Kapitel 1

Grundlagen

1.1 Algorithmen und ihre formalen Eigenschaften

In der Informatik unterscheidet man üblicherweise zwischen Verfahren zur Lösung von Problemen und ihrer Implementation in einer bestimmten Programmiersprache auf bestimmten Rechnern. Man nennt die Verfahren *Algorithmen*. Sie sind *das* zentrale Thema der Informatik. Die Entwicklung und Untersuchung von Algorithmen zur Lösung vielfältiger Probleme gehört zu den wichtigsten Aufgaben der Informatik. Die meisten Algorithmen erfordern jeweils geeignete Methoden zur Strukturierung der von den Algorithmen manipulierten Daten. Algorithmen und Datenstrukturen gehören also zusammen. Die richtige Wahl von Algorithmen und Datenstrukturen ist ein wichtiger Schritt zur Lösung eines Problems mit Hilfe von Computern. Thema dieses Buches ist das systematische Studium von Algorithmen und Datenstrukturen aus vielen Anwendungsbereichen. Bevor wir damit beginnen, wollen wir einige grundsätzliche Überlegungen zum Algorithmenbegriff vorausschicken.

Was ist ein Algorithmus?
Dies ist eine eher philosophische Frage, auf die wir in diesem Buch keine präzise Antwort geben werden. Das ist glücklicherweise auch nicht nötig. Wir werden nämlich in diesem Buch (nahezu) ausschließlich positive Aussagen über die Existenz von Algorithmen durch explizite Angabe solcher Algorithmen machen. Dazu genügt ein intuitives Verständnis des Algorithmenbegriffs und die Einsicht, daß sich die konkret angegebenen Algorithmen etwa in einer höheren Programmiersprache wie Pascal formulieren lassen. Erst wenn man eine Aussage der Art „Es gibt *keinen* Algorithmus, der dieses Problem löst“ beweisen will, benötigt man eine präzise formale Fassung des Algorithmenbegriffs. Sie hat ihren Niederschlag in der bereits 1936 aufgestellten Church'schen These gefunden, in der Algorithmen mit den auf bestimmten Maschinen, zum Beispiel auf sogenannten Turing-Maschinen, ausführbaren Programmen identifiziert werden. Das Studium des formalisierten Algorithmenbegriffs ist aber nicht das Thema dieses Buches.

Wie teilt man Algorithmen mit?
Das ist die Frage nach der sprachlichen Formulierung von Algorithmen. Wir legen Wert darauf, die Mitteilung oder Formulierung von Algorithmen deutlich von ihrer Realisierung durch ein Programm, durch einen Schaltkreis, eine mechanische Vorrichtung usw. zu trennen. Algorithmen haben eine davon unabhängige Existenz und können durchaus auf sehr verschiedene Arten mitgeteilt werden. Wir werden meistens die deutsche Umgangssprache und eine um umgangssprachliche Mittel erweiterte Pascal-ähnliche Programmiersprache benutzen.

Obwohl wir uns stets bemühen, dem Prinzip der Entwicklung von Algorithmen durch schrittweise Verfeinerung zu folgen und gut strukturierte und dokumentierte Programme, d.h. Formulierungen von Algorithmen, anzugeben, ist die Programmier*methodik*, mit der man das erreicht, ebenfalls nicht Gegenstand dieses Buches.

Welche formalen Eigenschaften von Algorithmen werden studiert?
Die wichtigste formale Eigenschaft eines Algorithmus ist zweifellos dessen Korrektheit. Dazu muß gezeigt werden, daß der Algorithmus die jeweils gestellte Aufgabe richtig löst. Man kann die Korrektheit eines Algorithmus im allgemeinen nicht durch Testen an ausgewählten Beispielen nachweisen. Denn — dies hat E. Dijkstra in einem berühmt gewordenen Satz bemerkt — man kann durch Testen zwar die Anwesenheit, nicht aber die Abwesenheit von Fehlern, also die Korrektheit eines Programmes, zeigen. Präzise oder gar voll formalisierte Korrektheitsbeweise verlangen, daß auch das durch einen Algorithmus zu lösende Problem vollständig und präzise spezifiziert ist. Da wir uns in der Regel mit einer recht informellen, inhaltlichen Problembeschreibung begnügen, verzichten wir auch auf umfangreiche, formale Korrektheitsbeweise. Wo aber die Korrektheit eines Algorithmus nicht unmittelbar offensichtlich ist, geben wir durch Kommentare, Angabe von Schleifen- und Prozedurinvarianten und andere Hinweise im Text ausreichende Hilfen, auf die der Leser selbst formalisierte Korrektheitsbeweise gründen kann.

Die zweite wichtige formale Eigenschaft eines Algorithmus, für die wir uns in diesem Buch interessieren, ist seine Effizienz. Die weitaus wichtigsten Maße für die Effizienz sind der zur Ausführung des Algorithmus benötigte Speicherplatz und die benötigte Rechenzeit. Man könnte beides durch Implementation des Algorithmus in einer konkreten Programmiersprache auf einem konkreten Rechner für eine Menge repräsentativ gewählter Eingaben messen. Solche experimentell ermittelten Meßergebnisse lassen sich aber nicht oder nur schwer auf andere Implementationen und andere Rechner übertragen.

Aus dieser Schwierigkeit bieten sich zwei mögliche Auswege an. Erstens kann man einen idealisierten Modellrechner als Referenzmaschine benutzen und die auf diesem Rechner zur Ausführung des Algorithmus benötigte Zeit und den benötigten Speicherplatz messen. Ein in der Literatur zu diesem Zweck sehr häufig benutztes Maschinenmodell ist das der (real) RAM (Random-Access-Maschine, gegebenenfalls mit Real-Zahl-Arithmetik). Eine solche Maschine verfügt über einige Register und eine abzählbar unendliche Menge einzeln adressierbarer Speicherzellen. Die Register und Speicherzellen können je eine im Prinzip unbeschränkt große ganze (oder gar reelle) Zahl aufnehmen. Das Befehlsrepertoire für eine RAM ähnelt einfachen, herkömmlichen Assemblersprachen. Neben Transportbefehlen zum Laden von und Speichern in direkt

und indirekt adressierten Speicherzellen gibt es arithmetische Befehle zur Verknüpfung zweier Registerinhalte mit den üblichen für ganze (oder reelle) Zahlen erklärten Operationen sowie bedingte und unbedingte Sprungbefehle. Die Kostenmaße Speicherplatz und Laufzeit erhalten dann folgende Bedeutung: Der von einem Algorithmus benötigte Platz ist die Anzahl der zur Ausführung benötigten RAM-Speicherzellen; die benötigte Zeit ist die Zahl der ausgeführten RAM-Befehle.

Natürlich ist die Annahme, daß Register und Speicherzellen eine im Prinzip unbeschränkt große ganze oder gar reelle Zahl enthalten können, eine idealisierte Annahme, über deren Berechtigung man in jedem Einzelfall erneut nachdenken sollte. Sofern die in einem Problem auftretenden Daten, wie etwa die zu sortierenden ganzzahligen Schlüssel im Falle des Sortierproblems, in einigen Speicherwörtern realer Rechner Platz haben, ist die Annahme wohl gerechtfertigt. Kann man die Größe der Daten aber nicht von vornherein beschränken, ist es besser, ein anderes Kostenmaß zu nehmen und die Länge der Daten explizit zu berücksichtigen. Man spricht im ersten Fall vom Einheitskostenmaß und im letzten Fall vom logarithmischen Kostenmaß. Wir werden in diesem Buch durchweg das Einheitskostenmaß verwenden. Wir messen also etwa die Größe eines Sortierproblems in der Anzahl der zu sortierenden ganzen Zahlen, nicht aber in der Summe ihrer Längen in dezimaler oder dualer Darstellung.

Wir werden in diesem Buch Algorithmen nicht als RAM-Programme formulieren und dennoch versuchen, stets die Laufzeit abzuschätzen, die sich bei Formulierung des Algorithmus als RAM-Programm (oder in der Assemblersprache eines realen Rechners) ergeben würde. Dabei geht es uns in der Regel um das *Wachstum* der Laufzeit bei wachsender Problemgröße und nicht um den genauen Wert der Laufzeit. Da es dabei auf einen konstanten Faktor nicht ankommt, ist das keineswegs so schwierig, wie es auf den ersten Blick scheinen mag.

Eine zweite Möglichkeit zur Messung der Komplexität, d.h. insbesondere der Laufzeit eines Algorithmus, besteht darin, einige die Effizienz des Algorithmus besonders charakterisierende Parameter genau zu ermitteln. So ist es beispielsweise üblich, die Laufzeit eines Verfahrens zum Sortieren einer Folge von Schlüsseln durch die Anzahl der dabei ausgeführten Vergleichsoperationen zwischen Schlüsseln und die Anzahl der ausgeführten Bewegungen von Datensätzen zu messen. Bei arithmetischen Algorithmen interessiert beispielsweise die Anzahl der ausgeführten Additionen oder Multiplikationen.

Laufzeit und Speicherbedarf eines Algorithmus hängen in der Regel von der Größe der Eingabe ab, die im Einheitskostenmaß oder logarithmischen Kostenmaß gemessen wird. Man unterscheidet zwischen dem Verhalten im besten Fall (englisch: best case), dem Verhalten im Mittel (average case) und dem Verhalten im schlechtesten Fall (worst case).

Wir können uns beispielsweise für die bei Ausführung eines Algorithmus für ein Problem der Größe N im besten bzw. im schlechtesten Fall erforderliche Laufzeit interessieren. Dazu betrachtet man sämtliche Probleme der Größe N, bestimmt die Laufzeit des Algorithmus für alle diese Probleme und nimmt dann davon das Minimum bzw. Maximum. Auf den ersten Blick scheint es viel sinnvoller zu sein, die durchschnittliche Laufzeit des Algorithmus für ein Problem der Größe N zu bestimmen, also eine Average-case-Analyse durchzuführen. Es ist aber in vielen Fällen gar nicht klar, worüber man denn den Durchschnitt bilden soll, und insbesondere die Annahme, daß etwa jedes Problem der Größe N gleichwahrscheinlich ist, ist in der Praxis oft nicht

gerechtfertigt. Hinzu kommt, daß eine Average-case-Analyse häufig technisch schwieriger durchzuführen ist als etwa eine Worst-case-Analyse. Wir werden daher in den meisten Fällen eine Worst-case-Analyse für Algorithmen durchführen. Dabei kommt es uns auf einen konstanten Faktor bei der Ermittlung der Laufzeit und auch des Speicherplatzes in Abhängigkeit von der Problemgröße N in der Regel nicht an. Wir versuchen lediglich, die Größenordnung der Laufzeit- und Speicherplatzfunktionen in Abhängigkeit von der Größe der Eingabe zu bestimmen. Um solche Größenordnungen, also Wachstumsordnungen von Funktionen auszudrücken und zu bestimmen hat sich eine besondere Notation eingebürgert, die sogenannte *Groß-Oh-* und *Groß-Omega-Notation*
.

Statt zu sagen, „für die Laufzeit $T(N)$ eines Algorithmus in Abhängigkeit von der Problemgröße N gilt für alle N: $T(N) \leq c_1 \cdot N + c_2$ mit zwei Konstanten c_1 und c_2“, sagt man „$T(N)$ ist *von der Größenordnung* N“ (oder: „$T(N)$ ist $O(N)$“, oder: „$T(N)$ ist in $O(N)$“) und schreibt: $T(N) = O(N)$ oder $T(N) \in O(N)$. Genauer definiert man für eine Funktion f die Klasse der Funktionen$O(f)$ wie folgt:

$$O(f) = \{g | \exists\, c_1 > 0 : \exists\, c_2 > 0 : \forall N \in Z^+ : g(N) \leq c_1 \cdot f(N) + c_2\}$$

Dabei werden nur Funktionen mit nichtnegativen Werten betrachtet, weil negative Laufzeiten und Speicherplatzanforderungen keinen Sinn machen.

Die üblicherweise gewählten Schreibweisen $O(N)$, $O(N^2)$, $O(N \log N)$, usw. sind insofern formal nicht ganz korrekt, als die Variable N eigentlich als gebundene Variable gekennzeichnet werden müßte. D.h. man müßte statt $O(N^2)$ beispielsweise folgendes schreiben:

$$O(f), \quad \text{mit} \quad f(N) = N^2,$$

oder unter Verwendung der λ-Notation für Funktionen

$$O(\lambda N.N^2).$$

Beispiel: Die Funktion $g(N) = 3N^2 + 6N + 7$ ist in $O(N^2)$. Denn es gilt beispielsweise mit $c_1 = 9$ und $c_2 = 7$ für alle nichtnegativen, ganzzahligen N $g(N) \leq c_1 N^2 + c_2$, also $g(N) = O(N^2)$. Man kann ganz allgemein leicht zeigen, daß das Wachstum eines Polynoms vom Grade k von der Größenordnung $O(N^k)$ ist.

Das im Zusammenhang mit der Groß-Oh-Notation benutzte Gleichheitszeichen hat *nicht* die für die Gleichheitsrelation üblicherweise geltenden Eigenschaften. So folgt beispielsweise aus $f(N) = O(N^2)$ auch $f(N) = O(N^3)$; aber natürlich ist $O(N^2) \neq O(N^3)$.

Mit Hilfe der Groß-Oh-Notation kann man also eine Abschätzung des Wachstums von Funktionen nach oben beschreiben. Zur Angabe von unteren Schranken für die Laufzeit und den Speicherbedarf von Algorithmen muß man das Wachstum von Funktionen nach unten abschätzen können. Dazu benutzt man die *Groß-Omega*-Notation und schreibt $f \in \Omega(g)$ oder $f = \Omega(g)$, um auszudrücken, daß f mindestens so stark wächst wie g. D.E. Knuth schlägt in [90] vor, die Groß-Omega-Notation präzise wie folgt zu definieren.

$$\Omega(g) = \{h | \exists\, c > 0 : \exists\, n_0 > 0 : \forall n > n_0 : h(n) \geq c \cdot g(n)\}$$

Es ist also $f \in \Omega(g)$ genau dann, wenn $g \in O(f)$ ist. Uns scheint diese Forderung zu scharf. Denn ist etwa $f(N)$ eine Funktion, die für alle geraden N den Wert 1 und für

alle ungeraden N den Wert N^2 hat, so könnte man nur $f \in \Omega(1)$ schließen, obwohl für unendlich viele N gilt $f(N) = N^2$. Man wird einem Algorithmus intuitiv einen großen Zeitbedarf zuordnen, wenn er für beliebig große Probleme diesen Bedarf hat. Um die Effizienz von Algorithmen nach unten abzuschätzen, definieren wir daher

$$\Omega(g) = \{h | \exists\, c > 0 : \exists \text{ unendlich viele } n : h(n) \geq c \cdot g(n)\}.$$

Gilt für eine Funktion f sowohl $f \in O(g)$ als auch $f \in \Omega(g)$, so schreiben wir $f = \Theta(g)$.

Die weitaus häufigsten und wichtigsten Funktionen zur Messung der Effizienz von Algorithmen in Abhängigkeit von der Problemgröße N sind folgende:

logarithmisches Wachstum: $\log N$
lineares Wachstum: N
N-log$N$-Wachstum: $N \cdot \log N$
quadratisches, kubisches,... Wachstum: N^2, N^3,...
exponentielles Wachstum: 2^N, 3^N,...

Da es uns in der Regel auf einen konstanten Faktor nicht ankommt, ist es nicht erforderlich, die Basis von Logarithmen in diesen Funktionen anzugeben. Wenn nichts anderes gesagt ist, setzen wir immer voraus, daß alle Logarithmen zur Basis 2 gewählt sind.

Es ist heute allgemeine Überzeugung, daß höchstens solche Algorithmen praktikabel sind, deren Laufzeit durch ein Polynom in der Problemgröße beschränkt bleibt. Algorithmen, die exponentielle Schrittzahl erfordern, sind schon für relativ kleine Problemgrößen nicht mehr ausführbar.

1.2 Zwei Beispiele arithmetischer Algorithmen

Wir wollen jetzt das Führen eines Korrektheitsnachweises und das Analysieren von Laufzeit und Speicherbedarf an zwei Algorithmen erläutern, die wohlbekannte arithmetische Probleme lösen. Wir behandeln zunächst ein Verfahren zur Berechnung des Produkts zweier nichtnegativer ganzer Zahlen und dann ein rekursives Verfahren zur Berechnung des Produkts zweier Polynome mit ganzzahligen Koeffizienten.

1.2.1 Ein Multiplikationsverfahren

Wendet man das aus der Schule für Zahlen in Dezimaldarstellung bekannte Verfahren zur Multiplikation auf zwei in Dualdarstellung gegebene Zahlen an, so erhält man beispielsweise für die zwei Zahlen 1101 und 101 folgendes Schema.

$$\begin{array}{r} 1\,1\,0\,1 \cdot\ 1\,0\,1 \\ \hline 1\,1\,0\,1 \\ 0\,0\,0\,0 \\ 1\,1\,0\,1 \\ \hline 1\,0\,0\,0\,0\,0\,1 \end{array}$$

Der Multiplikand 1101 wird der Reihe nach von rechts nach links mit den Ziffern des Multiplikators 101 multipliziert, wobei man das Gewicht der Ziffern durch entsprechendes Herausrücken nach links berücksichtigt. Am Schluß werden die Teilsummen aufaddiert. Das Herausrücken um eine Position nach links entspricht einem Verdopplungsschritt für in Dualdarstellung gegebene Zahlen. Statt alle Zwischenergebnisse auf einmal am Schluß aufzuaddieren, kann man sie natürlich Schritt für Schritt akkumulieren.

Nehmen wir an, daß a und b die zwei zu multiplizierenden ganzen Zahlen sind, und daß x, y und z Variablen vom Typ *integer* sind, so kann ein dieser Idee folgendes Multiplikationsverfahren durch folgendes Programmstück beschrieben werden:

```
    x := a;
    y := b;
    z := 0;
    while y > 0 do
{*}     if not odd(y)
          then
            begin
              y := y div 2;
              x := x + x
            end
          else
            begin
              y := y - 1;
              z := z + x
            end;
    {jetzt ist z = a · b}
```

Wir verfolgen die Wirkung dieses Programmstücks am selben Beispiel, d.h. für die Anfangswerte 1101 für x und 101 für y. Wir notieren in Tabelle 1.1 die Werte der Variablen in Dualdarstellung zu Beginn eines jeden Durchlaufs durch die **while**-Schleife, d.h. jedesmal, wenn die die Schleife kontrollierende Bedingung $y > 0$ überprüft wird.

x	y	z	Anzahl Schleifeniterationen
1101	101	0	0
1101	100	1101	1
11010	10	1101	2
110100	1	1101	3
110100	0	1000001	4

Tabelle 1.1

Es ist nicht schwer, in Tabelle 1.1 die gleichen Rechenschritte wiederzuerkennen, die vorher beim aus der Schule bekannten Verfahren zur Multiplikation dieser zwei Zahlen ausgeführt wurden. Ein Beweis für die Korrektheit des Verfahrens ist diese Beobachtung jedoch nicht. Dazu müssen wir vielmehr zeigen, daß für zwei beliebige nichtnegative ganze Zahlen a und b gilt, daß das Programmstück für diese Zahlen terminiert und es das Produkt der Zahlen a und b als Wert der Variablen z liefert.

Um das zu zeigen, benutzen wir eine sogenannte *Schleifeninvariante*; das ist eine den Zustand der Rechnung charakterisierende, von den Variablen abhängende Bedingung. In unserem Fall nehmen wir die Bedingung

$$P:\ y \geq 0 \quad \text{und} \quad z + x \cdot y = a \cdot b$$

und zeigen, daß die folgenden drei Behauptungen gelten.

Behauptung 1: P ist *vor* Ausführung der **while**-Schleife richtig, d.h. vor erstmaliger Ausführung der **if**-Anweisung $\{*\}$.

Behauptung 2: P bleibt bei einmaliger Ausführung der in der **while**-Schleife zu iterierenden Anweisung richtig. D.h. genauer, gelten die die **while**-Schleife kontrollierende Bedingung und die Bedingung P vor Ausführung der Anweisung $\{*\}$, so gilt nach Ausführung der Anweisung $\{*\}$ ebenfalls P.

Behauptung 3: Die in der **while**-Schleife zu iterierende **if**-Anweisung wird nur endlich oft ausgeführt. Man sagt statt dessen auch kurz, daß die **while**-Schleife terminiert.

Nehmen wir einmal an, diese drei Behauptungen seien bereits bewiesen. Dann erhalten wir die gewünschte Aussage, daß das Programmstück terminiert und am Ende $z = a \cdot b$ ist, mit den folgenden Überlegungen. Daß das Programmstück für beliebige Zahlen a und b terminiert, folgt sofort aus Behauptung 3. Wegen Behauptung 1 und Behauptung 2 muß nach der letzten Ausführung der in der **while**-Schleife zu iterierenden Anweisung $\{*\}$ P gelten und die die **while**-Schleife kontrollierende Bedingung $y > 0$ natürlich falsch sein. D.h. wir haben

$$(y \geq 0 \quad \text{und} \quad z + x \cdot y = a \cdot b) \quad \text{und} \quad (y \leq 0),$$

also $y = 0$ und damit $z = a \cdot b$ wie gewünscht.

Die Gültigkeit von Behauptung 1 ist offensichtlich. Zum Nachweis von Behauptung 2 nehmen wir an, es gelte *vor* Ausführung der **if**-Anweisung $\{*\}$

$$(y \geq 0 \quad \text{und} \quad z + x \cdot y = a \cdot b) \quad \text{und} \quad (y > 0).$$

Fall 1: [y ist gerade] Dann wird y halbiert und x verdoppelt. Es gilt also nach Ausführung der **if**-Anweisung $\{*\}$ immer noch ($y \geq 0$ und $z + x \cdot y = a \cdot b$).

Fall 2: [y ist ungerade] Dann wird y um 1 verringert und z um x erhöht und daher gilt ebenfalls nach Ausführung der **if**-Anweisung wieder ($y \geq 0$ und $z + x \cdot y = a \cdot b$).

Zum Nachweis der Behauptung 3 genügt es zu bemerken, daß bei jeder Ausführung der in der **while**-Schleife zu iterierenden **if**-Anweisung der Wert von y um mindestens 1 abnimmt. Nach höchstens y Iterationen muß also $y \leq 0$ werden und damit die Schleife terminieren. Damit ist insgesamt die Korrektheit dieses Multiplikationsalgorithmus bewiesen.

Wie effizient ist das angegebene Multiplikationsverfahren? Zunächst ist klar, daß das Verfahren nur konstanten Speicherplatz benötigt, wenn man das Einheitskostenmaß zugrundelegt, denn es werden nur drei Variablen zur Aufnahme beliebig großer ganzer Zahlen verwendet. Legt man das logarithmische Kostenmaß zugrunde, ist der Speicherplatzbedarf linear von der Summe der Längen der zu multiplizierenden Zahlen abhängig. Es macht wenig Sinn, zur Messung der Laufzeit das Einheitskostenmaß zugrunde zu legen. Denn in diesem Maß gemessen ist die Problemgröße konstant gleich 2. Wir interessieren uns daher für die Anzahl der ausgeführten arithmetischen Operationen in Abhängigkeit von der Länge und damit der Größe der zwei zu multiplizierenden Zahlen a und b.

Beim Korrektheitsbeweis haben wir uns insbesondere davon überzeugt, daß die in der **while**-Schleife iterierte **if**-Anweisung höchstens y-mal ausgeführt werden kann, mit $y = b$. Das ist eine sehr grobe Abschätzung, die allerdings sofort zu einer Schranke in $O(b)$ für die zur Berechnung des Produkts $a \cdot b$ ausgeführten Divisionen durch 2, Additionen und Subtraktionen führt. Eine genaue Analyse zeigt, daß die Zahl y, die anfangs den Wert b hat, genau einmal weniger halbiert wird, als ihre Länge angibt; eine Verringerung um 1 erfolgt gerade so oft, wie die Anzahl der Einsen in der Dualdarstellung von y angibt.

Nehmen wir an, daß alle Zahlen in Dualdarstellung vorliegen, so ist die Division durch 2 nichts anderes als eine Verschiebe- oder Shift-Operation um eine Position nach rechts, wobei die am weitesten rechts stehende Ziffer (eine 0) verlorengeht; die Verdoppelung von x entspricht einer Shift-Operation um eine Position nach links, wobei eine 0 als neue am weitesten rechts stehende Ziffer nachgezogen wird. Das Verkleinern der ungeraden Zahl y um 1 bedeutet, die Endziffer 1 in eine 0 zu verwandeln. Es ist realistisch anzunehmen, daß alle diese Operationen in konstanter Zeit ausführbar sind. Nimmt man an, daß auch die Anweisung $z := z + x$ in konstanter Zeit ausgeführt werden kann, ergibt sich eine Gesamtlaufzeit des Verfahrens von der Größenordnung $O(\text{Länge}(b))$. Legt man die vielleicht realistischere Annahme zugrunde, daß die Berechnung der Summe $z + x$ in der Zeit $O(\text{Länge}(z) + \text{Länge}(x))$ ausführbar ist, ergibt sich eine Gesamtlaufzeit von der Größenordnung $O(\text{Länge}(b) \cdot (\text{Länge}(a) + \text{Länge}(b)))$.

1.2.2 Polynomprodukt

Ein ganzzahliges Polynom vom Grade $N-1$ kann man sich gegeben denken durch die N ganzzahligen Koeffizienten $a_0, \ldots, a_{N-1}$. Es hat die Form

$$p(x) = a_0 + a_1 x^1 + \ldots + a_{N-1} x^{N-1}.$$

Wir lassen zunächst offen, wie ein solches Polynom programmtechnisch realisiert wird. Seien nun zwei Polynome $p(x)$ und $q(x)$ vom Grade $N-1$ gegeben; $p(x)$ wie oben angegeben und

$$q(x) = b_0 + b_1 x^1 + \ldots + b_{N-1} x^{N-1}.$$

Wie kann man das Produkt der beiden Polynome $r(x) = p(x) \cdot q(x)$ berechnen? Bereits in der Schule lernt man, daß das Produktpolynom ein Polynom vom Grade $2N-2$ ist, das man erhält, wenn man jeden Term $a_i x^i$ des Polynoms p mit jedem Term $b_j x^j$ des

Polynoms q multipliziert und dann die Terme mit gleichem Exponenten sammelt. Es ist leicht, eine Implementation dieses sogenannten naiven Verfahrens anzugeben, wenn man voraussetzt, daß die Polynome durch Arrays realisiert werden, die die Koeffizienten enthalten. Setzen wir die Deklarationen

```
var
p, q: array [0 .. N − 1] of integer;
   r: array [0 .. 2N − 2] of integer
```

voraus, so kann das Produktpolynom durch eine doppelt geschachtelte **for**-Schleife wie folgt berechnet werden.

```
for i := 0 to 2N − 2 do r[i] := 0;
for i := 0 to N − 1 do
   for j := 0 to N − 1 do r[i + j] := r[i + j] + p[i] ∗ q[j]
```

Diese Darstellung zeigt unmittelbar, daß zur Berechnung der Koeffizienten des Produktpolynoms genau N^2 Koeffizientenprodukte berechnet werden.

Wir wollen jetzt ein anderes Verfahren zur Berechnung des Produktpolynoms angeben, das mit weniger als N^2 Koeffizientenproduktberechnungen auskommt. Das Verfahren folgt der Divide-and-conquer-Strategie, die ein sehr allgemeines und mächtiges Prinzip zur algorithmischen Lösung von Problemen darstellt. Es kann als Problemlösungsschema wie folgt formuliert werden.

Divide-and-conquer-Verfahren zur Lösung eines Problems der Größe N

1. *Divide: Teile das Problem der Größe N in (wenigstens) zwei annähernd gleich große Teilprobleme, wenn $N > 1$ ist; sonst löse das Problem der Größe 1 direkt.*
2. *Conquer: Löse die Teilprobleme auf dieselbe Art (rekursiv).*
3. *Merge: Füge die Teillösungen zur Gesamtlösung zusammen.*

Um dieses Prinzip auf das Problem, das Produkt zweier Polynome zu berechnen, einfach anwenden zu können, nehmen wir an, daß die Koeffizientenzahl N beider Polynome p und q eine Potenz von 2 ist. Dann kann man schreiben

$$p(x) = p_l(x) + x^{\frac{N}{2}} p_r(x)$$

mit

$$\begin{aligned} p_l(x) &= a_0 + a_1 x^1 + \cdots + a_{\frac{N}{2}-1} x^{\frac{N}{2}-1} \\ p_r(x) &= a_{\frac{N}{2}} + a_{\frac{N}{2}+1} x^1 + \cdots + a_{N-1} x^{\frac{N}{2}-1}. \end{aligned}$$

Ebenso kann man auch schreiben

$$q(x) = q_l(x) + x^{\frac{N}{2}} q_r(x)$$

mit zwei analog definierten Polynomen $q_l(x)$ und $q_r(x)$ vom Grade $\frac{N}{2}-1$.
Dann ist

$$\begin{aligned} r(x) &= p(x)q(x) \\ &= p_l(x)q_l(x) + (p_l(x)q_r(x) + p_r(x)q_l(x))\,x^{\frac{N}{2}} + p_r(x)q_r(x)x^N. \end{aligned}$$

Wir haben also das Problem, das Produkt zweier Polynome (vom Grade $N-1$) mit jeweils N Koeffizienten zu berechnen, zerlegt in das Problem, vier Produkte von Polynomen mit jeweils $N/2$ Koeffizienten zu berechnen: Das sind die Produkte p_lq_l, p_lq_r, p_rq_l, p_rq_r. Wie man daraus die Koeffizienten des Produktpolynoms $r(x)$ erhält, ohne daß weitere Koeffizientenprodukte berechnet werden müssen, ist ebenfalls aus der oben angegebenen Gleichung für $r(x)$ abzulesen. Wir wollen die Anzahl der Multiplikationen von Koeffizienten, die ausgeführt werden, wenn man zwei Polynome vom Grade $N-1$ mit N Koeffizienten miteinander multipliziert, mit $M(N)$ bezeichnen. Ein dem Divide-and-conquer-Prinzip folgender Algorithmus zur Berechnung des Produktpolynoms, der auf der oben angegebenen Zerlegung des Problems in vier Teilprobleme halber Größe beruht, führt also eine Anzahl von Koeffizientenproduktberechnungen durch, die durch folgende Rekursionsformel beschrieben werden kann.

$$M(N) = 4 \cdot M\left(\frac{N}{2}\right)$$

Natürlich ist

$$M(1) = 1.$$

Weil wir angenommen hatten, daß N eine Potenz von 2 ist, also $N = 2^k$ für ein $k \geq 0$, erhält man als Lösung dieser Rekursionsgleichung sofort

$$M(N) = 4^k = (2^2)^k = (2^k)^2 = N^2.$$

Ein auf der oben angegebenen Zerlegung gegründetes Divide-and-conquer-Verfahren liefert also keine Verbesserung gegenüber dem naiven Verfahren. Es ist aber nicht schwer, eine andere Zerlegung des Problems anzugeben, so daß ein auf dieser Zerlegung gegründetes Divide-and-conquer-Verfahren mit weniger Koeffizientenproduktberechnungen auskommt. Wir setzen

$$\begin{aligned} z_l(x) &= p_l(x)q_l(x), \\ z_r(x) &= p_r(x)q_r(x), \end{aligned}$$

und

$$z_m(x) = (p_l(x) + p_r(x))\,(q_l(x) + q_r(x))\,.$$

Dann ist

$$p(x)q(x) = z_l(x) + (z_m(x) - z_l(x) - z_r(x))\,x^{\frac{N}{2}} + z_r(x)x^N.$$

Ein auf dieser Zerlegung gegründetes Divide-and-conquer-Verfahren zur Berechnung des Produkts zweier Polynome mit N Koeffizienten kann also wie folgt formuliert werden.

Verfahren zur Multiplikation zweier Polynome $p(x)$ und $q(x)$ mit N Koeffizienten:

Falls $N = 1$ ist, berechne das Produkt der beiden Koeffizienten; sonst:

1. *Divide: Zerlege die Polynome $p(x)$ und $q(x)$ in der Form $p(x) = p_l(x) + p_r(x) \cdot x^{\frac{N}{2}}$ und $q(x) = q_l(x) + q_r(x) \cdot x^{\frac{N}{2}}$, setze $p_m(x) = p_l(x) + p_r(x)$ und $q_m(x) = q_l(x) + q_r(x)$.*
2. *Conquer: Wende das Verfahren (rekursiv) an, um die folgenden Polynomprodukte zu berechnen:*

$$z_l(x) = p_l(x)q_l(x),\ z_m(x) = p_m(x)q_m(x),\ z_r(x) = p_r(x)q_r(x)$$

3. *Merge: Setze $p(x)q(x) = z_l(x) + (z_m(x) - z_l(x) - z_r(x))x^{\frac{N}{2}} + z_r(x)x^N$.*

Offenbar sind p_l, p_m, p_r und q_l, q_m, q_r Polynome mit $N/2$ Koeffizienten. Da außer im Fall $N = 1$ keinerlei Koeffizientenprodukte berechnet werden müssen, erhält man für die Anzahl der nach diesem Verfahren berechneten Koeffizientenprodukte die Rekursionsformel

$$M(1) = 1$$

und

$$M(N) = 3 \cdot M\left(\frac{N}{2}\right).$$

Für $N = 2^k$ hat diese Formel die Lösung

$$M(N) = 3^k = 2^{(\log 3)k} = 2^{k(\log 3)} = N^{\log 3} = N^{1.58\ldots}.$$

Wir haben also eine Verbesserung gegenüber dem naiven Verfahren erreicht.

Das angegebene Verfahren ist nicht das beste bekannte Verfahren zur Berechnung des Produkts zweier Polynome in dem Sinne, daß die Anzahl der ausgeführten Koeffizientenproduktberechnungen möglichst klein wird. Es zeigt aber die Anwendbarkeit des Divide-and-conquer-Prinzips sehr schön und ebenso, wie man nach dieser Strategie entworfene Algorithmen analysiert, nämlich durch Aufstellen und Lösen einer Rekursionsgleichung. Wir werden in diesem Buch noch zahlreiche weitere Beispiele für dieses Prinzip bringen.

Wir nennen an dieser Stelle nur einige weitere Probleme, die auf diese Weise gelöst und analysiert werden können, ohne daß wir dabei auf irgendwelche Details eingehen. Es sind die Multiplikation langer ganzer Zahlen, die Multiplikation zweier $N \times N$-Matrizen nach der Methode von Strassen [177], binäres Suchen (vgl. dazu Kapitel 3), die Sortierverfahren Quicksort, Heapsort, Mergesort (vgl. dazu Kapitel 2) und Verfahren aus der Geometrie, zum Beispiel zur Berechnung aller Schnitte von Liniensegmenten in der Ebene (vgl. dazu Kapitel 7).

1.3 Verschiedene Algorithmen für dasselbe Problem

Wie am Beispiel des Polynomprodukts im vorigen Abschnitt bereits gezeigt wurde, kann man dasselbe Problem durchaus mit verschiedenen Algorithmen lösen. Das Ziel ist natürlich, den für ein Problem besten Algorithmus zu finden und zu implementieren. Das verlangt insbesondere eine möglichst optimale Nutzung der Ressourcen Speicherplatz und Rechenzeit. Wie wichtig die richtige Algorithmenwahl zur Lösung eines Problems sein kann, zeigt ein von Jon Bentley behandeltes Problem [16], das wir in diesem Abschnitt genauer diskutieren wollen. Es handelt sich um das *Maximum-Subarray-Problem*.

Gegeben sei eine Folge X von N ganzen Zahlen in einem Array. Gesucht ist die maximale Summe aller Elemente in einer zusammenhängenden Teilfolge. Sie wird als maximale Teilsumme bezeichnet, jede solche Folge als maximale Teilfolge.

Für die Eingabefolge $X[1\,..\,10]$

$$31,\ -41,\ 59,\ 26,\ -53,\ 58,\ 97,\ -93,\ -23,\ 84$$

ist die Summe der Teilfolge $X[3\,..\,7]$ mit Wert 187 die Lösung des Problems. Eine Variante dieses Problems wurde übrigens im Rahmen des 4. Bundeswettbewerbs Informatik 1985 als Aufgabe gestellt (Aktienkurs-Analyse [77], vgl. Aufgabe 1.5).

Ein sofort einsichtiges, *naives* Verfahren zur Lösung des Problems benutzt drei ineinandergeschachtelte **for**-Schleifen, um die maximale Teilsumme als Wert der Variablen *maxtsumme* zu berechnen.

```
maxtsumme := 0;
for u := 1 to N do
  for o := u to N do
    begin
      {bestimme die Summe der Elemente in der Teilfolge X[u .. o]}
      Summe := 0;
      for i := u to o do Summe := Summe + X[i];
      {bestimme den größeren der beiden Werte Summe
       und maxtsumme}
      maxtsumme := max(Summe, maxtsumme)
    end
```

Die Lösung ist einfach, aber ineffizient, denn sie benötigt für eine Folge der Länge N offenbar

$$\sum_{u=1}^{N}\sum_{o=u}^{N}\sum_{i=u}^{o} 1 = \Theta(N^3)$$

Schritte, d.h. genauer Zuweisungen, Additionen und Maximumbildungen.

Jetzt folgen wir dem Divide-and-conquer-Prinzip zur Lösung des Maximum-Subarray-Problems. Die Anwendbarkeit dieses Prinzips ergibt sich aus folgender Überlegung.

Wird eine gegebene Folge in der Mitte geteilt, so liegt die maximale Teilfolge entweder ganz in einem der beiden Teile oder sie umfaßt die Trennstelle, liegt also teils im linken und teils im rechten Teil. Im letzteren Fall gilt für das in einem Teil liegende Stück der maximalen Teilfolge: Die Summe der Elemente ist maximal unter allen zusammenhängenden Teilfolgen in diesem Teil, die das Randelement an der Trennstelle enthalten.

Wir wollen die maximale Summe von Elementen, die das linke bzw. das rechte Randelement einer Folge von Elementen enthält, kurz das linke bzw. rechte Randmaximum nennen. Das linke Randmaximum *lmax* für eine Folge $X[l], \ldots, X[r]$ ganzer Zahlen kann man in $\Theta(r-l)$ Schritten wie folgt bestimmen.

```
lmax := 0;
summe := 0;
for i := l to r do
  begin
    summe := summe + X[i];
    lmax := max(lmax, summe)
  end
```

Entsprechend kann man auch das rechte Randmaximum *rmax* für eine Folge ganzer Zahlen in einer Anzahl von Schritten bestimmen, die linear mit der Anzahl der Folgenelemente wächst. Das dem Divide-and-conquer-Prinzip folgende Verfahren zur Berechnung der maximalen Teilsumme in einer Folge X ganzer Zahlen kann nun wie folgt formuliert werden.

```
Algorithmus maxtsum (X);
{liefert eine maximale Teilsumme der Folge X ganzer Zahlen}
begin
  if X enthält nur ein Element a
    then
      if a > 0
        then maxtsum := a
        else maxtsum := 0
    else
      begin
        {Divide:}
        teile X in eine linke und eine rechte Teilfolge A und B
          annähernd gleicher Größe;
        {Conquer:}
        maxtinA := maxtsum(A);
        maxtinB := maxtsum(B);
        bestimme das rechte Randmaximum rmax(A) der
          linken Teilfolge A;
        bestimme das linke Randmaximum lmax(B) der
          rechten Teilfolge B;
        {Merge:}
        maxtsum := max(maxtinA, maxtinB, rmax(A) + lmax(B))
      end
end {maxtsum}
```

Bezeichnet nun $T(N)$ die Anzahl der Schritte, die erforderlich ist, um den Algorithmus *maxtsum* für eine Folge der Länge N auszuführen, so gilt offenbar folgende Rekursionsformel:

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + \text{Const} \cdot N$$

Da natürlich $T(1)$ konstant ist, erhält man als Lösung dieser Gleichung und damit als asymptotische Laufzeit des Verfahrens

$$T(N) = \Theta(N \log N).$$

Das ist schon viel besser als die Laufzeit des naiven Verfahrens. Aber ist es bereits das bestmögliche Verfahren? Nein — denn die Anwendung eines weiteren algorithmischen Lösungsprinzips, des Scan-line-Prinzips, liefert uns ein noch besseres Verfahren. Wir haben eine aufsteigend sortierte, lineare Folge von Inspektionsstellen (oder: Ereignispunkten), die Positionen $1, \ldots, N$ der Eingabefolge. Wir durchlaufen die Eingabe in der durch die Inspektionsstellen vorgegebenen Reihenfolge und führen zugleich eine vom jeweiligen Problem abhängige, dynamisch veränderliche, d.h. an jeder Inspektionsstelle gegebenenfalls zu korrigierende Information mit. In unserem Fall ist das die maximale Summe *bisMax* einer Teilfolge im gesamten bisher inspizierten Anfangsstück und das an der Inspektionsstelle endende rechte Randmaximum *ScanMax* des bisher inspizierten Anfangsstücks. Das ist in Abbildung 1.1 dargestellt.

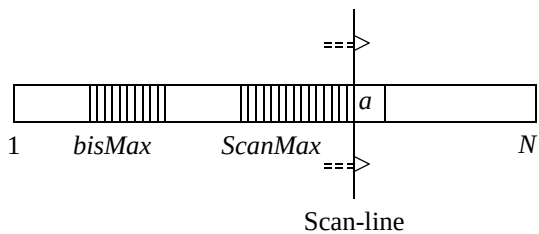


Abbildung 1.1: Scan–Line

Nehmen wir nun an, daß wir bereits ein Anfangsstück der Länge l der gegebenen Folge inspiziert haben und die maximale Teilsumme *bisMax* sowie das rechte Randmaximum *ScanMax* in diesem Anfangsstück kennen. Was ist die maximale Teilsumme, wenn man das $(l+1)$-te Element, sagen wir a, hinzunimmt? Die maximale Teilfolge des neuen Anfangsstücks der Länge $l+1$ liegt entweder bereits im Anfangsstück der Länge l, oder sie enthält das neu hinzugenommene Element a, reicht also bis zum rechten Rand. Das rechte Randmaximum der neuen Folge mit $l+1$ Elementen erhält man nun aus dem rechten Randmaximum der Folge durch Hinzunahme von a, also aus dem alten Wert von *ScanMax*, indem man a hinzuaddiert, vorausgesetzt, daß dieser Wert insgesamt positiv bleibt. Ist das nicht der Fall, so ist die maximale Summe von Elementen, die das rechte Randelement enthält, die Summe der Elemente der leeren Folge, also 0. Damit erhält man folgendes Verfahren, das hier etwas allgemeiner beschrieben ist, als es für das behandelte Problem nötig wäre.

```
Q := Folge der Inspektionsstellen von links nach rechts;
        {= Folge der Positionen 1,...,N}
{Initialisiere}
ScanMax := 0;
bisMax := 0;
while Q noch nicht erschöpft do
  begin
    q := nächstes Element von Q;
    a := das Element an Position q;
    {update ScanMax und bisMax}
    if ScanMax + a > 0
      then ScanMax := ScanMax + a
      else ScanMax := 0;
    bisMax := max(bisMax, ScanMax)
  end
```

Am Ende enthält dann *bisMax* den gewünschten Wert.

Dies ist ein Algorithmus, der in linearer Zeit ausführbar ist. Denn an jeder der N Inspektionsstellen müssen nur konstant viele Schritte (u.a. zum Update von *ScanMax* und *bisMax*) und damit insgesamt nur $\Theta(N)$ Schritte ausgeführt werden. Das ist asymptotisch optimal. Es gibt keinen Algorithmus zur Bestimmung der maximalen Teilsumme einer Folge von N Elementen, der für beliebig viele N mit weniger als $c \cdot N$ Schritten, für eine positive Konstante c, auskommt. Der Grund ist, daß zur Bestimmung der maximalen Teilfolge offensichtlich alle Folgenelemente wenigstens einmal betrachtet werden müssen. Das sind aber bereits N Schritte.

1.4 Die richtige Wahl einer Datenstruktur

Die beiden ersten der im vorigen Abschnitt angegebenen drei verschiedenen Algorithmen zur Lösung des Maximum-Subarray-Problems haben vorausgesetzt, daß die Folge der ganzen Zahlen, für die die maximale Teilsumme ermittelt werden sollte, in einem Array gegeben ist. Wir haben die gleiche Datenstruktur zur Implementation verschiedener Verfahren benutzt.

Bereits im täglichen Leben machen wir aber die Erfahrung, daß die richtige Organisationsform für eine Menge von Daten und damit die richtige Datenstrukturwahl ganz erheblichen Einfluß darauf hat, wie effizient sich bestimmte Operationen für die Daten ausführen lassen. Denken wir etwa an ein Telefonbuch: Es ist leicht, zu einem gegebenen Namen die zugehörige Telefonnummer zu finden; für die umgekehrte Aufgabe ist aber das bei Telefonbüchern übliche Gliederungsprinzip, zunächst nach Orten und innerhalb eines Ortes nach Namen alphabetisch sortiert, wenig geeignet. Da der normale Telefonbenutzer aber höchst selten den zu einer Telefonnummer gehörigen Namen sucht, lohnt es sich nicht, etwa nach Nummern aufsteigend sortierte Telefonbücher an die Telefonkunden auszugeben.

Nicht immer ist die richtige Wahl einer Datenstruktur so einfach. Es gibt viele Fälle, in denen es keineswegs auf der Hand liegt, welche Organisationsform für eine Menge von Daten zu wählen ist, um bestimmte Operationen auf der Datenmenge effizient ausführen zu können. Wir geben ein Beispiel, das als *Post-office-Problem* bekannt ist: Für eine gegebene, als fest vorausgesetzte Menge M von Orten (mit Postämtern) und für einen beliebig gegebenen Ort p, der in der Regel nicht zu M gehört (also kein Postamt hat), soll festgestellt werden, welches der dem Ort p nächstgelegene Ort aus M ist. Wie kann man die Menge M strukturieren, um derartige Anfragen, sogenannte *Nearest-neighbor-queries*, möglichst effizient ausführen zu können? Eine alphabetische Reihenfolge der Orte hilft offenbar wenig. Auf den ersten Blick scheint nichts anderes übrig zu bleiben, als für einen gegebenen Ort p wie folgt vorzugehen. Man betrachtet der Reihe nach jeden Ort $q \in M$ und berechnet die Distanz $d(p,q)$ zwischen p und q. Schließlich stellt man fest, für welches q die Distanz $d(p,q)$ minimalen Wert hat. Es ist offensichtlich, daß der Aufwand zur Beantwortung einer derartigen Nachbarschaftsanfrage wenigstens linear mit der Anzahl der Orte in M wächst, wenn man so vorgeht. Kann man es besser machen? Die Idee liegt nahe, die Orte aus M zunächst in eine Landkarte einzutragen und dann für einen gegebenen Ort p nachzusehen, welchem Ort aus M p am nächsten liegt. Die Landkarte mit den darin eingetragenen Orten aus M ist also eine Datenstruktur für M, die Anfragen nach nächsten Nachbarn besser unterstützt. Wir idealisieren und präzisieren diese Idee noch weiter und nehmen an, daß der Abstand zwischen je zwei Orten die gewöhnliche, euklidische Distanz ist. Sind p und q Punkte mit reellwertigen Koordinaten $p = (p_x, p_y)$ und $q = (q_x, q_y)$ in einem kartesischen Koordinatensystem, so sei also die Distanz $d(p,q)$ zwischen p und q definiert durch

$$d(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Dann kann man die euklidische Ebene für eine gegebene Menge M von Punkten in Gebiete gleicher nächster Nachbarn einteilen. Jedem Punkt $p \in M$ ordnet man ein Gebiet $VR(p)$ der Ebene zu, das genau alle Punkte enthält, deren Distanz zu p geringer ist als zu allen anderen Punkten aus M. Auf diese Weise erhält man für jede (feste) Menge M von Punkten eine vollständige Aufteilung der Ebene in disjunkte Gebiete, die sich höchstens an den Rändern berühren. Abbildung 1.2 zeigt ein Beispiel einer derartigen Struktur für eine Menge von 16 Punkten.

Man nennt eine solche Einteilung der Ebene das zur Menge M gehörende *Voronoi-Diagramm* $VD(M)$ und die einem Punkt $p \in M$ zugeordnete Region $VR(p)$ die *Voronoi-Region* von p. Für eine genaue Definition von $VD(M)$, für Algorithmen zur Konstruktion von $VD(M)$ und für die Möglichkeit zur Speicherung von $VD(M)$ verweisen wir auf Kapitel 7. Es ist bereits jetzt klar, wie man zu einem gegebenen Punkt p den nächsten Nachbarn von q in M finden kann: Man bestimmt die Voronoi-Region, in die p fällt. Ist $p \in VR(q)$, so ist q nächster Nachbar von p. Man kann zeigen, daß die Region $VR(q)$, in die p fällt, in $O(\log N)$ Schritten bestimmt werden kann, wenn N die Gesamtzahl der Punkte in der gegebenen Menge M ist. Das Voronoi-Diagramm, auf Papier gezeichnet oder mit den Mitteln einer Programmiersprache beschrieben und im Rechner geeignet gespeichert, ist also eine Datenstruktur, die Nearest-neighbor-queries gut unterstützt.

Die Frage nach der richtigen Datenstruktur kann man also genauer so formulieren: Gegeben sei eine Menge von Daten und eine Folge von Operationen mit diesen Daten; man finde eine Speicherungsform für die Daten und Algorithmen für die auszufüh-

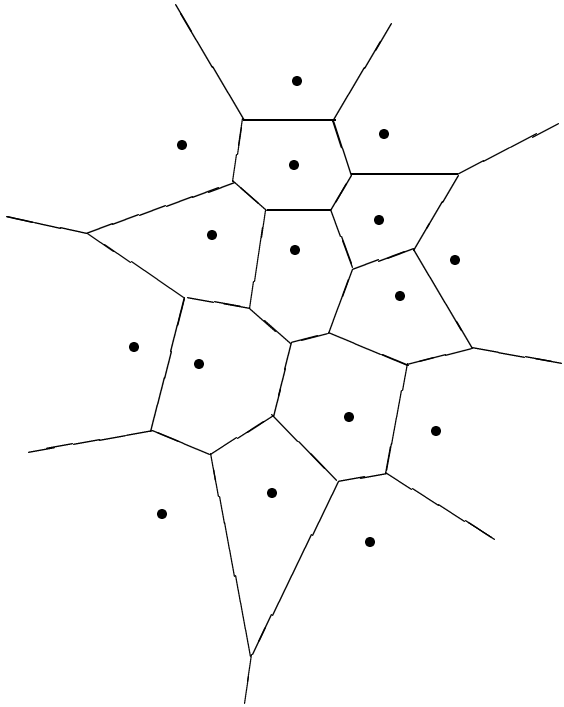


Abbildung 1.2

renden Operationen so, daß die Operationen der gegebenen Folge möglichst effizient ausführbar sind. Auch in dieser Formulierung sind noch viele für die richtige Wahl wesentliche Parameter offengelassen: Ist die Folge der Operationen vorher bekannt? Wenn nicht, kennt man dann wenigstens die (relativen) Häufigkeiten der verschiedenen in der Folge auftretenden Operationen? Kommt es bei der Effizienz in erster Linie auf die Ausführungszeit, auf den Speicherbedarf, auf die leichte Programmierbarkeit, usw. an? Auf jeden Fall dürfte klar sein, daß man die richtige Speicherungsform für eine Menge von Daten nicht unabhängig davon wählen kann, welche Operationen mit welcher Häufigkeit mit den Daten ausgeführt werden.

Daten und Operationen mit den Daten gehören also zusammen. Es ist heute üblich geworden, sie als Einheit aufzufassen und von *abstrakten Datentypen* (ADT) zu sprechen: Ein ADT besteht aus einer oder mehreren Mengen von Objekten und darauf definierten Operationen, die mit in der Mathematik üblichen Methoden spezifiziert werden können. Wir geben einige Beispiele an, zuerst den ADT *Polynom*. Die Menge der Objekte ist die Menge der Polynome mit ganzzahligen Koeffizienten. Die Menge der Operationen enthält genau die Addition und Multiplikation zweier Polynome. Nimmt man zur Menge der Operationen weitere hinzu, z.B. die erste Ableitung eines Polynoms (die etwa für ein Polynom $p(x) = 3x^3 + 6x - 7$ das Polynom $p'(x) = 9x^2 + 6$ liefert), so erhält man

einen anderen als den oben angegebenen ADT. In beiden Fällen hat man nur eine Sorte von Objekten. Das ist eher die Ausnahme.

Meistens hat man mehrere verschiedene Mengen von Objekten, und die Operationen sind nicht nur auf Objekte einer Sorte beschränkt, wie in dem oben schon diskutierten Beispiel einer Menge von Punkten, für die Nearest-neighbor-queries beantwortet werden sollen. Als ADT *Punktmenge* kann man dieses Beispiel folgendermaßen beschreiben. Eine erste Menge von Objekten ist die Klasse aller endlichen Mengen von Punkten in der Ebene; eine weitere Menge von Objekten ist die Menge aller Punkte der Ebene. Die Operation *nächster Nachbar* ordnet einer Menge M von Punkten und einem Punkt p einen Punkt aus M zu. Weitere Beispiele für Operationen auf diesen Objektmengen sind die Operation des Einfügens eines Punktes in eine Menge und das Entfernen eines Punktes aus einer Menge. Sie liefern als Ergebnis wieder eine Menge von Punkten. Will man auch noch zu je zwei Punkten die euklidische Distanz ermitteln können, muß man die Menge der reellen Zahlen als weitere Objektmenge und die oben definierte Distanzfunktion als weitere Operation hinzunehmen.

Die zur Definition eines ADT benutzten Objektmengen und Operationen werden, wie in der Mathematik üblich, ohne Rücksicht auf ihre programmtechnische Realisierung spezifiziert. Die zwei wichtigsten Methoden sind die *konstruktive* und die *axiomatische* Methode.

Bei der *konstruktiven Methode* geht man von bekannten mathematischen Modellen aus und konstruiert daraus neue; die jeweils benötigten Operationen werden explizit oder implizit mit Hilfe schon bekannter definiert. So kann man beispielsweise Punkte in der euklidischen Ebene als Paare reeller Zahlen auffassen und die Operation „nächster Nachbar“ auf bekannte Operationen für reelle Zahlen zurückführen. Gemeint sind hier natürlich die reellen Zahlen als Objekte der Mathematik und nicht ihre Realisierung als Daten vom Typ *real* in einer konkreten Programmiersprache auf einem konkreten Rechner.

Bei der *axiomatischen Methode* werden die Objektmengen nur implizit durch die Angabe von Axiomen für die mit den Objekten auszuführenden Operationen festgelegt. Das geschieht ganz analog etwa zur üblichen Definition einer Gruppe in der Mathematik: Eine Menge G zusammen mit einer auf G definierten Verknüpfungsoperation heißt Gruppe, wenn für die Elemente von G und die Verknüpfungsoperation die üblichen Gruppenaxiome gelten.

Es ist möglich, Algorithmen so zu formulieren, daß man nur auf Objekte und Operationen abstrakter Datentypen zurückgreift. Wir geben dafür ein Beispiel und formulieren einen Algorithmus, der zu einer gegebenen, endlichen Menge M von Punkten ein Paar (p,q) von zwei verschiedenen Punkten aus M liefert, dessen (euklidische) Distanz minimal unter allen Distanzen von Punkten aus M ist. Dabei setzen wir einen ADT „Punktmenge“ voraus, für den insbesondere die Operationen „nächster Nachbar“ und „Distanz zweier Punkte“ definiert sind.

Algorithmus *Nearest-neighbors* (M);
$\{$*liefert ein Paar* (p_0, q_0) *von Punkten aus M mit minimaler euklidischer Distanz*$\}$

Fall 1: $[M = \emptyset$ *oder M enthält nur einen Punkt*$]$ *Dann ist das Paar nächster Nachbarn nicht definiert.*

Fall 2: $[M$ *enthält wenigstens zwei verschiedene Punkte p und q*$]$

(a) *Wähle zwei verschiedene Punkte* p_0 *und* q_0 *aus M und berechne ihre Distanz dist.*

(b) *Bestimme für jeden Punkt p aus M den nächsten Nachbarn q von p in* $M \backslash \{p\}$*; berechne die Distanz* $d(p,q)$ *der Punkte p und q; falls* $d(p,q) < dist$*, setze* $p_0 := p$, $q_0 := q$, $dist := d(p,q)$.

Man sieht in dieser Formulierung, daß auch einige weitere Operationen für eine Punktmenge M (nicht nur „nächster Nachbar“ und „Distanz zweier Punkte“) ausführbar sein müssen: Es muß möglich sein, festzustellen, ob $M = \emptyset$ ist oder ob M nur einen Punkt enthält; ferner muß es möglich sein, einen Punkt aus M auszuwählen und aus M zu entfernen. Es werden jedoch keinerlei implementationsabhängige Details für Punktmengen benötigt.

Soll der Algorithmus in einer konkreten Programmiersprache implementiert werden, ist es nötig, den ADT Punktmenge durch Angabe von Datenstrukturen für die Objektmengen und Algorithmen für die benutzten Operationen zu realisieren. Dazu müssen wir sie auf die in der jeweils benutzten Sprache vorhandenen Datentypen und Grundoperationen zurückführen. Denn die Programmiersprache besitzt in der Regel keine Datentypen und Operationen für Variablen des entsprechenden Typs, die man direkt zur Implementation des abstrakten Datentyps nehmen könnte. Ist die Programmiersprache beispielsweise die Sprache Pascal, so kann man Punktmengen ausgehend vom Grundtyp *real* und unter Benutzung der in Pascal vorhandenen Möglichkeiten zur Definition strukturierter Datentypen definieren. Da der **set**-Typ in Pascal nur die Zusammenfassung einer Menge von Objekten eines einfachen Typs außer *real* erlaubt, kann man diese Strukturierungsmethode nicht nehmen, um Punktmengen in Pascal zu realisieren. Eine Möglichkeit ist beispielsweise, die Punkte einer Menge als Elemente eines Arrays passender maximaler Größe zu vereinbaren.

```
const
  maxZahl = {passend gewählte Zahl};
type
  Punkt = record
            xcoord, ycoord: real
          end;
  Punktmenge = record
                 elementzahl: integer;
                 element: array [1 .. maxZahl] of Punkt
               end
```

Eine Punktmenge *M* ist dann nichts anderes als eine Variable vom oben vereinbarten Typ. Es ist nicht schwer, alle zur Formulierung des Algorithmus *Nearest-neighbors* benutzten Operationen als Funktionen und Prozeduren zu formulieren, die diese Datenstruktur benutzen. Wir geben ein einfaches Beispiel.

```
function empty (M: Punktmenge) : boolean;
{liefert true genau dann, wenn M die leere Menge ist}
begin
  empty := (0 = M.elementzahl)
end
```

Eine Realisierung des ADT Punktmenge als Voronoi-Diagramm ist nicht so offensichtlich, weil nicht klar ist, wie diese Struktur mit den Mitteln einer Programmiersprache, wie z.B. Pascal, beschrieben werden kann (vgl. hierzu Kapitel 7).

Wir unterscheiden also zwischen Datentypen, abstrakten Datentypen und Datenstrukturen. *Datentypen* sind die in Programmiersprachen üblicherweise vorhandenen Grundtypen, wie *integer, real, boolean, character,* und die daraus mit den jeweils vorhandenen Strukturierungsmethoden, wie **record, array, set, file**, gebildeten zusammengesetzten Typen. Ein Datentyp legt die Menge der möglichen Werte und die zulässigen Operationen mit Variablen dieses Typs fest.

Ein *abstrakter Datentyp* ist das Analogon zu einer mathematischen Theorie. Er besteht aus einer oder mehreren, mit üblichen mathematischen Methoden festgelegten Mengen von Objekten und darauf definierten Operationen.

Eine *Datenstruktur* ist eine Realisierung der Objektmengen eines ADT mit den Mitteln einer Programmiersprache, z.B. als Kollektion von Variablen verschiedener Datentypen. Man geht häufig nicht ganz bis auf die programmiersprachliche Ebene hinunter und beschreibt eine Datenstruktur nur soweit, daß die endgültige Festlegung mit Mitteln einer Programmiersprache nicht mehr schwierig ist. Man kann eine Datenstruktur auch als Speicherstruktur auffassen, nämlich als Abbild der im mathematischen Sinne idealen Objektmengen eines ADT im Speicher eines realen Rechners. Zur Realisierung oder, wie man auch sagt, zur *Implementierung* eines ADT gehört aber nicht nur die Wahl einer Datenstruktur, sondern auch die Angabe von Algorithmen (Prozeduren und Funktionen) für die Operationen des ADT.

Die begriffliche Unterscheidung zwischen ADT und Datenstruktur wird in diesem Buch nicht immer streng durchgehalten. Wir sprechen manchmal von der *Implementation einer Datenstruktur* und meinen damit eigentlich die Implementation eines ADT durch eine Datenstruktur. Wir möchten aber ausdrücklich betonen, daß der Begriff Datenstruktur sich stets auf Objekte der realen Welt und Operationen mit ihnen, nicht auf ideale Objekte der Mathematik bezieht.

Wir werden im folgenden die wichtigsten elementaren ADT (lineare Listen, Stapel, Schlangen, Bäume, Mengen) und mögliche Implementationen besprechen.

1.5 Lineare Listen

Lineare Listen basieren auf dem in der Mathematik wohlbekannten Konzept einer endlichen Folge von Elementen eines bestimmten Grundtyps. Man denke etwa an eine endliche Folge ganzer oder reeller Zahlen. Für eine endliche Folge von Zahlen spricht man üblicherweise vom ersten, zweiten und allgemein vom *i*-ten Element und bezeichnet sie mit a_1, a_2 und a_i. Man kann an eine Folge ein Element anhängen, ein Element an einer bestimmten Stelle einfügen oder entfernen und aus zwei Folgen durch „Hintereinanderhängen“ (Verketten) eine neue Folge machen. Es ist ferner üblich, auch die leere Folge explizit zuzulassen.

Entsprechend kann man den ADT „(lineare) Liste“ wie folgt definieren. Die Menge der Objekte ist die Menge aller endlichen Folgen von Elementen eines gegebenen Grundtyps. Strenggenommen müßte man für jeden Grundtyp einen eigenen ADT angeben. Wir werden das nicht tun, sondern uns den jeweiligen Grundtyp beliebig, aber fest gegeben denken. Wir setzen allerdings meistens voraus, daß der Grundtyp wenigstens zwei Komponenten hat, eine ganzzahlige Schlüsselkomponente und eine Komponente, die die „eigentliche“ Information enthält. Das heißt, mögliche Grundtypen sind wie folgt vereinbart:

```
type
  Grundtyp = record
               key : integer;
               info : {infotype}
               {eventuell weitere Komponenten}
             end
```

Wir beschreiben eine lineare Liste L mit $N \geq 1$ Elementen durch $L = \langle a_1, \ldots, a_n \rangle$; $\langle\rangle$ bezeichnet die leere Liste. Folgende Operationen mit linearen Listen werden betrachtet.

Einfügen(x, p, L): Das Einfügen eines neuen Elementes x (vom jeweiligen Grundtyp) in die Liste L an der Position p; alle Elemente ab Position p rücken dabei um eine Position nach hinten (man sagt auch: nach rechts). Diese Operation verändert die Liste L zur Liste L' wie folgt. Ist $L = \langle a_1, \ldots, a_n \rangle$ und $1 \leq p \leq n$, so ist das Ergebnis die Liste $L' = \langle a_1, \ldots, a_{p-1}, x, a_p, \ldots, a_n \rangle$; ist $L = \langle\rangle$ und $p = 1$, so ist $L' = \langle x \rangle$ das Ergebnis der Einfügeoperation. Ist $p = n + 1$, so ist $L' = \langle a_1, \ldots, a_n, x \rangle$ das Ergebnis. In allen anderen Fällen ist das Ergebnis undefiniert.

Entfernen(p, L): Das Entfernen eines Elementes an der Position p macht aus der Liste $L = \langle a_1, \ldots, a_{p-1}, a_p, a_{p+1}, \ldots, a_n \rangle$ die Liste $L' = \langle a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n \rangle$, falls $1 \leq p \leq n$. Sonst ist die Operation *Entfernen* undefiniert.

Suchen(x, L): Diese Operation liefert die Position des Elementes x in der Liste L, falls x in L vorkommt, und 0 sonst. Kommt x mehr als einmal in L vor, wird die von links oder von rechts her erste Position geliefert, an der x vorkommt.

Zugriff(p, L): Diese Operation liefert das Element a_p an der p-ten Position in $L = \langle a_1, \ldots, a_n \rangle$, falls $1 \leq p \leq n$. Sonst ist die Operation undefiniert.

Wir wollen uns zunächst mit diesen Operationen begnügen. Je nach Anwendungsfall kann es aber sinnvoll sein, weitere Operationen mit linearen Listen vorzusehen. Das können beispielsweise sein: Eine Funktion, die prüft, ob eine Liste L leer ist oder nicht, die Operation des Hintereinanderhängens (Verkettens) zweier linearer Listen, das Bilden von Teillisten oder das Ausgeben (Drucken) aller Elemente einer linearen Liste nach aufsteigenden Positionen.

Sehr oft möchte man auch statt der oben angegebenen Einfüge- und Entferne-Operationen Elemente nicht an einer bestimmten, explizit gegebenen Position, sondern nur abhängig vom Wert (der Schlüsselkomponente) des Elementes einfügen oder entfernen können. Wir bezeichnen diese Operationen mit

$$\mathit{Einfügen}(x, L) \quad \text{und} \quad \mathit{Entfernen}(x, L).$$

Alle bisher genannten Operationen operieren auf bereits bestehenden linearen Listen. Es ist meistens üblich, wenigstens eine Operation explizit vorzusehen, die eine lineare Liste erzeugt, die Initialisierung einer linearen Liste als leere Liste. Natürlich könnte man auch die Initialisierung nichtleerer linearer Listen zu einer gegebenen, nichtleeren Menge von Elementen des Grundtyps explizit vorsehen. Andererseits lassen sich solche Listen aber offensichtlich aus der anfangs leeren Liste durch iteriertes Einfügen sämtlicher Elemente erzeugen.

Wir geben jetzt mögliche Implementationen linearer Listen an. Dabei kommt es uns nicht so sehr darauf an, sämtliche für lineare Listen interessanten Operationen programmtechnisch zu realisieren, als vielmehr darauf, die Auswirkungen einer bestimmten Datenstrukturwahl auf die Komplexität der Operationen exemplarisch zu zeigen. Man kann die zahlreichen möglichen Implementation linearer Listen in zwei Klassen einteilen.

1. *Sequentiell gespeicherte lineare Listen*: Hier sind die Listenelemente in einem zusammenhängenden Speicherbereich so abgelegt, daß man — wie bei Arrays — auf das i-te Element über eine Adreßrechnung zugreifen kann.

2. *Verkettet gespeicherte lineare Listen*: Hier sind die Listenelemente in Speicherzellen abgelegt, deren Zusammenhang durch Zeiger hergestellt wird.

Wir behandeln beide Speicherungsformen getrennt.

1.5.1 Sequentielle Speicherung linearer Listen

Wir wählen als Datenstruktur zur Implementation sequentiell gespeicherter linearer Listen ein Array von Elementen des Grundtyps.

```
const
  maxelzahl = {genügend groß gewählte Konstante};
type
  Liste =  record
             element: array [0 .. maxelzahl] of Grundtyp;
             elzahl: integer
           end
```

Eine lineare Liste ist dann gegeben durch eine Variable

```
var L: Liste
```

L.elzahl ist die Anzahl der Listenelemente. Falls diese Zahl nicht 0 und kleiner oder gleich der maximalen Elementzahl *maxelzahl* ist, sind

$$L.element[1], \ldots, L.element[elzahl]$$

die Listenelemente an den Positionen $1, \ldots, elzahl$. Wir haben im Array der Elemente des Grundtyps eine 0-te Position als uneigentliche Listenposition vorgesehen, weil wir so die Suchoperationen besonders bequem implementieren können. Vor Beginn der Suche nach x schreiben wir das gesuchte Element x an diese Position. Damit wirkt x als sogenannter *Stopper* im Falle einer erfolglosen Suche.

```
function Suchen (x: Grundtyp; L: Liste) : integer;
{liefert die von rechts her erste Position, an der x in L
 vorkommt, und den Wert 0, falls x in L nicht vorkommt}
var
   pos: integer;
begin
   L.element[0] := x;
   pos := L.elzahl;
   while L.element[pos] ≠ x do
      pos := pos − 1;
   Suchen := pos
end {Suchen}
```

Wird ein Element durch seinen Schlüssel eindeutig identifiziert, genügt es natürlich

$$L.element[0].key := x.key \quad \text{statt} \quad L.element[0] := x$$

und

$$L.element[pos].key \neq x.key \quad \text{statt} \quad L.element[pos] \neq x$$

zu schreiben.

Wir geben noch die Prozeduren zum Einfügen und Entfernen eines Elementes für den Fall an, daß die Position, an der ein Element eingefügt bzw. entfernt werden soll, gegeben ist. Sie zeigen, daß es im allgemeinen nötig ist, für ein neu einzufügendes Element zunächst Platz zu schaffen und eine durch Entfernen eines Elementes entstehende Lücke durch Verschieben von Elementen wieder zu schließen.

```
procedure Einfügen (x: Grundtyp; p: integer; var L: Liste);
  {liefert die durch Einfügen von x an Position p in L entstehende
   Liste, wenn p eine gültige Position innerhalb L oder die Position
   unmittelbar nach Listenende ist, und eine Fehlermeldung sonst}
  var
     pos: integer;
  begin
     if L.elzahl = maxelzahl
        then Fehler (`Liste voll')
```

```
      else
        if (p > L.elzahl+1) or (p < 1)
          then Fehler (`ungültige Position')
          else
            begin
              for pos := L.elzahl downto p do {verschieben}
                L.element[pos+1] := L.element[pos];
              L.element[p] := x;
              L.elzahl := L.elzahl +1
            end
end {Einfügen}


procedure Entfernen (p: integer; var L: Liste);
{entfernt das Element an Position p aus der Liste L, falls p eine
 gültige Position innerhalb L ist, und liefert eine Fehlermeldung
 sonst}
var
  pos: integer;
begin
  if L.elzahl = 0
    then Fehler ('Liste ist leer`)
    else
      if (p > L.elzahl) or (p < 1)
        then Fehler ('ungültige Position`)
        else
          begin
            L.elzahl := L.elzahl -1;
            for pos := p to L.elzahl do {verschieben}
              L.element[pos] := L.element[pos +1]
          end
end {Entfernen}
```

Um in eine sequentiell gespeicherte Liste der Länge N ein neues Element einzufügen oder ein Element zu entfernen, müssen offenbar im ungünstigsten Fall $\Omega(N)$ Elemente verschoben werden. Der günstigste Fall liegt vor, wenn nur am Ende eingefügt und entfernt wird; dann sind keine Verschiebungen notwendig. Wenn man annimmt, daß jede der N möglichen Positionen gleichwahrscheinlich ist, kann man erwarten, daß im Mittel etwa die Hälfte der Elemente verschoben werden muß. Das Einfügen und Entfernen eines Elementes erfordert bei sequentieller Speicherung einer linearen Liste also sowohl im Mittel wie im schlechtesten Fall $\Omega(N)$ Schritte. Dabei spielt es keine Rolle, ob die Einfüge- bzw. Entferne-Position explizit gegeben ist oder mit Hilfe der Suchoperation zunächst gefunden werden muß. Denn ist der Schlüssel eines Elementes gegeben, muß man ebenfalls im Mittel und im schlechtesten Fall $\Theta(N)$ Schritte ausführen, um das Element mit diesem Schlüssel in einer Liste der Länge N zu finden bzw. festzustellen, daß es kein Element mit diesem Schlüssel in der Liste gibt.

Sind jedoch die Elemente einer sequentiell gespeicherten linearen Liste nach auf- oder absteigenden Schlüsselwerten sortiert, gibt es effizientere Verfahren zum Suchen eines Elementes. Verfahren zum Sortieren werden in Kapitel 2, Verfahren zum Suchen in sequentiell gespeicherten linearen Listen in Kapitel 3 genauer diskutiert. Wir halten hier nur fest, daß das Einfügen und Entfernen in sequentiell gespeicherten linearen Listen „teuer“ ist, das Suchen aber jedenfalls dann sehr effizient möglich ist, wenn die Liste sortiert ist.

1.5.2 Verkettete Speicherung linearer Listen

Statt die Listenelemente so in einem zusammenhängenden Speicherbereich abzulegen, daß man den Speicherplatz des i-ten Listenelementes durch eine Adreßrechnung leicht bestimmen kann, gehen wir jetzt so vor: Wir speichern zusammen mit jedem Listenelement einen Verweis auf das jeweils nächste Element ab. Die Listenelemente können also beliebig über den Speicher verstreut sein; insbesondere ist es nicht mehr erforderlich, vorab einen Bereich hinreichender Größe zur Aufnahme aller Listenelemente zu reservieren. Der belegte Speicherplatz paßt sich vielmehr dynamisch der jeweiligen aktuellen Größe der Liste an. Man benötigt allerdings nicht nur für die Listenelemente selbst, sondern auch für die Zeiger Speicherplatz.

Eine lineare Liste kann implementiert werden als eine Folge von Knoten; jeder Knoten enthält ein Listenelement des jeweiligen Grundtyps und einen Zeiger auf das jeweils nächste Listenelement. Die Knoten haben also folgenden Typ.

```
type
  Zeiger = ↑Knoten;
  Knoten = record
             dat: Grundtyp;
             next: Zeiger
           end
```

Eine Liste $L = \langle a_1, \ldots, a_n \rangle$ von n Elementen des jeweiligen Grundtyps kann man wie in Abbildung 1.3 veranschaulichen.

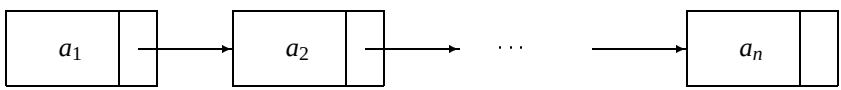


Abbildung 1.3

Wir müssen aber noch festlegen, wie wir den Listenanfang, das Listenende und die leere Liste kennzeichnen. Hier gibt es zahlreiche Möglichkeiten, die alle verschiedene Vor- und Nachteile haben, d.h. insbesondere Auswirkungen auf die Implementation der

für Listen auszuführenden Operationen. Wir geben im folgenden einige Möglichkeiten an und diskutieren ausgewählte Listenoperationen exemplarisch.

Eine erste Möglichkeit ist die, eine Liste durch einen Zeiger auf den Listenanfang zu realisieren und das Listenende durch einen **nil**-Zeiger zu markieren. Eine lineare Liste ist also vollständig beschrieben durch eine Variable L vom Typ *Zeiger*.

var L: *Zeiger*

L ist leer genau dann, wenn L den Wert **nil** hat. Eine Position in einer verkettet gespeicherten Liste wird also nicht, wie bei sequentiell gespeicherten Listen, durch eine laufende Nummer, sondern durch einen Zeiger auf ein Listenelement angegeben. Um in der Liste L nach einem Element x des Grundtyps zu suchen, muß man nicht nur den Fall gesondert betrachten, daß L leer ist, sondern auch jedesmal prüfen, ob beim Inspizieren des jeweils nächsten Listenelements nicht schon das Listenende erreicht ist, das durch einen **nil**-Zeiger (graphisch: durch einen Punkt, wie in Abbildung 1.4 zu sehen) markiert ist.

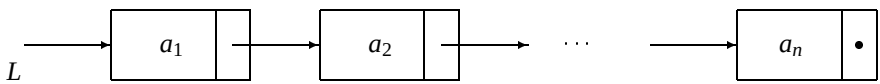


Abbildung 1.4

```
function Suchen (x: Grundtyp; L: Zeiger) : Zeiger;
{liefert einen Zeiger auf das von links her erste Vorkommen des
 Elementes x, falls x in L vorkommt, und den Wert nil sonst}
var
  pos: Zeiger;
begin
  if L = nil
    then Suchen := nil
    else
      begin
        pos := L;
        while (pos↑.dat ≠ x) and (pos↑.next ≠ nil) do
          pos := pos↑.next;
        {jetzt ist pos↑.dat = x oder pos↑.next = nil}
        if pos↑.dat = x
          then {x gefunden} Suchen := pos
          else {x kommt nicht vor} Suchen := nil
      end
end {Suchen}
```

Man beachte, daß wir die Position eines Elementes durch einen Zeiger auf einen Knoten realisiert haben, dessen *dat*-Komponente das Element ist.

Diese Implementation einer linearen Liste hat offensichtlich mehrere Schönheitsfehler. Man muß den Fall der leeren Liste und die explizite Abfrage auf das Listenende nicht nur beim Suchen gesondert behandeln. Auch beim Einfügen eines Elementes und beim Entfernen treten zahlreiche Sonderfälle auf.

Alle diese Schwierigkeiten entfallen bei der folgenden Implementation. Eine lineare Liste ist gegeben durch einen Kopfzeiger *head* und einen Schwanzzeiger *tail*, die jeweils auf zwei uneigentliche, sogenannte Dummy-Elemente zeigen; die eigentlichen Listenelemente befinden sich zwischen diesen beiden Dummy-Elementen (vgl. Abbildung 1.5).

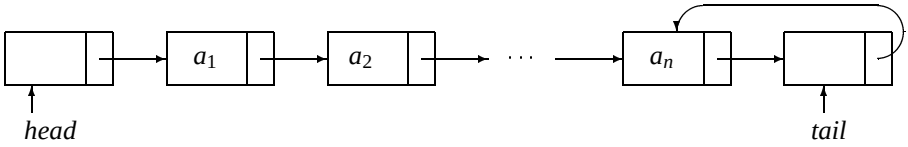


Abbildung 1.5

Die Liste ist durch den Kopf- und Schwanzzeiger gegeben.

var *head, tail*: *Zeiger*

Wie vorher wird die *i*-te Position realisiert durch einen Zeiger auf den Knoten, der das *i*-te Listenelement enthält. Der Schwanzzeiger *tail* markiert also die Position $n+1$, d.h. die Position nach dem Listenende.

Wir setzen (willkürlich) fest, daß die *next*-Komponente des das Listenende markierenden Dummy-Elementes auf das vorangehende Element zurückverweist. Das erleichtert das Hintereinanderhängen zweier Listen, wie wir weiter unten zeigen werden. Die leere Liste hat also die in Abbildung 1.6 gezeigte Form. Sie wird durch die Prozedur *Initialisiere* erzeugt.

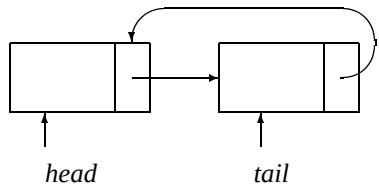


Abbildung 1.6

```
procedure Initialisiere (var head, tail: Zeiger);
begin
  new(head);
  new(tail);
  head↑.next := tail;
  tail↑.next := head
end {Initialisiere}
```

Zum Suchen eines Elementes x vom Grundtyp kann man die schon bei der sequentiellen Speicherung linearer Listen benutzte Stopper-Technik anwenden und das gesuchte Element vor Beginn der Suche in das Dummy-Element am Listenende schreiben.

```
function Suchen (x: Grundtyp; head, tail: Zeiger) : Zeiger;
{liefert einen Zeiger auf das von links her erste Vorkommen
 des Elementes x, falls x in der Liste mit Kopfzeiger head und
 Schwanzzeiger tail vorkommt, und den Wert tail sonst}
var
  pos: Zeiger;
begin
  tail↑.dat := x; {Stopper}
  pos := head;
  repeat pos := pos↑.next
  until pos↑.dat = x;
  Suchen := pos
end {Suchen}
```

Beim Einfügen und Entfernen eines Elementes an einer gegebenen Position p ist es notwendig, den *next*-Zeiger des Vorgängers des p-ten Knotens der Liste umzulegen; auf diesen Zeiger kann man aber nicht mehr ohne weiteres (in konstanter Zeit) zugreifen, wenn man Position p wie bisher als einen Zeiger auf den Knoten auffaßt, dessen Datenkomponente das p-te Listenelement ist.

Nehmen wir beispielsweise an, daß ein neues Element x an Position p eingefügt werden soll. Die Situation vor dem Einfügen kann graphisch wie in Abbildung 1.7 dargestellt werden.

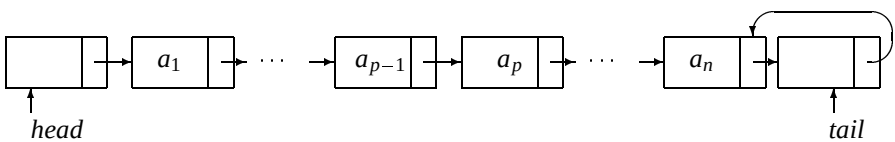


Abbildung 1.7

Nach dem Einfügen wird daraus die Situation von Abbildung 1.8.

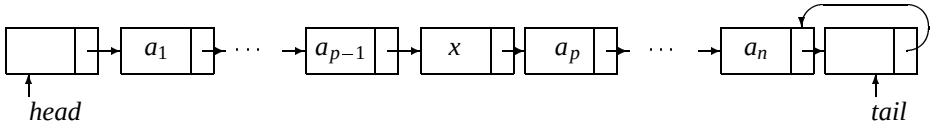


Abbildung 1.8

Die gewünschte Situation kann im Falle des Einfügens durch einen Kunstgriff erreicht werden. Man ersetzt das p-te Element a_p durch x und fügt a_p an der $(p+1)$-ten Position ein.

```
procedure Einfügen (x : Grundtyp; p, head : Zeiger; var tail : Zeiger);
{liefert die Liste mit Kopfzeiger head und Schwanzzeiger tail, die
 durch Einfügen von x an der Stelle, auf die p zeigt, entsteht}
var
   hilf : Zeiger;
begin
   if p = tail
      then hilf := tail
      else hilf := p↑.next;
   new(p↑.next);
   p↑.next↑.next := hilf;
   p↑.next↑.dat := p↑.dat;
   p↑.dat := x;
   if p = tail  {eingefügt an letzter Position}
      then tail := tail↑.next;
   if hilf = tail  {eingefügt an vorletzter Position}
      then tail↑.next := p↑.next
end {Einfügen}
```

Man beachte, daß diese Prozedur das Einfügen eines neuen Elementes x auch dann korrekt bewerkstelligt, wenn p die Position des letzten Elementes oder die Position unmittelbar nach Listenende (also die Position *tail*) ist.

Das Entfernen eines Elementes an einer gegebenen Position p läßt sich so im allgemeinen nicht durchführen, weil beim Entfernen des letzten Elementes der Zeiger *tail*↑.*next* nicht korrekt adjustiert werden kann, wenn man auf den Vorgänger von p in der Liste keinen Zugriff hat. Man muß die Liste vom Anfang an durchlaufen, um den dem Element an Position p vorangehenden Knoten in der Liste zu bestimmen, damit man dessen *next*-Komponente über p hinweg auf den nächstfolgenden Knoten zeigen lassen kann. Diese Schwierigkeit entfällt, wenn man eine andere Implementation des Positionsbegriffs vornimmt. Statt zu sagen: „Die Position p innerhalb der Liste ist gegeben durch einen Zeiger auf den Knoten mit *dat*-Komponente a_p, der das p-te Listenelement enthält“, kann man auch sagen: „Die Position p ist gegeben durch einen Zeiger auf den Knoten, dessen *next*-Komponente einen Zeiger auf den Knoten mit *dat*-

Komponente a_p enthält.“ Die Position 1 ist also gegeben durch den Zeiger *head* auf das Dummy-Element am Listenkopf usw.

Man „hängt“ also gewissermaßen mit dem Zeiger einen Knoten „zurück“ und schaut auf den nächstfolgenden voraus, um das gegebenenfalls notwendige Umlegen von Zeigern zu erleichtern. Wir verzichten darauf, Prozeduren zum Einfügen, Entfernen usw. für lineare Listen anzugeben, wenn der Positionsbegriff wie zuletzt beschrieben implementiert wird. Vielmehr begnügen wir uns damit zu zeigen, wie man ein Listenelement mit gegebenem Wert *x* (dessen Position also zunächst bestimmt werden muß) nach dieser Technik des *Zurückhängens mit Vorausschauen* entfernt.

```
procedure Entfernen (x : Grundtyp; head, tail : Zeiger);
{entfernt den von links her ersten Knoten mit Datenkomponente x
 aus einer Liste mit Kopfzeiger head und Schwanzzeiger tail, falls x
 in der Liste vorkommt; sonst wird eine Fehlermeldung ausgegeben}
var
   pos : Zeiger;
begin
   pos := head;
   tail↑.dat := x;  {Stopper}
   while pos↑.next↑.dat ≠ x do pos := pos↑.next;
   if pos↑.next ≠ tail
      then pos↑.next := pos↑.next↑.next
      else Fehler (`x kommt nicht vor');
   if pos↑.next = tail
   {letztes Element wurde entfernt}
        then tail↑.next := pos
end {Entfernen}
```

Dabei soll die Prozedur *Fehler* das Programm nach der entsprechenden Fehlermeldung beenden. Wir haben hier, wie auch im Falle der anderen Listenoperationen, besonders darauf achten müssen, den *next*-Zeiger des Dummy-Elementes am Listenende auf den vorangehenden Knoten zeigen zu lassen. Das macht es möglich, das Hintereinanderhängen (Verketten) zweier Listen in konstanter Schrittzahl auszuführen.

```
procedure Verketten (head1, head2, tail1, tail2 : Zeiger;
                                  var head, tail : Zeiger);
{liefert zu zwei Listen mit Kopf- und Schwanzzeiger head1, head2,
 tail1, tail2 eine neue Liste mit Kopfzeiger head und Schwanzzeiger
 tail, die durch Anhängen der zweiten Liste an das Ende der
 ersten entsteht}
begin
   head := head1;
   tail1↑.next↑.next := head2↑.next;
   tail := tail2;
   if tail2↑.next = head2  {leere Liste 2}
      then tail↑.next = tail1↑.next
end {Verketten}
```

Um das Einfügen und Entfernen von Listenelementen bei gegebener Position möglichst einfach ausführen zu können, kann man zu jedem Listenelement nicht nur einen Zeiger auf das nächstfolgende, sondern auch auf das jeweils vorangehende Listenelement abspeichern. Man spricht in diesem Fall von *doppelt verketteter Speicherung* einer linearen Liste; entsprechend nennt man die bisher besprochene Form der Speicherung auch *einfach verkettete Speicherung*. In einer doppelt verketteten linearen Liste haben die Knoten also folgendes Format.

```
type
  Zeiger = ↑Knoten;
  Knoten = record
             dat : Grundtyp;
             vor, nach : Zeiger
           end
```

Nehmen wir beispielsweise an, es soll das Element an Position p im Innern der Liste entfernt werden; die Position p sei durch einen Zeiger auf einen Knoten mit Datenkomponente a_p realisiert, wie in Abbildung 1.9 zu sehen.

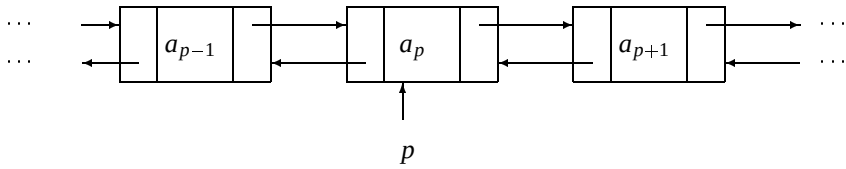


Abbildung 1.9

Das Entfernen wird erreicht durch folgende Zuweisung:

$p\uparrow.vor\uparrow.nach := p\uparrow.nach;$
$p\uparrow.nach\uparrow.vor := p\uparrow.vor;$

Natürlich kann man auch doppelt verkettete lineare Listen mit und ohne Kopf- und Schwanzzeiger bzw. mit und ohne ein Dummy-Element am Listenanfang oder -ende implementieren.

Eine abschließende, allgemeine Bemerkung zum Entfernen von Listenelementen: Wir haben die nach dem Entfernen nicht mehr benötigten Knoten nicht zur neuen und eventuell anderen Verwendung explizit freigegeben, sondern sie nur aus der die jeweilige Liste realisierenden verketteten Struktur durch Umlegen von Zeigern entfernt. Man kann diese Knoten bei manchen Pascal-Implementationen durch einen Aufruf der Standardprozedur *dispose* explizit freigeben. Man kann sie aber auch in einer eigenen Freiliste sammeln und jedesmal zunächst dort nachsehen, ob man nicht von dieser Freiliste einen Knoten nehmen kann, bevor man einen neuen durch einen Aufruf der Standardprozedur *new* schafft.

Wir fassen einige Varianten verkettet gespeicherter linearer Listen noch einmal stichwortartig zusammen.

Implementation 1: Einfach verkettete Liste; gegeben durch Zeiger auf Listenanfang; Listenende durch **nil**-Zeiger markiert, kein Schwanzzeiger; Position p durch Zeiger auf Knoten mit Datenkomponente a_p realisiert.

Implementation 2: Einfach verkettete Liste; gegeben durch einen Kopf- und einen Schwanzzeiger, die jeweils auf ein Dummy-Element zeigen; Position p durch Zeiger auf Knoten mit Datenkomponente a_p realisiert.

Implementation 3: Wie Implementation 2, aber Position p durch Zeiger auf Knoten realisiert, dessen *next*-Komponente einen Zeiger auf Knoten mit Datenkomponente a_p enthält.

Implementation 4: (Vgl. Abbildung 1.10) Doppelt verkette lineare Liste, mit Kopfzeiger *head* und Schwanzzeiger *tail*, die auf das erste bzw. letzte Listenelement zeigen; die *vor*-Komponente des ersten und die *nach*-Komponente des letzten Listenelementes haben den Wert **nil**; die Position p ist durch einen Zeiger auf das Listenelement mit Datenkomponente a_p realisiert.

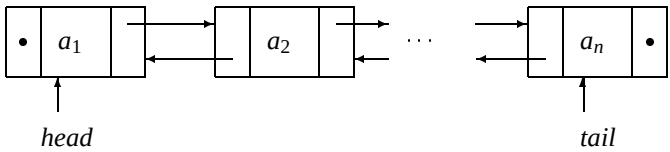


Abbildung 1.10

Wir stellen für diese vier Implementationen die im schlechtesten Fall zur Ausführung ausgewählter Listenoperationen benötigten Schrittzahlen für Listen der Länge N in Tabelle 1.2 zusammen.

Im Gegensatz zur sequentiellen Speicherung bringt es kaum Vorteile, die Elemente einer verkettet gespeicherten linearen Liste etwa nach aufsteigenden Schlüsselwerten in den Knoten zu speichern. Lediglich die erfolglose Suche kann unter Umständen etwas verkürzt werden, weil beim Durchlaufen der Liste vom Anfang her die Suche bereits abgebrochen werden kann, sobald man auf ein Listenelement gestoßen ist, dessen Wert größer als der des gesuchten ist.

Es kann jedoch sinnvoll sein, eine lineare Liste etwa nach abnehmenden Suchhäufigkeiten zu ordnen, wenn diese vorher bekannt sind. Kennt man die (relativen) Suchhäufigkeiten nicht, so kann man verschiedene Strategien implementieren, die mit der Zeit eine für das Suchen günstige Anordnung (nach abnehmenden Suchhäufigkeiten) entstehen lassen. Wir gehen auf diese Strategien in Kapitel 3 genauer ein.

Wenn wir offenlassen wollen, wie eine lineare Liste implementiert wird, schreiben wir:

type
Grundtyp = {*der jeweilige Grundtyp*};
Liste = **list of** *Grundtyp*

	Implementation			
	1	2	3	4
Einfügen eines neuen Elementes am Listenanfang	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Einfügen eines Elementes an gegebener Position	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Entfernen eines Elementes an gegebener Position	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$
Suchen eines Elementes mit gegebenem Wert	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$
Hintereinanderhängen zweier Listen	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Tabelle 1.2

Dann können wir eine Liste L einfach als Variable vom Typ *Liste* vereinbaren und diesen Typ auch in den jeweils benötigten Funktionen und Prozeduren zur Manipulation von Listen verwenden.

1.5.3 Stapel und Schlangen

Statt das Einfügen und Entfernen von Elementen an einer beliebigen Position innerhalb einer linearen Liste zuzulassen, genügt es für viele Anwendungen, wenn diese Operationen am Anfang oder am Ende einer Liste ausgeführt werden können. Wir führen für diese Operationen eigene Bezeichnungen ein.

pushhead(L,x): Fügt das Element x am Anfang der Liste L ein.

Wir nehmen also an, daß man vom Anfang der Liste L sprechen kann. Dies kann man auch explizit machen und eine Funktion *top* mit folgender Bedeutung definieren.

top(L): Liefert den Wert des ersten („obersten“) Elementes der Liste L.

top(L) ist natürlich nur dann definiert, wenn die Liste L nicht leer ist. Sei *leer* eine Funktion, die für eine Liste L den Wert *true* liefert, wenn L leer ist, und *false* sonst. Dann ist *top*(L) nicht definiert, falls *leer*(L) gilt. Es ist jedoch stets, also für jedes L und x,

$$top(pushhead(L,x)) = x.$$

Entsprechend definiert man eine Funktion *pushtail* wie folgt:

pushtail(L,x): Fügt das Element x am Ende der Liste L ein.

Operationen zum Entfernen von Elementen am Anfang bzw. Ende von L werden so definiert:

pophead(L,x): Entfernt das erste Element (am Anfang) von L und weist es der Variablen x vom Grundtyp zu; falls *leer*(L), ist *pophead*(L,x) nicht definiert.

poptail(L,x): Entfernt das letzte Element (am Ende) von L und weist es der Variablen x vom Grundtyp zu; falls *leer*(L), ist *poptail*(L,x) nicht definiert.

Es ist ferner möglich, auch eine Funktion *bottom* (oder: *rear*) zu definieren, die den Wert des letzten (untersten) Elementes einer Liste L liefert.

Werden für lineare Listen nur die Operationen bzw. Funktionen *Initialisieren, leer, top, bottom, pushhead, pophead, pushtail, poptail* benötigt, hat man Listen mit kontrollierten Zugriffspunkten. Sie können leicht so implementiert werden, daß alle Operationen in konstanter Schrittzahl ausführbar sind, und zwar gilt das sowohl bei sequentieller als auch bei geketteter Speicherung der Liste L.

Zwei Spezialfälle haben eine besondere Bedeutung und auch einen eigenen Namen erhalten. *Stapel*: Hier sind *Initialisieren, leer, top, pushhead* und *pophead* die einzigen zugelassenen Operationen. *Schlange*: Hier sind *Initialisieren, leer, top, pushtail* und *pophead* die einzigen zugelassenen Operationen.

Im Stapel lassen sich also Elemente nach dem sogenannten LIFO-Prinzip (last in first out) und in Schlangen nach dem FIFO-Prinzip (first in first out) speichern. Die Operationen *pushhead* und *pophead* bei Stapeln werden meistens einfach *push* und *pop* genannt. Ferner nimmt man meistens an, daß die *pop*-Operation nur das oberste Element vom Stapel entfernt, ohne es zugleich einer Variablen vom Grundtyp zuzuweisen. Denn man kann ja, falls nötig, das oberste Element von S mit Hilfe von $top(S)$ zunächst einer Variablen vom Grundtyp zuweisen, bevor man $pop(S)$ ausführt. Bei Schlangen spricht man statt von *pophead* und *pushtail* auch von *dequeue* und *enqueue*.

Wir überlassen es dem Leser, sich eine geeignete Implementation für eine lineare Liste mit kontrollierten Zugriffspunkten, insbesondere also für Stapel und Schlangen, genau zu überlegen. Dabei ist darauf zu achten, daß die jeweiligen Operationen in konstanter Schrittzahl ausführbar sind. Daher ist es beispielsweise nicht ohne weiteres möglich, für einen sequentiell gespeicherten Stapel einfach die Implementation aus Abschnitt 1.5.1 zu übernehmen und dabei das Element an Position 1 als oberstes Element des Stapels anzusehen. Abbildung 1.11 zeigt, wie ein sequentiell gespeicherter Stapel implementiert werden kann.

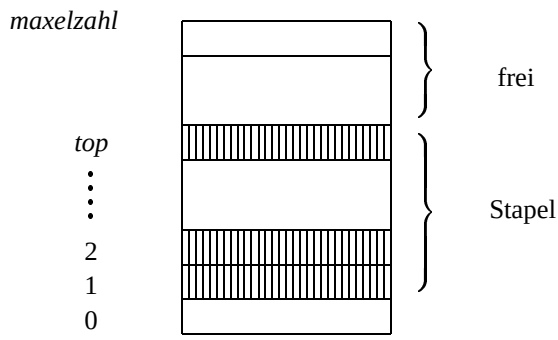


Abbildung 1.11

Werden in einer Schlange etwa ebenso häufig neue Elemente hinten angehängt wie vorne entfernt werden, bleibt die Länge der Schlange nahezu unverändert. Übernimmt man einfach die Implementation aus Abschnitt 1.5.1 für eine sequentiell gespeicherte Schlange, so „kriecht“ die Schlange offenbar im anfangs reservierten Speicherbereich maximaler Länge an das Ende dieses Bereichs, wenn man das vordere Element der Schlange im Array zunächst an Index 1 ablegt. Um zu verhindern, daß man keine Elemente am Ende mehr anfügen kann, wenn die Schlange am Ende angestoßen ist, obwohl vorne noch viel Platz ist, ist es sinnvoll, sich den reservierten Speicherbereich zyklisch geschlossen vorzustellen: Stößt die Schlange am rechten Ende des reservierten Bereichs an, beginnt man, am Anfang dieses Bereichs weitere Elemente einzufügen. Abbildung 1.12 veranschaulicht dies.

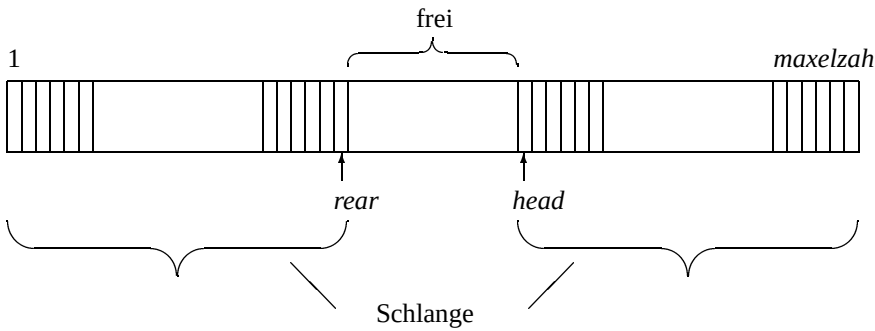


Abbildung 1.12

Wir überlassen es dem Leser, sich genau zu überlegen, wie die Operationen *pophead* und *pushtail* implementiert werden können.

Ein wichtiger Anwendungsfall für Schlangen sind *Warteschlangen* aller Art, z.B. Kunden vor Kassen, Akten vor Sachbearbeitern, Druckaufträge vor Druckern usw. Häufig ordnet man den in eine (Warte-)Schlange einzureihenden Elementen des jeweiligen Grundtyps Prioritäten zu und erwartet, daß Elemente mit höherer Priorität Vorrang vor solchen mit niedrigerer Priorität haben; d.h. sie müssen entsprechend eher aus der Schlange entfernt werden. Man spricht in diesem Fall von *Vorrangswarteschlangen* (englisch: *priority queues*). Sie werden in Kapitel 6 genauer behandelt.

Wichtige Anwendungen für Stapel findet man im Zusammenhang mit dem Erkennen und Auswerten wohlgeformter Klammerausdrücke, bei der Realisierung von Unterprogrammaufrufen und der Auflösung rekursiver Funktionen und Prozeduren in iterative. Wir bringen dazu zwei einfache Beispiele.

Beispiel 1: *Erkennen wohlgeformter Klammerausdrücke*

Wir wollen Zeichenreihen, die aus öffnenden und schließenden Klammern bestehen, daraufhin überprüfen, ob sie wohlgeformt sind, d.h. ob sie aus passenden Paaren öffnender und schließender Klammern aufgebaut sind. $(()())$ ist ein wohlgeformter Klam-

merausdruck; ((() ist keiner. Die Menge der wohlgeformten Klammerausdrücke kann man wie folgt induktiv definieren.

(0) () ist ein wohlgeformter Klammerausdruck.

(1) Sind w_1 und w_2 wohlgeformte Klammerausdrücke, so ist auch der durch Hintereinanderschreiben von w_1 und w_2 entstehende Ausdruck $w_1 w_2$ ein wohlgeformter Klammerausdruck.

(2) Mit w ist auch (w) ein wohlgeformter Klammerausdruck.

(3) Nur die nach (0) bis (2) gebildeten Zeichenreihen sind wohlgeformte Klammerausdrücke.

Wie kann man durch einmaliges, zeichenweises Lesen von links nach rechts feststellen, ob eine nur aus den Zeichen „(“ und „)“ gebildete Zeichenreihe ein wohlgeformter Klammerausdruck ist? Es ist nicht schwer, sich davon zu überzeugen, daß man das feststellen kann, wenn man nach folgender Methode verfährt. Wir benutzen einen Stapel zur Speicherung öffnender Klammern. Immer wenn wir beim Lesen von links nach rechts auf eine öffnende Klammer stoßen, legen wir sie auf dem Stapel ab. Treffen wir auf eine schließende Klammer, sehen wir im Stapel nach, ob dort noch eine öffnende Klammer steht; wenn ja, entfernen wir sie. Wenn nein, gibt es mehr schließende als öffnende Klammern. Im letzten Fall ist die Zeichenreihe kein wohlgeformter Klammerausdruck. Ist am Ende der Stapel leer, ist die gelesene Zeichenreihe ein wohlgeformter Klammerausdruck, sonst nicht.

Wir geben eine genauere Formulierung dieses Verfahrens an, ohne daß wir dabei auf eine spezielle Implementation von Stapeln zurückgreifen wollen. Daher nehmen wir an, daß wir einen Stapel als Liste des gewünschten Grundtyps wie folgt vereinbart haben.

```
type
    Stapel = list of Klammerauf;
var
    S : Stapel
```

Wir verwenden nur die für Stapel zugelassenen Operationen *Initialisieren*, *push*, *pop* und *leer* und eine Prozedur zum Lesen des jeweils nächsten Zeichens:

```
Initialisiere S als leeren Stapel;
while noch nicht alle Zeichen gelesen do
  begin lies nächstes Zeichen x;
    if x = `('
      then push(S,x)
        else {x = `)'}
        {hole zugehörige `(' vom Stapel}
          if leer(S)
            then {kein wohlgeformter Klammerausdruck}
            else pop(S)
  end; {while}
if not leer(S)
  then {kein wohlgeformter Klammerausdruck}
```

Ein solcher Stapel, der nur gleiche Elemente speichert, kann natürlich auch einfach durch einen Zähler modelliert werden, der die Anzahl der Elemente auf dem Stapel angibt. Wir haben dieses Beispiel gewählt, weil man auf ähnliche Art auch das Erkennen und Auswerten arithmetischer Ausdrücke erledigen kann. Das Verfahren wird allerdings komplizierter, wenn man die üblichen Vorrangsregeln (Punktrechnung geht vor Strichrechnung) beim Auswerten arithmetischer Ausdrücke beachten muß.

Beispiel 2: *Iterative Auswertung einer rekursiv definierten Funktion oder Prozedur*
Wir nehmen den Binomialkoeffizienten als Beispiel einer rekursiv definierten Funktion. Für zwei natürliche Zahlen n und k, mit $0 \le k \le n$, ist $\binom{n}{k}$ wie folgt definiert.

$$\binom{n}{k} = \begin{cases} 1, & \text{falls } k = 0 \text{ oder } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{falls } 0 < k < n \end{cases}$$

$\binom{n}{k}$ ist die Anzahl der verschiedenen Möglichkeiten, k Elemente aus einer Menge von n Elementen auszuwählen. Man kann diese Definition unmittelbar in eine Funktionsdeklaration übersetzen.

```
function bin (n, k: integer) : integer;
{berechnet die Anzahl der Möglichkeiten, k aus n Elementen
 zu wählen, unter der Annahme, daß 0 ≤ k ≤ n ist}
begin
  if (k = 0) or (k = n)
    then bin := 1
    else bin := bin(n − 1, k − 1) + bin(n − 1, k)
end {bin}
```

Um dieses Programm abzuarbeiten, muß offenbar einer der zwei rekursiven Funktionsaufrufe zunächst zurückgestellt werden und der andere (auf dieselbe Art) soweit abgearbeitet werden, bis man schließlich bei einem Funktionsaufruf angelangt ist, der unmittelbar den Wert 1 liefert. Erst dann können die vorher zurückgestellten Funktionsaufrufe weiter bearbeitet werden. Entscheiden wir uns (willkürlich) dafür, den ersten Funktionsaufruf zunächst zurückzustellen und den zweiten weiterzubearbeiten, ergibt sich beispielsweise das Berechnungsschema von Tabelle 1.3 bei der Berechnung von $\binom{4}{2}$.

Man erhält also einen Stapel noch nicht erledigter Teilprobleme. Anfangs enthält der Stapel das zu lösende Anfangsproblem, das ist die Berechnung von $\binom{n}{k}$ bzw. die Aufforderung zur Auswertung von $bin(n,k)$. Ein Problem ist in diesem Fall durch die beiden Argumente n und k vollständig beschrieben. Dann schaut man jeweils nach, ob auf dem Stapel noch unerledigte Probleme liegen. Ist das oberste Problem von der Form $\binom{n}{k}$ mit $0 < k < n$, so ersetzt man es durch zwei (Teil-)Probleme: $\binom{n-1}{k-1}$ wird das zweitoberste und $\binom{n-1}{k}$ das neue oberste Element. Ist das oberste Problem von der Form $\binom{n}{k}$ mit $k = 0$ oder $n = k$, entfernt man es und erhöht das anfangs mit 0 initialisierte Zwischenergebnis um 1. Das wird solange durchgeführt, bis der Problemstapel leer ist.

noch zu berechnen (Problemstapel)	bisheriges Zwischenergebnis z
$\binom{4}{2}$	$z = 0$
$\binom{3}{1} + \binom{3}{2}$	
$\binom{3}{1} + \binom{2}{1} + \binom{2}{2}$	
$\binom{3}{1} + \binom{2}{1}$	$z = 1$
$\binom{3}{1} + \binom{1}{0} + \binom{1}{1}$	
$\binom{3}{1} + \binom{1}{0}$	$z = 2$
$\binom{3}{1}$	$z = 3$
$\binom{2}{0} + \binom{2}{1}$	
$\binom{2}{0} + \binom{1}{0} + \binom{1}{1}$	
$\binom{2}{0} + \binom{1}{0}$	$z = 4$
$\binom{2}{0}$	$z = 5$
	$z = 6$

Tabelle 1.3

Als Grundtyp für den Problemstapel können wir in diesem Fall wählen

```
type
  problem = record
              o, u : integer
            end
```

Wir setzen voraus, daß folgende Vereinbarungen getroffen sind:

```
type
  stack = list of problem;
var
  S : stack;
  p, q, x : problem
```

Dann kann die Berechnung der Binomialkoeffizienten $\binom{n}{k}$, d.h. das Abarbeiten der rekursiv deklarierten Funktion $bin(n,k)$ mit Hilfe des Stapels S und der für Stapel zugelassenen Operationen folgendermaßen ausgeführt werden.

```
Initialisiere S {mit dem Anfangsproblem p, für das p.o = n
   und p.u = k gilt};
z := 0;  {Zwischenergebnis initialisiert}
repeat
   x := top(S);
   pop(S);
   if (x.u = 0) or (x.u = x.o)
      then z := z + 1
      else
         begin
            q.o := x.o − 1;
            q.u := x.u − 1;
            push(q, S);
            q.u := x.u;
            push(q, S)
         end
until leer(S)
```

Dies ist ein einfaches Beispiel für ein allgemeines Prinzip, nach dem sich rekursive Funktionen und Prozeduren mit Hilfe eines Stapel von (Teil-)Problemen abarbeiten lassen. Es kann als *Schema zur Rekursionselimination* wie folgt formuliert werden.

1. *Initialisiere den Problemstapel S mit dem zu lösenden Anfangsproblem.*
2. **repeat**
   bearbeite das oberste Problem p von S;
   müssen (bei der Bearbeitung von p) Teilprobleme p_1, p_2, ... zurückgestellt werden, staple sie auf S
   until *Stapel S leer.*

Nach diesem Schema erhält man dann ein gleichwertiges nichtrekursives, also iteratives Programm. Der Leser wird in den folgenden Kapiteln zahlreiche Beispiele finden, auf die dieses Schema zur Rekursionselimination anwendbar ist. So einfach, wie es hier scheint, ist die Anwendung des Schemas in den meisten Fällen allerdings nicht. Es ist oft nicht klar, wie ein Problem so vollständig beschrieben werden kann, daß es zurückgestellt und auf einem Problemstapel abgelegt werden kann. Es ist ferner häufig nicht möglich, das jeweils oberste Problem vollständig zu bearbeiten, weil in die Bearbeitung durchaus Ergebnisse von zunächst zurückgestellten Teilproblemen eingehen können. Bevor man dann mit der Bearbeitung eines Teilproblems beginnt, muß man sich unter Umständen den gesamten, bis zur Zurückstellung erreichten Zwischenzustand der Rechnung genau merken, zunächst das Teilproblem lösen, und dann die (Haupt-) Rechnung fortsetzen. Das oben formulierte Schema zur Rekursionsauflösung ist also eher als ein sehr grober Rahmen, aber keinesfalls als eine mechanisch anwendbare Regel zu verstehen.

1.6 Ausblick auf weitere Datenstrukturen

In diesem Abschnitt wollen wir eine kurze Vorschau auf weitere abstrakte Datentypen und Datenstrukturen geben, die in späteren Kapiteln ausführlich behandelt werden. Dazu gehören insbesondere Mengen.

Mengen unterscheiden sich von (linearen) Listen vor allem dadurch, daß man den Elementen einer Menge üblicherweise keine Ordnungsnummer zuordnet, also nicht vom ersten, zweiten, dritten, ... Element einer Menge spricht. Das mathematische Mengenkonzept geht davon aus, daß man alle Objekte eines gegebenen Universums, die eine bestimmte Eigenschaft haben, zu einer neuen Gesamtheit zusammenfassen kann — zur Menge aller Elemente mit dieser Eigenschaft. Dieses Prinzip zur Bildung von Mengen wird *Komprehensionsschema* genannt. Es ist ein sehr mächtiges Mittel zur Mengenbildung, das allerdings mit gehöriger Vorsicht benutzt werden muß, um widersprüchliche Aussagen über Mengen zu vermeiden. (Ein berühmtes Beispiel ist die Menge U aller Mengen, die sich nicht selbst als Element enthalten. Für U gilt: U enthält sich selbst als Element genau dann, wenn U sich nicht selbst als Element enthält.)

Mathematiker lernen den sinnvollen Gebrauch des Komprehensionsschemas zur Mengenbildung in der Regel durch Erfahrung. Daneben gibt es eine axiomatisierte Mengenlehre als mathematische Theorie. Sie ist auf der Elementbeziehung $\in$ als einzigem Grundbegriff aufgebaut. Dementsprechend könnte man sich einen abstrakten Datentyp Menge gegeben denken durch den Bereich aller Mengen im mathematischen Sinne, zusammen mit einer einzigen, zweistelligen Relation **in**: x **in** S ist wahr genau dann, wenn x ein Element der Menge S ist.

Das Komprehensionsschema als Operation zur Bildung von Mengen ist als Operation eines abstrakten Datentyps zu allgemein; die Elementbeziehung als einzige zugelassene Operation ist in vielen Fällen nicht ausreichend. Als Bausteine in Algorithmen treten durchweg nur endliche Mengen, aber neben der Elementbeziehung zahlreiche weitere Operationen auf. Je nach dem Spektrum der jeweils zugelassenen Operationen werden eigene abstrakte Datentypen mit besonderen Implementationen eingeführt. Wir behandeln einige wichtige Fälle im Kapitel 6 unter dem Stichwort *Mengenmanipulationsprobleme* und begnügen uns hier mit einer groben Übersicht.

Der Datentyp **set**: In Pascal kann man eine variable Anzahl von Elementen desselben Grundtyps zu einem **set** zusammenfassen und Variablen vom **set**-Typ verwenden. Als Grundtypen sind nur einfache Typen, aber nicht der Typ *real* zugelassen. Die Menge der durch den **set**-Typ beschriebenen Werte ist — idealerweise — die Menge aller Teilmengen der Menge der Werte des Grundtyps. In Wirklichkeit sind aber durch die Implementation der Sprache starke Beschränkungen in der Anzahl der zugelassenen Elemente gegeben. Somit ist dieser in die Programmiersprache eingebaute Datentyp von sehr eingeschränktem Wert für die Anwendungen. Wir werden ihn in diesem Buch nicht verwenden.

Wörterbücher (*Dictionaries*): Als Wörterbuch wird eine Menge von Elementen eines gegebenen Grundtyps bezeichnet, auf der man die Operationen Suchen, Einfügen und Entfernen von Elementen ausführen kann. Darüberhinaus wird stillschweigend vorausgesetzt, daß es eine Operation zur Initialisierung des leeren Wörterbuches gibt. Man nimmt — wie bei linearen Listen — meistens an, daß alle Elemente über einen in der

Regel ganzzahligen Schlüssel identifizierbar sind. Es ist üblich, die Such-, Einfüge- und Entferne-Operation nur vom jeweiligen Schlüssel abhängig zu machen, so daß man zur weiteren Vereinfachung häufig annimmt, daß ein Wörterbuch eine Menge S ganzzahliger Schlüssel ist, auf der folgende Operationen ausgeführt werden.

Suchen(x):	Liefert den Wert *true* genau dann, wenn x in S vorkommt, und *false* sonst.

Wenn x in S vorkommt und x Schlüssel eines Elementes mit vielleicht umfangreicher Datenkomponente ist, so soll als Ergebnis der Suchoperation natürlich auch der Zugriff auf die jeweilige Datenkomponente möglich sein.

Einfügen(x):	Ersetze S durch $S \cup \{x\}$.
Entfernen(x):	Ersetze S durch $S \setminus \{x\}$.

Das Problem, eine geeignete Implementation für Wörterbücher zu finden, also eine Datenstruktur zusammen mit möglichst effizienten Algorithmen zum Suchen, Einfügen und Entfernen von Schlüsseln, nennt man das *Wörterbuchproblem*.

Es ist offensichtlich, daß sequentiell oder verkettet gespeicherte lineare Listen eine mögliche Implementation von Wörterbüchern (also eine Lösung des Wörterbuchproblems) darstellen. Hashverfahren (vgl. Kapitel 4) und Bäume aller Art (vgl. hierzu Kapitel 5) liefern weitere Implementationsmöglichkeiten.

Kollektionen paarweise disjunkter Mengen: In einer Reihe von Anwendungen treten Kollektionen von paarweise disjunkten Mengen auf, für die einige oder alle der folgenden Operationen ausgeführt werden können.

Einfügen(S,x):	Fügt das Element x in Menge S ein.
Entfernen(S,x):	Entfernt das Element x aus Menge S.
Suchen(S,x):	Liefert *true*, wenn Element x in Menge S vorkommt, und *false* sonst.
Find(x):	Liefert den Namen derjenigen Menge, die Element x enthält, wenn es eine solche Menge in der Kollektion gibt; sonst ist der Wert undefiniert.

Diese Operationen verändern wohl einzelne Mengen der Kollektion, aber nicht die Kollektion selbst. Die beiden folgenden Operationen dagegen verändern die Kollektion.

Union(A,B,C):	Vereinigt die Mengen A und B zur Menge C.

Es wird hier also angenommen, daß die Mengen A und B aus der Kollektion entfernt werden und dafür $C = A \cup B$ neu aufgenommen wird. Für vollständig geordnete Mengen von Schlüsseln kann man in offensichtlicher Weise auch eine Operation *Split* zum Zerteilen einer Menge nach einem bestimmten Schlüssel definieren.

Split(S,x):	Zerteilt die Menge S in zwei Mengen A und B mit: $A = \{y \mid y \in S \text{ und } y \leq x\}$ und $B = \{y \mid y \in S \text{ und } y > x\}$.

Es wird also S aus der Kollektion entfernt und dafür A und B neu aufgenommen. Man nimmt in der Regel an, daß alle Mengen der Kollektion einen eindeutigen Namen besitzen. Ferner wird stillschweigend vorausgesetzt, daß man eine Menge (z.B. als leere oder einelementige Menge) initialisieren kann. Das impliziert insbesondere die Vergabe eines die Menge eindeutig identifizierenden Namens. Das Problem, eine geeignete Implementation für eine Kollektion von Mengen zu finden, so daß sich jede der hier genannten Operationen effizient ausführen läßt, nennen wir das *allgemeine Mengenmanipulationsproblem*. Es wird in Kapitel 6 behandelt.

Ein besonders wichtiger Spezialfall ist der, daß man mit einer Kollektion von lauter einelementigen Mengen startet und dann eine Reihe von Union- und Find-Operationen ausführt. Die Aufgabe, für diesen Fall eine effiziente Implementation zu finden, ist als *Union-Find-Problem* bekannt und jede dazu geeignete Datenstruktur als Union-Find-Struktur. Dieses Problem wird ebenfalls in Kapitel 6 behandelt.

1.7 Skip-Listen

In diesem Abschnitt wird eine mögliche Implementation von Wörterbüchern durch verkettet gespeicherte lineare Listen vorgestellt, die es — anders als die im Abschnitt 1.5.2 diskutierten Varianten — erlaubt, alle drei Wörterbuchoperationen Suchen, Einfügen und Entfernen von Schlüsseln für eine Liste von N Elementen mit hoher Wahrscheinlichkeit in Zeit $O(\log N)$ auszuführen. Diese von W. Pugh [152, 151] vorgeschlagene Datenstruktur mit dem Namen *Skip-Liste* und die zugehörigen Algorithmen zum Suchen, Einfügen und Entfernen sind ein erstes Beispiel für eine *randomisierte* Datenstruktur. Ein weiteres Beispiel bringt Abschnitt 5.3. Der Algorithmus zum Einfügen von Elementen in eine Skip-Liste verwendet einen Zufallsgenerator (Münzwurf). Die Struktur der durch iteriertes Einfügen einer Folge von Schlüsseln in die anfangs leere Liste entstehenden Skip-Liste hängt vom Ausgang zufälliger Münzwürfe ab. Dadurch kann zwar nicht verhindert werden, daß wie im Fall gewöhnlicher sortierter, linearer Listen, vgl. Abschnitt 1.5.2, Strukturen zur Speicherung von N Schlüsseln entstehen, für die das Ausführen einer einzelnen Wörterbuchoperation Zeit $\Omega(N)$ kostet; dieser Fall ist jedoch sehr unwahrscheinlich. Man kann erwarten, daß eine Skip-Liste entsteht, die es erlaubt, Suchen, Einfügen und Entfernen von Schlüsseln in Zeit $O(\log N)$ auszuführen. Wendet man das durch den Zufall (Münzwurf) gesteuerte Einfügeverfahren mehrfach auf dieselbe Schlüsselfolge, jedesmal beginnend mit der anfangs leeren Liste, iteriert an, so ist der Erwartungswert (gemittelt über alle zufälligen Folgen von Münzwürfen) für die zur Ausführung einer Such-, Einfüge- und Entferneoperation erforderliche Zeit in einer Skip-Liste mit N Elementen von der Größenordnung $O(\log N)$.

Wir stellen im folgenden Abschnitt die Struktur und die zugehörigen Algorithmen für die Wörterbuchoperationen vor und analysieren anschließend ihr Laufzeitverhalten.

1.7.1 Perfekte und randomisierte Skip-Listen

Wir nehmen ohne Einschränkung an, daß die Menge der als Wörterbuch zu organisierenden Daten eine Menge ganzzahliger Schlüssel ist. Die durch den jeweiligen Schlüssel identifizierbare „eigentliche“ Information wird also zur Vereinfachung der Darstellung unterdrückt. Um in einer „gewöhnlichen“ sortierten, verkettet gespeicherten linearen Liste einen Schlüssel x zu suchen, muß man die Liste unter Umständen vom Anfang bis zum Ende vollständig durchlaufen, um x zu finden oder festzustellen, daß x in der Liste nicht vorkommt. Die Suche geht offensichtlich schneller, wenn man Elemente überspringen (englisch: skip) kann. Nehmen wir beispielsweise an, daß die Listenelemente die Schlüssel der Reihe nach in aufsteigender Reihenfolge speichern und es nicht nur von jedem Listenelement einen Zeiger auf das nächste, sondern darüberhinaus auch von jedem zweiten Listenelement einen Zeiger auf das übernächste Element gibt. Abbildung 1.13 (a) zeigt eine solche Liste, die die Schlüssel $\{2, 4, 8, 15, 17, 20, 43, 47\}$ speichert.

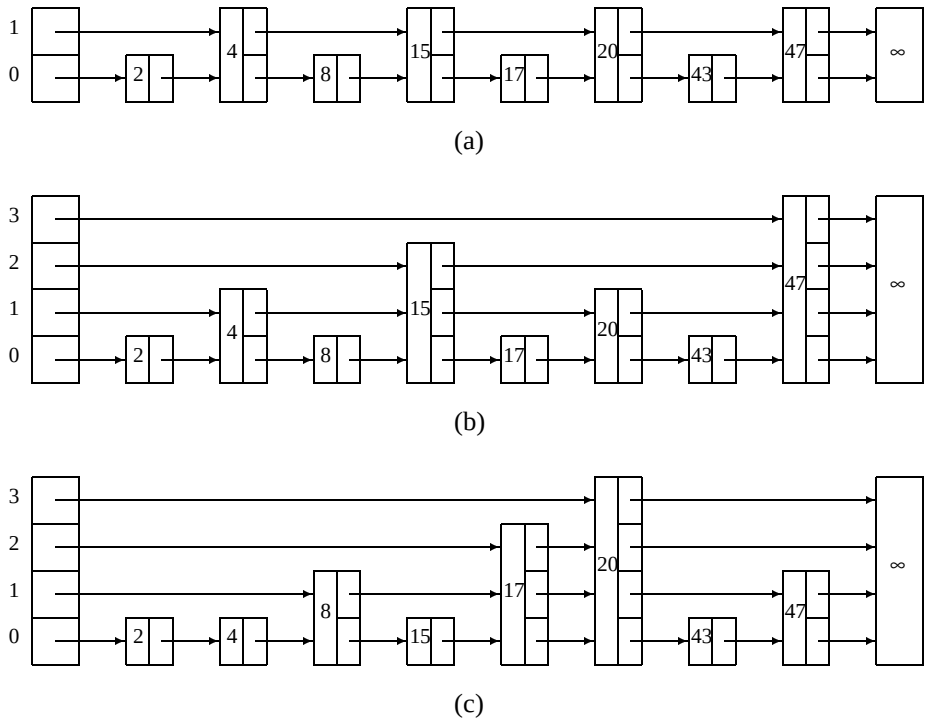


Abbildung 1.13

Jedes Listenelement ist durch einen Zeiger auf Niveau 0 mit dem nächstfolgenden Listenelement verbunden. Ferner ist jedes zweite Listenelement durch einen zusätzli-

chen Zeiger auf Niveau 1 mit dem übernächsten Element verbunden. Am Anfang der Liste befindet sich ein Kopfelement (ohne Schlüssel), das Anfangszeiger auf die Listen der auf Niveau 0 und 1 miteinander verketteten Listenelemente enthält. Am Ende befindet sich ein Endelement mit Schlüssel ∞, der größer als alle in der Liste auftretenden Schlüssel ist. (Es spielt die Rolle eines Stoppers für die Suche.) Um nach einem Schlüssel x zu suchen, folgt man zunächst den Zeigern auf Niveau 1 bis ein Element angetroffen wird, dessen Schlüssel größer als x ist. Dann wechselt man von dem diesem Element in der Niveau-1-Liste unmittelbar vorangehenden Element auf das Niveau 0 und findet dort entweder x oder stellt fest, daß x in der Liste nicht vorkommt. Bei der Suche nach dem Schlüssel 17 werden also in der Liste von Abbildung 1.13 (a), beginnend mit dem Kopfelement, der Reihe nach die Elemente mit den Schlüsseln 4, 15, 20, 17 inspiziert. Man inspiziert also im ungünstigsten Fall nur etwa die Hälfte der Listenelemente. Durch Einführung zusätzlicher Zeiger konnte die Suchzeit in einer verkettet gespeicherten linearen Liste verkürzt werden.

Eine Verallgemeinerung dieser Beobachtung führt zu folgender Definition: Eine *perfekte Skip-Liste* ist eine sortierte, verkettete lineare Liste mit Kopf- und Endelement, für die gilt: Jedes 2^i-te (eigentliche) Element hat einen Zeiger auf das 2^i Positionen weiter rechts stehende Element, für jedes $i = 0, \ldots, \lfloor \log N \rfloor$. Dabei ist N die Anzahl der (eigentlichen) Listenelemente. D.h. jedes Element hat einen Zeiger auf Niveau 0 auf das nächstfolgende; die Elemente an den Positionen 2, 4, 6... sind zusätzlich durch Zeiger auf Niveau 1 miteinander verkettet; die Elemente an den Positionen 4, 8, 12... sind zusätzlich durch Zeiger auf Niveau 2 miteinander verkettet usw. Das Kopfelement enthält Anfangszeiger auf die (aufsteigend sortierten) Niveau-i-Listen, für jedes $i = 0, \ldots, \lfloor \log N \rfloor$; das Endelement hat einen Schlüssel ∞, der größer ist als alle in der Liste gespeicherten Schlüssel. Jedes (eigentliche) Listenelement hat also einen Niveau-0-Zeiger, die Hälfte der Elemente hat zusätzlich einen Niveau-1-Zeiger, ein Viertel zusätzlich einen Niveau-2-Zeiger usw. Nehmen wir zur Vereinfachung einmal an, daß N eine Potenz von 2 ist und zählen wir die vom Kopfelement ausgehenden Zeiger nicht mit, so ist die Gesamtzahl der Zeiger einer perfekten Skip-Liste also

$$N + \frac{N}{2} + \frac{N}{4} + \ldots + 1 = \sum_{i=0}^{\lfloor \log N \rfloor} \frac{N}{2^i} \leq 2N,$$

d.h. nur doppelt so groß wie in einer „gewöhnlichen“ verkettet gespeicherten linearen Liste. Abbildung 1.13 (b) zeigt ein Beispiel für eine perfekte Skip-Liste mit acht Schlüsseln.

Ist N die Anzahl der gespeicherten Schlüssel, so hat jedes Element höchstens $\lfloor \log N \rfloor + 1$ Zeiger. Hat ein Element $p\uparrow$ $i+1$ Zeiger auf den Niveaus $0, \ldots, i$, so sagen wir: $p\uparrow$ ist ein Element mit *Höhe* i. Wir bezeichnen die Höhe von $p\uparrow$ mit $p\uparrow.höhe$.

Für jedes i mit $0 \leq i \leq p\uparrow.höhe$ sei $p\uparrow.next[i]$ der Zeiger von $p\uparrow$ auf das 2^i Positionen weiter rechts stehende Element oder das Endelement, wenn es 2^i Positionen rechts von $p\uparrow$ kein Element mehr gibt. Die maximale Höhe eines Elementes in einer (perfekten) Skip-Liste wird *Listenhöhe* genannt. Dies ist zugleich die Höhe des Kopfelements. Sie hat für eine perfekte Skip-Liste mit N Elementen den Wert $\lfloor \log N \rfloor$. Ist die perfekte Skip-Liste L durch einen Zeiger $L.kopf$ auf das Kopfelement gegeben und hat L die Listenhöhe $L.höhe$, so kann die Suche nach einem Schlüssel x wie folgt beschrieben werden:

```
        function Suchen (x : integer; L : liste) : Zeiger;
        {liefert einen Zeiger auf das Element mit Schlüssel x, falls x in der
         Skip-Liste L mit Zeiger L.kopf auf das Kopfelement und
         Listenhöhe L.höhe vorkommt, und nil sonst}
        var
           p : Zeiger;
           i : integer;
        begin
           p := L.kopf;
           for i := L.höhe downto 0 do
              {folge Niveau-i-Zeigern}
(*)           while p↑.next[i]↑.key < x do
                    p := p↑.next[i];
           {jetzt ist (p = L.kopf und x ≤ p↑.next[0]↑.key) oder
            (p ≠ L.kopf und p↑.key < x ≤ p↑.next[0]↑.key)}
           p := p↑.next[0];
(**)       if p↑.key = x
              then {x kommt an Position p in L vor}
                 Suchen := p
              else {x kommt nicht in L vor}
                 Suchen := nil
        end {Suchen}
```

Verfolgen wir beispielsweise die Suche nach dem Schlüssel $x = 17$ in der perfekten Skip-Liste von Abbildung 1.13 (b), so wird der Schlüssel x der Reihe nach mit den folgenden Schlüsseln verglichen (in den mit $(*)$ und $(**)$ markierten Programmzeilen): 47, 15, 47, 20, 17.

Folgt man also Zeigern der perfekten Skip-Liste nach abnehmenden Niveaus, so muß man einem Zeiger auf Niveau i höchstens einmal folgen. Dabei trifft man dann auf ein Element der Höhe i. D.h. die Anweisung $p := p\uparrow.\text{next}[i]$ wird für festes i höchstens einmal ausgeführt; lediglich die die **while**-Schleife $(*)$ kontrollierende Bedingung kann zweimal geprüft werden, ist aber beim zweiten Mal garantiert nicht erfüllt. Man hätte daher statt der **while**-Schleife auch eine **if**-Anweisung nehmen können, um zu erreichen, daß an der Stelle $(*)$ in der Funktion *Suchen* auf Niveau i vorgerückt wird, bis $x \leq p\uparrow.\text{next}[i]\uparrow.\text{key}$ ist. Die hier angegebene Realisierung des Suchverfahrens für perfekte Skip-Listen kann jedoch unverändert auch für die später erklärten randomisierten Skip-Listen benutzt werden.

Aus der Beschränkung für die Höhe einer perfekten Skip-Liste folgt natürlich sofort, daß das Suchen stets in $O(\log N)$ Schritten ausgeführt werden kann. Einfügen oder Entfernen eines Schlüssels x würde jedoch eine vollständige Reorganisation der perfekten Skip-Liste erfordern und daher $\Omega(N)$ Schritte benötigen. Will man beispielsweise in die perfekte Skip-Liste von Abbildung 1.13 (b) ein neues kleinstes Element mit Schlüssel 1 einfügen, so müssen sämtliche bisherigen Elemente ihre Höhen ändern, um wieder eine perfekte Skip-Liste zu ergeben. Man verzichtet daher auf die Forderung, daß die Höhen aufeinanderfolgender Elemente dem starren Schema perfekter Skip-Listen unterliegen und sorgt vielmehr dafür, daß Elemente mit verschiedenen Höhen etwa im gleichen Ver-

hältnis wie bei perfekten Skip-Listen auftreten, ihre Verteilung innerhalb der Liste aber zufällig erfolgt. Abbildung 1.13 (c) zeigt ein Beispiel einer Skip-Liste, die für jedes i, $0 \leq i \leq 3$, die gleiche Zahl von Elementen mit Höhe i hat wie die perfekte Skip-Liste von Abbildung 1.13 (b), aber in anderer Weise über die Liste verteilt. Solange Elemente mit großen Höhen relativ selten und solche mit niedrigen Höhen dafür häufiger auftreten, kann man erwarten, daß die Suche nach einem Schlüssel x nach dem für perfekte Skip-Listen angegebenen Verfahren nicht nur weiterhin unverändert durchgeführt werden kann, sondern auch effizient bleibt. Das werden wir im Abschnitt 1.7.2 genauer analysieren. Statt also eine perfekte Skip-Liste zu erzeugen, sorgt man lediglich dafür, daß Elemente mit jeweils verschiedenen Höhen im selben Verhältnis auftreten wie in perfekten Skip-Listen, diese aber gleichmäßig und zufällig über die Liste verteilt werden. Dieser Effekt wird dadurch erreicht, daß man beim Einfügen eines Schlüssels x die Höhe $p\uparrow.höhe$ des Elementes p, das x speichert, unabhängig von allen anderen Elementen zufällig wählt im Bereich $[0,maxhöhe]$ und zwar so, daß die Wahrscheinlichkeit dafür, daß $p\uparrow.höhe = i$ ist, gleich $1/2^{i+1}$ ist:

$$\text{prob}(p\uparrow.höhe = i) = \frac{1}{2^{i+1}}, \quad 0 \leq i \leq maxhöhe.$$

Dabei ist *maxhöhe* eine (global festgesetzte) obere Schranke für die Listenhöhe und damit auch für die Höhe jedes einzelnen Elementes. Der Wert von *maxhöhe* wird orientiert an der Listenhöhe einer perfekten Skip-Liste für N Elemente, wobei N groß genug gewählt wird, um alle je auftretenden Elemente in Skip-Listen mit höchstens N Elementen unterbringen zu können. Man wählt also $maxhöhe = \lfloor \log N \rfloor$ für genügend groß gewähltes N. Um die nachfolgende Analyse zu vereinfachen, ignorieren wir allerdings die Höhenbeschränkung und tun so, als könne die Höhe eines Listenelementes beliebig groß werden. Denn die Wahrscheinlichkeit dafür, daß ein Listenelement eine Höhe hat, die $\lfloor \log N \rfloor$ übersteigt, ist so gering, daß wir sie vernachlässigen können.

Die auf diese Weise durch iteriertes Einfügen in die anfangs leere Liste entstehenden *randomisierten* Skip-Listen heißen (entsprechend dem Vorschlag von Pugh [151]) einfach *Skip-Listen*.

Nehmen wir also an, wir hätten eine parameterlose Funktion *randomhöhe*(), die bei ihrem Aufruf eine Höhe mit den genannten Eigenschaften liefert. Dann können wir das *Einfügen* eines neuen Schlüssels x in eine Skip-Liste L mit Kopfzeiger $L.kopf$ und Höhe $L.höhe$ wie folgt beschreiben. Wir suchen zunächst nach x. Da wir annehmen, daß x in der Skip-Liste noch nicht vorkommt, endet die Suche erfolglos beim Element mit dem größten Schlüssel, der kleiner oder gleich x ist. (Falls x kleiner als alle Schlüssel in Liste L ist, endet die Suche schon beim Kopfelement.) Hinter dieses Element wird ein neues Element $p\uparrow$ mit Schlüssel x und zufällig gewählter Höhe $p\uparrow.höhe$ eingeschoben. Es müssen dazu alle über $p\uparrow$ hinwegführenden Niveau-i-Zeiger, mit $0 \leq i \leq p\uparrow.höhe$ verändert werden. Damit das möglich ist, sammelt man während der Suche nach x die Quellen aller dieser Zeiger in einem Zeiger-Array *update*: Für jedes i enthält $update[i]$ einen Zeiger auf das am weitesten rechts liegende Listenelement mit Höhe i links von der Einfügestelle. Ist die $p\uparrow$ zufällig zugewiesene Höhe $p\uparrow.höhe$ größer als die bisherige Listenhöhe $L.höhe$, müssen das Kopfelement durch zusätzliche Zeiger auf $p\uparrow$ und $L.höhe$ entsprechend verändert werden. Genauer kann das Verfahren wie folgt beschrieben werden:

```
procedure Einfügen (x : integer; var L : Liste);
{fügt Schlüssel x in Skip-Liste L mit Zeiger L.kopf auf das
 Anfangselement und Listenhöhe L.höhe ein}
var
   update : array [0 .. maxhöhe] of Zeiger;
   p : Zeiger;
   i : integer;
   neuehöhe : 0 .. maxhöhe;
begin
   p := L.kopf
   for i := L.höhe downto 0 do
      begin
         while p↑.next[i]↑.key < x do
            p := p↑.next[i];
         update[i] := p
      end {for};
   p := p↑.next[0];
   if p↑.key = x
      then {Schlüssel x kommt schon vor}
      else {einfügen}
         begin
         neuehöhe := randomhöhe();
         if neuehöhe > L.höhe
            then
               begin
                  {neues Element direkt mit Kopfelement verknüpfen
                   und Listenhöhe adjustieren}
                  for i := L.höhe + 1 to neuehöhe do
                     update[i] := L.kopf;
                  L.höhe := neuehöhe
               end;
         {schaffe neues Element mit Höhe neuehöhe und Schlüssel x}
         new(p);
         p↑.höhe := neuehöhe; p↑.key := x;
         for i := 0 to neuehöhe do
         {schiebe p↑ in die Niveau-i-Listen jeweils unmittelbar nach
          dem Element update[i]↑ ein}
            begin
               p↑.next[i] := update[i]↑.next[i];
               update[i]↑.next[i] := p
            end
         end
end {Einfügen}
```

Das *Entfernen* eines Elementes mit Schlüssel x aus einer Skip-Liste L erfolgt völlig analog: Zunächst sucht man nach x. Dabei benutzt man wieder ein Array *update* und merkt sich für jedes i in $update[i]$ einen Zeiger auf das rechteste Element in L links von

x mit Höhe i. Dann kann man das Element $p\uparrow$ mit Schlüssel x aus allen Niveau-i-Listen, $0 \leq i \leq p\uparrow.höhe$, entfernen. Falls nach Entfernen von $p\uparrow$ die Listenhöhe gesunken ist, muß man sie entsprechend adjustieren. Um festzustellen, ob dieser Fall vorliegt, muß man dem Zeiger des Kopfelementes auf dem höchsten Niveau folgen und nachsehen, ob er noch auf ein eigentliches Listenelement oder auf das Endelement mit Schlüssel ∞ zeigt. D.h. die Listenhöhe kann um 1 verringert werden, wenn gilt

$$L.kopf\uparrow.next[L.höhe]\uparrow.key = \infty.$$

Genauer kann das Verfahren zum Entfernen eines Schlüssels x aus einer Skip-Liste L wie folgt beschrieben werden:

```
procedure Entfernen (x : integer; var L : Liste);
var
  update : array [0 .. maxhöhe] of Zeiger;
  p : Zeiger;
  i : integer;
begin
  p := L.kopf;
  for i := L.höhe downto 0 do
    begin
      while p↑.next[i]↑.key < x do
        p := p↑.next[i];
      update[i] := p
    end; {for}
  p := p↑.next[0];
  if p↑.key = x

    then
      {Element p↑ entfernen und ggfs. Listenhöhe adjustieren}
    begin
      for i := 0 to p↑.höhe do
        {entferne p↑ aus Niveau-i-Liste}
        update[i]↑.next[i] := p↑.next[i];
      while (L.höhe ≥ 1) and (L.kopf↑.next[L.höhe]↑.key = ∞) do
        L.höhe := L.höhe −1
    end
end {Entfernen}
```

Die Verfahren zum Einfügen und Entfernen von Elementen in Skip-Listen haben eine Eigenschaft, die sie von den entsprechenden Verfahren für die im Kapitel 5 ausführlich behandelten Suchbäume, insbesondere von den Verfahren für natürliche Bäume, ganz wesentlich unterscheidet: Eine Skip-Liste, aus der ein Element entfernt wurde, hat dieselbe Struktur, als wäre das Element niemals dagewesen. Daher bleibt auch nach einer längeren Folge von Updates die „Zufälligkeit“ der Struktur erhalten. In diesem Sinne sind Skip-Listen unabhängig von der Erzeugungshistorie. Anders als etwa natürliche Suchbäume können Skip-Listen durch iteriertes Einfügen und Entfernen von Elementen nicht „degenerieren“.

1.7.2 Analyse

Das Einfügeverfahren für Skip-Listen benutzt eine Funktion *randomhöhe*(), die eine zufällige Höhe erzeugt und zwar so, daß gilt: Die Wahrscheinlichkeit dafür, daß die Höhe 0 erzeugt wird, ist $1/2$ und für jedes $i \geq 0$ ist die Wahrscheinlichkeit dafür, daß die Höhe $i+1$ erzeugt wird, halb so groß wie die, daß die Höhe i erzeugt wird. Also ist die Wahrscheinlichkeit dafür, daß genau die Höhe i erzeugt wird, gleich $1/2^{i+1}$, und die Wahrscheinlichkeit dafür, daß eine Höhe $\geq i$ erzeugt wird, gleich $1/2^i$, für jedes $i \geq 0$. Bei der Implementation des Einfügeverfahrens haben wir zur Vereinfachung zusätzlich vorausgesetzt, daß die von *randomhöhe*() gelieferte Höhe stets kleiner oder gleich einer global festgesetzten maximalen Höhe *maxhöhe* bleibt. Weil die Wahrscheinlichkeit, ein Element mit einer Höhe von etwa 15 zu erzeugen, schon „praktisch“ gleich Null ist, werden wir in der Analyse von dieser globalen Höhenbeschränkung der Einfachheit halber zunächst absehen. Zur Realisierung von *randomhöhe*() setzen wir eine parameterlose Funktion *random*() voraus, die unabhängige und gleichverteilte Zufallszahlen im Bereich $(0,1)$ liefert. Dann erzeugt die folgende Funktion *randomhöhe*() Höhen im Bereich $[0, \mathit{maxhöhe}]$ mit exponentiell (mit dem Faktor $1/2$) abnehmenden Wahrscheinlichkeiten.

```
function randomhöhe() : integer;
var
  höhe : integer;
begin
  höhe := 0;
  while (random() < 1/2) and (höhe < maxhöhe) do
    höhe := höhe +1;
  randomhöhe := höhe
end
```

Erzeugt man die Höhen mit dieser Funktion *randomhöhe*(), so ist der Erwartungswert für die Anzahl der Elemente mit Höhe i oder größer in einer Liste mit N Elementen gleich $N/2^i$, für jedes $i \geq 0$. Die Höhenverteilung der Elemente stimmt also mit der von perfekten Skip-Listen überein.

Wir schätzen nun die *Suchkosten* in einer Skip-Liste ab. Nach dem in Abschnitt 1.7.1 angegebenen Verfahren beginnen wir die Suche beim Kopfelement der Liste und führen dann jeweils einen der folgenden beiden Schritte aus: Entweder folgen wir einem Zeiger auf dem gerade aktuellen Niveau von einem Element zum nächstfolgenden oder aber wir gehen innerhalb eines Elementes von einem Niveau zum nächstniedrigeren über. Tritt der erste Fall ein, d.h. folgen wir einem Zeiger auf Niveau i, so hat das Element, auf das dieser Zeiger zeigt, die Höhe i, für jedes $i \geq 0$. Natürlich gibt es auch Zeiger auf Höhe i, die auf Elemente mit Höhe $> i$ zeigen, aber denen *folgt* unser Algorithmus nicht (er *prüft* sie höchstens), weil für solche Zeiger ein Zeiger auf dasselbe Element mit größerer Höhe ebenfalls bereits ge*prüft* wurde. Das Entlanglaufen von Niveau-i-Zeigern zu Elementen mit Höhe i, $i \geq 0$, und das Herabsetzen des aktuellen Niveaus wird solange durchgeführt, bis das Niveau 0 erreicht ist und dort der gesuchte Schlüssel gefunden wird oder aber festgestellt wird, daß der gesuchte Schlüssel in der Skip-Liste nicht vorkommt. Abbildung 1.14 zeigt ein Beispiel eines solchen Suchpfades nach dem Schlüssel 16 in der Skip-Liste von Abbildung 1.13 (c).

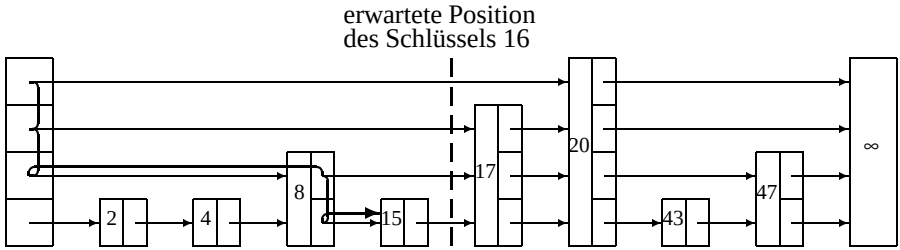


Abbildung 1.14

Um den Erwartungswert für die Länge des Suchpfades zu berechnen, verfolgen wir den Suchpfad rückwärts, beginnend beim Niveau-0-Zeiger auf das Element, das den gesuchten Schlüssel enthält oder das, falls der gesuchte Schlüssel nicht vorkommt, den kleinsten Schlüssel enthält, der größer als der gesuchte ist. Dazu nehmen wir allgemeiner an, daß wir uns innerhalb eines Elementes $p\uparrow$ auf dem Niveau i befinden, $i \geq 0$, und fragen uns, was der Erwartungswert $EC(k)$ für die Länge eines Suchpfades ist, der vom Niveau i in $p\uparrow$ aus gerechnet nach links zurückverfolgt k Niveaus hinaufsteigt. $EC(k)$ ist also die Anzahl der Schritte, die man vom Niveau i in $p\uparrow$ aus beim Zurückverfolgen des Suchpfades benötigt, um erstmals auf ein k Niveaus über Niveau i liegendes Niveau zu gelangen. Als Schritt zählen wir dabei jeweils das Heraufklettern um ein Niveau und das Zurücklaufen eines Niveau-i-Zeigers von einem Element mit Höhe i zu seinem Ursprung. Wir machen keine Annahmen über die Höhe von $p\uparrow$ oder die Höhen der Elemente links von $p\uparrow$. Wir setzen allerdings voraus, daß $p\uparrow$ nicht das Kopfelement der Skip-Liste ist. (Diese letzte Annahme ist gleichbedeutend mit der Annahme, daß die Liste nach links unbegrenzt ist.)

Wir haben angenommen, daß wir uns in $p\uparrow$ auf Niveau i befinden. Also ist $p\uparrow.höhe \geq i$ und mit Wahrscheinlichkeit von jeweils $1/2$ ist $p\uparrow.höhe = i$ und $p\uparrow.höhe > i$ aufgrund unseres Verfahrens zur Erzeugung einer zufälligen Höhe. D.h. sind wir mit unserem durch zufällige Münzwürfe gesteuerten Heraufsetzen der Höhe bereits bis zur Höhe i gekommen, so ist die Wahrscheinlichkeit dafür, daß wir aufhören oder fortfahren, die Höhe hinaufzusetzen, jeweils $1/2$.

Fall 1: $i = p\uparrow.höhe$. Das impliziert, daß der zurückverfolgte Suchpfad vom Element $p\uparrow$ zu einem Element mit wenigstens Höhe i geht und von diesem Element noch immer k Niveaus hinaufklettern muß.

Fall 2: $i < p\uparrow.höhe$. Das impliziert, daß der zurückverfolgte Suchpfad $p\uparrow$ wenigstens ein Niveau hinaufklettert und nicht einen Niveau-i-Zeiger zurückläuft. Also muß der Suchpfad von diesem neuen Niveau $i+1$ aus gerechnet noch $k-1$ Niveaus hinaufsteigen, um beim Zurückverfolgen insgesamt k Niveaus hinaufzusteigen.

Damit erhalten wir die folgende Rekursionsgleichung für $EC(k)$:

$$\begin{aligned} EC(k) &= \frac{1}{2} \cdot ((\text{Kosten, um einen Niveau-}i\text{-Zeiger zurückzulaufen}) + EC(k)) \\ &+ \frac{1}{2} \cdot ((\text{Kosten, um vom Niveau } i \text{ zu Niveau } i+1 \text{ hinaufzusteigen}) \end{aligned}$$

$$
\begin{aligned}
& + EC(k-1)) \\
= \quad & \frac{1}{2} \cdot (1 + EC(k)) + \frac{1}{2}(1 + EC(k-1))
\end{aligned}
$$

Es gilt also

$$EC(k) = 2 + EC(k-1).$$

Da die Länge eines kürzesten Pfades, der beim Zurückverfolgen 0 Niveaus hinaufklettert, natürlich 0 ist, gilt

$$EC(0) = 0.$$

Die Rekursionsformel hat die Lösung

$$EC(k) = 2k.$$

Wir verwenden dieses Ergebnis, um den Erwartungswert für die Länge eines Suchpfades in einer Skip-Liste mit Länge N zu berechnen. Dazu zerlegen wir den (zurückverfolgten) Suchpfad in drei Teile.

Teil 1: Zuerst betrachten wir den Teil des Suchpfades, den man ausgehend vom Niveau 0 im Element mit dem gesuchten Schlüssel zurücklaufen muß, um $\log_2 N - 1$ Niveaus hinaufzusteigen. Den Erwartungswert für die Länge dieses Teils des Suchpfades haben wir gerade berechnet. Er ist $EC(\log_2 N - 1) = 2(\log_2 N - 1)$.

Teil 2: Dann schätzen wir ab, wieviele Knoten mit Höhe wenigstens $\log_2 N - 1$ es in der Skip-Liste höchstens gibt. Denn sind wir beim Zurückverfolgen des Suchpfades bereits auf Niveau $\log_2 N - 1$ angekommen, so wird man im weiteren Verlauf sicher noch höchstens so viele Zeiger zurückverfolgen müssen, wie es insgesamt Knoten mit Höhe mindestens $\log_2 N - 1$ in der Skip-Liste gibt. Offenbar ist der Erwartungswert der Anzahl der Elemente mit Höhe mindestens $\log_2 N - 1$ gleich dem Produkt aus der Anzahl N der Listenelemente und der Wahrscheinlichkeit dafür, daß ein Listenelement die Höhe mindestens $\log_2 N - 1$ hat, also höchstens

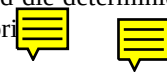
$$N \cdot \left(\frac{1}{2}\right)^{\log_2 N - 1} = N \left(\frac{1}{2}\right)^{\log_2 N} \cdot 2 = 2.$$

Teil 3: Schließlich schätzen wir ab, wieviele Niveaus man noch vom Niveau $\log_2 N - 1$ bis zur Listenhöhe, also bis zur Höhe des Kopfelementes, hinaufsteigen muß. Wir haben die Listenhöhe willkürlich global beschränkt durch *maxhöhe*, also eine nicht allzu weit oberhalb von $\log_2 N - 1$ liegende Konstante. Man kann allerdings auch ohne diese Beschränkung argumentieren und (durch einen nicht ganz einfachen Beweis) zeigen, daß der Erwartungswert für die Höhe einer Skip-Liste mit N Elementen gleich $\log_2 N + 1$ ist, wenn man die Höhenbeschränkung fallen läßt. Nehmen wir an, daß der Erwartungswert für die Differenz zwischen der Listenhöhe und $\log_2 N - 1$ gleich 2 ist, dann ergibt sich insgesamt als obere Schranke für die Suchkosten der Wert

$$2(\log N - 1) + 2 + 2 = O(\log N).$$

Es ist klar, daß auch die Kosten für das *Einfügen* und *Entfernen* von Elementen in Skip-Listen von derselben Größenordnung sind.

Wir haben hier nur eine obere Schranke für die Kosten der drei Wörterbuchoperationen Suchen, Einfügen und Entfernen hergeleitet. In [143] ist der Erwartungswert für die Kosten exakt berechnet worden. Das Ergebnis zeigt, daß die oben angegebene Abschätzung recht scharf ist. Kürzlich haben Munro und Papadakis [127] eine determinierte Variante von Skip-Listen vorgestellt, die es erlaubt, alle drei Wörterbuchoperationen stets, also auch im schlechtesten Fall, in Zeit $O(\log N)$ auszuführen. Diese Struktur hat Ähnlichkeiten mit den im Abschnitt 5.2 eingeführten balancierten Bäumen. Anders als die in diesem Abschnitt eingeführten Skip-Listen sind die determinierten Skip-Listen aber nicht mehr unabhängig von der Entstehungshistori



1.8 Aufgaben

Aufgabe 1.1

Für welche der folgenden Paare von Funktionen f und g gilt $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ bzw. $f(n) = \Theta(g(n))$ für natürliches Argument n? Dabei soll stets $[x]$ den ganzzahligen Anteil von x bezeichnen.

(i)	$f(n) = [\sqrt{n}]$;	$g(n) = 1000n$
(ii)	$f(n) = [\log_{10} n]$;	$g(n) = [\log_2 n]$
(iii)	$f(n) = [\sqrt[3]{n}]$;	$g(n) = [\sqrt{n}]$
(iv)	$f(n) = n^2$;	$g(n) = [n \log n]$
(v)	$f(n) = 176n^2 - 36n + 17$;	$g(n) = n^2$
(vi)	$f(n) = [n \log n] + [\sqrt{n}]$;	$g(n) = [n \log^2 n]$

Aufgabe 1.2

Ein Polynom vom Grade N läßt sich auch schreiben in der Form

$$p(x) = r_0(x - r_1)(x - r_2)\ldots(x - r_N).$$

Leiten Sie aus dieser Schreibweise eine mögliche Form zur Repräsentation von Polynomen ab und beschreiben Sie, wie zwei Polynome bei der gewählten Repräsentation miteinander multipliziert werden. Wie würden Sie zwei Polynome bei dieser Repräsentation addieren?

Aufgabe 1.3

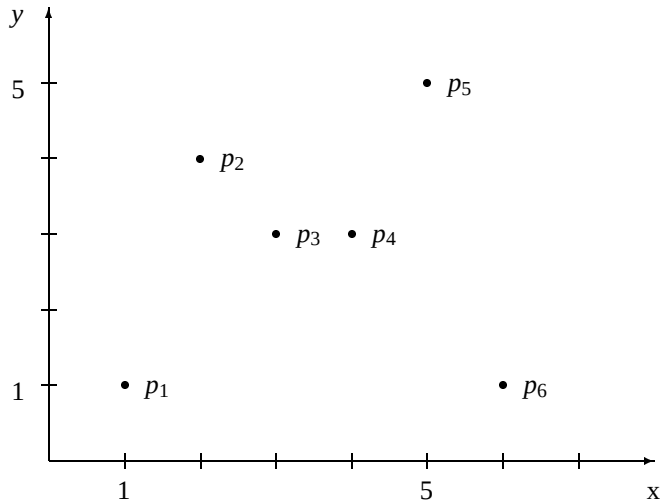
Seien u und v zwei Dualzahlen der Länge N, wobei $N = 2^n$ eine Zweierpotenz sei. Überzeugen Sie sich davon, daß das „Schulverfahren“ zur Multiplikation $O(N^2)$ Schritte benötigt. Entwerfen Sie dann ein Divide-and-conquer-Verfahren zur Berechnung des Produkts analog zum Verfahren zur Berechnung des Produkts zweier Polynome und analysieren Sie die Laufzeit des Verfahrens.

Aufgabe 1.4

Im $\mathbb{N} \times \mathbb{N}$-Gitter sei eine Menge M von Punkten mit paarweise verschiedenen x-Werten gegeben. Die Dominanzzahl $dz(p, M)$ eines Punktes $p \in M$ bezüglich M ist die Anzahl aller Punkte aus M, die links unterhalb von p liegen, also

$$dz(p, M) = \#\{q = (x_q, y_q) \in M \mid x_q < x_p \wedge y_q \leq y_p\} \text{ für } p = (x_p, y_p).$$

Beispiel:



Für $M = \{p_1, \ldots, p_6\}$ ist

$$dz(p_1, M) = 0,\ dz(p_2, M) = 1,\ dz(p_3, M) = 1,\ dz(p_4, M) = 2,$$
$$dz(p_5, M) = 4),\ dz(p_6, M) = 1.$$

a) Eine Möglichkeit zur Bestimmung der Dominanzzahlen aller Punkte einer Menge M ist das folgende, der Divide-and-conquer-Strategie folgende Verfahren:

DZ-Bestimmung $(M : Punktmenge)$

Besteht M nur aus einem einzigen Element p, dann ist $dz(p, M) = 0$, sonst:

1. {Divide} Wähle einen x-Wert x_0 so, daß die vertikale Gerade $x = x_0$ die Menge M in zwei nahezu gleichgroße Teilmengen M_1 und M_2 aufteilt. M_1 sei dabei die Menge mit den kleineren x-Werten.
2. {Conquer}
   2.1 Bestimme die Dominanzzahlen aller Punkte in M_1 bezüglich M_1 durch *DZ-Bestimmung* (M_1).
   2.2 Bestimme die Dominanzzahlen aller Punkte in M_2 bezüglich M_2 durch *DZ-Bestimmung* (M_2).
3. {Merge} Sortiere die Elemente in $M = M_1 \cup M_2$ nach aufsteigenden y-Werten zu einer Folge $p_1, \ldots, p_n$ von Punkten. Bei gleichen y-Werten ordne die Elemente aus M_1 vor denen aus M_2 ein. Setze $M1Count := 0$, und durchlaufe die Punkte gemäß der Sortierung wie folgt:

   for $i := 1$ **to** n **do**
   if $p_i \in M_1$
   then

```
        begin
          M1Count := M1Count + 1;
          dz(p_i, M) := dz(p_i, M_1);
        end
      else {p_i ∈ M_2}
        dz(p_i, M) := dz(p_i, M_2) + M1Count
```

Stellen Sie eine Rekursionsformel auf für

$T(N)$ = Anzahl der Schritte, die zur Bestimmung der Dominanzzahlen einer Menge mit N Elementen nach dem Verfahren *DZ-Bestimmung* benötigt wird.

Begründen Sie diese und geben Sie eine Lösung der Rekursionsformel an. Nehmen Sie dabei an, daß N Zahlen in $N \log N$ Schritten sortiert werden können, und beachten Sie b).

b) Zeigen Sie, daß die Rekursionsformel

$$T(N) = 2 \cdot T(\frac{N}{2}) + c_o \cdot N \cdot \log^k N + c_1 N$$

die Lösung

$$T(N) = O(N \cdot \log^{k+1} N)$$

hat.

c) Geben Sie ein anderes als das in a) angegebene Verfahren an, das zu gegebener Menge M und gegebenem Punkt $p \in M$ die Zahl $dz(p,M)$ berechnet. Schätzen Sie die Laufzeit ab, wenn mit diesem Verfahren die Dominanzzahlen aller Punkte p aus M bezüglich M berechnet werden.

Aufgabe 1.5
(Bundeswettbewerb Informatik 1985 [77])

Ein großes Wirtschaftsmagazin will seinen Lesern eine Analyse der Börsenentwicklung der letzten fünf Jahre präsentieren. Dazu sollen unter anderem die Kurse der wichtigsten Aktien in diesem Zeitraum untersucht werden. Für jede Aktie soll nachträglich ein bester Einkaufstag festgestellt werden.

Dabei wird angenommen, daß ein Kapitalanleger jede Aktie höchstens einmal eingekauft hätte, und zwar in einer beliebigen Stückzahl, und daß er zum Ende des betrachteten Zeitraums alle Stücke wieder verkauft hat. Der beste Einkaufstag für eine Aktie wäre dann derjenige gewesen, der zu einem eingesetzten Betrag den höchsten Gewinn geliefert hätte (Steuern, Gebühren und alternative Anlagemöglichkeiten sollen außer Betracht bleiben).

Das Wirtschaftsmagazin hat von einem Börsendienst Informationen über die Notierungen jeder Aktie für alle Börsentage der letzten fünf Jahre gekauft. Für jede Aktie erhält es eine Zahlenfolge. Die erste Zahl ist der Kurs der Aktie am ersten Börsentag und jede folgende Zahl gibt die absolute Kursveränderung gegenüber dem Vortag an, in der Reihenfolge der Börsentage. Der Kurs, der sich für einen gewissen Tag ergibt, gilt für alle Käufe und Verkäufe dieses Tages.

Unterstützen Sie die Kursanalyse durch Schreiben eines Programms, das für eine Aktie aus der gegebenen Zahlenfolge nachträglich einen besten Einkaufstag, einen besten Verkaufstag und den dabei höchsten erzielbaren Gewinn (in Prozent vom eingesetzten Betrag) ermittelt. Da das Programm sehr lange Zahlenfolgen bearbeiten muß, ist es außerordentlich wichtig, daß die Laufzeit bei zunehmender Zahlenfolgenlänge nicht stärker als nötig wächst.

Beispiel: Die Eingabe

„127.5 -0.5 2 -1 1 3.5 -13 7 -2 -6 -9 -21 -17 -5 0.5 4 -7 -12 2.5 -3 2“

liefert die Ausgabe:

„Ein bester Einkaufstag wäre der 14. Börsentag gewesen, ein dazugehöriger Verkaufstag der 16. Börsentag. Der so realisierbare Gewinn wäre 6.7669 % vom eingesetzten Betrag gewesen.“

Aufgabe 1.6

Gegeben sei ein lineares Feld positiver reeller Zahlen, in Pascal beschrieben durch die folgenden Vereinbarungen:

```
const n = {irgendeine positive Zahl, z.B.} 500;
type feld = array [1 .. n] of real;
var a : feld
```

Gegeben seien außerdem eine Funktion $g : \mathbb{R} \to \{0,1\}$, die als Werte 0 oder 1 liefert, und die folgende Funktion *gtest*:

```
function gtest (li, re: integer) : integer;
var m : integer;
begin
  if li > re
    then gtest := 0
    else
      begin
        m := (li + re) div 2;
        gtest := gtest(li, m − 1) + g(a[m]) + gtest(m + 1, re)
      end
end
```

a) Beschreiben Sie, welches Resultat die Funktion beim Aufruf $gtest(1,n)$ für ein gegebenes Feld *a* liefert.

b) Ermitteln Sie größenordnungsmäßig die Anzahl der Additionen bei der Ausführung eines Aufrufs von $gtest(li,re)$ im schlimmsten Fall, in Abhängigkeit von $|re - li|$, mit Hilfe einer Rekursionsformel.

c) Geben Sie in Pascal ein alternatives (iteratives) Verfahren zur Ermittlung des Funktionswertes *gtest* an; verwenden Sie denselben Funktionskopf.

Aufgabe 1.7

Das maximale Element in einem linearen Feld kann auf folgende Weise bestimmt werden.

```
program Maximum (input, output);
const N = {eine feste Zahl};
type feld = array[1 .. N] of integer;
var a : feld;
procedure max (var a : feld; i, j : integer; var m : integer);
{bestimmt das maximale Element im Bereich a[i], ..., a[j]
 und weist es m zu}
var m1, m2, mitte : integer;
begin
  if i = j
    then m := a[i]
    else
      if i = j - 1
        then
          begin
            if a[i] < a[j]
              then m := a[j]
              else m := a[i]
          end
        else
          begin
            mitte := (i + j) div 2;
            max(a, i, mitte, m1);
            max(a, mitte+1, j, m2);
            if m1 < m2
              then m := m2
              else m := m1
          end
end; {max}


begin {Maximum}
  {Eingabe der Werte von a[1], ..., a[N]}
  max (a, 1, N, m)
end {Maximum}.
```

a) Berechnen Sie die Anzahl der Vergleichsoperationen, die zwischen Elementen des Feldes ausgeführt werden, durch Aufstellen und Lösen einer Rekursionsgleichung.

b) Vergleichen Sie das angegebene Verfahren mit dem „naiven“ Verfahren zur Bestimmung des Maximums.

c) Ändern Sie das Verfahren so ab, daß zugleich das maximale und das minimale Element des linearen Feldes bestimmt wird und ermitteln Sie ebenfalls die Anzahl der ausgeführten Vergleichsoperationen zwischen Feldelementen.

Aufgabe 1.8

Gegeben sei eine sortierte Liste L voneinander verschiedener ganzer Zahlen in sequentieller Speicherung. Gegeben sei außerdem eine Zahl x. Gesucht ist das größte Element in L, das $\leq x$ ist. Die Länge der Liste L sei N.

a) Geben Sie in verbaler Beschreibung einen Algorithmus an, der diese Aufgabe in logarithmischer Schrittzahl löst.

b) Folgende Pascal-Vereinbarungen seien gegeben:

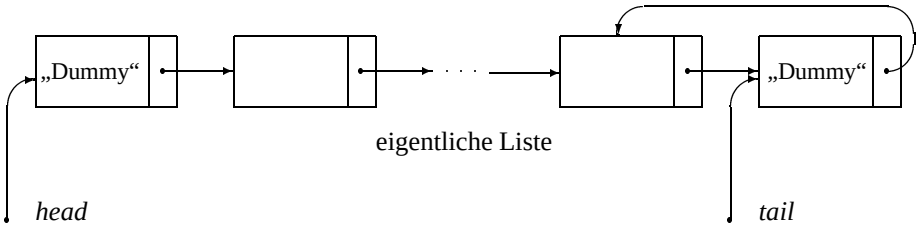
```
const N = {z.B.} 500;
type feld = array [1 .. N] of integer
```

Schreiben Sie eine Funktion zu dem in a) entwickelten Algorithmus.

```
function suche (var liste: feld; x, li, re: integer) : integer;
{sucht das größte Element ≤ x im Bereich liste[li .. re]}
```

Aufgabe 1.9

Gegeben sei eine nichtleere verkettete lineare Liste L ganzer Zahlen mit ungerader Elementzahl. Die Liste beginnt und endet mit je einem bedeutungslosen Dummy-Element, das einen beliebigen Wert haben kann. Zwischen den beiden Dummy-Elementen befinden sich die eigentlichen Listenelemente.



Die Struktur der Knoten der Liste in Pascal wie folgt gegeben.

```
type  Zeiger = ↑Knoten;
      Knoten =    record
                    key : integer;
                    next : Zeiger
                  end
```

a) Schreiben Sie eine Prozedur

```
procedure mitte (head, tail : Zeiger);
```

die das Element an der mittleren Position der Liste entfernt.

b) Schreiben Sie eine Prozedur

procedure *teilen* (**var** *head, headeven, headodd* : *Zeiger*);

die die Liste L mit Anfangszeiger *head* aufteilt in zwei (anfangs leere, also durch je zwei Dummy-Elemente gegebene) Listen mit Anfangszeiger *headeven* bzw. *headodd*. In die eine Liste sollen die Elemente aus L mit geradzahligem Eintrag gehängt werden, in die andere die Elemente mit ungeradzahligem Eintrag. Aufrufe von *new* sind dabei nicht erlaubt.

c) Schreiben Sie eine Prozedur

procedure *umdrehen* (*head, tail* : *Zeiger*);

die die Reihenfolge der Elemente, d.h. die Zeiger, in der Liste umdreht, ohne eine zweite Liste zu verwenden (d.h. Aufrufe von *new* sind nicht erlaubt).

Aufgabe 1.10

Schreiben Sie ein Programm, das ein Element mit gegebenem Schlüssel aus einer gekettet gespeicherten linearen Liste mit Dummy-Elementen am Anfang und Ende entfernt. Verwenden Sie nicht die Technik des „Zurückhängens mit Vorausschauen“, sondern verfahren Sie wie folgt. Durchsuchen Sie die Liste nach dem zu entfernenden Element. Ersetzen Sie das zu entfernende Element durch dessen Nachfolger in der Liste und entfernen Sie diesen. Achten Sie auf mögliche Sonderfälle wie Entfernen des ersten oder letzten Elementes, und schätzen Sie den Aufwand ab.

Aufgabe 1.11

Schreiben Sie ein Programm, das ein Element mit gegebenem Schlüssel aus einer aufsteigend sortierten linearen Liste, die verkettet gespeichert ist, entfernt. Schätzen Sie den Aufwand ab.

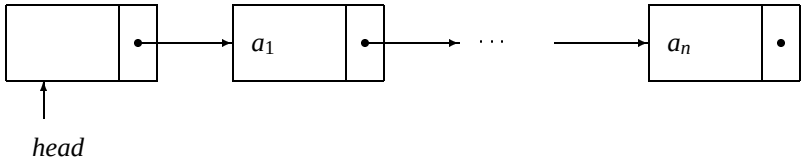
Aufgabe 1.12

Gegeben sei eine verkettet gespeicherte lineare Liste mit Anfangszeiger *head* und Listenelementen des in Aufgabe 1.9 vereinbarten Typs. Die *key*-Komponenten seien entlang der Verkettung aufsteigend sortiert. Das Ende der Liste ist durch ein Listenelement gekennzeichnet, dessen *next*-Komponente den Wert **nil** hat. Schreiben Sie eine Prozedur *Teile* mit folgenden Eigenschaften. Aus der *head*-Liste werden alle Listenelemente, deren *key*-Komponente kleiner als ein gegebener Schlüssel k ist — und nur diese — an die anfangs leere *kleiner*-Liste übergeben und dabei aus der *head*-Liste entfernt. Es kann vorausgesetzt werden, daß die *head*-Liste zuvor sowohl Elemente mit *key*-Komponente $< k$ als auch solche mit *key*-Komponente $\geq k$ enthält. Verwenden Sie folgenden Prozedurkopf:

procedure *Teile* (*k* : *integer*; **var** *head, kleiner*: *Zeiger*);

Aufgabe 1.13

Gegeben sei eine verkettet gespeicherte Liste durch einen Zeiger auf das Kopfelement *head*, der Typ der Elemente sei wie in Aufgabe 1.9 vereinbart.



Position *i* sei implementiert als ein Zeiger auf das Element, dessen *next*-Zeiger auf ein Element mit Schlüssel a_i zeigt. Schreiben Sie eine Prozedur, die die Elemente an den Positionen *p* und $p\uparrow.next$ miteinander vertauscht, wenn $p\uparrow.next <>$ **nil** ist.

Aufgabe 1.14

Erstellen Sie zwei Pascal-Programme zur iterativen Berechnung der folgenden verallgemeinerten Binomialkoeffizienten.

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 2 & & & \\ & & 1 & & 3 & & 4 & & \\ & 1 & & 4 & & 7 & & 8 & \\ 1 & & 5 & & 11 & & 15 & & 16 \end{array}$$

Das allgemeine Bildungsgesetz lautet $\binom{d}{0} = 1$, $\binom{d}{d} = 2^d$ und $\binom{d}{h} = \binom{d-1}{h} + \binom{d-1}{h-1}$.

a) Das erste Pascal-Programm verwende einen Stapel, der verkettet gespeichert, also über Zeiger realisiert wird.

b) Das zweite Pascal-Programm verwende ein Berechnungsschema, das mehrfache Berechnung von gleichen Teilresultaten vermeidet.

Aufgabe 1.15

Einfache, vollständig geklammerte, arithmetische Ausdrücke können wie folgt definiert werden.

(1) Jede Variable $a, b, c \ldots$ ist ein einfacher, vollständig geklammerter, arithmetischer Ausdruck.

(2) Mit α und β sind auch $(\alpha + \beta)$, $(\alpha - \beta)$, $(\alpha * \beta)$, (α / β) einfache, vollständig geklammerte, arithmetische Ausdrücke.

(3) Sonst nichts.

Geben Sie ein Verfahren zur Auswertung von Ausdrücken mit Hilfe eines Stapels an, das mit Hilfe eines Stapels Variablen, Operatoren und Zwischenergebnisse speichert. Das Verfahren soll nur auf die für Stapel üblichen Operationen, aber nicht auf eine konkrete Implementation Bezug nehmen.

Beispiel: Die Auswertung des Ausdrucks

$$(c + ((a + b) * a))$$

erzeugt bei Auswertung (d.h. Lesen von links nach rechts) folgende Stapelbelegungen (jede Zeile ist eine Stapelbelegung, das oberste Element steht jeweils rechts).

c
$c, +$
$c, +, a$
$c, +, a, +$
$c, +, a, +, b$
$c, +, (a + b)$
$c, +, (a + b), *$
$c, +, (a + b), *, a$
$c, +, ((a + b) * a)$
$(c + ((a + b) * a))$

Aufgabe 1.16
Ein einfacher, vollständig geklammerter, arithmetischer Ausdruck ist wie in Aufgabe 1.15 definiert. Ein solcher Ausdruck heißt auch arithmetischer Ausdruck in Infixnotation. Der äquivalente arithmetische Ausdruck in Postfixnotation ist analog wie folgt definiert.

(1) Jede Variable a, b, $c, \ldots$ ist ein Ausdruck in Postfixnotation.

(2) Sind $(\alpha + \beta)$, $(\alpha - \beta)$, $(\alpha * \beta)$, (α / β) Ausdrücke in Infixnotation, so sind $\alpha\beta+$, $\alpha\beta-$, $\alpha\beta*$, $\alpha\beta/$ die äquivalenten arithmetischen Ausdrücke in Postfixnotation.

(3) Sonst nichts.

Beispiel: Der zu $(((a + b) * a) + c)$ äquivalente arithmetische Ausdruck in Postfixnotation ist $ab + a * c+$.

a) Geben Sie ein Verfahren an, das einen Ausdruck in Infixnotation in den äquivalenten arithmetischen Ausdruck in Postfixnotation mit Hilfe zweier Stapel umwandelt (einen Stapel für den arithmetischen Ausdruck, den zweiten als Hilfsstapel für Operationen). Das Verfahren soll nur auf die für Stapel üblichen Operationen, aber nicht auf eine konkrete Implementation Bezug nehmen.

b) Geben Sie eine mögliche Implementation des Verfahrens in Pascal an.

c) Geben Sie ein Verfahren an, das mit Hilfe eines Stapels den Wert eines in Postfixnotation gegebenen Ausdrucks errechnet.

Aufgabe 1.17
Geben Sie ein Verfahren an, das mit Hilfe eines Stapels eine Folge von Buchstaben einliest und in umgekehrter Reihenfolge wieder ausgibt. Die Länge der Folge ist unbekannt, das Ende der Eingabe durch einen Punkt markiert. Das Verfahren soll nur auf die für Stapel üblichen Operationen, aber nicht auf eine konkrete Implementation Bezug nehmen.

Aufgabe 1.18
In einem Sackbahnhof mit drei Gleisen befinden sich in den Gleisen S_1 und S_2 zwei Züge mit Waggons für Zielbahnhof *A* bzw. *B*. Gleis S_3 sei leer (vgl. Abbildung).

Betrachten Sie S_1, S_2, S_3 als Stapel und erstellen Sie ein Programmstück in Pseudo-Pascal unter Verwendung der unten angegebenen Funktionen bzw. Prozedur, das zwei beliebige aus Waggons für *A* und *B* bestehende Züge so umordnet, daß anschließend in S_1 alle Waggons für *A* und in S_2 alle Waggons für *B* stehen.

```
procedure push(var S: Stapel; X: Waggon);
   {push stellt X auf S ab}
function pop(var S: Stapel) : Waggon;
   {pop liefert vordersten Waggon von S und entfernt ihn von S,
    wenn S nicht leer ist; Fehler sonst};
function top(S: Stapel) : Zielbahnhof;
   {top liefert den Zielbahnhof des 1. Waggons in S, ohne ihn
    zu entfernen};
function leer(S: Stapel) : boolean;
   {leer liefert true, wenn S leer; false sonst}.
```