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*).

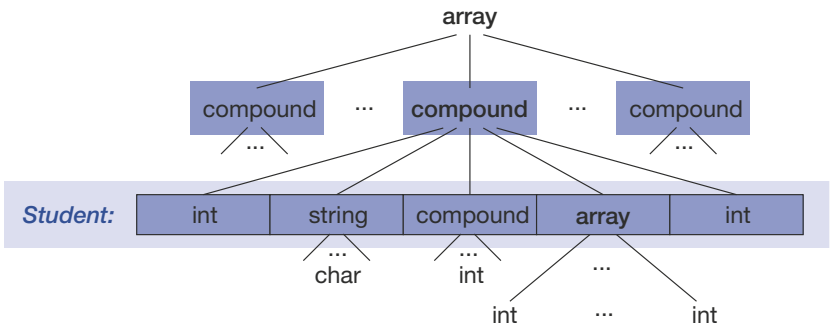

Abbildung 3.9: Hierarchischer Aufbau der Variablen *students*

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.

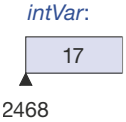

Abbildung 3.10: Variable *intVar* im Speicher, mit Wert und Startadresse

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.

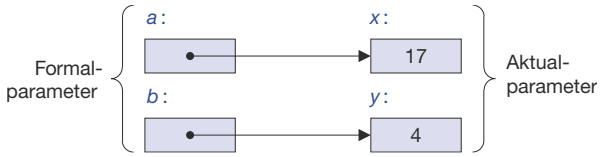

Abbildung 3.12: Formal- und Aktualparameter nach dem Aufruf des Algorithmus *Swap*

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.

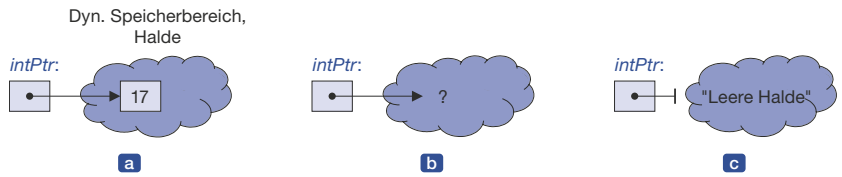

Abbildung 3.13: Zustände nach dem Allokieren und Freigeben von Speicher im dynamischen Speicherbereich

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.)

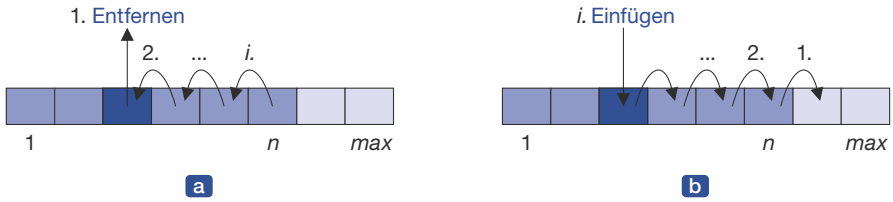

Abbildung 3.14: Entfernen und Einfügen von Elementen in einem Feld

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.