
- rightchild: if c_pre hat keinen rechten Sohn
 then c_pre hat einen rechten Sohn, dieser Knoten enthält *e* als Datenelement, und *insert* ist true
 else *insert* ist false.
- parent: *insert* ist false.
- 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. *stdelement retrieve*
- pre Der Baum ist nicht leer.
- post *retrieve* ist eine Kopie des Datenelements von *c_pre*.
- e. *update (const stdelement e)*
- pre Der Baum ist nicht leer.
- post *c_pre* enthält *e* als sein Datenelement.
- f. *bool find(relative rel)*
- pre Der Baum ist nicht leer.
- post *current_node* ist in Abhängigkeit von *rel* wie folgt definiert: switch (*rel*)
- rootv: *current_node* ist die Wurzel, und *find* ist true.
- leftchild: if c_pre hat linken Sohn
 then *current_node* ist der linke Sohn, *find* ist true
 else *find* ist false.
- rightchild: if c_pre hat rechten Sohn
 then *current_node* ist der rechte Sohn, *find* ist true
 else *find* ist false.
- parent: if c_pre hat Vater
 then *current_node* ist der Vater, *find* ist true
 else *find* ist false.
- g. *struct status characteristics*
- post *characteristics* gibt die Anzahl der Knoten, die Höhe des Baumes und die durchschnittliche Pfadlänge von der Wurzel zu einem Blatt an.
- h. *bool empty*
- post if Baum nicht leer
 then *empty* ist false.
 else *empty* ist true.
- i. *create*
- post Ein leerer binärer Baum existiert.
- j. *clear*
- post Der binäre Baum ist leer.

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.

**Abbildung 3.24: Baumdarstellung mit Pointer**

	Element		left	right
	Y	0	-1	-1
		1		
root →	Wurzel	2	5	6
	Z	3	-1	-1
		4		
	V	5	0	3
	X	6	-1	-1
	.		.	.
	.		.	.
	.		.	.

Abbildung 3.25: Baumdarstellung mit statischen Feldern

Realisation eines Binärbaumes

1. Repräsentation

```

/* von aussen, d.h. von Anwendungsprogrammen, sichtbare */
/* Vereinbarungen: */
typedef enum {preorder, inorder, postorder} order;
typedef enum {rootv, leftchild, rightchild, parent} relative;

struct status {
    int    size;    /* number of nodes */
    int    height;
    double average_path_length;
};

/* Typdefinitionen: */
typedef struct node *node_pointer;

struct node {
    stdelement    element;
    node_pointer left;
    node_pointer right;
};

struct characteristics_data {
    struct status st;
    int total_path_length;
};

/* globale Variablen: */
static node_pointer root, current_node;

/* lokale Variablen: */
struct characteristics_data d; /* lokal zu characteristics */
. . .

```

2. Implementation

Externe Funktionen

- a. aus der Stack-Realisierung, siehe Kapitel 3.2.3 auf Seite 48:


```
extern void push (void *node_ptr);
extern void *pop (void);
```
- b. eine Funktion zur Bearbeitung der Knoten:


```
extern void process (stdelement *e_ptr, int level, void *data_ptr);
```

static node_pointer newnode(void)
--

p = (node_pointer) malloc(sizeof(struct node))
--

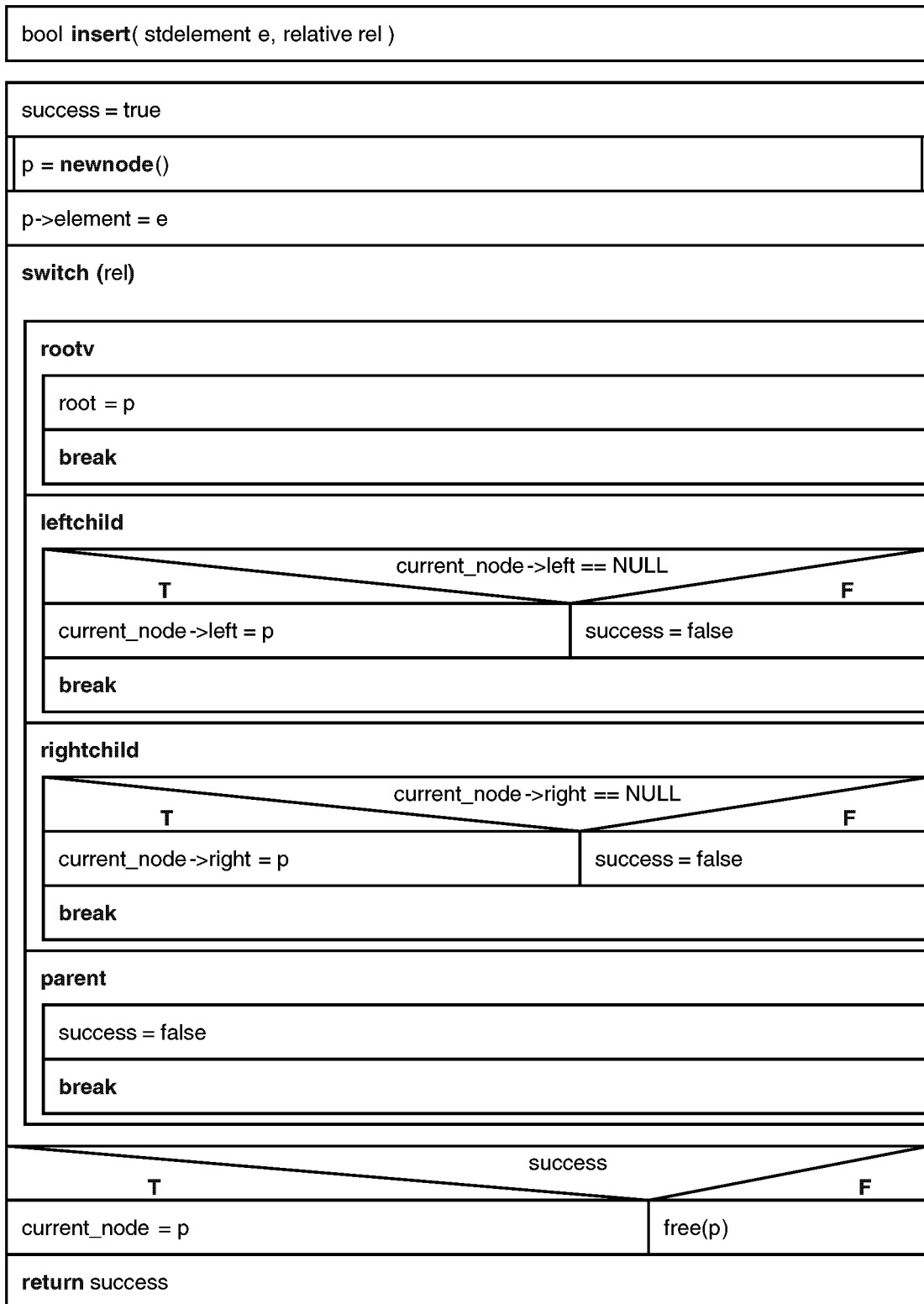
assert(p != NULL)

p->left = NULL

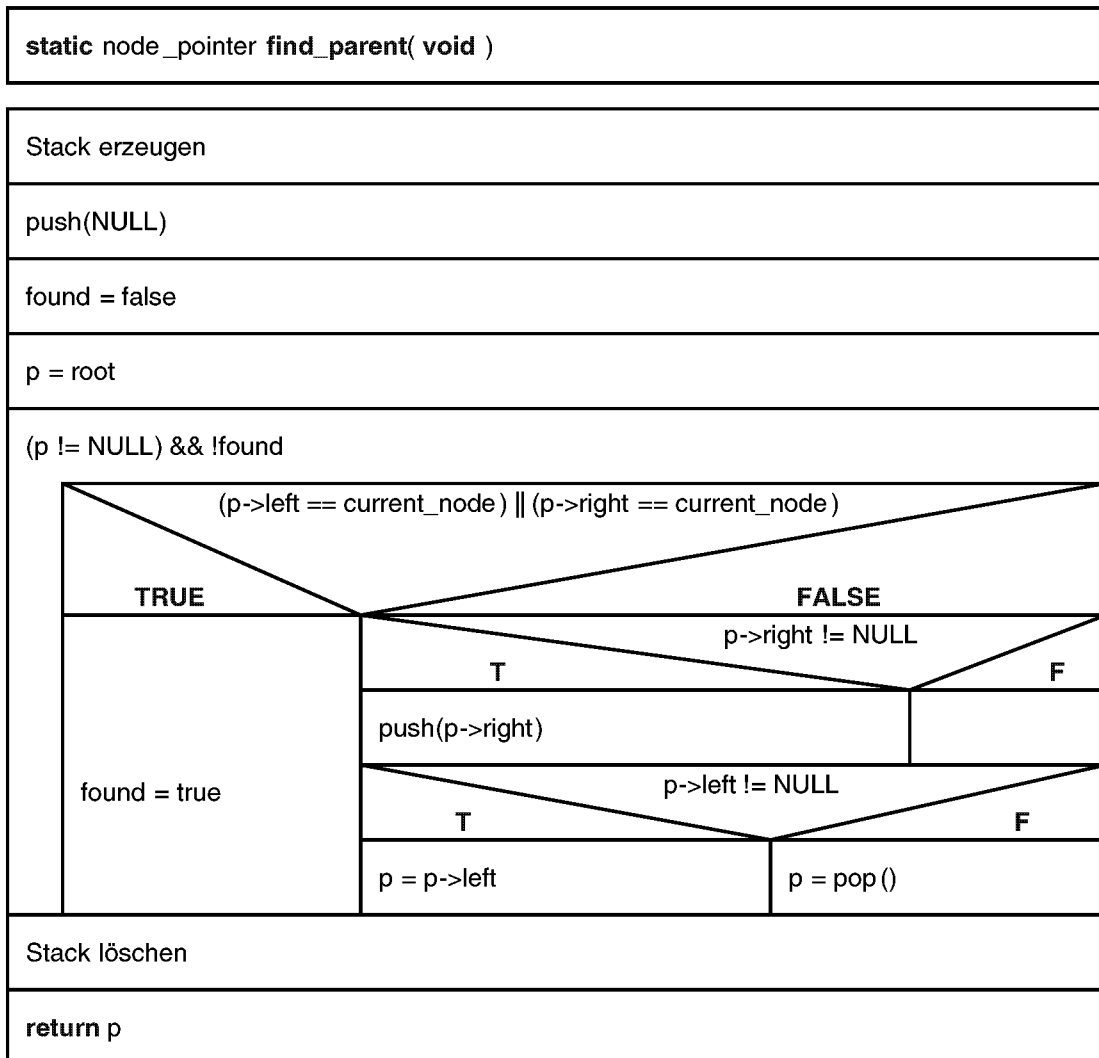
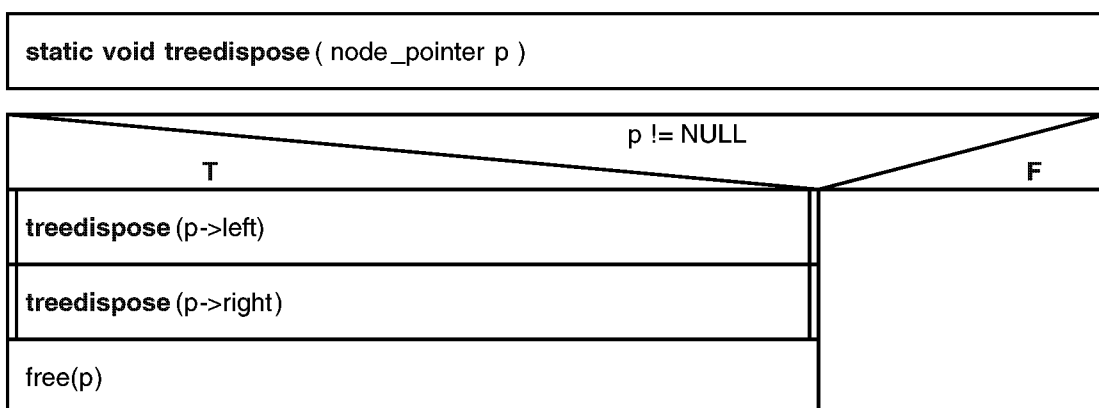
p->right = NULL

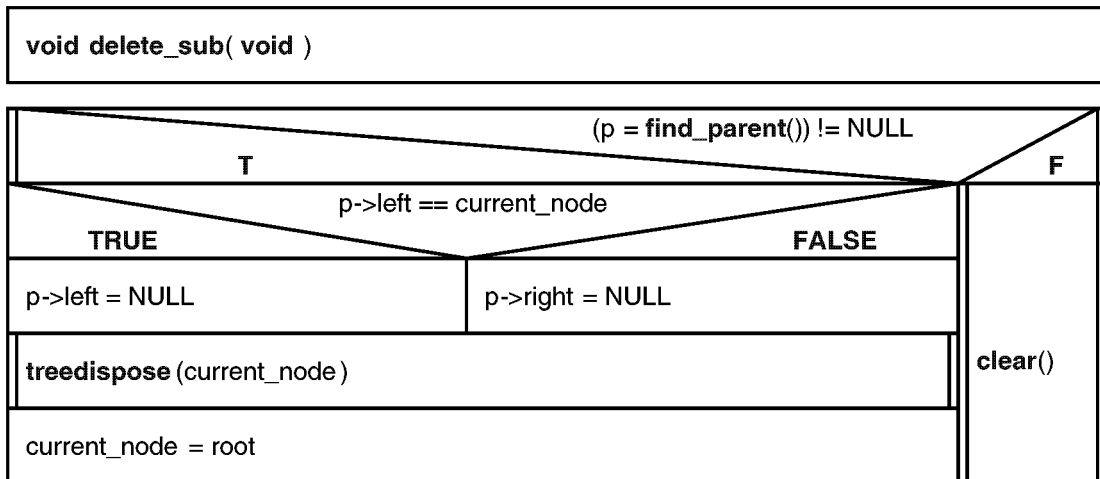
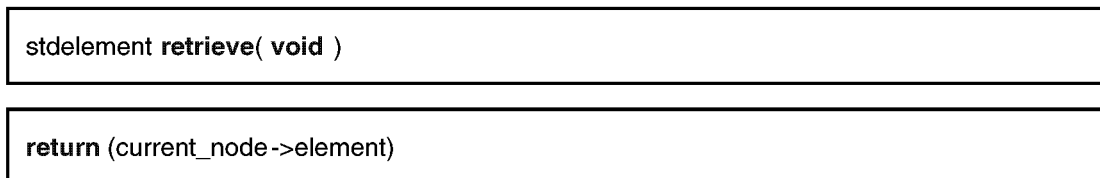
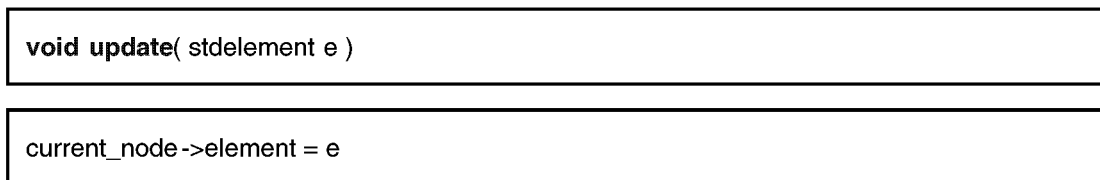
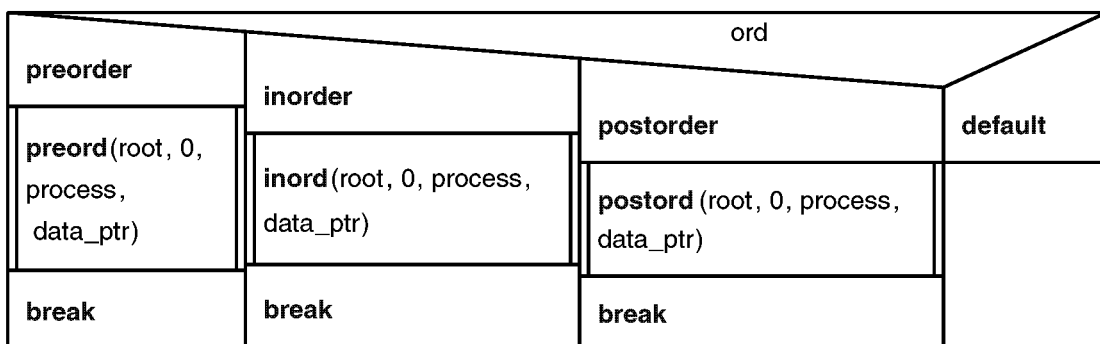
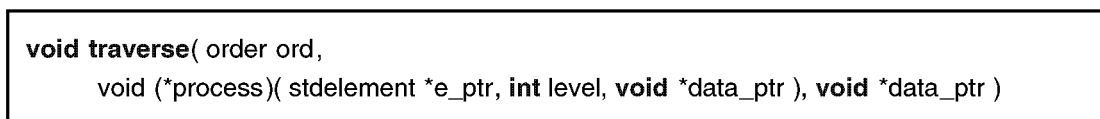
return p

Funktion newnode



Funktion insert

**Funktion find_parent****Funktion treedispense**

**Funktion delete_sub****Funktion retrieve****Funktion update****Funktion traverse**

```
static void preord( node_pointer p, int level,
    void (*process)( stdelement *e_ptr, int level, void *data_ptr ), void *data_ptr )
```

p != NULL	
T	F
process(&(p->element), level, data_ptr)	
preord(p->left, level+1, process, data_ptr)	
preord(p->right, level+1, process, data_ptr)	

Funktion preord

```
static void inord( node_pointer p, int level,
    void (*process)( stdelement *e_ptr, int level, void *data_ptr ), void *data_ptr )
```

p != NULL	
T	F
inord(p->left, level+1, process, data_ptr)	
process(&(p->element), level, data_ptr)	
inord(p->right, level+1, process, data_ptr)	

Funktion inord

```
static void postord ( node_pointer p, int level,
    void (*process)( stdelement *e_ptr, int level, void *data_ptr ), void *data_ptr )
```

p != NULL	
T	F
postord (p->left, level+1, process, data_ptr)	
postord (p->right, level+1, process, data_ptr)	
process(&(p->element), level, data_ptr)	

Funktion postord

struct status characteristics (void)	
d.st.size = 0	
d.st.height = 0	
d.total_path_length = 0	
!empty()	
TRUE	FALSE
traverse(inorder, process_characteristics, &d)	
d.st.average_path_length = (double) d.total_path_length / (double) d.st.size	d.st.average_path_length = 0.0
return (d.st)	

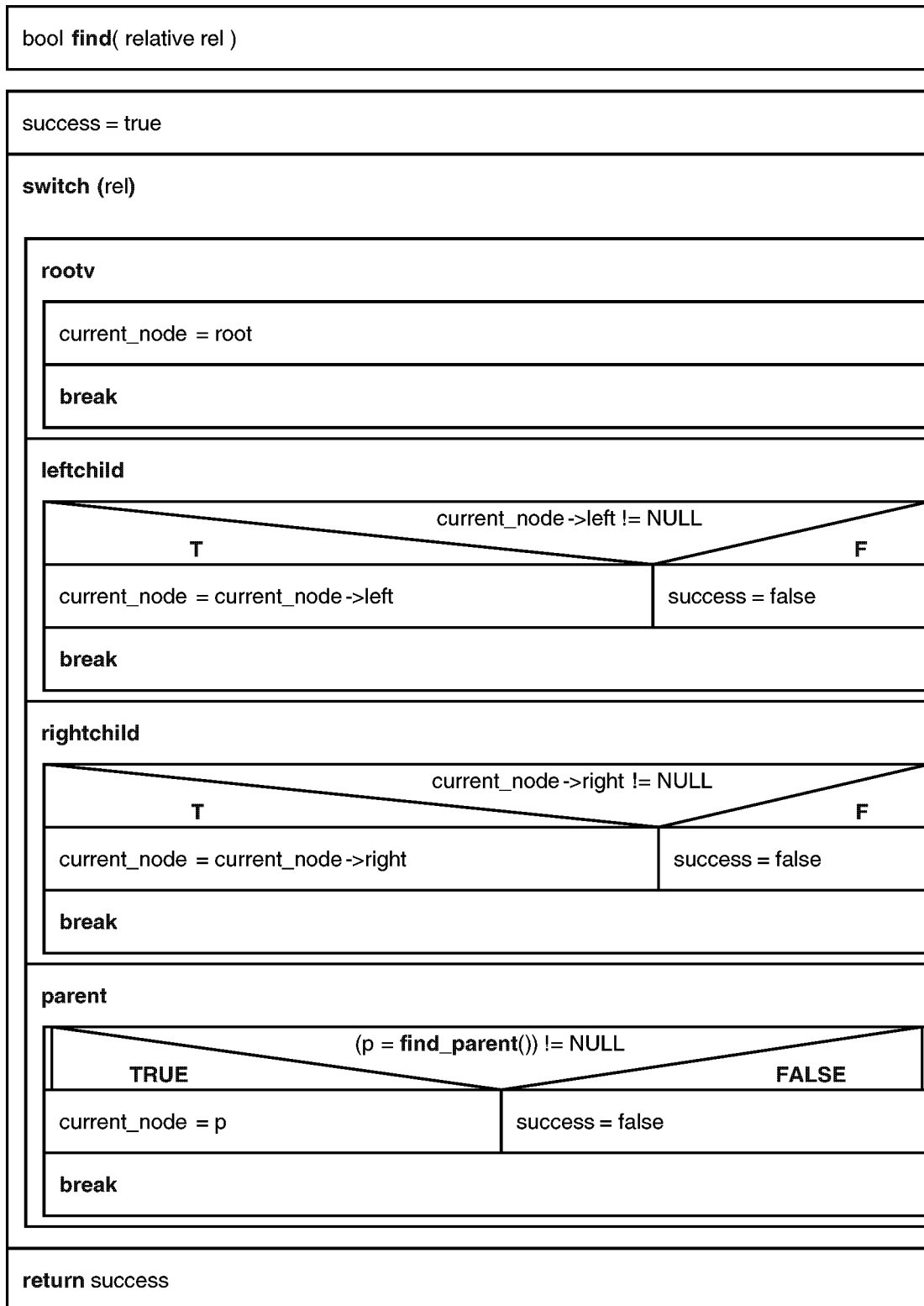
Funktion characteristics

static void process_characteristics (stdelement *e_ptr, int level, void *data_ptr)	
dp = (struct characteristics_data *) data_ptr	
dp->st.size++	
dp->total_path_length += level	
dp->st.height < level	
T	F
dp->st.height = level	

Funktion process_characteristics

bool empty(void)
return (root == NULL)

Funktion empty

**Funktion find**

```
void create( void )
```

```
root = NULL
```

```
current_node = NULL
```

Funktion create

```
void clear( void )
```

```
treedisperse (root)
```

```
create()
```

Funktion clear

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.

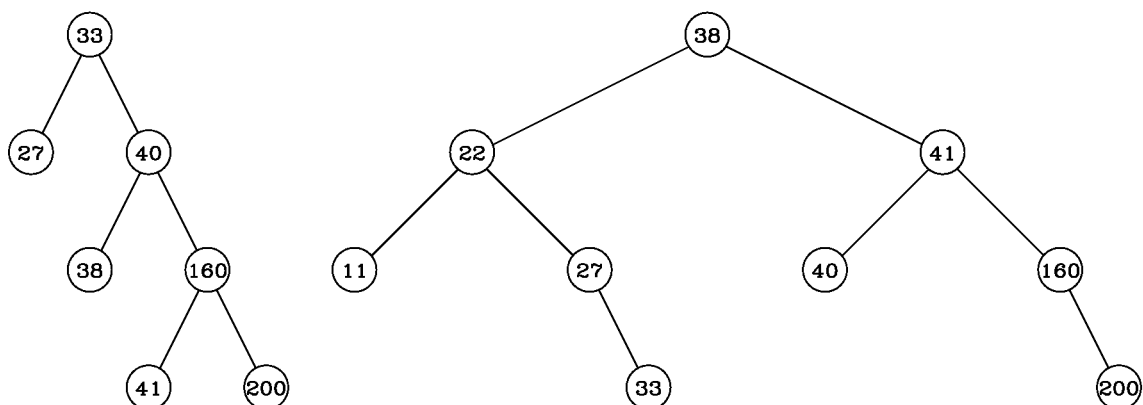


Abbildung 3.26: Beispiele für binäre Suchbäume

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 .
- Diese Eigenschaften gelten rekursiv, d.h. jeder Teilbaum ist wiederum ein binärer Suchbaum.

3. Operationen

Hier werden vier wichtige Operationen spezifiziert. Die restlichen noch notwendigen Operationen entnehmen Sie bitte der Implementation der Definition 3.21.

a. *bool insert (const stdelement e)*

```
pre   Keine
post  if      t_pre enthält e nicht
      then    e ist Element des Baumes, und insert ist true
      else    insert ist false
```

b. *bool delete_key (const stdelement e)*

```
post  if      t_pre enthält das Element e
      then    dieses Element ist nicht mehr im Baum enthalten, und delete_key ist
              true
      else    delete_key ist false
```

c. *update(const stdelement e)*

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

d. *bool findkey (const stdelement e)*

```
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 findkey ist true
      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 findkey
              ist false
```

Realisation eines binären Suchbaumes

1. Repräsentation

```

typedef enum {preorder, inorder, postorder} order;
typedef enum {rootv, leftchild, rightchild, parent} relative;

typedef struct node *node_pointer;

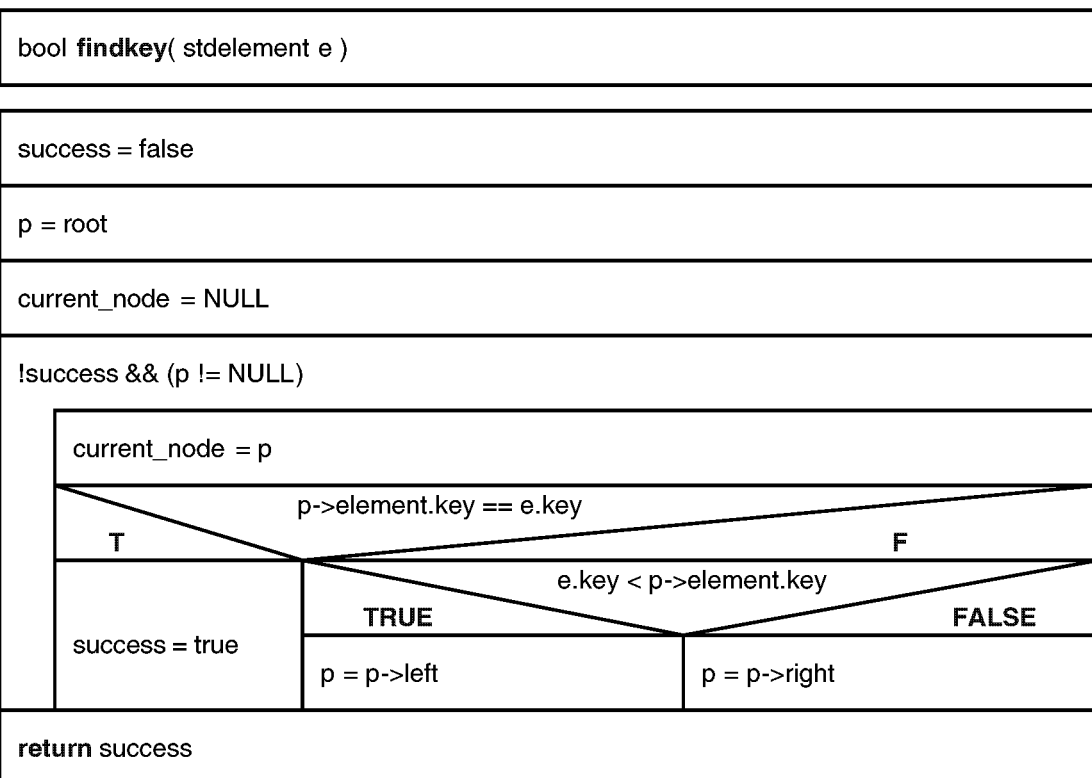
struct node {
    stdelement element;
    node_pointer left;
    node_pointer right;
};

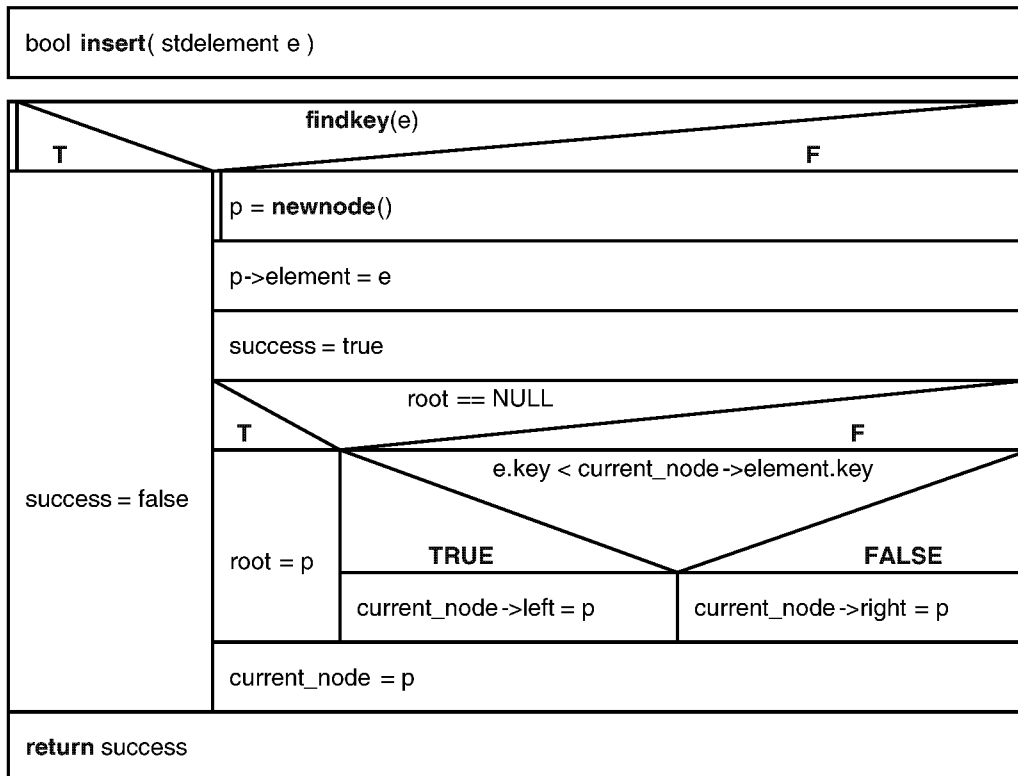
/* globale Variablen: */
static node_pointer root, current_node;

```

2. Implementation

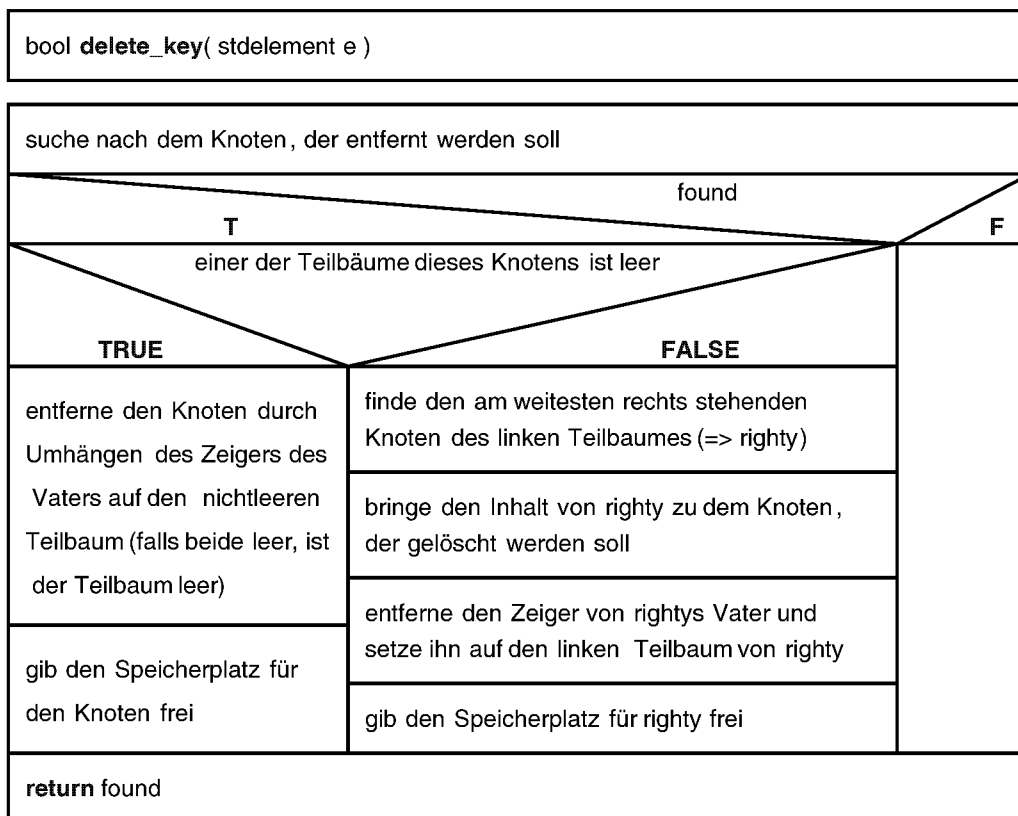
Hier werden nur die vier wichtigen Operationen findkey, insert, delete_key und update besprochen.

**Funktion findkey**



Funktion insert

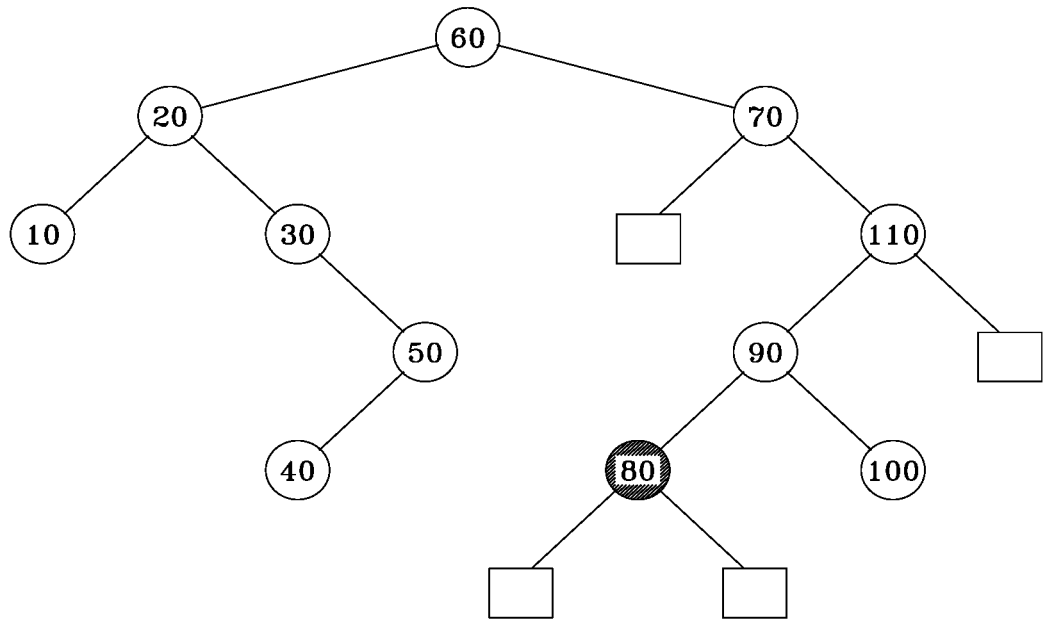
Die Operation delete_key ist eine Funktion mit komplexer Aufgabenstellung und soll deshalb zunächst durch eine Funktionsskizze veranschaulicht werden.



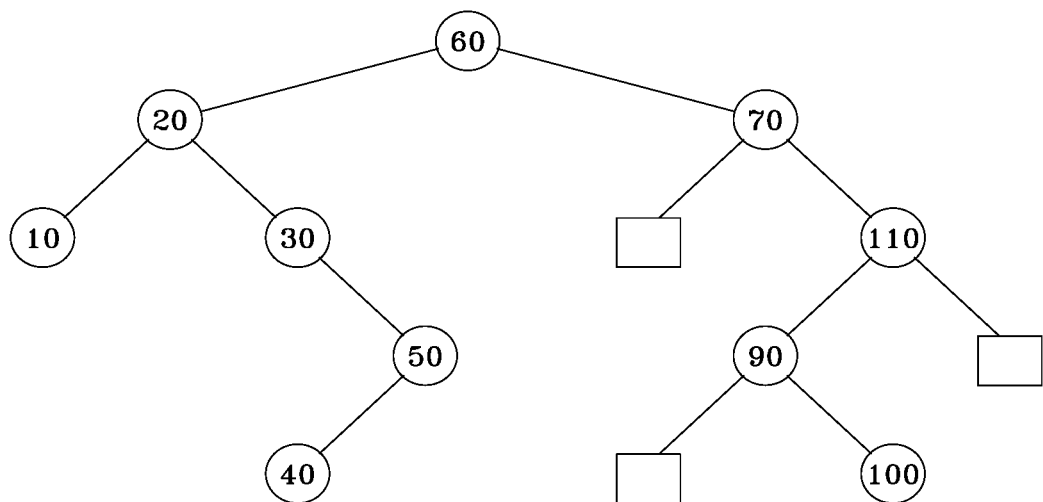
Funktion Funktionsskizze für die Operation delete_key

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.

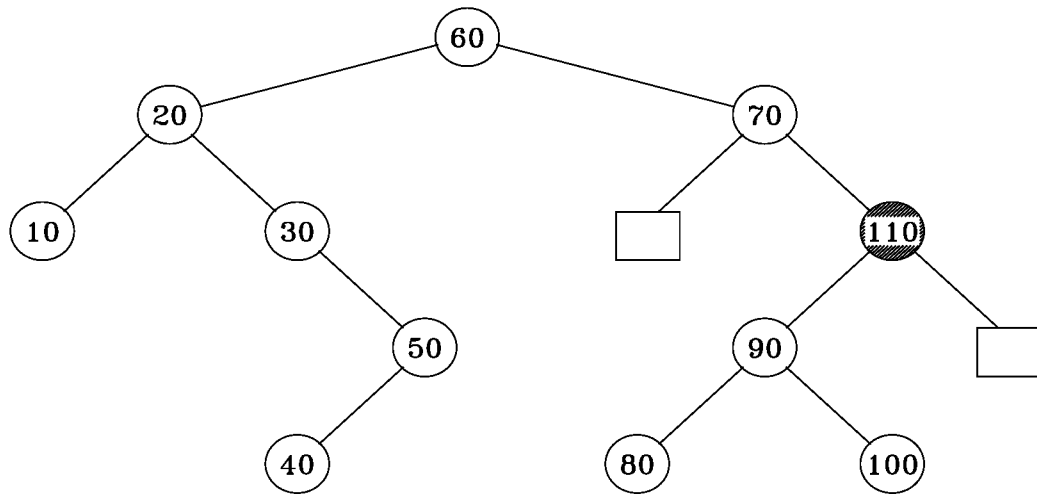


Darstellung 1a: Baum vor dem Löschen von 80

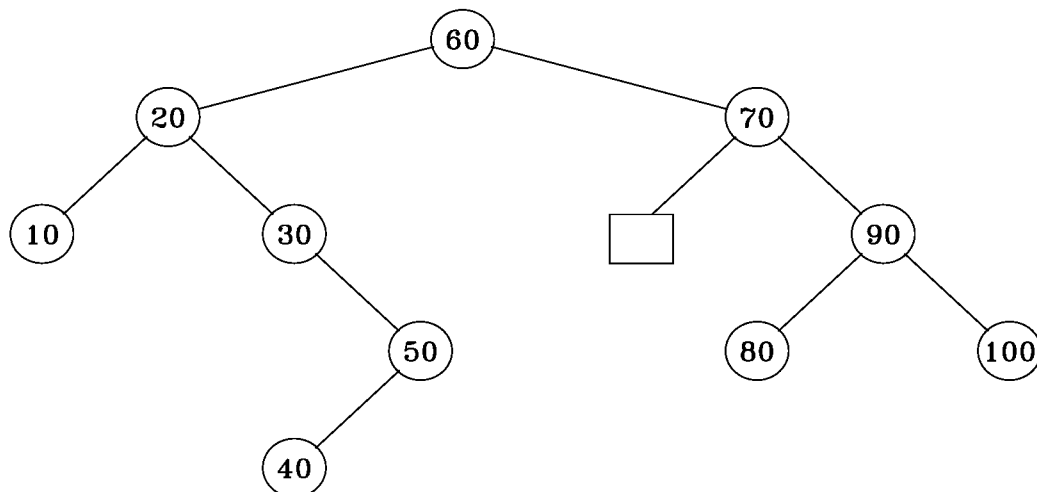


Darstellung 1b: Baum nach dem Löschen von 80

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.

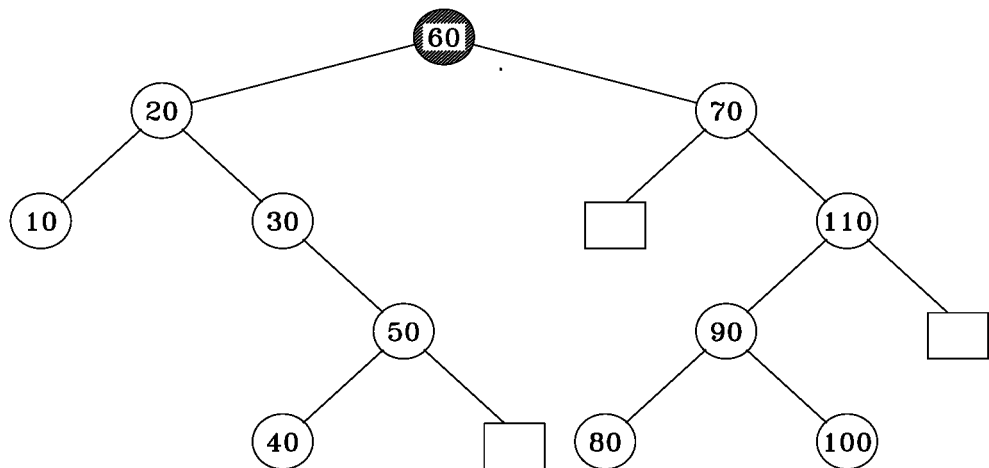


Darstellung 2a: Baum vor dem Löschen von 110

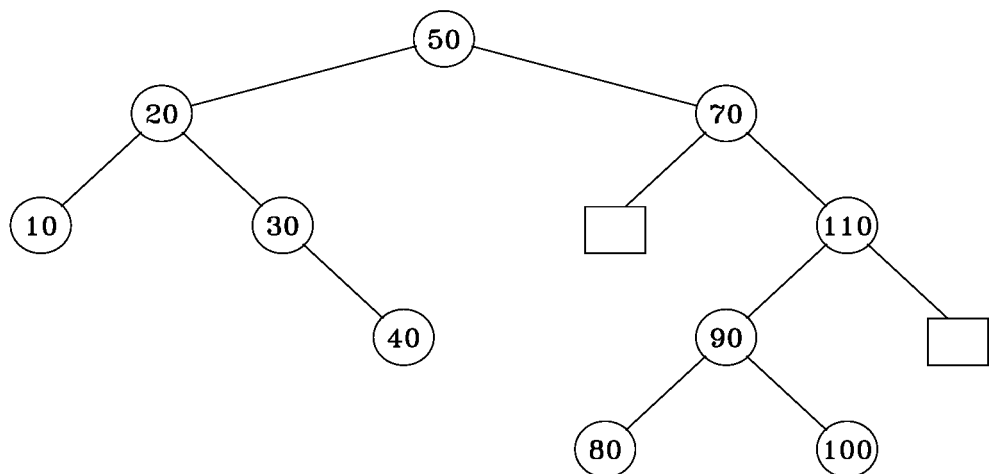


Darstellung 2b: Baum nach dem Löschen von 110

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 Funktion `delete.key` verwendet.

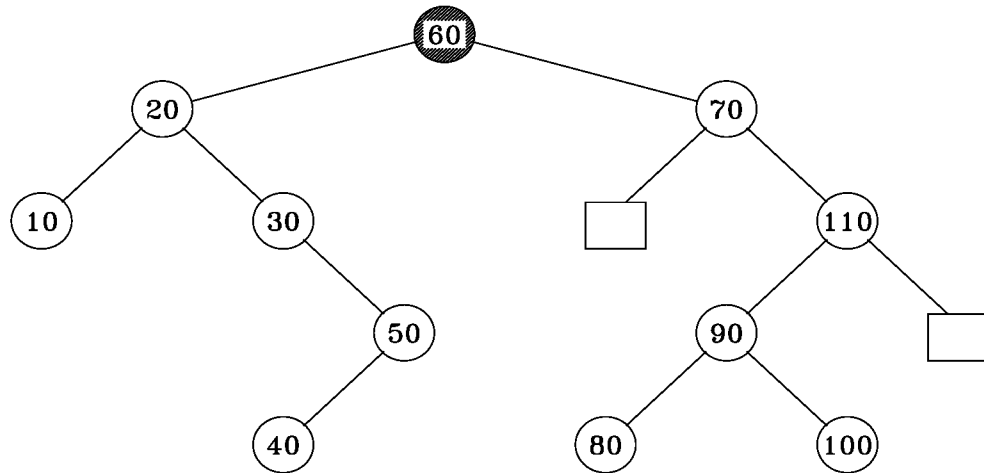


Darstellung 3a: Baum vor dem Löschen von 60

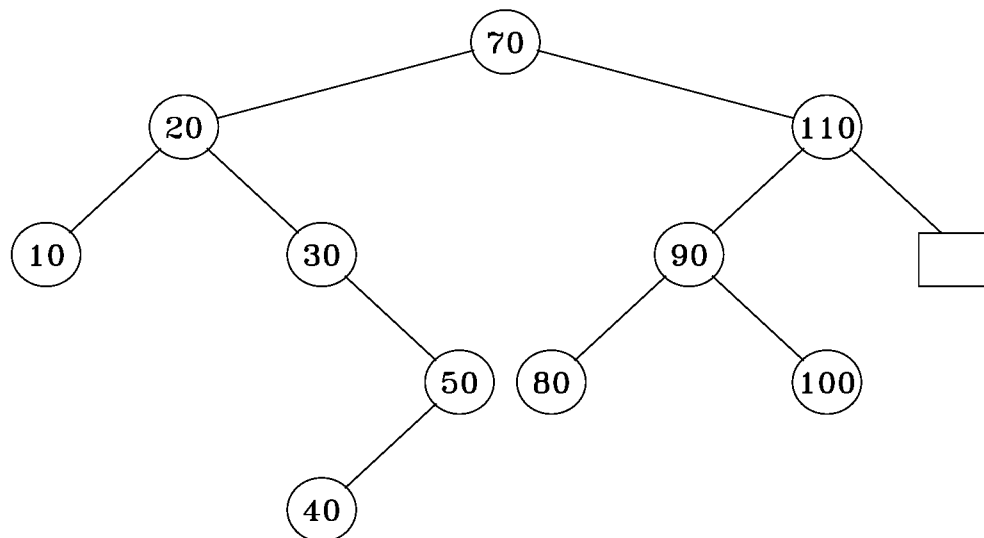


Darstellung 3b: Baum nach dem Löschen von 60

- 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.

bool delete_key (stdelement e)	
!empty()	
TRUE	FALSE
success = del (e, &root)	success = false
current_node = root	
return success	

Funktion delete_key

static bool del (stdelement e, node_pointer *addr_of_nodeptr)	
p = *addr_of_nodeptr	
p == NULL	
T	F
e.key < p->element.key	
T	F
e.key > p->element.key	
T	F
success	remove = p
	success = true
	p->right == NULL
	T
	p->left == NULL
	TRUE
	FALSE
	*addr_of_nodeptr = p->right
	subdel (&(p->left), &remove)
	free (remove)
return success	

Funktion del (interne Funktion zur Implementierung von delete_key)

static void subdel(node_pointer *addr_of_nodeptr , node_pointer *addr_of_remove)	
q = *addr_of_nodeptr	
q->right != NULL	
TRUE	FALSE
subdel(&(q->right), addr_of_remove)	(*addr_of_remove)->element = q->element
	*addr_of_remove = q
	*addr_of_nodeptr = q->left

Funktion subdel (interne Funktion zur Implementierung von delete_key)

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

void update(stdelement e)
delete_key(current_node->element)
insert(e)

Funktion 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. 3.27 auf der nächsten Seite).

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. 3.27 auf der nächsten Seite).

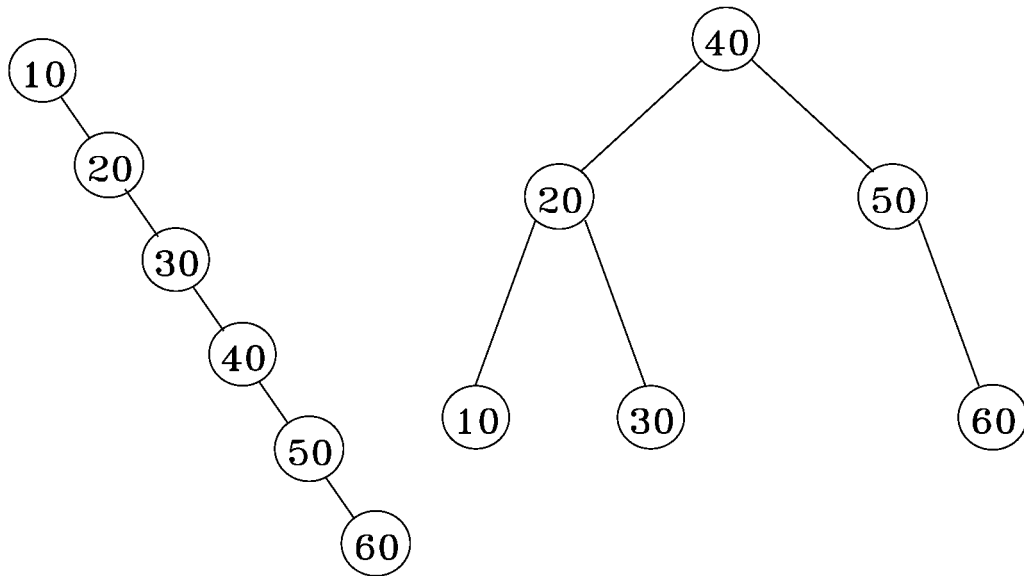


Abbildung 3.27: Unterschiedliche Suchbäume mit gleichem Informationsgehalt

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:

$$n = 2^{k+1} - 1 \Rightarrow k = \log(n + 1) - 1.$$

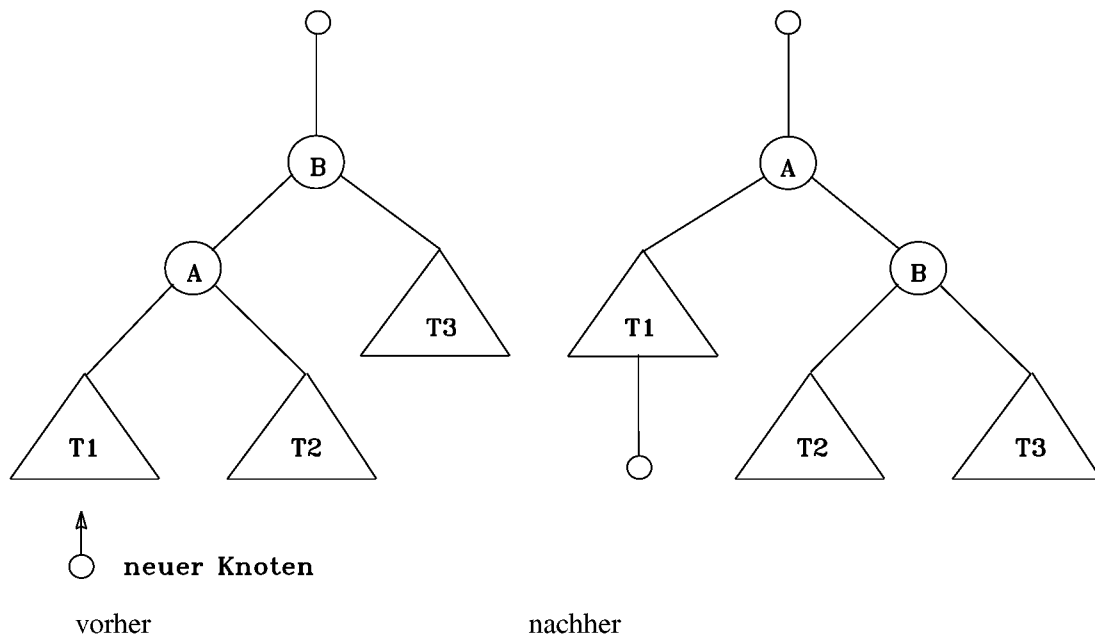
Sei die Suchlänge zu einem Knoten die Anzahl von Knoten, die bei der Suche nach diesem Knoten in einem binären Suchbaum inspiziert werden muß, bis der Knoten gefunden ist:

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 auf der nächsten Seite 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 Knotens 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.

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

3.4.3.1 Realisierung eines B-Baumes

1. Repräsentation

```
#define ORDER ...          /* Ordnung des B-Baumes */
#define MAXNODE (2*ORDER) /* max. Anzahl Elemente in einem Knoten */

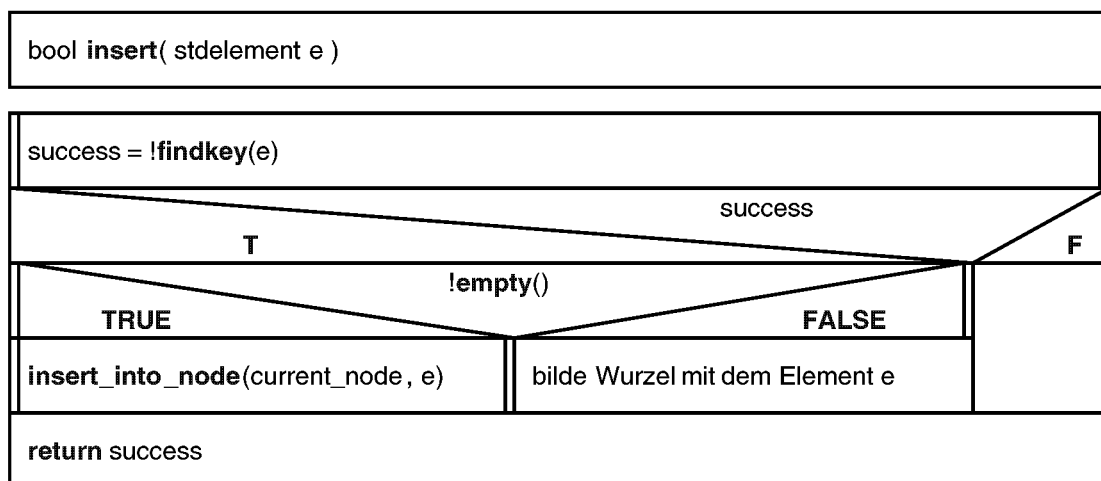
typedef struct node *node_pointer;

struct node {
    node_pointer parent;
    short        number_of_elements; /* benoetigter Wertebereich: */
                                   /*      0..MAXNODE           */
    stdelement  elements[MAXNODE];
    node_pointer children[MAXNODE+1];
};

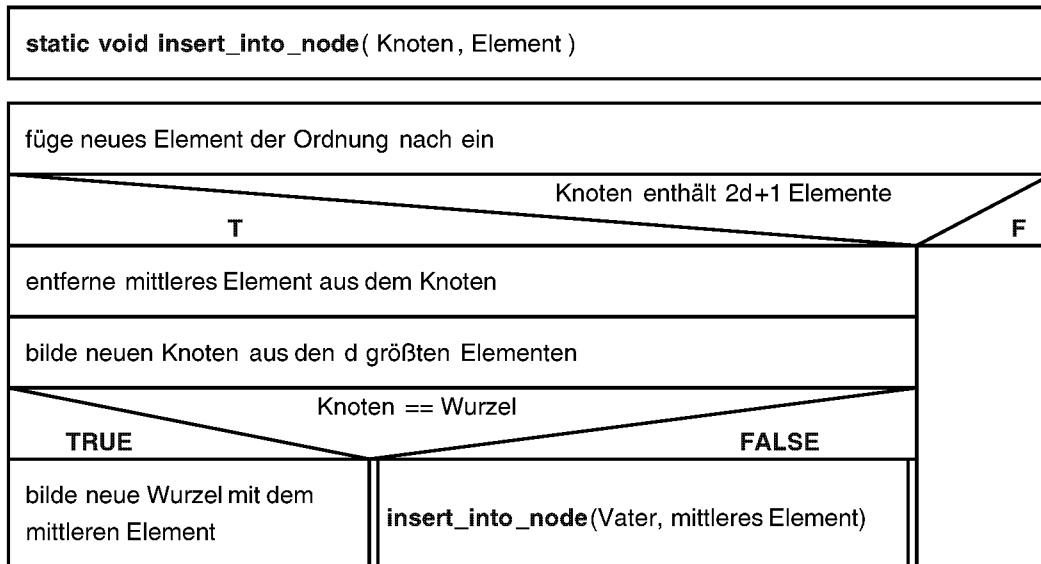
/* globale Variablen: */
static node_pointer root, current_node;
```

2. Implementation

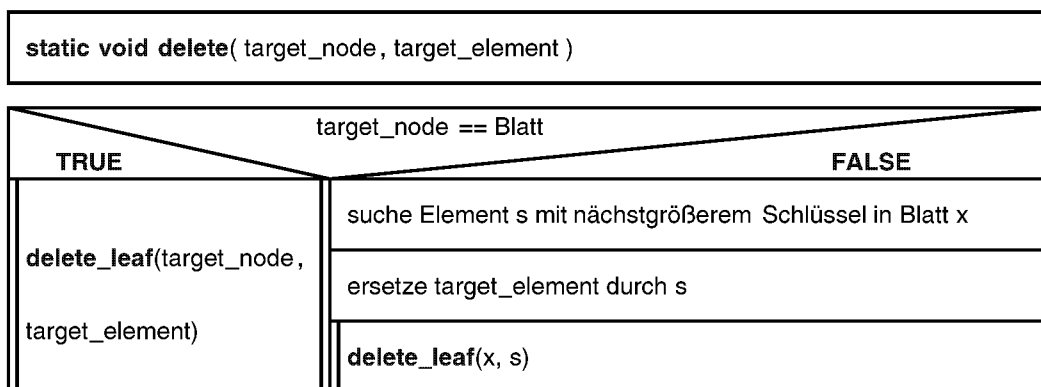
Die genaue Implementation eines B-Baumes kann in [12] nachgelesen werden. Hier sollen nur die beiden wichtigen Operationen insert und delete veranschaulicht werden.



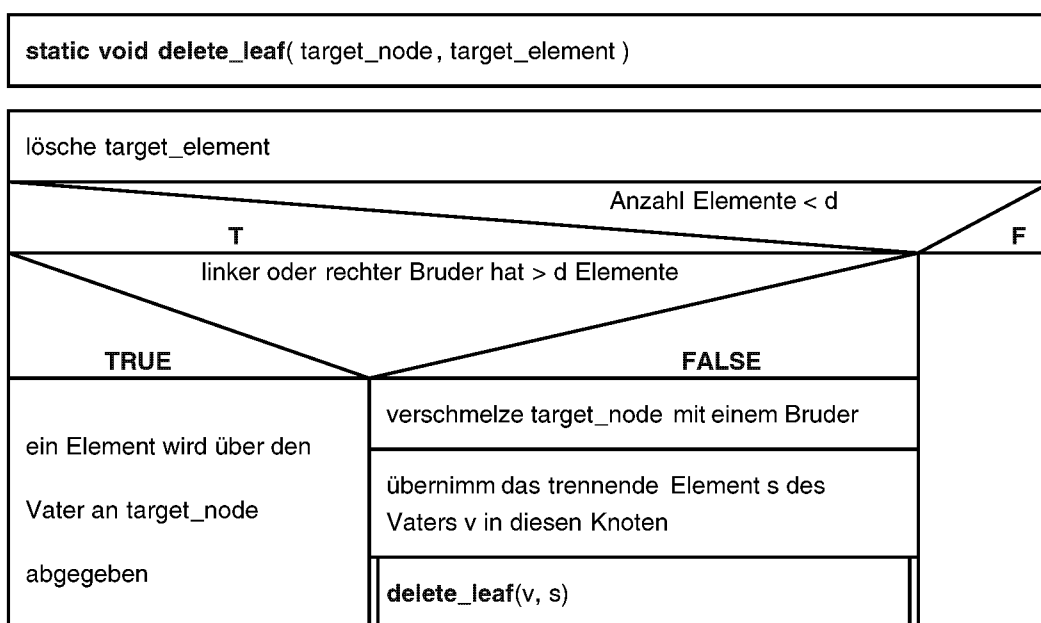
Funktion insert



Funktion insert_into_node (interne Funktion zur Implementierung von insert)



Funktion delete (interne Funktion zur Implementierung von delete_key)



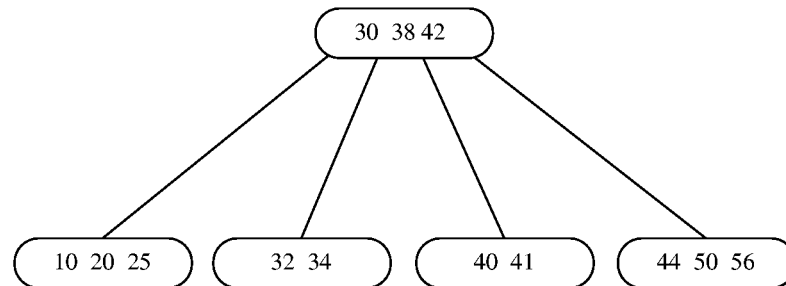
Funktion delete_leaf (interne Funktion zur Implementierung von delete_key)

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

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

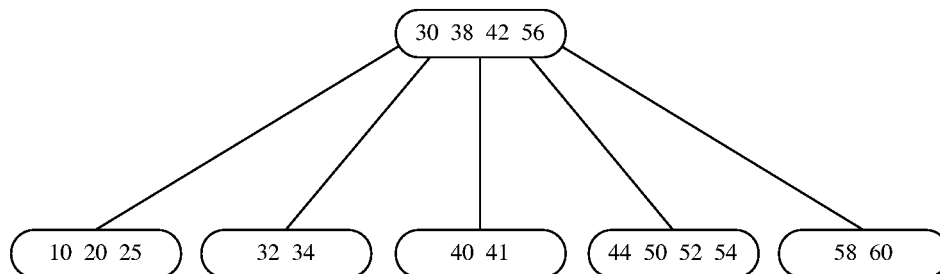
Dieses Beispiel soll die Operationen Einfügen und Löschen von Elementen in einem B-Baum veranschaulichen.

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



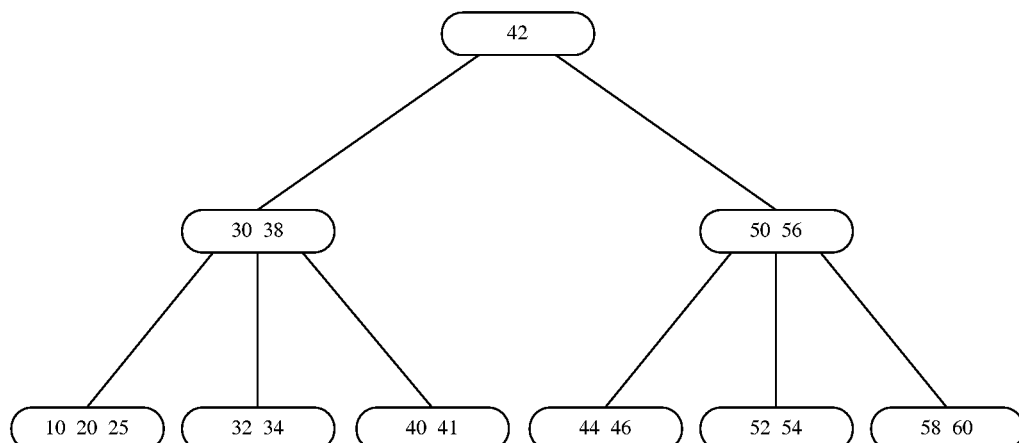
Einfügen von 58, 60, 52 und 54:

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:

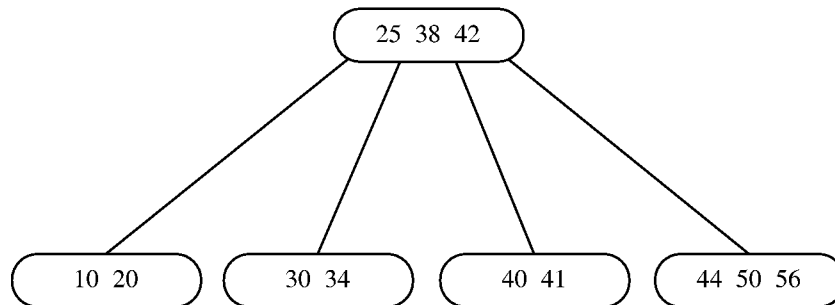
Wenn nun ein Element zwischen 42 und 56 eingefügt werden soll, muß der Knoten aufgeteilt und das mittlere Element aus dem vierten Blatt an den Vaterknoten abgegeben werden. Dies ist beispielsweise beim Einfügen von 46 das Element 50. Da der Vaterknoten, in diesem Fall die Wurzel des Baumes, die maximale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

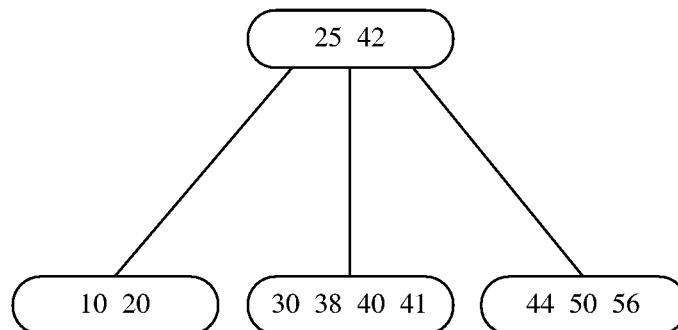
Löschen von 32:

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

$$a[i] \leq a[2 \cdot i]$$

$$a[i] \leq a[2 \cdot i + 1]$$

für alle zugehörigen Indizes.

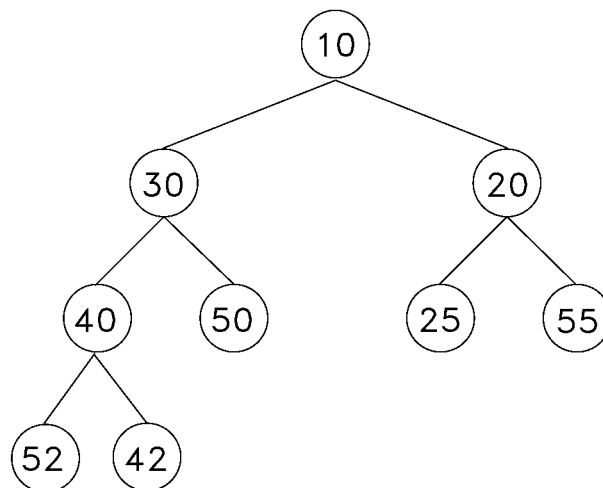
Aus dieser Definition folgt, daß sich das kleinste Element an der Wurzel des Heap befindet.

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* (*int n*)

pre vollständiger Binärbaum mit *n* Elementen

post Heap mit *n* Elementen

b. *siftdown* (*int q, int n*)

pre $a[q+1], \dots, a[n]$ erfüllt die Heap-Bedingung.

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

3.5.0.2 Realisierung eines Heap

1. Repräsentation

```
typedef struct {
    keytype key;
    datatype data;
} stdelement;

extern stdelement a[]; /* Vektor mit n+1 Elementen a[0], ..., */
                        /* a[n], von denen das Element a[0] */
                        /* nicht benutzt wird. */

/* lokale Variablen: */
int      insert, i, j;
stdelement x;
bool     finished;
```

2. Implementation:

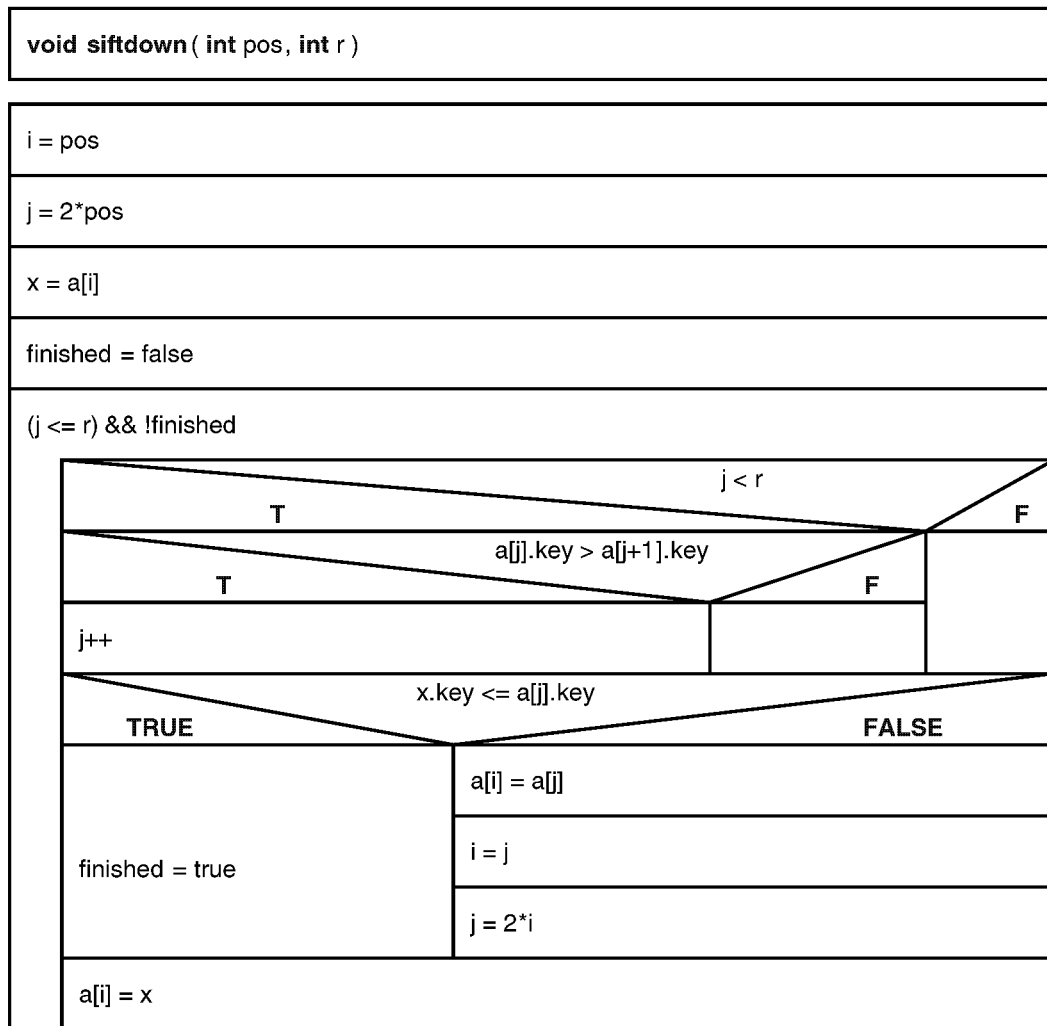
```
void heapcreate( int k )
```

```
    insert = k/2 + 1
```

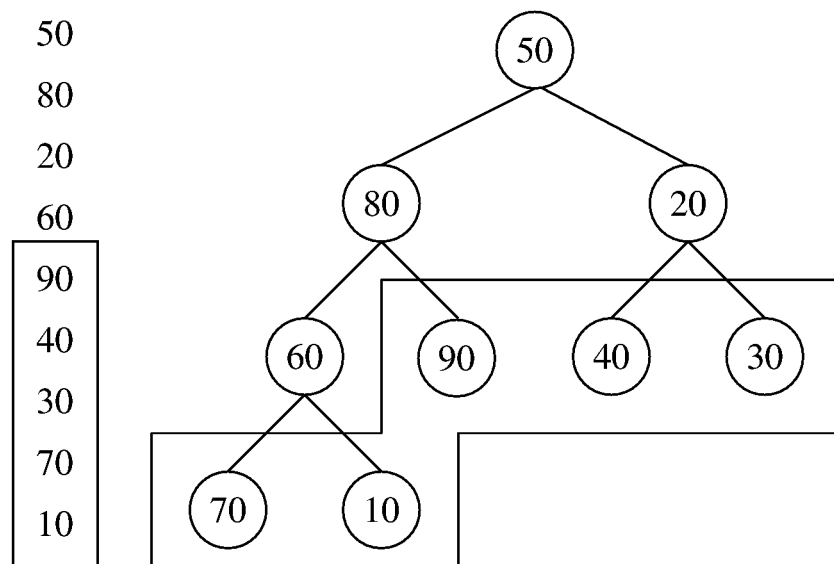
```
    while ( insert > 1
```

```
        siftdown(--insert, k)
```

Funktion heapcreate

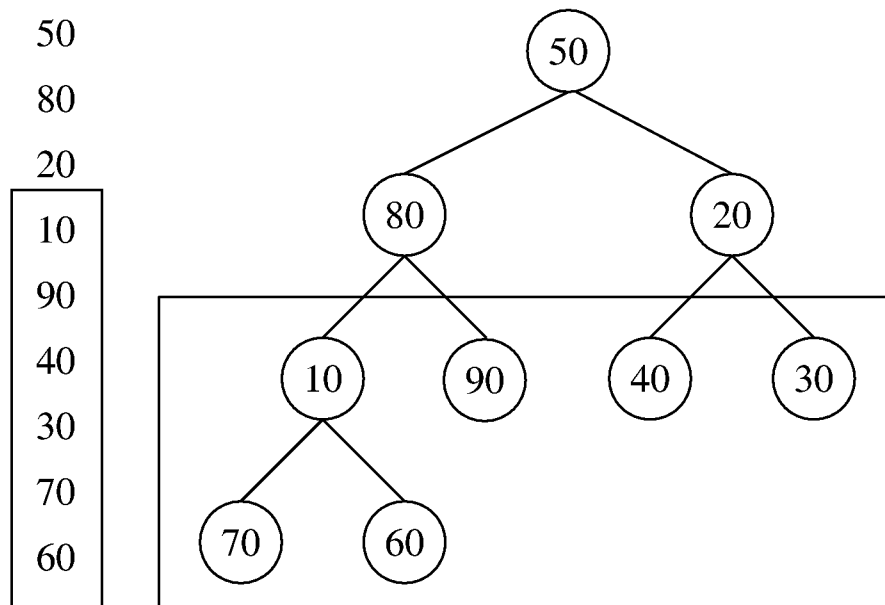
**Funktion siftdown****Beispiel 3.25** *Bilden eines Heap*

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



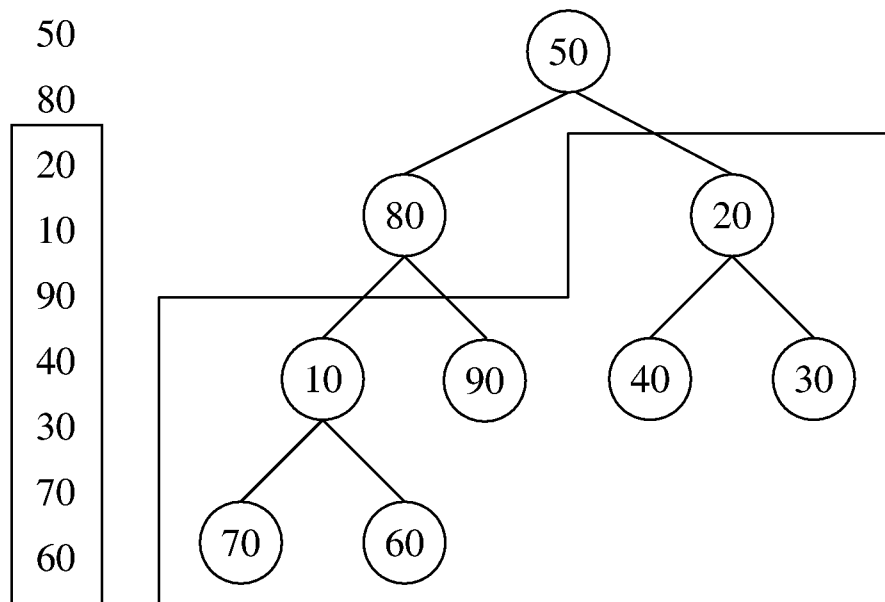
1. Operation: Einfügen von 60

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



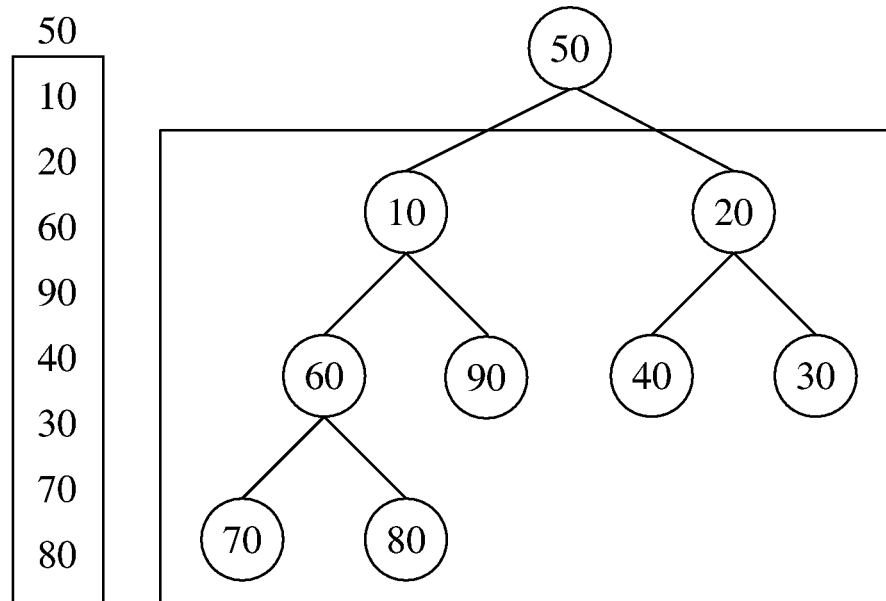
2. Operation: Einfügen von 20

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



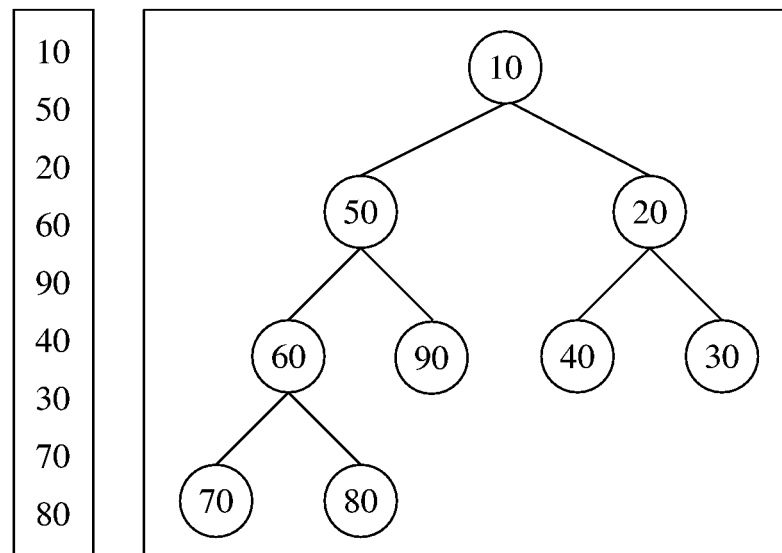
3. Operation: Einfügen von 80

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. 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

3.5.0.3 Zeitkomplexität:

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

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

Kapitel 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. Darüberhinaus wird vorausgesetzt, daß die Listenelemente von 1 an durchnummeriert sind. (Dies steht im Gegensatz zu der Numerierung von Feldelementen in C, die bei 0 beginnt. Daher sind die im folgenden angegebenen Implementationen bei der Integration in praktische Problemlösungen u.U. geringfügig zu modifizieren. Falls anstelle der Elemente $a[1], \dots, a[n]$ die Elemente $a[0], \dots, a[n-1]$ angesprochen werden sollen, muß bei jedem Zugriff auf ein Feldelement der Index um eins vermindert werden.)

4.1 Suchverfahren

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

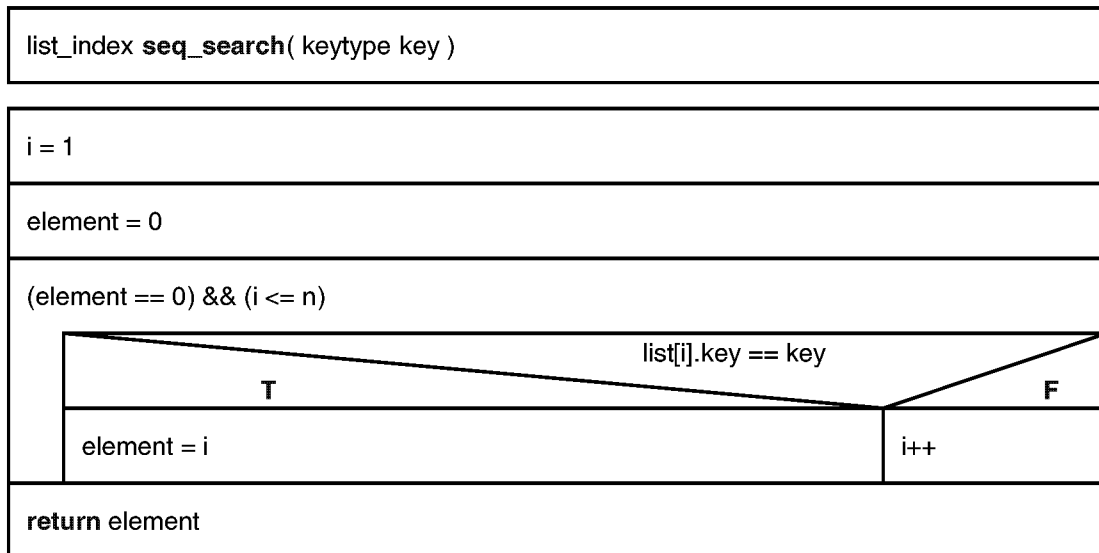
4.1.1 Lineares Suchen (Sequential Search)

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.



Funktion seq_search

4.1.2 Binäres Suchen (Binary Search)

Definition 4.1 Sortierte Liste

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

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

Sortierverfahren, 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
 1. kleiner
 2. gleich
 3. größer
 ist 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.

$$\begin{aligned}
 1. \quad & u = 1 \\
 & o = 9 \\
 & \left\lfloor \frac{u + o}{2} \right\rfloor = 5
 \end{aligned}$$

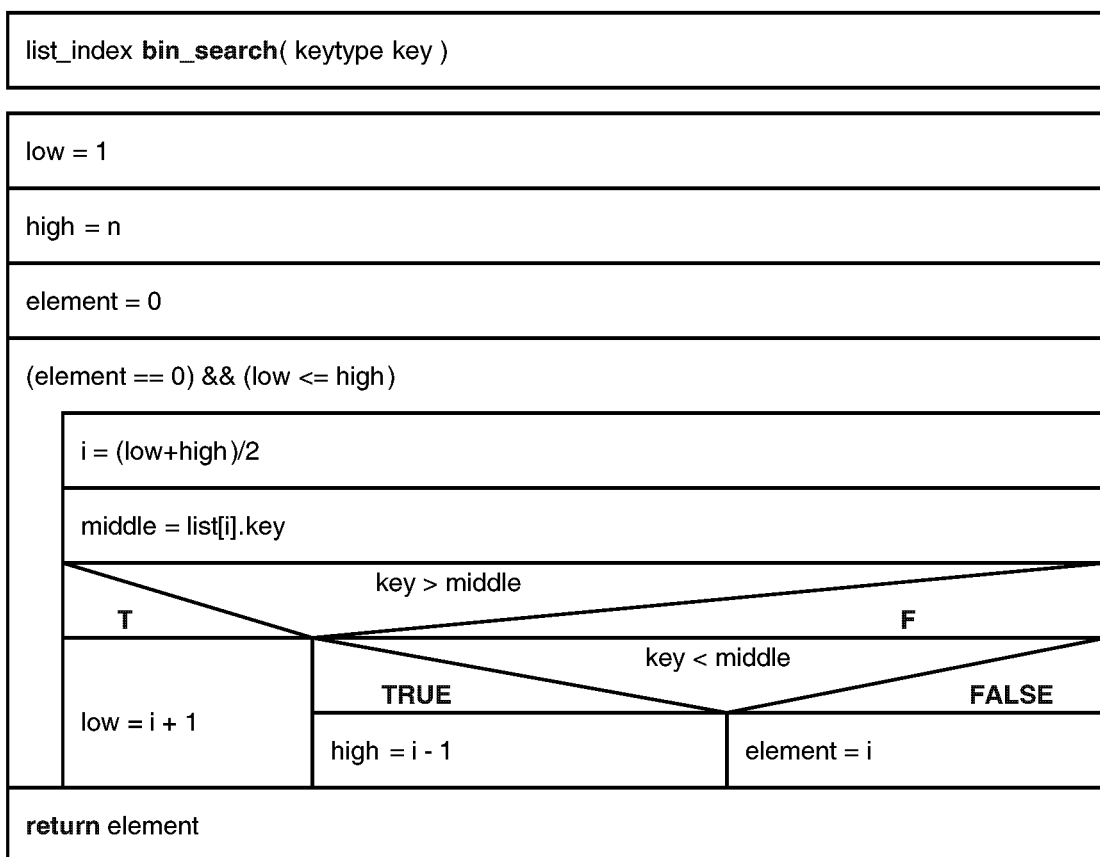
Feld[5].Schlüssel = 38 → linkes Teilfeld weiter betrachten

$$\begin{aligned}
 2. \quad & u = 1 \\
 & o = 4 \\
 & \left\lfloor \frac{u + o}{2} \right\rfloor = 2
 \end{aligned}$$

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

$$\begin{aligned}
 3. \quad & u = 3 \\
 & o = 4 \\
 & \left\lfloor \frac{u + o}{2} \right\rfloor = 3
 \end{aligned}$$

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



Funktion bin_search

4.1.2.1 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.

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

Das Suchverfahren kann durch einen binären Suchbaum veranschaulicht werden (s. Abb. 4.1 auf der nächsten Seite).

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.

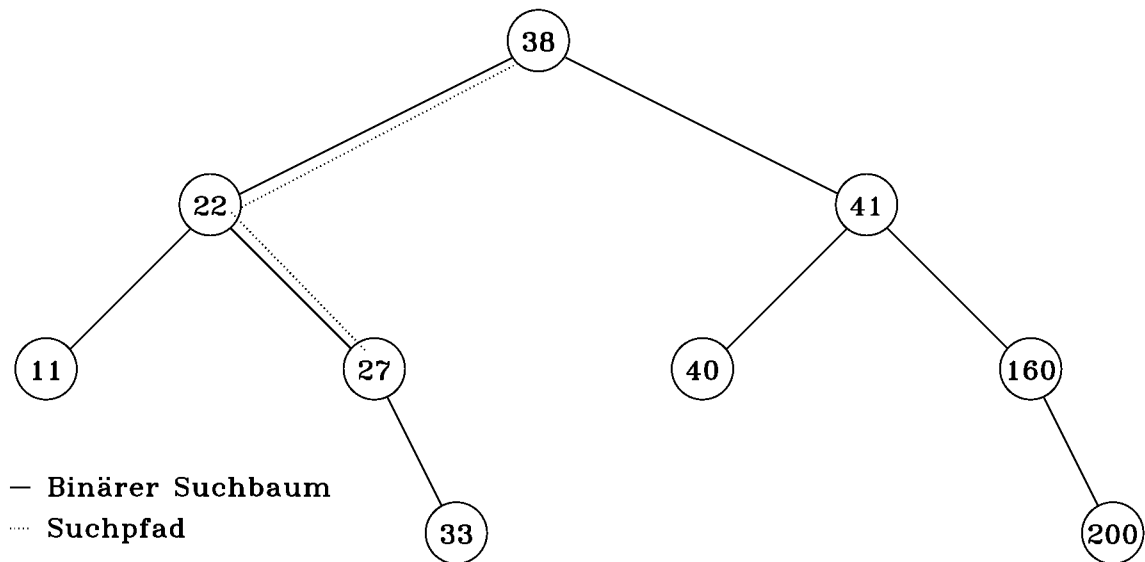


Abbildung 4.1: Binärer Suchbaum und Suchpfad zu Element 27

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 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 4.2 Sortierpermutation

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

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

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. $IPERM[1], \dots, IPERM[N]$).
- Die Adressierung des Feldes A geschieht immer über $A[IPERM[I]]$.
- 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 $IPERM[I] = I \ \forall I = 1, \dots, N$.

Definition 4.3 *Stabiles Sortierverfahren*

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.

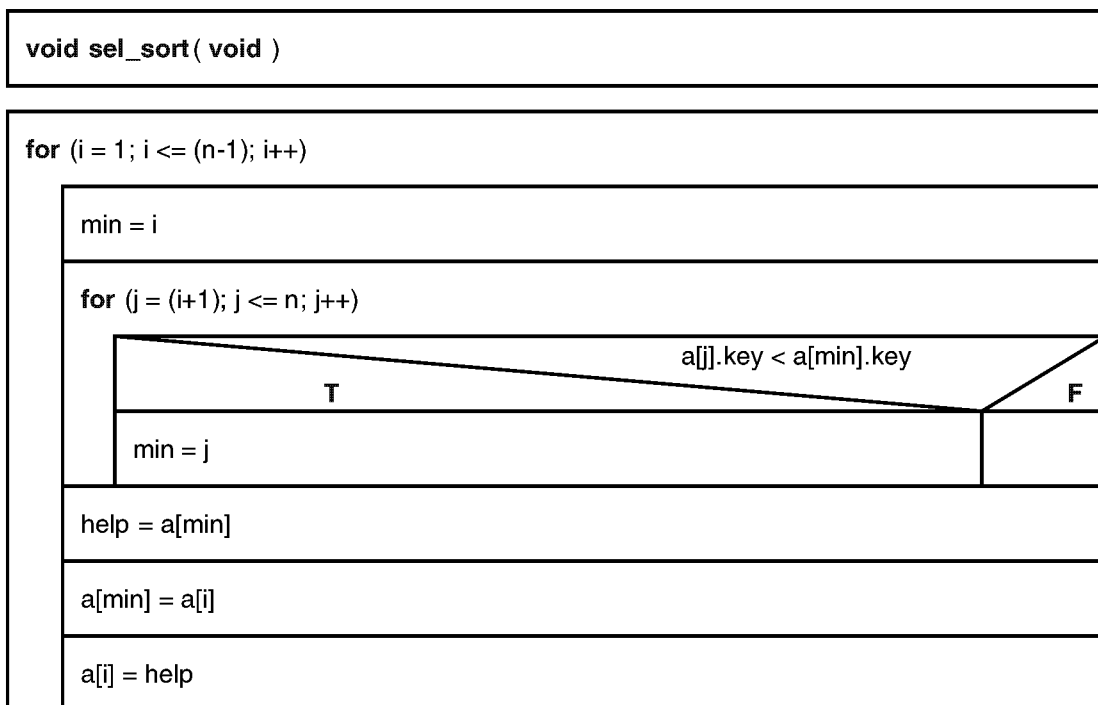
Von den beschriebenen Sortierverfahren sind nur das Sortieren durch Einfügen und der Bubblesort-Algorithmus stabile Verfahren.

4.2.2 Sortieren durch Auswahl (Selection Sort)

Prinzipielle Vorgehensweise:

for ($i = 1$; $i \leq (n-1)$; $i++$) 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 $a[min]$



Funktion 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.

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

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:

void sel_sort_permutation(void)	
for (i = 1; i <= n; i++)	
s[i] = i	
for (i = 1; i <= (n-1); i++)	
min = i	
for (j = (i+1); j <= n; j++)	
T a[s[j]].key < a[s[min]].key F	
min = j	
help = s[min]	
s[min] = s[i]	
s[i] = help	

Funktion 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; i <= n; i++) führe durch

1. x = a[i]
2. Füge x am entsprechenden Platz in a[1] ... a[i] ein

```
void ins_sort( void )
```

```
for (i = 2; i <= n; i++)
```

```
    x = a[i]
```

```
    a[0] = x
```

```
    for (j = (i-1); a[j].key > x.key; j--)
```

```
        a[j+1] = a[j]
```

```
    a[j+1] = x
```

Funktion 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 und mindestens 1 und somit, unter der Annahme, daß alle n! Permutationen gleich wahrscheinlich sind, im Mittel $\frac{i+1}{2}$.

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

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.

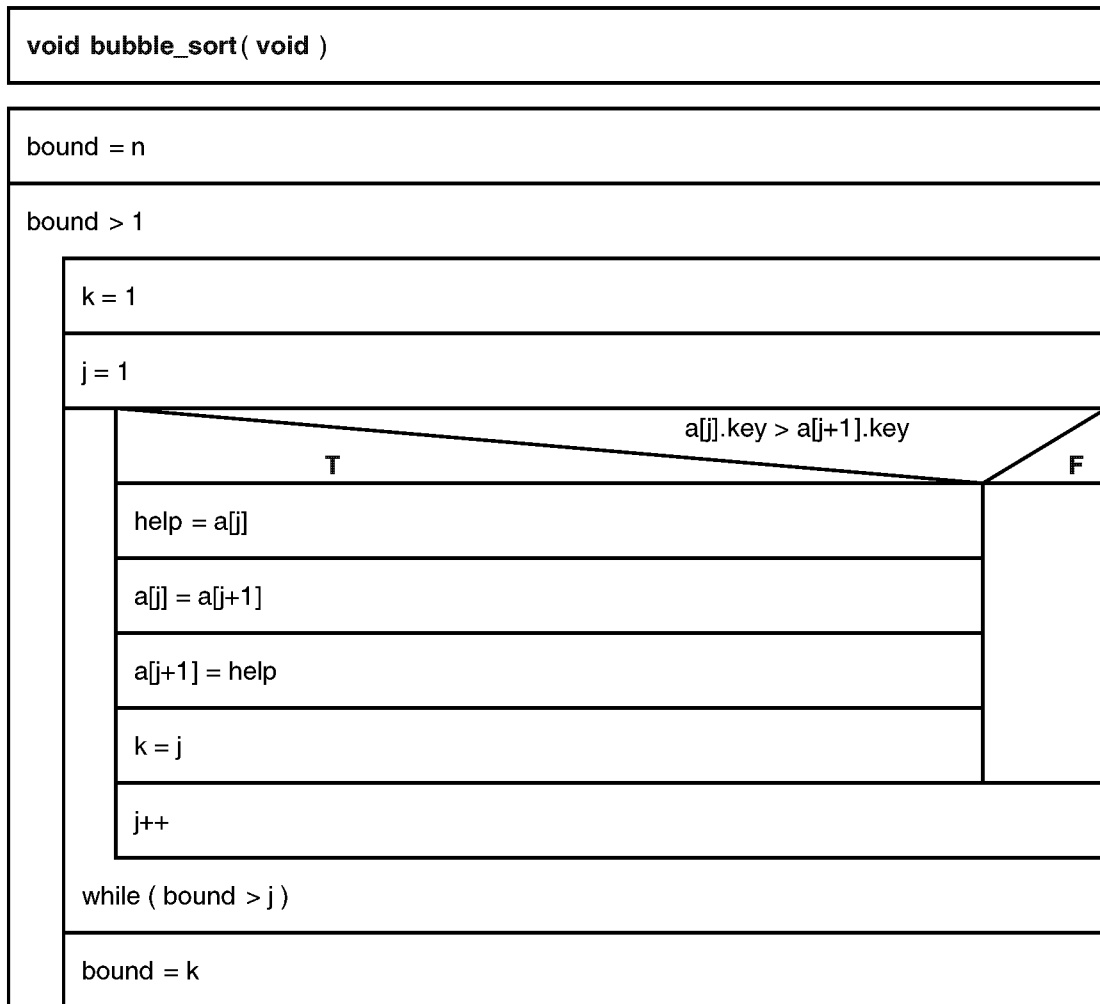
Nachteile des Algorithmus *Sortieren durch Einfügen*:

- 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.

**Funktion 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:

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

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

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

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:

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

Vorteile des Algorithmus *Sortieren durch Vertauschen*:

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

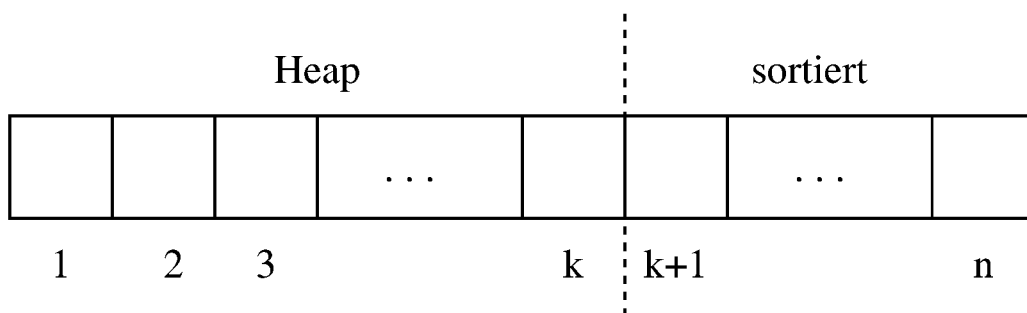
Nachteile des Algorithmus *Sortieren durch Vertauschen*:

- 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 auf Seite 95 gebildet werden. Man beachte, daß das kleinste Element das erste Element ist. Die Elemente sind in $a[1], a[2], \dots, a[n]$ gespeichert.



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

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

void heapsort(void)	
heapcreate(n)	
k = n	
k > 1	
temp = a[1]	
a[1] = a[k]	
a[k] = temp	
siftdown (1, --k)	

Funktion 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 Funktion für das Einfügen (siftdown) ändert.

Zeitkomplexität des Algorithmus:

Für das Bilden des Heap (heapcreate) wird für $\frac{n}{2}$ Elemente die Funktion 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.

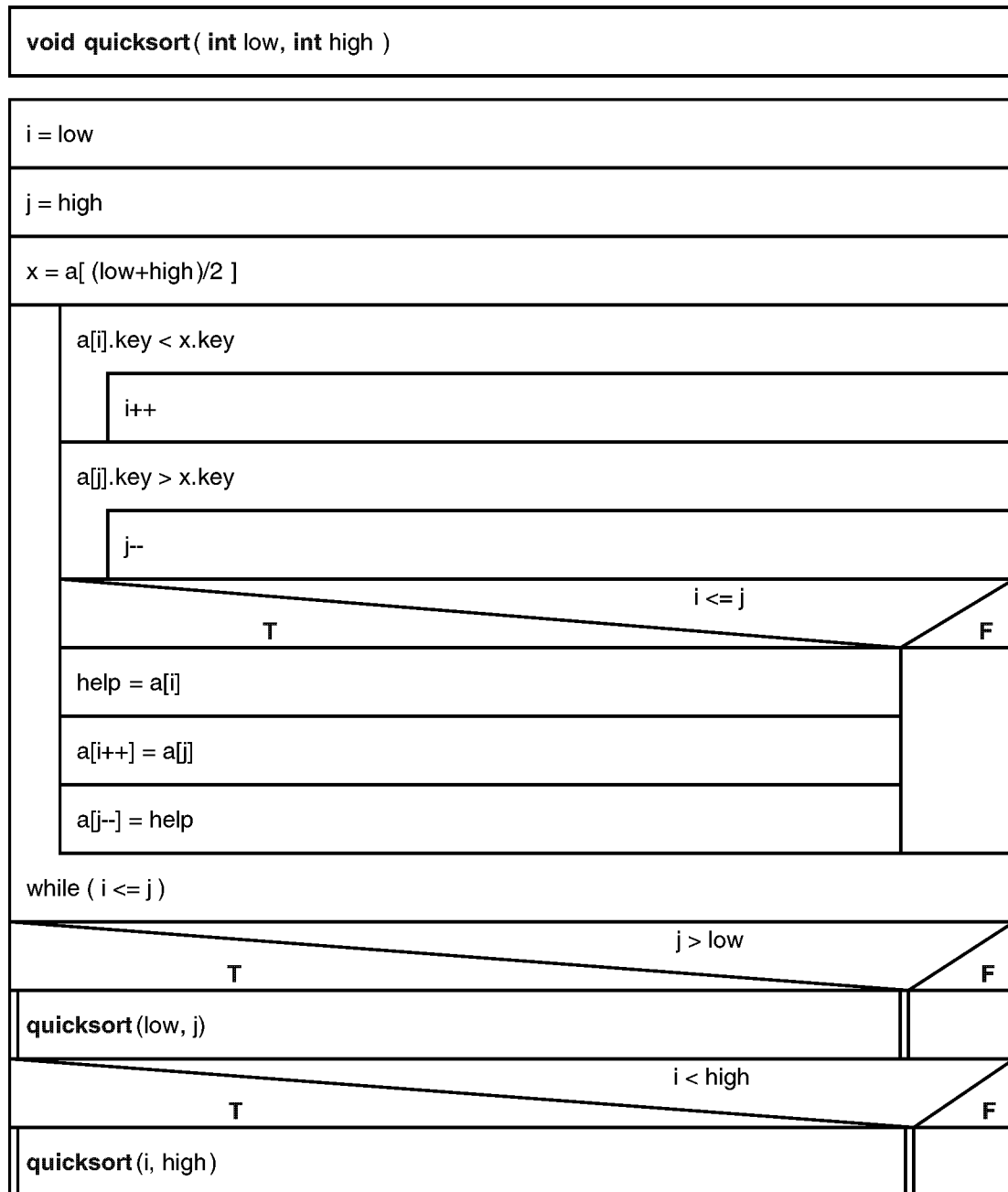
4.2.6 Quicksort

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

**Funktion quicksort (rekursive Lösung)****Korrektheit des Algorithmus:**

Nach Ende der Partitionierung in zwei Teillisten (Ende der do-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 do-Schleife $i==j$ und $a[i]==a[j]==x$. Dann folgt $i++$ und $j--$, und damit endet die do-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 Funktion quicksort												
1	2	3	4	5	7	8	9	10				

Bemerkung:

In Abb. 4.2 auf der nächsten Seite 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.

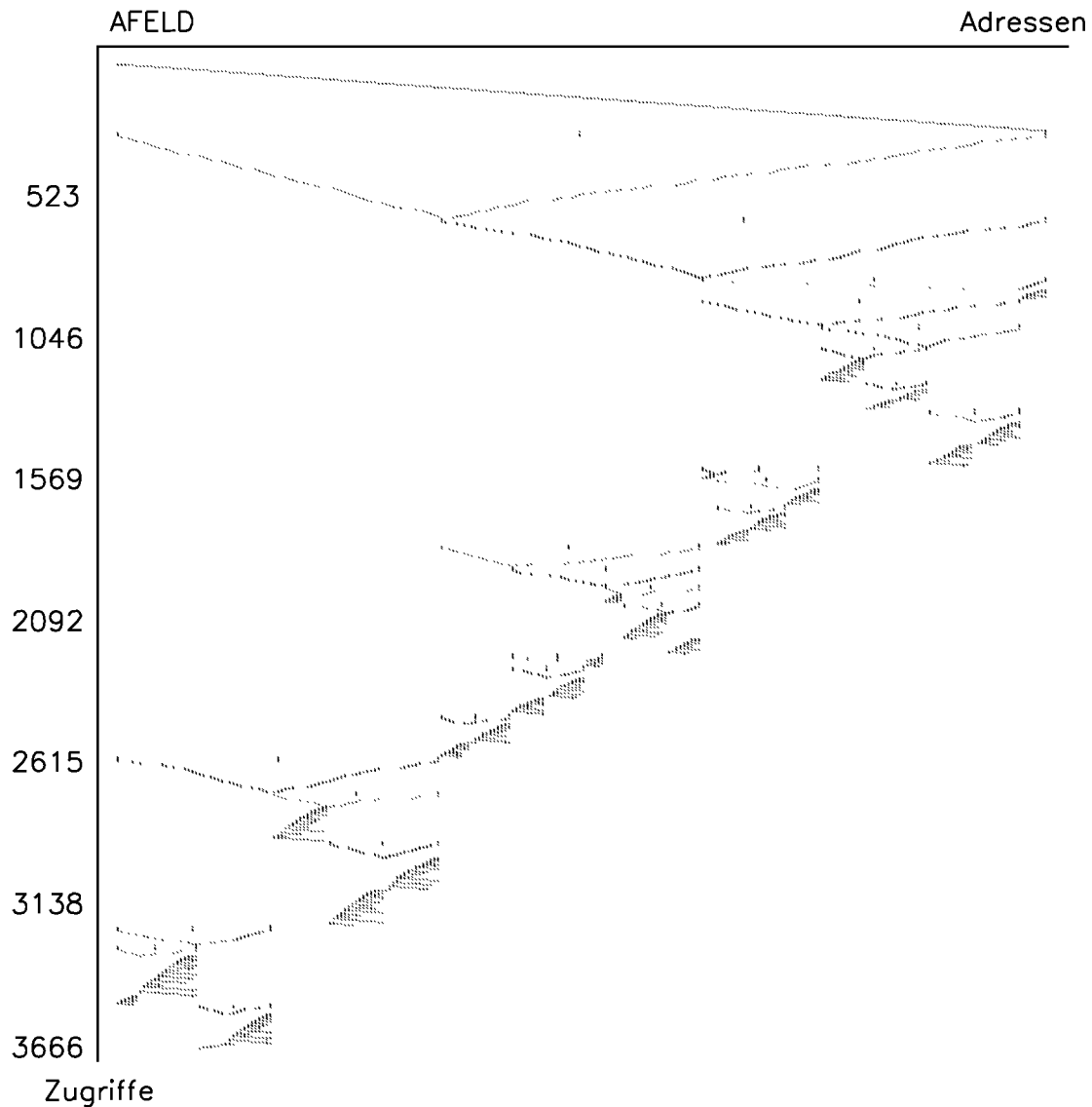
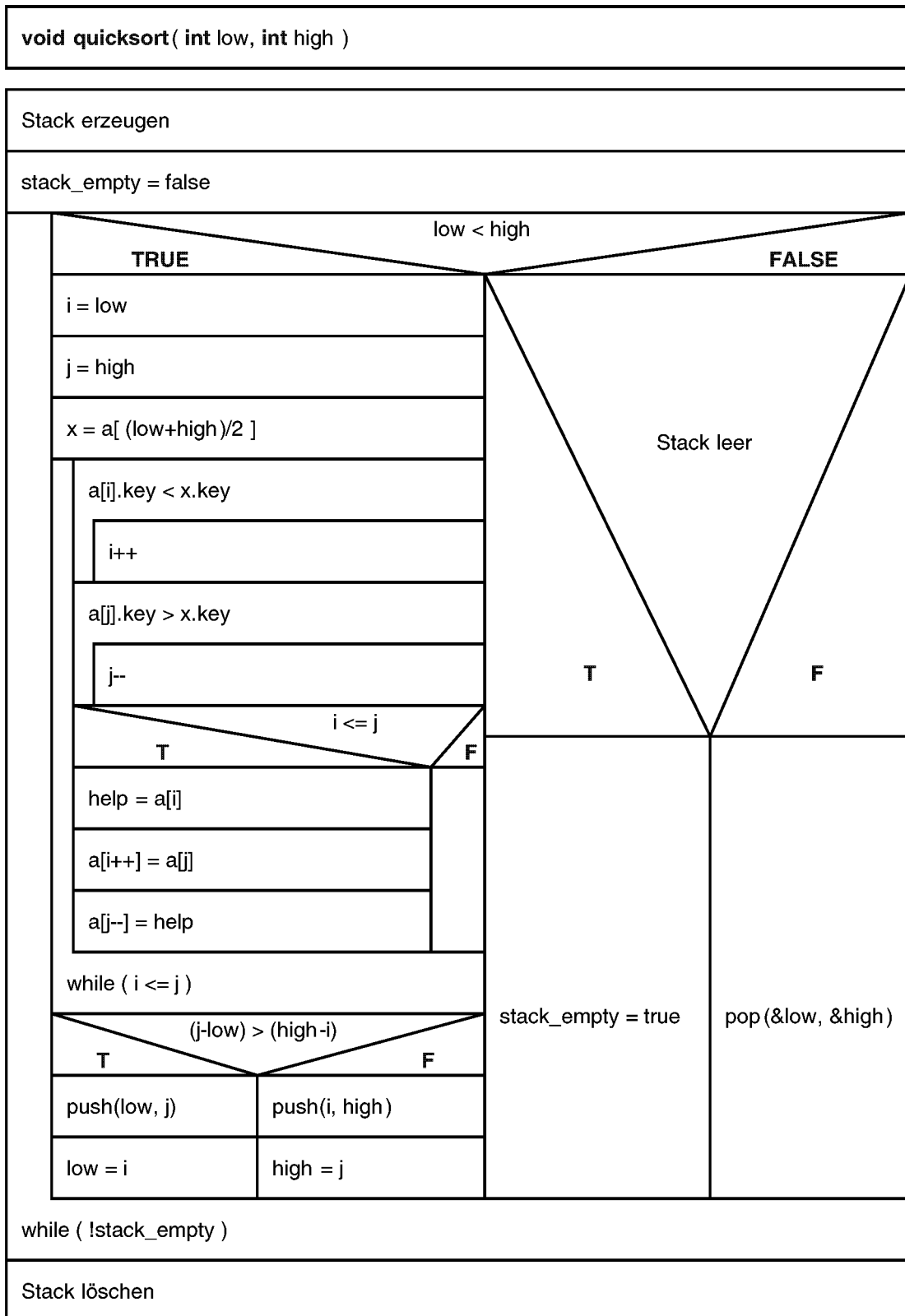


Abbildung 4.2: Speicherreferenzdiagramm für den Quicksort-Algorithmus

Bemerkung:

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

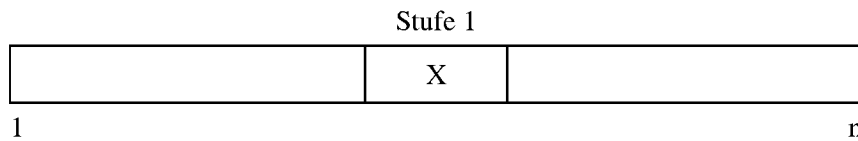
1. Nach der Partitionierung (Ende der inneren do-Schleife) wird der größere Bereich auf den Stack geschrieben. Der kleinere Bereich wird sofort weiterbehandelt, d.h. nicht auf den Stack geschrieben. Auf diese Weise wird gewährleistet, daß der Stack nicht mehr als $O(\log n)$ Elemente aufnehmen muß. (Bem.: 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 Funktion gestellt worden.

**Funktion quicksort (iterative Lösung)**

Zeitkomplexität von Quicksort:

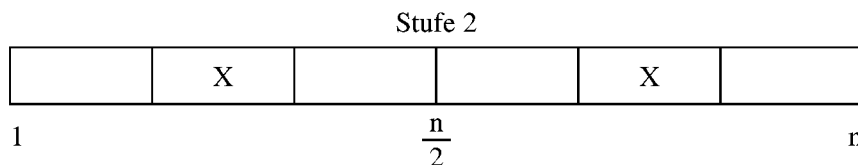
Aufwand im günstigsten Fall:

Die Teillisten werden stets halbiert.



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

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



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 $\lfloor \log n \rfloor$ Stufen.

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

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

Aufwand im Mittel:

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

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

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.

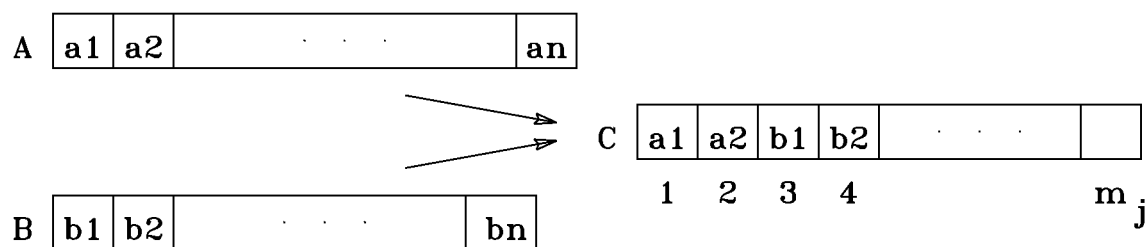


Abbildung 4.3: Mischen von sortierten Listen A und B zu einer sortierten Liste C

Zunächst gelte:

A,B,C : Listen, die Listen A und B seien vorsortiert

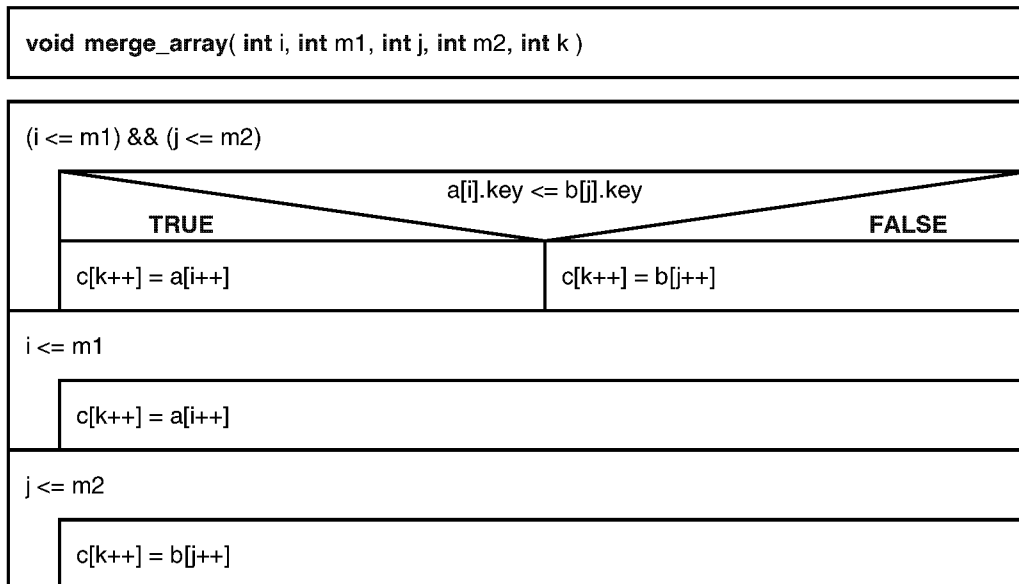
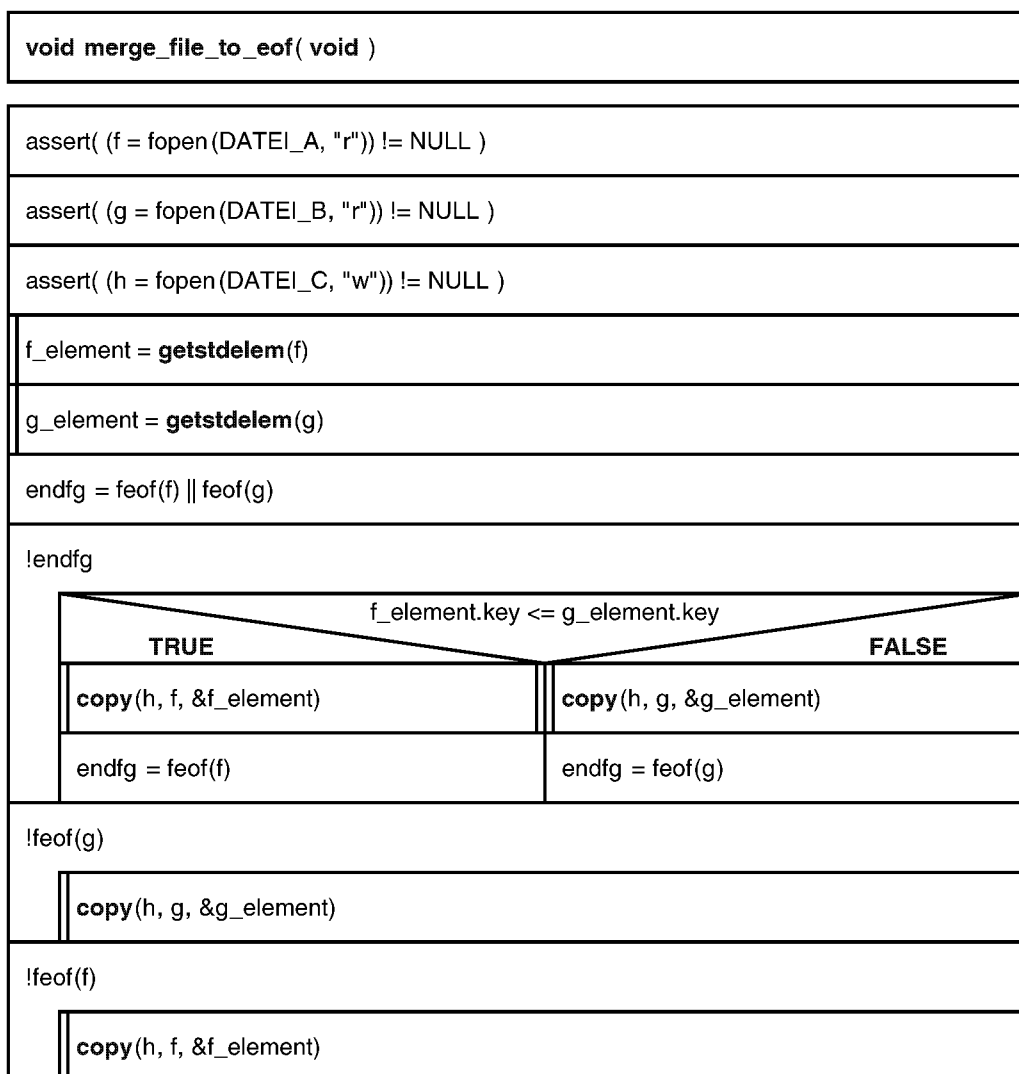
i,m1: untere/obere Grenze von A

j,m2: untere/obere Grenze von B

k: untere Grenze von C

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

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

**Funktion merge_array****Funktion merge_file_to_eof**

```
static void copy( FILE *out, FILE *in, stdelement *elem_ptr )
```

```
    putstdelem(*elem_ptr, out)
```

```
    *elem_ptr = getstdelem(in)
```

Funktion copy

```
stdelement getstdelem( FILE *istream )
```

```
    lese Element e (vom Typ stdelement) von der mit istream verknüpften Datei ein
```

```
    return e
```

Funktion getstdelem

```
void putstdelem( stdelement e, FILE *ostream )
```

```
    gebe Element e auf die mit ostream verknüpfte Datei aus, so daß es mit getstdelem
    wieder eingelesen werden kann
```

Funktion putstdelem

Die Funktion `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 Funktion `merge_file` entspricht dann der Funktion `merge_array`.

```
void merge_file( int i, int m1, int j, int m2, stdelement *a_ptr, stdelement *b_ptr )
```

```
(i <= m1) && (j <= m2)
```

```
a_ptr->key <= b_ptr->key
```

```
TRUE
```

```
FALSE
```

```
copy(c, a, a_ptr)
```

```
copy(c, b, b_ptr)
```

```
i++
```

```
j++
```

```
i <= m1
```

```
copy(c, a, a_ptr)
```

```
i++
```

```
j <= m2
```

```
copy(c, b, b_ptr)
```

```
j++
```

Funktion 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.

log2 und ipow2 können als Makros implementiert werden:

```
#define log2(x) (log(x)/log(2.0))
#define ipow2(i) (1<<(i))
```

void mergesort_array(void)
n1 = n/2
k = (int) log2(n)
for (i = 1; i <= k; i++)
distribute_array(n1)
length = ipow2(i-1) /* Länge der Teillisten */
mmax = n/ipow2(i) /* Anzahl der Mischvorgänge */
for (m = 1; m <= mmax; m++)
anfx = (m-1)*length + 1 /* Anfangsindex */
endx = m*length /* Endindex */
anfxc = (m-1)*ipow2(i) + 1 /* Anfangsindex für Array c */
merge_array(anfx, endx, anfx, endx, anfxc)

Funktion mergesort_array

static void distribute_array(int n1)
i = 1
for (j = 1; j <= n1; j++)
a[j] = c[i++]
for (j = 1; j <= n1; j++)
b[j] = c[i++]

Funktion distribute_array

void mergesort_file(void)
n1 = n/2
k = (int) log2(n)
for (i = 1; i <= k; i++)
distribute_file (n1)
assert((c = fopen(DATEI_C, "w")) != NULL)
assert((a = fopen(DATEI_A, "r")) != NULL)
assert((b = fopen(DATEI_B, "r")) != NULL)
a_element = getstdelem (a)
b_element = getstdelem (b)
length = ipow2(i-1) /* Länge der Teillisten */
mmax = n/ipow2(i) /* Anzahl der Mischvorgänge */
for (m = 1; m <= mmax; m++)
anfx = (m-1)*length + 1 /* Anfangsindex */
endx = m*length /* Endindex */
anfxc = (m-1)*ipow2(i) + 1 /* Anfangsindex für Array c */
merge_file (anfx, endx, anfx, endx, &a_element, &b_element)
fclose(a), fclose(b), fclose(c)

Funktion mergesort_file

static void distribute_file(int n1)
assert((c = fopen(DATEI_C, "r")) != NULL)
assert((a = fopen(DATEI_A, "w")) != NULL)
assert((b = fopen(DATEI_B, "w")) != NULL)
c_element = getstdelem (c)
for (j = 1; j <= n1; j++)
copy (a, c, &c_element)
for (j = 1; j <= n1; j++)
copy (b, c, &c_element)
fclose(a), fclose(b), fclose(c)

Funktion 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:

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

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* (die Elemente $a[1]$, ..., $a[n]$ seien vom Typ *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 Index i der Speicherzelle, in die dieses Element eingefügt werden kann.

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

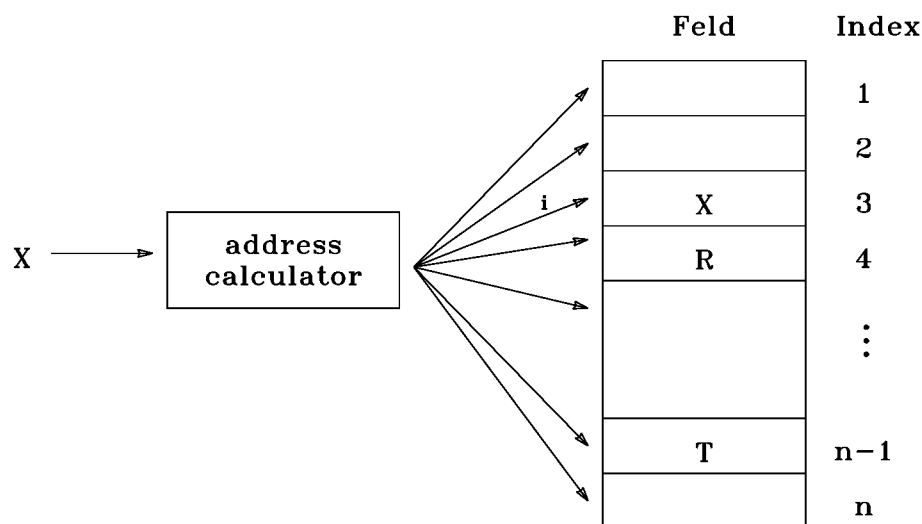


Abbildung 4.4: address calculator

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

```
void insert( hasharray a, stdelement newitem )

{
    i = address_calculator( newitem, key )
    a[i] = newitem
}
```

Funktion 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 4.4 *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 im allgemeinen *sehr viel* größer als die verfügbare Adreßmenge, d.h. die Hash-Funktion wird im allgemeinen nicht injektiv sein (d.h. mehrere Schlüssel haben dieselbe Adresse).

Definition 4.5 *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 Speicherversuch 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	⇒ Kollision ⇒ 1.Ersatzadresse 35 ⇒ Kollision ⇒ 2.Ersatzadresse 36
2034	Meier	35	
9539	Schulze	40	
3433	Schmidt	34	

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 4.6 *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 Sondierverfahren* genannt werden.

Definition 4.7 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 $n \equiv 3 \bmod 4$ (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

$$e_i(s) = \left(h(s) + (-1)^{i+1} i^2 \right) \bmod n$$

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

$$i = 4: \Rightarrow (-1)^5 = -1, i^2 = 16 \Rightarrow -i^2 \bmod 11 = -5 \equiv 6$$

Bemerkung:

- Ist $(-1)^{i+1} i^2 \bmod n$ negativ, so muß der Wert um 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 4.8 *Fibonacci-Hash*

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

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

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 4.9 *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.

i	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

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

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:

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

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	$5 \cdot \Theta \cdot \sigma$	$g(\sigma)$	$\lfloor g(\sigma) \rfloor$
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	3
7	21.63	1.63	1
8	24.72	4.72	4
9	27.81	2.81	2
10	30.90	0.90	0
11	33.99	3.99	3
12	37.08	2.08	2
13	40.17	0.17	0
14	43.26	3.26	3
15	46.35	1.35	1

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.).

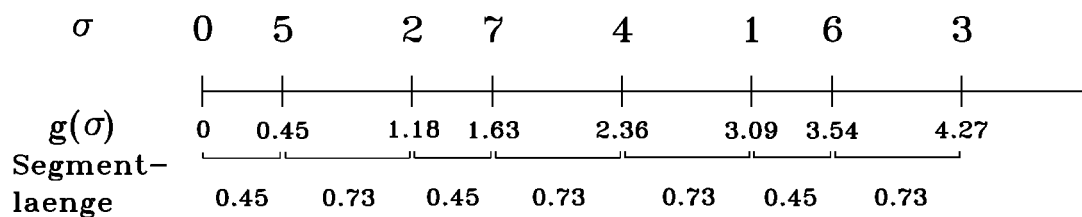


Abbildung 4.5: Darstellung der Swierczkowski-Verteilung für $N=7$

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:

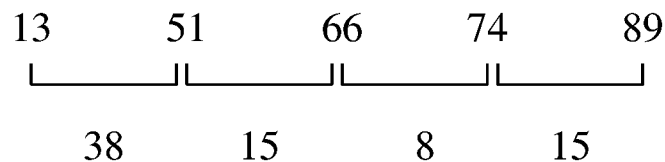
$$\frac{c_1}{c_2} = \frac{c_2}{c_3} = 0.61...$$

Beispiel 4.15 (Vergleiche Beispiel 4.10 auf Seite 127)

$S = \{1..9999\}$, $n = 100$ (Tabellengröße) Speichern aufeinanderfolgender Schlüssel:

S	Name	h(s)
1234	Schmitt	$\lfloor 100 \cdot 1234 \cdot 0.618034 \bmod 100 \rfloor + 1 = 66$
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 4.10 *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 4.11 *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:

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

$S(n, k)$

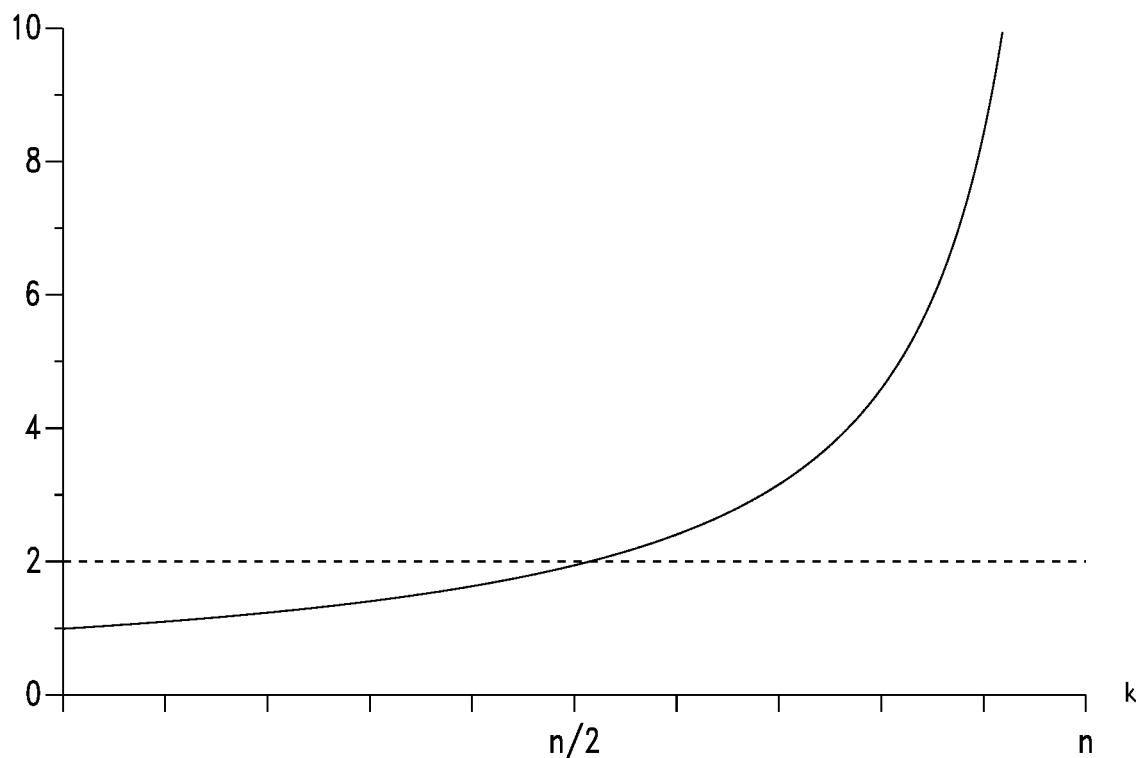


Abbildung 4.6: Qualitativer Verlauf von $S(n,k)$ für ein festes n

Beispiel 4.16 *Anzahl der Kollisionen beim Speichern eines neuen Datenrecords*

Sei $n=991$. Die Records werden in der Reihenfolge $k=1..maxn$ eingetragen (k bezeichnet die laufende Nummer des Records, $maxn$ 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).

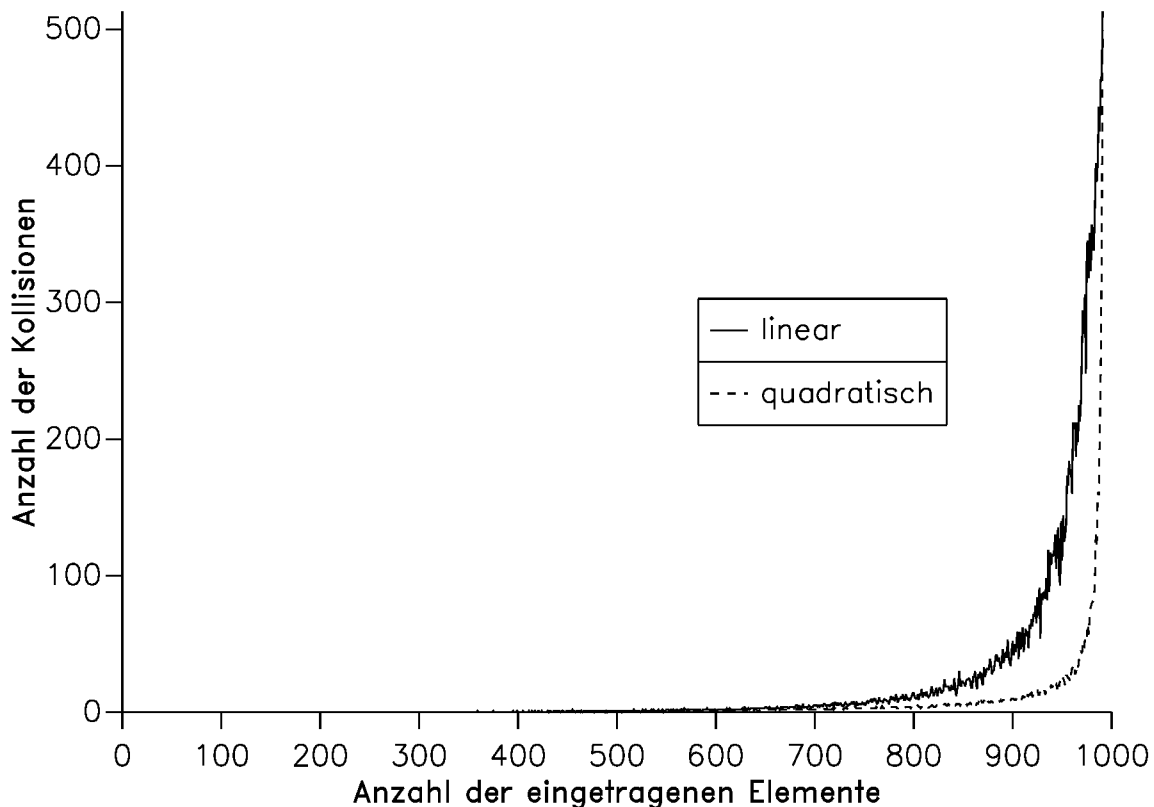


Abbildung 4.7: Kollisionen beim Fibonacci-Hash (mit linearem bzw. quadratischem Sondieren)

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 sollten die auftretenden Schlüssel möglichst gleichmäßig auf die verfügbaren Adressen verteilt werden (keine Häufung).
- Beim Berechnen der Ersatzadressen sollten 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 vollständigen Hash 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.

0	1		k-1
1			
.			
.			
.			
.			
.			
.			
n-1			

Speicherung:

1. Wie beim offenen Hash-Verfahren wird zunächst mit Hilfe der offenen Hash-Funktion aus dem Schlüssel der Hashindex ($0 \leq h(s) < n$) 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

```
#define N 1000
#define K 10

typedef struct {
    int key;
    char name[20];
    char ort[20];
} daten;

static daten tabelle[N][K];
static short pointer[N];    /* fuer alle i = 0, ..., N-1 gilt: */
                          /* 0 <= pointer[i] <= K */
```

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 $O\left(\frac{m}{n}\right)$ Schritte ($m \geq n$).

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

Bemerkung:

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. 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.

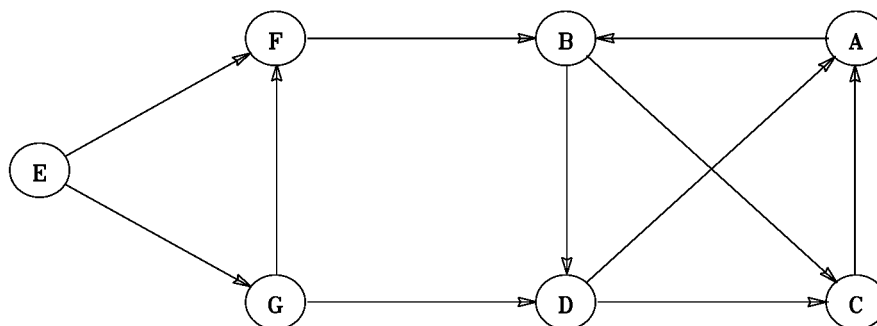
Diese Vorgehensweise wird DFS genannt, weil mit der Suche in Vorwärtsrichtung (tiefer, deeper) fortgefahren wird, solange dies möglich ist. Es gilt:

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

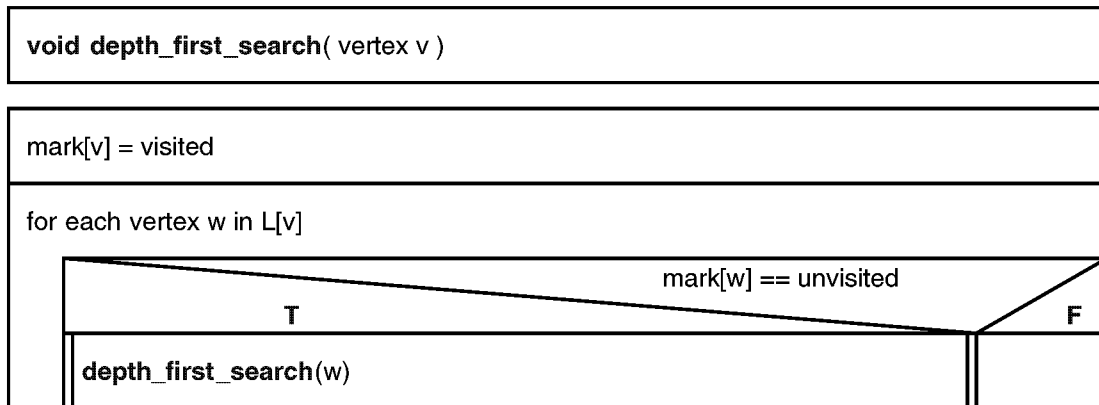
mark: Feld, dessen Elemente (Knoten) 'visited' oder 'unvisited' sein können. Alle Knoten sind zu Beginn unvisited.

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

Beispiel 4.17 Depth-First-Search



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



Funktion depth_first_search

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 Funktion 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 Funktion 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 Funktion 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'.

$$T_{dfs}^{av}(|E|) = T_{dfs}^w(|E|) = T_{dfs}^b(|E|) \in O(|E|)$$

Falls noch Knoten als nicht besucht markiert sind, wird DFS erneut aufgerufen. Damit erhöht sich die Zeitkomplexität für das Durchlaufen des gesamten Graphen auf:

$$\begin{aligned} T_{dfs}^{av}(|V|, |E|) &= T_{dfs}^w(|V|, |E|) = T_{dfs}^b(|V|, |E|) \\ &\in O(|V| + |E|) = O(\max\{|V|, |E|\}) \end{aligned}$$

Bemerkung:

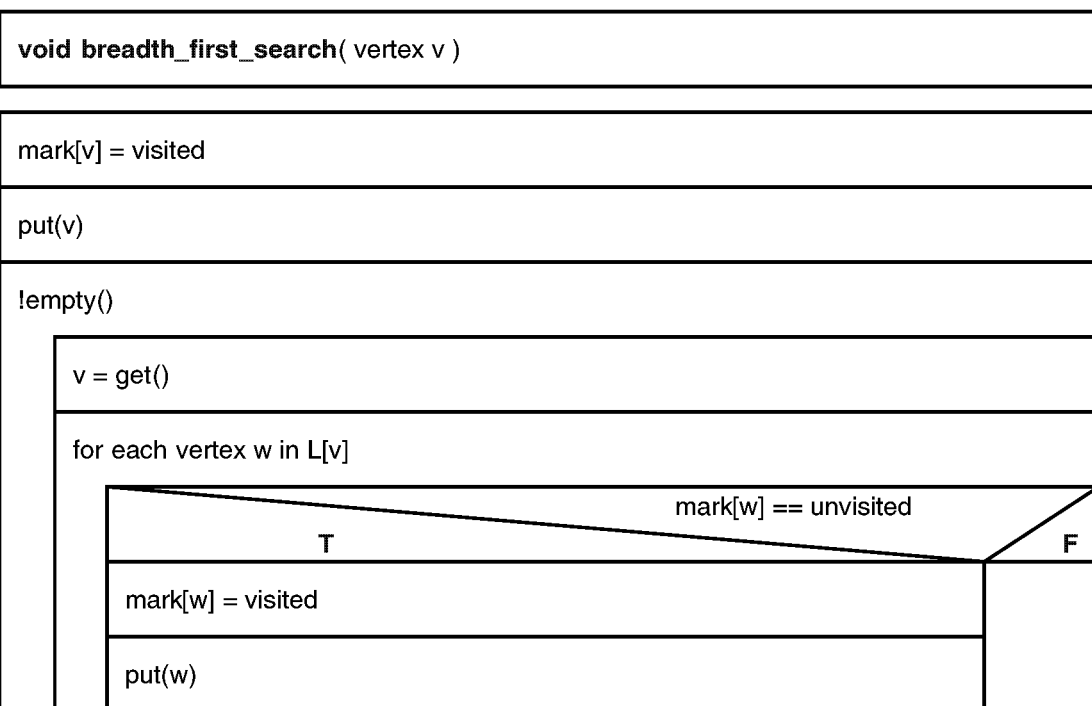
Falls der Algorithmus auf einer Adjazenzmatrix ausgeführt wird, ergibt sich als Zeitkomplexität $O(|V|^2)$.

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 auf Seite 57).



mark[w] == unvisited

Funktion breadth_first_search

Beispiel 4.18 *Breadth-First-Search*

Wende die Funktion BFS auf den Graphen aus Beispiel 4.17 auf Seite 137 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-paths).

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 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, oder ähnliches 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.

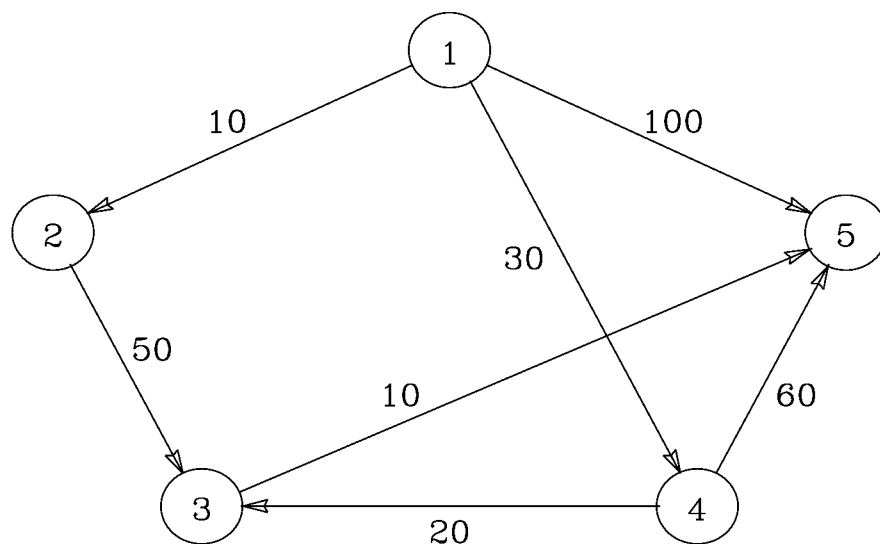
Voraussetzung: $C(u, v) \geq 0 \forall u, v \in V$.

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 $v \neq \text{Quelle}$). Am Ende des Algorithmus kann der kürzeste Weg zu jedem Knoten mit Hilfe des Feldes P bestimmt werden.

void dijkstra(void)	
S[1] = true /* Knoten 1 ist in der Menge S enthalten */	
for (i = 2; i <= N; i++)	
S[i] = false	/* keine weiteren Elemente in S */
d[i] = c[1][i]	
p[i] = 1	
for (i = 1; i <= N-1; i++)	
wähle einen Knoten w in V-S, so daß d[w] ein Minimum ist	
S[w] = true /* füge w zu S hinzu */	
for (jeden Knoten v in V-S)	
d[w]+c[w][v] < d[v] T F	
d[v] = d[w]+c[w][v]	
p[v] = w	

Funktion 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:

$$T_{dijkstra}^{av}(|V|) = T_{dijkstra}^w(|V|) = T_{dijkstra}^b(|V|) \in O(|V|^2)$$

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-paths). 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 .

4.4.3.1 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.

4.4.3.2 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:

$$A_k[i][j] = \min(A_{k-1}[i][j], A_{k-1}[i][k] + A_{k-1}[k][j])$$

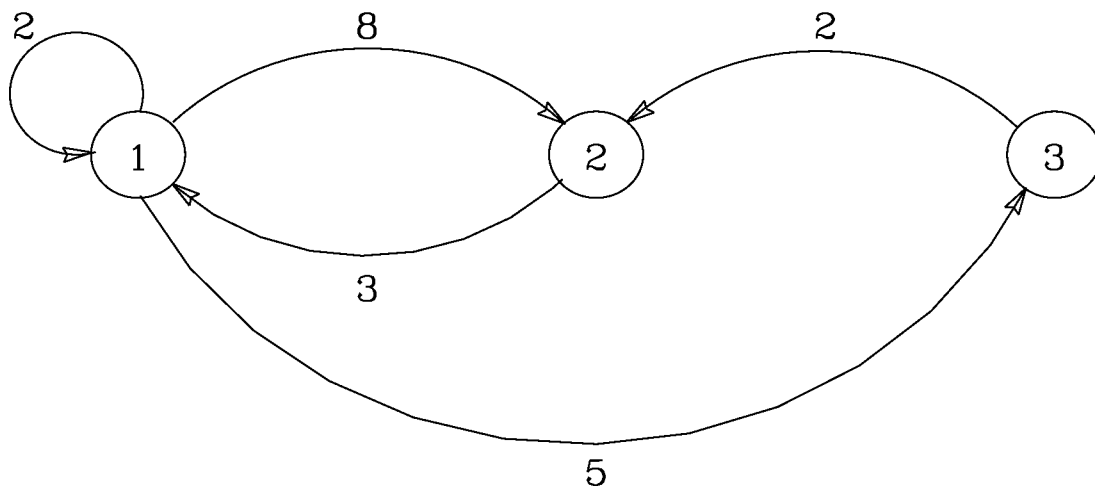
Zum Schluß enthält $A_n[i][j]$ den kürzesten Weg von i nach j , denn nach der k -ten Iteration ist $A_k[i][j]$ die kürzeste Entfernung von i nach j mit Zwischenknoten aus der Menge $\{1, \dots, k\}$.

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 $P[i][j] = -1$ ist, dann ist die kürzeste Entfernung die direkte Verbindung.

void floyd(double A[N+1][N+1], double C[N+1][N+1])	
for (i = 1; i <= N; i++)	
for (j = 1; j <= N; j++)	
A[i][j] = C[i][j]	
P[i][j] = -1	
for (i = 1; i <= N; i++)	
A[i][i] = 0.0	
for (k = 1; k <= N; k++)	
for (i = 1; i <= N; i++)	
for (j = 1; j <= N; j++)	
A[i][k]+A[k][j] < A[i][j]	
T	F
A[i][j] = A[i][k] + A[k][j]	
P[i][j] = k	

Funktion floyd

Beispiel 4.20 Veranschaulichung des Floyd-Algorithmus



$A_0[i][j]$	1	2	3	$A_1[i][j]$	1	2	3
1	0	8	5	1	0	8	5
2	3	0	∞	2	3	0	8
3	∞	2	0	3	∞	2	0

$A_2[i][j]$	1	2	3	$A_3[i][j]$	1	2	3
1	0	8	5	1	0	7	5
2	3	0	8	2	3	0	8
3	5	2	0	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:

$$T_{floyd}^{av}(|V|) = T_{floyd}^w(|V|) = T_{floyd}^b(|V|) \in O(|V|^3)$$

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

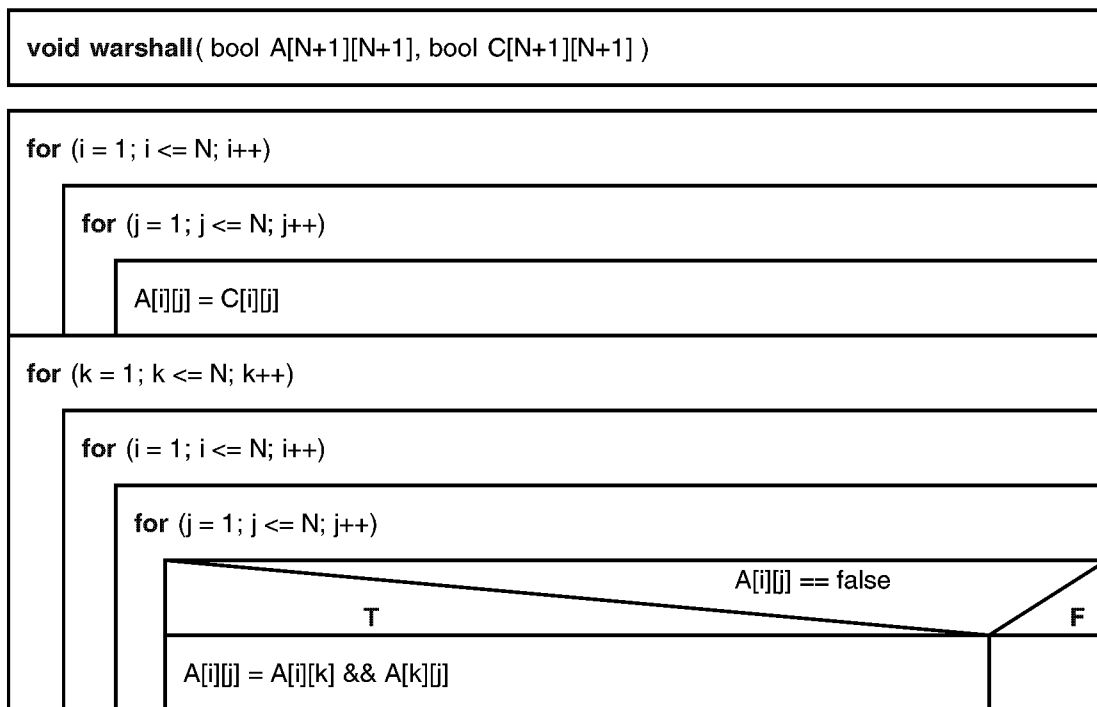
In speziellen Fällen ist man lediglich daran interessiert, ob ein Weg einer beliebigen Länge von Knoten i zu Knoten j existiert. Der Floyd-Algorithmus kann dann so modifiziert werden, daß er dieses Problem löst; der entsprechende Algorithmus heißt *Warshall-Algorithmus*.

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:



Funktion warshall

Beispiel 4.21 Veranschaulichung des Warshall-Algorithmus

Zu der Adjazenzmatrix A des gerichteten Graphen aus Beispiel 4.20 auf Seite 144 erhält man den transitiven Abschluß A^* wie folgt:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad A^* = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Die Zeitkomplexität ist gegenüber dem Floyd-Algorithmus unverändert, lediglich die Formel ist modifiziert:

$$T_{warshall}^{av}(|V|) = T_{warshall}^w(|V|) = T_{warshall}^b(|V|) \in O(|V|^3)$$

4.4.4.1 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. A.1) 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. $\Sigma = \{a,b\}$, $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$ (Σ^* bezeichnet die Menge aller Worte über dem Alphabet Σ)

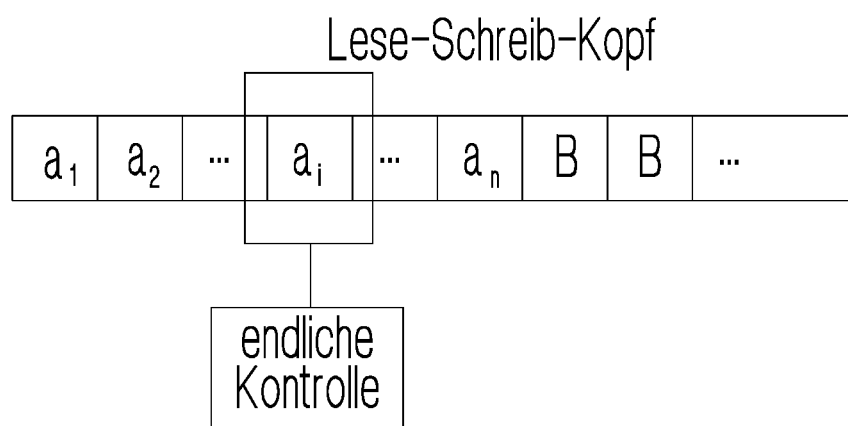


Abbildung A.1: Turingmaschine

Funktionsbeschreibung der Turingmaschine:

Die n linken Felder ($n \geq 0$) enthalten die Eingabe, d.h. ein Wort aus Zeichen des Eingabealphabetes. 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:

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 A.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	<i>Zustandsmenge</i>
Γ	<i>Bandalphabet</i>
B	<i>Blank</i>
$\Sigma \subseteq \Gamma \setminus \{B\}$	<i>Eingabealphabet</i>
$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$	<i>Übergangsfunktion</i> (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 A.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 A.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 ausgezeichnete Endzustand aus F erreicht wird, so spricht man davon, daß das Wort w akzeptiert wird.

Definition A.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 A.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:

$q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash$
 $q_2 X 0 Y 1 \vdash X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash$
 $X X q_2 Y Y \vdash X q_2 X Y Y \vdash X X q_0 Y Y \vdash X X Y q_3 Y \vdash$
 $X X Y Y q_3 \vdash X X Y Y B q_4$

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

Beispiel: Realisierung von Mengen (sets)

1. Repräsentation

```
/* Typdefinitionen: */
typedef enum { false, true } bool;
typedef struct set set;

struct set {
    int type_id;
    int lb;
    int ub;
    int start;
    int size;
    unsigned char *elem;
};
```

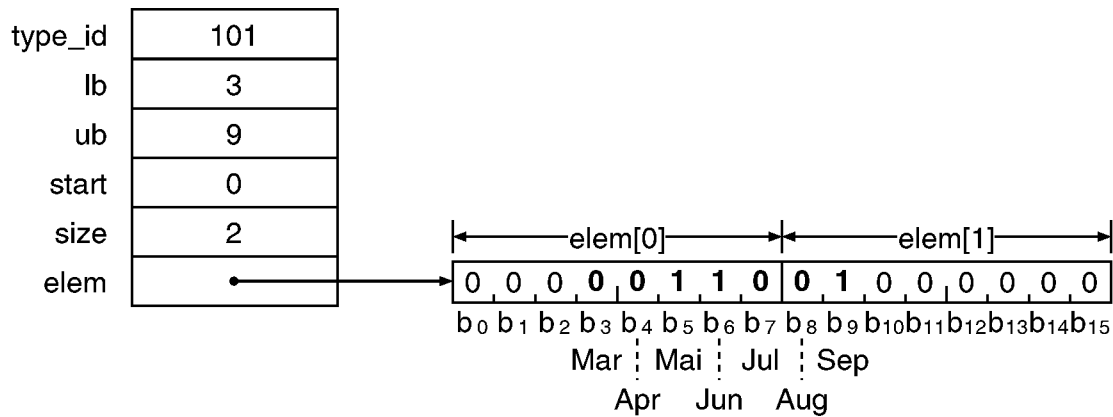
Bedeutung der einzelnen Komponenten:

<i>type_id</i>	ganzzahlige Kennung, die die Art der Mengenelemente festlegt.
<i>lb</i>	kleinstes Element, das in die Menge aufgenommen werden kann.
<i>ub</i>	größtes Element, das in die Menge aufgenommen werden kann.
<i>start</i>	ist der größte durch 8 ohne Rest teilbare Wert $\leq lb$. Dadurch, daß das erste Bit des Vektors <i>elem</i> immer mit einem Mengenelement korrespondiert, dessen Wert (Ordnungszahl) ein Vielfaches von 8 ist, können Mengen mit unterschiedlichem 'lower bound' <i>lb</i> ohne zusätzlichen Aufwand miteinander verknüpft werden.
<i>size</i>	Länge des Vektors <i>elem</i> .
<i>elem</i>	Zeiger auf einen dynamisch erzeugten Vektor. Die einzelnen Bits der Vektorelemente <i>elem</i> [0], ..., <i>elem</i> [<i>size</i> -1] repräsentieren die Elemente der Menge, wobei ein auf 1 gesetztes Bit anzeigt, daß das korrespondierende Element in der Menge enthalten ist.

Beispiel B.1 Erzeugung und interne Repräsentation einer Menge von Monaten

```
#include "set.h"
#define SET_OF_MONTH 101

typedef enum { Jan=1, Feb, Mar, Apr, Mai, Jun,
               Jul, Aug, Sep, Okt, Nov, Dez } month_t;
month_t mvec[3] = {Mai, Jun, Sep};
. . .
{
    set *mset = mkset(create(Mar, Sep, SET_OF_MONTH),
                      mvec, 3, sizeof(month_t));
    . . .
}
```



2. Implementation

Die Übergabe von Sets erfolgt aus Effizienzgründen grundsätzlich per 'call by reference'. Außerdem braucht dann dem Anwendungsprogramm der genaue Aufbau des Records *struct set* nicht bekannt zu sein ('black box', Geheimnisprinzip).

```
/* lokale Variablen der Funktion assign: */
int i, j;
```

set * assign (set *s1, const set *s2)
assert(s1->type_id == s2->type_id)
for (i = 0; (i < s1->size) && (s1->start+i*CHAR_BIT < s2->start); i++)
s1->elem[i] = 0
for (j = 0; (j < s2->size) && (s2->start+j*CHAR_BIT < s1->start); j++)
T s2->elem[j] F
fatal("Assignment out of bounds")
(i < s1->size) && (j < s2->size)
s1->elem[i++] = s2->elem[j++]
i < s1->size
s1->elem[i++] = 0
j < s2->size
T s2->elem[j++] F
fatal("Assignment out of bounds")
return s1

Funktion assign

```

/* lokale Variablen der Funktion in: */
bool found;
int i, n;

```

bool in(const set *s1, int elem)	
(elem >= s1->lb) && (elem <= s1->ub)	
T	F
i = (elem - s1->start)/CHAR_BIT	found = false
n = (CHAR_BIT-1) - (elem - s1->start)%CHAR_BIT	
found = ((s1->elem[i] & (1<<n)) != 0)	
return found	

Funktion in

Die folgenden drei Funktionen speichern ihr Ergebnis jeweils in einer lokalen statischen Variablen

```
static struct set result;
```

und geben einen Zeiger auf diese Variable zurück. Es ist zu beachten, daß diese statisch erzeugte Menge nicht durch *delete* freigegeben werden darf (Die Vereinbarung des Funktionswertes als *const set ** soll unbeabsichtigte Aufrufe von *delete* sowie Zuweisungen auf *result* verhindern.). Außerdem steht das Ergebnis in der Variablen *result* auch nur bis zum nächsten Aufruf der jeweiligen Operation zur Verfügung. Daraus ergibt sich die Einschränkung, daß geschachtelte Aufrufe

(z.B. `intersection(a, intersection(b,c))`)
nicht zulässig sind.

```

/* lokale Variablen der Funktion intersection: */
static struct set result;
static int maxsize = 0;
bool overlapping;
int i, j, k;

```

const set *intersection (const set *s1, const set *s2)	
assert((s1 != &result) && (s2 != &result))	
assert(s1->type_id == s2->type_id)	
result.type_id = s1->type_id	
s1->lb < s2->lb	
TRUE	FALSE
result.lb = s2->lb	result.lb = s1->lb
result.start = s2->start	result.start = s1->start
i = (s2->start - s1->start)/CHAR_BIT	i = 0
j = 0	j = (s1->start - s2->start)/CHAR_BIT
s1->ub < s2->ub	
TRUE	FALSE
result.ub = s1->ub	result.ub = s2->ub
overlapping = (result.lb <= result.ub)	
!overlapping	
T	F
result.ub = result.lb	
result.size = (result.ub - result.start)/CHAR_BIT + 1	
result.size > maxsize	
T	F
result.elem = (unsigned char *) realloc(result.elem, result.size)	
assert(result.elem != NULL)	
maxsize = result.size	
overlapping	
T	F
for (k = 0; k < result.size; k++)	result.elem[0] = 0
result.elem[k] = s1->elem[i++] & s2->elem[j++]	
return (&result)	

Funktion intersection

```

/* lokale Variablen der Funktion union: */
static struct set result, int maxsize = 0;
const set *temp;
int i, j;

```

const set *union(const set *s1, const set *s2)		
assert((s1 != &result) && (s2 != &result))		
assert(s1->type_id == s2->type_id)		
result.type_id = s1->type_id		
s1->lb > s2->lb		F
T		
temp = s1		
s1 = s2		
s2 = temp		
result.lb = s1->lb		
result.start = s1->start		
s1->ub >= s2->ub		FALSE
TRUE		
result.ub = s1->ub	result.ub = s2->ub	
result.size = (result.ub - result.start)/CHAR_BIT + 1		
result.size > maxsize		F
T		
result.elem = (unsigned char *) realloc(result.elem, result.size)		
assert(result.elem != NULL)		
maxsize = result.size		
for (i = 0; i < s1->size; i++)		
result.elem[i] = s1->elem[i]		
i < result.size		
result.elem[i++] = 0		
i = (s2->start - s1->start)/CHAR_BIT		
for (j = 0; j < s2->size; j++)		
result.elem[i++] = s2->elem[j]		
return (&result)		

Funktion union

```

/* lokale Variablen der Funktion difference: */
static struct set result;
static int maxsize = 0;
int i, j;

```

const set *difference(const set *s1, const set *s2)	
assert(s2 != &result)	
assert(s1->type_id == s2->type_id)	
result.type_id = s1->type_id	
result.lb = s1->lb	
result.ub = s1->ub	
result.start = s1->start	
result.size = s1->size	
result.size > maxsize	
T	F
result.elem = (unsigned char *) realloc(result.elem, result.size)	
assert(result.elem != NULL)	
maxsize = result.size	
s1->start < s2->start	
T	F
for (i = 0; (i < s1->size) && (s1->start+i*CHAR_BIT < s2->start); i++)	i = 0
result.elem[i] = s1->elem[i]	j = (s1->start - s2->start)/CHAR_BIT
j = 0	
(i < s1->size) && (j < s2->size)	
result.elem[i] = s1->elem[i] & ~s2->elem[j++]	
i++	
i < s1->size	
result.elem[i] = s1->elem[i]	
i++	
return (&result)	

Funktion difference

```

/* lokale Variablen der Funktion equal: */
bool eq;
int i, j;

```

```
bool equal( const set *s1, const set *s2 )
```

```
assert(s1->type_id == s2->type_id)
```

```
eq = true
```

```
for (i = 0; eq && (i < s1->size) && (s1->start+i*CHAR_BIT < s2->start); i++)
```

```
    eq = (s1->elem[i] == 0)
```

```
for (j = 0; eq && (j < s2->size) && (s2->start+j*CHAR_BIT < s1->start); j++)
```

```
    eq = (s2->elem[j] == 0)
```

```
eq && (i < s1->size) && (j < s2->size)
```

```
    eq = (s1->elem[i++] == s2->elem[j++])
```

```
eq && (i < s1->size)
```

```
    eq = (s1->elem[i++] == 0)
```

```
eq && (j < s2->size)
```

```
    eq = (s2->elem[j++] == 0)
```

```
return eq
```

Funktion equal

```

/* lokale Variablen der Funktion subset: */
bool sub;
int i, j;

```

bool subset (const set *s1, const set *s2)	
assert(s1->type_id == s2->type_id)	
sub = true	
s1->start < s2->start	
TRUE	FALSE
for (i = 0; sub && (i < s1->size) && (s1->start+i*CHAR_BIT < s2->start); i++)	i = 0
sub = (s1->elem[i] == 0)	j = (s1->start - s2->start)/CHAR_BIT
j = 0	
sub && (i < s1->size) && (j < s2->size)	
sub = ((s1->elem[i++] & ~s2->elem[j++]) == 0)	
sub && (i < s1->size)	
sub = (s1->elem[i++] == 0)	
return sub	

Funktion subset

```

/* lokale Variablen der Funktion mkset: */
const char *v = (const char *) elem_vec;
int elem, i, j, k;

```

set *mkset(set *s, const void *elem_vec, int n, int size)	
assert(size <= sizeof(int))	
for (i = 0; i < s->size; i++)	
	s->elem[i] = 0
for (i = 0; i < n; i++)	
	elem = 0
	for (j = 0; j < size; j++)
	elem <= CHAR_BIT
	elem += *v++
	(elem < s->lb) (elem > s->ub)
T	F
	fatal("Source contains elements out of target set")
	k = (elem - s->start)/CHAR_BIT
	s->elem[k] = 1 << (CHAR_BIT-1) - (elem - s->start)%CHAR_BIT
return s	

Funktion mkset

Durch den zusätzlichen Parameter *size* wird die Größe (in Bytes) eines Elementes von *elem_vec* festgelegt.

```
/* lokale Variablen der Funktion create: */
set *s;
int n;
```

set *create(int lbound, int ubound, int basetype_id)	
assert(lbound <= ubound)	
s = (set *) malloc(sizeof(set))	
assert(s != NULL)	
s->type_id = basetype_id	
s->lb = lbound	
s->ub = ubound	
lbound >= 0	
TRUE	FALSE
n = lbound%CHAR_BIT	n = (CHAR_BIT-1) - (-lbound-1)%CHAR_BIT
s->start = lbound - n	
s->size = (ubound - s->start)/CHAR_BIT + 1	
s->elem = (unsigned char *) calloc(s->size, 1)	
assert(s->elem != NULL)	
return s	

Funktion create

void delete(set *s)
free(s->elem)
free(s)

Funktion delete

static void fatal(const char *msg)
fprintf(stderr, "Error: %s\n", msg)
exit(EXIT_FAILURE)

Funktion fatal

Anhang C

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

Index

A	
Abstrakter Datentyp	11
Abstraktionsebenen	11
Acht-Damen-Problem	55
Adjazenzmatrix	64
Akzeptierte Sprache einer Turingmaschine	149
Algorithmen	
Ausgewählte Algorithmen	101
Effizienzbeurteilung	6
Spezifikation durch	
Pre- und Post-Conditions	13
Zeit- und Speicherkomplexität	6
Arrays	17
Atomare Objekte	10
Aufwandsbetrachtungen	8
Aufzählungstyp	19
Ausgewählte Algorithmen	101
Graphenalgorithmen	137
Hash-Verfahren	126
Sortiervverfahren	105
Suchverfahren	101
average-case	8
B	
B-Bäume	90
Backtracking	54
Bäume	66
Binärbäume	66
Binäre Suchbäume	79, 103
Haufen (heaps)	95
Höhenbalancierte Bäume	90
B-Bäume	90
Begriffe bei Bäumen	
Blatt	66
Bruder eines Knoten	66
Ebene eines Knoten (level)	66
Höhe eines Baumes	66
Innerer Knoten	66
Nachfolger in einem Baum	66
Sohn eines Knoten	66
Teilbaum	66
Vater eines Knoten	66
Vorgänger in einem Baum	66
Berechenbare Funktion	6
best-case	8
BFS	139
Binärbäume	66
Geordneter Binärbaum	67
Minimaler Binärbaum	89
Voller Binärbaum	67
Vollständiger Binärbaum	89
Binäre Suchbäume	79, 89
Binäres Einfügen	109
Binäres Suchen	102
Binary Search	102
bool	25
Breadth-First-Search-Algorithmus	139
Bubblesort	109
C	
C Datentypen	
array	17
file	27
pointer	22
record (struct)	19
C Textdateien	29
C-Dateien	27
D	
Datenstrukturen	
Bäume	66
Einfache Datenstrukturen	16
Einführung der Grundbegriffe	5
Graphen	62
Haufen (heaps)	95
Heterogene Datenstruktur	16
Homogene Datenstruktur	16
Lineare Datenstrukturen	30
Datentypen	
abstrakt	11
atomar	10
physikalisch	11
strukturiert	10
virtuell	11
Depth-First-Search-Algorithmus	137
DFS	137
Dijkstra-Algorithmus	141
Divisions-Hash	128
Doppelt verkettete Liste	43
Durchlaufen von gerichteten Graphen	
Breadth-First-Search	139
Depth-First-Search	137
Kürzeste Wege	141
Durchlaufstrategien in Bäumen	
Inorder	67, 69, 76
Postorder	67, 69, 76
Preorder	67, 69, 76
Dynamische Datenstrukturen	21, 27

C-Operation free	22
C-Operation malloc	22
E	
Effizienz von Algorithmen	6
Einfach verkettete Liste	30
Abstrakter Datentyp	30
Realisierung durch drei Felder	40
Realisierung durch ein Feld	41
Realisierung über Pointer	37
Einfache Datenstrukturen	16
Mengen (sets)	23
Sätze (records)	19
Sequenzen (sequences)	25
Zeiger (pointer)	21
Einfügen	
in binären Suchbaum	80, 82
in eine lineare Liste	31, 34
in einen B-Baum	91
in einen Heap	97
Sortieren durch	107
Externe Sortierverfahren	105
F	
Felder (arrays)	17
Fibonacci-Hash	130
Fibonacci-Folge	130
Swierczkowski-Verteilung	131
Zugriffszeiten	134
FIFO (First-In-First-Out)	57
Files	27
Floyd-Algorithmus	143
free	22
G	
Gerichtete Graphen	62
Gewichtete Graphen	64
Graphen	62
Graphenalgorithmen	137
Breadth-First-Search-Algorithmus	139
Depth-First-Search-Algorithmus	137
Dijkstra-Algorithmus	141
Floyd-Algorithmus	143
Warshall-Algorithmus	145
Graphentheoretische Begriffe	
Adjazenz	64
Einfacher Weg	64
Gerichteter Graph	62
Gewichteter Graph	64
Kantenmenge E	62
Knotenmenge V	62
Kreis	64
Teilgraph	64
Ungerichteter Graph	62
Weg	64
Greedy-Algorithmen	141
Groß-O-Notation	7

H	
Hash-Funktion	127
Hash-Verfahren	126
Divisions-Hash	128
Fibonacci-Hash	130, 134
Füllungsgrad	133
Hash in Teillisten	135
Ideales Hashing	133
Kollisionen beim Hashing	127, 133
Lineares Sondieren (linear probing) ..	128
Offene Hash-Verfahren	127, 134
Quadratisches Sondieren (quadratic probing)	129
Schrittzahl	133
Vollständiger Hash	129
Haufen (heaps)	95, 111
Heap	95
Heapsort	111
Höhenbalancierte Bäume	90, 126
B-Bäume	90

I	
Inorder	67, 69, 76
Insertion Sort	107
Interne Sortierverfahren	105

K	
Knoten	
Bruder eines	66
Ebene eines	66
Innerer	66
Sohn eines	66
Vater eines	66
Kollisionsstrategien	
Lineares Sondieren (linear probing) ..	128
Quadratisches Sondieren (quadratic probing)	129
Komplexität eines Algorithmus	
Groß-O-Notation	7
Königsberger Brückenproblem	63
Kürzeste Wege in Graphen	
Dijkstra-Algorithmus	141
Floyd-Algorithmus	143

L	
LIFO (Last-In-First-Out)	48
Linear Probing	128
Lineare Datenstrukturen	30
Einfach verkettete Liste	
Abstrakter Datentyp	30
Realisierung durch drei Felder	40
Realisierung durch ein Feld	41
Realisierung über Pointer	37
Mehrfach verkettete Liste	43
Stapel (stack)	48
Warteschlange(queue)	57
Lineares Sondieren	128
Lineares Suchen	101

- Linked List 30
- Löschen
 - in einem B-Baum 92
 - in einem binären Suchbaum 80
 - in einer linearen Liste 31, 35
- M**
- malloc 22
- Matrix 18
 - Adjazenz 64
- Median-of-three-Methode 118
- Mehrfach verkettete Liste 43
- Mengen (sets) 23
- Mergesort 119
- N**
- NULL pointer 21
- O**
- O-Notation 7
- Offene Hash-Verfahren 127
 - Divisions-Hash 128
 - Fibonacci-Hash 130, 134
 - Zusammenfassung der Eigenschaften 134
- P**
- Physikalischer Datentyp 11
- Pointer 21
- Postorder 67, 69, 76
- Pre- und Post-Conditions 13
- Preorder 67, 69, 76
- Q**
- Quadratic Probing 129
- Quadratisches Sondieren 129
- Queue 57
- Quicksort 113
 - Median-of-three-Methode 118
- R**
- Records 19
- Rekursion 52
 - Rekursive Funktion 52
- S**
- Sätze (records) 19
- Schlange (queue) 57
- Schlüssel 16
- Schlüsselmenge 16
 - Felder (arrays) 17
- Selection Sort 106
- Sequences 25
- Sequential Search 101
- Sequenzen (sequences) 25
- Sets 23
- Sinking Sort 109
- Sortierpermutation 105
- Sortierte Liste 102
- Sortiervverfahren 105
- Externe 105
- Heapsort 111
- Interne 105
- Quicksort 113
- Sortieren durch Auswahl
 - (selection sort) 106
- Sortieren durch Einfügen
 - (insertion sort) 107
- Sortieren durch Mischen (Mergesort) 119
- Sortieren durch Vertauschen
 - (Bubblesort, sinking sort) 109
- Stabile 106
- Speichermedien
 - Externe Speicher 13, 119
 - Hauptspeicher 13
 - Magnetbänder 13, 119
 - Magnetplatte 13
 - Trommel 13
- Speicherung von Graphen
 - Adjazenzliste 65
 - Adjazenzmatrix 64, 143
- Stabile Sortiervverfahren 106
- Stack 48
- Stapel (stack) 48
- stdelement 25, 32
- Struktureigenschaften
 - Baumstrukturen 15
 - Graphstrukturen 15
 - Hierarchische Strukturen 15
 - Lineare Strukturen 15
 - Menge 15
 - Netzwerkstrukturen 15
- Suchbaum 79, 103
- Suchverfahren 101
 - Binäres Suchen 102
 - Breadth-First-Search 139
 - Depth-First-Search 137
 - Lineares Suchen 101
- T**
- Transitiver Abschluß
 - Warshall-Algorithmus 145
- Turingmaschine 147
 - Konfiguration 148
 - Übergangsschritt 148
- U**
- Ungerichtete Graphen 62
- V**
- Vektor 18
- Verkettete Listen 30, 43
- Virtueller Datentyp 11
- W**
- Warshall-Algorithmus 145
- Warteschlange (queue) 57
- Wege (in Graphen) 64
- worst-case 8

Z

Zeiger (pointer)	21
Zeitkomplexität eines Algorithmus	7