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.

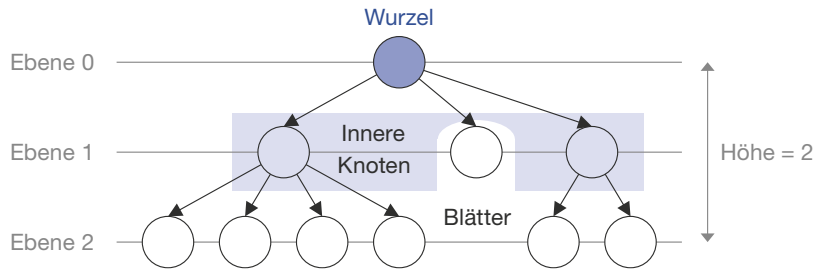

Abbildung 3.23: Baumstruktur und zugeordnete Begriffe

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***.

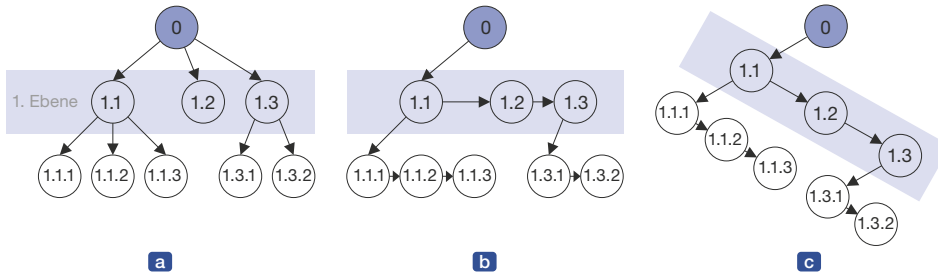

Abbildung 3.24: Repräsentation eines allgemeinen Baums in (a) als Binärbaum in (b) und (c)

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.

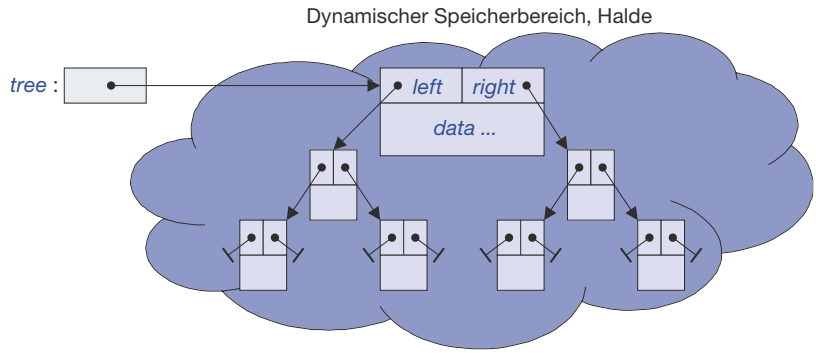

Abbildung 3.25: Perfekter Binärbaum im dynamischen Speicherbereich

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.

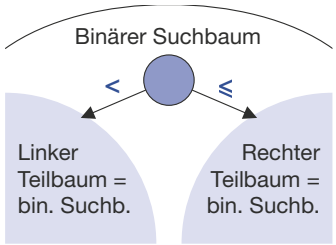

Abbildung 3.26: Binärer Suchbaum

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.

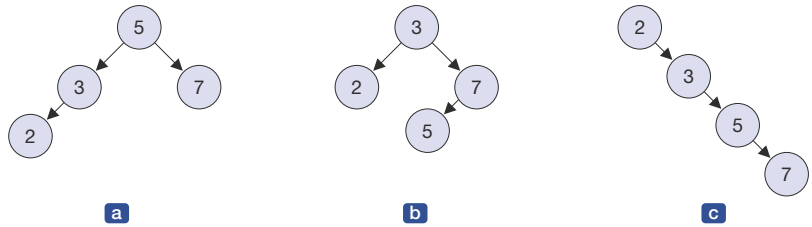

Abbildung 3.27: Auswahl möglicher Anordnungen der ersten vier Primzahlen in binären Suchbäumen

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.