WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE

it
informatik

Gustav Pomberger
Heinz Dobler

Algorithmen und Datenstrukturen

Eine systematische Einführung in die Programmierung

PEARSON
Studium

Grundkonzepte zur Modellierung von Datenobjekten

3

ÜBERBLICK

Motivation

» Basierend auf den im ersten Kapitel eingeführten Konzepten für einfache Datenobjekte, die wir als atomare oder elementare Datenobjekte bezeichnen, wird in diesem Kapitel erörtert, dass man mit diesen elementaren Datenobjekten alleine in der Praxis nicht auskommt. Für die algorithmische Lösung komplexerer Aufgabenstellungen sind sowohl Sammlungen von Datenobjekten gleichen Typs als auch solche unterschiedlichen Typs erforderlich, dazu werden die so genannten Felder (arrays) und Verbunde (compounds, structures oder records) eingeführt.

Den bis hierher diskutierten Datentypen (inklusive Feldern und Verbunden) ist gemeinsam, dass Variablen dieser Typen während ihrer Lebensdauer zwar ihren Wert, nicht aber ihre Struktur ändern können. Für die Lösung vieler Aufgabenstellungen benötigt man jedoch Datenobjekte, die nicht nur ihren Wert, sondern auch ihre Größe und u. U. sogar ihre Struktur ändern, die also insbesondere dynamisch wachsen und schrumpfen können – davon rührt auch die häufig verwendete Bezeichnung dynamische Datenstrukturen (dynamic data structures). Solche Datenobjekte sind Sammlungen mehrerer „kleinerer" Datenobjekte, die untereinander in Beziehung stehen, also vernetzt sind, weshalb wir für sie auch den Begriff vernetzte Datenstrukturen (linked data structures) verwenden. In diesem Kapitel führen wir die zur Realisierung solcher flexiblen Datenobjekte erforderlichen Konzepte und Konstrukte ein und erörtern die Besonderheiten von und den Umgang mit wichtigen Repräsentanten solcher Datenstrukturen, insbesondere werden verkettete Listen, Bäume und binäre (Such-)Bäume ausführlich beschrieben.

Das wichtigste Mittel zur Meisterung der Komplexität ist die Abstraktion. Das gilt sowohl für den Entwurf von Algorithmensystemen (in Kapitel 2 behandelt), als auch für die Modellierung komplexer Datenstrukturen. Deshalb führen wir in diesem Kapitel auch Konzepte zur Konstruktion und Verwendung abstrakter Datenstrukturen (abstract data structures), auch Datenkapseln (data capsules) genannt, und abstrakter Datentypen (abstract data types) ein und widmen diesen wegen ihrer Bedeutung jeweils einen eigenen Abschnitt. «

Ziele

Nach dem Lesen dieses Kapitels haben Sie Folgendes erreicht:

- Sie wissen, dass für die algorithmische Lösung komplexer Aufgaben analog zur Gestaltung der Ablaufstrukturen von Algorithmen auch entsprechende Konzepte und Konstrukte zur Modellierung der in den Algorithmen benötigten Datenobjekte erforderlich sind.
- Sie wissen, dass wir zwischen elementaren und strukturierten Datenobjekten unterscheiden, und kennen die Unterschiede zwischen diesen beiden Kategorien von Datenobjekten und den Datentypen, die zu ihrer Realisierung benötigt werden.
- Sie wissen, dass wir zudem zwischen Datenobjekten, bei denen im Zuge der Algorithmenausführung nur ihr Wert verändert werden kann (sie werden hier in elementare und strukturierte Datenobjekte unterteilt) und solchen, bei denen sich neben ihrem Wert auch ihre Größe und/oder ihre Struktur ändern kann (vernetzte Datenobjekte), unterscheiden.
- Sie wissen, was man unter Feld, Verbund, verkettete Liste, Baum, Binärbaum und binärer Suchbaum versteht, kennen Konzepte und Konstrukte zu deren Realisierung, wissen wozu man solche Datenobjekte benötigt und kennen wichtige Standardoperationen zu ihrer Erzeugung und Manipulation.
- Sie wissen, was man unter einer abstrakten Datenstruktur, einer Datenkapsel und einem abstrakten Datentyp versteht, wozu man diese benötigt und wie man sie realisieren kann.

3.1 Atomare Datenobjekte und -typen

In Kapitel 1 haben wir (neben Anderem) grundsätzliche Eigenschaften von Algorithmen erörtert und darauf aufbauend elementare Konzepte und Konstrukte zur Gestaltung derselben erläutert. In diesem Zusammenhang haben wir auch bereits einige elementare Konstrukte zur Definition von Datenobjekten eingeführt. Wir haben dabei zunächst unterschieden zwischen

1. *unveränderbaren* Datenobjekten in Form von Literalen (z.B. 17) oder Konstanten (z.B. *pi*) und
2. *veränderbaren* Datenobjekten, also Variablen (z.B. *x*), die während der Ausführung des Algorithmus, dessen Bestandteil sie sind – vor allem durch die Zuweisungsanweisung –, unterschiedliche Werte aus einer bestimmten Wertemenge (die durch ihren im Zuge der Deklaration festgelegten Datentyp bestimmt ist) annehmen können.

Eine weitere, von dieser Unterscheidung unabhängige, also zu ihr orthogonale Unterscheidungsmöglichkeit ergibt sich aufgrund der den Datenobjekten zugeordneten Datentypen, die neben einer Wertemenge auch die Menge der möglichen Operationen definieren.

In dieser Hinsicht haben wir in Abschnitt 1.3.1 bereits Datenobjekte eingeführt, die ganze Zahlen (Datentyp *integer*), reelle Zahlen (Datentyp *real*), Wahrheitswerte (Datentyp *boolean*) und Zeichen (Datentyp *char*) repräsentieren. Diesen Datenobjekten ist gemeinsam, dass sie in unserem Sinne als logisch ganze, nicht mehr weiter zerlegbare Artefakte aufgefasst werden und in diesem Sinne als atomar angesehen und deshalb auch als *atomare*, *elementare*, *einfache* oder *unstrukturierte* Datenobjekte (bzw. Datentypen) bezeichnet werden, wobei sich keine dieser Bezeichnungen gegenüber den anderen klar durchgesetzt hat.

Wir haben zwar auch schon Texte bzw. Zeichenketten (Datentyp *string*) kurz behandelt, diese zählen wir aber nicht mehr zu den elementaren Datenobjekten. Wir widmen uns den Zeichenketten weiter hinten bei der Behandlung von Feldern noch einmal ausführlicher.

Analog zur Gestaltung der Ablaufstruktur von Algorithmen benötigt man bei der algorithmischen Lösung komplexer Aufgaben auch Konzepte und Konstrukte zur Gestaltung (wir sagen dazu auch *Modellierung*) der zu manipulierenden Datenobjekte. Neben den elementaren Datenobjekten sind dafür vor allem die so genannten *strukturierten* und *dynamischen* oder *vernetzten* Datenobjekte notwendig. In den folgenden Abschnitten behandeln wir Konzepte und Konstrukte für solche Datenobjekte und -typen mit dem Ziel, möglichst adäquate Mittel für eine realitätskonforme Modellierung der in Algorithmen benötigten Daten bereitzustellen.

3.2 Strukturierte Datenobjekte und -typen

In vielen Algorithmen ist es sinnvoll, wenn nicht gar notwendig, eine Menge einzelner Datenobjekte in einer Variablen zusammenzufassen, weil sie z.B. logisch zusammen gehören und daher eine logische (wenn auch strukturierte) Einheit bilden. Bei derartigen Datenobjekten unterscheiden wir zwischen solchen, deren einzelne Elemente alle denselben Datentyp haben – wir nennen diese ***Felder*** (*arrays*) – und solchen, deren Elemente unterschiedliche Datentypen haben können – wir nennen diese ***Verbunde*** (*compounds*).

3.2.1 Felder

Bevor wir unterschiedliche Ausprägungen von Feldern im Detail behandeln, wollen wir eine Definition für dieses Konzept geben.

Definition: *Feld*

Ein *Feld* (*array*) ist ein Datenobjekt, das eine Sammlung einer fixen Anzahl von Elementen repräsentiert, die alle denselben Datentyp haben und auf die über einen *Index* zugegriffen werden kann.

Felder verwenden wir in verschiedenen Ausprägungen, sowohl als *eindimensionale Felder*, wenn wir z.B. Vektoren (im Sinne der Mathematik) benötigen, als auch als *mehrdimensionale Felder*, wenn wir z.B. Matrizen (wieder im Sinne der Mathematik) benötigen.

Eindimensionale Felder

Ein *eindimensionales Feld* (*one-dimensional array*) *a* fasst eine fixe Anzahl von Elementen desselben Datentyps, des so genannten *Elementdatentyps* (*element data type*), so zu einer Einheit – in der diese Elemente durchnummeriert sind – zusammen, dass über einen *Index i* auch auf die einzelnen Elemente zugegriffen werden kann. Dazu wird der *Indexoperator* [] auf das Feld *a* mit dem Index *i* angewandt: *a*[*i*] liefert das Element mit dem Index *i* im Feld *a*. Der Wertebereich, aus dem dieser Index stammen muss, wird als *Indexbereich* (*index range*) bezeichnet. Dabei handelt es sich um einen Unterbereich der ganzen Zahlen, der durch den kleinsten (ersten, *first*) und den größten (letzten, *last*) Indexwert angegeben wird.[1]

In der von uns verwendeten Notation zur Beschreibung von Algorithmen sieht die Deklaration einer eindimensionalen Feldvariable *a* folgendermaßen aus:

```
var a = array [first:last] of ElementType
```

Dabei müssen *first* und *last* Literale oder Konstanten sein und es muss $first \leq last$ gelten. Das Feldobjekt *a* hat dann $last - first + 1$ Elemente. Die eckigen Klammern, die den Indexbereich in der Deklaration einschließen, dürfen nicht mit dem Indexoperator in Ausdrücken verwechselt werden, sie dienen in der Deklaration nur der Abgrenzung des Indexbereichs von den restlichen Bestandteilen.

Beispiel 3.1 zeigt eine einfache Deklaration einer Feldvariablen.

Beispiel 3.1 **Deklaration einer Feldvariablen**

▶Abbildung 3.1 zeigt, wie eine Feldvariable *a* (also ein Feldobjekt) mit zehn Elementen vom Datentyp *integer* und einem Indexbereich von 1 bis 10 deklariert wird.

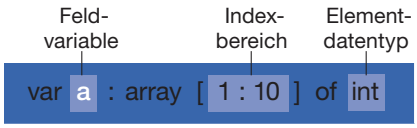

Abbildung 3.1: Deklaration einer Feldvariablen

1 Es gibt zwar Programmiersprachen, Pascal etwa, bei denen neben ganzen Zahlen z.B. auch Zeichen zum Indizieren von Feldern erlaubt sind, da dies in den meisten Sprachen aber nicht möglich ist, wollen wir nur ganze Zahlen als Index zulassen. Das ist ohne Beschränkung der Allgemeinheit möglich, denn jeder Indexdatentyp muss auch in Sprachen wie Pascal auf die ganzen Zahlen abgebildet werden können.

Es gibt Programmiersprachen (Beispiele s. u.), bei denen anstelle eines Indexbereichs die Anzahl der Elemente eines Felds anzugeben ist. Wir wollen das in unserem Pseudocode durch eine Deklaration nach folgendem Muster

```
var a = array [n] of ElementType
```

nachbilden, wobei n ein Literal oder eine Konstante (jeweils ganzzahlig) mit $n > 0$ sein muss. Je nach Programmiersprache wird dann dem ersten Element des Felds der Index 0 (z.B. in C/C++, Java und C#) oder der Index 1 (z.B. in BASIC, MatLab und Smalltalk) zugeordnet. Aufgrund dieser Eigenschaften werden Felder in solchen Programmiersprachen entweder als *null-basierend* (*zero-based*) oder als *eins-basierend* (*one-based*) bezeichnet.

In ▶Abbildung 3.2 ist dargestellt, wie man sich die Repräsentation der Feldvariablen a aus Abbildung 3.1 im Hauptspeicher eines Computers vorstellen kann: Wie jede andere Variable hat sie eine Startadresse und ab dieser Adresse liegen die einzelnen Elemente des Felds unmittelbar hintereinander im Speicher. Mit dem Indexoperator und einem Index i (wobei in diesem Fall $1 \leq i \leq 10$ gelten muss) kann mittels des Ausdrucks $a[i]$ ein beliebiges, in diesem Fall – weil eins-basierend – das i-te Element des Felds adressiert werden.

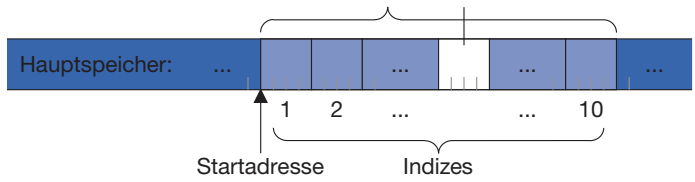

Abbildung 3.2: Anordnung der Elemente einer Feldvariablen aus Beispiel 3.1 im Speicher

Hinweis: Die in Abbildung 3.2 gezeigte, sequenzielle und lückenlose Anordnung der einzelnen Feldelemente im Speicher ermöglicht z.B. einem Compiler die einfache Abbildung der Indexoperation $a[i]$ auf maschinennahe Konzepte (Speicheradressen), denn aus der Startadresse $AddrOf(\downarrow a)$ des Felds, einem Index i, der Untergrenze des Indexbereichs *first* und dem Speicherbedarf eines Elements in Byte $elementSize = SizeOf(\downarrow ElementType)$ kann die Adresse von $a[i]$ durch Anwendung der so genannten *Speicherabbildungsfunktion* (*memory addressing function*) berechnet werden:

$$AddrOf(\downarrow a[i]) = AddrOf(\downarrow a) + (i - first) \cdot elementSize$$

Diese Speicherabbildungsfunktion ermöglicht den effizienten Zugriff – nämlich in konstanter Zeit, also unabhängig vom Wert des Index – auf jedes Element des Felds.

Werden mehrere Feldobjekte mit den gleichen Eigenschaften, also mit gleichem Elementdatentyp und identischem Indexbereich, benötigt, ist es ratsam, dafür einen eigenen Datentyp vorzusehen. Beispiel 3.2 zeigt, wie wir – hier in Pseudocode, aber bei vielen Programmiersprachen ist das in ähnlicher Weise möglich – mit dem *type*-Konstrukt einen neuen, so genannten *benutzerdefinierten Datentyp* (*user-defined data type*) einführen können, der dann genauso verwendet werden kann wie die elementaren (Standard-)Datentypen, mit denen wir bisher gearbeitet haben (z.B. *integer* und *real*).

Beispiel 3.2

Deklaration und Verwendung eines benutzerdefinierten Felddatentyps

Wenn für die Lösung einer Aufgabe mit Koordinaten aus dem dreidimensionalen Raum gerechnet werden muss, ist es sinnvoll, dafür beispielsweise einen Datentyp *Position3D* gemäß folgender Typdeklaration vorzusehen:

```
type Position3D = array [1:3] of real
```

Dieser benutzerdefinierte Felddatentyp kann nun zur Deklaration von Variablen und in Formalparameterlisten verwendet werden, z.B.:

```
var p1, p2: Position3D
Distance(↓a: Position3D  ↓b: Position3D): real
```

Benutzerdefinierte Datentypen wie *Position3D* in Beispiel 3.2 sind selbstverständlich nicht auf die Verwendung für Felder beschränkt, sondern können auch zur Definition anderer Datenobjekte herangezogen werden.

Felder haben eine fixe, durch ihre Deklaration definierte Anzahl von Feldelementen. Beispielsweise wird durch die Deklaration des Datentyps *Position3D* in Beispiel 3.2 festgelegt, dass alle Datenobjekte dieses Typs Felder sind, die drei Elemente vom Typ *real* enthalten.

Oft tritt die Situation ein, dass in einem konkreten Anwendungsfall nur ein Teil der (durch die Deklaration festgelegten) Elemente eines Felds tatsächlich benötigt wird. In solchen Fällen muss man Vorsorge treffen, dass der aktuelle „Füllstand" des betreffenden Feldobjekts jederzeit festgestellt werden kann.

Beispiel 3.3 zeigt nicht nur, wie man solche *teilweise gefüllten Felder* (*partially filled arrays*) deklariert, sondern auch wie man mit ihnen und mit Feldern generell arbeitet.

Beispiel 3.3

Deklaration und Verwendung eines teilweise gefüllten Felds

Für die Speicherung aller Noten (*marks* oder *grades*), die ein Student während seines Studiums erzielt hat, kann man (um daraus z.B. das Studienergebnis entsprechend der Aufgabenstellung in Abschnitt 2.5 zu berechnen) ein Feld *marks* heranziehen, in dem höchstens *max* Noten gespeichert werden können. Die aktuelle Anzahl der eingetragenen Noten muss dann in einer zusätzlichen Variablen *n* verwaltet werden, wobei zu jedem Zeitpunkt $0 \leq n \leq max$ gelten muss. Das führt zu folgenden Deklarationen:

```
const max = ...  -- maximum number of marks
var marks: array [1:max] of int
    n: int  -- mark counter, marks[1:n] filled
```

Das Einlesen der Noten von einem Datenstrom, der von einem Eingabegerät (*input device*) zur Verfügung gestellt wird, kann dann wie folgt formuliert werden (wobei wir davon ausgehen, dass nicht mehr als *max* Noten im Datenstrom enthalten sind):

```
n := 0  -- no marks up to now
while not end of file do
  n := n + 1
  Read(↑marks[n])  -- assume reading a value in the range from 1 to 5
end -- while
```

Nach der Ausführung des obigen Codestücks ergibt sich der in ▶Abbildung 3.3 dargestellte Zustand eines teilweise gefüllten Felds: der dunklere Bereich des Felds enthält bereits Noten, der hellere Bereich ist (noch) frei.

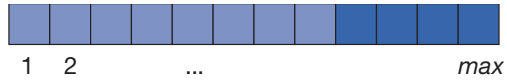

Abbildung 3.3: Teilweise gefülltes Notenfeld *marks*

Zur Berechnung des Notendurchschnitts (*average* vom Datentyp *real*) kann dann das Feld *marks* und der aktuelle Füllstand *n* wie folgt herangezogen werden (wobei wir davon ausgehen, dass $n > 0$ gilt):

```
sum := 0
for i := 1 to n do
  sum := sum + marks[n]
end -- for
average := Real(↓sum) / Real(↓n)
```

Unter der Annahme, dass es einen Algorithmus *Sort* gibt, der die Noten im Feld *marks* aufsteigend sortiert (siehe Kapitel 7), können wir beispielsweise auch den *Median* (*median*), also den Wert in der Mitte des sortierten Bereichs, auf einfache Weise ermitteln:

```
Sort(↕marks  ↓1  ↓n)
if n mod 2 = 1 then  -- odd number of marks:
                     -- there is an element in the middle
  median := marks[(n + 1) / 2]
else  -- even number of marks:
      --   use average of elements left and right of the middle
  median := ( marks[n / 2] + marks[(n / 2) + 1] ) / 2
end -- if
```

Wollen wir auch noch die *Häufigkeiten* (*frequencies*) der einzelnen Noten berechnen, können wir dies durch Verwendung eines zweiten Felds *freqs*, das Platz für die Speicherung der Häufigkeiten der fünf Noten bietet und die jeweilige Note als Index in *freqs* benutzt (siehe ▶Abbildung 3.4, die zeigt, wie sich beispielsweise die Häufigkeit der Note 1 ergibt):

```
  var freqs: array [1:5] of int
begin
  -- initialize frequencies to zero
  for i := 1 to 5 do
    freqs[i] := 0
  end -- for
  -- count each mark in frequencies array
  for i := 1 to n do
    mark := marks[i]
    freqs[mark] := freqs[mark] + 1
  end -- for
```

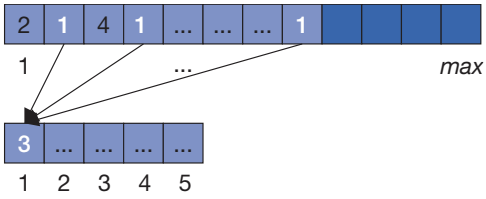

Abbildung 3.4: Noten und ihre Häufigkeiten

Zeichenketten

Bei der Behandlung der Datentypen in Abschnitt 1.3.2 haben wir für Datenobjekte, die Texte oder Zeichenketten repräsentieren, zwar schon den Datentyp *string* eingeführt, die Details seiner Realisierung aber noch offen gelassen. Dies wollen wir im Folgenden nachholen und damit gleichzeitig weitere Möglichkeiten des Einsatzes eindimensionaler Felder aufzeigen.

Zeichenketten sind Folgen von einzelnen Zeichen. Deshalb bietet es sich an, Zeichenketten als Felder mit dem Elementtyp Zeichen (*char*) zu realisieren. Dazu benötigt man die maximal erlaubte Anzahl der Zeichen, für die das Feld Platz bieten soll, und kann dann den Datentyp *string* folgendermaßen definieren:

```
const max = …
type string = array [1:max] of char
```

Nachdem man auch unterschiedlich lange Zeichenketten darstellen möchte, muss ihre aktuelle Länge separat verwaltet werden, was eine zusätzliche Variable vom Datentyp *integer* erforderlich macht. Für die Repräsentation einer Zeichenkette wären daher – analog zu den teilweise gefüllten Feldern in Beispiel 3.3 – zwei Variablen erforderlich, z.B.:

```
var s: string
    sLen: int  -- holds current length of s, 0 ≤ sLen ≤ max
```

Das ist umständlich und fehleranfällig, da bei Operationen auf Zeichenketten immer beide Variablen (z.B. *s* und *sLen*) betroffen wären und konsistent gehalten werden müssten.

Um das zu vermeiden, wurden bei der Entwicklung von Programmiersprachen spezielle Zeichenketten-Repräsentationen entwickelt. Zwei Arten davon, nämlich Pascal- und C-Zeichenketten, werden im Folgenden erörtert, um dem Leser einen Eindruck über typische Repräsentationsformen zu vermitteln.

Pascal-Zeichenketten *Pascal-Zeichenketten* werden in der Programmiersprache Pascal verwendet (vgl. [Wirth 1971a]) und sind durch folgende Typdeklaration definiert:

```
type PascalString = array [0:255] of char
```

Dabei ist festgelegt, dass das Element mit dem Index 0 nicht Teil der Zeichenkette ist, sondern ihre aktuelle Länge repräsentiert. Es wird daher auch als *Längenbyte* bezeichnet. Durch das Längenbyte können nur Werte zwischen 0 und 255 repräsentiert werden, die maximale Länge von Pascal-Zeichenketten beträgt daher 255 Zeichen.

Beispiel 3.4 zeigt die Deklaration und die Verwendung einer Pascal-Zeichenketten-Variablen sowie ihre Repräsentation im Hauptspeicher eines Computers.

Beispiel 3.4 **Pascal-Zeichenkette und ihre Repräsentation im Speicher**

Das folgende Pseudocode-Fragment deklariert eine Pascal-Zeichenketten-Variable *ps* und weist ihr einen Wert zu:

```
  var ps: PascalString
begin
  ps := "Hello"
```

▶Abbildung 3.5 zeigt die Repräsentation des Zeichenkettenobjekts *ps* im Hauptspeicher eines Computers, wobei die Fragezeichen beliebige Zeichen bedeuten, die nicht mehr zur aktuellen Zeichenkette gehören.

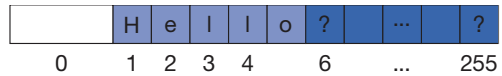

Abbildung 3.5: Repräsentation der Zeichenkette "Hello" als Pascal-Zeichenkette im Speicher

Der Vorteil von Pascal-Zeichenketten liegt darin, dass die aktuelle Länge einfach und effizient mit der Funktion

```
Length(↓ps: PascalString): int
begin
  return Int(↓ps[0])
end Length
```

zu ermitteln ist. Der wesentliche Nachteil von Pascal-Zeichenketten liegt (wegen des Längenbytes) in der Beschränkung auf die maximale Länge von 255 Zeichen; ein weiterer in der Verschwendung von Speicherplatz für die Repräsentation kurzer Zeichenketten.

C-Zeichenketten *C-Zeichenketten* werden in der Programmiersprache C verwendet (vgl. [Kernighan u. Ritchie 1988]) und sind durch folgende Deklarationen definiert:

```
const max = …
type CString = array [0:max] of char
```

Dabei ist festgelegt, dass eine C-Zeichenkette durch ein spezielles Zeichen, das sogenannte *Terminierungszeichen* mit dem Code 0, also *Char*(↓0), abgeschlossen und in C mit '\0' notiert wird. Deshalb werden C-Zeichenketten im Engl. auch als *null-terminated strings* bezeichnet. Das Terminierungszeichen kann deshalb innerhalb der Zeichenkette nicht vorkommen.

Beispiel 3.5 zeigt die Deklaration und die Verwendung einer C-Zeichenketten-Variablen sowie ihre Repräsentation im Hauptspeicher eines Computers.

Beispiel 3.5

C-Zeichenkette und ihre Repräsentation im Speicher

Das folgende Pseudocode-Fragment deklariert eine C-Zeichenketten-Variable *cs* und weist ihr einen Wert zu:

```
  var cs: CString
begin
  cs := "Hello"
```

▶Abbildung 3.6 zeigt die Repräsentation des Zeichenkettenobjekts *cs* im Hauptspeicher eines Computers, wobei die Fragezeichen beliebige Zeichen bedeuten, die nicht mehr zur aktuellen Zeichenkette gehören.

Abbildung 3.6: Repräsentation der Zeichenkette "Hello" als C-Zeichenkette im Speicher

Der Vorteil von C-Zeichenketten liegt darin, dass ihre Länge nicht beschränkt ist, denn die maximale Länge (*max* in der obigen Deklaration) kann beliebig groß gewählt werden. Der wesentliche Nachteil ist, dass die Ermittlung der Länge aufwändig ist, denn dafür muss das Terminierungszeichen (beispielsweise auf folgende Weise) gesucht werden:

```
Length(↓cs: CString): int
  var i: int
begin
  i := 0
  while cs[i] ≠ Char(↓0) do
    i := i + 1
  end -- while
  return i
end Length
```

Hinweis: In objektorientierten Programmiersprachen (z.B. in C++, Java und C#) werden Zeichenketten durch „echte“ Objekte[2] spezieller Zeichenketten-Klassen (z.B. der Klasse *java.lang.String* in Java) dargestellt. (Kapitel 13 behandelt die Grundlagen der objektorientierten Programmierung.) Die interne Repräsentation dieser Zeichenkettenobjekte bleibt dem Benutzer zwar verborgen, es werden aber auch Zeichenfelder verwendet, nur ist das für den Benutzer nicht mehr relevant. Diese Zeichenkettenobjekte beseitigen durch geschickte Realisierung beide Nachteile der oben behandelten Zeichenketten-Repräsentationen: die Zeichenketten können somit beliebig lang werden (wie bei C-Zeichenketten) *und* ihre Länge kann effizient ermittelt werden (wie bei Pascal-Zeichenketten).

Zwei- und mehrdimensionale Felder

Die Nützlichkeit bzw. Notwendigkeit zwei- und mehrdimensionaler Felder (*two-* und *multi-dimensional arrays*) braucht nicht weiter argumentiert werden; bei vielen Aufgabenstellungen finden wir z.B. Matrizenobjekte, etwa für eine Entscheidungstabelle oder ein Schachbrett.

Beispiel 3.6 zeigt die Verwendung eines zweidimensionalen Felds.

Beispiel 3.6 **Zweidimensionales Feld (Matrix) und Repräsentation im Speicher**

Für die Lösung einer Mathematikaufgabe benötigt man eine Matrix mit vier Zeilen und drei Spalten, eine 4x3-Matrix. Den zweidimensionalen Felddatentyp *Matrix* und eine Variable *m* von diesem Typ können wir wie folgt festlegen:

```
type Matrix = array [1:4, 1:3] of real
var m: Matrix
```

▶Abbildung 3.7 (a) zeigt eine schematische Darstellung der Matrix *m*, wie man sie z.B. in Mathematikbüchern findet, während in (b) illustriert wird, wie man sich die Repräsentation der Variablen *m* im Hauptspeicher eines Computers vorstellen kann. Nachdem der Hauptspeicher konzeptuell eine lineare, also eindimensionale, Anordnung von Bytes ist, muss die Matrix, also das zweidimensionale Feld, linearisiert (in eine Dimension transformiert) werden. Da es in den meisten Programmiersprachen so gehandhabt wird, gehen wir davon aus, dass die Zeilen (*rows*) der Matrix nacheinander im Speicher angeordnet werden, die Matrix wird also *zeilenweise* im Speicher abgelegt (*row major order*).

2 Damit sind Objekte im Sinne der objektorientierten Programmierung (OOP) gemeint, deren Datentyp eine Klasse ist und die über Datenkomponenten und Methoden verfügen (siehe dazu Kapitel 13).

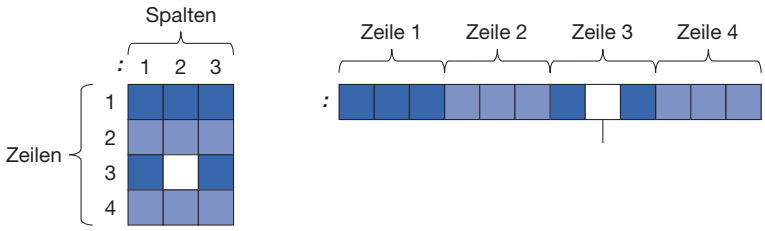

Abbildung 3.7: Anordnung der Elemente eines zweidimensionalen Felds (a) im Speicher (b)

Wenn eine Bearbeitung aller Elemente einer Matrix erforderlich ist, erfolgt dies typischerweise mittels zweier ineinander geschachtelter Zählschleifen. Das folgende Codestück zeigt, wie die Matrix *m* mit dem Wert 0.0 initialisiert werden könnte:

```
for i := 1 to 4 do     -- for all rows
  for j := 1 to 3 do   -- for all columns
    m[i, j] := 0.0
  end -- for
end -- for
```

Da es für den Elementtyp keinerlei Einschränkungen gibt, kann ein zweidimensionales Feld – was Abbildung 3.7 (b) nahe legt – auch als eindimensionales Feld interpretiert werden, dessen Elemente eindimensionale Felder sind, also ein „Feld von Feldern".

Hinweis: Wie bei eindimensionalen Feldern muss auch bei zweidimensionalen für den Zugriff auf ein Element durch Anwendung der *Speicherabbildungsfunktion* (*memory addressing function*) die Speicheradresse dieses Elements berechnet werden.

Für eine Matrix *m* mit der Deklaration

```
var m: array [first1:last1, first2:last2] of ElementType
```

kann aus der Startadresse *AddrOf*(↓*m*), den beiden Indizes *i* und *j*, den beiden Untergrenzen der Indexbereiche *first1* und *first2*, dem Speicherbedarf eines Elements in Byte *elementSize* = *SizeOf*(↓*ElementType*) sowie dem Speicherbedarf einer ganzen Zeile *rowSize* = (*last1* – *first1* + 1) · *elementSize* die Adresse von *m*[*i, j*] wie folgt berechnet werden:

$$AddrOf(\downarrow m[i, j]) = AddrOf(\downarrow m) + (i - first1) \cdot rowSize + (j - first2) \cdot elementSize$$

Diese Speicherabbildungsfunktion gewährleistet – wie bei eindimensionalen Feldern – einen effizienten Zugriff (in konstanter Zeit) auf alle Elemente der Matrix, unabhängig von den Werten der beiden Indizes.

Die Anzahl der Dimensionen eines Feldobjekts ist in der Regel nicht begrenzt, so dass neben den ein- und zweidimensionalen Feldobjekten, die wir bisher betrachtet haben, in den meisten Programmiersprachen auch Feldobjekte mit drei und mehr Dimensionen möglich sind. Da sich weder die Deklaration noch die Verwendung solcher mehrdimensionalen Felder grundsätzlich von den ein- und zweidimensionalen unterscheidet, brauchen wir darauf nicht näher einzugehen.

3.2.2 Verbunde

In der Praxis finden wir häufig Aufgabenstellungen, bei deren algorithmischer Lösung wir, wenn der Algorithmus möglichst realitätsnah gestaltet werden soll, auf Datenobjekte stoßen, die mehrere Elemente unterschiedlichen Datentyps zu einem logischen Ganzen zusammenfassen, wie in Beispiel 3.7 gezeigt. Deshalb ist es naheliegend, auch dafür ein Konzept und ein Konstrukt zur Umsetzung vorzusehen.

Beispiel 3.7 **Getrennte Speicherung von Studierendendaten**

Für ein Softwaresystem zur Verwaltung der Daten von Studierenden müssen deren Daten wie z.B. die Identifikationsnummer (*id*), der Name (*name*) und – vorerst – das Alter (*age*) verfügbar sein. Dazu könnte man (mit den bisher eingeführten Möglichkeiten) z.B. folgende drei Felder vorsehen:

```
const max = …
var studentIds:   array [1:max] of int
    studentNames: array [1:max] of string
    studentAges:  array [1:max] of int
```

Durch die in Beispiel 3.7 gezeigte Lösungsvariante werden zusammengehörende Eigenschaften auf mehrere Datenobjekte verteilt. Die Zusammengehörigkeit geht dabei verloren und ist aus der dazugehörigen Algorithmenbeschreibung nicht mehr so ohne Weiteres erkennbar.

Um diesen Nachteil zu vermeiden, führen wir das Konzept des Verbunds ein, für das wir eine Definition angeben, bevor wir uns mit den Details und den dafür notwendigen Konstrukten beschäftigen.

Definition: *Verbund*

Ein *Verbund* (*compound*, *record* oder *structure*) ist ein Datenobjekt, das eine fixe Anzahl von Komponenten, deren Datentyp beliebig ist, so zusammenfasst, dass man auf das Datenobjekt als Ganzes und über die entsprechenden Bezeichner auch auf die einzelnen Komponenten zugreifen kann.

Verbunde verwenden wir immer dann, wenn wir für eine algorithmische Lösung ein Objekt aus der Problemdomäne modellieren müssen, das mehrere für den Algorithmus relevante Eigenschaften aufweist, die deshalb zusammengefasst werden sollen.

Ein Verbund (*compound*) *c* fasst benannte Komponenten so zu einer Einheit zusammen, dass auf ihn sowohl als Ganzes, als auch auf die einzelnen Komponenten separat zugegriffen werden kann. Für den Zugriff auf eine Verbundkomponente wird der *Komponentenselektionsoperator* . (Punkt) auf den Verbund *c* mit einem Komponentennamen *n* angewandt: *c.n* bezeichnet die Komponente mit dem Namen *n* des Verbunds *c*.

In der von uns verwendeten Notation zur Beschreibung von Algorithmen sieht die Deklaration einer Verbundvariable folgendermaßen aus:

```
var c = compound
      a: AnyType
      b: SameOrAnyOtherType
      …
      z: SameOrYetAnotherType
    end -- compound
```

▶Abbildung 3.8 zeigt, wie man sich die Repräsentation der Komponenten des oben deklarierten Verbunds *c* im Hauptspeicher eines Computers vorstellen kann. Wie die Elemente eines Feldobjekts liegen die Komponenten eines Verbundobjekts lückenlos hintereinander im Speicher, können aber – je nach Datentyp – unterschiedlichen Speicherbedarf aufweisen.

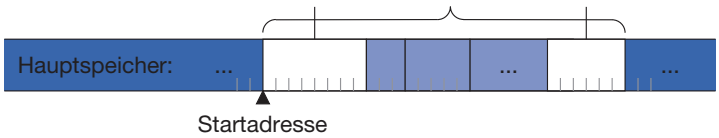

Abbildung 3.8: Anordnung der Komponenten eines Verbunds im Speicher

Hinweis: Wie ein Feld hat auch ein Verbund, wie in Abbildung 3.8 gezeigt, eine Startadresse. Da die einzelnen Komponenten aber unterschiedliche Datentypen haben und somit unterschiedlichen Speicherplatz beanspruchen können, existiert keine Speicherabbildungsfunktion zur effizienten Berechnung der Adresse der *i*-ten Komponente von Verbunden analog zu jener von Feldern; der Zugriff auf die einzelnen Komponenten ist nur über ihre Namen möglich.

Beispiel 3.8 zeigt eine Anwendung des Verbundkonzepts für die in Beispiel 3.7 noch separiert voneinander gespeicherten Daten von Studierenden.

Beispiel 3.8

Speicherung von Studierendendaten in einem Verbund

Für das in Beispiel 3.7 erwähnte Softwaresystem zur Verwaltung der Daten von Studierenden ist es sinnvoll, einen benutzerdefinierten Verbundtyp *Student* wie folgt einzuführen:

```
type Student = compound
       id: int
       name: string
       age: int
     end -- compound
```

Diesen Datentyp können wir nun verwenden, um Variablen für die einzelnen Studierenden zu deklarieren und ihre Komponenten mit entsprechenden Werten zu füllen:

```
  var s1, s2, … : Student  -- two students
begin
  s1.id := 2006099
  s1.name := "Max Mustermann"
  s1.age := 19
  s2.id := …
```

Da es hinsichtlich des Datentyps einer Komponente eines Verbunds keinerlei Einschränkungen gibt, kann (in Analogie zu Feldern) der Datentyp einer Verbundkomponente selbst wieder ein Verbund sein, so dass „Verbunde von Verbunden" möglich sind.

Beispiel 3.9 zeigt, wie man damit die Daten von Studierenden praxisgerechter speichern kann.

Beispiel 3.9

Verbund in Verbund

Wir können nun die nicht praxisgerechte Definition des in Beispiel 3.8 eingeführten Datentyps *Student* verbessern, denn es war keine gute Idee, wie in dieser Typdeklaration festgelegt, das Alter eines Studierenden zu speichern, da sich dieses permanent ändert. Es ist vielmehr naheliegend, eine Komponente für das Geburtsdatum vorzusehen.

Da Kalenderdaten öfter benötigt werden, ist es sinnvoll, dafür wiederum einen eigenen (Verbund-)Datentyp *Date* einzuführen und diesen bei der Definition des Datentyps *Student* zu verwenden, um statt der Alterskomponente eine Komponente für das Geburtsdatum (*dateOfBirth*) vorzusehen:

```
type Date = compound
       day, month, year: int
     end -- compound
     Student = compound
       …
       dateOfBirth: Date
     end -- compound
var s: Student
```

Die folgenden drei Beispiele für komponentenbezogene Wertzuweisungen zeigen, wie der Komponentenselektionsoperator – analog zum Indizierungsoperator – mehrfach angewendet werden kann, um auf „ineinander geschachtelte“ Komponenten zugreifen zu können:

```
s.dateOfBirth.day   :=    1
s.dateOfBirth.month :=    4
s.dateOfBirth.year  := 1987
```

3.2.3 Gegenüberstellung und Kombination von Feldern und Verbunden

Feld- und Verbundtypen gestatten es, strukturierte Datenobjekte zu bilden. Gemeinsam ist solchen Datenobjekten, dass sie mehrere Elemente/Komponenten so zu einer Einheit zusammenfassen, dass auf diese sowohl als Ganzes als auch auf die einzelnen Bestandteile zugegriffen werden kann.

Tabelle 3.1 stellt die beiden Konzepte einander gegenüber und zeigt wesentliche Unterschiede zwischen Feldern und Verbunden.

Sowohl Feld- als auch Verbundobjekte – jeweils für sich alleine betrachtet – ermöglichen es, bestimmte Sachverhalte in Algorithmen besser zu modellieren, als dies mit elementaren Datenobjekten alleine möglich wäre.

Tabelle 3.1

Unterschiede von Feldern und Verbunden

Unterscheidungsmerkmal	Felder	Verbunde
Datentyp der Bestandteile	*Elemente* müssen gleichen Datentyp haben	*Komponenten* können unterschiedliche Datentypen haben
Zugriff auf die Bestandteile	Mit Index i und Indexoperator: $a[i]$	Mit Name n und Komponentenselektionsoperator: $c.n$

Das Potenzial der Verfügbarkeit strukturierter Datenobjekte entfaltet sich aber erst, wenn Felder und Verbunde miteinander kombiniert werden. Das ist möglich, da es weder bezüglich des Elementtyps eines Felds noch bezüglich der Komponententypen in einem Verbund Einschränkungen gibt. Somit sind Felder von Verbunden und Verbunde mit Feldkomponenten möglich – und das in beliebig tiefer Schachtelung.

Beispiel 3.10 zeigt die Möglichkeit der Schachtelung von Feldern und Verbunden anhand der Speicherung mehrerer Studenten mit ihren Noten.

Beispiel 3.10 **Mehrere Studenten mit ihren Noten**

Für das in Beispiel 3.8 und Beispiel 3.9 erwähnte Softwaresystem zur Verwaltung der Daten von Studierenden ist es notwendig, auch die Noten der Studierenden zu speichern. Dazu kann beispielsweise – wie das Codestück unten zeigt – eine Komponente *marks*, die ein Feldobjekt repräsentiert, vorgesehen werden (vgl. auch Beispiel 3.3). Um eine einfache Verarbeitung der Daten aller Studierenden zu ermöglichen, ist es außerdem sinnvoll, ein Feld *students* mit dem Elementdatentyp *Student* vorzusehen.

```
const maxMarks = …
      maxStudents = …

type Student = compound
       id: int
       name: string
       dateOfBirth: Date
       marks: array [1:maxMarks] of int
       nMarks: int -- number of marks
     end -- compound

var students: array [1:maxStudents] of Student
    nStudents: int -- number of Students
```

Das folgende Codestück zeigt anhand einer Zuweisung, wie man Index- und Komponentenselektionsoperator kombinieren kann, um ausgehend von der Variablen *students* auf alle Bestandteile dieses strukturierten Datenobjekts zuzugreifen, beispielsweise auf die zweite Note des ersten Studenten.

```
students[1].marks[2] := 3
```

▶Abbildung 3.9 zeigt den hierarchischen Aufbau der Variablen *students*. Daraus ist ersichtlich, dass es sich um ein Feld von Verbunden des Datentyps *Student* handelt, wobei in jedem dieser Verbunde auch Felder (z.B. in Form des Datentyps *string*) als Komponenten vorkommen. Auf den untersten Ebenen der Hierarchie kommen nur mehr elementare Datentypen vor (hier z.B. *char* als Elemente von *string* und *int*).

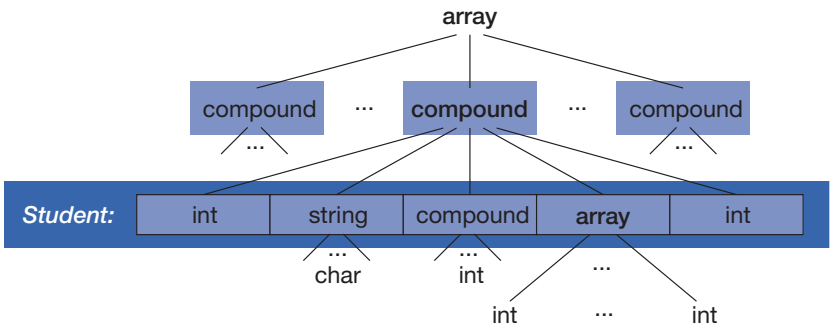

Abbildung 3.9: Hierarchischer Aufbau der Variablen *students*

3.3 Vernetzte oder dynamische Datenobjekte und -typen

Den elementaren und strukturierten Datentypen, die wir bisher behandelt haben, ist gemeinsam, dass Variablen dieser Datentypen während ihrer Lebensdauer ihren Wert, aber nicht ihre Struktur ändern können. Folglich bleibt die Größe des Speicherplatzes, den sie benötigen, konstant.

In der Praxis begegnen wir Aufgabenstellungen, in denen Datenobjekte vorkommen, die charakterisiert sind durch die Veränderung ihres Werts *und* ihrer Struktur, also Datenobjekte, die aus Komponenten bestehen, die miteinander in Beziehung stehen, also vernetzt sind. Wir nennen diese deshalb *vernetzte Datenstrukturen*, um hervorzuheben, dass der Struktur des Datenobjekts besondere Bedeutung zukommt. Solche Datenobjekte können zur Laufzeit eines Algorithmus, also dynamisch, wachsen und schrumpfen und/oder es kann sich die Beziehungsstruktur zwischen ihren Komponenten ändern, weshalb sie auch als *dynamische Datenstrukturen* bezeichnet werden. Das heißt, dass sowohl die Größe als auch die Struktur und der Inhalt der Datenstruktur veränderbar sind. Wichtige Repräsentanten solcher Datenobjekte mit Strukturcharakter sind *verkettete Listen* und *Bäume*, vor allem *Binärbäume*; sie werden später in eigenen Abschnitten ausführlich behandelt.

Bevor wir uns den verschiedenen Ausprägungen vernetzter Datenobjekte bzw. ihrer Datentypen widmen, müssen die dafür notwendigen Konzepte eingeführt werden: *Zeiger*(*datentypen*) und die dynamische *Speicherallokation* sowie *-freigabe*.

3.3.1 Zeiger und Zeigerdatentypen

Wie in Abschnitt 1.3 erläutert, werden alle Variablen, die in einem Algorithmus vorkommen (nach der Transformation des Algorithmus in ein Programm einer bestimmten Programmiersprache und seiner Übersetzung in eine ausführbare Version bei der Ausführung), im Speicher eines Computers repräsentiert: Für jede Variable wird dazu

ab einer bestimmten Startadresse so viel Speicher zur Verfügung gestellt, wie aufgrund ihres Datentyps erforderlich ist. Beispiel 3.11 zeigt das anhand einer Variablen vom Datentyp *integer*.

Beispiel 3.11 **Variable mit Wert und Startadresse**

Gegeben sei die Variable *intVar* vom Datentyp *integer*, welcher gemäß dem folgenden Codestück der Wert 17 zugewiesen wurde:

```
  var intVar: int
begin
  intVar := 17
```

▶Abbildung 3.10 zeigt die Repräsentation von *intVar* im Hauptspeicher eines Computers. Der aktuelle Wert ist durch die Zuweisung entstanden; die Startadresse 2468 ist hypothetisch.

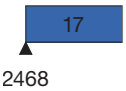

Abbildung 3.10: Variable *intVar* im Speicher, mit Wert und Startadresse

Nun ist eine (Start-)Adresse selbst wieder ein Datum, das in einer Variablen gespeichert werden kann. Aber von welchem Datentyp soll diese Variable sein?

Da Adressen stets positive ganze Zahlen sind, könnten sie in Variablen vom Datentyp *integer* gespeichert werden. Dadurch würde aber ihre besondere Eigenschaft verloren gehen und sie könnten nicht mehr dazu verwendet werden, über die in ihnen gespeicherten Adressen „indirekt“ auf die Werte an den entsprechenden Stellen im Speicher „zuzugreifen“ und diese verändern zu können. Deshalb ist ein neuer Datentyp für solche „Adressvariablen“ zweckmäßig und wir führen dazu das Konzept der *Zeiger* und *Zeigerdatentypen* ein. Bevor wir uns mit den Details dieses Konzepts beschäftigen und die dafür notwendigen Konstrukte einführen, geben wir Definitionen für diese beiden zentralen Begriffe an.

Definition: *Zeiger und Zeigerdatentyp*

Ein *Zeiger* (*pointer*) ist ein Datenobjekt, dessen Datentyp ein Zeigerdatentyp ist.

Ein *Zeigerdatentyp* (*pointer data type*) ist ein benutzerdefinierter Datentyp, der in unserer Algorithmennotation durch das vorangestellte Symbol → als solcher gekennzeichnet wird und bei dessen Deklaration der Datentyp jener Datenobjekte anzugeben ist, auf die durch einen Zeiger dieses Zeigerdatentyps verwiesen werden kann. Dieser „referenzierte“ Datentyp wird als *Basisdatentyp* (*base data type*) bezeichnet.

Wir deklarieren einen Zeigerdatentyp oder kurz *Zeigertyp* (*pointer type*) und eine Zeigervariable dieses Typs gemäß obiger Definition wie folgt:

```
type PointerType = →BaseType
var p: PointerType
```

Der oben deklarierten (Zeiger-)Variablen *p* vom Typ *PointerType* kann nur die Adresse eines Datenobjekts des Basisdatentyps *BaseType* oder der spezielle Wert *null* zugewiesen werden. Wobei der spezielle Wert *null* (als „ungültige Adresse“) ausdrückt, dass mit dem Zeiger, dem er zugewiesen wurde, (noch) kein Datenobjekt (des Typs *BaseType*) verbunden ist.

Sobald einer Zeigervariablen *p* eine „gültige Adresse“ (ungleich *null*) zugewiesen wurde, kann mit dem *Dereferenzierungsoperator* → über diesen Zeiger indirekt auf die entsprechende Stelle im Speicher – also auf ein Datenobjekt des Typs *BaseType* – zugegriffen werden. Wir notieren das im Pseudocode mit *p*→.

Beispiel 3.12 zeigt zur Erläuterung dieses Konzepts, wie ein Zeiger auf eine Variable gebildet wird. Zeigervariablen, die „normale“ Variablen referenzieren wie in Beispiel 3.12, sind allerdings wenig sinnvoll, denn auf solche Variablen kann direkt zugegriffen werden. Wir benutzen sie hier dennoch, weil so auf leicht verständliche Art und Weise die Idee und die Mechanismen, die hinter dem Zeigerkonzept stecken, erklärt werden können.

Beispiel 3.12 **Zeiger auf Variable**

Für die *integer*-Variable *intVar* aus Beispiel 3.11 zeigt das folgende Codestück, wie einer Zeigervariablen *ptrToIntVar* vom Basisdatentyp *integer* die Adresse von *intVar* zugewiesen und mittels Dereferenzierungsoperationen indirekt damit gearbeitet werden kann.

```
  type IntPtr = →int
  var intVar: int
      ptrToIntVar: IntPtr
begin
  intVar := 17  -- direct access to intVar
  ptrToIntVar := AddrOf(↓intVar)
  Write(ptrToIntVar)    -- writes value of ptrToIntVar, e.g., 2468
  Write(ptrToIntVar→)   -- indirect access to intVar, writes 17
  ptrToIntVar→ := 21    -- indirect access to intVar,
                        --   equivalent to intVar := 21
```

▶Abbildung 3.11 zeigt den Zustand im Hauptspeicher eines Computers nach der Ausführung des obigen Codestücks: In (a) wird der tatsächliche Speicherinhalt wiedergegeben und (b) zeigt die grafische Notation (einen Pfeil), die wir verwenden, um darzustellen, auf welches Datenobjekt eine Zeigervariable zeigt (zur Repräsentation von Zeigern). Der tatsächliche Adresswert ist aus abstrakter Sicht nämlich unerheblich. Aus dem Pfeilsymbol erklärt sich auch die Bezeichnung Zeiger sowie die Verwendung des Zeichens → für die Deklaration eines Zeigertyps und den Dereferenzierungsoperator.

Abbildung 3.11: Zeigervariable, die auf „normale" Variable zeigt

In Abbildung 3.11 sind die Speicherbereiche für die *integer*- und die Zeigervariable deshalb gleich lang gezeichnet, weil für Datenobjekte des Datentyps *integer* in den meisten Programmiersprachen ebenso viel Bytes (meist vier) verwendet werden, wie für Zeigervariablen. Natürlich können mit Zeigervariablen Datenobjekte beliebigen (Basis-)Datentyps referenziert werden, d.h. auch solche, die weniger Speicherplatz (z.B. *char*) oder mehr Speicherplatz (z.B. *real* oder der Verbunddatentyp *Student* aus Abschnitt 3.2) benötigen.

Wir sind nun in der Lage, die Referenzübergabe von Parametern (*call by reference*), auf die wir schon hingewiesen haben, im Detail zu erläutern. Bisher haben wir nur davon gesprochen, dass bei Ausgangs- und Übergangsparametern eine Referenzübergabe zur Anwendung kommt („bei der Ausführung des Algorithmus wird über den Namen des Formalparameters *indirekt* auf den Aktualparameter zugegriffen"), sind aber nicht darauf eingegangen, wie dieser indirekte Zugriff tatsächlich realisiert wird. Die Realisierung erfolgt durch Zeiger.

Beispiel 3.13 zeigt anhand eines Algorithmus zum Vertauschen der Werte zweier Variablen die technische Realisierung der Referenzübergabe durch Zeiger, wie sie üblicherweise von Compilern (bei Verwendung von Programmiersprachen wie Pascal oder C++) umgesetzt wird.

Beispiel 3.13 **Realisierung der Referenzübergabe durch Zeiger**

Ein Algorithmus *Swap* zum Vertauschen der Werte zweier *integer*-Variablen könnte in Pseudocode wie folgt aussehen:

```
Swap(↕a: int  ↕b: int)
  var h: int
begin
  h := a; a := b; b := h
end Swap
```

und folgendermaßen verwendet werden:

```
  var x, y: int
begin
  x := 17; y := 4
  Swap(↕x  ↕y)
```

Transformiert man diesen Algorithmus in ein Programm (d.h. notiert man ihn in einer bestimmten Programmiersprache, z.B. in Pascal oder in C++), übersetzt ihn in eine lauffähige Version und führt ihn aus, dann ergibt sich der in ▶Abbildung 3.12 dargestellte Sachverhalt: die beiden formalen Übergangsparameter *a* und *b* werden zur Laufzeit durch Zeiger realisiert. Die Abbildung stellt den Zustand der Aktual- und Formalparameter im Hauptspeicher unmittelbar nach dem Aufruf des Algorithmus *Swap* dar.

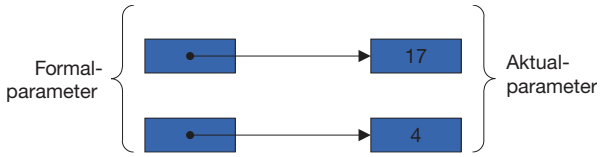

Abbildung 3.12: Formal- und Aktualparameter nach dem Aufruf des Algorithmus *Swap*

Der Algorithmus *Swap* könnte auch so gestaltet werden, dass anstelle der beiden Übergangsparameter vom Datentyp *integer* zwei Eingangsparameter des entsprechenden Zeigerdatentyps verwendet werden:

```
type IntPtr = →int

SwapViaPointers(↓aPtr: IntPtr  ↓bPtr: IntPtr)
  var h: int
begin
  h := a→; a→ := b→; b→ := h
end SwapViaPointers
```

Diesen Algorithmus könnten wir dann zum Vertauschen zweier Zahlen wie folgt verwenden:

```
  var x, y: int
begin
  …
  SwapViaPointers( ↓AddrOf(↓x)  ↓AddrOf(↓y) )
```

Hinweis: Aus verständlichen Gründen raten wir allerdings von dieser Form der Algorithmengestaltung ab, denn die Algorithmenschnittstelle suggeriert, dass der Algorithmus keine Ergebnisobjekte hat. Trotzdem ist es nützlich, die Realisierung der Referenzübergabe (*call by reference*) mittels Zeiger zu kennen, weil es Programmiersprachen gibt (z.B. C und Java), die nur eine Wertübergabe (*call by value*) erlauben. Bei diesen Sprachen muss, wenn der Programmierer eine Referenzübergabe vornehmen möchte, diese von ihm selbst – wie oben gezeigt – implementiert, also mittels Zeigern nachgebildet, also simuliert werden.

Wir haben in Beispiel 3.13 die Ausprägung eines Algorithmus zum Vertauschen der Werte zweier Variablen herangezogen, um das Konzept der Referenzübergabe von Parametern zu illustrieren.

3.3.2 Allokieren und Freigeben von Speicher

Für eine sinnvolle Anwendung des Konzepts der Zeiger-(Datentypen), insbesondere wenn es darum geht, Datenstrukturen wachsen und schrumpfen zu lassen, ist es notwendig, dem Programmierer die Möglichkeit zu geben, Datenobjekte „zur Laufzeit“ zu erzeugen und sie mit anderen bereits existierenden Datenobjekten zu einer vernetzten oder dynamischen Datenstruktur zu verbinden bzw. Datenobjekte daraus wieder zu entfernen. Damit dies möglich ist, muss während der Ausführung eines Algorithmus, also dynamisch, die *Allokation*[3] *von Speicher* (*memory allocation*) erlaubt sein, und dieser muss später auch wieder freigegeben[4] werden können.

Zur Umsetzung dieser Anforderung dient ein ***dynamischer Speicherbereich***, auch *Halde* (*heap*) genannt, und wir benötigen zwei Algorithmen, um diesen Speicherbereich zu verwalten:

1. Einen *Algorithmus für das **Allokieren*** des für ein zu erzeugendes Datenobjekt benötigten Speicherbereichs. Wir verwenden dazu den Funktionsalgorithmus *New* mit folgender Schnittstelle:

   ```
   New(↓Type): →Type
   ```

   Der Algorithmenname *New* drückt aus, dass ein neues Exemplar eines Datenobjekts vom Datentyp *Type* (den wir deshalb als Eingangsparameter dem Algorithmus zur Verfügung stellen) erzeugt wird, d.h. ein Speicherbereich in der Größe *SizeOf* (↓ *Type*) in der Halde allokiert wird. Als Ergebnis liefert *New* die Startadresse des allokierten, aber noch nicht initialisierten Bereichs. Wenn in der Halde kein hinreichend großer zusammenhängender Speicherbereich für das zu erzeugende Datenobjekt mehr frei ist, liefert *New* als Ergebnis den Zeigerwert *null*. (Zur Erinnerung: Der spezielle Wert *null* drückt aus, dass mit der Zeigervariablen, der er zugewiesen wird, noch kein Datenobjekt verbunden, d.h. kein Speicherbereich allokiert ist.)

2. Einen *Algorithmus für das **Freigeben*** eines Speicherbereichs (in der Halde) im Zuge des Entfernens (oft auch Löschen genannt) eines dynamisch erzeugten Datenobjekts aus einer vernetzten Datenstruktur. Wir verwenden dazu den Algorithmus *Dispose* mit folgender Schnittstelle:

   ```
   Dispose(↓p: →Type)
   ```

   Der Algorithmus *Dispose* bekommt als Eingangsparameter den Zeiger *p* auf das zu entfernende Datenobjekt. Nach der Ausführung von *Dispose* steht in der Halde der an der durch *p* angegebenen Adresse beginnende Speicherbereich der Größe *SizeOf*(↓ *Type*) wieder als freier Speicherbereich zur Verfügung. Die Ausführung von *Dispose* bewirkt allerdings nicht, dass dem Zeiger *p* der Wert *null* zugewiesen wird (*p* ist „nur“ ein Eingangsparameter).

---

3 Das zugehörige Verb lautet zwar allozieren, nachdem sich in der Informatik aber die Schreibweise *allokieren* (abgeleitet aus dem Engl. *to allocate*) eingebürgert hat, verwenden wir diese auch im Rahmen des vorliegenden Buchs.

4 Wir gehen davon aus, dass sich der Programmierer selbst um die Freigabe von allokiertem Speicher kümmern muss wie das z.B. bei Pascal und C der Fall ist, dass also keine *automatische Speicherbereinigung* (*garbage collection*) erfolgt, die z.B. in Java und C# durchgeführt wird.

Da ein mit *New* erfolgreich allokierter Speicherbereich bis auf den Namen alle Eigenschaften einer „normalen“ Variablen hat, nämlich eine Startadresse, einen Datentyp und einen aktuellen Wert (der definiert ist, wenn nach der Allokation des Speicherbereichs eine Wertzuweisung erfolgt ist), bezeichnet man solch einen Speicherbereich auch als *anonyme Variable*. Nachdem man diese nicht direkt über einen Namen ansprechen kann, muss der Zugriff indirekt über einen Zeiger, welcher ihre Adresse enthält, erfolgen (wir nennen das auch *Dereferenzierung*).

Beispiel 3.14 zeigt, wie anonyme Variablen erzeugt und wieder freigegeben werden können.

Beispiel 3.14

Allokieren, Verwenden und Freigeben von anonymen Variablen

Das folgende Codestück zeigt, wie Speicherplatz für ein *integer*-Datenobjekt (eine anonyme *integer*-Variable) allokiert, dieses dann über die Zeigervariable *intPtr* indirekt manipuliert wird, und wie der Speicherbereich schließlich wieder freigegeben wird.

```
  var intPtr: →int  -- no explicitly defined data type necessary
begin
  intPtr := New(↓int)
  if intPtr = null then
    Write(↓“Error: heap overflow”)
    halt
  end -- if
  Write(↓intPtr)   -- writes address of reserved memory area
  intPtr→ := 17
  Write(↓intPtr→) -- writes 17
  Dispose(↓intPtr)
  intPtr := null   -- assign correct actual value
```

▶Abbildung 3.13 zeigt in (a) den Zustand nach der Speicherallokation mittels *New* und der Ausführung aller Anweisungen bis zum Aufruf des Algorithmus *Dispose*, in (b) den Zustand, unmittelbar nachdem der allokierte Speicherbereich mittels *Dispose* wieder freigegeben und in (c) den Zustand, nachdem die Zeigervariable auf *null* gesetzt wurde.

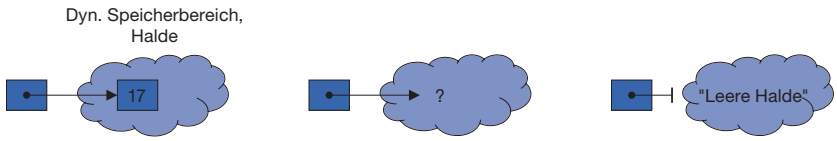

Abbildung 3.13: Zustände nach dem Allokieren und Freigeben von Speicher im dynamischen Speicherbereich

Hinweis: Der in ▶Abbildung 3.13 (b) dargestellte Zustand wird als *baumelnder Zeiger* (*dangling pointer*) bezeichnet. Eine Verwendung eines Zeigers in diesem Zustand führt zu schwer zu lokalisierenden Fehlern.

Dynamische Felder

Primär wird die Möglichkeit der dynamischen Speicherverwaltung zwar zur Realisierung von vernetzten Datenstrukturen genutzt, wie in den folgenden Abschnitten über verkettete Listen und Bäume gezeigt wird, aber auch Felder können damit in einem gewissen Sinn dynamisch gestaltet werden.

Während für „normale“ Felder die Anzahl der Elemente durch die Angabe des Indexbereichs in Form von Literalen oder Konstanten bei der Deklaration fixiert werden muss (deshalb auch die Bezeichnung *statische Felder*), können mit Zeigertypen, die als Basisdatentypen Felder mit oben offenen Indexbereichen haben, und einer speziellen Version des Speicher-Allokationsalgorithmus *New* so genannte *dynamische Felder* (*dynamic arrays*) realisiert werden, deren Größe (Anzahl der Elemente) erst zur Laufzeit festgelegt wird. Diese neue Version des *New*-Algorithmus erlaubt es, Speicher für mehrere, im Speicher unmittelbar hintereinander liegende, Exemplare des Basisdatentyps zu allokieren.

Beispiel 3.15 zeigt eine einfache Anwendung dynamischer Felder für die Realisierung von C-Zeichenketten, deren Länge erst zur Laufzeit feststeht; vgl. auch Beispiel 3.5.

Beispiel 3.15 **Dynamisches Feld für eine C-Zeichenkette**

Das folgende Codestück zeigt ein einfaches Beispiel eines dynamischen Felds für die Realisierung einer C-Zeichenkette. Hier wird die maximale Anzahl der Elemente (Zeichen) zur Laufzeit eingelesen, bevor Speicher für ein Feld mit dieser Größe allokiert wird. Der Zugriff auf das Feld erfolgt über eine Zeigervariable.

```
  type CStringPtr: →array [0:] of char  -- no upper limit for index
                                        -- range
  var maxStrLen: int  -- number of elements in dynamic array
      csp: CStringPtr
begin
  Read(↑maxStrLen)
  csp := New(↓char ↓maxStrLen) -- allocate SizeOf(↓char) * maxStrLen bytes
  csp→[0] := Char(↓0)          -- initialize to empty string
  …  -- operate with csp, at least assign a string to csp
  -- write actual string
  i := 0
  while csp→[i] ≠ Chr(↓0) do
    Write(csp→[i])
    i := i + 1
  end -- while
  Dispose(↓csp)
  csp := null
```

3.4 Verkettete Listen

In der Einleitung des Abschnitts 3.3 haben wir darauf hingewiesen, dass *verkettete Listen* (*linked lists*) zu den wichtigen Repräsentanten vernetzter Datenstrukturen gehören. Nun stellt sich die Frage, wann und wozu wir verkettete Listen benötigen. Um diese Frage zu beantworten, wollen wir noch einmal auf die bereits bekannten Felder zurückkommen.

3.4.1 Von Feldern zu verketteten Listen

Eindimensionale Felder fassen Elemente gleichen Datentyps in einer sequenziellen Anordnung von Speicherzellen so zusammen, dass ein effizienter Zugriff (in konstanter Zeit) auf die einzelnen Elemente des Feldobjekts über einen Index möglich ist. Darin liegt auch der große Vorteil von Feldobjekten. Der Nachteil, dass die Anzahl der Elemente bei statischen Feldern bekannt sein muss, lässt sich zwar – wie im vorigen Abschnitt gezeigt – mit dynamisch erzeugten Feldern beseitigen, aber auch dynamische Felder können voll werden.

Einen weiteren und gravierenden Nachteil weisen aber alle Felder auf: Das Entfernen bzw. das Einfügen eines Datenobjekts in einem Feld verursacht, wie in ▶Abbildung 3.14 (a) bzw. (b) gezeigt, großen Aufwand, da die Datenobjekte, die hinter der Entfern- oder Einfügeposition liegen, durch elementweises Umkopieren um eine Stelle nach vorne (beim Entfernen) bzw. nach hinten (beim Einfügen) verschoben werden müssen. (Die einzuhaltende Reihenfolge der Operationen ist durch Nummern in Abbildung 3.14 angegeben.)

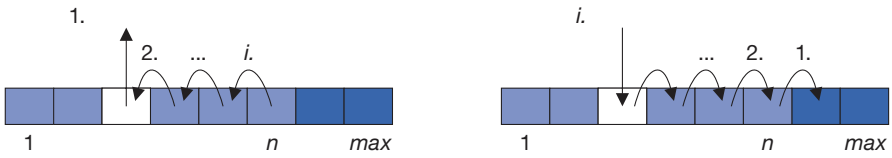

Abbildung 3.14: Entfernen und Einfügen von Elementen in einem Feld

Mittels verketteter Listen ist es möglich – wie bei eindimensionalen Feldern – eine sequenzielle Anordnung von so genannten *Knoten* (*nodes*) im Speicher zu bilden und dabei die oben erwähnten Nachteile von Feldern zu umgehen.

Definition: *Verkettete Liste*

Eine *verkette Liste* (*linked list*) ist entweder leer oder sie repräsentiert eine sequenzielle Anordnung von *Knoten* (*nodes*) gleichen Datentyps, deren Reihenfolge explizit durch *Verkettung* (*linkage*) festgelegt ist.

Diese Verkettung wird durch in den Knoten enthaltene Zeiger realisiert, die

- bei *einfach-verketteten Listen* (*singly-linked lists*) auf den jeweils nächsten Knoten, den *Nachfolger* (*successor*), und
- bei *doppelt-verketteten Listen* (*doubly-linked lists*) auch auf den jeweils vorangehenden Knoten, den *Vorgänger* (*predecessor*),

verweisen.

Der wesentliche Unterschied im Vergleich zu Feldern liegt also darin, dass sich die Reihenfolge der Knoten nicht (implizit) aus deren lückenloser Aneinanderreihung im Speicher (mit daraus resultierenden aufsteigenden Startadressen wie bei Feldelementen), sondern durch die explizite *Verkettung* ergibt.

▶Abbildung 3.15 zeigt in abstrakter Form in (a) eine einfach-verkettete und in (b) eine doppelt-verkettete Liste.

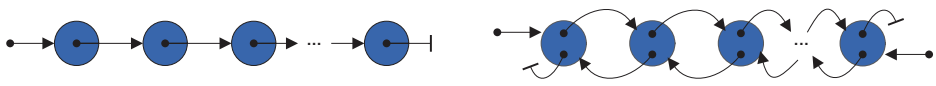

Abbildung 3.15: Einfach- und doppelt-verkettete Listen

Damit können die Nachteile von Feldern vermieden werden: Die Liste kann (solange im dynamischen Speicherbereich noch genügend freier Speicherplatz vorhanden ist) beliebig ausgedehnt werden und Knoten können effizient entfernt und hinzugefügt werden, da nur Zeiger in jenen Knoten, die in der Umgebung der Entfern- oder Einfügestelle liegen, verändert werden müssen, also kein Verschieben der restlichen Knoten erforderlich ist. Auch ein Umreihen der Knoten ist ohne großen Aufwand durch Ändern der Zeigerwerte möglich.

Einfach- und doppelt-verkettete Listen können knotenweise von vorne und doppelt-verkettete auch von hinten her durchlaufen und manipuliert werden. Ein effizienter „direkter“ Zugriff (in konstanter Zeit) auf einen beliebigen Knoten einer verketteten Liste ist – im Gegensatz zum Index-basierten Zugriff auf Elemente von Feldern – aber nicht möglich. Das ist der wesentliche Nachteil verketteter Listen. Darüber hinaus erfordert die Verwaltung verketteter Listen nicht nur komplexere Algorithmen, sondern durch die dynamische Speicherverwaltung für das Allokieren und Einfügen sowie das Entfernen und Freigeben eines Knotens zusätzlichen Zeitaufwand, und die Zeiger, die in den Knoten enthalten sind, um die Verkettung zu bewerkstelligen, verursachen zusätzlichen Speicherbedarf.

Bevor wir uns in den folgenden beiden Abschnitten mit der konkreten Realisierung verketteter Listen auseinander setzen, fassen wir die wesentlichen Unterschiede zwischen Feldern und verketteten Listen in Tabelle 3.2 zusammen.

Tabelle 3.2

Gegenüberstellung von Feldern und verketteten Listen

Unterscheidungsmerkmal	Eindimensionales Feld	Verkettete Liste
Größe und Struktur	Fix	Veränderlich
Zugriff auf *i*-tes Element bzw. *i*-ten Knoten	Effizient, in konstanter Zeit mit Indizierungsoperation	Ineffizient, da (z.B. in einer Schleife) *i*-ter Knoten gesucht werden muss
Löschen und Einfügen	Ineffizient, da restliche Datenobjekte elementweise verschoben werden müssen	Effizient, da die Struktur (Verkettung) nur lokal verändert werden muss

3.4.2 Einfach-verkettete Listen

Was wir generell unter einer vernetzten oder dynamischen Datenstruktur verstehen, haben wir am Beginn des Abschnitts 3.3 bereits erörtert. Wir können ihre spezielle Ausprägung in Form einer einfach-verketteten Liste – in Anlehnung an die allgemeine Definition einer verketteten Liste – nun auch rekursiv definieren. Rekursiv heißt vereinfacht ausgedrückt: auf sich selbst zurückgreifend. (Der Leser sei darauf hingewiesen, dass Kapitel 4 dem Thema Rekursion und rekursiven Algorithmen gewidmet ist.)

Rekursive Definition: *Einfach-verkettete Liste*

Eine *einfach-verkettete Liste* (*singly-linked list*) ist entweder leer (d.h. sie hat keine Knoten) oder sie besteht aus einem ausgezeichneten Knoten, dem *Kopf* der Liste (*head*), der mit einer einfach-verketteten Liste, dem *Rest* der Liste oder *Schwanz* (*tail*), mittels eines Zeigers verbunden ist.

Die rekursive Definition einer einfach-verketteten Liste legt nahe, dass wir für ihre Realisierung einen so genannten *rekursiven Datentyp* (*recursive data type*) verwenden. Rekursiv heißt dieser Datentyp, da in seiner Deklaration auf ihn selbst zurückgegriffen wird. Wir nennen ihn *ListPtr* und deklarieren ihn wie folgt:

```
type ListPtr = →compound  -- pointer to first node = head of list
       tail: ListPtr      -- pointer to tail of list
       … -- any data
     end -- compound
```

Wegen der Möglichkeit dieser rekursiven Definition werden verkettete Listen auch *rekursive Datenobjekte* oder *-strukturen* genannt. (Die Rekursion ermöglicht auch die einfache Anwendung rekursiver Algorithmen zu ihrer Bearbeitung, wie wir in Kapitel 4 zeigen werden.)

Um aber schon vom Datentyp her zwischen „einzelnen“ Knoten und einer „ganzen“ Liste unterscheiden zu können, wie es auch in der allgemeinen Definition für verkettete Listen gemacht wurde, wollen wir einfach-verkettete Listen wie folgt deklarieren:

```
type ListNodePtr = →ListNode
     ListNode = compound
       next: ListNodePtr
       data: int
     end -- compound
     ListPtr = ListNodePtr  -- synonym for ListNodePtr,
                            -- but ListPtr is for pointers to head node

var  list: ListPtr
```

Wir haben den Zeigertyp *ListPtr* so definiert, dass Zeiger dieses Typs Datenobjekte des Typs *ListNode* referenzieren. Der Datentyp für die Knoten der Liste (*ListNode*) ist klarerweise ein Verbunddatentyp, der

- eine Komponente *next* zur Realisierung der Verbindung eines Knotens mit seinem Nachfolgerknoten und
- eine Komponente *data* zur Aufnahme der Knotendaten (in der Deklaration oben ist das „nur“ ein *integer*-Wert, weil das für unsere Demonstrationszwecke ausreicht, in der Praxis können das auch mehrere Datenkomponenten beliebiger Datentypen sein)

vorsieht. Für die Repräsentation der bzw. den Zugriff auf die Liste, d.h. auf ihren ersten Knoten (Kopf), führen wir die Variable *list* ein, die deshalb vom Datentyp *ListPtr* sein muss. Vom Kopf aus kann man jeweils über die Komponente *next* zu den restlichen Knoten der Liste gelangen.

▶Abbildung 3.16 zeigt die einfach-verkettete Liste *list* mit ihren Knoten gemäß obiger Deklarationen, die im dynamischen Speicherbereich angelegt wurden.

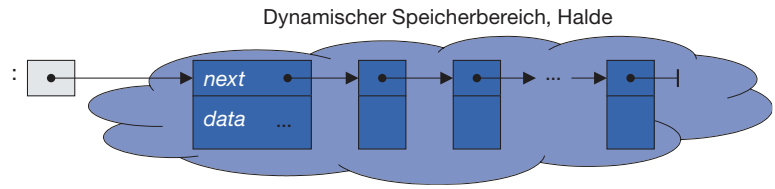

Abbildung 3.16: Einfach-verkettete Liste *list* mit ihren Knoten im dynamischen Speicherbereich

In den meisten Anwendungsfällen ist es angebracht, bevor die eigentliche Listenverarbeitung beginnt, eine sinnvolle Initialisierung der Liste, z.B. als leere Liste, vorzunehmen. Ein Algorithmus, der eine leere Liste liefert, kann z.B. folgendermaßen aufgebaut sein:

```
EmptyList(): ListPtr
begin
  return null
end EmptyList
```

Durch Aufruf von *EmptyList* kann die Listenvariable *list* initialisiert werden:

```
list := EmptyList()
```

Damit eine Liste mit Knoten „bevölkert“ werden kann, benötigt man einen Algorithmus, der einen Knoten erzeugt. Zwar bietet die dynamische Speicherverwaltung mit dem Aufruf des Funktionsalgorithmus *New*(↓*ListNode*) bereits die Möglichkeit, Speicherplatz für einen neuen Knoten anzulegen, es ist jedoch zweckmäßig, dass wir einen speziellen (Funktions-)Algorithmus *NewListNode* definieren, der nicht nur die Speicherallokation übernimmt, sondern auch prüft, ob der dynamische Speicher voll ist (und in diesem Fall eine Fehlermeldung ausgibt und die Ausführung beendet) bzw., wenn der benötigte Speicherplatz erfolgreich allokiert werden konnte, alle Komponenten des neu erzeugten Knotens initialisiert.

```
NewListNode(↓data: int): ListNodePtr
  var n: ListNodePtr
begin
  n := New(↓ListNode)
  if n = null then
    Write(↓"Error: heap overflow")
    halt
  end -- if
  n→next := null
  n→data := data
  return n
end NewListNode
```

Einfügen eines Knotens vorne

Oft ist es erforderlich, einen Knoten *n* in eine einfach-verkettete Liste *list* vorne, d.h. als neuen ersten Knoten, einzufügen (siehe ▶Abbildung 3.17; durch die Nummerierung der neu einzurichtenden Zeiger wird ausgedrückt, in welcher Reihenfolge die entsprechenden Wertzuweisungen erfolgen müssen).

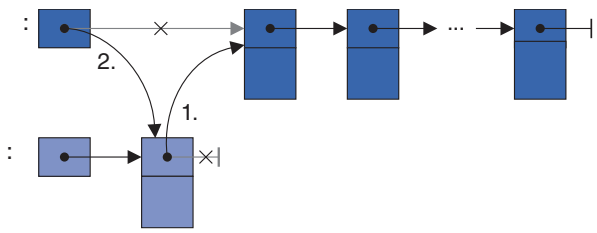

Abbildung 3.17: Einfügen eines Knotens vorne *n* in die Liste *list*

Ein Algorithmus *Prepend* zum Einfügen eines Knotens *n* vorne in eine einfach-verkettete Liste *list* (wie in Abbildung 3.17 exemplarisch dargestellt) lautet folgendermaßen:

```
Prepend(↕list: ListPtr  ↓n: ListNodePtr)
begin
  n→next := list    -- 1. append old list to new node
  list := n         -- 2. new list = ( n, old list )
end Prepend
```

Beispiel 3.16 zeigt, wie man mittels *Prepend* eine einfach-verkettete Liste mit zehn Knoten aufbauen kann, welche die Werte von 1 bis 10 enthalten.

Beispiel 3.16 **Aufbau einer einfach-verketteten Liste mit zehn Knoten durch Einfügen vorne**

Zur Illustration des bisher zu einfach-verketteten Listen Gesagten, geben wir ein Codestück an, durch das eine einfach-verkettete Liste von Knoten des Typs *ListNode* mit zehn Knoten, deren Datenkomponenten die Werte 1, 2, … bis 10 in der angegebenen Reihenfolge enthalten, erzeugt wird:

```
  var n: ListNodePtr
begin
  list := EmptyList()
  i := 10
  while i > 0 do
    n := NewListNode(↓i)  -- create and initialize new node
    Prepend(↕list  ↓n)  -- insert node as new front node
    i := i - 1
  end -- while
```

Natürlich kann man in obigem Codestück auch ohne die Hilfsvariable *n* auskommen, indem man das Erzeugen eines neuen Knotens in die Aktualparameterliste für den Aufruf des Algorithmus *Prepend* verlagert:

```
Prepend(↕list  ↓NewListNode(↓i))  -- insert new node in front of list
```

Anfügen eines Knotens hinten

Das Anfügen eines Knotens *n* am Ende der einfach-verketteten Liste *list* ist im allgemeinen Fall schwieriger und aufwändiger als das Einfügen vorne. Das Listenende ist nur über die Verweiskette identifizierbar, d.h. es muss ausgehend vom ersten Knoten der letzte Knoten (*last*) gesucht werden, um dahinter den Knoten *n* anzufügen ▶Abbildung 3.18.

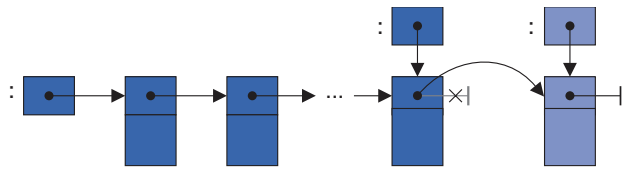

Abbildung 3.18: Anfügen eines Knotens *n* am Ende der Liste *list*

Ein Algorithmus *Append* zum Anfügen eines Knotens *n* hinten an eine einfach-verkettete Liste *list* (wie in Abbildung 3.18 dargestellt) lautet folgendermaßen:

```
Append(↕list: ListPtr  ↓n: ListNodePtr)
  var last: ListNodePtr
begin
  if list = null then  -- list is empty
    list := n
  else  -- list ≠ null, list is not empty
    -- search last node
    last := list
    while last→next ≠ null do
      last := last→next
    end -- while
    -- append n to last node
    last→next := n  -- new list = ( old list, n )
  end -- if
end Append
```

Beispiel 3.17 zeigt nun, wie man mittels *Append* eine einfach-verkettete Liste mit zehn Knoten aufbauen kann, welche die Werte von 1 bis 10 enthalten.

Beispiel 3.17

Aufbau einer einfach-verketteten Liste mit zehn Knoten durch Anfügen hinten

Nachdem in Beispiel 3.16 eine einfach-verkettete Liste von Knoten des Typs *ListNode* mit zehn Knoten, deren Datenkomponenten die Werte 1, 2, … bis 10 in der angegebenen Reihenfolge enthalten, durch Einfügen von Knoten vorne erzeugt wurde, geben wir nun ein (kürzeres) Codestück an, das dieselbe Liste aufbaut, indem die Knoten hinten angefügt werden:

```
list := EmptyList()
for i := 1 to 10 do
  Append(↕list  ↓NewListNode(↓i))  -- insert new node as last node
end -- for
```

Beide Strategien (vorne ein- bzw. hinten anfügen) liefern zwar dasselbe Ergebnis, die zweite ist jedoch wesentlich langsamer, da jedes Mal erneut und von vorne weg das Ende der Liste gesucht werden muss, bevor ein Knoten angefügt werden kann.

Hinweis: Wenn es häufig notwendig ist, Knoten hinten an eine Liste anzufügen, ist es sinnvoll, einen zusätzlichen Zeiger auf den letzten Knoten der Liste zu verwalten. Damit kann man die *Append*-Operation genauso effizient implementieren wie die *Prepend*-Operation.

Eine häufig vorkommende Operation ist das Suchen eines bestimmten Knotens. Die unten angegebene Funktion *FirstNodeWith* ist ein Algorithmus, der, wenn die einfach-verkettete Liste *list* einen Knoten, der in seiner Datenkomponente den gesuchten Wert *x* enthält, einen Zeiger auf diesen Konten (genauer gesagt: auf den ersten Knoten in

der Verweiskette, für den die Bedingung zutrifft) als Ergebnis liefert (vgl. *sequenzielle Suche* in Kapitel 6). Kommt der gesuchte Wert *x* in *list* nicht vor, liefert *FirstNodeWith* als Ergebnis *null*.

```
FirstNodeWith(↓list: ListPtr  ↓x: int): ListNodePtr
  var n: ListNodePtr
begin
  n := list
  while (n ≠ null) and (n→data ≠ x) do
    n := n→next
  end -- while
  return n
end FirstNodeWith
```

Sortierte Listen und sortiertes Einfügen

Eine besondere Art von Listen sind solche, bei denen die Datenkomponenten der Knoten in einer bestimmten Weise geordnet sind. Beispielsweise sprechen wir von einer *aufsteigend sortierten Liste*, wenn für jeden Knoten gilt, dass der Wert in seiner Datenkomponente kleiner oder gleich dem Wert in der Datenkomponente des nachfolgenden Knotens ist (siehe ▶Abbildung 3.19).

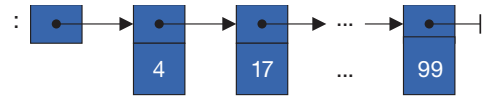

Abbildung 3.19: Aufsteigend sortierte Liste *list*

Sortierte Listen haben den Vorteil, dass bei der Suche eines Knotens der Suchvorgang (frühzeitig) abgebrochen werden kann, wenn man auf einen Knoten trifft, dessen Datenkomponente einen Wert enthält der größer oder kleiner (je nach Sortierung) als der gesuchte Wert ist.

Der unten angegebene Funktionsalgorithmus *IsSorted* prüft, ob eine einfach-verkettete Liste aufsteigend sortiert ist.

```
IsSorted(↓list: ListPtr): bool
  var n: ListNodePtr
begin
  if (list = null) or (list→next = null) then
    return true
  else  -- list has at least two nodes
    n := list
    while (n→next ≠ null) and (n→data ≤ n→next→data) do
      n := n→next
    end -- while
    return n→next = null
  end -- if
end IsSorted
```

Zwar können unsortierte verkettete Listen durch Umreihen der Knoten in sortierte transformiert werden, die einfachste und effizienteste Art, eine sortierte Liste herzustellen, besteht allerdings darin, beginnend mit der leeren Liste neue Knoten von vornherein an der richtigen Stelle einzufügen. Die korrekte Einfügeposition für einen Knoten *n* in der aufsteigend sortierten Liste wird ermittelt, indem der Vorgänger (*predecessor*, kurz *pred*) und der Nachfolger (*successor*, kurz *succ*) für den neuen Knoten ermittelt werden. Wenn die Einfügeposition gefunden ist, kann der Knoten *n* in die Liste *list* eingefügt werden, wie beispielhaft in ▶Abbildung 3.20 dargestellt.

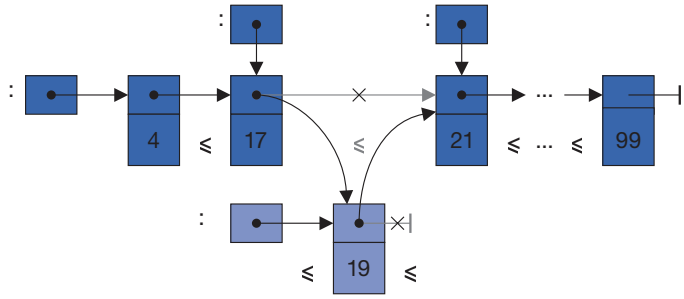

Abbildung 3.20: Einfügen eines Knotens *n* in die aufsteigend sortierte Liste *list*

Für das Einfügen eines Knotens *n* in eine aufsteigend sortierte, einfach-verkettete Liste *list* können wir folgenden Algorithmus *SortedInsert* heranziehen:

```
SortedInsert(↕list: ListPtr  ↓n: ListNodePtr)
  var pred, succ: ListNodePtr
begin
  -- 1. look for insertion position
  pred := null
  succ := list
  while (succ ≠ null) and (n→data > succ→data) do
    pred := succ
    succ := succ→next
  end -- while
  -- 2. insert node between pred and succ
  if pred = null then  -- prepend new node
    list := n
  else -- pred ≠ null
    pred→next := n
  end -- if
  n→next := succ
end SortedInsert
```

Löschen eines Knotens und der gesamten verketteten Liste

Nach den erörterten drei Möglichkeiten, einen Knoten in eine verkettete Liste einzufügen (vorne, hinten oder an der richtigen Stelle in einer sortierten Liste), wollen wir zeigen, wie der erste Knoten, dessen Datenkomponente einen bestimmten Wert *x* enthält, aus der Liste *list* entfernt und gelöscht werden kann. Dazu muss nicht nur dieser Knoten *n*, sondern auch sein unmittelbarer Vorgänger *pred* ermittelt werden, wie in ▶Abbildung 3.21 dargestellt.

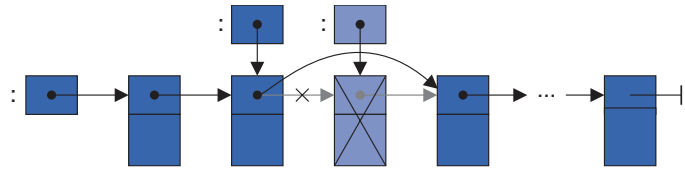

Abbildung 3.21: Entfernen eines Knotens *n* aus der Liste *list*

Dazu formulieren wir (in Analogie zum Funktionsalgorithmus *FirstNodeWith* für die Suche) einen Algorithmus *DeleteNodeWith*:

```
DeleteNodeWith(↕list: ListPtr  ↓x: int)
  var pred, n: ListNodePtr
begin
  -- 1. look for node n to delete
  pred := null
  n := list
  while (n ≠ null) and (n→data ≠ x) do
    pred := n
    n := n→next
  end -- while
  -- 2. delete node n
  if n ≠ null then  -- node found
    if pred = null then  -- node to delete is first node in list
      list := n→next
    else
      pred→next := n→next
    end -- if
    Dispose(↓n)
  end -- if
end DeleteNodeWith
```

Wollen wir aus einer Liste alle darin enthaltenen Knoten entfernen und den von ihnen belegten Speicher wieder freigeben, so können wir dies, indem wir beispielsweise folgenden Algorithmus *DisposeList* dazu heranziehen:

```
DisposeList(↕list: ListPtr)
  var next: ListNodePtr
begin
  while list ≠ null do
    next := l→next  -- save pointer to next node
    Dispose(↓list)
    list := next
  end -- while
end DisposeList
```

Über die in diesem Abschnitt gezeigten hinaus sind weitere nützliche Operationen auf einfach-verketteten Listen denkbar, die in Anlehnung an die hier vorgestellten Algorithmen auf einfache Art und Weise realisiert werden können. Viele dieser Operationen können auch oder sogar einfacher mittels rekursiver Algorithmen implementiert werden, weil wir – wie eingangs erläutert – die verketteten Listen als rekursive Datenstrukturen auffassen können. Beispiele für rekursive Listenalgorithmen finden sich im Kapitel 4.

3.4.3 Doppelt-verkettete Listen

Ein Nachteil einfach-verketteter Listen liegt darin, dass ihre Knoten nur in einer Richtung, nämlich von vorne nach hinten, besucht werden können.

Doppelt-verkettete Listen sind eine Erweiterung der einfach-verketteten, in der jeder Knoten nicht nur über einen Zeiger mit dem nächsten (*next*), sondern auch mit dem vorangehenden (*previous*, kurz *prev*) Knoten verbunden ist. Dazu muss der Verbunddatentyp *ListNode* für die Knoten um eine Komponente erweitert werden:

```
type ListNodePtr = →ListNode
     ListNode = compound
       prev, next: ListNodePtr
       data: int
     end -- compound
```

Anmerkung: Es wäre durchaus überlegenswert, den Typnamen so zu wählen, dass daraus hervorgeht, ob wir es mit einfach- oder doppelt-verketteten Listen zu tun haben, indem wir z.B. zwischen *DoublyLinkedListNode(Ptr)* und *SinglyLinkedListNode(Ptr)* unterscheiden. Da es selten vorkommt, dass beide Listenarten gleichzeitig verwendet werden, und die spezifischen Namen lang sind, wollen wir bei den kurzen und allgemeinen Typnamen *ListNode(Ptr)* bleiben, denn aus dem Kontext ist klar, für welche Art von Knoten/Liste diese Typen verwendet werden.

Um aus der doppelten Verkettung Nutzen ziehen zu können, müssen wir den Datentyp *List*, den wir für die Realisierung einer doppelt-verketteten Liste verwenden, so definieren, dass aus dem bisherigen Zeigerdatentyp *ListPtr* (Zeiger auf den ersten Knoten der einfach-verketteten Liste) ein Verbunddatentyp wird, der über zwei Komponenten verfügt: einen Zeiger auf den ersten (*first*) und einen Zeiger auf den letzten (*last*) Knoten, gemäß folgender Deklaration:

```
type List = compound
       first, last: ListNodePtr
     end -- compound
```

Beim Versuch der Implementierung der elementaren Listenoperationen (vor allem *Prepend*, *Append* und *SortedInsert*) auf Basis des obigen *List*-Datentyps stellt man aber schnell fest, dass diese wegen der zu berücksichtigenden Sonderfälle und der Tatsache, dass viele der beteiligten Zeiger (*first*, *last*, *prev* im ersten und *next* im letzten Knoten) den Wert *null* annehmen können, kompliziert und umfangreich werden.

Ein einfacher Trick, der dafür sorgt, dass die Listenverarbeitungsalgorithmen vereinheitlicht werden können, besteht darin, einen eigenen Listenkoten als ***Anker*** (*anchor*) zu verwenden, von dem aus die Liste verwaltet wird. ▶Abbildung 3.22 illustriert die Repräsentation einer solchen doppelt-verketteten Liste. Wie in (a) dargestellt, enthält die leere Liste bereits einen Knoten: den Ankerknoten, dessen *prev*- und *next*-Komponenten auf ihn selbst verweisen. Bei einer nicht leeren Liste – wie in (b) dargestellt – können

1. ausgehend von der *prev*-Komponente des Ankerknotens alle Listenknoten über deren *prev*-Komponente von hinten her besucht werden, so dass man wieder beim Anker landet, und
2. ausgehend von der *next*-Komponente des Ankerknotens alle Listenknoten über deren *next*-Komponente von vorne weg besucht werden, so dass man ebenfalls wieder beim Anker landet.

Diese Art der Verkettung bietet also die Möglichkeit, die Listenknoten im Kreis zu besuchen, man nennt deshalb solche Listen auch *zirkuläre Listen* (*circular lists*). Diese Eigenschaft kann zwar auch bei einfacher Verkettung erreicht werden, indem die *next*-Komponente des letzten Knotens auf den ersten verweist, ist dort aber wenig nützlich.

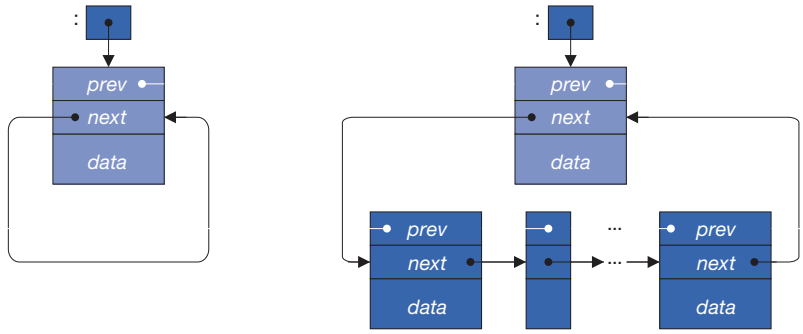

Abbildung 3.22: Realisierung doppelt-verketteter Listen mit einem Ankerknoten

Als Datentyp für eine doppelt-verkettete Liste mit Ankerknoten verwenden wir – wie bei einer einfach-verketteten Liste – wieder einen Zeigerdatentyp mit dem Basisdatentyp *ListNode* (wie oben angegeben mit der zusätzlichen *prev*-Komponente), also:

```
type ListPtr = ListNodePtr
```

Die Implementierung dieser Listenform erfordert eine Änderung des Funktionsalgorithmus *NewNode* für das Anlegen eines initialisierten Knotens:

```
NewNode(↓data: int): ListNodePtr
  var n: ListNodePtr
begin
  n := New(↓ListPtr)
  if n = null then …  -- report heap overflow error and terminate
  n→prev := n
  n→next := n
  n→data := data
  return n
end NewNode
```

Nachdem eine leere Liste aus dem Ankerknoten besteht, muss der Funktionsalgorithmus *EmptyList* wie folgt abgeändert werden:

```
EmptyList(): ListPtr
begin
  return NewNode(↓0)  -- anchor node with dummy data
end EmptyList
```

Im Vergleich zur einfach-verketteten Liste ist der Algorithmus *Prepend* für das Einfügen eines Knotens *n* vorne in die Liste *list* etwas länger, jener für das Anfügen (*Append*) hinten etwas kürzer. Beide erledigen ihre Aufgabe aber schnell und in konstanter Zeit. Sie sind außerdem einheitlich, weil sich der eine aus dem anderen ergibt, indem man an allen Stellen die Komponentenbezeichnungen *prev* und *next* miteinander vertauscht. Tabelle 3.3 stellt die beiden Algorithmen einander gegenüber.

Tabelle 3.3

Algorithmen *Prepend* und *Append* für doppelt-verkettete Listen mit Anker

```
Prepend(↕list: ListPtr
        ↓n: ListNodePtr)
begin
  n→prev := list
  n→next := list→next
  list→next := n
  n→next→prev := n
end Prepend
```

```
Append(↕list: ListPtr
       ↓n: ListNodePtr)
begin
  n→next := list
  n→prev := list→prev
  list→prev := n
  n→prev→next := n
end Append
```

Auch der Algorithmus für das Einfügen eines Knotens *n* in eine sortierte Liste *list* wird im Vergleich zur einfach-verketteten Liste kürzer und einfacher, da nur mehr der Nachfolger (*successor*, kurz *succ*) des einzufügenden Knotens gesucht werden muss. Die Effizienz steigt nur unwesentlich. Der für doppelt-verkettete Listen entsprechend angepasste Algorithmus *SortedInsert* lautet dann:

```
SortedInsert(↕list: ListPtr  ↓n: ListNodePtr)
  var succ: ListNodePtr
begin
  -- 1. look for insertion position
  succ := list→next
  while (succ ≠ list) and (n→data > succ→data) do
    succ := succ→next
  end -- while
  -- 2. insert node before succ
  n→prev := succ→prev
  n→next := succ
  succ→prev→next := n
  succ→prev := n
end SortedInsert
```

Beim Suchen eines bestimmten Knotens in der Liste kann die Komponente *data* im Anker als so genannter ***Wächter*** (*sentinel*) herangezogen werden: Der gesuchte Wert wird zu Beginn im Ankerknoten abgelegt, damit garantiert ist, dass er dort nach einem vollen Durchlauf gefunden wird – unabhängig davon, ob von vorne weg nach dem ersten oder von hinten her nach dem letzten Auftreten gesucht wird. Damit kann die Abbruchbedingung in der *while*-Schleife vereinfacht werden, denn der Test, ob $n \neq null$ ist, kann entfallen, wodurch sich die Suchzeiten in langen Listen reduzieren (vgl. *sequenzielle Suche* in Kapitel 6).

```
FirstNodeWith(↓list: ListPtr  ↓x: int): ListNodePtr
  var n: ListNodePtr
begin
  list→data := x  -- initialize sentinel
  n := list→next
  while n→data ≠ x do
    n := n→next
  end -- while
  if n = list then  -- x found in anchor
    return null
  else
    return n
  end -- if
end FirstNodeWith
```

Viele der Algorithmen für die Bearbeitung einfach-verketteter Listen können praktisch unverändert für doppelt-verkettete Listen mit Anker übernommen werden (z.B. der Funktionsalgorithmus *IsSorted* aus Abschnitt 3.4.1).

Die zusätzlichen Möglichkeiten (vor allem das Durchlaufen der Liste auch von hinten her) und die einfacheren und z. T. effizienteren Algorithmen legen nahe, verkettete Listen nur in Form doppelt-verketteter Listen mit Anker zu implementieren.

3.5 Bäume, Binärbäume und binäre Suchbäume

In der Praxis begegnen wir oft Aufgabenstellungen, in denen hierarchisch strukturierte Sachverhalte oder Objekte vorkommen; beispielsweise der hierarchische Aufbau einer Organisation oder die (hierarchische) Strukturierung eines Dokuments, z.B. die Struktur des vorliegenden Buchs. Es versteht sich daher von selbst, dass wir uns im Sinne einer möglichst realitätsnahen Modellierung wünschen, beim Entwurf von Algorithmen Datenobjekte modellieren zu können, die solche hierarchisch organisierten Sachverhalte oder Objekte repräsentieren. In der Informatik nennen wir hierarchisch strukturierte Objekte *Bäume* (*trees*).

3.5.1 Bäume

In der Literatur findet man eine Vielzahl unterschiedlicher Definitionen für den Baum-Begriff. In Anlehnung an die Definition von Niklaus Wirth in [Wirth 1975] und in Analogie zur Definition der verketteten Liste definieren wir einen Baum wie folgt:

Rekursive Definition: *(Allgemeiner) Baum*

Ein (*allgemeiner*) *Baum* (*tree*) ist entweder leer, oder er repräsentiert eine Menge von *Knoten* (*nodes*), gleichen Datentyps in der es genau einen ausgezeichneten Knoten, die *Wurzel* (*root*), und eine endliche Menge disjunkter Bäume, die so genannten *Teilbäume* (*subtrees*) gibt, die mit der Wurzel verbunden sind.

Aus dieser Definition und ihrem Vergleich mit der Definition für verkettete Listen in Abschnitt 3.4.2 geht hervor, dass Bäume den verketteten Listen ähnlich sind: Wie Listen bestehen auch Bäume aus Knoten und wie bei Listen hat auch bei Bäumen jeder Knoten höchstens einen *Vorgänger* (*predecessor*). Während bei Listen jeder Knoten höchstens einen *Nachfolger* (*successor*) hat, kann jeder Baumknoten aber beliebig viele Nachfolger aufweisen.

▶Abbildung 3.23 zeigt die Struktur eines Baums mit wichtigen Eigenschaften und weiteren Begriffen. Die Knoten eines Baums werden nach ihrer Position im Baum Wurzel (wie schon in der Definition erwähnt), *Blätter* (*leaves*), das sind Knoten ohne Nachfolger, oder *innere Knoten* (*inner nodes*) genannt. Jeder Knoten ist einer bestimmten *Ebene* (*level*) im Baum zugeordnet, die Wurzel der Ebene 0. Die *Höhe* (*height*) eines Baums wird bestimmt durch die Länge des Wegs, d.h. durch die Anzahl der Verbindungen von der Wurzel bis zum Blatt auf der höchsten Ebene.

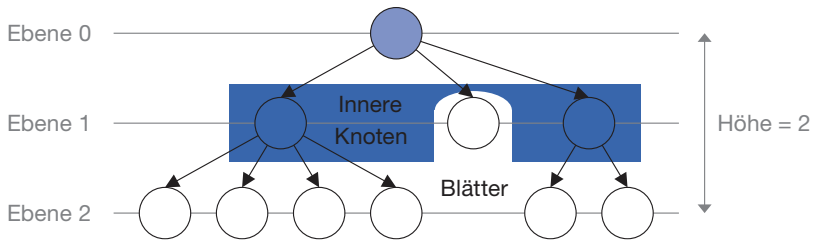

Abbildung 3.23: Baumstruktur und zugeordnete Begriffe

Wie in ▶Abbildung 3.24 an einem Beispiel gezeigt, lässt sich jeder allgemeine Baum, auch Mehrwegbaum genannt, dargestellt in (a), unabhängig davon, wie viele Nachfolger seine Knoten haben, durch einen äquivalenten Baum so repräsentieren, dass jeder Knoten höchstens zwei Nachfolger hat. Solche Bäume werden Binärbäume genannt, siehe nächster Abschnitt. Abbildung 3.24 (b) und (c) zeigen zwei unterschiedliche Darstellungsweisen des aus dem Mehrwegbaum von (a) abgeleiteten Binärbaums. Die Repräsentation eines allgemeinen Baums in Form eines Binärbaums heißt ***kanonische Baumdarstellung***.

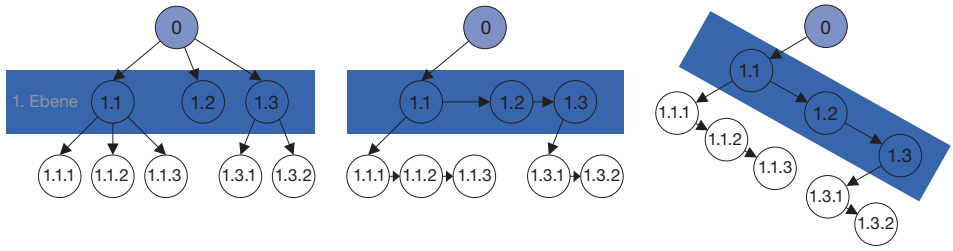

Abbildung 3.24: Repräsentation eines allgemeinen Baums in (a) als Binärbaum in (b) und (c)

Da sich jeder Mehrwegbaum in einen äquivalenten Binärbaum transformieren lässt, mit dem Vorteil, dass die Knoten des Binärbaums alle eine einheitliche Struktur aufweisen, ist es ausreichend, dass wir uns im Folgenden nur noch mit dieser speziellen Baumform beschäftigen.

3.5.2 Binärbäume

Den Begriff Binärbaum können wir wie folgt definieren.

Definition: *Binärbaum*

Ein *Binärbaum*er (*binary tree*) ist ein spezieller Baum, in dem jeder Knoten maximal zwei Nachfolger hat.

Es ist üblich, die beiden Nachfolger als *linken* (*left*) und *rechten* (*right*) Nachfolger bzw. die in diesen Knoten wurzelnden Teile des Binärbaums als linken und rechten *Teilbaum* (*subtree*) zu bezeichnen.

Je nach Anzahl und Position der Knoten im Binärbaum unterscheidet man verschiedene Arten, von denen nur die beiden wichtigsten erwähnt sind:

1. Ein *vollständiger Binärbaum* ist ein Baum, in dem jeder Knoten entweder keinen oder genau zwei Nachfolger hat.
2. Ein *perfekter Binärbaum* ist ein vollständiger Binärbaum, in dem alle Blätter auf ein und derselben Ebene liegen. Die Anzahl der Knoten eines perfekten Binärbaums beträgt $2^k - 1$ für k = Höhe + 1.

Aufgrund der Rekursion in der Baumdefinition, ist es naheliegend – analog zu verketteten Listen – einen rekursiven Datentyp *TreePtr* zur Deklaration von Baumobjekten einzuführen:

```
type TreePtr = →compound     -- pointer to root of tree
       left, right: TreePtr  -- pointers to left and right subtrees
       …  -- any data
     end  -- compound
```

Wie bei verketteten Listen wollen wir aber auch für das Arbeiten mit Binärbäumen wieder zwei Datentypen verwenden: einen Datentyp *TreeNode*, der den Aufbau der einzelnen Knoten festlegt (mit dem dazugehörigen Zeigerdatentyp *TreeNodePtr*) und den Datentyp *TreePtr* zur Referenzierung, d.h. zum Zugriff auf den „gesamten" Baum mit seinen Knoten. Und wie bei den Listen verwenden wir als Datenkomponente „nur" einen *integer*-Wert, weil das für unsere Demonstrationszwecke ausreichend ist:

```
type TreeNodePtr = →TreeNode
     TreeNode = compound
       left, right: TreeNodePtr
       data: int
     end -- compound
     TreePtr = TreeNodePtr  -- synonym for TreeNodePtr,
                            -- TreePtr is pointer to root node

var  tree: TreePtr
```

Anmerkung: Auch hier wäre es durchaus wieder überlegenswert (wie schon bei den verketteten Listen), die Typnamen so zu wählen, dass aus ihnen hervorgeht, dass wir es „nur" mit einem Binärbaum zu tun haben, also *BinaryTreeNode*(*Ptr*) und *BinaryTreePtr*. Da jeder allgemeine Baum (mittels kanonischer Baumdarstellung) auch als Binärbaum repräsentiert werden kann, verwenden wir die allgemeineren Typnamen *Tree*(*Node*)(*Ptr*).

▶Abbildung 3.25 zeigt exemplarisch einen perfekten Binärbaum mit seinen Knoten, die im dynamischen Speicherbereich angelegt wurden.

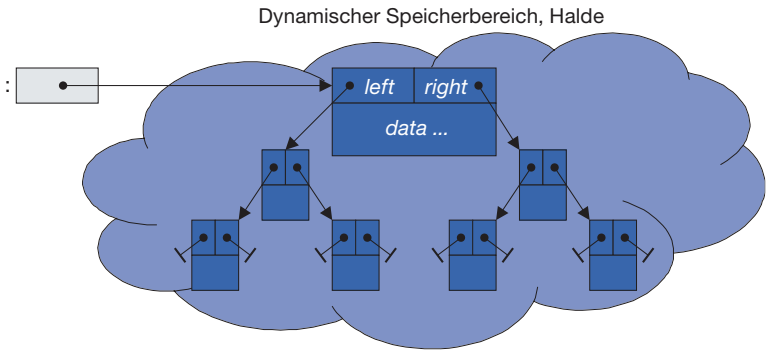

Abbildung 3.25: Perfekter Binärbaum im dynamischen Speicherbereich

Analog zu verketteten Listen sind auch für Binärbäume die Algorithmen *EmptyTree* zur Initialisierung einer Baumvariablen und *NewNode* zur Erzeugung eines neuen Baumknotens notwendig. Wir verzichten auf eine detaillierte Darstellung dieser Algorithmen für Binärbäume, da sie sich von den oben angeführten ähnlich oder gleich benannten Algorithmen für Listen im Wesentlichen nur durch die Datentypen, nicht aber im Prinzip unterscheiden.

Aus einem Wurzelknoten *n* und zwei Teilbäumen (bzw. Knoten) *l* und *r* kann mit dem unten angegebenen Algorithmus *BinTreeOf* ein Binärbaum gebildet werden. Der Algorithmus enthält als Kommentare die *Zusicherung* (*assertion*), die erfüllt sein muss, um den Binärbaum korrekt bilden zu können:

```
BinTreeOf(↓n: TreeNodePtr  ↓l: TreePtr  ↓r: TreePtr): TreePtr
begin
  -- assertion: (n ≠ null) and (n→left = null) and (n→rigth = null) and
  --            (n ≠ l) and (n ≠ r) and
  --            ( (l = null) or (r = null) or (l ≠ r) )
  n→left := l
  n→right := r
  return n
end BinTreeOf
```

Mit dem Algorithmus *BinTreeOf* können beliebig komplexe Bäume *von unten nach oben* (*bottom-up*) also von den Blättern ausgehend, konstruiert werden. Beispiel 3.18 führt vor, wie ein einfacher Binärbaum mittels *BinTreeOf* aufgebaut werden kann.

Beispiel 3.18 **Aufbau eines einfachen Binärbaums**

Ein Binärbaum mit dem Wert 1 im Wurzelknoten und den Werten 2 im linken bzw. 3 im rechten Nachfolger der Wurzel kann auf folgende Art und Weise aufgebaut und der Baumvariablen *tree* zugewiesen werden:

```
  var l, r: TreePtr
      n: TreeNodePtr
begin
  l := BinTreeOf(↓NewNode(↓2)  ↓null  ↓null)  -- left subtree
  r := BinTreeOf(↓NewNode(↓3)  ↓null  ↓null)  -- right subtree
  n := NewNode(↓1)  -- root node
  tree := BinTreeOf(↓n  ↓l  ↓r)
```

Nachdem auch ein einzelner Knoten ein Baum ist (siehe Definition), kann derselbe Baum ohne Hilfsvariable und kürzer wie folgt erzeugt werden:

```
tree := BinTreeOf(
          ↓NewNode(↓1)  -- root
          ↓NewNode(↓2)  -- left subtree
          ↓NewNode(↓3)  -- right subtree
        ) -- BinTreeOf
```

3.5.3 Binäre Suchbäume

Bevor wir uns mit den Eigenschaften und der Verwendung von binären Suchbäumen beschäftigen, müssen wir definieren, was unter diesem Begriff zu verstehen ist.

Definition: *Binärer Suchbaum*

Ein *binärer Suchbaum* (*binary search tree*:) ist ein Binärbaum, dessen Knoten so angeordnet sind, dass, wenn *data* die Datenkomponente eines betrachteten Knotens repräsentiert, die Werte der Datenkomponenten aller linken Nachfolger des betrachteten Knotens kleiner und die aller rechten Nachfolger größer oder gleich dem Wert von *data* sind.

▶Abbildung 3.26 zeigt die oben festgelegte Eigenschaft eines binären Suchbaums aus der Sicht der Wurzel: Die Wurzel enthält einen Wert, der größer ist als alle Werte, die in den Knoten des linken Teilbaums vorkommen, und der Wert in der Wurzel ist kleiner oder gleich (falls Werte mehrfach vorkommen) allen Werten, die in den Knoten des rechten Teilbaums vorkommen. Wenn diese Eigenschaft nicht nur auf die Wurzel des Baums, sondern auch auf die Wurzeln aller Teilbäume zutrifft, dann ist der Baum ein binärer Suchbaum. Diese Eigenschaft kann – wie unten gezeigt – zur effizienten Suche und zum raschen Einfügen eines bestimmten Knotens genutzt werden.

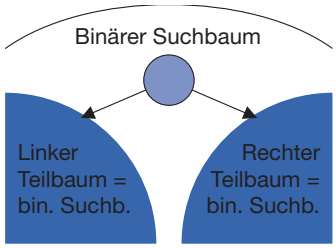

Abbildung 3.26: Binärer Suchbaum

Die Ausprägung eines binären Suchbaums hängt davon ab, in welcher Reihenfolge die Knoten bei der Bildung des Baums eingefügt werden. Die Anzahl der möglichen Ausprägungen eines binären Suchbaums für n darin zu speichernde Werte (und somit Knoten) ist durch die *Catalansche* Zahl $C(n)$, benannt nach dem belgischen Mathematiker Eugène C. Catalan, bestimmt:

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

Beispiel 3.19 zeigt drei binäre Suchbäume für die ersten vier Primzahlen.

Beispiel 3.19 **Binäre Suchbäume für die ersten vier Primzahlen**

Für die ersten vier Primzahlen gibt es $C(4) = 14$ mögliche binäre Suchbäume. ▶Abbildung 3.27 zeigt drei davon, wobei auffällt, dass die in (c) dargestellte Ausprägung einer einfach-verketteten Liste entspricht, d.h. ein so genannter *degenerierter* Suchbaum ist, wodurch die Eigenschaft des effizienten Auffindens eines bestimmten Knotens verloren gegangen ist.

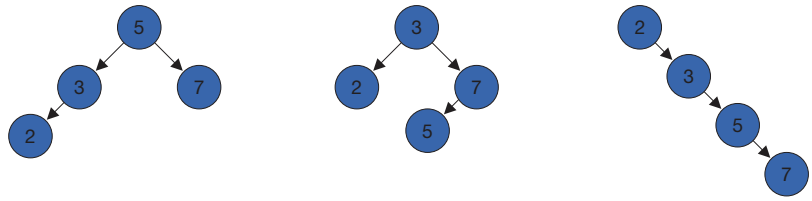

Abbildung 3.27: Auswahl möglicher Anordnungen der ersten vier Primzahlen in binären Suchbäumen

Der Vorteil, der sich aus der speziellen Knotenanordnung ergibt, wird offensichtlich, wenn man den Algorithmus *NodeWith* zur Suche eines bestimmten Werts *x* in der *data*-Komponente eines Knotens des binären Suchbaums *tree* entwickelt: Ausgehend von der Wurzel des Baums wird in einer *while*-Schleife solange in den linken oder rechten Teilbäumen abgestiegen, bis der gesuchte Wert gefunden oder kein Teilbaum mehr vorhanden ist. Wenn der Suchbaum einigermaßen ausbalanciert ist, findet man den gesuchten Wert, indem nur wenige Knoten besucht werden müssen (vgl. *binäre Suche* in Kapitel 6).

```
NodeWith(↓tree: TreePtr ↓x: int): TreeNodePtr
  var n: TreeNodePtr
begin
  n := tree
  while (n ≠ null) and (n→data ≠ x) do
    if x < n→data then
      n := n→left
    else  -- x ≥ n→data
      n := n→right
    end -- if
  end -- while
  return n
end NodeWith
```

Da wir den Begriff des binären Suchbaums rekursiv definiert haben – also zur Begriffsdefinition der Begriff selbst herangezogen wurde – wäre es naheliegend, auch einen Suchalgorithmus zu entwickeln, der sich selbst aufruft, also rekursiv ist. Wir führen daher im nächsten Kapitel das Konzept der rekursiven Algorithmen ein und geben dann auch eine rekursive Variante des Algorithmus *NodeWith* an.

Während bei der Suche eines bestimmten Knotens in einer verketteten Liste ein Knoten nach dem anderen untersucht werden muss, wird bei einem (perfekten) binären Suchbaum mit jedem Suchschritt (entspricht einem Schleifendurchlauf) im Suchalgorithmus *NodeWith* die Anzahl der noch zu untersuchenden Knoten auf die Hälfte reduziert. Deshalb erfordert die Suche nach einem bestimmten Knoten im schlimmsten Fall (d.h. wenn der gesuchte Knoten in der Datenstruktur nicht enthalten ist) in einer verketteten Liste mit n Knoten n Suchschritte, in einem (perfekten) binären Suchbaum mit $n = 2^k - 1$ Knoten jedoch nur k Suchschritte, also weniger als $ld\ n$ Schleifendurchläufe. Das bedeutet, dass sich bei Verwendung von verketteten Listen die Anzahl der erforderlichen Suchschritte verdoppelt, wenn sich die Listenlänge verdoppelt (d.h., dass die Laufzeit linear mit der Listenlänge steigt, vgl. dazu Kapitel 5), während sich bei Verwendung von (perfekten) binären Suchbäumen die erforderliche Anzahl von Suchschritten bei einer Verdoppelung der Knotenanzahl nur um einen weiteren Suchschritt erhöht (d.h., dass die Laufzeit nur mit dem Logarithmus der Knotenanzahl, also logarithmisch und damit deutlich weniger als linear ansteigt).

Eine wesentliche Eigenschaft vernetzter Datenstrukturen wie beispielsweise binärer Suchbäume besteht darin, dass sie wachsen und schrumpfen können. Es gilt also, neue Knoten an den richtigen Stellen einzufügen, bestimmte Knoten zu suchen und gegebenenfalls sie auch wieder zu entfernen.

Wir wollen zunächst, analog zum *SortedInsert*-Algorithmus für verkettete Listen, einen Algorithmus *Insert* (hier können wir das Attribut *Sorted* im Algorithmennamen weglassen, denn bei einem binären Suchbaum handelt es sich per definitionem um eine sortierte Datenstruktur) entwickeln, der in einem binären Suchbaum *tree* einen Knoten n an der richtigen Stelle (als neues Blatt) so in den Baum einfügt, dass die Suchbaum-Eigenschaft erhalten bleibt. Dazu muss zuerst die Einfügeposition für den neuen Knoten gesucht werden, z.B. analog zum oben angeführten Algorithmus *NodeWith*, wobei wir uns beim Absteigen in die Teilbäume zusätzlich den jeweiligen Vorgängerknoten (*predecessor*, kurz *pred*) merken, so dass der Knoten n als neues Blatt unter dem letzten Vorgänger (*pred*) eingefügt werden kann.

```
Insert(↕tree: TreePtr  ↓n: TreeNodePtr)
  var pred: TreeNodePtr  -- predecessor of …
      st:  TreePtr       -- … current subtree
begin
  if tree = null then  -- empty tree
    tree := n
  else  -- tree is not epmty
    pred := null
    st := tree
    while st ≠ null do
      pred := st
      if n→data < st→data then
        st := st→left
      else  -- n→data ≥ st→data
        st := st →right
      end -- if
    end -- while
```

```
    -- pred is predecessor of node n
    if n→data < pred→data then  -- insert n as left son
      pred→left := n
    else  -- n→data ≥ pred→data, so insert n as right son
      pred→right := n
    end -- if
  end -- if
end Insert
```

Mit wiederholten Aufrufen des *Insert*-Algorithmus können wir einen binären Suchbaum von der Wurzel ausgehend zu den Blättern hin, also von oben nach unten (*top-down*) aufbauen. Da wir den Begriff des binären Suchbaumes rekursiv definiert haben – also zur Begriffsdefinition der Begriff selbst herangezogen wurde – wäre es naheliegend, dies auch bei der Entwicklung des Algorithmus *Insert* auszunutzen und eine rekursive Lösung dafür anzugeben, die dann wesentlich einfacher sein müsste als die hier angegebene iterative Lösung. Wir führen daher im folgenden Kapitel das Konzept der rekursiven Algorithmen ein und geben dann auch eine rekursive Variante des Algorithmus *Insert* an.

Beispiel 3.20 zeigt, wie man mittels *Insert* verschiedene binäre Suchbäume aufbauen kann.

Werden beim Aufbau des binären Suchbaums die Knoten in einer Reihenfolge, die der Sortierung des Inhalts ihrer Datenkomponente *data* entspricht, eingefügt, entsteht zwar ein binärer Suchbaum, allerdings einer, bei dem jeder Knoten nur einen rechten (bei aufsteigender Sortierung) bzw. nur einen linken (bei absteigender Sortierung) Nachfolger hat, siehe ▶Abbildung 3.27 (c). Solche Bäume werden – wie bereits erwähnt – als *degenerierte Bäume* bezeichnet, da sie letztendlich eine einfach-verkettete Liste bilden.

Beispiel 3.20 **Konstruktion zweier binärer Suchbäume**

Die beiden in Abbildung 3.27 (a) und (b) dargestellten binären Suchbäume können durch Verwendung des Algorithmus *Insert,* wie in Tabelle 3.4 gezeigt, gebildet werden.

Tabelle 3.4

Konstruktion zweier binärer Suchbäume für die ersten vier Primzahlen

```
-- build binary search tree
-- for (a)
tree := EmptyTree()
Insert(↕tree ↓NewNode(5))
Insert(↕tree ↓NewNode(3))
Insert(↕tree ↓NewNode(2))
Insert(↕tree ↓NewNode(7))
```

```
-- build binary search tree
-- for (b)
tree := EmptyTree()
Insert(↕tree ↓NewNode(3))
Insert(↕tree ↓NewNode(2))
Insert(↕tree ↓NewNode(7))
Insert(↕tree ↓NewNode(5))
```

Zwar können alle oben vorgestellten Baumalgorithmen auch bei degenerierten Bäumen verwendet werden, aber der Effizienzvorteil binärer Suchbäume geht durch die Degeneration verloren. Deshalb empfiehlt es sich, beim Aufbau eines binären Suchbaums darauf zu achten, dass Degenerationen vermieden werden. Dafür gibt es im Wesentlichen zwei Möglichkeiten:

1. Man muss die Knoten mit dem Algorithmus *Insert* in zufälliger Reihenfolge einfügen (keine auf- oder absteigenden *data*-Werte). Dabei entsteht ein Baum, dessen Höhe nur um einen konstanten Faktor von der optimalen Höhe – bei n Knoten ist das $ld\ n$ – abweicht.

2. Man entwickelt einen speziellen *Insert*-Algorithmus, der bei einer ungünstigen Einfügereihenfolge die Knoten des Baums automatisch so umordnet, dass Degeneration vermieden wird. Dieses Vorgehen nennt man *Balancieren* und die dadurch entstehenden Bäume heißen *balancierte* oder *ausgeglichene binäre Suchbäume*, die nach ihren Erfindern Georgii M. Adelson-Velsky und Yevgeniy M. Landis auch *AVL-Bäume* genannt werden (siehe dazu z.B. [Adelson-Velsky u. Landis 1962]). Bei AVL-Bäumen unterscheiden sich die Höhen der Teilbäume eines jeden Knotens höchstens um den Wert 1.

Neben den AVL-Bäumen werden auch balancierte (nicht mehr notwendigerweise binäre) Suchbäume verwendet, deren Knoten zwei oder drei (so genannte 2-3-Bäume) bzw. zwei, drei oder vier Teilbäume (so genannte 2-3-4-Bäume) referenzieren können. Dadurch wird zwar zusätzliche Flexibilität beim Balancieren gewonnen, allerdings wird Speicherplatz verschwendet, wenn z.B. ein 4-Baum-Knoten keinen oder nur einen Teilbaum hat.

Die effizienteste Form balancierter binärer Suchbäume – sowohl hinsichtlich Speicherplatz als auch Laufzeit – basiert auf solchen 2-3-4-Bäumen: die so genannten *Rot-Schwarz-Bäume* (*red-black trees*), die erstmals 1972 von Rudolf Bayer unter der engl. Bezeichnung *symmetric binary B-trees* eingeführt und 1978 von Leonidas J. Guibas und Robert Sedgewick (siehe [Sedgewick 1988]) ausführlich beschrieben wurden. Sie erfordern einen erweiterten Datentyp *TreeNode*, da eine Komponente für die „Einfärbung“ der Knoten und eine Komponente für den Verweis auf den Vorgänger des Knotens erforderlich sind. Wir gehen hier auf die algorithmischen Details dieser Algorithmen aus Platzgründen nicht weiter ein und verweisen den interessierten Leser dazu auf die Literatur, beispielsweise auf [Sedgewick 2002].

3.6 Datenkapselung und abstrakte Datenstrukturen

Die algorithmische Lösung umfangreicherer Aufgabenstellungen umfasst in der Regel mehrere Algorithmen, die zusammen ein ***Algorithmensystem*** bilden. Ein typisches Beispiel finden wir in Abschnitt 2.7 als Ergebnis des schrittweisen Verfeinerungsprozesses bei systematischer Algorithmenentwicklung.

Die Algorithmen eines solchen Algorithmensystems kommunizieren miteinander über die in ihren Schnittstellen festgelegten Parameter. Oft tritt der Fall ein, dass bestimmte Datenobjekte von zwei oder mehreren Algorithmen eines Systems verwendet und/oder manipuliert werden. Dann müssen sie über den Parametermechanismus zwischen diesen Algorithmen hin- und hergereicht werden. Dies bläht die Schnittstellen der betroffenen Algorithmen auf und vermindert u. U. die Effizienz des Gesamtsystems, weil der Austausch dieser Datenobjekte einen Aufwand erfordert, der sich z.B. negativ auf die Laufzeit auswirkt.

Wie in ▶Abbildung 3.28 schematisch dargestellt, wäre es denkbar, dass solche Datenobjekte aus den Algorithmen herausgelöst und diesen in Form so genannter *gemeinsamer* oder *globaler Variablen* (*common* oder *global variables*) zur Verfügung gestellt werden, so dass alle Algorithmen direkt auf sie zugreifen können.

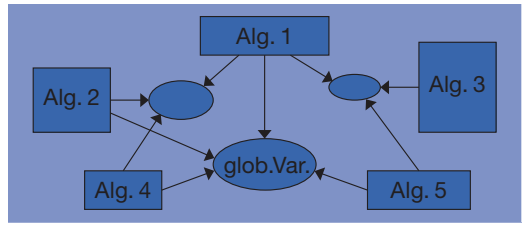

Abbildung 3.28: Algorithmen und globale Variablen

Die Verwendung globaler Datenobjekte ist jedoch mit negativen Konsequenzen verbunden, von denen hier nur die gravierendsten genannt werden:

- Gefahr der Inkonsistenz: Wenn mehrere globale Datenobjekte logisch zusammenhängen (z.B. ein Feld *a*[1:*max*] und die aktuelle Anzahl seiner Elemente *n*), müssen diese Datenobjekte konsistent gehalten werden (z.B. erfordert der Eintrag eines zusätzlichen Werts in das Feld *a*, dass auch der Wert des Datenobjekts *n* verändert, d.h. die Anzahl seiner Elemente um 1 erhöht wird).
- Undurchsichtiger Datenfluss: Nachdem alle Algorithmen auf alle globalen Datenobjekte lesend und schreibend zugreifen können, ist nur schwer nachzuvollziehen, wann und von welchem Algorithmus der Inhalt der globalen Datenobjekte verwendet oder verändert wurde. Deshalb sind Algorithmensysteme mit globalen Variablen oft schwer zu verstehen und nur mit großem Aufwand zu testen.
- Hoher Änderungsaufwand: Wenn die Repräsentation eines der globalen Datenobjekte (ihre Datenstrukturen) verändert werden soll (z.B. wenn sich herausstellt, dass statt eines Felds eine verkettete Liste besser geeignet wäre), sind Änderungen an all jenen Algorithmen erforderlich, die auf dieses globale Datenobjekt zugreifen.

Um solche negativen Konsequenzen hintanzuhalten, hat David L. Parnas in [Parnas 1972] vorgeschlagen, auf die Verwendung globaler Datenobjekte zu verzichten und statt dessen diese Datenobjekte in so genannte ***Datenkapseln*** (*data capsules*) zu verpacken, um sie so vor ihrer Umgebung, also den Algorithmen, von denen bisher auf sie direkt zugegriffen werden konnte, geheim zu halten. Dieses Konzept heißt daher ***Geheimnisprinzip*** (*principle of information hiding*).

Den Algorithmen ist der direkte Zugriff auf die gekapselten Daten verwehrt, auf sie kann nur mehr indirekt über so genannte *Zugriffsoperationen* (*access routines*) zugegriffen werden. Nur die Zugriffsoperationen „kennen“ den genauen Aufbau der gekapselten Datenobjekte. Eine missbräuchliche Verwendung der Datenobjekte ist damit ausgeschlossen und wenn ihre Struktur verändert werden soll, schlägt sich dies nicht in den sie verwendenden Algorithmen, sondern nur in den Zugriffsoperationen (die zusammen mit den Datenobjekten eine Einheit, die Datenkapsel bilden) nieder.

In Abschnitt 2.7 wurde anhand des Prinzips der schrittweisen Verfeinerung gezeigt, wie für komplexe Aufgabenstellungen systematisch algorithmische Lösungen entwickelt werden können, indem ausgehend von einer grundsätzlichen, d.h. abstrakten Lösung der Gesamtaufgabe durch fortgesetzte Verfeinerungen die bei der Lösung der Gesamtaufgabe vorausgesetzten (d.h. abstrahierten) Lösungen der Teilaufgaben immer mehr konkretisiert werden, bis schließlich für alle Teilaufgaben eine algorithmische Lösung vorliegt. Eine ähnliche Vorgehensweise empfiehlt sich auch bei der Modellierung der Datenobjekte: Zuerst wird nur festgelegt, welche Operationen auf den Datenobjekten möglich sein sollen, d.h. von ihrem konkreten Aufbau wird abstrahiert. Daher wird dieser Ansatz als *Datenabstraktion* (*data abstraction*) bezeichnet. Die Entscheidung darüber, welche die beste konkrete Repräsentation dieser Datenobjekte ist, wird erst dann getroffen, wenn alle anderen Details gelöst sind.

Definition: *Abstrakte Datenstruktur (ADS)*

Eine *abstrakte Datenstruktur* (*abstract data structure*), kurz ADS, ist ein algorithmischer Baustein, der über eine spezielle Schnittstelle eine Menge von (abstrakten) Zugriffsoperationen zur Verfügung stellt, mit denen eine Menge gekapselter, d.h. abstrakter Datenobjekte (deren konkrete Realisierung dem Benutzer einer Zugriffsoperation verborgen bleibt) einzeln oder als Ganzes manipuliert werden kann.

▶Abbildung 3.29 zeigt schematisch das Konzept der abstrakten Datenstruktur: Eine Sammlung von Datenobjekten (die vor der Außenwelt verborgen sind) und von Zugriffsoperationen (die in Form von Algorithmen realisiert sind, deren Schnittstelle der Außenwelt bekannt gemacht wird und deren Implementierung vor der Außenwelt verborgen bleibt). In Kapitel 11 werden wir zeigen, dass *Module* (*modules*) adäquate Möglichkeiten bieten, abstrakte Datenstrukturen zu implementieren.

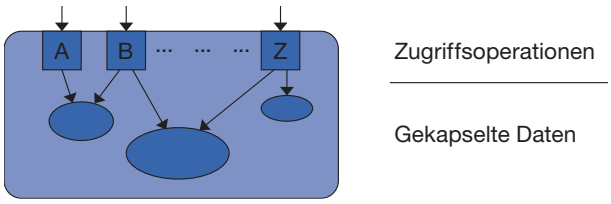

Abbildung 3.29: Konzept der abstrakten Datenstruktur (ADS)

3.6.1 Kellerspeicher (*stack*) als abstrakte Datenstruktur

Zur Illustration der Bildung einer Datenkapsel betrachten wir einen *Kellerspeicher* oder *Stapel* (*stack*). Wie in ▶Abbildung 3.30 dargestellt, repräsentiert ein Kellerspeicher ein *Behälter*-Datenobjekt (*container*), in das andere Datenobjekte nur von „oben“ eingefügt (eingekellert) und nur von „oben“ wieder entnommen (ausgekellert) werden können, d.h. das jeweils zuletzt eingefügte Datenobjekt wird als erstes wieder entnommen (*last in / first out*, kurz *LIFO*).

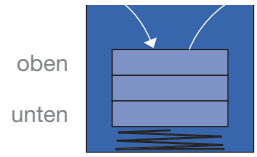

Abbildung 3.30: Konzept des Kellerspeichers

Einen Kellerspeicher für *integer*-Werte können wir beispielweise durch eine abstrakte Datenstruktur realisieren. Dazu ist es notwendig, die für die Verwaltung des Kellerspeichers erforderlichen Zugriffsoperationen mit geeigneten Parametern in der Schnittstelle für die abstrakte Datenstruktur, wir nennen sie *IntStackADS*, festzulegen. Für das Einkellern eines Datenobjekts (in unserem Falle eines *integer*-Werts) sehen wir beispielsweise eine Operation *Push* und für das Auskellern eine Operation *Pop* vor, und um feststellen zu können, ob der Kellerspeicher leer ist, die Operation *IsEmpty*. Im Sinne der Abstraktion und des Geheimnisprinzips trennen wir (textuell) die Schnittstellenbeschreibung von der Implementierungsbeschreibung.

Die Schnittstelle (*interface*) von *IntStackADS* legt fest, welche Operationen auf der abstrakten Datenstruktur möglich und erlaubt sind und sieht dementsprechend folgendermaßen aus:

```
interface of IntStackADS
  Push(↓e: int)
  Pop(↑e: int)
  IsEmpty(): bool
end IntStackADS
```

Wenn in einem Algorithmus die Schnittstelle *IntStackADS* bekannt ist, dann kann dieser mit einem Kellerspeicher arbeiten, ohne dass er wissen muss, wie die konkrete Datenstruktur, welche diesen Kellerspeicher realisiert, aufgebaut ist (gemäß Geheimnisprinzip). Das folgende Codestück zeigt, wie mittels *Push* die Werte 1, 2 und 3 eingekellert und mittels *Pop* wieder ausgekellert werden. Entsprechend dem Kellerspeicher-Prinzip (*last in / first out*) werden in der *while*-Schleife mittels *Pop* die Werte in der umgekehrten Reihenfolge der Einkellerung wieder ausgekellert und demnach wird durch die *Write*-Operation in der *while*-Schleife 3, 2 und 1 ausgegeben.

```
  var e: int
begin
  Push(↓1)
  Push(↓2)
```

```
Push(↓3)
while not IsEmpty() do
  Pop(↑e)
  Write(↓e)
end -- while
```

Bezüglich der konkreten Implementierung der abstrakten Datenstruktur ist der Entwickler von *IntStackADS* frei; er könnte dazu ein Feld, eine verkettete Liste oder sonst etwas heranziehen. Wir wollen – um die Größe des Kellerspeichers nicht unnötig zu beschränken – die Datenobjekte wie in ▶Abbildung 3.31 dargestellt in einer einfach-verketteten Liste mit dem Namen *stack* verwalten.

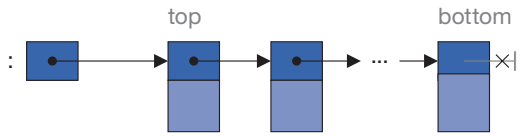

Abbildung 3.31: Kellerspeicher in Form einer einfach-verketteten Liste *stack*

Zur Implementierung (*implementation*) der abstrakten Datenstruktur *IntStackADS* verwenden wir eine Zeigervariable *stack* vom Datentyp *ListPtr*, die als leere Liste initialisiert wird. Der Implementierungsteil von *IntStackADS* sieht dann beispielweise folgendermaßen aus:

```
implementation of IntStackADS

  type ListNodePtr = …  -- see details for both types
       ListPtr = …      -- in section 3.4.2

  var stack: ListPtr

  NewNode(↓data: int): ListNodePtr …  -- see implementation of both
  EmptyList(): ListPtr …              -- algorithms in section 3.4.2

  Push(↓e: int)
    var n: ListNodePtr
  begin  -- construct new node n and prepend it to list
    n := NewNode(↓e)
    n→next := stack
    stack := n
  end Push

  Pop(↑e: int)
    var n: ListNodePtr
  begin
    if IsEmpty() then
      Write(↓"Error: stack is empty")
      halt
    else  -- stack not empty
      n := stack
      e := n→data
      stack := n→next
      Dispose(↓n)
    end -- if
  end Pop
```

```
  IsEmpty(): bool
  begin
    return stack = EmptyList()
  end IsEmpty

init
  stack := EmptyList()
end StackADS
```

Sollte sich bei der Verwendung der abstrakten Datenstruktur *IntStackADS* herausstellen, dass die maximale Anzahl der Elemente gering und die Effizienz durch das häufige Anlegen/Freigeben der Listenknoten unbefriedigend ist, kann die Implementierung jederzeit von der verketteten Liste auf ein Feld umgestellt werden, ohne dass in den Algorithmen, in denen die abstrakte Datenstruktur verwendet wird, eine Änderung vorgenommen werden muss.

3.7 Abstrakte Datentypen

Oft ist es wünschenswert oder gar notwendig, dass in einem Algorithmus mehrere Exemplare einer abstrakten Datenstruktur zur Verfügung stehen. Um dies zu ermöglichen, ist es naheliegend, das Konzept der abstrakten Datenstruktur dahingehend zu erweitern.

Definition: *Abstrakter Datentyp (ADT)*

Ein *abstrakter Datentyp* (*abstract data type*), kurz ADT, definiert eine Menge von Datenobjekten, die alle dieselbe abstrakte Datenstruktur haben und auf die nur mittels (abstrakter) Zugriffsoperationen zugegriffen werden kann.

Wie in ▶Abbildung 3.32 dargestellt, legen wir fest, dass für abstrakte Datentypen die Schnittstelle der dazugehörenden abstrakten Datenstruktur um zwei (Standard-) Operationen erweitert wird. Eine dieser beiden Operationen (in der Abbildung mit * bezeichnet) dient der Erzeugung eines Exemplars des abstrakten Datentyps und die andere (mit † bezeichnete) Operation dient der Beseitigung eines nicht mehr benötigten Exemplars.

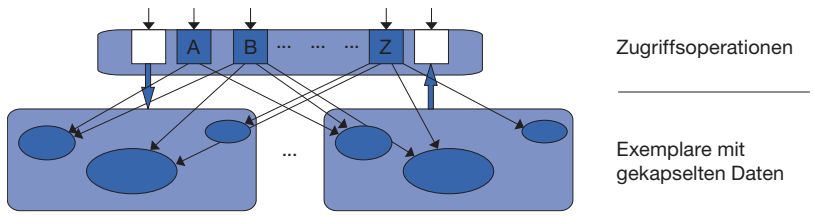

Abbildung 3.32: Konzept des abstrakten Datentyps (ADT)

Wie aus dem Beispiel einer Warteschlange in Form eines abstrakten Datentyps im nächsten Abschnitt zu erkennen ist, erfordert die Realisierung eines abstrakten Datentyps im Vergleich zu der einer abstrakten Datenstruktur nur geringfügig mehr Aufwand, weshalb stets überlegt werden soll, ob es nicht zweckmäßig ist, statt einer abstrakten Datenstruktur von vorneherein einen abstrakten Datentyp zu realisieren.

3.7.1 Warteschlange (*queue*) als abstrakter Datentyp

Zur Illustration der Verwendung abstrakter Datentypen betrachten wir eine Warteschlange. Wie in ▶Abbildung 3.33 dargestellt, repräsentiert eine *Warteschlange* (*queue*) ein *Behälter*-Datenobjekt (*container*), in das nur „hinten“ Datenobjekte eingefügt und nur „vorne“ wieder entfernt werden können; d.h. das erste eingefügte Datenobjekt wird auch als erstes wieder aus dem Behälter-Datenobjekt entfernt (*first in / first out*, kurz *FIFO*).

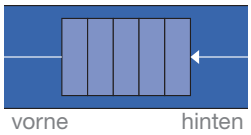

Abbildung 3.33: Konzept der Warteschlange

Wenn es im Zuge der algorithmischen Lösung einer bestimmten Aufgabenstellung erforderlich ist, dass beliebig viele Warteschlangen, deren Datenobjekte *integer*-Werte repräsentieren, verwaltet werden müssen, dann können wir dies durch die Verwendung eines abstrakten Datentyps ermöglichen. Dazu ist es notwendig, die für die Verwaltung der Warteschlangenobjekte erforderlichen Zugriffsoperationen mit geeigneten Parametern, in der Schnittstelle eines abstrakten Datentyps, wir nennen diesen *IntQueueADT*, festzulegen. Für das Einbringen eines Datenobjekts (in unserem Falle eines *integer*-Werts) sehen wir beispielsweise eine Operation *Enqueue* und für das Entfernen eine Operation *Dequeue* vor. Darüber hinaus benötigen wir Operationen zum Erzeugen und Beseitigen von Wartenschlangenobjekten, wir nennen diese Operationen *NewQueue* und *DisposeQueue,* und zur Prüfung, ob ein Warteschlangenobjekt noch „leer“ oder bereits „voll“ ist, sehen wir die Operationen *IsEmpty* bzw. *IsFull* vor.

Da (im Sinne eines abstrakten Datentyps) mehrere Warteschlangenobjekte erzeugt und manipuliert werden können, benötigen alle Zugriffsoperationen einen Parameter zur Angabe des zu manipulierenden Warteschlangenobjekts. Da es sich bei diesem Datenobjekt um eine abstrakte Datenstruktur handelt, deren konkreter Aufbau verborgen bleiben soll, benötigen wir dafür einen entsprechenden Datentyp, wir nennen diesen *IntQueue*. Die Schnittstelle von *IntQueueADT* sieht dementsprechend folgendermaßen aus:

```
interface of IntQueueADT
  type IntQueue  -- only name of type in the interface
  NewQueue(): IntQueue
  Enqueue(↕q: IntQueue  ↓e: int)
```

```
  Dequeue(↕q: IntQueue  ↑e: int)
  IsEmpty(↓q: IntQueue): bool
  IsFull(↓q: IntQueue): bool
  DisposeQueue(↕q: IntQueue)
end IntQueueADT
```

Wenn in einem Algorithmus die Schnittstelle *IntQueueADT* bekannt ist, kann dieser mit Warteschlangenobjekten arbeiten, ohne dass er wissen muss, wie die konkrete Datenstruktur, die diese realisiert, aufgebaut ist (der Typ *IntQueue* ist in der Schnittstelle von *IntQueueADT* nicht näher spezifiziert, gemäß Geheimnisprinzip). Das folgende Codestück zeigt, wie mittels *Enqueue* die Werte 1, 2 und 3 in ein Warteschlangenobjekt *q* eingebracht und danach mittels *Dequeue* wieder entfernt werden, weshalb dann durch die *Write*operation die Werte 1, 2 und 3 (d.h. in der Reihenfolge ihrer Einbringung) ausgegeben werden.

```
  var q: IntQueue
      e: int
begin
  q := NewQueue()
  Enqueue(↕q  ↓1)
  Enqueue(↕q  ↓2)
  Enqueue(↕q  ↓3)
  while not IsEmpty(↓q) do
    Dequeue(↕q  ↑e)
    Write(↓e)
  end -- while
  DisposeQueue(↕q)
```

Bezüglich der konkreten Implementierung des abstrakten Datentyps ist der Entwickler von *IntQueueADT* frei; er könnte beispielsweise eine doppelt-verkettete Liste heranziehen. Wir wollen aber – um eine besonders geschickte Verwendung eines Felds zu zeigen – die Elemente eines Warteschlangenobjekts, wie in ▶Abbildung 3.34 dargestellt, in einem Feld verwalten. Dazu wird für jedes Exemplar des abstrakten Datentyps ein Feld *elements* der Größe *n* mit dem Indexbereich von 0 bis $n - 1$ und zwei Indizes *first* und *free* verwendet. Der Index *first* gibt die Position des vordersten Elements in der Wartschlange an und der Index *free* die Position des nächsten freien Elements im Feld.

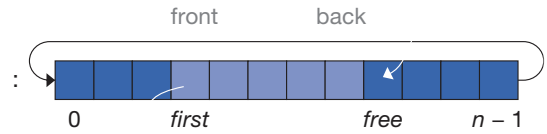

Abbildung 3.34: Warteschlange mittels Ringpuffer

Durch das Einbringen und Entfernen von Elementen wandert der den aktuellen Zustand der Warteschlange repräsentierende zusammenhängende Bereich (in Abbildung 3.34 dunkelblau dargestellt) im Feld nach hinten. Ist das Feld groß genug, wird zu jenem Zeitpunkt, zu dem das Ende der Warteschlange am Ende des Felds anlangt ($free = n - 1$), im Feld vorne wieder Platz frei sein ($first > 0$). Verbindet man das Ende

des Felds „virtuell“ mit dem Anfang des Felds, entsteht ein so genannter ***Ringpuffer*** (*ring* oder *circular buffer*), in dem sich das Warteschlangenobjekt endlos weiterbewegen kann. Damit kann ein Warteschlangenobjekt mit maximal $n - 1$ Elementen im Feld untergebracht werden – ein Feldelement muss frei bleiben, um den Anfang vom Ende des belegten Bereichs unterscheiden zu können.

Eine Implementierung des abstrakten Datentyps *IntQueueADT*, in welcher der in der Schnittstelle nicht näher spezifizierte und daher abstrakte Typ *IntQueue* nun entsprechend der gewählten Realisierungsstrategie definiert werden muss, sieht beispielsweise folgendermaßen aus:

```
implementation of IntQueueADT

  const n = …
  type IntQueue = →compound
         elements: array [0:n - 1] of int
         first, free: int
       end -- compound

  NewQueue(): IntQueue
    var q: IntQueue
  begin
    q := New(↓IntQueue)
    if q = null then …  -- report heap overflow error and terminate
    q→first:= 0
    q→free := 0
    return q
  end NewQueue

  Enqueue(↕q: IntQueue  ↓e: int)
  begin
    if IsFull(↓q) then
      …  -- report queue overflow error and terminate
    else
      q→elements[q→free] := e
      q→free := q→free + 1
      if q→free = n then  -- continue with first array element
        q→free := 0
      end -- if
    end -- if
  end Enqueue

  Dequeue(↕q: IntQueue  ↑e: int)
  begin
    if IsEmpty(↓q) then
      …  -- report queue underflow error and terminate
    else
      e := q→elements[q→first]
      q→first := (q→first + 1) mod n  -- mod operation instead of branch
    end -- if
  end Dequeue

  IsEmpty(↓q: IntQueue): bool
  begin
    return q→first = q→free
  end IsEmpty
```

```
  IsFull(↓q: IntQueue): bool
  begin
    return  (q→first = q→free + 1) or
            ((q→first = 0) and (q→free = n - 1))
  end IsFull

  DisposeQueue(↕q: IntQueue)
  begin
    Dispose(↓q)
    q := null
  end DisposeQueue

end IntQueueADT
```

Sollte sich bei der Verwendung der Exemplare des abstrakten Datentyps *IntQueue* herausstellen, dass die Einschränkung auf eine maximale Länge der Warteschlange zu restriktiv ist, kann die Implementierung jederzeit von dem Feld auf eine doppeltverkettete Liste (ev. mit Ankerknoten) umgestellt werden, ohne dass in den Algorithmen, in denen der abstrakte Datentyp verwendet wird, eine Änderung vorgenommen werden muss.

Zusammenfassung

In diesem Kapitel haben wir uns mit Fragen zur Modellierung von Datenobjekten auseinander gesetzt. Ziel dieses Kapitels war es, das im ersten Kapitel eingeführte Konzept des Datentyps vertiefend zu behandeln, verschiedene Ausprägungen von Datenobjekten nicht nur zu motivieren, sondern auch im Detail anhand von Beispielen zu erläutern und den für das systematische Programmieren wichtigen Schritt der Abstraktion vorzunehmen, nämlich Daten zusammen mit den auf ihnen zulässigen Operationen als logische Einheit zu betrachten.

Basierend auf den im ersten Kapitel eingeführten elementaren Konzepten für Datenobjekte und Datentypen haben wir zuerst noch einmal hervorgehoben, dass wir zwischen unveränderbaren und veränderbaren Datenobjekten, also Konstanten und Variablen, unterscheiden müssen, und haben dann erörtert, was wir unter atomaren (oder elementaren, unstrukturierten) Datenobjekten und Datentypen verstehen. (Abschnitt 3.1)

Danach wurde erörtert, dass man mit atomaren Datenobjekten alleine in der Praxis nicht auskommt, sondern dass es für die Bewältigung komplexerer Aufgabenstellungen oft sinnvoll ist, eine Menge einzelner Datenobjekte zusammenzufassen, weil sie logisch zusammen gehören und daher eine logische und strukturierte Einheit bilden (Abschnitt 3.2). Es wurde herausgearbeitet, dass wir dabei sowohl Sammlungen von Datenobjekten gleichen Typs, die Felder (Abschnitt 3.2.1), als auch solchen unterschiedlichen Typs, die Verbunde (Abschnitt 3.2.2), und Kombinationen aus beiden (Abschnitt 3.2.3) begegnen und gezeigt, wie wir solche Datenobjekte und benutzerdefinierte Datentypen dafür realisieren und mit ihnen arbeiten können.

Den in den ersten beiden Abschnitten diskutierten Datenobjekten ist gemeinsam, dass während ihrer Lebensdauer ihr Wert, nicht aber ihre Struktur verändert werden kann. Für viele Aufgabenstellungen benötigt man jedoch Datenobjekte, die aus Einheiten bestehen, die miteinander in Beziehung stehen, also vernetzt werden und dynamisch wachsen und schrumpfen können. Deshalb haben wir den vernetzten oder dynamischen Datenstrukturen einen eigenen Abschnitt (Abschnitt 3.3) gewidmet. In Abschnitt 3.3.1 haben wir das grundlegende Konzept des Zeiger (datentyp)s eingeführt, danach sind wir auf die für das Wachsen und Schrumpfen von Datenstrukturen wichtige Frage des Allokierens und des Freigebens von Speicherplatz eingegangen. (Abschnitt 3.3.2)

Im Anschluss daran – nachdem die erforderlichen Grundlagen erörtert waren – sind wir auf die wichtigsten Repräsentanten vernetzter oder dynamischer Datenstrukturen, nämlich auf verkettete Listen (Abschnitt 3.4) und Bäume (Abschnitt 3.5) ausführlich eingegangen und haben anhand von Beispielen gezeigt, wie diese wichtigen Datenstrukturen in der Praxis verwendet werden können und was es dabei zu beachten gibt.

Das Mittel zur Meisterung der Komplexität ist die Abstraktion. Das gilt für den Entwurf von Algorithmensystemen gleichermaßen wie für die Modellierung komplexer Datenstrukturen. Deshalb haben wir die beiden letzten Abschnitte dieses Kapitels einer Diskussion der Bildung und Verwendung von abstrakten Datenstrukturen (Abschnitt 3.6) und abstrakten Datentypen (Abschnitt 3.7) gewidmet.

Der Leser sollte nunmehr in der Lage sein, im Zuge der systematischen Entwicklung von Algorithmen auftretende Entscheidungsprobleme im Hinblick auf die Wahl der richtigen Datenstrukturen zu lösen und Datenobjekte in Algorithmen den realen Erfordernissen entsprechend angemessen zu modellieren und zu verwalten.

Aufgaben

Aufgabe 3.1 (Auswertung von Polynomen)
Ein Polynom n-ten Grads ist wie folgt definiert:

$$P(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \ldots + a_1 \cdot x + a_0.$$

Gegeben sind ein Feld $a[0{:}max]$ mit $max \geq 1$, das an den Positionen $0 \,..\, n \leq max$ die Folge reellwertiger Koeffizienten a_i eines Polynoms enthält und eine reelle Zahl x. Gesucht ist ein Funktionsalgorithmus *Polynom*(↓*a*: *RealArray* ↓*n*: *int* ↓*x*: *real*): *real*, der den Wert von $P(x)$ berechnet und als Ergebnis liefert.

Eine erste Fassung dieses Algorithmus soll nach der obigen Definition vorgehen und die Potenzen von x durch Verwendung eines als vorhanden vorauszusetzenden Algorithmus *Power* (↓*x*: *real* ↓*n*: *int*): *real* ermitteln.

Eine zweite Fassung soll das Polynom nach dem *Horner-Schema* (benannt nach dem britischen Mathematiker William George Horner) auswerten:

$$P(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x \ldots + a_1) \cdot x + a_0.$$

Hinweis: Das Horner-Schema ist nicht nur einfacher, weil es keine Potenzen benötigt, es liefert auf Computern genauere Ergebnisse und das noch dazu schneller.

Aufgabe 3.2 (*The Missing Element*)
Gegeben ist ein Feld $a[1{:}n]$ mit $n \geq 1$, das eine Folge ganzer Zahlen aus dem Wertebereich 1 bis $n + 1$ enthält. Bis auf eine Zahl kommen alle Zahlen aus diesem Wertebereich genau einmal vor, z. B. für $n = 4$: (3, 2, 4, 5), es fehlt also die Zahl 1. Gesucht ist Funktionsalgorithmus *MissingElement*(↓*a*: *IntArray* ↓*n*: *int*): *int*, der die fehlende Zahl als Ergebnis liefert.

Hinweis: Man kann diese Aufgabe mit einem Hilfsfeld h mit Elementen vom Datentyp *boolean* lösen, wobei im ersten Schritt alle Elemente von 1 bis $n + 1$ mit *false* initialisiert werden, im zweiten Schritt jedes Element in h, dessen Index in a vorkommt auf *true* gesetzt wird und im dritten Schritt der Index jenes Elements gesucht wird, das noch den Wert *false* enthält. – Es geht aber auch ohne Hilfsfeld und noch dazu viel schneller!

Aufgabe 3.3 (Anagramme)
Zwei Zeichenketten sind Anagramme, wenn sie dieselben Zeichen enthalten (*Beispiele*: "WEIN" und "WIEN", "LAMPE" und "PALME"). Gesucht ist ein Funktionsalgorithmus *Anagram*(↓*s1*: *string* ↓*s2*: *string*): *bool*, der feststellt ob die zwei Zeichenketten *s1* und *s2* Anagramme sind.

Aufgaben

Aufgabe 3.4 (Plateau-Problem)
Gegeben ist ein Feld $a[1{:}n]$ mit $n \geq 1$, das eine Folge von aufsteigend sortierten positiven ganzen Zahlen (Datentyp *integer*) enthält. Ein *Plateau* ist eine Folge von gleichen Werten. Gesucht ist ein Algorithmus *Plateau*($\downarrow a$: *IntArray* $\downarrow n$: *int* $\uparrow p$: *int*), der die Länge p des längsten Plateaus in a berechnet und dazu das Feld a nur einmal durchläuft.

Beispiel:
Für $a = (1, 2, 2, 3, 4, 4, 6, 7, 7, 7, 8, 9, 9)$ und $n = 13$ muss der Algorithmus $p = 3$ liefern.

Aufgabe 3.5 (Exponentielle Glättung)
Gegeben ist eine Folge reeller Zahlen (Datentpy *real*) in einem Feld $a[1{:}n]$ mit $n \geq 1$. Gesucht ist ein Algorithmus *ExpSmooth*($\downarrow a$: *RealArray* $\downarrow n$: *int* $\downarrow s$: *real* $\uparrow aes$: *RealArray*), der für die Werte in a eine exponentielle Glättung (*exponential smoothing*) mit dem Glättungsfaktor $0.1 \leq s \leq 0.3$ vornimmt und das Ergebnis in einem neuen Feld $aes[1{:}n]$ liefert. Die Elemente der geglätteten Zahlenfolge in *aes* sind wie folgt zu berechnen:

$$aes[1] = 0.0$$

$$aes[i] = \sum_{j=0}^{i-2} s \cdot (1-s)^j \cdot a[i-j-1] \quad \text{für } 2 \leq i \leq n.$$

Aufgabe 3.6 (Vergleich von Feldern)
Gegeben sind folgende Deklarationen:

```
const max = …
type IntArray = array [1:max] of int
```

Gesucht ist ein Funktionsalgorithmus *SameElements*($\downarrow a$: *IntArray* $\downarrow na$: *int* $\downarrow b$: *IntArray* $\downarrow nb$: *int*): *bool*, der für die beiden teilweise gefüllten Felder a und b feststellt, ob sie dieselben Elemente enthalten, ob also jedes Element von a auch in b vorkommt und umgekehrt.

Beispiel:
Für die beiden Felder $a = (1, 2, 3)$ mit $na = 3$ und $b = (3, 2, 2, 1)$ mit $nb = 4$ soll *SameElements* das Ergebnis *true* liefern.

Aufgabe 3.7 (Invertieren eines Felds)
Gegeben ist ein Feld $a[1{:}max]$ mit $max \geq 1$, das an den Positionen $1 \,..\, n \leq max$ gültige Elemente eines beliebigen Datentyps T enthält. Gesucht ist Algorithmus *Invert*($\updownarrow a$: *TArray* $\downarrow n$: *int*), der die Reihenfolge der Elemente im Feld umkehrt.

Beispiel:
Für ein Feld mit *integer*-Elementen (T = *integer*) soll aus $a = (1, 2, 2, 3)$ das Feld $a = (3, 2, 2, 1)$ werden.

Aufgaben

Aufgabe 3.8 (Matrizenmultiplikation)
Gesucht ist die Deklaration eines Datentyps *Matrix* zur Repräsentation von 3×3-Matrizen vom Elementtyp *real* sowie ein Algorithmus *MatMult*(↓*a*: *Matrix* ↓*b*: *Matrix* ↑*c*: *Matrix*) zur Multiplikation dieser Matrizen. Matrizenmultiplikation ist wie folgt definiert: Das Element c_{ik} des Matrizenprodukts $C = A \cdot B$ ergibt sich als skalares Produkt $a^i \cdot b_k$ des Zeilenvektors a^i mit dem Spaltenvektor b_k:

$$(c_{ik})_{(m,\,p)} = (a_{ik})_{(m,\,n)} \cdot (b_{ik})_{(n,\,p)}$$

Voraussetzung: Spaltenzahl von A = Zeilenzahl von B.

Aufgabe 3.9 (Entfernen und Einfügen von Elementen in Feldern)
Gegeben ist ein Feld *a*[1:*max*] mit $max \geq 1$, das an den Positionen $1 .. n \leq max$ gültige Elemente eines beliebigen Datentyps *T* enthält. Gesucht sind zwei Algorithmen (vgl. ▶Abbildung 3.14):

1. ein Algorithmus *RemoveElementAt*(↕*a*: *TArray* ↕*n*: *int* ↓*i*: *int*), der das Element an der Stelle *i* (mit $1 \leq i \leq n$) aus dem Feld *a* entfernt, in dem die restlichen Elemente um eine Stelle nach vorne verschoben werden und *n* um 1 vermindert wird.
2. ein Algorithmus *InsertElementAt*(↕*a*: *TArray* ↕*n*: *int* ↓*i*: *int* ↓*e*: *T*), der ein neues Element *e* vom Datentyp *T* an der Stelle *i* (mit $1 \leq i \leq n + 1 < max$) in das Feld *a* einfügt, in dem die restlichen Elemente um eine Stelle nach hinten verschoben werden und *n* um 1 erhöht wird.

Aufgabe 3.10 (Einfach-verkettete Liste: Anzahl der Konten)
Gegeben ist eine einfach-verkettete Liste *list* auf Basis der Deklarationen aus Abschnitt 3.4.2. Gesucht ist ein Funktionsalgorithmus *NrOfNodes*(↓*list*: *ListPtr*): *int*, der die Anzahl der Knoten der Liste als Ergebnis liefert.

Aufgabe 3.11 (Einfach-verkettete Liste: Ausgeben von hinten nach vorne, iterativ)
Gegeben ist eine einfach-verkettete Liste *list* auf Basis der Deklarationen aus Abschnitt 3.4.2. Gesucht ist ein (iterativer) Algorithmus *WriteNodesInReverseOrder*(↓*list*: *ListPtr*), der die Werte der Knoten-Komponente *data* in zur Liste umgekehrter Reihenfolge, d.h. von hinten beginnend nach vorne, ausgibt.

Aufgabe 3.12 (Einfach-verkettete Liste: Invertieren, iterativ)
Gegeben ist eine einfach-verkettete Liste *list* auf Basis der Deklarationen aus Abschnitt 3.4.2. Gesucht ist ein (iterativer) Algorithmus *Invert*(↕*list*: *ListPtr*), der die Reihenfolge der Knoten in der Liste umdreht; d.h. der letzte Knoten in der Ausgangsliste wird der erste Knoten in der Ergebnisliste usw.

Aufgaben

Aufgabe 3.13 (Doppelt-verkettete Liste: Ausgeben von hinten nach vorne, iterativ)
Gegeben ist eine doppelt-verkettete Liste *list* auf Basis der Deklarationen aus Abschnitt 3.4.3. Gesucht ist ein (iterativer) Algorithmus *WriteNodesInReverseOrder* (↓*list*: *ListPtr*), der die Werte der Knoten-Komponente *data* in zur Liste umgekehrter Reihenfolge, d.h. von hinten beginnend nach vorne, ausgibt. Vergleichen Sie Ihre Lösung mit jener für einfach-verkettete Listen (Aufgabe 3.11).

Aufgabe 3.14 (Doppelt-verkettete Liste: Invertieren, iterativ)
Gegeben ist eine doppelt-verkettete Liste *list* auf Basis der Deklarationen aus Abschnitt 3.4.2. Gesucht ist ein (iterativer) Algorithmus *Invert*(↕*list*: *ListPtr*), der die Reihenfolge der Knoten in der Liste umdreht, d.h. der letzte Knoten in der Ausgangsliste wird der erste Knoten in der Ergebnisliste usw. Vergleichen Sie Ihre Lösung mit jener für einfach-verkettete Listen (Aufgabe 3.12).

Aufgabe 3.15 (Binärbaum: Anzahl der Knoten)
Gegeben ist ein Binärbaum *tree* auf Basis der Deklarationen aus Abschnitt 3.5.2. Gesucht sind folgende Funktionsalgorithmen:

- *NrOfNodes*(↓*tree*: *TreePtr*): *int*, der die Anzahl der Knoten im Baum liefert,
- *NrOfInnerNodes*(↓*tree*: *TreePtr*): *int*, der die Anzahl der Knoten liefert, die noch mindestens einen Nachfolger haben (sogenannte *innere Knoten*) und
- *NrOfLeaves*(↓*tree*: *TreePtr*): *int*, der die Anzahl der Knoten liefert, die keine Nachfolger mehr haben (sogenannte Blätter).

Aufgabe 3.16 („Grafische“ Ausgabe eines Binärbaums)
Gegeben ist ein Binärbaum *tree* auf Basis der Deklarationen aus Abschnitt 3.5.2. Gesucht ist ein (iterativer) Algorithmus *WriteTreeGraphically*(↓*tree*: *TreePtr*), der die Werte in der *data*-Komponente der Knoten „graphisch“ in Baumform ausgibt, wobei die Wurzel ganz links auf „mittlerer Höhe“ und die Blätter ganz rechts stehen sollen (d.h. ein um 90 Grad nach links gedrehter Baum). Zur Darstellung der Äste im Baum verwenden Sie das Zeichen ‘<’.

Beispiel: Ein Binärbaum mit einer Wurzel deren *data*-Komponente den Wert 5 hat, die als linken Sohn einen Knoten mit dem Wert 2 und als rechten Sohn einen Knoten mit dem Wert 7 in der *data*-Komponente hat, soll wie folgt ausgegeben werden:

```
      7
5   <
      2
```

Aufgaben

Aufgabe 3.17 (ADS: Kellerspeicher mit Feld)
Gesucht ist *IntStackADS*, eine abstrakte Datenstruktur (ADS) für einen Kellerspeicher (*stack*) auf Basis eines Felds mit dem Elementdatentyp *integer*. Entwickeln Sie alle notwendigen Algorithmen (*Init*, *IsEmpty*, *Push*, *Pop*) für die Verwaltung des Kellerspeichers.

Aufgabe 3.18 (ADT: Kellerspeicher mit Feld)
Gesucht ist *IntStackADT*, ein abstrakter Datentyp (ADT) für einen Kellerspeicher (*stack*) auf Basis eines Felds mit dem Elementdatentyp *integer*. Entwickeln Sie alle notwendigen Algorithmen (*NewStack*, *IsEmpty*, *Push*, *Pop, DisposeStack*) für die Verwaltung des Kellerspeichers.