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.)

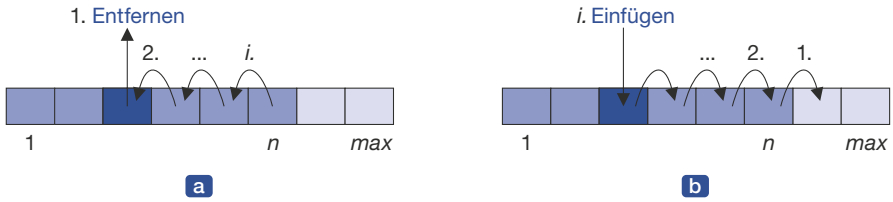

Abbildung 3.14: Entfernen und Einfügen von Elementen in einem Feld

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.

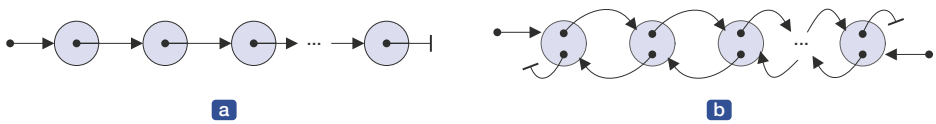

Abbildung 3.15: Einfach- und doppelt-verkettete Listen

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.

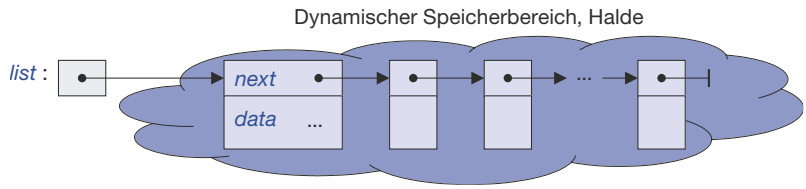

Abbildung 3.16: Einfach-verkettete Liste *list* mit ihren Knoten im dynamischen Speicherbereich

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).

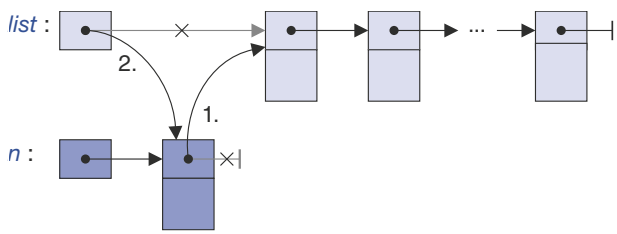

Abbildung 3.17: Einfügen eines Knotens vorne *n* in die Liste *list*

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.

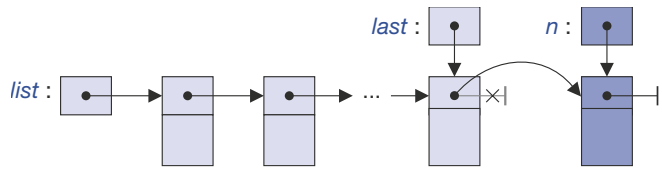

Abbildung 3.18: Anfügen eines Knotens *n* am Ende der Liste *list*

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).

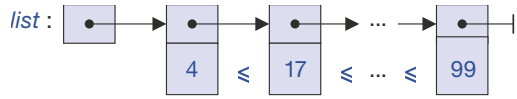

Abbildung 3.19: Aufsteigend sortierte Liste *list*

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.

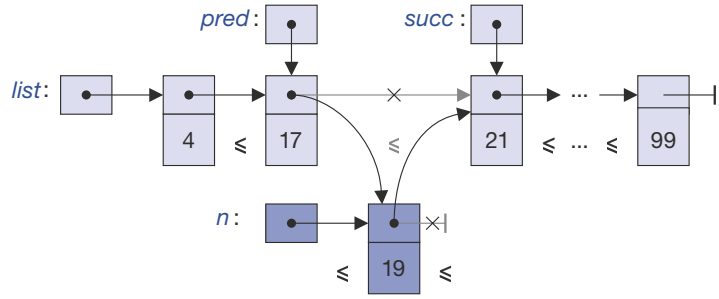

Abbildung 3.20: Einfügen eines Knotens *n* in die aufsteigend sortierte Liste *list*

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.

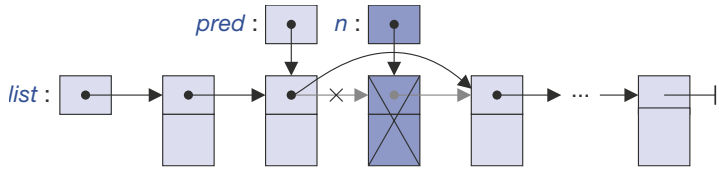

Abbildung 3.21: Entfernen eines Knotens *n* aus der Liste *list*

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.

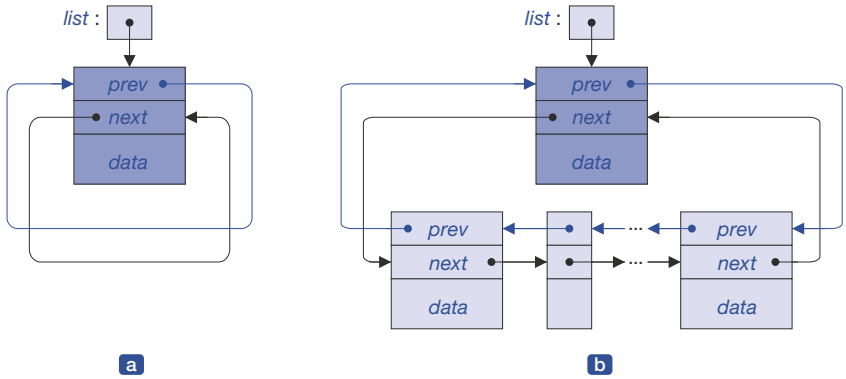

Abbildung 3.22: Realisierung doppelt-verketteter Listen mit einem Ankerknoten

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*).