Kapitel 5

Bäume

Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen. Entscheidungsbäume, Syntaxbäume, Ableitungsbäume, Codebäume, spannende Bäume, baumartig strukturierte Suchräume, Suchbäume und viele andere belegen die Allgegenwart von Bäumen. Wir haben in den vorangehenden Kapiteln bereits mehrfach Bäume als intuitives Konzept benutzt, so z.B. zur Erläuterung des Sortierverfahrens Heapsort in Abschnitt 2.3, beim Nachweis unterer Schranken für das Sortierproblem in Abschnitt 2.8 und beim Binärbaum-Sondieren in Abschnitt 4.3.4. Wir wollen jetzt eine systematische Behandlung von Begriffen im Zusammenhang mit Bäumen vornehmen und Algorithmen für Bäume behandeln.

Bäume sind verallgemeinerte Listenstrukturen. Ein Element — üblicherweise spricht man von *Knoten* — hat nicht, wie im Falle linearer Listen, nur einen Nachfolger, sondern eine endliche, begrenzte Anzahl von sogenannten *Söhnen*. In der Regel ist einer der Knoten als *Wurzel* des Baumes ausgezeichnet. Das ist zugleich der einzige Knoten ohne *Vorgänger*. Jeder andere Knoten hat einen (unmittelbaren) Vorgänger, der auch *Vater* des Knotens genannt wird. Eine Folge $p_0, \ldots, p_k$ von Knoten eines Baumes, die die Bedingung erfüllt, daß p_{i+1} Sohn von p_i ist für $0 \leq i < k$, heißt *Pfad* mit Länge k, der p_0 mit p_k verbindet. Jeder von der Wurzel verschiedene Knoten eines Baumes ist durch genau einen Pfad mit der Wurzel verbunden. Man kann Bäume als spezielle planare, zyklenfreie Graphen auffassen. Die Knoten des Baumes sind die Knoten des Graphen; je zwei Knoten p und q sind durch eine Kante miteinander verbunden, wenn q Sohn von p (und damit p Vater von q) ist. Ist unter den Söhnen eines jeden Knotens eines Baumes eine Anordnung definiert, so daß man vom ersten, zweiten, dritten usw. Sohn eines Knotens sprechen kann, so nennt man den Baum *geordnet*. Dies darf man nicht mit der *Ordnung eines Baumes* verwechseln. Darunter versteht man nämlich die maximale Anzahl von Söhnen eines Knotens. Besonders wichtig sind geordnete Bäume der Ordnung 2; sie heißen auch binäre Bäume oder *Binärbäume*. Statt vom ersten und zweiten Sohn spricht man bei Binärbäumen vom *linken* und *rechten* Sohn eines Knotens. Wir werden in diesem Kapitel nur geordnete Bäume betrachten.

Da die Menge der Knoten eines Baumes stets als endlich vorausgesetzt wird, muß es Knoten geben, die keine Söhne haben. Diese Knoten werden üblicherweise als *Blätter* bezeichnet; alle anderen Knoten nennt man *innere Knoten*. Die Menge aller Bäume der

Ordnung d, $d \geq 1$, kann man äquivalent auch rekursiv definieren und entlang dieser Definition auf natürliche Art veranschaulichen:

(1) Der aus einem einzigen Knoten bestehende Baum ist ein Baum der Ordnung d. Wir veranschaulichen ihn graphisch durch:

(2) Sind $t_1, \ldots, t_d$ beliebige Bäume der Ordnung d, so erhält man einen (weiteren) Baum der Ordnung d, indem man die Wurzeln von $t_1, \ldots, t_d$ zu Söhnen einer neugeschaffenen Wurzel w macht. t_i $(1 \leq i \leq d)$ heißt *i-ter Teilbaum* der Wurzel w. Wir veranschaulichen den neuen Baum graphisch wie in Abbildung 5.1.

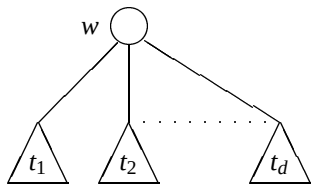


Abbildung 5.1

(In der Informatik wachsen die Bäume also in anderer Richtung als in der Natur: die Wurzel oben, die Blätter unten!)

Wir haben in dieser rekursiven Definition verlangt, daß jeder Knoten eines Baumes der Ordnung d entweder keinen oder genau d Söhne hat. Demzufolge sind die in der Abbildung 5.2 (a) und (b) gezeigten Bäume der Definition entsprechend gültige Binärbäume, der Baum aus Beispiel (c) aber nicht.

Die Anzahl der Söhne eines Knotens p nennt man häufig auch den *Rang* von p. Manchmal bezeichnet man den durch $\square$ veranschaulichten Baum auch als *leeren* Baum und fordert sogar explizit an Stelle der Bedingung (1), daß der aus *keinem* Knoten bestehende leere Baum ein Baum der Ordnung d ist. Dann besagt die Bedingung (2) zwar, daß jeder Knoten eines Baumes der Ordnung d genau d Söhne haben muß; von denen können aber einige oder gar alle leer sein, d.h. es handelt sich um gar nicht existierende Söhne. Das ist eine andere Möglichkeit, um auszudrücken, daß ein Knoten in einem Baum der Ordnung d auch weniger als d Söhne haben kann. Man findet in der Literatur beide Varianten, und wir werden in diesem Kapitel auch beide Varianten benötigen.

Bäume der Ordnung $d > 2$ nennt man auch *Vielwegbäume*. Wir bringen eine wichtige Klasse derartiger Bäume im Abschnitt 5.5, die Klasse der B-Bäume. Sie sind ein typischer Vertreter einer Klasse von Bäumen, für die man üblicherweise fordert, daß die Anzahl der Söhne jedes Knotens zwischen einer festen Unter- und Obergrenze liegen muß. Für Binärbäume werden wir jedoch durchweg verlangen, daß jeder Knoten genau zwei oder keinen Sohn haben soll. Die einzige Ausnahme bilden die im Abschnitt 5.2 behandelten Bruder-Bäume.

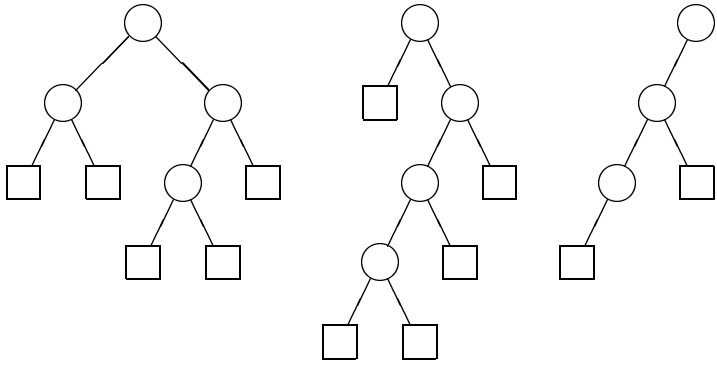


Abbildung 5.2

Wir haben bisher nur strukturelle Eigenschaften und Begriffe im Zusammenhang mit Bäumen besprochen. Dazu gehören auch noch die Begriffe *Höhe* eines Baumes und *Tiefe* eines Knotens. Die Höhe h eines Baumes ist der maximale Abstand eines Blattes von der Wurzel; sie kann auf naheliegende Weise rekursiv definiert werden, siehe Abbildung 5.3.

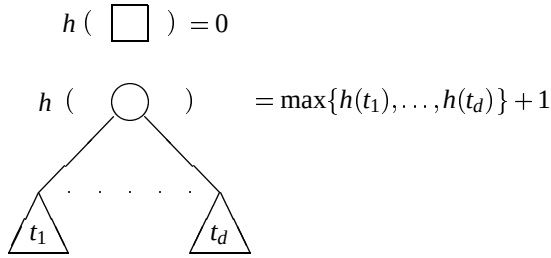


Abbildung 5.3

Der Binärbaum aus Abbildung 5.2 (a) hat also die Höhe 3 und der Binärbaum aus Abbildung 5.2 (b) die Höhe 4. Die *Tiefe* eines Knotens ist sein Abstand zur Wurzel, d.h. die Anzahl der Kanten auf dem Pfad von diesem Knoten zur Wurzel. Man faßt die Knoten eines Baumes gleicher Tiefe zu *Niveaus* zusammen. Die Knoten auf dem Niveau i sind alle Knoten mit Tiefe i.

Ein Baum heißt *vollständig*, wenn er auf jedem Niveau die maximal mögliche Knotenzahl hat und sämtliche Blätter dieselbe Tiefe haben.

Obwohl es eine ganze Reihe interessanter und tiefliegender Sätze über die strukturellen Eigenschaften von Bäumen gibt, ist der eigentliche Grund für die Bedeutung von Bäumen ein anderer. Bäume sind eine Struktur zur Speicherung von Schlüsseln. Wir werden der Einfachheit halber annehmen, daß die Schlüssel stets ganzzahlig sind, wenn nicht ausdrücklich etwas anderes gesagt ist. Die Schlüssel werden dabei so gespeichert, daß sie sich nach einem einfachen und effizienten Verfahren wiederfinden lassen. Das *Suchen* nach einem in einem Baum gespeicherten Schlüssel ist aber nur eine der üblicherweise für Bäume erklärten Operationen. Weitere sind das *Einfügen* eines neuen Knotens (mit gegebenem Schlüssel), das *Entfernen* eines Knotens (mit gegebenem Schlüssel), das *Durchlaufen* aller Knoten eines Baumes in bestimmter Reihenfolge, das *Aufspalten* eines Baumes in mehrere, das *Zusammenfügen* mehrerer Bäume zu einem neuen und das *Konstruieren* eines Baumes mit bestimmten Eigenschaften.

Die drei wichtigsten Operationen sind das Suchen, Einfügen und Entfernen. Man nennt diese drei Operationen auch die *Wörterbuchoperationen* und eine Struktur, die es erlaubt, eine Menge von Schlüsseln zu speichern, zusammen mit Algorithmen für diese Struktur für die Wörterbuchoperationen auch eine Implementation eines *Wörterbuches* (englisch: *dictionary*), vgl. dazu auch Abschnitt 1.6.

In manchen Anwendungen treten praktisch keine Einfügungen und Entfernungen von Knoten auf. Das Universum der in einem Suchbaum abzuspeichernden Schlüssel ist fest und das Suchen die bei weitem überwiegende Operation. Dann kann man einen statischen Suchbaum konstruieren und dabei gegebenenfalls unterschiedliche Suchhäufigkeiten für verschiedene Schlüssel berücksichtigen. Je nachdem, ob die Suchhäufigkeiten fest und vorher bekannt sind oder sich im Laufe der Zeit ändern können, hat man das Ziel, statisch optimale oder dynamisch optimale oder fast optimale Suchbäume zu erzeugen. Wir behandeln nur den statischen Fall genauer in den Abschnitten 5.6 und 5.7.

Das andere Extrem ist der Fall, daß Bäume durch fortgesetztes, iteriertes Einfügen aus dem anfangs leeren Baum erzeugt werden. Wir zeigen im Abschnitt 5.1 über natürliche Bäume, wie man auf einfache Weise zu einer gegebenen Folge von Schlüsseln einen binären Suchbaum so aufbauen kann, daß auch die meisten anderen Operationen einfach ausführbar sind. Es wird sich herausstellen, daß die Reihenfolge, in der die Schlüssel in den anfangs leeren Baum nach und nach eingefügt werden, die Struktur des entstehenden Baumes stark beeinflußt. Es können sowohl zu linearen Listen degenerierte als auch nahezu vollständig ausgeglichene Binärbäume erzeugt werden. Daher kann man nicht ohne weiteres garantieren, daß die drei wichtigsten Basisoperationen für Bäume, das Suchen, Einfügen und Entfernen von Schlüsseln, sämtlich in einer Anzahl von Schritten ausführbar sind, die logarithmisch mit der Anzahl der im Baum gespeicherten Schlüssel wächst.

Es gibt jedoch Techniken, die es erlauben, einen Baum, der nach einer Einfüge- oder Entferne-Operation in Gefahr gerät, *aus der Balance* zu geraten, also zu degenerieren, wieder so zu rebalancieren, daß alle drei Basisoperationen in logarithmischer Schrittzahl ausführbar sind. Einige solcher Rebalancierungstechniken besprechen wir im Abschnitt 5.2 über *balancierte Binärbäume*.

5.1 Natürliche Bäume

In diesem Abschnitt wollen wir zeigen, wie Binärbäume zur Speicherung von Schlüsseln eingesetzt werden können und zwar so, daß man die im Baum gespeicherten Schlüssel auf einfache Weise wiederfinden kann bzw. feststellen kann, daß ein Schlüssel nicht im Baum vorkommt. Wir nehmen an, daß sämtliche Schlüssel paarweise verschieden sind.

Wir können zwei prinzipiell verschiedene Speicherungsformen unterscheiden. Sind die Schlüssel nur in den inneren Knoten gespeichert und haben die Blätter keine Schlüssel, so spricht man von *Suchbäumen*. Sind die Schlüssel in den Blättern gespeichert, spricht man von *Blattsuchbäumen*.

Suchbäume lassen sich folgendermaßen charakterisieren. Für jeden Knoten p gilt: Die Schlüssel im linken Teilbaum von p sind sämtlich kleiner als der Schlüssel von p, und dieser ist wiederum kleiner als sämtliche Schlüssel im rechten Teilbaum von p.

Die Blätter repräsentieren die Intervalle zwischen den in den inneren Knoten gespeicherten Schlüsseln.

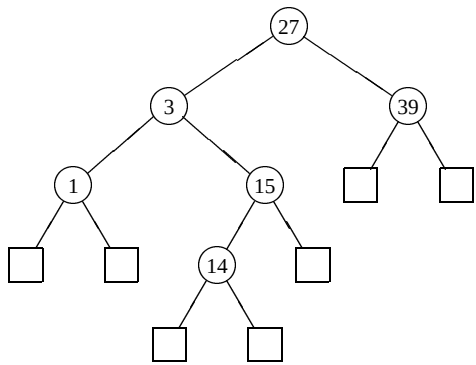


Abbildung 5.4

Abbildung 5.4 zeigt einen binären Suchbaum, der die Schlüsselmenge $\{1, 3, 14, 15, 27, 39\}$ speichert. Diese 6 Schlüssel sind die Schlüssel der inneren Knoten. Die 7 Blätter repräsentieren von links nach rechts die Intervalle $(-\infty, 1)$, $(1, 3)$, $(3, 14)$, $(14, 15)$, $(15, 27)$, $(27, 39)$, $(39, \infty)$.

Der Name *Suchbaum* und auch die Bemerkung, daß die Blätter Schlüsselintervalle repräsentieren, wird erst klar, wenn wir uns überlegen, wie man in einem solchen Baum nach einem Schlüssel x sucht. Wir beginnen bei der Wurzel p und vergleichen x mit dem bei p gespeicherten Schlüssel; ist x kleiner als der Schlüssel von p, setzen wir die Suche beim linken Sohn von p fort. Ist x größer als der Schlüssel von p, setzen wir die Suche beim rechten Sohn von p fort. Genauer verfahren wir nach folgender Methode:

Suche(p,x);
{*sucht im Baum mit Wurzel p nach einem Schlüssel x*}
Fall 1 [*p ist innerer Knoten mit linkem Sohn* p_l *und rechtem Sohn* p_r]

if $x <$ *Schlüssel*(p)
then *Suche*(p_l,x)
else
if $x >$ *Schlüssel*(p)
then *Suche*(p_r,x)
else {$x =$ *Schlüssel*(p),
d.h. gesuchter Schlüssel p gefunden}

Fall 2 [*p ist Blatt*]
{*gesuchter Schlüssel kommt im Baum nicht vor*}

Es ist offensichtlich, daß die Suche nach einem Schlüssel entweder beim Knoten endet, der x speichert, falls x im Baum vorkommt, oder aber an einem Blatt, und zwar an einem Blatt, das ein Intervall repräsentiert, das den gesuchten Schlüssel enthält.

Im Falle von *Blattsuchbäumen* speichern die Blätter die eigentlichen Schlüssel; die inneren Knoten speichern ebenfalls Werte. Die an den inneren Knoten gespeicherten Werte dienen aber lediglich als Wegweiser zu den an den Blättern gespeicherten Schlüsseln. Es gibt viele Möglichkeiten für die Wahl der an den inneren Knoten abzulegenden *Wegweiser*. Jeder zwischen dem maximalen Schlüssel im linken Teilbaum eines Knotens p und dem minimalen Schlüssel im rechten Teilbaum von p liegende Wert ist ein möglicher Kandidat, weil er es erlaubt, eine bei der Wurzel beginnende Suche nach einem an den Blättern gespeicherten Schlüssel bei p richtig zu dirigieren. Eine besonders einfache und übliche Wahl ist es, an jedem inneren Knoten stets den maximalen Schlüssel im linken Teilbaum abzulegen.

Ein Beispiel eines nach diesem Schema aufgebauten Blattsuchbaumes für die Menge $\{1, 3, 14, 15, 27, 39\}$ ist in Abbildung 5.5 dargestellt.

Das Verfahren zum Suchen eines Schlüssel x kann dann offenbar wie folgt beschrieben werden:

Suche(p,x);
{*sucht im Baum mit Wurzel p nach einem Blatt mit Wert x*}
Fall 1 [*p ist innerer Knoten mit linkem Sohn* p_l *und rechtem Sohn* p_r]

if $x \leq$ *Schlüssel*(p)
then *Suche*(p_l,x)
else *Suche*(p_r,x)

Fall 2 [*p ist Blatt*]

if $x =$ *Schlüssel*(p)
then {*Schlüssel bei p gefunden*}
else {*Schlüssel kommt im Baum nicht vor*}

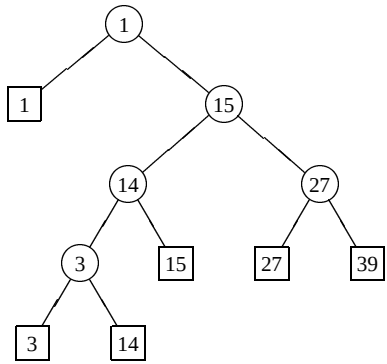


Abbildung 5.5

Wir beschränken uns im folgenden darauf, Algorithmen und Programme für die erste Variante (Suchbäume) anzugeben. Es sollte dem Leser nicht schwerfallen, entsprechende Algorithmen und Programme auch für die zweite Variante (Blattsuchbäume) zu entwickeln.

Es gibt grundsätzlich zwei verschiedene Möglichkeiten, Bäume programmtechnisch zu realisieren, die *Array*- und die *Zeiger*-Realisierung. Bei der *Array-Realisierung* werden die Knoten eines Baumes als Elemente eines Arrays vereinbart. Die Position der Söhne eines Knotens an Position *i* kann durch eine „Adreßrechnung“ aus *i* ermittelt werden. (Diese Art der Realisierung von Bäumen wurde für Heaps im Verfahren Heapsort benutzt.) Bei der *Zeiger-Realisierung* wird die Beziehung zwischen einem Knoten und seinen Söhnen über Zeiger hergestellt. Man vereinbart die Knoten also etwa wie folgt:

```
type
  Knotenzeiger = ↑Knoten;
  Knoten = record
             leftson, rightson : Knotenzeiger;
             key : integer;
             info : {infotype}
           end
```

Ein Baum ist dann gegeben durch einen Zeiger auf die Wurzel:

```
var root : Knotenzeiger
```

Da die Blätter eines Suchbaumes keine Schlüssel (oder andere Informationen) speichern, müssen sie auch nicht explizit als Knoten des oben angegebenen Typs repräsentiert werden. Man kann sie vielmehr einfach durch **nil**-Zeiger in den jeweiligen Vätern repräsentieren. Der in Abbildung 5.4 angegebene Suchbaum zur Speicherung der Schlüsselmenge {1, 3, 14, 15, 27, 39} kann dann etwas genauer wie in Abbildung 5.6

graphisch veranschaulicht werden. Die die Blätter repräsentierenden **nil**-Zeiger sind durch Punkte angedeutet.

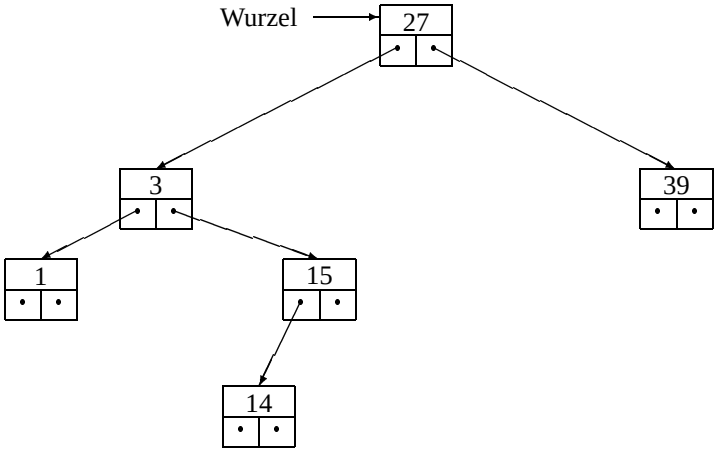


Abbildung 5.6

5.1.1 Suchen, Einfügen und Entfernen von Schlüsseln

Das angegebene Verfahren zum Suchen eines Schlüssels im Baum mit Wurzel *p* kann leicht in eine Pascal-Prozedur übersetzt werden.

```
procedure Suchen (p : Knotenzeiger; x : integer);
{sucht im Baum mit Wurzel p nach Schlüssel x}
begin
  if p = nil
    then write(`Es gibt keinen Knoten im Baum mit Schlüssel x')
    else
      if x < p↑.key
        then Suchen(p↑.leftson, x)
        else
          if x > p↑.key
            then Suchen(p↑.rightson, x)
            else {p↑.key = x}
              write(`Knoten mit Schüssel', x ,`gefunden')
end {Suchen}
```

Statt einer rekursiven hätte man natürlich auch leicht eine iterative Suchprozedur angeben können. Das angegebene Suchverfahren und seine Implementation hat allerdings zwei „Schönheitsfehler“. Erstens wird an jedem Knoten zunächst geprüft, ob der Knoten ein Blatt ist oder nicht. Diese Abfrage ist für alle Knoten mit Ausnahme höchstens des letzten auf jedem Suchpfad negativ zu beantworten. Zweitens kann man auf den Knoten mit Schlüssel *x* nicht wirklich zugreifen, sondern erhält lediglich eine Meldung, daß der Schlüssel *x* gefunden wurde.

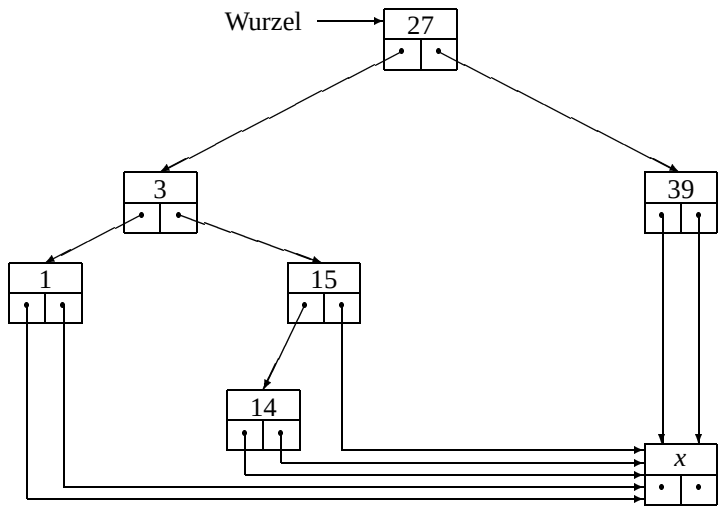


Abbildung 5.7

Den ersten Schönheitsfehler kann man mit einer von linearen Listen bekannten und bewährten Methode beheben. Man verwendet einen fiktiven *Dummy*-Knoten als Stopper, in dem man den gesuchten Schlüssel vor Beginn der Suche ablegt. Wenn sämtliche **nil**-Zeiger durch Zeiger auf diesen Knoten ersetzt werden, endet die Suche auf jeden Fall erfolgreich, nämlich spätestens beim Stopper. Man kann also auf die Abfrage $p =$ **nil** verzichten und kann stattdessen am Ende der Suche prüfen, ob der Schlüssel *x* im Stopper-Knoten gefunden wurde oder nicht. Abbildung 5.7 veranschaulicht diese Implementationsmöglichkeit.

Den zweiten Schönheitsfehler kann man dadurch beheben, daß man an Stelle eines Value-Parameters einen Variable-Parameter verwendet. Man kann aber auch eine Funktion deklarieren, die einen Zeiger auf den gesuchten Knoten abliefert, wenn der gesuchte Knoten im Baum vorkommt, und sonst den Wert **nil**. Wir überlassen die Ausführung der Details dem Leser.

Um einen Schlüssel in einen Suchbaum *einzufügen*, suchen wir zunächst nach dem einzufügenden Schlüssel im gegebenen Baum. Falls der einzufügende Schlüssel nicht schon im Baum vorkommt, endet die Suche erfolglos in einem Blatt, also je nach Implementation bei einem **nil**-Zeiger oder beim Stopper. Wir fügen dann den gesuchten

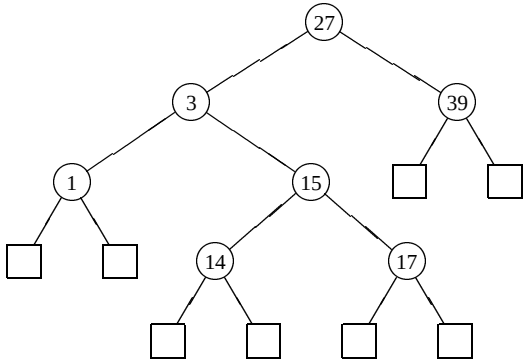


Abbildung 5.8

Schlüssel an der erwarteten Position unter den Blättern ein; d.h. wir ersetzen das Blatt durch einen inneren Knoten mit dem einzufügenden Schlüssel als Wert und zwei Blättern als Söhnen. Auf diese Weise erreicht man offensichtlich, daß der entstehende Baum wieder ein Suchbaum ist.

Fügt man beispielsweise in den eingangs dieses Abschnitts (Abbildung 5.4) angegebenen Suchbaum den Schlüssel 17 ein, so entsteht der Suchbaum in Abbildung 5.8.

Das folgende Programmstück liest eine Folge von paarweise verschiedenen Schlüsseln und fügt sie der Reihe nach in den anfangs leeren Baum ein. Der entstehende Baum ist ein Baum, dessen Blätter durch **nil**-Zeiger repräsentiert werden.

```
program Baumaufbau (input, output);
type
  Knotenzeiger = ↑Knoten;
  Knoten = record
             leftson, rightson : Knotenzeiger;
             key : integer;
             info : {infotype}
           end;
var
  wurzel : Knotenzeiger;
  k : integer;

procedure Einfügen (var p : Knotenzeiger; k : integer);
begin
  if p = nil
    then {neuen Knoten mit Schlüssel k einfügen}
      begin
        new(p);
        p↑.leftson := nil;
        p↑.rightson := nil;
```

```
            p↑.key := k
          end
        else
          if k < p↑.key
            then Einfügen(p↑.leftson, k)
            else
              if k > p↑.key
                then Einfügen(p↑.rightson, k)
                else write(`Schlüssel kam schon vor')
  end; {Einfügen}
  begin {Baumaufbau}
    wurzel := nil;
    while not eof(input) do
      begin
        read(k);
        Einfügen(wurzel, k)
      end
  end. {Baumaufbau}
```

Der auf diese Weise entstehende Suchbaum für eine Menge von Schlüsseln hängt sehr stark davon ab, in welcher Reihenfolge die Schlüssel in den anfangs leeren Baum eingefügt werden. Es können sowohl zu Listen degenerierte Suchbäume der Höhe N entstehen, wenn man N Schlüssel etwa in aufsteigend sortierter Reihenfolge einfügt. Es können aber auch niedrige, nahezu vollständige Suchbäume mit minimal möglicher Höhe $\lceil \log_2 N \rceil$ entstehen, bei denen sämtliche Blätter auf höchstens zwei verschiedenen Niveaus auftreten.

Abbildung 5.9 zeigt als Beispiel für diese beiden Extremfälle zwei Suchbäume für die Menge $\{1, 3, 14, 15, 27, 39\}$, die entstehen, wenn man die Schlüssel in der Reihenfolge 15, 39, 3, 27, 1, 14 bzw. in der Reihenfolge 1, 3, 14, 15, 27, 39 in den anfangs leeren Baum einfügt.

Ein auf diese Weise durch iteriertes Einfügen in den anfangs leeren Baum zu einer Schlüsselfolge entstehender binärer Suchbaum heißt *natürlicher Baum*.

Eine wichtige Frage ist, ob die gut ausgeglichenen niedrigen Bäume oder die hohen, zu Listen degenerierten Bäume häufiger auftreten, wenn man alle den $N!$ möglichen Anordnungen von N Schlüsseln entsprechenden natürlichen Bäume erzeugt. Wir werden diese Frage in Abschnitt 5.1.3 beantworten.

Zunächst überlegen wir uns, wie man einen Schlüssel aus einem Suchbaum *entfernen* kann, so daß der entstehende Baum wieder ein Suchbaum ist. Man sucht zunächst nach dem zu entfernenden Schlüssel x. Kommt x im Baum nicht vor, ist nichts zu tun. Ist x der Schlüssel eines Knotens, der keinen oder nur einen inneren Knoten als Sohn hat, ist das Entfernen einfach. Man entfernt den Knoten mit Schlüssel x und ersetzt ihn gegebenenfalls durch seinen einzigen Sohn. Schwieriger ist das Entfernen von x, wenn x Schlüssel eines Knotens ist, dessen beide Söhne innere Knoten sind, die Schlüssel gespeichert haben. Wir reduzieren in diesem Fall das Problem, den Schlüssel x zu entfernen, folgendermaßen auf einen der beiden einfacheren Fälle. Sei x der Schlüssel des Knotens p. Dann suchen wir im rechten Teilbaum von p den Knoten q mit dem kleinsten Schlüssel y, der größer als x ist. Der Knoten q (und y) heißt der *symmetrische*

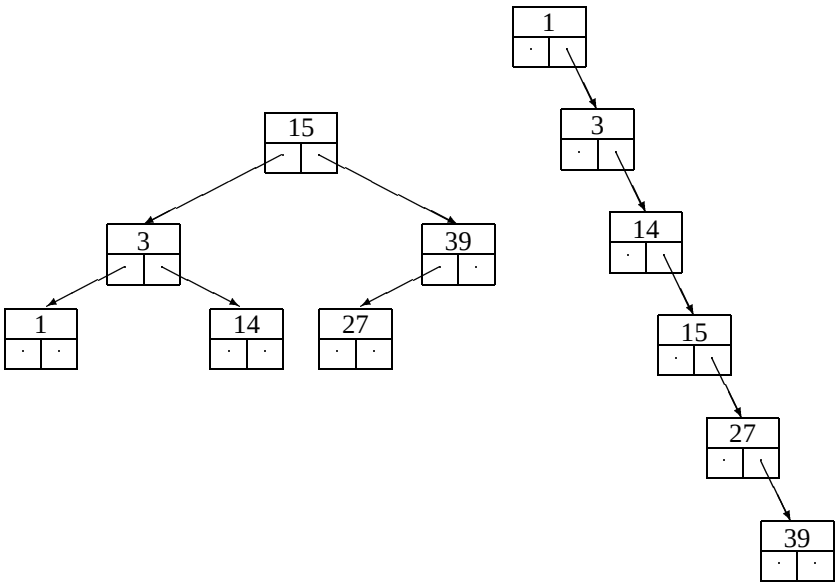


Abbildung 5.9

Nachfolger von p (und x) (vgl. hierzu auch Abschnitt 5.1.2). Der Knoten q ist der am weitesten links stehende innere Knoten im rechten Teilbaum von p und kann daher höchstens einen inneren Knoten als rechten Sohn haben. Man ersetzt nun den Schlüssel x des Knotens p durch den Schlüssel y und entfernt den Knoten q (mit seinem Schlüssel y). Abbildung 5.10 veranschaulicht dies.

Die im folgenden angegebene Prozedur *Entfernen* unter Verwendung der Funktion *vatersymnach* ist eine mögliche Implementation des Verfahrens.

```
function vatersymnach (p : Knotenzeiger) : Knotenzeiger;
{liefert für einen Knotenzeiger p mit p↑.rightson ≠ nil einen Zeiger
 auf den Vater des symmetrischen Nachfolgers von p↑}
begin
  if p↑.rightson↑.leftson ≠ nil
    then {sonst ist p das Ergebnis}
      begin
        p := p↑.rightson;
        while p↑.leftson↑.leftson ≠ nil do
          p := p↑.leftson
      end;
    vatersymnach := p
end {vatersymnach}

procedure Entfernen (var p : Knotenzeiger; k : integer);
{entfernt einen Knoten mit Schlüssel k aus dem Baum mit Wurzel p}
```

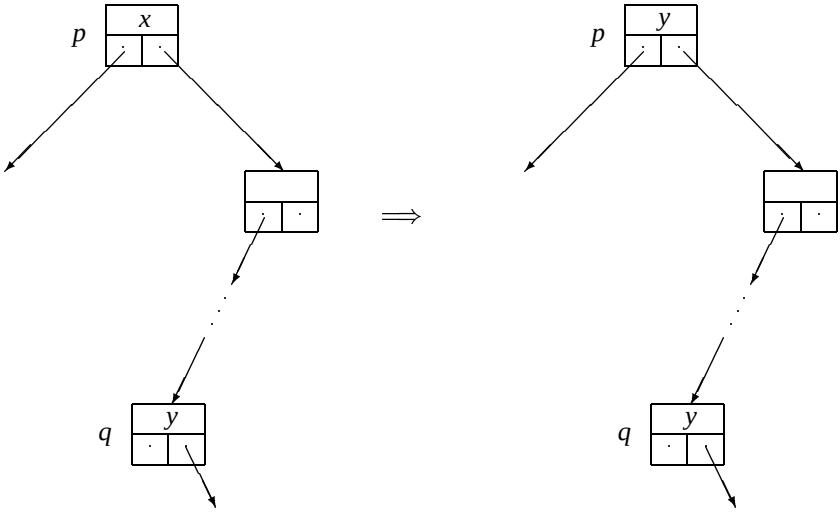


Abbildung 5.10

```
var
  q : Knotenzeiger;
begin
  if p = nil
    then {Schlüssel k nicht im Baum}
    else
      if k <p↑.key
        then Entfernen(p↑.leftson, k)
        else
          if k > p↑.key
            then Entfernen(p↑.rightson, k)
            else {p↑.key = k}
              if p↑.leftson = nil
                then p := p↑.rightson
                else
                  if p↑.rightson = nil
                    then p := p↑.leftson
                    else {p↑.leftson ≠ nil and p↑.rightson ≠ nil}
                      begin
                        q := vatersymnach(p);
                        if q = p
```

```
                    then {rechter Sohn von q ist
                      symmetrischer Nachfolger}
                      begin
                        p↑.key := q↑.rightson↑.key;
                        q↑.rightson := q↑.rightson↑.rightson
                      end
                    else {linker Sohn von q ist
                      symmetrischer Nachfolger}
                      begin
                        p↑.key := q↑.leftson↑.key;
                        q↑.leftson := q↑.leftson↑.rightson
                      end
                end
  end {Entfernen}
```

Wir haben das Entfernen eines Schlüssels eines Knotens p mit zwei inneren Knoten als Söhnen willkürlich auf das Entfernen des symmetrischen Nachfolgers reduziert. Stattdessen hätte man ebensogut den *symmetrischen Vorgänger* von p, d.h. den am weitesten rechts stehenden Knoten im linken Teilbaum von p nehmen können. Man kann auch Strategien implementieren, die mal die eine, mal die andere Möglichkeit wählen. Das hat durchaus Einfluß auf die Struktur der durch iteriertes Entfernen entstehenden Bäume. Wir kommen auf diesen Punkt im Abschnitt 5.1.3 wieder zurück.

5.1.2 Durchlaufordnungen in Binärbäumen

Das Inspizieren aller Knoten eines Graphen im allgemeinen und eines Baumes im besonderen ist häufig nötig, um bestimmte Eigenschaften von Knoten, der in den Knoten gespeicherten Schlüssel und der Struktur des Graphen bzw. Baumes zu ermitteln. Algorithmen zum Durchlaufen aller Knoten eines Baumes in einer bestimmten Reihenfolge bilden das weitgehend problemunabhängige Gerüst für spezifische Aufgaben. Solche Aufgaben sind beispielsweise das Ausdrucken, Markieren, Kopieren usw. aller in einem binären Suchbaum auftretenden Knoten oder Schlüssel in bestimmter Reihenfolge, die Berechnung der Summe, des Durchschnitts, der Anzahl usw. aller in einem Baum gespeicherten Schlüssel, die Ermittlung der Höhe eines Baumes oder der Tiefe eines Knotens, die Prüfung, ob alle Blätter eines Baumes auf demselben Niveau liegen, usw.

Die drei wichtigsten Reihenfolgen, in denen man sämtliche Knoten eines Binärbaumes durchlaufen kann, sind die *Hauptreihenfolge* (oder: *Preorder*), die *Nebenreihenfolge* (oder: *Postorder*) und die *symmetrische Reihenfolge* (oder: *Inorder*). Diese Reihenfolgen lassen sich sehr einfach rekursiv formulieren, das Verfahren zum Durchlaufen aller Knoten eines Baumes in Hauptreihenfolge beispielsweise so:

Durchlaufen aller Knoten eines Binärbaumes mit Wurzel p in Hauptreihenfolge:

1. *Besuche die Wurzel p;*
2. *durchlaufe den linken Teilbaum von p in Hauptreihenfolge;*
3. *durchlaufe den rechten Teilbaum von p in Hauptreihenfolge.*

Grob vereinfacht kann man die Hauptreihenfolge so charakterisieren:
Hauptreihenfolge: Wurzel, linker Teilbaum, rechter Teilbaum.
Entsprechend lauten die übrigen zwei Reihenfolgen:
Nebenreihenfolge: linker Teilbaum, rechter Teilbaum, Wurzel.
Symmetrische Reihenfolge: linker Teilbaum, Wurzel, rechter Teilbaum.
Eine mögliche Implementation etwa der symmetrischen Reihenfolge als rekursive Prozedur ist:

```
procedure symtraverse (p : Knotenzeiger);
{durchläuft sämtliche Knoten des Baumes mit Wurzel p in
 symmetrischer Reihenfolge}
begin
  if p ≠ nil
    then
      begin
        symtraverse(p↑.leftson);
{∗}     {besuche die Wurzel; d.h. gib z.B. den Schlüssel p↑.key
         aus durch write(p↑.key)}
        symtraverse(p↑.rightson)
      end
end {symtraverse}
```

Schreibt man an Stelle des Kommentars $\{*\}$ in dieser Prozedur wirklich die Anweisung *write*$(p\uparrow.key)$ und ruft die Prozedur *symtraverse* für die Wurzel eines binären Suchbaums auf, so werden die im Baum gespeicherten Schlüssel in aufsteigend sortierter Reihenfolge ausgegeben.

Abbildung 5.11 zeigt einen Suchbaum mit sechs Schlüsseln und die Folge der Schlüssel in Haupt-, Neben- und symmetrischer Reihenfolge.

Hauptreihenfolge:
17, 11, 7, 14, 12, 22

Nebenreihenfolge:
7, 12, 14, 11, 22, 17

Symmetrische Reihenfolge:
7, 11, 12, 14, 17, 22

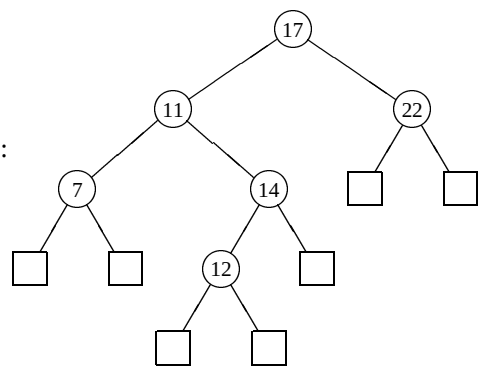


Abbildung 5.11

Die Bezeichnungen Haupt-, Neben- und symmetrische Reihenfolge bzw. Preorder, Postorder, Inorder sollen deutlich machen, wann die Wurzel eines Baumes betrachtet

wird: Vor, nach oder zwischen den Teilbäumen. Natürlich gibt es zu den von uns angegebenen Links-vor-rechts-Varianten auch die umgekehrten, in denen jeweils die rechten Teilbäume vor den linken betrachtet werden.

Da man bekanntlich jede rekursive Prozedur unter Zuhilfenahme eines Stapels in eine äquivalente iterative umwandeln kann, gilt dies natürlich insbesondere für die oben angegebenen Prozeduren zum Durchlaufen der Knoten in Haupt-, Neben- und symmetrischer Reihenfolge.

Eine Möglichkeit, Rekursion und Stapel beim Durchlaufen von Bäumen gänzlich zu vermeiden, besteht in der Einführung zusätzlicher Zeiger. Von jedem Knoten gibt es einen Zeiger auf dessen Nachfolger in der Haupt-, Neben- oder symmetrischen Reihenfolge; diese Zeiger müssen unter Umständen zusätzlich zu den schon bestehenden, von den Vätern auf die jeweiligen Söhne zeigenden Verweisen vorgesehen werden. Das ist im Falle der symmetrischen Reihenfolge jedoch nicht nötig. Der symmetrische Nachfolger eines inneren Knoten p ist nämlich entweder der linkeste Knoten im rechten Teilbaum, falls p überhaupt einen rechten Teilbaum hat, oder aber, falls p keinen rechten Teilbaum hat, ein weiter oben im Baum vorkommender Knoten. Im letzten Fall kann man an Stelle des **nil**-Zeigers, der andeutet, daß p keinen rechten Sohn hat, einen Zeiger auf den symmetrischen Nachfolger von p als Wert von $p\uparrow.rightson$ abspeichern.

Entsprechend kann man auch für die Knoten ohne linken Sohn an Stelle des **nil**-Zeigers einen Zeiger auf den symmetrischen Vorgänger in $p\uparrow.leftson$ ablegen. Dann treten je ein **nil**-Zeiger nur noch beim linkesten und rechtesten Knoten auf. Bäume mit dieser Zeigerstruktur heißen üblicherweise *gefädelte* Bäume. Ein Beispiel zeigt Abbildung 5.12.

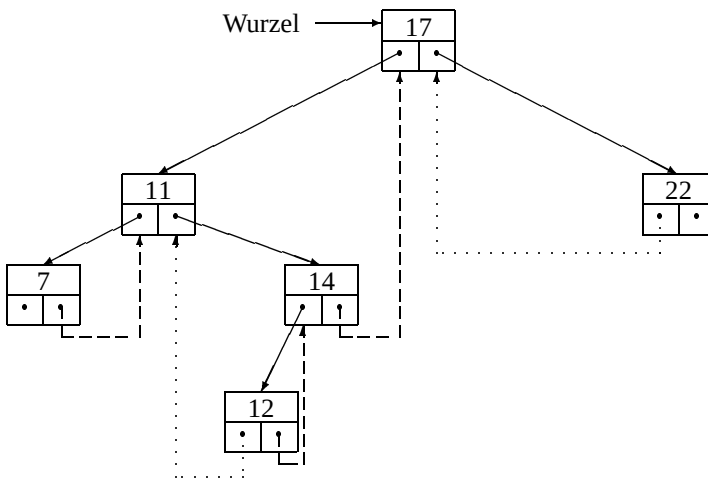


Abbildung 5.12

Natürlich muß man jetzt die *Fädelungszeiger* von den *echten* Zeigern unterscheiden können, die von den Vätern auf die jeweiligen Söhne zeigen. Setzen wir das einmal voraus, so kann man beispielsweise den symmetrischen Nachfolger eines Knotens wie folgt bestimmen:

Algorithmus *symnach* (*p* : *Knotenzeiger*) : *Knotenzeiger*;
Fall 1 [*p*↑.*rightson* = *nil*]
Dann hat p keinen symmetrischen Nachfolger.
Fall 2 [*p*↑.*rightson* ≠ *nil*]
[*Fall 2.1*] [*p*↑.*rightson ist Fädelungszeiger*]
symnach := *p*↑.*rightson*;
[*Fall 2.2*] [*p*↑.*rightson ist kein Fädelungszeiger*]
q :=*p*↑.*rightson*;
while *q*↑.*leftson* ≠ *p* **do** *q* := *q*↑.*leftson*;
symnach := *q*.

Um die Knoten in symmetrischer Reihenfolge zu durchlaufen, genügt es dann, den linkesten Knoten im Baum zu bestimmen und von dort aus mit Hilfe von *symnach* solange den symmetrischen Nachfolger des jeweils betrachteten Knotens zu besuchen, bis der rechteste Knoten *r* im Baum erreicht ist, der offenbar durch die Bedingung *r*↑.*rightson* = **nil** charakterisiert ist.

Man kann binäre Suchbäume von vornherein in dieser Form als gefädelte Bäume aufbauen. Dazu müssen natürlich beim Einfügen und Entfernen von Schlüsseln die Fädelungszeiger gegebenenfalls neu adjustiert werden. Wir überlassen es dem Leser, sich die Implementationsdetails zu überlegen.

5.1.3 Analytische Betrachtungen

Ein Binärbaum mit N inneren Knoten hat $N+1$ Blätter. Seine Höhe kann maximal N sein und muß mindestens $\lceil \log_2(N+1) \rceil$ sein. Der Aufwand zum Ausführen der drei wichtigsten Operationen für binäre Suchbäume, das Suchen, Einfügen und Entfernen von Schlüsseln, hängt unmittelbar von der Höhe des jeweiligen Baumes ab. In jedem Fall muß man ungünstigstenfalls einem Pfad von der Wurzel zu einem Blatt folgen, um die Operation auszuführen. Der im schlechtesten Fall erforderliche Aufwand zum Suchen, Einfügen und Entfernen eines Schlüssels in einem binären Suchbaum mit Höhe h ist damit von der Größenordnung $O(h)$. Dabei kann h zwischen $\lceil \log_2(N+1) \rceil$ und N liegen, wenn der Baum vor Ausführen der Operation N Schlüssel hatte. Im schlechtesten Fall sind Suchbäume und die wichtigsten für sie typischen Operationen nicht besser als verkettet gespeicherte lineare Listen. Wir wollen jetzt zeigen, daß das Verhalten im Mittel wesentlich besser ist. Um dieser Aussage einen präzisen Sinn zu geben, muß zunächst genau gesagt werden, worüber denn gemittelt wird. Dafür gibt es zwei grundsätzlich verschiedene Möglichkeiten.

(a) *Random-tree-Analyse:* Wir nehmen an, daß jede der $N!$ möglichen Anordnungen von N Schlüsseln gleichwahrscheinlich ist und betrachten den Suchbaum, der zu einer zufällig gewählten Folge von N Schlüsseln durch iteriertes Einfügen in den anfangs leeren Baum entsteht. Gemittelt wird hier also über die den $N!$ möglichen Schlüsselfolgen zugeordneten natürlichen Bäume.

(b) *Gestalts-Analyse:* Wir betrachten die Menge aller strukturell verschiedenen binären Suchbäume mit N Schlüsseln und bilden das Mittel über diese Menge.

Nehmen wir als Beispiel die Menge aller möglichen Anordnungen der drei Schlüssel $\{1, 2, 3\}$ und die Menge der strukturell verschiedenen Suchbäume zur Speicherung dieser drei Schlüssel: Fügt man die Schlüssel der Reihe nach in den anfangs leeren Baum ein, so werden gut ausgeglichene, niedrige Bäume und zu linearen Listen degenerierten, hohen Bäume mit jeweils unterschiedlicher Häufigkeit erzeugt. Die Übersicht in Abbildung 5.13 zeigt alle strukturell verschiedenen Suchbäume mit drei Schlüsseln und die Permutationen, die sie jeweils erzeugen.

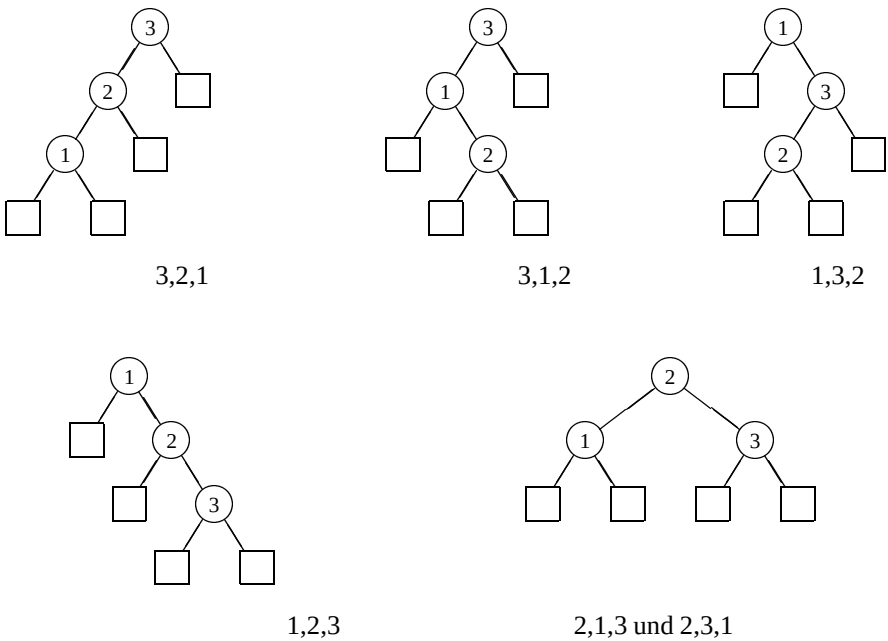


Abbildung 5.13

Der vollständige Binärbaum mit Höhe 2 wird von zwei, jeder der vier verschiedenen Bäume mit Höhe 3 nur von je einer Permutation erzeugt.

Als ein Maß für die Güte eines binären Suchbaumes führen wir die *interne Pfadlänge* und die *durchschnittliche Suchpfadlänge* ein. Die interne Pfadlänge $I(t)$ eines Baumes t ist die Summe aller Abstände der inneren Knoten zur Wurzel. Man kann die interne Pfadlänge rekursiv wie folgt definieren:

(0) Ist $t = \square$, so ist $I(t) = 0$.

(1) Ist t ein Baum mit linkem Teilbaum mit Wurzel t_l und rechtem Teilbaum mit Wurzel t_r, so ist

$$I(t) = I(t_l) + I(t_r) + \text{ Zahl der inneren Knoten von } t.$$

Denn von der Wurzel von t aus gesehen haben alle inneren Knoten von t_l und t_r einen um 1 größeren Abstand zur Wurzel von t als zur jeweiligen Wurzel von t_l bzw. t_r. Die Wurzel von t hat den Abstand 1 zur Wurzel von t. Die interne Pfadlänge mißt also die gesamten *Besuchskosten* für die inneren Knoten des Baumes. Es ist leicht zu sehen, daß gilt:

$$I(t) = \sum_{\substack{p \text{ innerer} \\ \text{Knoten von } t}} (\text{Tiefe}(p) + 1)$$

Ein Beispiel ist in Abbildung 5.14 dargestellt.

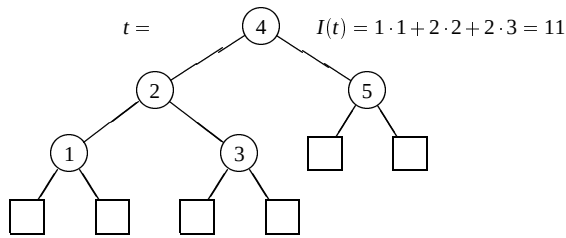


Abbildung 5.14

Bezeichnen wir die Anzahl der inneren Knoten eines Baumes t mit $|t|$, so ist die *durchschnittliche Suchpfadlänge*

$$\bar{I}(t) = \frac{I(t)}{|t|}.$$

Die durchschnittliche Suchpfadlänge mißt also, wieviele Knoten bei erfolgreicher Suche nach einem im Baum t gespeicherten Schlüssel im Mittel (über alle Schlüssel) zu besuchen sind.

Wir berechnen jetzt die Erwartungswerte von $I(t)$ und $\bar{I}(t)$ für einen zufällig erzeugten bzw. für einen der strukturell möglichen Bäume mit N inneren Knoten.

Random trees

Die Berechnung der internen Pfadlänge eines zufällig erzeugten, binären Suchbaumes kann sehr ähnlich erfolgen wie die Berechnung der mittleren Laufzeit des Sortierverfahrens Quicksort. Wir können ohne Einschränkung annehmen, daß die Menge der N iteriert in den anfangs leeren Baum einzufügenden Schlüssel die Menge $\{1,\ldots,N\}$ ist. Ist dann $s_1,\ldots,s_N$ eine zufällige Permutation dieser N Schlüssel, so ist die erste Zahl $s_1 = k$ mit Wahrscheinlichkeit $1/N$ für jedes k zwischen 1 und N. Wird k Schlüssel der Wurzel, so hat der linke Teilbaum der Wurzel, der alle Schlüssel enthält, die kleiner als k sind, $k-1$ Elemente und der rechte Teilbaum der Wurzel entsprechend $N-k$ Elemente.

Bezeichnen wir mit $EI(N)$ den Erwartungswert für die interne Pfadlänge eines zufällig erzeugten binären Suchbaumes mit N inneren Knoten, so erhält man aus der bereits angegebenen Rekursionsformel zur Berechnung der internen Pfadlänge unmittelbar:

$$\begin{aligned} EI(0) &= 0, \quad EI(1) = 1, \\ EI(N) &= \frac{1}{N}\sum_{k=1}^{N}(EI(k-1)+EI(N-k)+N) \\ &= N+\frac{1}{N}(\sum_{k=1}^{N}EI(k-1)+\sum_{k=1}^{N}EI(N-k)) \\ &= N+\frac{2}{N}(\sum_{k=0}^{N-1}EI(k)) \end{aligned}$$

Also ist

$$EI(N+1) = (N+1)+\frac{2}{N+1}\cdot\sum_{k=0}^{N}EI(k),$$

und daher

$$\begin{aligned} (N+1)\cdot EI(N+1) &= (N+1)^2+2\cdot\sum_{k=0}^{N}EI(k) \\ N\cdot EI(N) &= N^2+2\cdot\sum_{k=0}^{N-1}EI(k). \end{aligned}$$

Aus den beiden letzten Gleichungen folgt

$$\begin{aligned} (N+1)EI(N+1)-N\cdot EI(N) &= 2N+1+2\cdot EI(N) \\ (N+1)EI(N+1) &= (N+2)EI(N)+2N+1 \\ EI(N+1) &= \frac{2N+1}{N+1}+\frac{N+2}{N+1}EI(N). \end{aligned}$$

Nun zeigt man leicht durch vollständige Induktion über N, daß für alle $N \geq 1$ gilt:

$$EI(N) = 2(N+1)H_N - 3N$$

Dabei bezeichnet $H_N = 1 + \frac{1}{2} + \ldots + \frac{1}{N}$ die N-te harmonische Zahl, die wie folgt abgeschätzt werden kann:

$$H_N = \ln N + \gamma + \frac{1}{2N} + O(\frac{1}{N^2})$$

Dabei ist $\gamma = 0.5772\ldots$ die sogenannte Eulersche Konstante. Damit ist

$$EI(N) = 2N\ln N - (3-2\gamma)\cdot N + 2\ln N + 1 + 2\gamma + O(\frac{1}{N})$$

und daher

$$\begin{aligned}
\frac{EI(N)}{N} &= 2\ln N - (3-2\gamma) + \frac{2\ln N}{N} + \ldots \\
&= \frac{2}{\log_2 e}\cdot\log_2 N - (3-2\gamma) + \frac{2\ln N}{N} + \ldots \\
&= \frac{2\log_{10} 2}{\log_{10} e}\cdot\log_2 N - (3-2\gamma) + \frac{2\ln N}{N} + \ldots \\
&\approx 1.386\log_2 N - (3-2\gamma) + \frac{2\ln N}{N} + \ldots
\end{aligned}$$

Wir vergleichen diesen Wert für den mittleren Abstand zur Wurzel eines Knotens in einem zufällig erzeugten Baum mit dem mittleren Abstand eines Knotens in einem vollständigen Binärbaum mit $N = 2^h - 1$ inneren Knoten. In einem vollständigen Binärbaum mit Höhe h hat jeder innere Knoten zwei innere Knoten oder zwei Blätter als Söhne, und alle Blätter haben dieselbe Tiefe. Für einen solchen Baum ist die durchschnittliche Suchpfadlänge minimal unter allen Bäumen mit derselben Knotenzahl. Sie ist offenbar:

$$\bar{I}_{min}(N) = \frac{1}{N}\sum_{i=0}^{h-1}(i+1)\cdot 2^i = \frac{1}{2^h-1}[(h-1)\cdot 2^h + 1]$$

Wegen $h = \log_2(N+1)$ ist also:

$$\bar{I}_{min}(N) = \frac{1}{2^h-1}[(h-1)(2^h-1)+h] = \log_2(N+1) + \frac{\log_2(N+1)}{N} - 1$$

Vergleicht man dies mit der zuvor ermittelten durchschnittlichen Suchpfadlänge $\frac{EI(N)}{N}$ eines zufällig erzeugten Baumes, so ergibt sich das bemerkenswerte Ergebnis, daß der Wert für einen zufällig erzeugten Baum nur etwa 40% über dem minimal möglichen liegt.

Erzeugt man also einen binären Suchbaum aus dem anfangs leeren Baum durch iteriertes Einfügen von N Schlüsseln in zufällig gewählter Reihenfolge, so entsteht ein Suchbaum, für den die Suchoperation nur etwa 40% teurer ist als für einen vollständigen binären Suchbaum. Auch eine einzelne weitere Einfüge- und Entferne-Operation in einem solchen Baum kann durchschnittlich in $1.386\log_2 N$ Schritten ausgeführt werden. Führt man jedoch weitere Einfüge- und Entferne-Operationen aus, bleibt das nicht mehr so. Der Grund dafür ist, daß wir das Entfernen eines Schlüssels eines inneren Knotens mit zwei nichtleeren Teilbäumen auf das Entfernen des symmetrischen Nachfolgers reduziert haben. Es leuchtet ein, daß durch diese Vorschrift eher größere Schlüssel zu Schlüsseln der Wurzel werden, also nach vielen Einfügungen und Entfernungen

Bäume entstehen, die „linkslastig“ sind. Denn immer wenn die Wurzel eines (Teil-) Baumes entfernt wird, wird sie durch einen größeren Schlüssel ersetzt, wenn ihr rechter Teilbaum nicht leer war. Eine genaue quantitative Analyse dieses Sachverhaltes gelang J. Culberson [32]. Er hat den Fall analysiert, daß nach N zufälligen Einfügungen in den anfangs leeren Baum jeweils abwechselnd je ein zufällig gewählter Schlüssel entfernt und eingefügt wird. Nennt man ein Paar von Entferne- und Einfüge-Operationen eine *Update-Operation*, so gilt: Führt man in einem zufällig erzeugten Suchbaum mit N Schlüsseln wenigstens N^2 *Update*-Operationen aus, so ist der Erwartungswert für die durchschnittliche Suchpfadlänge $\Theta(\sqrt{N})$ für hinreichend große N. Den nicht einfachen Beweis dieses Sachverhaltes findet man in [32]. Es ist klar, daß ein entsprechendes Ergebnis gilt, wenn man das Entfernen eines Schlüssels statt auf den symmetrischen Nachfolger stets auf den symmetrischen Vorgänger reduziert. Daher liegt es nahe, bei jeder Entfernung zufällig zwischen symmetrischen Vorgängern und symmetrischen Nachfolgern zu wählen. Experimente zeigen, daß dann auch nach einer großen Zahl von Updates besser balancierte Bäume entstehen. Der analytische Nachweis dafür ist bisher nicht gelungen.

Gestaltsanalyse

Wir wollen jetzt die mittlere (gesamte) Pfadlänge eines Baumes mit N inneren Knoten berechnen, wobei über alle strukturell möglichen Bäume gemittelt wird. Es wird sich herausstellen, daß die mittlere Pfadlänge eines Baumes mit N inneren Knoten gleich $N \cdot \sqrt{N \cdot \pi} + O(N)$ ist; jeder Knoten hat also im Mittel einen Abstand $O(\sqrt{N})$ von der Wurzel.

Dieser Nachweis gelingt mit Hilfe sogenannter *erzeugender Funktionen*. Das sind formale Potenzreihen, die zur Analyse struktureller Eigenschaften von rekursiv definierten Strukturen — zu denen ja auch Binärbäume gehören — herangezogen werden können. Wir demonstrieren die Verwendung formaler Potenzreihen zunächst an einem sehr einfachen Beispiel und berechnen die Anzahl der strukturell verschiedenen Binärbäume mit N inneren Knoten. Um sämtliche strukturell möglichen Bäume mit N inneren Knoten zu erzeugen, kann man doch offenbar folgendermaßen vorgehen. Man macht einen Knoten zur Wurzel und wählt unabhängig voneinander alle strukturell möglichen linken und rechten Teilbäume, aber natürlich so, daß insgesamt ein Baum mit N inneren Knoten entsteht. Genauer: Bezeichnen wir mit B_N die Anzahl der strukturell möglichen Binärbäume mit N inneren Knoten, so erhält man alle strukturell möglichen Binärbäume, deren linker Teilbaum genau i innere Knoten enthält (für ein festes i, $0 \leq i \leq N-1$) wie folgt. Man wählt unabhängig voneinander alle strukturell möglichen Binärbäume mit i inneren Knoten als linke und mit $(N-i-1)$ inneren Knoten als rechte Teilbäume und verbindet sie zu einem neuen Binärbaum mit N inneren Knoten; dafür gibt es $B_i \cdot B_{N-i-1}$ Möglichkeiten, vgl. Abbildung 5.15.

Weil i beliebig zwischen 0 und $N-1$ liegen kann, muß also gelten:

$$B_N = B_0 \cdot B_{N-1} + B_1 \cdot B_{N-2} + \ldots + B_{N-1} \cdot B_0$$

Dieser Ausdruck hat eine formale Ähnlichkeit mit den bei der Multiplikation zweier Polynome auftretenden Koeffizienten, die man ausnutzen kann. Wir definieren eine formale Potenzreihe

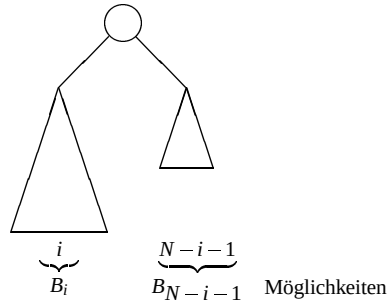


Abbildung 5.15

$$B(z) = \sum_{N \geq 0} B_N \cdot z^N \tag{5.1}$$

und interpretieren die Koeffizienten B_N wie oben angegeben. Dann gilt nach den Rechenregeln für das Multiplizieren formaler Potenzreihen:

$$\begin{aligned} B(z) \cdot B(z) &= (B_0 + B_1 z^1 + B_2 z^2 + \ldots)(B_0 + B_1 z^1 + B_2 z^2 + \ldots) \\ &= (\underbrace{B_0 B_0}_{=B_1} + \underbrace{(B_0 B_1 + B_1 B_0)}_{=B_2} z^1 + \\ &\quad + (\underbrace{B_0 B_2 + B_1 B_1 + B_2 B_0}_{=B_3}) z^2 + \ldots) \end{aligned}$$

Weil natürlich $B_0 = 1$ ist, erhält man also:

$$1 + z \cdot B(z) \cdot B(z) = B(z)$$

Das ist eine quadratische Gleichung für $B(z)$, die leicht formal aufgelöst werden kann und als eine mögliche Lösung liefert:

$$B(z) = \frac{1 - \sqrt{1-4z}}{2z} = \frac{1}{2z}(1 - (1-4z)^{\frac{1}{2}}) \tag{5.2}$$

(Die andere Lösung der quadratischen Gleichung für $B(z)$ kommt nicht in Frage, denn die Gleichung soll ja für beliebige z und damit insbesondere für $z = 0$ gelten, d.h. es muß $B(0) = 1$ sein. Das ist aber nur für die hier angegebene Lösung möglich.)

Bekanntlich gilt für beliebige x mit $|x| < 1$ und r:

$$(1+x)^r = \sum_{k \geq 0} \binom{r}{k} x^k$$

Wendet man das auf Gleichung (5.2) an und setzt $|z| < 1$ voraus, so ergibt sich:

$$
\begin{aligned}
B(z) &= \frac{1}{2z}(1-\sum_{k\geq 0}\binom{\frac{1}{2}}{k}(-4z)^k) \\
&= \frac{1}{2z}(1+\sum_{N+1\geq 0}\binom{\frac{1}{2}}{N+1}(-1)^N(4z)^{N+1}) \\
&= \frac{1}{2z}+\frac{1}{2z}\sum_{N+1\geq 0}\binom{\frac{1}{2}}{N+1}(-1)^N 2^{2N+2}z^{N+1} \\
&= \frac{1}{2z}+\sum_{N+1\geq 0}\binom{\frac{1}{2}}{N+1}(-1)^N 2^{2N+1}z^N \\
&= \frac{1}{2z}+\binom{\frac{1}{2}}{0}(-1)^{-1}2^{-1}z^{-1}+\sum_{N\geq 0}\binom{\frac{1}{2}}{N+1}(-1)^N 2^{2N+1}z^N \\
&= \sum_{N\geq 0}\binom{\frac{1}{2}}{N+1}(-1)^N 2^{2N+1}z^N
\end{aligned}
$$

Ein Koeffizientenvergleich dieser Darstellung mit der ursprünglich definierten Reihe (5.1) ergibt:

$$B_N = \binom{\frac{1}{2}}{N+1}(-1)^N 2^{2N+1} \tag{5.3}$$

Wir haben damit unser Ziel erreicht und einen expliziten Ausdruck für die Anzahl B_N der strukturell möglichen Bäume mit N inneren Knoten gefunden. Die Zahlenfolge (5.3) ist eine in der Zahlentheorie wohlbekannte Folge, nämlich die Folge der Catalanschen Zahlen. Man kann den in (5.3) angegebenen Ausdruck etwas anders schreiben und zeigen, daß gilt:

$$B_N = \frac{1}{N+1}\binom{2N}{N} = \frac{4^N}{N\cdot\sqrt{\pi\cdot N}}+O(\frac{4^N}{\sqrt{N^5}})$$

Auf ähnliche Weise können wir auch die *gesamte interne Pfadlänge* I_N aller strukturell möglichen Bäume mit N inneren Knoten berechnen.

Die gesuchte durchschnittliche Länge eines Suchpfades eines Baumes mit N inneren Knoten ist dann I_N/B_N.

Zur Berechnung von I_N nutzt man die bereits bekannte Möglichkeit zur rekursiven Berechnung der internen Pfadlänge eines Baumes mit N inneren Knoten aus. Ist t ein Baum mit N inneren Knoten und linkem Teilbaum t_l und rechtem Teilbaum t_r, so ist seine interne Pfadlänge $I(t)$ mit:

$$I(t) = \begin{cases} 0, & \text{falls } t = \square; \\ I(t_l)+I(t_r)+|t|, & \text{sonst.} \end{cases}$$

Fragen wir also zunächst: Was ist die gesamte interne Pfadlänge aller strukturell möglichen Binärbäume mit N inneren Knoten, deren linker Teilbaum genau i innere Knoten enthält für ein festes i mit $0 \leq i < N$? Es ist nicht schwer zu sehen, daß wegen der oben angegebenen, für jeden einzelnen Baum geltenden Rekursionsformel gilt:

$$I_i \cdot B_{N-i-1} + B_i \cdot I_{N-i-1} = \begin{array}{l}\text{Gesamtgröße aller Bäume mit} \\ N \text{ inneren Knoten, deren linker} \\ \text{Teilbaum } i \text{ innere Knoten hat.}\end{array}$$

Definiert man also S_N als Summe aller Knotenzahlen aller strukturell möglichen Bäume mit N inneren Knoten, und führt man zwei weitere formale Potenzreihen

$$S(z) = \sum_{N \geq 0} S_N \cdot z^N, \quad I(z) = \sum_{N \geq 0} I_N \cdot z^N$$

ein, so folgt offenbar:

$$\begin{aligned} I(z) &= z \cdot I(z) \cdot B(z) + z \cdot B(z) \cdot I(z) + S(z) \\ &= 2 \cdot z \cdot I(z) \cdot B(z) + S(z) \end{aligned} \tag{5.4}$$

Nun ist nach Definition von $S(z)$ und $B(z)$ natürlich

$$S(z) = \sum_{N \geq 0} N \cdot B_N \cdot z^N$$

und damit

$$S(z) = z \cdot \sum_{N \geq 0} N \cdot B_N \cdot z^{N-1} = z \cdot B'(z). \tag{5.5}$$

Dabei bezeichnet $B'(z)$ die (formale) Ableitung der Potenzreihe $B(z)$, d.h. $B'(z) = \frac{d}{dz}(B(z))$,

$$B'(z) = B_1 + 2B_2 z^1 + 3B_3 z^2 + \ldots$$

Wie im vorigen Fall kann man nun eine explizite Darstellung der gesuchten Koeffizienten I_N herleiten. Aus den Gleichungen (5.4) und (5.5) folgt:

$$\begin{aligned} I(z)(1 - 2zB(z)) &= z \cdot B'(z) \\ I(z) &= \frac{1}{1 - 2zB(z)} \cdot z \cdot \frac{d}{dz}(B(z)) \\ &= \frac{1}{1-4z} - \frac{1}{2z\sqrt{1-4z}} + \frac{1}{2z} \end{aligned}$$

Entwickelt man dies wie vorher in eine unendliche Reihe, erhält man nach einer längeren Rechnung:

$$I(z) = \sum_{N \geq 0} (4^N - (2N+1)B_N) z^N$$

Ein Koeffizientenvergleich ergibt also:

$$I_N = (4^N - (2N+1)B_N)$$

Damit ergibt sich für die mittlere interne Pfadlänge eines Baumes mit N inneren Knoten (gemittelt über alle strukturell möglichen Bäume mit N inneren Knoten):

$$\frac{I_N}{B_N} = N \cdot \sqrt{\pi N} + O(N)$$

Der mittlere Abstand eines Knotens von der Wurzel eines Binärbaumes mit N inneren Knoten ist also ungefähr $\sqrt{\pi \cdot N}$ und nicht $O(\log_2 N)$!

5.2 Balancierte Binärbäume

Das Suchen, Einfügen und Entfernen eines Schlüssels in einem zufällig erzeugten binären Suchbaum mit N Schlüsseln ist zwar im Mittel in $O(\log_2 N)$ Schritten ausführbar. Im schlechtesten Fall kann jedoch ein Aufwand von der Ordnung $\Omega(N)$ zur Ausführung dieser Operationen erforderlich sein, weil der gegebene Baum mit N Schlüsseln zu einer linearen Liste degeneriert ist. Es ist daher natürlich, durch zusätzliche Bedingungen an die Struktur der Bäume ein Degenerieren zu verhindern. Die Operationen zum Einfügen und Entfernen von Schlüsseln werden dann allerdings komplizierter als für die im Abschnitt 5.1 behandelten natürlichen Bäume. Man findet in der Literatur eine große Vielfalt von Bedingungen an die Struktur von Bäumen, die sichern, daß ein Baum mit N Knoten eine Höhe $O(\log N)$ hat und daß Suchen, Einfügen und Entfernen von Schlüsseln in logarithmischer Zeit möglich ist. Der historisch erste Vorschlag aus dem Jahr 1962 sind die AVL-Bäume, die auf Adelson-Velskij und Landis zurückgehen [1]. Hier wird ein Degenerieren von Suchbäumen verhindert durch eine Forderung an die Höhendifferenz der beiden Teilbäume eines jeden Knotens. Diese Bäume heißen daher auch höhenbalancierte Bäume. Wir behandeln AVL-Bäume im Abschnitt 5.2.1. Eng verwandt mit den höhenbalancierten Bäumen sind die in 5.2.2 behandelten Bruder-Bäume. Für sie wird die eine logarithmische Höhe garantierende Dichte erzwungen durch die Forderung, daß alle Blätter denselben Abstand zur Wurzel haben müssen, und durch eine Bedingung an den Verzweigungsgrad von Knoten. In Abschnitt 5.2.3 werden gewichtsbalancierte Bäume betrachtet. Das sind Binärbäume mit der Eigenschaft, daß für jeden Knoten die Gewichte der Teilbäume, das ist die Anzahl ihrer Knoten bzw. Blätter, in einem bestimmten Verhältnis zueinander stehen.

5.2.1 AVL-Bäume

Ein binärer Suchbaum ist *AVL-ausgeglichen* oder *höhenbalanciert*, kurz: ein AVL-Baum, wenn für jeden Knoten p des Baumes gilt, daß sich die Höhe des linken Teilbaumes von der Höhe des rechten Teilbaumes von p höchstens um 1 unterscheidet.

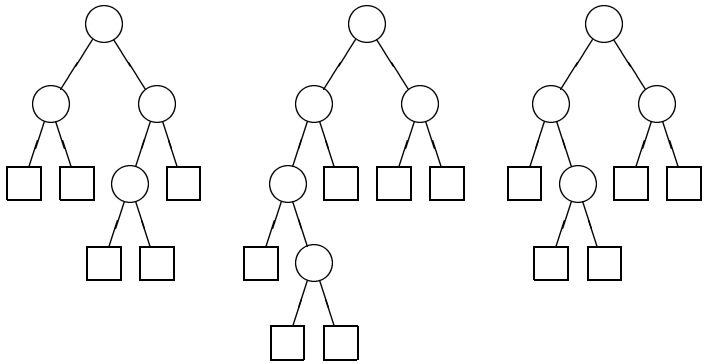


Abbildung 5.16

Die Bäume in Abbildung 5.16 (a) und (c) sind Beispiele für AVL-Bäume. Der Baum in Abbildung 5.16 (b) ist kein AVL-Baum. Da es uns nur auf die Struktur der Bäume ankommt, haben wir die Schlüssel in den Knoten weggelassen.

Wir wollen uns zunächst überlegen, daß AVL-Bäume nicht zu linearen Listen degenerieren können. Die Höhenbedingung sichert vielmehr, daß AVL-Bäume mit N inneren Knoten und $N+1$ Blättern eine Höhe von $O(\log N)$ haben. Dazu überlegen wir uns, was die minimale Blatt- und Knotenzahl eines AVL-Baumes gegebener Höhe h ist.

Offenbar gilt: Ein AVL-Baum der Höhe 1 hat 2 Blätter und ein AVL-Baum der Höhe 2 mit minimaler Blattzahl hat 3 Blätter (vgl. Abbildung 5.17). Einen AVL-Baum der Höhe $h+2$ mit minimaler Blattzahl erhält man, wenn man je einen AVL-Baum mit Höhe $h+1$ und h mit minimaler Blattzahl wie in Abbildung 5.18 zu einem Baum der Höhe $h+2$ zusammenfügt.

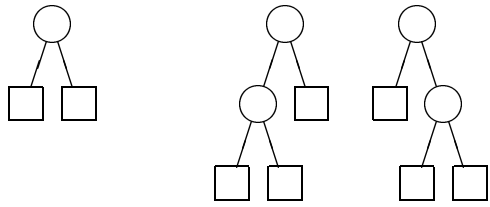


Abbildung 5.17

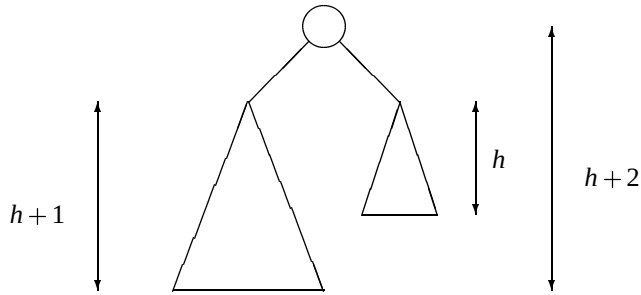


Abbildung 5.18

Bezeichnet nun F_i die i-te Fibonacci-Zahl, also $F_0 = 0$, $F_1 = 1$, $F_{i+2} = F_i + F_{i+1}$, so folgt unmittelbar aus den obigen Überlegungen: Ein AVL-Baum mit Höhe h hat wenigstens F_{h+2} Blätter.

Es gilt

$$F_h = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{h+1} - (\frac{1-\sqrt{5}}{2})^{h+1}),$$

wie man leicht durch vollständige Induktion beweist. Der negative Term $\frac{(1-\sqrt{5})}{2}$ ist dem Betrag nach kleiner als 1 und wird daher mit wachsendem h rasch kleiner. Daher gilt (vgl. auch Abschnitt 3.2.3):

$$F_h \approx \frac{1+\sqrt{5}}{2 \cdot \sqrt{5}}(\frac{1+\sqrt{5}}{2})^h = 0.7236\ldots \cdot 1.618\ldots^h$$

(Genauer: F_h ist die $\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{h+1}$ nächstgelegene ganze Zahl.)

Die Anzahl der Blätter eines AVL-Baumes wächst also exponentiell mit der Höhe. Daraus folgt umgekehrt, daß ein AVL-Baum mit N Blättern (und $N-1$ inneren Knoten) eine Höhe $h \leq 1.44\ldots \log_2 N$ hat. Denn sei ein AVL-Baum mit N Blättern gegeben und sei h seine Höhe. Dann muß gelten:

$$N \geq F_{h+2} \approx 1.894 \cdot 1.618^h,$$

also

$$\begin{aligned} h &\leq \frac{1}{\log_2 1.618\ldots} \cdot \log_2 N - \frac{\log_2 1.894\ldots}{\log_2 1.618\ldots} \\ &\leq 1.44\ldots \log_2 N + 1. \end{aligned}$$

Suchen, Einfügen und Entfernen von Schlüsseln

Da AVL-Bäume insbesondere binäre Suchbäume sind, kann man in ihnen nach einem Schlüssel genauso suchen wie in einem natürlichen Baum. Dazu folgt man im schlechtesten Fall einem Pfad von der Wurzel zu einem Blatt. Weil die Höhe logarithmisch beschränkt bleibt, ist klar, daß man in einem AVL-Baum mit N Schlüsseln in höchstens $O(\log N)$ Schritten einen Schlüssel wiederfinden kann bzw. feststellen kann, daß ein Schlüssel im Baum nicht vorkommt.

Um einen Schlüssel in einen AVL-Baum *einzufügen*, sucht man zunächst nach dem Schlüssel im Baum. Wenn der einzufügende Schlüssel noch nicht im Baum vorkommt, endet die Suche in einem Blatt, das die erwartete Position des Schlüssels repräsentiert. Man fügt den Schlüssel dort ein, wie im Falle natürlicher Bäume. Im Unterschied zu natürlichen Bäumen kann aber nunmehr ein Suchbaum vorliegen, der kein AVL-Baum mehr ist.

Betrachten wir als Beispiel den Baum in Abbildung 5.19.

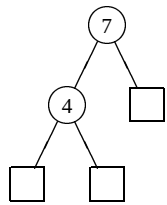


Abbildung 5.19

Fügen wir in diesen Baum den Schlüssel 5 ein, entsteht der Baum in Abbildung 5.20.

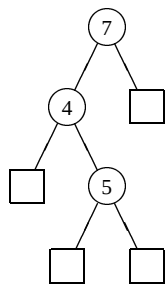


Abbildung 5.20

Das ist kein AVL-Baum mehr, weil für die Wurzel dieses Baumes sich die Höhen des rechten und linken Teilbaumes um mehr als 1 unterscheiden. Man muß also die AVL-

Ausgeglichenheit wiederherstellen. Dazu läuft man von der Einfügestelle den Suchpfad entlang zur Wurzel zurück und prüft an jedem Knoten, ob die Höhendifferenz zwischen linkem und rechtem Teilbaum noch innerhalb der vorgeschriebenen Grenzen liegt. Ist das nicht der Fall, führt man eine sogenannte Rotation oder eine Doppelrotation durch, die die Sortierung der Schlüssel nicht beeinflußt, aber die Höhendifferenzen in den richtigen Bereich bringt.

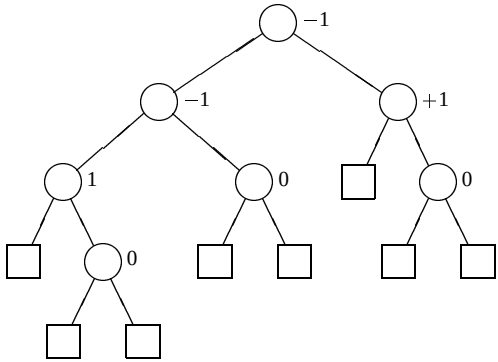
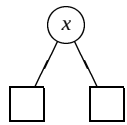


Abbildung 5.21

Man könnte vermuten, daß man zur Prüfung der Höhenbedingung an einem Knoten im Baum die Höhen der Teilbäume des Knotens kennen muß. Das ist jedoch glücklicherweise nicht der Fall. Es genügt, an jedem inneren Knoten p den sogenannten *Balancefaktor* $bal(p)$ mitzuführen, der wie folgt definiert ist:

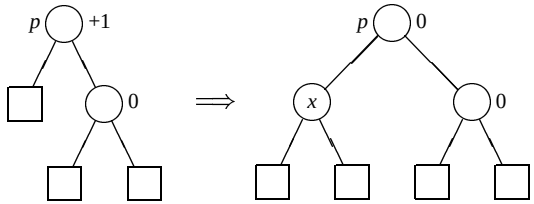
$$bal(p) = \text{Höhe des rechten Teilbaumes von } p \\ - \text{ Höhe des linken Teilbaumes von } p.$$

AVL-Bäume sind offenbar gerade dadurch charakterisiert, daß für jeden inneren Knoten p gilt: $bal(p) \in \{-1, 0, +1\}$. Abbildung 5.21 zeigt einen AVL-Baum mit rechts an die Knoten geschriebenen Balancefaktoren. Da es uns hier nur auf die Struktur des Baumes ankommt, haben wir keine Schlüssel in den Knoten angegeben. Wir geben das Verfahren zum Einfügen eines Schlüssels jetzt genauer an. Wird der Schlüssel x in den leeren Baum eingefügt, erhält man den Baum



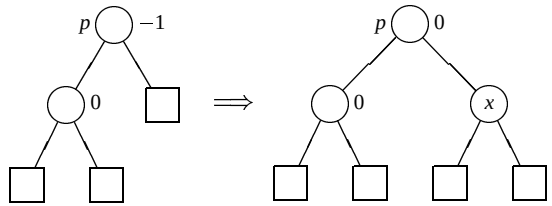
und ist fertig. Sonst sei p der Vater des Blattes, bei dem die Suche endet. Drei Fälle sind möglich:

Fall 1 $[bal(p) = +1]$



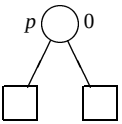
fertig!

Fall 2 $[bal(p) = -1]$



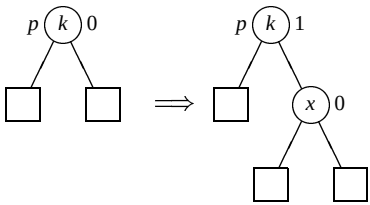
fertig!

Fall 3 $[bal(p) = 0]$



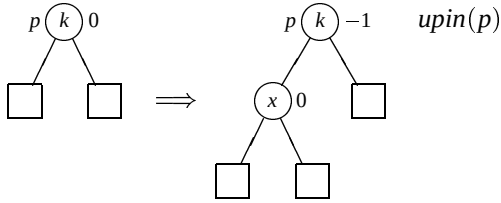
Durch Einfügen eines neuen Knotens als rechten oder linken Sohn von p wird p ein Knoten mit Balancefaktor -1 oder $+1$, und die Höhe des Teilbaumes mit Wurzel p wächst um 1. Wir rufen daher eine Prozedur $upin(p)$ für den Knoten p auf, die den Suchpfad zurückläuft, die Balancefaktoren prüft, gegebenenfalls adjustiert und Umstrukturierungen (sogenannte Rotationen oder Doppelrotationen) vornimmt, die sicherstellen, daß für alle Knoten die Höhendifferenzen der jeweils zugehörigen Teilbäume wieder höchstens 1 sind. Also:

Fall 3.1 $[bal(p) = 0$ und einzufügender Schlüssel $x >$ Schlüssel k von $p]$



$upin(p)$

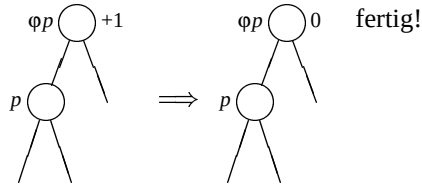
Fall 3.2 [$bal(p) = 0$ und einzufügender Schlüssel $x <$ Schlüssel k von p]



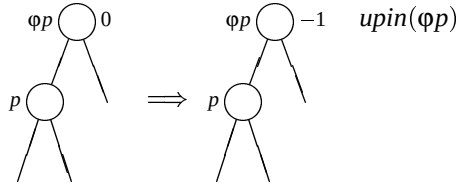
Wir erklären jetzt die Prozedur *upin*. Wenn $upin(p)$ aufgerufen wird, so ist $bal(p) \in \{+1, -1\}$ und die Höhe des Teilbaumes mit Wurzel p ist um 1 gewachsen. Wir müssen darauf achten, daß diese Invariante vor jedem rekursiven Aufruf von *upin* gilt; $upin(p)$ bricht ab, falls p keinen Vater hat, d.h. wenn p die Wurzel des Baumes ist. Wir unterscheiden zwei Fälle, je nachdem ob p linker oder rechter Sohn seines Vaters φp ist.

Fall 1 [p ist linker Sohn seines Vaters φp]

Fall 1.1 [$bal(\varphi p) = +1$]

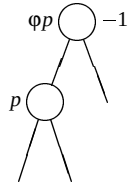


Fall 1.2 [$bal(\varphi p) = 0$]



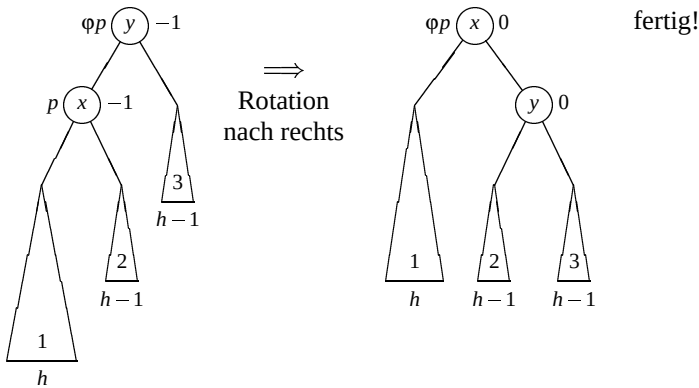
Man beachte, daß vor dem rekursiven Aufruf von *upin* die Invariante gilt.

Fall 1.3 [$bal(\varphi p) = -1$]



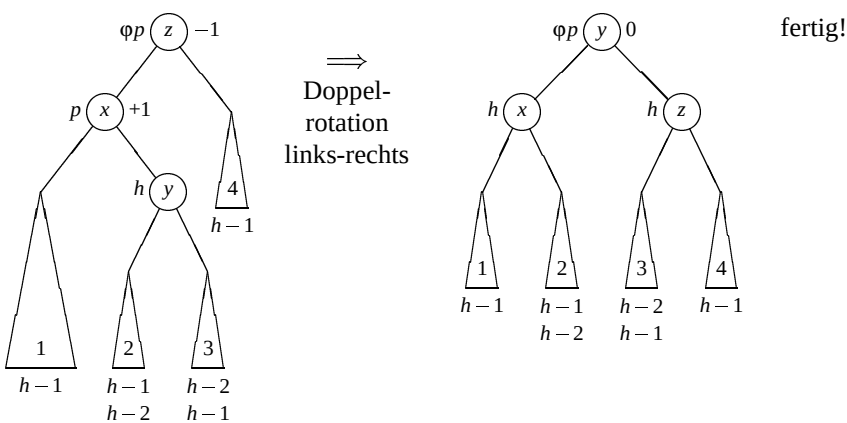
Die Invariante sagt, daß der Teilbaum mit Wurzel p in der Höhe um 1 gewachsen ist. Aus der Voraussetzung $bal(\varphi p) = -1$ kann man in diesem Fall schließen, daß bereits vor dem Einfügen des neuen Schlüssels in den linken Teilbaum von φp mit Wurzel p dieser Teilbaum eine um 1 größere Höhe hatte als der rechte Teilbaum von φp. Da der Teilbaum mit Wurzel p in der Höhe noch um 1 gewachsen ist, ist die AVL-Ausgeglichenheit bei φp verletzt. Wir müssen also umstrukturieren und unterscheiden dazu zwei Fälle, je nachdem, ob $bal(p) = +1$ oder $bal(p) = -1$ ist. (Wegen der Invariante ist $bal(p) = 0$ nicht möglich!)

Fall 1.3.1 $[bal(p) = -1]$



Man beachte: Nach Voraussetzung ist die Höhe des Teilbaumes mit Wurzel p um 1 gewachsen und der linke Teilbaum von p um 1 höher als der rechte. Eine *Rotation nach rechts* bringt den Baum bei φp wieder in die Balance. Es ist keine weitere Umstrukturierung nötig, weil der durch Rotation entstehende Teilbaum mit Wurzel φp in der Höhe nicht mehr gewachsen ist. Wir haben unter die drei Teilbäume die Höhen geschrieben, um so zu zeigen, daß der entstehende Baum nach der Umstrukturierung wieder ausgeglichen ist. Die Höhen sind aber selbstverständlich nicht explizit gespeichert und werden nicht benötigt, um festzustellen, daß die angegebene Umstrukturierung ausgeführt werden soll.

Fall 1.3.2 $[bal(p) = +1]$



Man beachte: Entweder sind die Teilbäume 2 und 3 beide leer oder die einzig möglichen Höhenkombinationen für die Teilbäume 2 und 3 sind $(h-1,h-2)$ und $(h-2,h-1)$. Falls nicht beide Teilbäume leer sind, können sie nicht gleiche Höhe haben. Denn auf Grund der Invarianten ist der Teilbaum mit Wurzel p in der Höhe um 1 gewachsen und wegen der Annahme von Fall 1.3.2 ist der rechte Teilbaum von p um 1 höher als sein linker. Eine *Doppelrotation*, d.h. zunächst eine Rotation nach links bei p und dann eine Rotation nach rechts bei φp, stellt die AVL-Ausgeglichenheit bei φp wieder her. Eine weitere Umstrukturierung ist nicht nötig, da der Teilbaum mit Wurzel φp in der Höhe nicht wächst.

Fall 2 [p ist rechter Sohn seines Vaters φp]

In diesem Fall geht man völlig analog vor und gleicht den Baum, wenn nötig, durch eine Rotation nach links bzw. eine Doppelrotation rechts-links bei φp wieder aus. Zur Veranschaulichung der Rotation nach links liest man die im Fall 1.3.1 gezeigte Abbildung von rechts nach links. Die Doppelrotation rechts-links erhält man aus der im Fall 1.3.2 gezeigten Figur durch Vertauschen der linken und rechten Teilbäume von p und φp.

Wir zeigen die Umstrukturierung noch einmal an einem Beispiel und beginnen mit dem Baum in Abbildung 5.22. Dieser Baum ist ein AVL-Baum. Wir fügen den Schlüssel 9 ein und erhalten Abbildung 5.23.

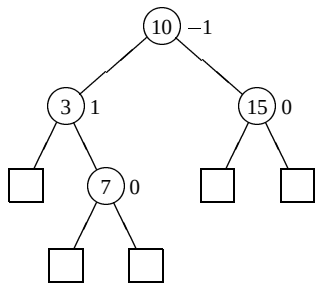


Abbildung 5.22

Das ist kein AVL-Baum mehr; eine Rotation nach links bei p stellt die AVL-Ausgeglichenheit wieder her (siehe Abbildung 5.24). Einfügen von 8 und anschließende Doppelrotation liefert Abbildung 5.25.

Ein Aufruf der Prozedur *upin* kann schlimmstenfalls für alle Knoten auf dem Suchpfad von der Einfügestelle zurück zur Wurzel erforderlich sein. In jedem Fall wird aber höchstens eine Rotation oder Doppelrotation durchgeführt. Denn nach Ausführung einer Rotation oder Doppelrotation in den Fällen 1.3.1 und 1.3.2 und den dazu symmetrischen Fällen wird die Prozedur *upin* nicht mehr aufgerufen. Die Umstrukturierung einschließlich der Adjustierung der Balancefaktoren ist also beendet und die AVL-Ausgeglichenheit wiederhergestellt. Damit ist klar, daß das Einfügen eines neuen Schlüssels in einen AVL-Baum mit N Schlüsseln in $O(\log N)$ Schritten ausführbar ist.

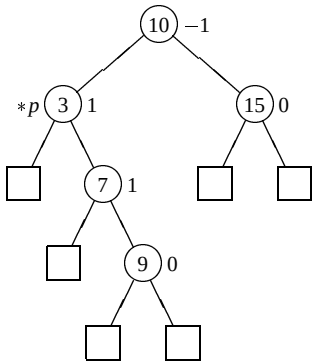


Abbildung 5.23

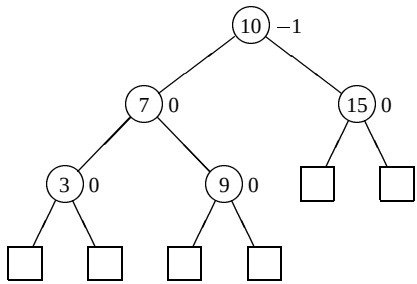


Abbildung 5.24

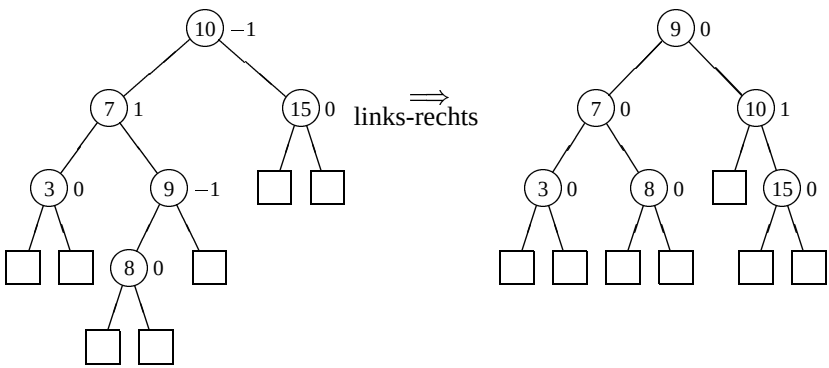


Abbildung 5.25

Das Entfernen eines Schlüssels aus einem AVL-Baum

Zunächst geht man genauso vor wie bei natürlichen Suchbäumen. Man sucht nach dem zu entfernenden Schlüssel. Findet man ihn nicht, ist das Entfernen bereits beendet. Sonst liegt einer der folgenden drei Fälle vor.

Fall 1: Der zu entfernende Schlüssel ist der Schlüssel eines Knotens, dessen beide Söhne Blätter sind. Dann entfernt man den Knoten und ersetzt ihn durch ein Blatt. Falls der Baum nunmehr nicht der leere Baum geworden ist, bezeichne p den Vater des neuen Blattes. Weil der Teilbaum von p, der durch das Blatt ersetzt wurde, die Höhe 1 hatte, muß der andere Teilbaum von p mit Wurzel q die Höhe 0,1 oder 2 haben. Hat er die Höhe 1, so ändert man einfach die Balance von p von 0 auf $+1$ oder -1 und ist fertig. Hat der Teilbaum mit Wurzel q die Höhe 0, so ändert man die Balance p von $+1$ oder -1 auf 0. In diesem Fall ist die Höhe des Teilbaums mit Wurzel p um 1 gefallen. Damit können sich auch für alle Knoten auf dem Suchpfad nach p die Balancefaktoren und die Höhen der Teilbäume verändert haben. Wir rufen daher eine Prozedur $upout(p)$ auf, die die AVL-Ausgeglichenheit wiederherstellt. Hatte schließlich der Teilbaum mit Wurzel q die Höhe 2, d.h. war $bal(p) = -1$ und q kein Blatt, so führt man zunächst eine Rotation oder Doppelrotation aus, um den Baum mit Wurzel p wieder auszugleichen. Dabei kann ein anderer Knoten r an die Wurzel dieses Teilbaumes gelangen. Wenn die Wurzelbalance dieses Teilbaumes auf 0 gesetzt wird, ist seine Höhe um 1 gesunken, so daß wieder $upout(r)$ aufgerufen wird, um die AVL-Ausgeglichenheit wiederherzustellen.

(Bemerkung: Die im letzten Fall erforderlichen Umstrukturierungen werden auch ausgeführt, wenn man die weiter unten beschriebene Prozedur *upout* einfach für das Blatt aufruft, das den entfernten Knoten ersetzt.)

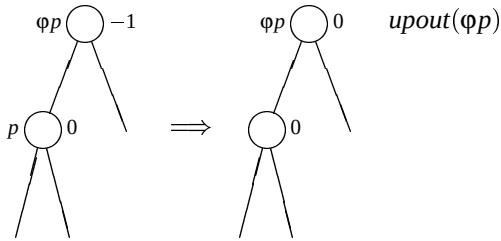
Fall 2: Der zu entfernende Schlüssel ist der Schlüssel eines Knotens p, der nur einen inneren Knoten q als Sohn hat. Dann müssen beide Söhne von q Blätter sein. Man ersetzt also den Schlüssel von p durch den Schlüssel von q und ersetzt q durch ein Blatt. Damit ist nunmehr p ein Knoten mit $bal(p) = 0$ und die Höhe des Teilbaums mit Wurzel p um 1 gesunken (von 2 auf 1). Auch in diesem Fall rufen wir $upout(p)$ auf, um die AVL-Ausgeglichenheit wiederherzustellen.

Fall 3: Der zu entfernende Schlüssel ist der Schlüssel eines Knotens p, dessen beide Söhne innere Knoten sind. Dann geht man wie im Falle natürlicher Suchbäume vor und ersetzt den Schlüssel durch den Schlüssel des symmetrischen Nachfolgers (oder Vorgängers) und entfernt den symmetrischen Nachfolger (oder Vorgänger). Das muß dann ein Knoten sein, dessen Schlüssel wie im Fall 1 und 2 beschrieben entfernt wird.

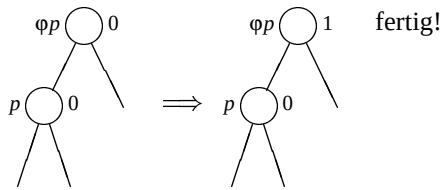
In jedem Fall haben wir das Entfernen reduziert auf die Ausführung der Prozedur $upout(p)$ für einen Knoten p mit $bal(p) = 0$, dessen Teilbaum in der Höhe um 1 gefallen ist.

Wir geben diese Prozedur *upout* nun genauer an. Sie kann längs des Suchpfades rekursiv aufgerufen werden, adjustiert die Höhenbalancen und führt gegebenenfalls Rotationen oder Doppelrotationen durch, um den Baum wieder auszugleichen. Wenn $upout(p)$ aufgerufen wird, gilt: $bal(p) = 0$ und der Teilbaum mit Wurzel p ist in der Höhe um 1 gefallen. Wir müssen darauf achten, daß diese Invariante vor jedem rekursiven Aufruf von *upout* gilt. Wir unterscheiden wieder zwei Fälle, je nachdem ob p linker oder rechter Sohn seines Vaters φp ist.

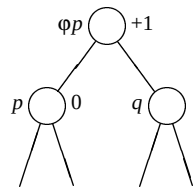
Fall 1 [p ist linker Sohn seines Vaters φp]
Fall 1.1 [$bal(\varphi p) = -1$]



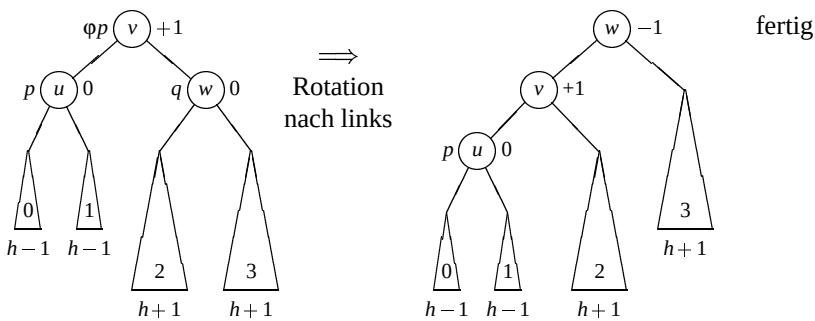
Man beachte, daß vor dem rekursiven Aufruf von *upout* die Invariante für φp gilt.
Fall 1.2 [$bal(\varphi p) = 0$]



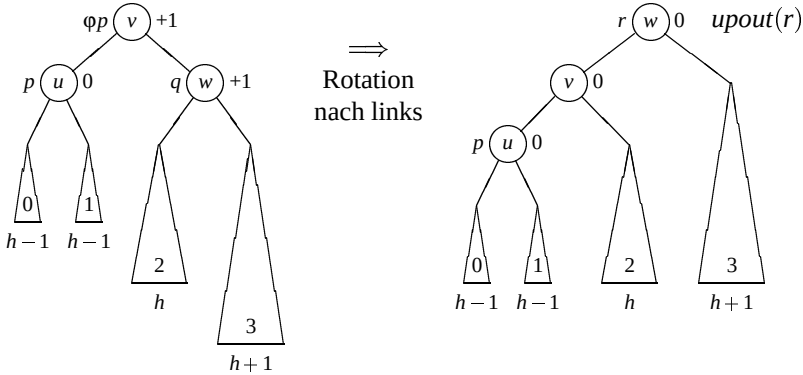
Fall 1.3 [$bal(\varphi p) = +1$]



Der rechte Teilbaum von φp mit Wurzel q ist also höher als der linke mit Wurzel p, der darüber hinaus noch in der Höhe um 1 gefallen ist. Wir machen eine Fallunterscheidung nach dem Balancefaktor von q.
Fall 1.3.1 [$bal(q) = 0$]

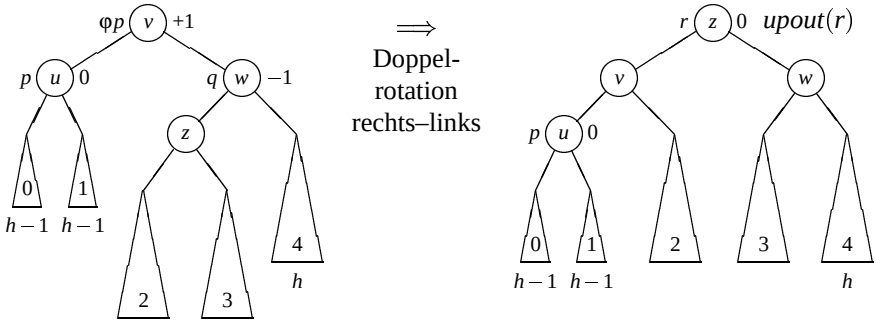


Fall 1.3.2 $[bal(q) = +1]$



Man beachte, daß vor dem rekursiven Aufruf von *upout* die Invariante für r gilt!

Fall 1.3.3 $[bal(q) = -1]$



Weil der Teilbaum mit Wurzel p in der Höhe um 1 gefallen ist und der rechte Teilbaum von φp vor dem Entfernen eines Schlüssels aus dem Teilbaum mit Wurzel p um 1 höher war als der linke, folgt, daß der Teilbaum mit Wurzel q die Höhe $h+2$ haben muß. Wegen $bal(q) = -1$ hat der linke Teilbaum von q mit Wurzel z die Höhe $h+1$ und der rechte die Höhe h. Die Teilbäume von z können entweder beide die Höhe h oder höchstens einer von ihnen die Höhe $h-1$ haben. In jedem Fall gleicht die angegebene Umstrukturierung den Baum wieder aus. Dabei hängen die Balancefaktoren der Knoten v und w vom Balancefaktor von z ab. Auf jeden Fall hat der Teilbaum mit Wurzel r den Balancefaktor 0 und seine Höhe ist um 1 gefallen. Es gilt also die Invariante für den Aufruf von *upout*.

Der Fall 2 [p ist rechter Sohn seines Vaters φp] ist völlig symmetrisch zum Fall 1 und wird daher nicht näher behandelt.

Anders als im Falle der Prozedur *upin* kann es vorkommen, daß auch nach einer Rotation oder Doppelrotation die Prozedur *upout* erneut aufgerufen werden muß. Daher reicht im allgemeinen eine einzige Rotation oder Doppelrotation nicht aus, um den Baum nach Entfernen eines Schlüssels wieder AVL-ausgeglichen zu machen. Es ist nicht schwer, Beispiele zu finden, in denen an allen Knoten auf dem Suchpfad von der Entfernestelle zurück zur Wurzel eine Rotation oder Doppelrotation ausgeführt werden muß. Da jedoch der Aufwand zum Ausführen einer einzelnen Rotation oder Doppelrotation konstant ist, und da die Höhe h von AVL-Bäumen mit N Schlüsseln durch

$1.44\ldots\log_2 N$ beschränkt ist, folgt unmittelbar: Das Enfernen eines Schlüssels aus einem AVL-Baum mit N Schlüsseln ist in $O(\log N)$ Schritten ausführbar. Damit sind alle drei Wörterbuchoperationen Suchen, Einfügen und Entfernen auch im schlechtesten Fall in $O(\log N)$ Schritten ausführbar.

AVL-Bäume sind also eine *worst-case-effiziente* Implementation von Wörterbüchern im Gegensatz zu natürlichen Bäumen, die im *Average-case* zwar genauso effizient sind, im Worst-case aber $\Omega(N)$ Schritte zum Ausführen der Wörterbuchoperationen benötigen.

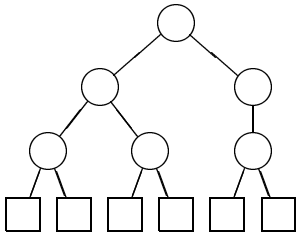
Eine interessante Frage für jede Klasse balancierter Bäume ist, was der mittlere Aufwand zur Ausführung der Wörterbuchoperationen ist, wenn man über eine Folge derartiger Operationen mittelt. Man kann für AVL-Bäume, die im nächsten Abschnitt 5.2.2 behandelten Bruder-Bäume und auch für die im Abschnitt 5.2.3 behandelten gewichtsbalancierten Bäume zeigen, daß der Aufwand pro Einfüge-Operation gemittelt über eine Folge von Einfüge-Operationen *konstant* ist. Dieser Nachweis ist am einfachsten für die Klasse der Bruder-Bäume zu führen. Darüberhinaus können auch weitere, das mittlere Verhalten des Einfügeverfahrens charakterisierende Parameter für die Klasse der Bruder-Bäume besonders leicht hergeleitet werden mit Hilfe einer Technik, die als *Fringe-Analyse* bekannt ist und in Abschnitt 5.2.2 genauer behandelt wird.

5.2.2 Bruder-Bäume

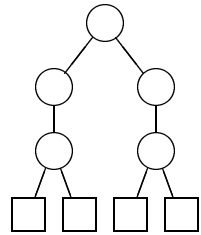
Bruder-Bäume kann man in einem präzisierbaren Sinn als expandierte AVL-Bäume auffassen [138]. Durch Einfügen unärer Knoten an den richtigen Stellen erhält man einen Baum, dessen sämtliche Blätter dieselbe Tiefe haben; und umgekehrt entsteht aus einem Bruder-Baum ein höhenbalancierter Baum, wenn man die unären Knoten mit ihren einzigen Söhnen verschmilzt. Man könnte diesen Zusammenhang dazu benutzen, Such-, Einfüge- und Entferne-Operationen für Bruder-Bäume zu gewinnen, indem man sie von den AVL-Bäumen herüberzieht. Wenn man das macht, erhält man aber Algorithmen, die sich von den im folgenden angegebenen unterscheiden, weniger leicht erklärbar und insbesondere nicht so einfach zu analysieren sind. Unsere Algorithmen folgen einer Strategie, die sich stark an die im Abschnitt 5.5 behandelten Verfahren für B-Bäume anlehnt.

Zunächst jedoch zur genauen Definition der Bruder-Bäume: Im Unterschied zu allen anderen Binärbäumen erlauben wir, daß ein innerer Knoten auch nur einen Sohn haben kann. Natürlich dürfen nicht zu viele unäre Knoten vorkommen, weil man dann offensichtlich entartete Bäume mit großer Höhe und wenigen Blättern erhalten könnte. Man erzwingt daher eine Mindestdichte durch eine Bedingung an die *Brüder* unärer Knoten. Dabei heißen zwei Knoten Brüder, wenn sie denselben Vater haben. Genauer definieren wir: Ein binärer Baum heißt ein *Bruder-Baum*, wenn jeder innere Knoten einen oder zwei Söhne hat, jeder unäre Knoten einen binären Bruder hat und alle Blätter dieselbe Tiefe haben. Abbildung 5.26 enthält einige Beispiele.

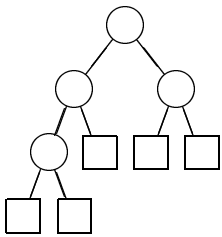
Als unmittelbare Folgerung aus der Definition erhält man: Ist ein Knoten p der einzige Sohn seines Vaters, so ist p ein Blatt oder binär. Von zwei Söhnen eines binären Knotens kann höchstens einer unär sein.



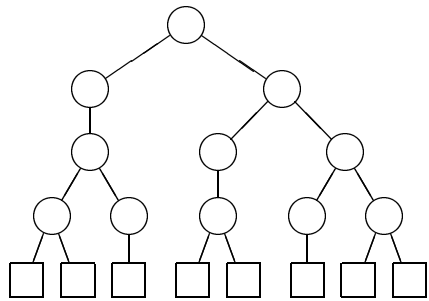
Bruder–Baum



kein Bruder-Baum



kein Bruder-Baum



Bruder-Baum

Abbildung 5.26

Offensichtlich ist die Anzahl der Blätter eines Bruder-Baumes stets um 1 größer als die Anzahl der binären (inneren) Knoten.

Betrachten wir die Folge der Bruder-Bäume mit einer gegebenen Höhe und minimaler Blattzahl in Tabelle 5.1.

Wie für AVL-Bäume folgt auch hier: Ein Bruder-Baum mit Höhe h hat wenigstens F_{h+2} Blätter. (F_i ist die i-te Fibonacci-Zahl.) Also umgekehrt: Ein Bruder-Baum mit N Blättern und $(N-1)$ inneren Knoten hat eine Höhe $h \leq 1.44\ldots \log_2 N$.

Wir haben bislang offengelassen, wie Schlüssel in Bruder-Bäumen gespeichert werden können. Dazu gibt es, wie bei binären Suchbäumen, bei denen jeder innere Knoten zwei Söhne hat, auch zwei Möglichkeiten. Erstens kann man Bruder-Bäume als Blattsuchbäume organisieren. Dann sind die Schlüssel die Werte der Blätter, z.B. von links nach rechts aufsteigend sortiert; innere Knoten enthalten Wegweiser zum Auffinden der Schlüssel an den Blättern. Natürlich genügt es, Wegweiser an den binären Knoten aufzustellen.

Die andere Möglichkeit besteht darin, die Schlüssel in den binären Knoten zu speichern und, wie für binäre Suchbäume, zu verlangen, daß für jeden binären Knoten p gilt: Die Schlüssel im linken Teilbaum von p sind sämtlich kleiner als der Schlüssel von p, und dieser ist wiederum kleiner als sämtliche Schlüssel im rechten Teilbaum von p. Die unären Knoten und die Blätter speichern natürlich keine Schlüssel. Wir wollen im folgenden nur noch diese Variante betrachten und sprechen von *1-2-Bruder-Bäumen*. Diese Bezeichnung hat ihren Ursprung in einer für Vielwegbäume üblichen

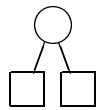
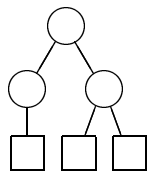
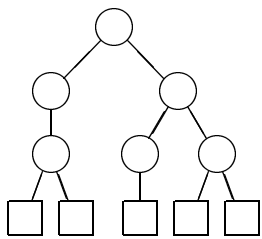
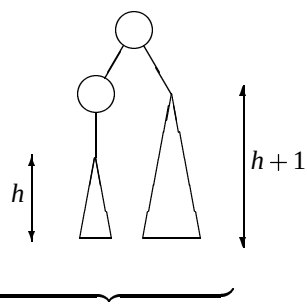
Höhe	Bruder-Bäume mit minimaler Blattzahl	Blattzahl
1		2
2		3
3		5
⋮		⋮
$h+2$		F_{h+4}

Tabelle 5.1

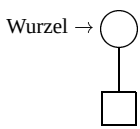
Sprechweise: Man spricht dort von *a-b-Bäumen*, wobei a und b zwei natürliche Zahlen mit $b \geq a$ sind, also z.B. von 2-3-Bäumen, 2-4-Bäumen oder $\lceil m/2 \rceil$-m-Bäumen für ein $m \geq 2$. Das sind Bäume mit der Eigenschaft, daß jeder innere Knoten mindestens a und höchstens b Söhne hat. Man fordert weiter, daß alle Blätter gleiche Tiefe haben müssen und jeder Knoten mit i Söhnen genau $(i-1)$ Schlüssel gespeichert hat. 1-2-Bruder-Bäume sind damit spezielle 1-2-Bäume. Die im Abschnitt 5.5 behandelten B-Bäume sind $\lceil m/2 \rceil$-m-Bäume.

Suchen, Einfügen und Entfernen von Schlüsseln

Bevor wir die Algorithmen zum Suchen, Einfügen und Entfernen von Schlüsseln in 1-2-Bruder-Bäumen angeben, wollen wir noch eine Vorbemerkung zur möglichen Implementation machen. Es ist natürlich, die Knoten eines 1-2-Bruder-Baumes als Record mit Varianten zu definieren. Blätter werden implizit durch **nil**-Zeiger in ihren Vätern repräsentiert. Alle anderen Knoten sind von folgendem Typ:

```
type
  arity = (unary, binary);
  Knotenzeiger = ↑Knoten;
  Knoten = record
             case tag : arity of
               unary : (son : Knotenzeiger);
               binary :(leftson, rightson : Knotenzeiger;
                        key : integer;
                        info : {infotype})
           end
```

Obwohl üblicherweise der leere Baum durch den Wert **nil** einer Variablen *wurzel* vom Typ Knotenzeiger repräsentiert wird, wollen wir hier eine für unsere Zwecke bequemere Form wählen:

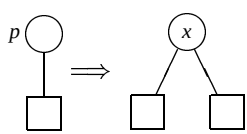


repräsentiert den leeren Baum.

Das *Suchen* in einem 1-2-Bruder-Baum nach einem gegebenen Schlüssel x unterscheidet sich nur unwesentlich vom Suchen in binären Suchbäumen. Man muß lediglich einen weiteren Fall vorsehen. Trifft man bei der Suche nach einem Schlüssel x auf einen unären Knoten, so setzt man die Suche bei dessen Sohn fort.

Zum *Einfügen* eines neuen Schlüssels x in einen 1-2-Bruder-Baum sucht man zunächst im Baum nach x. Wenn der Schlüssel x im Baum noch nicht vorkommt, endet die Suche erfolglos in einem Blatt. Sei p der Vater dieses Blattes.

Fall 1 [p hat nur einen Sohn]



fertig!

Fall 2 [p hat bereits zwei Söhne und damit einen Schlüssel $p.key$]

Wir können ohne Einschränkung annehmen, daß $x < p.key$ ist. (Sonst vertausche man x und $p.key$.) In diesem Fall kann man den Schlüssel x nicht mehr im Knoten p unterbringen. Man versucht daher, den Schlüssel x bzw. einen anderen Schlüssel, um Platz für x zu schaffen, beim Bruder von p oder beim Vater von p unterzubringen. Findet man in der unmittelbaren Verwandtschaft des Knotens p keinen Knoten, der noch Platz hat, also unär war und binär gemacht werden könnte, so verschiebt man das Einfügeproblem rekursiv um ein Niveau nach oben, bis man gegebenenfalls bei der Wurzel angelangt ist. Wenn dieser letzte Fall eintritt, wird der Baum durch Schaffen einer neuen Wurzel um ein Niveau aufgestockt. (Bruder-Bäume wachsen also an der Wurzel und nicht an den Blättern wie die AVL-Bäume!) Man teilt oder spaltet also einen unären bzw. binären Knoten in einen binären bzw. einen unären und einen binären Knoten. Diese intuitive Idee führt zu folgender Prozedur *up*, die in der in Abbildung 5.27 dargestellten Anfangssituation aufgerufen wird.

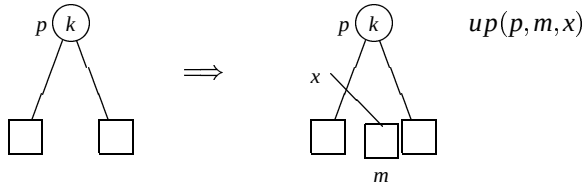


Abbildung 5.27

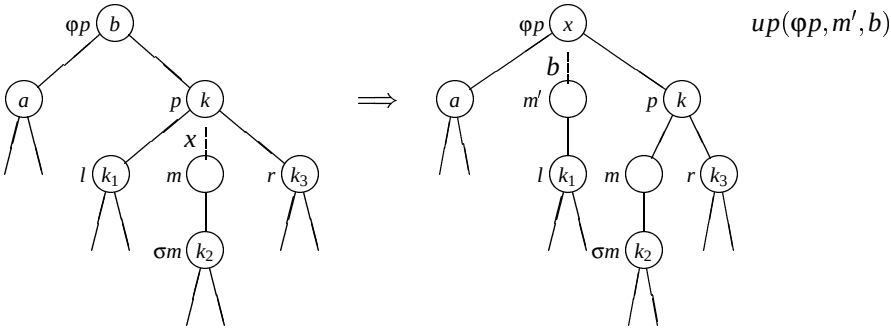
Vor dem ersten Aufruf der Prozedur *up* und vor jedem späteren rekursiven Aufruf gilt die folgende Invariante. Wenn $up(p,m,x)$ aufgerufen wird, gelten (1), (2) und (3):

(1) p hat zwei Söhne p_l und p_r, die beide Wurzeln von 1-2-Bruder-Bäumen sind.

(2) Der Knoten m ist entweder ein Blatt oder hat einen einzigen Sohn, der Wurzel eines 1-2-Bruder-Baumes ist.

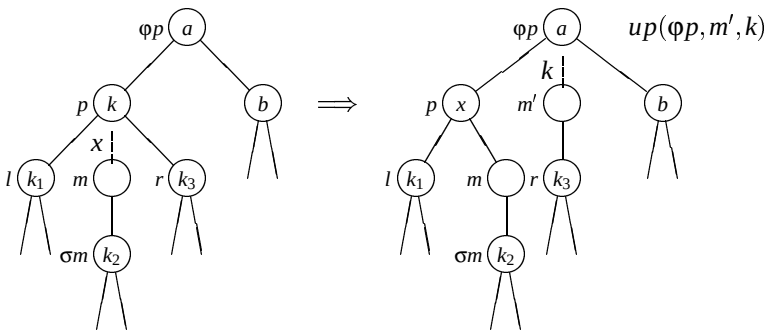
(3) Schlüssel im linken Teilbaum von $p < x$
$<$ Schlüssel im Teilbaum von m
$<$ Schlüssel von p
$<$ Schlüssel im rechten Teilbaum von p

Fall 1 [p hat einen linken Bruder mit zwei Söhnen]

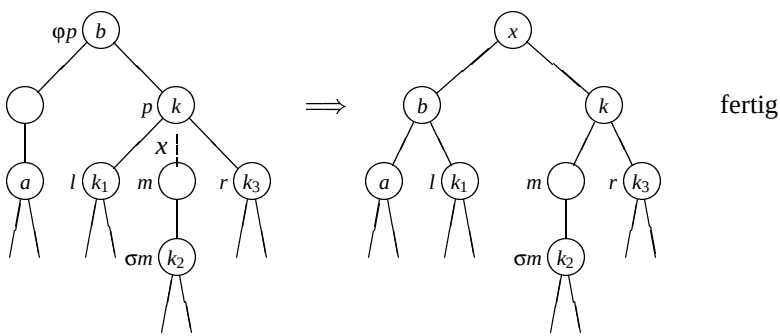


Falls l, m, r Blätter sind, wenn also die Prozedur $up(p, ., .)$ erstmals aufgerufen wird, existiert σm nicht. In diesem Fall muß man natürlich auch die Schlüssel k_1, k_2, k_3 weglassen. Ähnliche Annahmen muß man auch in den folgenden Figuren machen, um den Blattfall abzudecken.

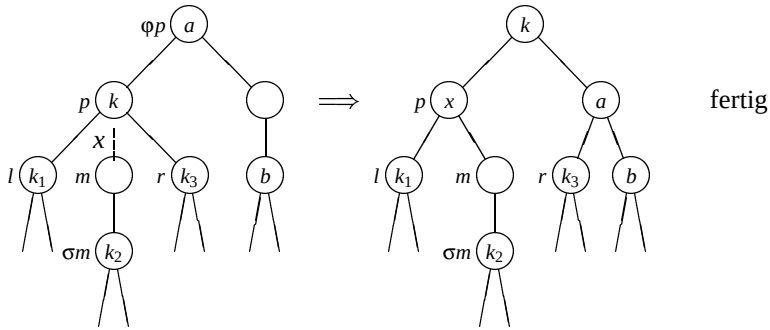
Fall 2 [p hat einen rechten Bruder mit zwei Söhnen]



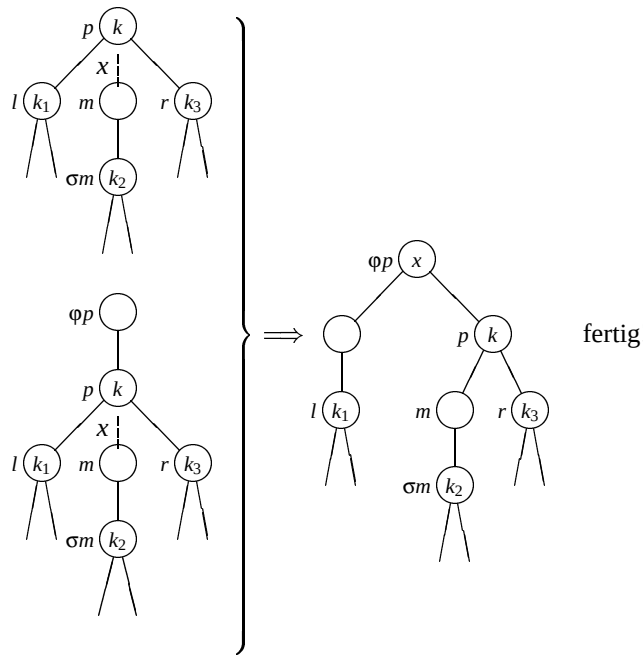
Fall 3 [p hat einen linken Bruder mit nur einem Sohn]



Fall 4 [p hat einen rechten Bruder mit nur einem Sohn]



Fall 5 [p hat keinen Bruder] Dann ist p entweder die Wurzel oder einziger Sohn seines Vaters.



Wir betrachten als Beispiel die Folge der 1-2-Bruder-Bäume, die sich durch iteriertes Einfügen der Schlüssel 1, 2, 3, 4, 5 in aufsteigender Reihenfolge in den anfangs leeren Baum ergibt, vgl. Abbildung 5.28. Weiteres Einfügen der Schlüssel 6 und 7 liefert den vollständigen Binärbaum mit Höhe 3. Durch einen nicht ganz einfachen Induktionsbeweis [33] läßt sich zeigen, daß iteriertes Einfügen von $2^k - 1$ Schlüsseln in aufsteigend sortierter Reihenfolge den vollständigen Binärbaum mit Höhe h liefert. Bruder-Bäume verhalten sich damit gerade entgegengesetzt zu den im Abschnitt 5.5 behandelten B-Bäumen. Iteriertes Einfügen in auf- oder absteigend sortierter Reihenfolge liefert besonders niedrige Bruder-Bäume, aber besonders hohe B-Bäume. In keinem

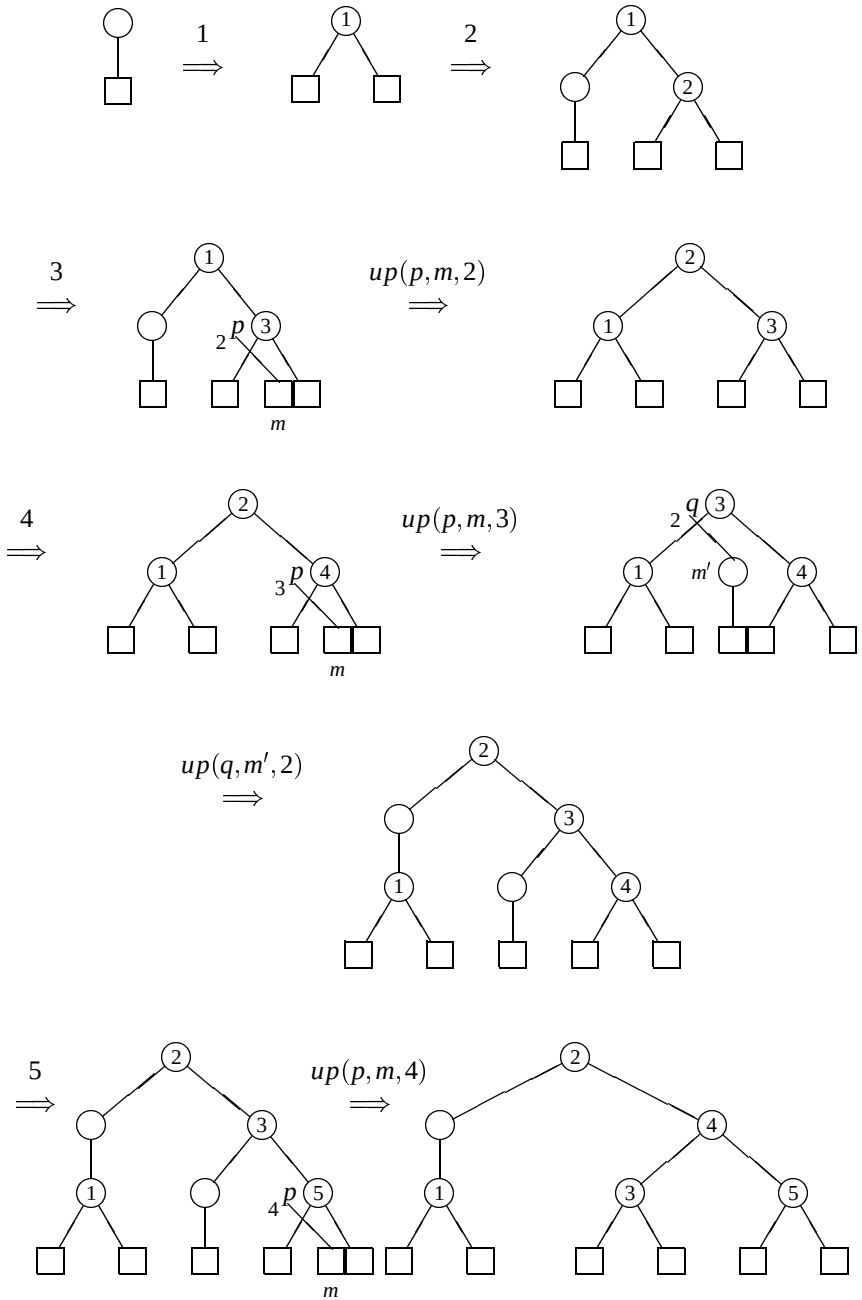


Abbildung 5.28

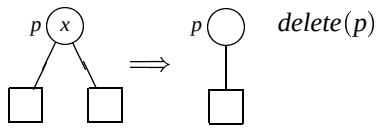
Fall kann die Höhe eines 1-2-Bruder-Baumes, der durch iteriertes Einfügen von $N-1$ Schlüsseln in den anfangs leeren Baum entsteht, größer sein als $1.44\ldots\log_2 N$. Welche Höhe wird man im Mittel erwarten können, wenn man über alle möglichen Anordnungen von Schlüsseln und die ihnen durch iteriertes Einfügen in den anfangs leeren Baum zugeordneten 1-2-Bruder-Bäume mittelt?

Eine Antwort auf diese Frage und andere das mittlere Verhalten des Einfügeverfahrens charakterisierende Eigenschaften werden wir mit Hilfe der Fringe-Analyse-Technik erhalten, die wir am Ende dieses Abschnitts besprechen.

Zunächst sieht man der Prozedur *up* unmittelbar an, daß sie im schlechtesten Fall längs des Suchpfades von der Einfügestelle zurück zur Wurzel aufgerufen werden kann. Damit gilt: Das Einfügen eines neuen Schlüssels in einen 1-2-Bruder-Baum mit N Schlüsseln ist in $O(\log N)$ Schritten ausführbar.

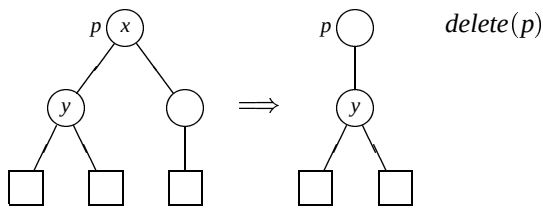
Um einen Schlüssel x aus einem 1-2-Bruder-Baum zu *entfernen*, sucht man zunächst nach x im Baum. Wenn es keinen (binären) Knoten mit Wert x im Baum gibt, ist man bereits fertig. Sonst ist der zu entfernende Schlüssel x der Schlüssel eines binären Knotens p. Wie im Fall binärer Suchbäume oder im Falle von AVL-Bäumen muß man auch hier unter Umständen das Entfernen des Schlüssels von p auf das Entfernen des symmetrischen Nachfolgers reduzieren. Dann kann man ohne Einschränkung annehmen, daß einer der folgenden Fälle vorliegt:

Fall 1 [Die Söhne von p sind Blätter]



Man macht p unär, entfernt den Schlüssel x von p und ruft die weiter unten erklärte Prozedur $delete(p)$ auf.

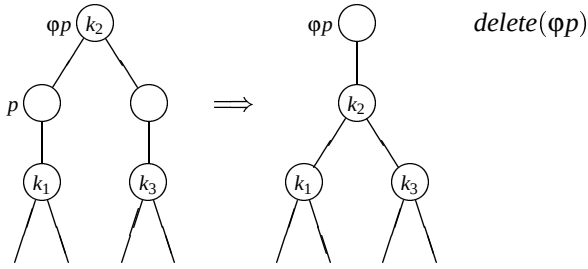
Fall 2 [Der rechte (oder linke) Sohn von p ist unär und hat ein Blatt als einzigen Sohn]



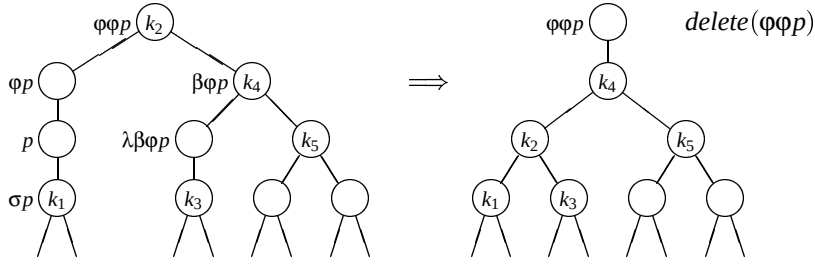
Da ein vorher binärer Knoten p unär gemacht worden ist, kann in der Verwandtschaft von p eine der Bruder-Bäume charakterisierenden Bedingungen verletzt sein. Ein unärer Knoten hat möglicherweise keinen binären Bruder mehr. Die Prozedur *delete* sorgt dafür, daß diese Bedingung wiederhergestellt wird, indem zwei unäre Knoten zu einem binären verschmolzen werden.

Wenn $delete(p)$ aufgerufen wird, gilt die folgende Invariante: Der Knoten p ist unär und der einzige Sohn von p ist die Wurzel eines 1-2-Bruder-Baumes. p hat seinen Schlüssel verloren, außer für p und den Bruder von p, falls es ihn gibt, gilt die Bedingung, daß unäre Knoten binäre Brüder haben.

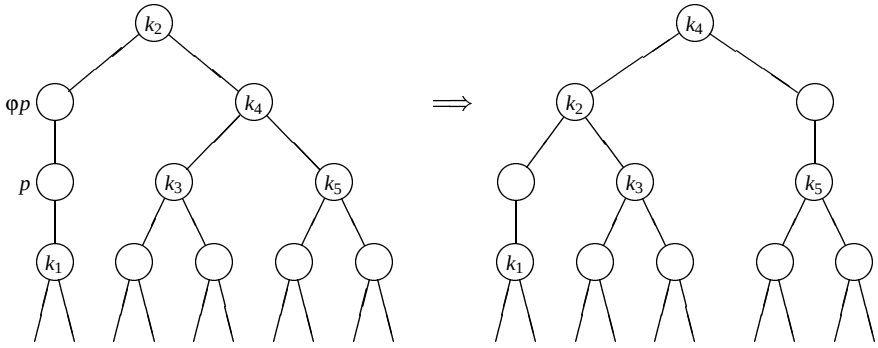
Fall 1 [p hat einen Bruder mit zwei Söhnen]
Dann ist nichts zu tun.
Fall 2 [p hat einen Bruder mit nur einem Sohn]



Der Fall, daß p rechter Sohn seines Vaters ist, wird natürlich genauso behandelt.
Fall 3 [p hat keinen Bruder]
Fall 3.1 [p ist die Wurzel]
Dann entfernt man p, macht den einzigen Sohn von p zur neuen Wurzel und ist fertig.
Fall 3.2 [p ist einziger Sohn seines Vaters φp]
Aufgrund der Invarianten muß φp einen binären Bruder $\beta\varphi p$ haben. Wir machen eine Fallunterscheidung je nachdem, ob $\beta\varphi p$ drei oder vier Enkel hat:
Fall 3.2.1 [Der linke oder der rechte Sohn von $\beta\varphi p$hat nur einen Sohn]
Wir nehmen an, daß φp der linke Sohn seines Vaters ist, und daß der linke Sohn von $\beta\varphi p$ nur einen Sohn hat. Die übrigen, zu diesem Fall symmetrischen Fälle werden analog behandelt.



Fall 3.2.2 [Beide Söhne von $\beta\varphi p$ haben zwei Söhne]
Wir behandeln nur den Fall, daß φp linker Sohn seines Vaters ist, und überlassen den symmetrischen Fall dem Leser.



fertig!

Man sieht der Prozedur *delete* unmittelbar an, daß sie schlechtestenfalls längs eines Pfades von den Blättern zurück zur Wurzel aufgerufen wird. Damit gilt: Das Entfernen eines Schlüssel x aus einem 1-2-Bruder-Baum mit N Schlüsseln ist in $O(\log N)$ Schritten ausführbar.

Wir haben also insgesamt eine weitere Implementationsmöglichkeit für Wörterbücher, die es erlaubt, jede der Operationen Suchen, Einfügen und Entfernen eines Schlüssels auch im schlechtesten Fall in $O(\log N)$ Schritten auszuführen.

Analytische Betrachtungen

1-2-Bruder-Bäume enthalten im allgemeinen unäre Knoten, die keine Schlüssel speichern. Wieviele können das sein? Wir diskutieren diese Frage zunächst im statischen Fall: D.h. wir betrachten einen beliebigen 1-2-Bruder-Baum und setzen nichts über seine Entstehungsgeschichte voraus. Dann untersuchen wir dieselbe Frage im dynamischen Fall: D.h. wir schätzen die Anzahl der unären Knoten in einem 1-2-Bruder-Baum ab, der aus dem anfangs leeren Baum durch eine Folge von N zufälligen Einfügungen entsteht.

Die Analyse des statischen Falls ist einfach. Wir betrachten zwei beliebige benachbarte Niveaus im Baum und sehen, daß nur die Knotenkonfigurationen aus Abbildung 5.29 möglich sind.

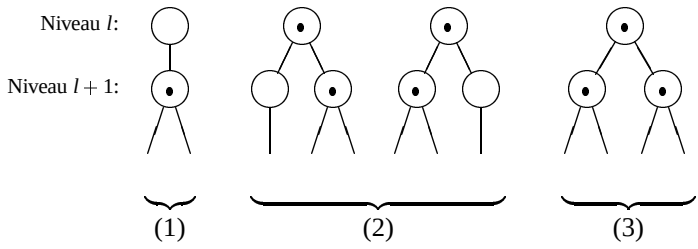


Abbildung 5.29

Für jeden unären Knoten auf Niveau l muß es einen binären Bruder auf demselben Niveau geben. Daher gilt für das Verhältnis

$$U = \frac{\text{Anzahl binäre Knoten auf Niveau } l \text{ und } l+1}{\text{Anzahl Knoten insgesamt auf Niveau } l \text{ und } l+1} :$$

Konfiguration	U
(2)	$\frac{2}{3}$
(3)	$\frac{3}{3}$
(1) und eine Konfig. aus (2)	$\frac{3}{5}$
(1) und (3)	$\frac{4}{5}$

Folglich ist $\frac{3}{5} \leq U \leq 1$.

Was für je zwei beliebige benachbarte Niveaus gilt, muß auch für einen 1-2-Bruder-Baum insgesamt gelten. Damit gilt: Wenigstens $3/5$ der inneren Knoten eines 1-2-Bruder-Baumes müssen binär sein und speichern also einen Schlüssel. Ein 1-2-Bruder-Baum mit N Schlüsseln hat daher höchstens $\frac{5}{3}N$ innere (unäre und binäre) Knoten.

Aus dieser einfachen Beobachtung kann man bereits eine wichtige Folgerung für den über eine Folge iterierter Einfügungen gemittelten *mittleren* Aufwand zum Einfügen eines Schlüssels ziehen. Eine Inspektion der aufwärts umstrukturierenden Prozedur *up* zeigt, daß jeder Aufruf dieser Prozedur zur Schaffung eines oder höchstens zweier Knoten führt. Beim ersten Aufruf wird ein zusätzliches Blatt erzeugt. Jeder weitere Aufruf für einen Knoten, der verschieden von der Wurzel ist, erzeugt genau einen weiteren (unären) Knoten. Ein Aufruf von *up* für die Wurzel erzeugt einen unären und einen binären Knoten. Das sind auch bereits alle Möglichkeiten, wie neue Knoten erzeugt werden können. Sonst werden höchstens vorher unäre Knoten binär gemacht, und die Umstrukturierung mit Hilfe von *up* endet. Fügt man also N Schlüssel in den anfangs leeren Baum ein, so kann man aus der insgesamt erzeugten Knotenzahl auf die insgesamt ausgeführten Aufrufe von *up* schließen. Da höchstens $\frac{5}{3}N$ innere Knoten und ebensoviele Blätter insgesamt erzeugt worden sind, ist die durchschnittliche Anzahl der Aufrufe von *up* pro Einfügung konstant. Zählt man den Suchaufwand zum Finden der jeweiligen Einfügestelle nicht mit, so folgt: Der durchschnittliche Aufwand zum Einfügen eines Schlüssels in einen 1-2-Bruder-Baum ist konstant, wenn man den Durchschnitt über eine Folge von Einfügungen in den anfangs leeren Baum nimmt.

Eine entsprechende Aussage ist für AVL-Bäume übrigens bei weitem nicht so leicht herzuleiten. Denn es ist zwar richtig, daß für einen AVL-Baum nach dem Einfügen eines neuen Schlüssels höchstens eine einzige Rotation oder Doppelrotation ausgeführt werden muß; zu den Umstrukturierungen muß man aber auch das Adjustieren der Balancefaktoren hinzurechnen, das an jedem Knoten längs des Suchpfades erforderlich sein kann.

Wir kommen jetzt zum dynamischen Fall und wollen den Erwartungswert für die Anzahl der unären und binären Knoten ausrechnen, wenn man eine zufällig gewählte Folge von N Schlüsseln in den anfangs leeren 1-2-Bruder-Baum einfügt. Genau werden wir diese Werte nur für den *Rand* (englisch: *fringe*), d.h. für die Knoten auf den blattnahen Niveaus ausrechnen. Die dafür von A.Yao [196] entwickelte Methode heißt daher auch *Fringe-Analyse*. Sie ist nicht nur auf 1-2-Bruder-Bäume, sondern auch auf viele andere Baumklassen anwendbar.

Wir begnügen uns damit, die Anzahl der binären Knoten auf den zwei untersten, den Blättern nächsten Niveaus innerer Knoten [illegible]erechnen, für einen 1-2-Bruder-Baum, der durch eine Folge von N zufälligen Einfügungen in den anfangs leeren 1-2-Bruder-Baum entsteht. Dazu schauen wir uns zunächst einmal an, welche Teilbäume mit niedriger Höhe 1 oder 2 am Rand eines 1-2-Bruder-Baumes auftreten können. Es gibt offenbar die in Abbildung 5.30 dargestellten Möglichkeiten.

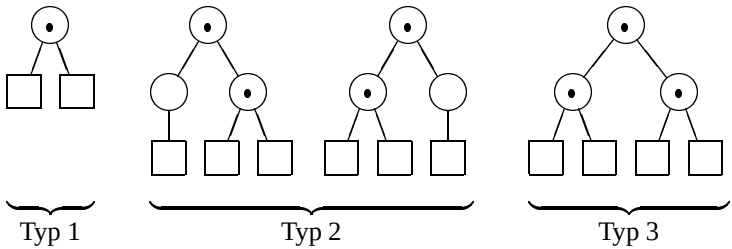


Abbildung 5.30

Sei T ein 1-2-Bruder-Baum. Wir sagen: T gehört zur Klasse (x_1, x_2, x_3), wenn T x_i Teilbäume vom Typ i hat $(1 \leq i \leq 3)$. Dabei darf kein Teilbaum doppelt gezählt werden, d.h. die Anzahl der Blätter von T muß gleich $2x_1 + 3x_2 + 4x_3$ sein. Derselbe 1-2-Bruder-Baum kann aber durchaus zu mehreren Klassen gehören.

Sei nun ein 1-2-Bruder-Baum mit $N-1$ Schlüsseln und N Blättern gegeben. Dann sagen wir: Das Einfügen des N-ten Schlüssels x erfolgt zufällig, wenn die Wahrscheinlichkeit dafür, daß x in eines der durch die bereits vorhandenen Schlüssel bestimmten N Schlüsselintervalle fällt, für jedes dieser Intervalle gleich groß ist, nämlich $1/N$. Die Wahrscheinlichkeit dafür, daß x in einen Teilbaum vom Typ i fällt, ist damit gleich dem Anteil, den die Blätter von Teilbäumen vom Typ i zur gesamten Blattzahl beisteuern; sie ist also $(i+1) \cdot \frac{x_i}{N}$ für jedes i, $1 \leq i \leq 3$.

Beispiel: Der 1-2-Bruder-Baum aus Abbildung 5.31 gehört zur Klasse (2, 1, 0) und (0, 1, 1).

Sei nun $A_i(N)$ der Erwartungswert für die Anzahl von Teilbäumen des Typs i nach N zufälligen Einfügungen in den anfangs leeren Baum. Für kleine Werte von N kann man $A_i(N)$ leicht explizit ausrechnen, weil es nicht schwer ist, sich eine vollständige Übersicht über alle durch iteriertes Einfügen entstehenden 1-2-Bruder-Bäume zu verschaffen. Beispielsweise entsteht nach vier Einfügungen stets, d.h. mit Wahrscheinlich-

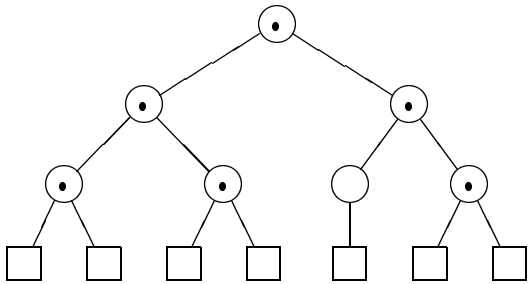


Abbildung 5.31

keit 1, der Baum in Abbildung 5.32. Tabelle 5.2 enthält mögliche Werte von $A_i(N)$ für $N = 1, \ldots, 6$.

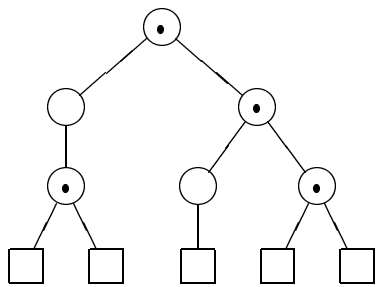


Abbildung 5.32

Zur Berechnung von $A_i(N)$ für beliebige N benutzen wir die folgenden Hilfssätze:

Lemma 5.1 *Sei T ein 1-2-Bruder-Baum. Wird ein neuer Schlüssel in einen Teilbaum des Typs 1 (bzw. des Typs 2) von T eingefügt, so erhöht sich die Zahl der Teilbäume vom Typ 2 (bzw. 3) um 1 und die Zahl der Teilbäume vom Typ 1 (bzw. 2) erniedrigt sich um 1.*

Beweis: Wir beschränken uns auf die erste Aussage: Die Wurzel eines Teilbaumes vom Typ 1 ist entweder einziger Sohn eines unären Vaters oder hat einen binären Bruder. Damit folgt die Behauptung aus der Definition des Einfügeverfahrens. □

Genauso einfach zeigt man:

Lemma 5.2 *Sei T ein 1-2-Bruder-Baum. Wird ein neuer Schlüssel in einen Teilbaum vom Typ 3 von T eingefügt, so erhöht sich die Zahl der Teilbäume vom Typ 1 und 2 jeweils um 1 und die Zahl der Teilbäume vom Typ 3 erniedrigt sich um 1.*

N	$A_1(N)$	$A_2(N)$	$A_3(N)$
1	1	0	0
2	0	1	0
3	0	0	1
4	1	1	0
5	$\frac{3}{5}$	$\frac{4}{5}$	$\frac{3}{5}$
6	$\begin{cases} 0 \\ \frac{4}{5} \end{cases}$	$\begin{cases} 1 \\ 1 \end{cases}$	$\begin{cases} 1 \\ \frac{3}{5} \end{cases}$

Tabelle 5.2

Ist also T ein 1-2-Bruder-Baum mit $N-1$ Schlüsseln der Klasse (x_1, x_2, x_3), so wird aus T mit Wahrscheinlichkeit p ein Teilbaum der Klasse (x'_1, x'_2, x'_3) mit folgenden Werten für x'_i und p:

x'_1	x'_2	x'_3	p
$x_1 - 1$	$x_2 + 1$	x_3	$2 \cdot \frac{x_1}{N}$
x_1	$x_2 - 1$	$x_3 + 1$	$3 \cdot \frac{x_2}{N}$
$x_1 + 1$	$x_2 + 1$	$x_3 - 1$	$4 \cdot \frac{x_3}{N}$

$\Big\}\ \Sigma = 1$

$A_1(N-1)$ nimmt also mit Wahrscheinlichkeit $2 \cdot \frac{A_1(N-1)}{N}$ um 1 ab und nimmt mit Wahrscheinlichkeit $4 \cdot \frac{A_3(N-1)}{N}$ um 1 zu, d.h. es gilt:

$$A_1(N) = A_1(N-1) - \frac{2}{N} A_1(N-1) + \frac{4}{N} A_3(N-1)$$

Analog gilt:

$$\begin{aligned} A_2(N) &= A_2(N-1) - \frac{3}{N} A_2(N-1) + (1 - \frac{3}{N} A_2(N-1)) \\ &= (1 - \frac{6}{N}) A_2(N-1) + 1 \\ A_3(N) &= A_3(N-1) + \frac{3}{N} A_2(N-1) - \frac{4}{N} A_3(N-1) \\ &= (1 - \frac{4}{N}) A_3(N-1) + \frac{3}{N} A_2(N-1) \end{aligned}$$

Durch vollständige Induktion zeigt man leicht, daß dieses System von Rekursionsgleichungen mit den oben angegebenen Anfangsbedingungen folgende Lösung hat:

$$\left.\begin{array}{lcl} A_1(N) & = & \frac{4}{7\cdot 5}(N+1) \\ A_2(N) & = & \frac{1}{7}(N+1) \\ A_3(N) & = & \frac{3}{7\cdot 5}(N+1) \end{array}\right\} \quad \text{für} \quad N \geq 6.$$

Wir nennen einen 1-2-Bruder-Baum *zufällig*, wenn er durch eine Folge zufälliger Einfügungen in den anfangs leeren Baum entsteht.

Als *untere Schranke* für die Anzahl der Schlüssel auf den zwei untersten Niveaus innerer Knoten in zufälligen 1-2-Bruder-Bäumen mit N Schlüsseln erhalten wir:

$$1 \cdot A_1(N) + 2 \cdot A_2(N) + 3 \cdot A_3(N) = \frac{23}{35}(N+1) = 0.657\ldots(N+1)$$

Da ungünstigstenfalls jeder Typ-1-Teilbaum einen unären Vater hat, erhalten wir als *obere Schranke* für die Gesamtzahl der inneren Knoten auf den zwei untersten Niveaus:

$$2A_1(N) + 3(A_2(N) + A_3(N)) = \frac{32}{35}(N+1)$$

Für die zwei untersten Niveaus eines zufällig erzeugten 1-2-Bruder-Baumes ist also das Verhältnis der Anzahl der binären Knoten zur Gesamtzahl der Knoten auf diesen Niveaus wenigstens $\frac{23}{32} = 0.71875$. Wir können demnach erwarten, daß wenigstens 23 von 32 Knoten binär sind und nicht nur 3 von 5, wie unsere statische Abschätzung ergeben hat.

Eine genauere Abschätzung für das Verhältnis der Zahl der binären zur Gesamtzahl von Knoten auf den zwei untersten Niveaus ist nur eine mögliche Folgerung, die man aus der Berechnung der Erwartungswerte $A_i(N)$ für die Anzahl der Teilbäume vom Typ i in einem zufällig erzeugten 1-2-Bruder-Baum ziehen kann.

Da in einem Binärbaum etwa die Hälfte der inneren Knoten unmittelbar oberhalb der Blätter vorkommt, kann man über die Erwartungswerte für die Anzahl der binären und unären Knoten auf den zwei untersten Niveaus auch bessere Schranken für die entsprechenden Anzahlen im gesamten Baum erhalten. Man schätzt diese Zahl auf den untersten Niveaus wie oben angegeben ab und benutzt oberhalb die aus der statischen Betrachtung gewonnene Abschätzung.

Weiter liefern die Erwartungswerte $A_i(N)$, für $i = 1,2,3$, auch eine Aussage darüber, wie groß die Wahrscheinlichkeit dafür wenigstens ist, daß eine weitere Einfügung in einen zufällig erzeugten 1-2-Bruder-Baum zu höchstens einem bzw. mindestens zwei (rekursiven) Aufrufen der Prozedur *up* führt. Fällt nämlich der nächste einzufügende Schlüssel in einen Teilbaum des Typs 2, so wird *up* genau einmal, fällt sie in einen Teilbaum des Typs 3, so wird *up* wenigstens zweimal aufgerufen.

Das sind einige Beispiele für Aussagen, die mit Hilfe der Fringe-Analyse-Methode hergeleitet werden können. Die Methode führt im allgemeinen nicht zu so einfach elementar lösbaren Rekursionsgleichungen wie für die Erwartungswerte $A_i(N)$ im Falle von 1-2-Bruder-Bäumen. Man muß vielmehr im allgemeinen stärkere mathematische Hilfsmittel heranziehen, um die Erwartungswerte für Teilbäume, die im Rand zufällig erzeugter Bäume auftreten, zu berechnen. Das ist z.B. erforderlich, wenn man die im Abschnitt 5.5 behandelten B-Bäume mit dieser Methode analysiert.

5.2.3 Gewichtsbalancierte Bäume

Balancierte Binärbäume sind ganz grob dadurch charakterisiert, daß für jeden Knoten p der linke und rechte Teilbaum von p nicht zu unterschiedliche Größe haben dürfen. Die Größe kann dabei, wie im Falle der AVL-Bäume, durch die Höhe oder — und das ist der in diesem Abschnitt diskutierte Fall — über die Anzahl der Knoten bzw. Blätter bestimmt sein. Bei gewichtsbalancierten Bäumen wird gefordert, daß die Anzahl der Knoten bzw. Blätter im linken und rechten Teilbaum eines jeden Knotens sich nicht zu stark unterscheiden dürfen [131, 128]. Wir wissen bereits, daß für jeden Binärbaum die Anzahl der Blätter stets um 1 größer ist als die Anzahl der binären inneren Knoten.

Wir wollen für einen Knoten p eines Binärbaumes, der Wurzel eines Teilbaumes T_p ist, mit $W(p)$ und $W(T_p)$ die Anzahl der Blätter des Teilbaumes T_p bezeichnen; $W(p)$ und $W(T_p)$ nennt man üblicherweise auch das Gewicht (englisch: weight) von p bzw. von T_p.

Ist T ein Baum mit $W(T)$ Blättern, dessen linker Teilbaum T_l $W(T_l)$ Blätter hat, so nennt man den Quotienten

$$\rho(T) = \frac{W(T_l)}{W(T)}$$

die *Wurzelbalance* von T. Man fordert nun, daß die Wurzelbalance für jeden Teilbaum innerhalb bestimmter Grenzen liegen muß. Ist α eine Zahl mit $0 \leq \alpha \leq \frac{1}{2}$, so heißt ein binärer Suchbaum T *von beschränkter Balance* α oder *gewichtsbalanciert mit Balance* α oder kurz ein BB[α]-Baum, wenn für jeden Teilbaum T' von T gilt:

$$\alpha \leq \rho(T') \leq (1-\alpha) \qquad (*)$$

Durch diese Forderung ist natürlich nicht nur das Verhältnis der Knotenzahlen im linken Teilbaum eines jeden Knotens zur gesamten Knotenzahl im Teilbaum dieses Knotens festgelegt. Denn ist p ein Knoten mit linkem Sohn p_l und rechtem Sohn p_r, so ist natürlich

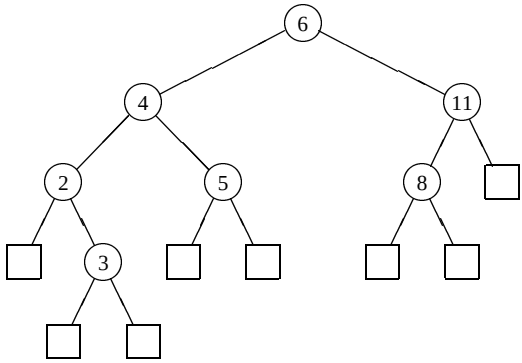
$$W(p_r) = W(p) - W(p_l)$$

und daher gilt mit $(*)$ nicht nur, daß für jeden Knoten p eines BB[α]-Baumes

$$\alpha \leq \frac{W(p_l)}{W(p)} \leq (1-\alpha)$$

ist, sondern auch

$$\alpha \leq 1 - \frac{W(p_r)}{W(p)} \leq (1-\alpha). \qquad (**)$$



Wurzelbalancen:	
Knoten mit Schlüssel	Balance
6	$\frac{5}{8}$
4	$\frac{3}{5}$
11	$\frac{2}{3}$
2	$\frac{1}{3}$
5	$\frac{1}{2}$
3	$\frac{1}{2}$
8	$\frac{1}{2}$

Abbildung 5.33

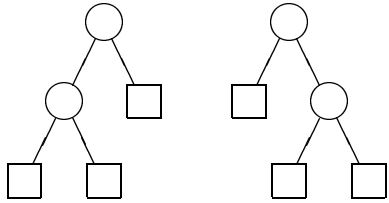


Abbildung 5.34

Als Beispiel betrachte man Abbildung 5.33. Offenbar gilt für $\alpha = \frac{1}{4}$, daß alle Wurzelbalancen zwischen $1/4$ und $3/4$ liegen. Der Baum ist damit ein BB[$\frac{1}{4}$]-Baum.

Über den Parameter α läßt sich die Güte der Ausgeglichenheit steuern. Je näher α bei 0 liegt, um so weniger restriktiv ist die Forderung der Gewichtsbalanciertheit; je näher α bei $1/2$ liegt, um so besser ausgeglichen müssen die Bäume in BB[α] sein. Man kann aber α nicht gleich $1/2$ setzen oder auch nur beliebig nahe an den Wert $1/2$ herankommen lassen, weil dann die Forderung $(*)$ so restriktiv ist, daß nicht mehr für jede Knotenzahl N ein Baum existiert, der in BB[α] liegt. So gibt es beispielsweise nur zwei Suchbäume mit zwei inneren Knoten, wie in Abbildung 5.34 dargestellt wird. Die Wurzelbalance des linken Baumes ist $2/3$ und die des rechten ist $1/3$. Beide Bäume liegen in BB[$\frac{1}{3}$], aber BB[$\frac{1}{2}$] enthält keinen Baum mit 2 inneren Knoten.

Wir setzen im folgenden voraus, daß α stets so gewählt ist, daß in BB[α] wenigstens ein Baum mit N Knoten für jedes N liegt. (Wählt man $\alpha \in [\frac{1}{4}, 1 - \frac{\sqrt{2}}{2}]$, so gilt die Bedingung; vgl. hierzu [131] oder [123].)

Der Aufwand zur Ausführung der für Suchbäume typischen Operationen Suchen, Einfügen und Entfernen hängt unmittelbar von der Höhe der jeweils betrachteten Bäume ab. Wir wollen uns daher zunächst überlegen, daß die über die Knotengewichte definierte Balancebedingung impliziert, daß gewichtsbalancierte Bäume eine Höhe haben, die logarithmisch von der Anzahl der Knoten abhängt. Gewichtsbalancierte Bäume sind dadurch charakterisiert, daß man beim Hinabsteigen von einem Knoten p zu einem seiner Söhne stets einen bestimmten Teil der Blätter verliert, der durch den Balancefaktor α bestimmt ist. Genauer: Ist p ein Knoten mit linkem Sohn p_l und rechtem Sohn p_r, so folgt aus $(*)$ (und $(**)$):

(i) $W(p_l) \leq (1-\alpha)W(p)$

(ii) $W(p_r) \leq (1-\alpha)W(p)$

Bemerkung: Eine analoge Bedingung gilt weder für höhenbalancierte Bäume noch für Bruder-Bäume. Wenn man beispielsweise einen Bruder-Baum T betrachtet, dessen Wurzel als linken Teilbaum T_l einen "Fibonacci-Baum" mit Höhe h und F_{h+1} Blättern hat und als rechten Teilbaum T_r einen vollständigen Binärbaum mit derselben Höhe, so gilt:

$$W(T_l) = F_{h+1} = c \cdot 1.618\ldots^h$$

mit einer Konstanten c und

$$W(T) = c \cdot 1.618\cdots^h + 2^h.$$

Nehmen wir nun an, es gibt ein α, $0 < \alpha < 1$, so daß $W(T_r) \leq (1-\alpha)W(T)$. Dann folgt aus (ii)

$$2^h \leq (1-\alpha)(c \cdot 1.618\cdots^h + 2^h)$$

und damit

$$\frac{1}{1-\alpha} \leq (1 + c \cdot (\frac{1.618\cdots}{2})^h).$$

Weil $\alpha < 1$ ist, muß $1/(1-\alpha) > 1$ sein. Man erhält also einen Widerspruch, da $(1.618\cdots/2)^h$ mit wachsendem h gegen 0 geht. □

Sei nun ein gewichtsbalancierter Baum T aus BB[α] mit Höhe h gegeben. Wir betrachten einen Pfad maximaler Länge von der Wurzel zu einem Blatt. Seien $p_1, p_2, \ldots, p_h$ die (inneren) Knoten auf diesem Pfad. Der Knoten p_1 ist also die Wurzel und p_h ist ein Knoten, dessen beide Söhne Blätter sind. Daher ist

$$W(T) = W(p_1) \quad \text{und} \quad W(p_h) = 2.$$

Wegen (i) und (ii) gilt:

$$\begin{aligned} W(p_2) &\leq (1-\alpha)W(p_1) \\ W(p_3) &\leq (1-\alpha)W(p_2) \\ &\vdots \\ W(p_h) &\leq (1-\alpha)W(p_{h-1}) \end{aligned}$$

Also

$$2 \leq (1-\alpha)^{h-1} \cdot W(p_1) = (1-\alpha)^{h-1} \cdot N,$$

wenn $N = W(p_1)$ die Anzahl der Blätter des Baumes T bezeichnet. Durch Logarithmieren dieser Ungleichung erhält man

$$1 \leq (h-1)\log_2(1-\alpha) + \log_2 N,$$

also

$$h - 1 \leq \frac{\log_2 N - 1}{-\log_2(1-\alpha)} = O(\log N).$$

Die Höhe h eines Baumes aus BB[α] ist also logarithmisch in der Anzahl der Blätter oder Knoten beschränkt.

Suchen, Einfügen und Entfernen von Schlüsseln

Da gewichtsbalancierte Bäume insbesondere binäre Suchbäume sind, kann man in ihnen genauso suchen wie in natürlichen Bäumen. Weil die Höhe eines Baumes aus BB[α] mit N Knoten von der Größenordnung $O(\log N)$ ist, kann man die Operation *Suchen* ebenfalls stets in $O(\log N)$ Schritten ausführen.

Um einen Schlüssel in einen Baum T aus BB[α] *einzufügen*, sucht man zunächst nach dem einzufügenden Schlüssel im Baum. Wenn der Schlüssel in T noch nicht vorkommt, endet die Suche erfolglos in einem Blatt, das die erwartete Position des einzufügenden Schlüssel repräsentiert. Wie bei natürlichen Bäumen ersetzt man dieses Blatt durch einen inneren Knoten, der den neu einzufügenden Schlüssel aufnimmt, und gibt ihm zwei Blätter als Söhne. Der resultierende Baum ist damit zwar wieder ein Suchbaum, aber möglicherweise kein gewichtsbalancierter Baum aus BB[α] mehr. Denn man hat ja durch Schaffen eines weiteren inneren Knotens und eines neuen Blattes die Gewichte aller Knoten auf dem Pfad von der Wurzel zur Einfügestelle verändert. Beim *Entfernen* eines Schlüssels tritt eine ähnliche Situation ein. Man entfernt einen Schlüssel zunächst genauso, wie man es von natürlichen Bäumen kennt. Man reduziert das Entfernen also gegebenenfalls auf das Entfernen des symmetrischen Nachfolgers oder Vorgängers eines Knotens und kann daher ohne Einschränkung annehmen, daß man den Schlüssel

eines Knotens entfernt, dessen beide Söhne Blätter sind. Ersetzt man nun diesen Knoten durch ein Blatt, so haben sich wieder die Gewichte aller Knoten auf dem Pfad von der Wurzel bis zur Entfernestelle verändert. Man muß also unter Umständen den Baum umstrukturieren, um wieder einen BB[α]-Baum zu erhalten. Dazu geht man ähnlich vor wie bei AVL-Bäumen. Man läuft den Suchpfad zurück und prüft an jedem Knoten, ob die Wurzelbalance an diesem Knoten noch im Bereich $[\alpha, 1-\alpha]$ liegt. Ist das nicht der Fall, führt man eine Rotation oder Doppelrotation durch, um die Wurzelbalance an dieser Stelle wieder in den vorgeschriebenen Bereich zurückzubringen.

Hier stellt sich natürlich zunächst die Frage, wie man denn überhaupt erkennen kann, ob an einem bestimmten Knoten die Wurzelbalance noch im vorgeschriebenen Bereich liegt. Darüber hinaus muß man natürlich zeigen, daß Rotationen und Doppelrotationen wirklich geeignete Maßnahmen sind, um die Wurzelbalance an einem bestimmten Knoten in den verlangten Bereich zurückzuführen.

Wir führen an jedem Knoten dessen *Gewicht* (weight) als zusätzliches Attribut mit. Die Knotengewichte kann man bei jeder Einfüge- und Entferne-Operation leicht ändern; notwendige Änderungen bleiben auf den Suchpfad beschränkt. Aus den Gewichten kann man die benötigten Wurzelbalancen leicht berechnen. Das Knotenformat von BB[α]-Bäumen kann man in Pascal etwa wie folgt vereinbaren:

```
type
  Knotenzeiger = ↑Knoten;
  Knoten = record
             key : integer;
             leftson, rightson : Knotenzeiger;
             weight : integer;
             info : {infotype}
           end
```

Gegenüber AVL-Bäumen und Bruder-Bäumen muß man also im Falle gewichtsbalancierter Bäume an jedem Knoten eine im Prinzip unbeschränkt große Information mitführen, die zur Überprüfung und Sicherung der Ausgeglichenheit herangezogen wird. Das ist natürlich ein Nachteil, wenn es auf eine besonders Speicherplatz sparende Implementation einer Klasse balancierter Bäume ankommt.

Nach dem Einfügen oder Entfernen eines Schlüssels läuft man nun auf dem Suchpfad zur Wurzel zurück und überprüft an jedem Knoten die Wurzelbalance des zugehörigen Teilbaumes.

Liegt die Wurzelbalance $\rho(T_p)$ des Teilbaumes mit Wurzel p außerhalb des Bereiches $[\alpha, 1-\alpha]$, sind zwei Fälle möglich:

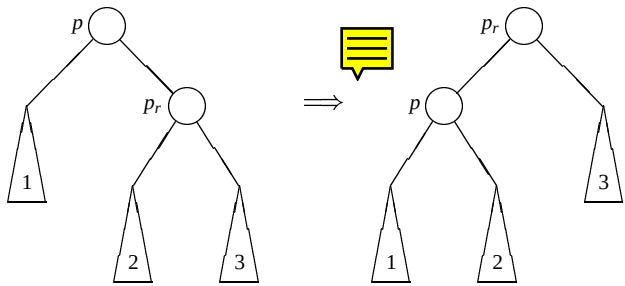
Fall 1: $\rho(T_p) < \alpha$

Fall 2: $\rho(T_p) > (1-\alpha)$

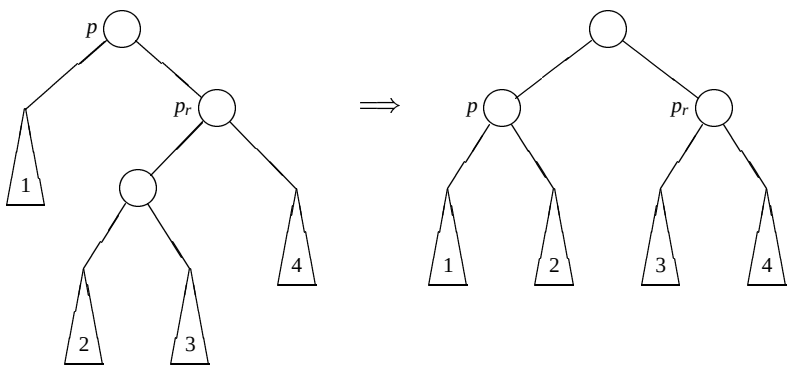
Betrachten wir zunächst den Fall 1 etwas genauer. Die Bedingung $\rho(T_p) < \alpha$ bedeutet, daß der rechte Teilbaum gegenüber dem linken zu schwer geworden ist, und zwar entweder, weil im rechten Teilbaum ein Knoten (und ein Blatt) eingefügt wurde oder weil im linken Teilbaum ein Knoten entfernt wurde. Um die Wurzelbalance bei p wieder in den Bereich $[\alpha, 1-\alpha]$ zurückzubringen, müssen wir den rechten Sohn p_r von p leichter machen. Wie im Falle von AVL-Bäumen versuchen wir das mit Hilfe

einer Rotation nach links oder einer Doppelrotation rechts-links. Welche dieser Operationen gewählt werden muß, hängt ab vom Balancefaktor α und vom Wert der Wurzelbalance von p_r. Man kann zeigen (vgl. z.B. [123]), daß es eine von α abhängige Zahl $d \in [\alpha, 1-\alpha]$ gibt, derart, daß eine Umstrukturierung entsprechend der folgenden Fallunterscheidung auf jeden Fall die Wurzelbalance in den Bereich $[\alpha, 1-\alpha]$ zurückführt, wenn α im Bereich $[\frac{1}{4}, 1-\frac{\sqrt{2}}{2}]$ liegt.

Fall 1.1 $[\rho(T_{p_r}) \leq d]$ Ausgleichen durch einfache Rotation nach links



Fall 1.2 $[\rho(T_{p_r}) > d]$ Ausgleichen durch Doppelrotation rechts-links



Wir betrachten als Beispiel den Baum mit den vier Schlüsseln $\{2, 5, 6, 8\}$ aus BB$[\frac{2}{7}]$ in Abbildung 5.35.

Eine Überprüfung der Wurzelbalancen nach Einfügen des Schlüssels 9 zeigt, daß die Wurzelbalance beim Knoten p nicht mehr im vorgeschriebenen Bereich $[\frac{2}{7}, \frac{5}{7}]$ liegt. Eine Rotation bei p genügt, um beim Knoten p die Wurzelbalance in den vorgeschriebenen Bereich zurückzuführen.

Fügen wir in den Baum aus BB$[\frac{1}{4}]$ in Abbildung 5.36 den Schlüssel 2 ein, so genügt eine einfache Rotation nach links an der Wurzel des neuen Baumes nicht mehr, um die Wurzelbalance dort in den Bereich $[\frac{1}{4}, \frac{3}{4}]$ zurückzuführen. Eine Doppelrotation leistet dies aber.

Bisher haben wir nur den Fall 1 betrachtet; er kann eintreten, wenn ein Knoten auf der rechten Seite von p eingefügt oder auf der linken Seite von p entfernt wurde. Der Fall 2, $\rho(T_p) > (1-\alpha)$, kann eintreten, wenn in einem zuvor ausgeglichenen Baum entweder links ein Knoten eingefügt oder rechts einer entfernt wurde. Dann wird in Abhängigkeit

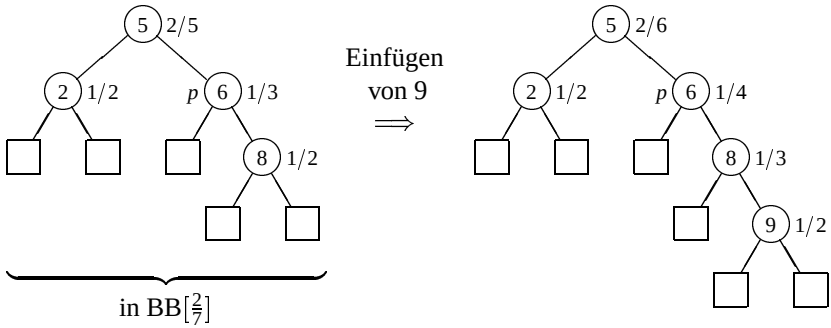


Abbildung 5.35

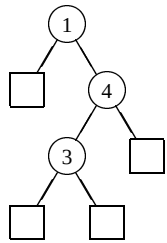


Abbildung 5.36

von einem geeignet gewählten Wert $d \in [\alpha, 1-\alpha]$ eine Rotation nach rechts oder eine Doppelrotation links-rechts ausgeführt, die dafür sorgt, daß die Wurzelbalance bei p in den vorgeschriebenen Bereich zurückkehrt.

Der Nachweis, daß nach einer Rotation oder Doppelrotation die Wurzelbalance bei einem Knoten p wieder im vorgeschriebenen Bereich liegt, ist technisch umständlich, aber nicht schwierig. Er verläuft im Prinzip so: Man berechnet die Wurzelbalancen der transformierten Bäume aus den ursprünglichen Wurzelbalancen. Weil man weiß, daß die ursprünglichen Wurzelbalancen im Bereich $[\alpha, 1-\alpha]$ lagen, erhält man automatisch Schranken für die Wurzelbalancen der transformierten Bäume; man muß sich dann nur noch davon überzeugen, daß die letzteren im vorgeschriebenen Bereich liegen. Dieser Nachweis gelingt allerdings nur, wenn $\alpha \in [\frac{1}{4}, 1-\frac{\sqrt{2}}{2}]$ ist.

Wir verzichten auf die Ausführung der Details und fassen nur das Ergebnis noch einmal zusammen. Gewichtsbalancierte Bäume sind eine Möglichkeit zur Implementierung von Wörterbüchern, die es erlaubt, jede der Operationen Suchen, Einfügen und Entfernen von Schlüsseln auch im schlechtesten Fall in $O(\log N)$ Schritten auszuführen.

Die über eine Anzahl iterierter Einfüge- und Entferne-Operationen gemittelte Anzahl von Rotationen und Doppelrotationen, die erforderlich ist, um stets Bäume in BB[α] zu erhalten, ist konstant, obwohl im schlechtesten Fall eine einzelne Einfüge- oder Entferne-Operation durchaus längs sämtlicher Knoten des Suchpfades, also $\Omega(h),\ h =$

Höhe des Baumes, viele Rotationen und Doppelrotationen auslösen kann. Auch dieses Ergebnis wollen wir hier nicht beweisen, sondern verweisen dazu auf [123].

5.3 Randomisierte Suchbäume

Fügt man N Schlüssel der Reihe nach in einen anfangs leeren binären Suchbaum ein, so kann, wie wir in Abschnitt 5.1 gesehen haben, ein natürlicher Suchbaum entstehen, dessen durchschnittliche Suchpfadlänge von der Größenordnung $N/2$ ist. Glücklicherweise treten solche zu linearen Listen „degenerierten“ binären Suchbäume unter den den $N!$ möglichen Anordnungen von N Schlüsseln entsprechenden Suchbäumen nicht allzu häufig auf. Daher sind die Erwartungswerte für die durchschnittliche Suchpfadlänge und die Kosten zur Ausführung einer Einfüge- oder Entferne-Operation für einen zufällig erzeugten binären Suchbaum mit N Schlüsseln nur von der Größenordnung $O(\log N)$.

Wir wollen in diesem Abschnitt zeigen, wie eine einfache Randomisierungsstrategie helfen kann, „schlechte“ Eingabefolgen zu vermeiden. Durch geeignete Randomisierung der Verfahren zum Einfügen und Entfernen von Schlüsseln analog zu randomisiertem Quicksort, vgl. Abschnitt 2.2.2, wird gesichert, daß unabhängig von der Einfügereihenfolge für jede Menge von N Schlüsseln gilt: Der Erwartungswert für die Kosten einer einzelnen Such-, Einfüge- oder Entferne-Operation in einem randomisierten Suchbaum mit N Schlüsseln ist von der Größenordnung $O(\log N)$. Das wird auf folgende Weise erreicht: Man ordnet jedem Schlüssel eine zufällig gewählte „Zeitmarke“ als Priorität zu. Die Einfüge- und Entferne-Verfahren werden dann so verändert, daß gilt: Unabhängig von der tatsächlichen Reihenfolge, in der die Update-Operationen ausgeführt werden, die eine aktuelle Schlüsselmenge S liefern, wird immer derjenige natürliche Suchbaum zur Speicherung von S erzeugt, der entstanden wäre, wenn man die Elemente von S in der durch ihre Prioritäten gegebenen zeitlichen Reihenfolge in den anfangs leeren Baum der Reihe nach eingefügt hätte. Wir beschreiben nun diese Idee im folgenden genauer und analysieren die Verfahren anschließend.

Randomisierte Suchbäume wurden von Aragon und Seidel [10] erfunden. Unsere Analyse folgt der vereinfachten Darstellung in [93].

5.3.1 Treaps

Gegeben sei eine Menge S von Objekten mit der Eigenschaft, daß jedes Element $x \in S$ zwei Komponenten hat, eine Schlüsselkomponente $x.key$ und eine Prioritätskomponente $x.priority$. Wir nehmen an, daß die Schlüsselkomponenten einem vollständig geordneten Universum entstammen, also ohne Einschränkung ganzzahlig sind. Die Prioritäten sollen einem davon möglicherweise verschiedenen, ebenfalls vollständig geordneten Universum entstammen. Ein *Treap* zur Speicherung von S ist ein binärer Suchbaum für die Schlüsselkomponenten und ein Min-heap für die Prioritäten der Elemente von S. Ein Treap ist also eine Hybridstruktur, die die Eigenschaften von binären Suchbäumen

(*trees*) und Vorrangswarteschlangen (*heaps*), vgl. Abschnitt 2.3 und 6.1, miteinander verbindet. Im Abschnitt 7.4.4 werden wir eine Variante dieser Struktur zur Speicherung von Punkten in der Ebene diskutieren, die von McCreight [119] vorgeschlagen und Prioritäts-Suchbaum genannt wurde.

Genauer gilt für jeden Knoten p eines Treaps: Speichert p das Element x, so gelten für p die folgende Suchbaum- und Heapbedingung.

Suchbaumbedingung: Für jedes Element y im linken Teilbaum von p ist $y.key \leq x.key$ und für jedes Element y im rechten Teilbaum von p ist $x.key \leq y.key$.

Heapbedingung: Für jedes in einem Sohn von p gespeicherte Element z gilt $x.priority \leq z.priority$.

Beispiel: Abbildung 5.37 zeigt einen Treap, der die Elemente der Menge $S = \{(1,4),(2,1),(3,8),(4,5),(5,7),(6,6),(8,2),(9,3)\}$ speichert. Dabei soll die erste Zahl jeweils den Schlüssel und die zweite die Priorität bezeichnen.

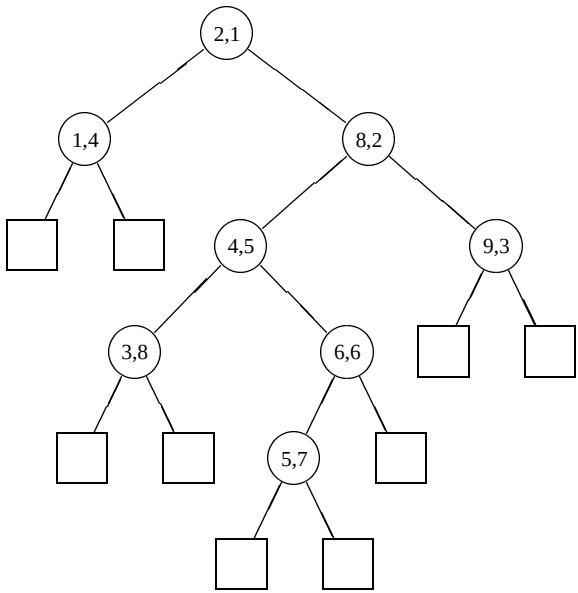


Abbildung 5.37

Wir überlegen uns zunächst, daß es für jede Menge S von Elementen mit paarweise verschiedenen Schlüsseln und Prioritäten genau einen Treap gibt, der S speichert. Ist nämlich x das eindeutig bestimmte Element von S mit minimaler Priorität, so muß x an der Wurzel des Treap gespeichert werden. Teilt man nun die restlichen Elemente von S in die zwei Mengen $S_1 = \{y \mid y.key < x.key\}$ und $S_2 = \{y \mid y.key > x.key\}$, so müssen auf

dieselbe Weise konstruierte Treaps jeweils linke und rechte Teil-Treaps der Wurzel (mit Element x) werden. Die Eindeutigkeit des S speichernden Treap folgt damit induktiv.

Suchen und Einfügen in Treaps

Sei nun ein Treap gegeben, der eine Menge S von Elementen speichert. Die *Suche* nach einem Element x kann wie bei normalen binären Suchbäumen nur unter Benutzung der Schlüsselkomponenten durchgeführt werden. Wie kann man ein neues Element x mit neuer Schlüssel- und Prioritätskomponente in einen Treap *einfügen*? Dazu geht man wie folgt vor: Zunächst wird das Blatt, bei dem die Suche nach x.*key* (erfolglos) endet, durch einen inneren Knoten ersetzt, der x speichert. Der resultierende Baum ist ein Suchbaum für die Schlüsselkomponenten, aber im allgemeinen kein Treap, weil die Heapbedingung für die Prioritäten möglicherweise nicht gilt. Denn x.*priority* kann kleiner sein als die Priorität des beim Vater von x gespeicherten Elements. Die uns schon bekannten Rotationsoperationen zur lokalen Umstrukturierung von binären Suchbäumen können dazu benutzt werden, die Heapbedingung wieder herzustellen. Abbildung 5.38 zeigt diese Operationen. Offenbar kann man durch Ausführen einer Rotation (nach links oder rechts) ein Element um ein Niveau heraufbewegen; gleichzeitig wird dadurch ein anderes herabbewegt. Dabei bleibt die Suchbaumstruktur erhalten.

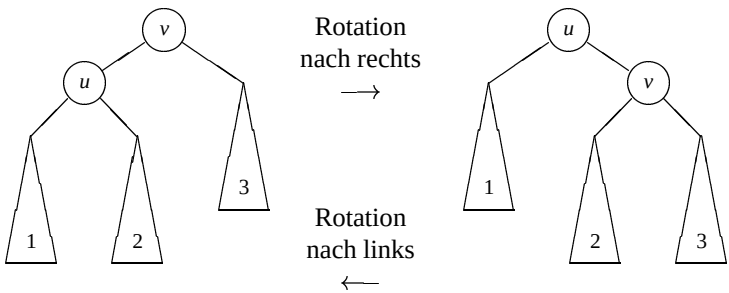


Abbildung 5.38

Falls also die Heapbedingung für x nicht gilt, wird x durch Rotationen nach links oder rechts solange nach oben bewegt, bis die Heapbedingung wieder gilt oder x bei der Wurzel angelangt ist. Abbildung 5.39 zeigt die zur Wiederherstellung der Heapbedingung nach Einfügen des Elements $(7,0)$ in den Treap von Abbildung 5.37 erforderlichen Schritte. Darin sind die zwei Knoten, für die eine Rotation nach links oder rechts durchgeführt wird, jeweils durch einen „$*$“ gekennzeichnet.

Entfernen von Elementen aus Treaps

Zum *Entfernen* eines Elements verfährt man genau umgekehrt. Durch Rotationen nach links oder rechts bewegt man das zu entfernende Element x solange abwärts, bis beide Söhne des Knotens, der x speichert, Blätter sind. Dabei hängt die Entscheidung, ob x durch eine Rotation nach links oder rechts um ein Niveau nach unten bewegt wird,

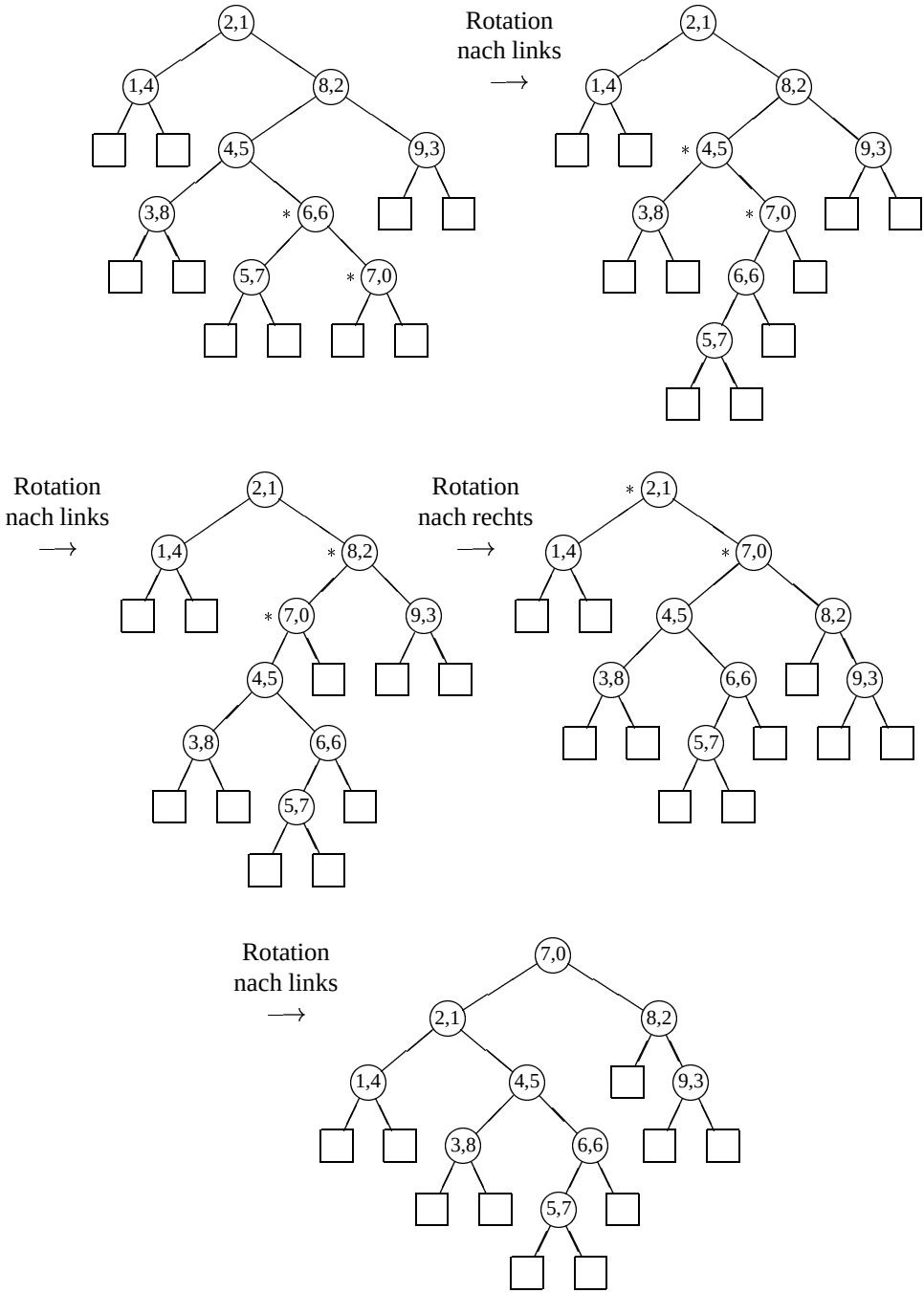


Abbildung 5.39

jeweils davon ab, welcher der beiden Söhne des Knotens, der x gespeichert hat, das Element mit kleinerer Priorität gespeichert hat. Dies Element muß durch die Rotation um ein Niveau hochgezogen werden. Ist x bei einem Knoten angelangt, dessen beide Söhne Blätter sind, entfernt man diesen Knoten und ersetzt ihn durch ein Blatt. Abbildung 5.39 zeigt zugleich ein Beispiel für eine Entferne-Operation: Um aus dem letzten Treap das Element $(7,0)$ zu entfernen, muß das Element $(7,0)$ durch Ausführung der angegebenen Rotationen in umgekehrter Reihenfolge und Richtung nach unten bewegt werden, bis es entfernt werden kann.

5.3.2 Treaps mit zufälligen Prioritäten

Ein *randomisierter Suchbaum* für eine Menge S von Schlüsseln ist ein Treap für eine Menge von Elementen, deren Schlüssel genau die Schlüssel in S sind und deren Prioritäten unabhängig und gleichverteilt zufällig gewählt sind. Wir setzen also voraus, daß keine zwei Schlüssel die gleiche Priorität erhalten. Ferner soll die Zuweisung von Prioritäten so erfolgen, daß jede Permutation der Elemente von S gleich wahrscheinlich ist, wenn man die Elemente von S nach wachsenden Prioritäten ordnet. Um die Zufälligkeit auch nach einer Einfüge- oder Entferne-Operation sicherzustellen, muß der Mechanismus der Zuweisung von Prioritäten zu Schlüsseln, z.B. durch einen Zufallszahlengenerator, vor dem Benutzer verborgen bleiben. Denn sonst könnte er leicht durch „einseitige“ Wahl von Schlüsseln (und Prioritäten) dennoch degenerierte Bäume erzeugen. Fügen wir also in eine Menge S von N Schlüsseln einen weiteren Schlüssel x ein, so nehmen wir an, daß x eine Priorität zugewiesen wird, für die gilt: Die Wahrscheinlichkeit dafür, daß die x zugewiesene Priorität in eines der durch die den bisherigen Elementen zugewiesenen Prioritäten definierten Prioritätsintervalle fällt, ist für jedes Intervall gleich groß. Damit ist klar, daß die Struktur eines randomisierten Suchbaumes für eine Menge von N Schlüsseln mit der eines zufällig erzeugten Suchbaumes für diese Schlüssel identisch ist. Insbesondere ist damit der Erwartungswert für die durchschnittliche Suchpfadlänge von der Größenordnung $O(\log N)$, vgl. Abschnitt 5.1.3.

Wir berechnen jetzt die Erwartungswerte für die Kosten einer einzelnen Such-, Einfüge- und Entferne-Operation. Da eine Einfüge-Operation als invers ausgeführte Entferne-Operation aufgefaßt werden kann, genügt es, die Kosten der Such- und Entferne-Operation abzuschätzen. Die Kosten der Entferne-Operation setzen sich aus zwei Anteilen zusammen, den Kosten, um auf das zu entfernende Element x zuzugreifen (Suchkosten) und den Kosten, x zu den Blättern hinunter zu rotieren und dort zu entfernen (Entfernungskosten).

Suchkosten

Wir berechnen den Erwartungswert für die Kosten, um auf den m-ten Schlüssel in einem randomisierten Suchbaum mit N Schlüsseln zuzugreifen. Dazu nehmen wir ohne Einschränkung an, daß im Suchbaum die Schlüssel $1,\ldots,N$ gespeichert sind und auf m, $1 \leq m \leq N$, zugegriffen wird. Um auf den Schlüssel m zuzugreifen, müssen wir dem Pfad von der Wurzel zu m im Treap folgen. Zur Berechnung der Kosten einer Suchoperation (für die erfolgreiche Suche nach m) genügt es also, den Erwartungswert für den Abstand eines Schlüssels m, $1 \leq m \leq N$, in einem zufällig erzeugten Baum zu

berechnen, der die Schlüssel $\{1,\ldots,N\}$ speichert. Wir betrachten dazu sämtliche Permutationen der Schlüssel $\{1,\ldots,N\}$ und für jede Permutation σ den natürlichen Baum, der sich ergibt, wenn man die Schlüssel in der durch σ bestimmten Reihenfolge in den anfangs leeren Baum einfügt. Dann berechnen wir den Abstand von m von der Wurzel dieses Baumes und mitteln über alle Permutationen. Anders formuliert: Wählen wir eine Permutation σ der Schlüssel $\{1,\ldots,N\}$ zufällig und jede der $N!$ möglichen Permutationen mit derselben Wahrscheinlichkeit, so berechnen wir den Erwartungswert für den Abstand des m-ten Schlüssels von der Wurzel des zu σ gehörenden natürlichen Baumes. Jeden Pfad von der Wurzel eines natürlichen Baumes zum Schlüssel m kann man in zwei Teile zerlegen, in $P_{\leq}(m)$ und $P_{\geq}(m)$.

$P_{\leq}(m)$ enthält genau die Schlüssel, die auf dem Pfad von der Wurzel zu m liegen und kleiner oder gleich m sind.

$P_{\geq}(m)$ enthält genau die Schlüssel, die auf dem Pfad von der Wurzel zu m liegen und größer oder gleich m sind.

Aus Symmetriegründen genügt es, den Erwartungswert für $P_{\leq}(m)$ zu berechnen.

Ist eine Permutation $\sigma = (a_1,\ldots,a_N)$, also $a_i = \sigma(i)$, $1 \leq i \leq N$, gegeben, so liegen genau die Schlüssel k im σ zugeordneten natürlichen Baum auf $P_{\leq}(m)$, für die gilt:

(1) $k \leq m$

(2) k kommt in σ links von m (einschließlich m) vor (d.h. k wurde vor m eingefügt).

(3) k ist größer als alle in σ links von k auftretenden Elemente, die ebenfalls $\leq m$ sind.

Beispiel: Sei $\sigma = (7,2,8,9,1,4,6,5,3)$. Der σ entsprechende natürliche Baum ist der Baum mit derselben Struktur wie der letzte Treap aus Abbildung 5.39; er ist noch einmal in Abbildung 5.40 dargestellt. Dann ist $P_{\leq}(5) = (2,4,5)$, $P_{\leq}(3) = (2,3)$, $P_{\leq}(9) = (7,8,9)$ und $P_{\geq}(5) = (7,6,5)$.

Betrachtet man in einer Permutation σ der Zahlen $\{1,\ldots,N\}$ nur die Elemente, die kleiner oder gleich m sind, in derselben Reihenfolge, in der sie in σ auftreten, so erhält man aus allen Permutationen von $\{1,\ldots,N\}$ alle Permutationen von $\{1,\ldots,m\}$ und zwar jede Permutation mit gleicher Wahrscheinlichkeit, wenn man jede Permutation von $\{1,\ldots,N\}$ mit gleicher Wahrscheinlichkeit wählt. Zur Berechnung des Erwartungswertes für $P_{\leq}(m)$ genügt es also, eine zufällige Permutation τ von $\{1,\ldots,m\}$ zu betrachten und dafür den Erwartungswert EH_m für die Anzahl der Zahlen k zu bestimmen mit der Eigenschaft, daß k größer ist als alle links von k in τ auftretenden Schlüssel. Offenbar hat eine Zahl $k > 1$ diese Eigenschaft genau dann, wenn k sie auch in der Folge hat, die entsteht, wenn man 1 wegläßt. Der Erwartungswert für die Anzahl dieser Zahlen ist daher EH_{m-1}. Die Zahl 1 muß noch hinzugezählt werden, wenn 1 das erste Element in τ ist. Das ist mit Wahrscheinlichkeit $1/m$ der Fall. Damit erhält man die Rekursionsformel

$$EH_m = EH_{m-1} + \frac{1}{m}$$

mit der Lösung $EH_m = \sum_{k=1}^{m} \frac{1}{k} = O(\log m)$.

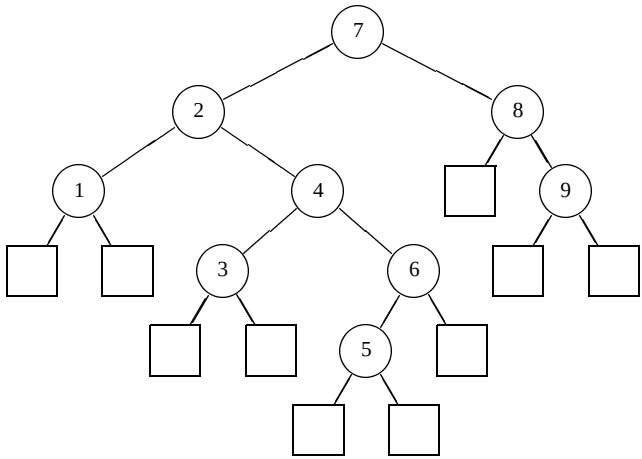


Abbildung 5.40

Man erhält also als Erwartungswert für $P_{\leq}(m)$ den Wert $O(\log m) = O(\log N)$, weil $m \leq N$ ist. Analog folgt, daß auch der Erwartungswert von $P_{\geq}(m)$ von der Größenordnung $O(\log N)$ ist. Die Suche ist daher in jedem Fall in $O(\log N)$ Schritten ausführbar.

Entfernungskosten

Um ein Element m aus einem Treap zu entfernen, muß man zunächst auf m zugreifen und m dann durch Rotationen solange abwärts bewegen, bis m bei den Blättern angelangt ist. Wir müssen also noch den Erwartungswert für die Anzahl der auszuführenden Rotationen berechnen. Zunächst zeigen die in Abbildung 5.38 erläuterten Rotationsoperationen folgendes: Wird ein Element durch eine Rotation nach rechts um ein Niveau abwärts bewegt (Element v in Abbildung 5.38), so nimmt dadurch die Länge des rechtesten Pfades im linken Teilbaum des Knotens, der das Element speichert, um 1 ab; die Länge des linkesten Pfades im rechten Teilbaum des Knotens, der das hinunterbewegte Element speichert, bleibt unverändert.

Analog gilt: Wird ein Element durch eine Rotation nach links um ein Niveau abwärts bewegt (Element u in Abbildung 5.38), so nimmt dadurch die Länge des linkesten Pfades im rechten Teilbaum des Knotens, der das Element speichert, um 1 ab; die Länge des linkesten Pfades im rechten Teilbaum des Knotens, der das hinterunterbewegte Element speichert, bleibt unverändert.

Aus diesen Beobachtungen folgt sofort, daß die Anzahl der Rotationen, um ein Element m von einem Knoten p bis zu den Blättern hinunterzubewegen, gleich der Summe der Länge des rechtesten Pfades im linken Teilbaum von p und der Länge des linkesten Pfades im rechten Teilbaum von p ist.

Beispiel: Für den Baum aus Abbildung 5.40 gilt: Die Knoten mit den Schlüsseln 2, 4, 6 bilden den rechtesten Pfad im linken Teilbaum des Knotens, der 7 speichert; und

der Knoten mit Schlüssel 8 ist der einzige Knoten auf dem linkesten Pfad im rechten Teilbaum des Knotens, der 7 speichert. Die Summe der Längen dieser Pfade ist 4. Vier Rotationen genügen also, um 7 von der Wurzel zu den Blättern zu bewegen. Das sind genau die in Abbildung 5.39 gezeigten Rotationen in umgekehrter Richtung und Reihenfolge.

Aus Symmetriegründen genügt es natürlich, den Erwartungswert EG_m für die Länge des rechtesten Pfades im linken Teilbaum des Knotens zu berechnen, der m gespeichert hat, wenn m ein Schlüssel in einem zufällig erzeugten binären Suchbaum für N Schlüssel $\{1,\ldots,N\}$ und $1 \leq m \leq N$ ist. Natürlich können im linken Teilbaum des Knotens, der m gespeichert hat, nur Schlüssel $k < m$ auftreten. Betrachten wir also eine Permutation σ der Schlüssel $\{1,\ldots,N\}$, so liegt ein Schlüssel k auf dem rechtesten Pfad im linken Teilbaum des Knotens, der m gespeichert hat im σ entsprechenden Baum, wenn folgendes gilt: k tritt rechts von m in σ auf (d.h. k wurde nach m eingefügt) und k ist größer als alle Schlüssel aus $\{1,\ldots,m-1\}$, die k in σ vorangehen und links oder rechts von m auftreten.

Beispiel: Ist $\sigma = (7,2,8,9,1,4,6,5,3)$ und $m = 7$, so haben genau $2,4,6$ die genannte Eigenschaft; ist $m = 4$, so nur $k = 3$.

Es genügt also, für eine zufällig gewählte Permutation τ von $\{1,\ldots,m\}$ die Anzahl EG_m der Zahlen k zu bestimmen, für die gilt:

(1) k tritt in τ rechts von m auf,

(2) k ist größer als alle in τ k vorangehenden Elemente aus $\{1,\ldots,m-1\}$, die rechts von m liegen.

Wenn wir die Bedingung (1) einfach weglassen und nur die Anzahl der Zahlen bestimmen wollen, die (2) erfüllen, können wir direkt das zuvor bei der Analyse der Suchkosten bereits hergeleitete Ergebnis übernehmen; der gesuchte Erwartungswert ist von der Größenordnung $O(\log m)$. Man kann aber mehr zeigen, nämlich, daß $EG_m < 1$ ist, und zwar wie folgt: In einer zufällig gewählten Permutation τ von $\{1,\ldots,m\}$ erfüllt eine Zahl $k > 1$ die Bedingungen (1) und (2) genau dann, wenn sie die entsprechenden Bedingungen für die (ebenfalls zufällige) Permutation erfüllt, die man erhält, wenn man 1 wegläßt. Der Erwartungswert für die Anzahl der Zahlen $k > 1$, die (1) und (2) erfüllen, ist daher gleich EG_{m-1}. Die Zahl 1 erfüllt die Bedingungen (1) und (2) genau dann, wenn m die erste Zahl und 1 die zweite Zahl in der Permutation τ ist. Die Wahrscheinlichkeit dafür ist $1/m(m-1)$. Also gilt für EG_m die folgende Rekursionsformel:

$$\begin{aligned} EG_m &= EG_{m-1} + \frac{1}{m \cdot (m-1)} \text{ und} \\ EG_1 &= 0. \end{aligned}$$

Diese Gleichung hat die Lösung $EG_m = (m-1)/m < 1$.

Insgesamt ergibt sich damit, daß der Erwartungswert für die Anzahl der Rotationen nach der Entfernung eines Schlüssels aus einem randomisierten Suchbaum kleiner als 2 ist. Dasselbe gilt natürlich auch für das Einfügen, weil Einfügen und Entfernen in randomisierten Suchbäumen invers zueinander sind.

Praktische Realisierung
Eine Implementation randomisierter Suchbäume erfordert es, Schlüsseln zufällige Prioritäten zuzuweisen und zwar so, daß nach jeder Update-Operation die Prioritäten der Schlüssel der jeweils vorliegenden Menge unabhängige und gleichverteilte Zufallsvariablen sind. Irgendwelche Annahmen über die Verteilung der Schlüssel selbst werden nicht gemacht. Aragon und Seidel [10] schlagen dazu vor, als Prioritäten zufällige und gleichverteilte reelle Zahlen aus dem Intervall $[0,1)$ zu nehmen und sie wie folgt zu erzeugen: Man generiert die Dualdarstellung der den Schlüsseln zugewiesenen Prioritäten nach Bedarf bitweise Stück für Stück, indem man mit Hilfe eines 0-1-wertigen Zufallszahlengenerators immer gerade soviele Bits erzeugt, wie erforderlich sind, um eine eindeutige Anordnung der den Schlüsseln zugewiesenen Prioritäten zu ermöglichen. Wird also z.B. ein neuer Schlüssel x in einen randomisierten Suchbaum eingefügt, so fügt man x an der vom Suchverfahren ermittelten Position unter den Blättern ein. Ist p der Vater dieses Blattes und hat p einen Schlüssel y gespeichert, dem als Priorität durch n zufällig erzeugte Bits a_i bisher ein Wert $0.a_1 \ldots a_n$ zugewiesen wurde, so erzeugt man so viele neue Bits b_j bis die Bitfolgen $0.a_1a_2\ldots$ und $0.b_1b_2\ldots$ erstmals eine eindeutige Anordnung ermöglichen; unter Umständen kann es erforderlich werden, auch die Bitfolge a_i zu verlängern. Meistens wird aber schon nach wenigen Bits klar sein, welche Bitfolge Anfangsstück der Dualdarstellung der reellen Zahl mit größerem oder kleinerem Wert ist. Dann weist man die so erhaltene Bitfolge x als Priorität zu. Wird nun x nach oben rotiert, so kann es erforderlich werden, die x zugewiesene Priorität mit den anderen Schlüsseln zugewiesenen Prioritäten zu vergleichen. Wenn die bisher erzeugten Bitfolgen keine eindeutige Entscheidung zur Anordnung der Prioritäten erlauben, werden in jedem Fall so viele weitere Bits zufällig erzeugt, bis erstmals eine eindeutige Entscheidung möglich ist. Man kann zeigen [10], daß der Erwartungswert für die zusätzliche Zahl von Bits, die nötig ist, um nach einer Update-Operation eine eindeutige Anordnung der Prioritäten zu ermöglichen, konstant ist (höchstens 12).

5.4 Selbstanordnende Binärbäume

Ganz ähnlich wie bei linearen Listen, vgl. Abschnitt 3.3, kann man auch für binäre Suchbäume Strategien zur Selbstanordnung entwickeln. Das Ziel ist dabei, möglichst ohne explizite Speicherung von Balance-Informationen oder Häufigkeitszählern eine Strukturanpassung an unterschiedliche Zugriffshäufigkeiten zu erreichen. Schlüssel, auf die relativ häufig zugegriffen wird, sollen näher zur Wurzel wandern. Dafür können andere, auf die seltener zugegriffen wird, zu den Blättern hinabwandern. Sind die Zugriffshäufigkeiten fest und vorher bekannt, so kann man Suchbäume konstruieren, die optimal in dem Sinne sind, daß sie die Suchkosten minimieren unter der Voraussetzung, daß sich die Struktur des Suchbaumes während der Folge der Suchoperationen nicht ändert. Verfahren zur Konstruktion optimaler Suchbäume werden im Abschnitt 5.7 vorgestellt. Wir behandeln in diesem Abschnitt den Fall, daß die Zugriffshäufigkeiten nicht bekannt und möglicherweise (über die Zeit) variabel sind.

Durch Ausführung von Rotationen kann der Abstand zur Wurzel eines in einem binären Suchbaum gespeicherten Schlüssels verändert werden, ohne daß die Suchbaumstruktur dadurch zerstört wird. Es ist daher naheliegend, diese Beobachtung für die Entwicklung von Heuristiken zur Selbstanordnung von binären Suchbäumen zu nutzen. So entspricht der *T-Regel* (Transpositionsregel) für lineare Listen die Strategie, nach Ausführung einer Suche das gefundene Element durch eine Rotation um ein Niveau hinaufzubewegen, falls es nicht schon an der Wurzel gefunden wird. Analog entspricht der *MF-Regel* (Move-to-front) für lineare Listen die folgende Move-to-root-Strategie für binäre Suchbäume: Nach jedem Zugriff auf einen Schlüssel wird er durch Rotationen solange hinauf bewegt, bis er bei der Wurzel angekommen ist. Leider haben diese beiden einfachen und naheliegenden Strategien die unangenehme Eigenschaft, daß es beliebig lange Zugriffsfolgen gibt, für die die pro Zugriff benötigte Zeit für einen Baum mit N Schlüsseln von der Größenordnung $\Theta(N)$ ist, vgl. [7]. Wir werden im folgenden Abschnitt jedoch eine Variante der Move-to-root-Heuristik zur Selbstanordnung von binären Bäumen kennenlernen, die amortisierte logarithmische Kosten für alle drei Wörterbuchoperationen garantiert. D.h. die über eine beliebige Folge von Such-, Einfüge- und Entferne-Operationen gemittelten Kosten pro Operation sind von der Größenordnung $O(\log N)$. Natürlich kann eine einzelne Operation für einen nach dieser Strategie entstandenen sogenannten *Splay-Baum* mit N Schlüsseln durchaus $\Theta(N)$ Schritte kosten. Das ist aber nur möglich, wenn vorher genügend viele „billige“ Operationen vorgekommen sind, so daß die Durchschnittskosten über die gesamte Operationsfolge pro Operation $O(\log N)$ sind. Wir erhalten damit zwar nicht dasselbe Verhalten wie bei der Verwendung von balancierten Bäumen im schlechtesten Fall für jede einzelne Operation, aber ein gleich gutes Verhalten für die Operationenfolge im schlechtesten Fall und damit für jede einzelne Operation im Durchschnitt und sogar ein wesentlich besseres, wenn die Zugriffshäufigkeiten auf Schlüssel sehr stark unterschiedlich sind.

5.4.1 Splay-Bäume

Splay-Bäume sind reine binäre Suchbäume, d.h. ohne jede zusätzliche Information wie Balance-Faktoren oder Häufigkeitszähler o.ä., die sich durch eine Variante der Move-to-root-Strategie selbst anordnen. Die wichtigste Operation ist die *Splay-Operation:* Sie verbreitert (englisch: splay) den Suchbaum so, daß nicht nur jeder Schlüssel x, auf den zugegriffen wurde, durch Rotationen zur Wurzel bewegt wird; sondern durch geschickte Zusammenfassung der Rotationen zu Paaren wird darüberhinaus zugleich erreicht, daß sich die Längen sämtlicher Pfade zu Schlüsseln auf dem Suchpfad zu x etwa halbieren. Eine künftige Suche nach einem dieser Schlüssel wird also als Folge der Suche nach x schneller.

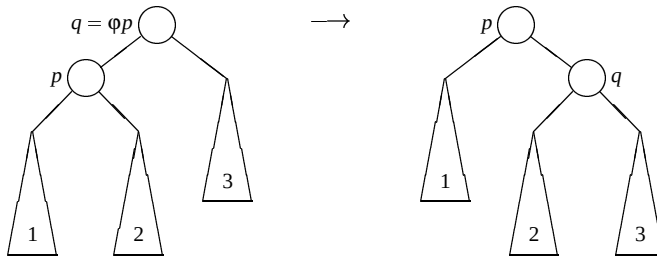
Wir erläutern jetzt zunächst die Splay-Operation, und dann, wie die Wörterbuchoperationen darauf zurückgeführt werden können.

Sei t ein binärer Suchbaum und x ein Schlüssel. Dann ist das Ergebnis der Operation $Splay(t,x)$ der binäre Suchbäum, den man wie folgt erhält.

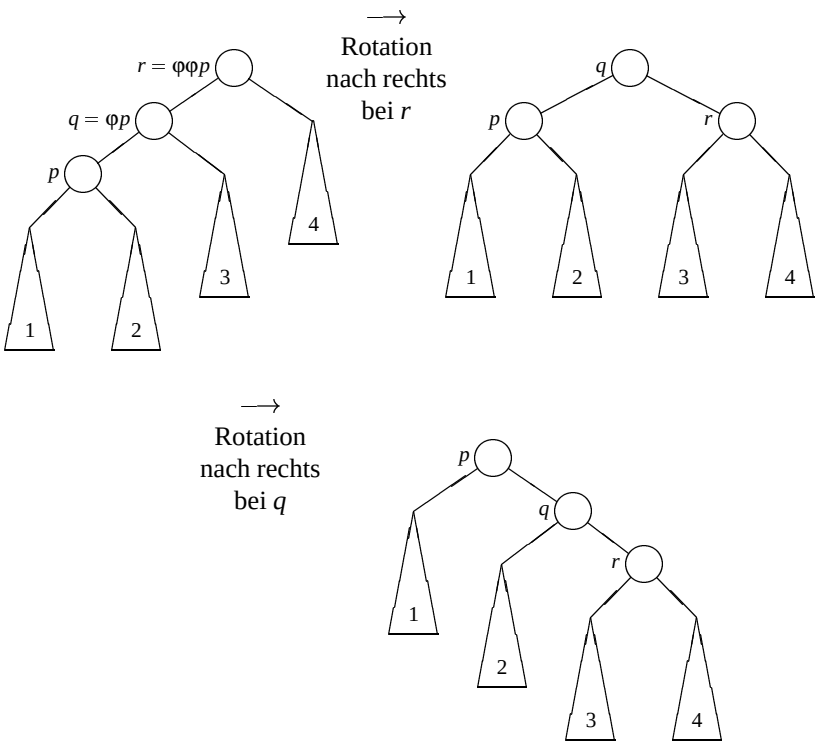
Schritt 1: Suche nach x in t. Sei p der Knoten, bei dem die (erfolgreiche) Suche endet, falls x in t vorkommt, und sei p der Vater des Blattes, bei dem eine erfolglose Suche nach x in t endet, sonst.

Schritt 2: Wiederhole die folgenden Operationen *zig*, *zig-zig* und *zig-zag* beginnend bei p solange, bis sie nicht mehr ausführbar sind, weil p Wurzel geworden ist.

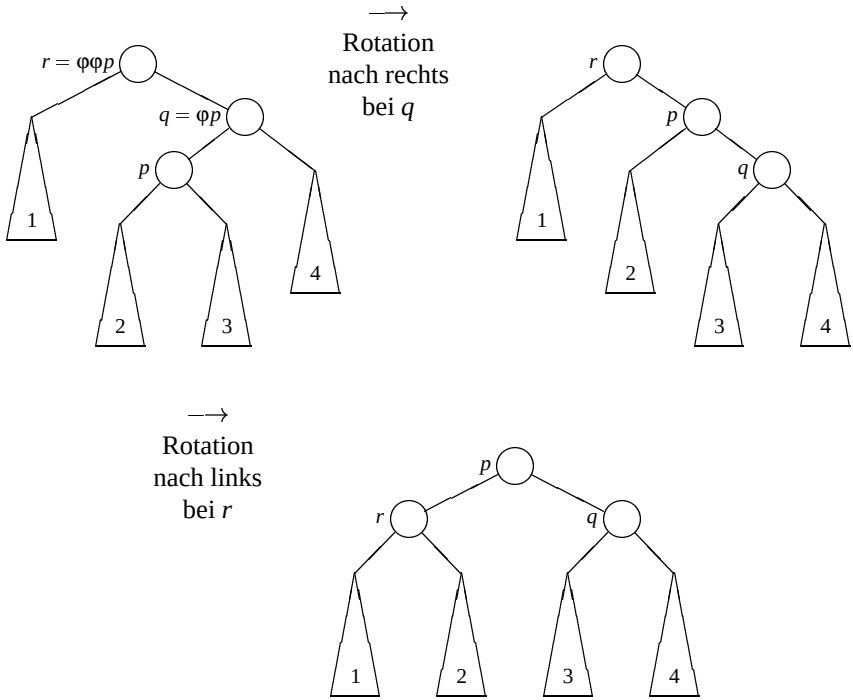
Fall 1: [p hat Vater φp und φp ist die Wurzel]
Dann führe die Operation „*zig*“ aus, d.h. eine Rotation nach links oder rechts, die p zur Wurzel macht.



Fall 2: [p hat Vater φp und Großvater $\varphi\varphi p$ und p und φp sind beides *rechte* oder beides *linke* Söhne]
Dann führe die Operation „*zig-zig*“ aus, d.h. zwei aufeinanderfolgende Rotationen in dieselbe Richtung, die p zwei Niveaus hinaufbewegen.

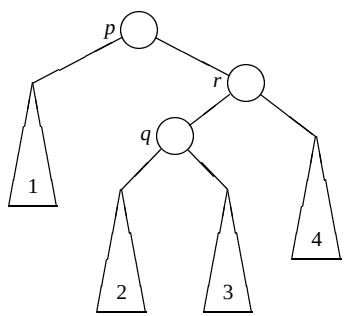


Fall 3: [p hat Vater φp und Großvater $\varphi\varphi p$ und einer der beiden Knoten p und φp ist linker und der andere rechter Sohn seines jeweiligen Vaters]
Dann führe die Operation „*zig-zag*“ aus, d.h. zwei Rotationen in entgegengesetzte Richtungen, die p zwei Niveaus hinaufbewegen.



In jedem dieser drei Fälle haben wir nur jeweils eine der möglichen symmetrischen Varianten veranschaulicht.

Die Splay-Operation kann als eine Variante der Move-to-root-Strategie aufgefaßt werden: Der Schlüssel, auf den zugegriffen wird, wird zur Wurzel rotiert. Während bei der Move-to-root-Strategie jedoch Rotationen strikt „von unten nach oben“ durchgeführt werden, werden bei der Splay-Operation Rotationen nicht immer (nämlich im zig-zig-Fall nicht) strikt in dieser Reihenfolge durchgeführt. Hier liegt der einzige Unterschied zur Move-to-root-Strategie; sie würde im zig-zig-Fall zunächst eine Rotation nach rechts bei q und dann eine Rotation nach rechts bei r durchführen. Als Ergebnis würde man statt des Baumes im Fall 2 erhalten:



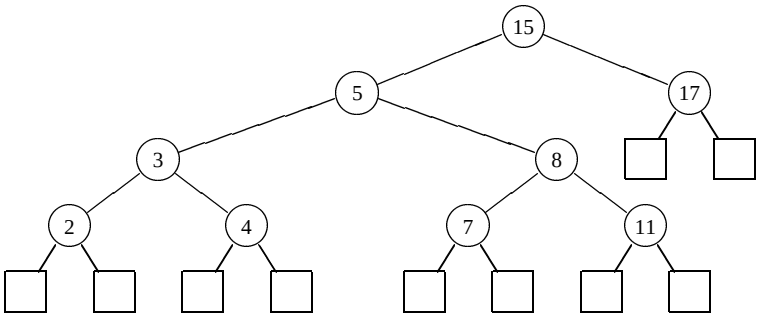


Abbildung 5.41

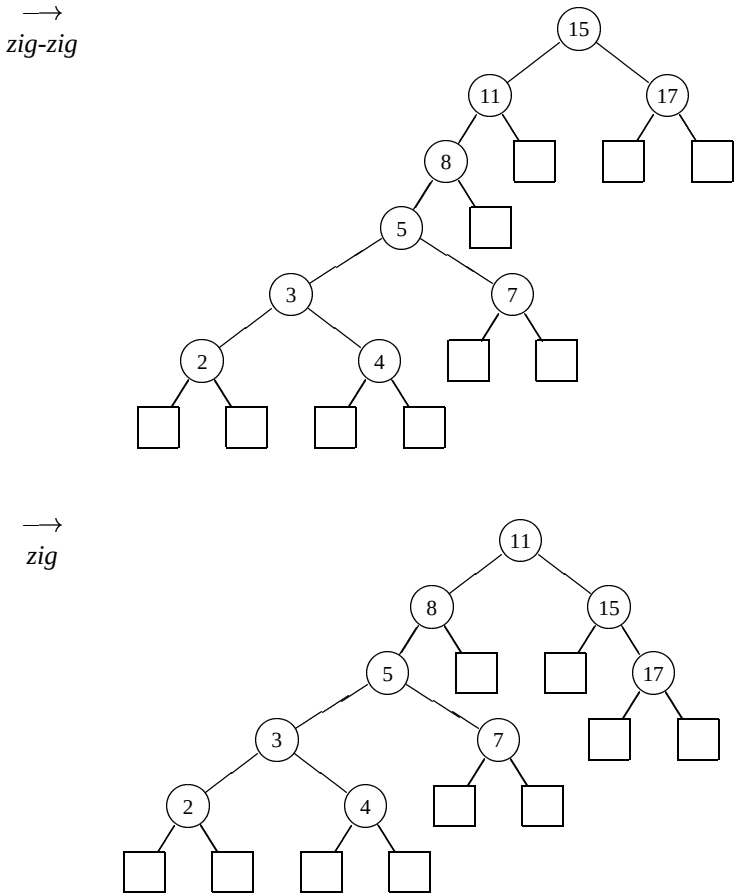


Abbildung 5.42

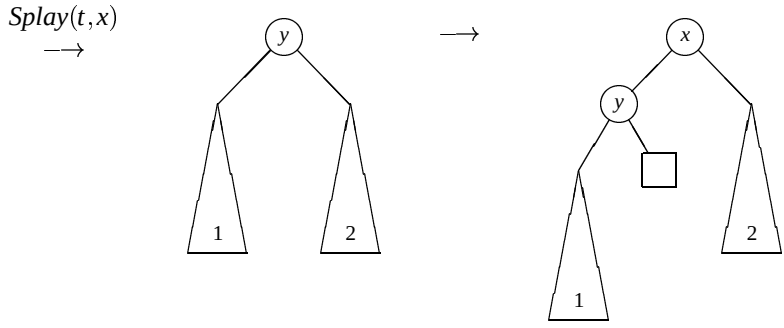
Betrachten wir als Beispiel den Binärbaum t aus Abbildung 5.41.

Das Ausführen der Operationen $Splay(t, 11)$ für diesen Baum erfordert das Ausführen einer *zig-zig*- und einer *zig*-Operation, vgl. Abbildung 5.42.

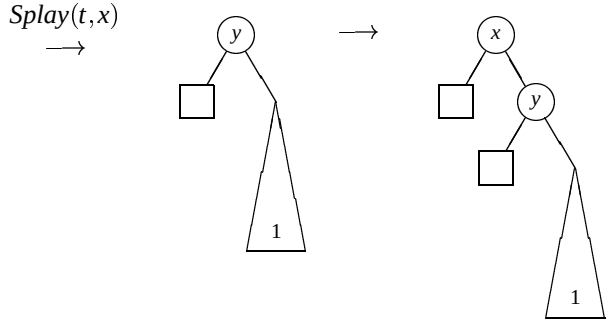
Kommt der Schlüssel x im Baum t vor, so erzeugt $Splay(t,x)$ einen Baum, der x als Schlüssel der Wurzel hat. Kommt x in t nicht vor, so wird durch Ausführen von $Splay(t,x)$ der in der symmetrischen Reihenfolge dem Schlüssel x unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel. Das hängt davon ab, wie die erfolglose Suche nach x endet. Wir können ohne Einschränkung annehmen, daß die erfolglose Suche stets beim symmetrischen Vorgänger von x endet, falls x nicht kleiner ist als alle Schlüssel im Baum, und beim kleinsten Schlüssel im Baum sonst.

Um nach einem Schlüssel x in einem Baum t zu *suchen*, führt man $Splay(t,x)$ aus und sieht dann bei der Wurzel des resultierenden Baumes nach, ob sie den Schlüssel x enthält.

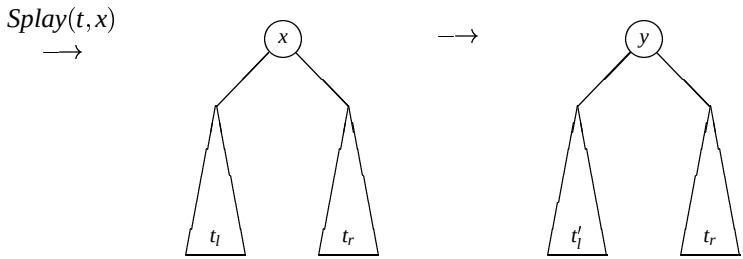
Zum *Einfügen* eines Schlüssels x in t führe zunächst $Splay(t,x)$ aus. Falls dadurch x Schlüssel der Wurzel wird, ist nichts mehr zu tun; denn dann kam x in t schon vor. Kam x in t noch nicht vor, so entsteht durch $Splay(t,x)$ ein Baum, der den symmetrischen Vorgänger y von x in t als Schlüssel der Wurzel hat (oder den kleinsten Schlüssel, falls x kleiner ist als alle Schlüssel in t). Dann schaffe eine neue Wurzel mit x als Schlüssel der Wurzel. Ist also x nicht kleiner als alle Schlüssel in t, so entsteht:



Falls x kleiner ist als alle Schlüssel in t, so entsteht:



Zum *Entfernen* eines Schlüssels x aus einem Baum t führe zunächst wieder $Splay(t,x)$ aus. Falls x nicht Schlüssel der Wurzel ist, ist nichts zu tun; denn dann kam x in t gar nicht vor. Andernfalls hat der Baum den Schlüssel x an der Wurzel und einen linken Teilbaum t_l und einen rechten Teilbaum t_r. Dann führe $Splay(t_l,+\infty)$ aus, wobei $+\infty$ ein Schlüssel ist, der größer ist als alle Schlüssel in t_l. Dadurch entsteht ein Baum t_l' mit dem größten Schlüssel y von t_l an der Wurzel und einem leeren rechten Teilbaum. Ersetze diesen leeren Teilbaum durch t_r.



Man beachte, daß die Ausführung einer Operation $Splay(t,x)$ stets eine Suche nach x im Baum t einschließt. Dasselbe gilt daher auch für jede Wörterbuchoperation. Bei der Analyse der Kosten für die einzelnen Operationen kann man daher die Suchkosten unberücksichtigt lassen, da sie durch die Kosten der längs des Suchpfades auszuführenden Rotationen dominiert werden.

Offensichtlich kann jede Operation Splay, Suchen, Einfügen und Entfernen auf einen beliebigen binären Suchbaum angewandt werden.

Die Klasse aller Bäume, die man erhält, wenn man ausgehend vom anfangs leeren Baum eine beliebige Folge von Such-, Einfüge- und Entferne-Operationen ausführt mit den hier dafür angegebenen Verfahren, heißt die Klasse der *Splay-Bäume.*

5.4.2 Amortisierte Worst-case-Analyse

Zur Abschätzung der Kosten der drei Wörterbuchoperationen müssen wir die Kosten zur Ausführung einer Splay-Operation abschätzen. Denn alle Wörterbuchoperationen wurden auf die Splay-Operation zurückgeführt. Ähnlich wie im Fall selbstanordnender linearer Listen werden wir dazu das Bankkonto-Paradigma verwenden, um die amortisierten Kosten pro Operation zu berechnen. Eine Splay-Operation $Splay(t,x)$ für einen Baum t und einen Schlüssel x besteht darin, auf x zuzugreifen, den Suchpfad zurückzulaufen und entlang dieses Pfades eine Folge von *zig-zag-*, *zig-zig-* und *zig*-Operationen durchzuführen. Wir messen die Kosten durch die Anzahl der ausgeführten Rotationen (plus 1, falls keine Rotation ausgeführt wird). Darin sind die Suchkosten enthalten. Jede *zig*-Operation schlägt mit einer und jede *zig-zig-* oder *zig-zag*-Operation mit zwei Rotationen zu Buche. Manchmal muß man viele, ein anderes Mal wenige Rotationen ausführen. Betrachten wir z.B. den Fall, daß wir der Reihe nach die Schlüssel $1,2,\ldots,N$ in den anfangs leeren Baum nach dem im vorigen Abschnitt angegebenen Verfahren einfügen. Dann wird der jeweils nächste Schlüssel zur neuen Wurzel. Es entsteht also ein zu einer linearen Liste „degenerierter“ Baum. Führt man jetzt als nächstes eine Suchoperation nach dem Schlüssel 1 durch, so müssen nach dem Zugriff auf diesen Schlüssel

N Rotationen durchgeführt werden, um den Schlüssel 1 zur Wurzel zu befördern. Der entstandene Baum hat dann aber die Eigenschaft, daß die weitere Suche nach anderen Schlüsseln billiger wird. Abbildung 5.43 zeigt ein Beispiel für den Fall $N = 5$.

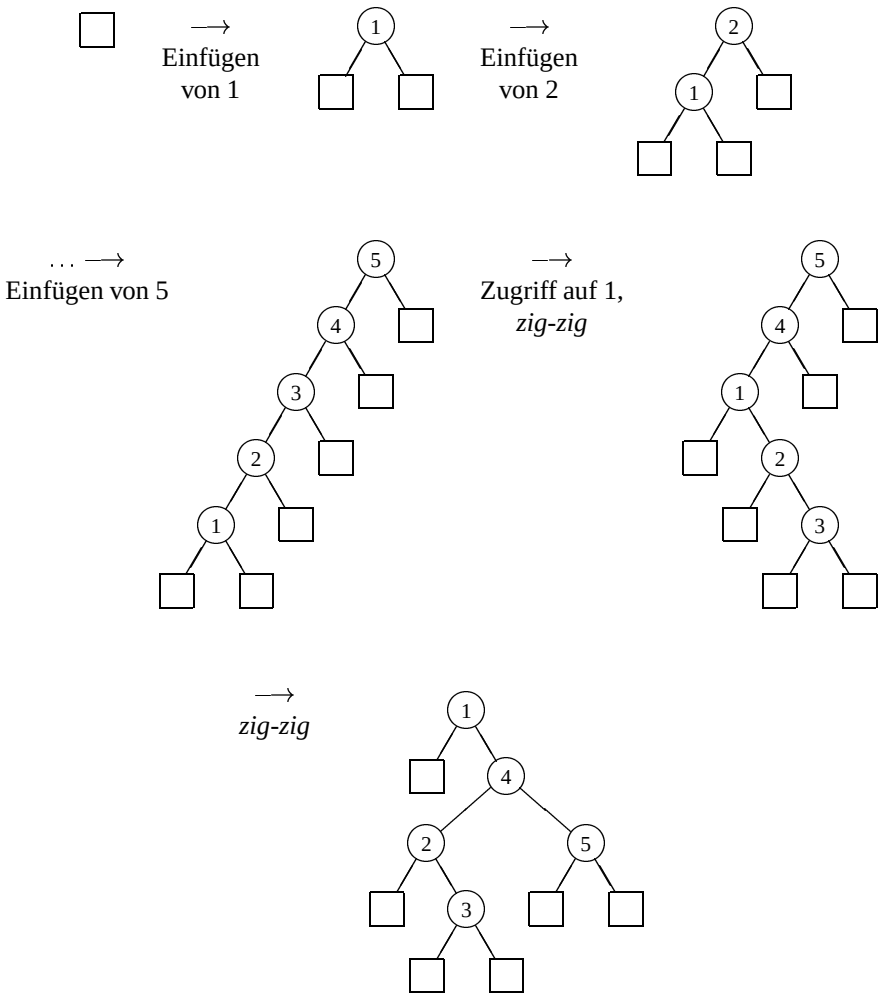


Abbildung 5.43

Manchmal muß man also zur Ausführung einer Splay-Operation viele, ein anderes Mal wenige Einzeloperationen (Rotationen) ausführen. Stellen wir uns daher vor, wir hätten einen festen, nur von der Größe der Struktur abhängigen Durchschnittsbetrag zur Verfügung, den wir für eine Splay-Operation insgesamt ausgeben dürfen. Führen wir dann eine „billige“ Splay-Operation durch, so sparen wir Geld, das wir einem Kon-

to gutschreiben. Dann können wir bei „teuren“ Operationen Geld vom Konto entnehmen, um den erforderlichen Mehraufwand zu bezahlen. Der Gesamtbetrag des für eine Operationsfolge ausgegebenen Geldes ist ein Maß für die Kosten.

Wir ordnen also jedem binären Suchbaum einen nur von seiner Größe abhängigen Kontostand zu. Nehmen wir an, daß niemals Strukturen mit mehr als N Knoten entstehen. Dann werden wir zeigen, daß jede Folge von m Operationen mit einer „Gesamtinvestition“ von $O(m \cdot \log N)$ Geldeinheiten, also im Durchschnitt mit Kosten $O(\log N)$ pro Operation, ausführbar ist.

Genauer sei ϕ_l der nach Ausführung der l-ten Operation vorliegende Kontostand. Dann sind die *amortisierten Kosten* (Zeit) a_l der l-ten Operation in der Folge der m Operationen die Summe der tatsächlichen Kosten (Zeit) t_l plus die Differenz der Kontostände:

$$a_l = t_l + \phi_l - \phi_{l-1}, \text{ für } 1 \leq l \leq m.$$

Dabei ist ϕ_0 der Kontostand am Anfang und ϕ_m der Kontostand der Struktur, die am Ende der Operationsfolge vorliegt. Ist $\phi_0 \leq \phi_m$, so ist die gesamte zur Ausführung der m Operationen verbrauchte amortisierte Zeit $\sum_{i=1}^{m} a_i$ eine obere Schranke für die wirklich verbrauchte Zeit $\sum_{i=1}^{m} t_i$. Denn es gilt dann

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \phi_0 - \phi_m \leq \sum_{i=1}^{m} a_i.$$

Dazu müssen wir zunächst eine geeignete Funktion ϕ finden, die einem Baum einen Kontostand zuordnet.

Wir benutzen die von Sleator und Tarjan [173] vorgeschlagene Funktion ϕ. Sie erlaubt es nicht nur, die behauptete amortisierte Zeitschranke von $O(\log N)$ für jede Wörterbuchoperation herzuleiten, sondern auch weitere Eigenschaften von Splay-Bäumen.

Für jeden Schlüssel x sei $w(x)$ ein beliebiges, aber festes, positives *Gewicht* (englisch: *weight*). Für einen Knoten p sei $s(p)$, die *Größe* von p (englisch: *size*), die Summe aller Gewichte von Schlüsseln im Teilbaum mit Wurzel p. Schließlich sei $r(p)$, der *Rang* von p, definiert durch

$$r(p) = \log_2 s(p).$$

Für einen Baum t mit Wurzel p und für einen in p gespeicherten Schlüssel x sind $r(t)$ und $r(x)$ definiert als Rang $r(p)$.

Man beachte, daß verschiedene Schlüsselgewichte lediglich ein Parameter der Analyse, aber nicht der Algorithmen von Splay-Bäumen sind.

Wir werden später insbesondere den Fall $w(x) = 1$ für alle Schlüssel x betrachten.

Nun definieren wir den einem Splay-Baum t zugeordneten *Kontostand* $\phi(t)$ als die Summe aller Ränge von (inneren) Knoten von t.

Basis der Splay-Baum Analyse ist das folgende Lemma.

Lemma 5.3 (*Zugriffs-Lemma*) *Die amortisierte Zeit, um eine Operation* $\text{Splay}(t,x)$ *auszuführen, ist höchstens* $3 \cdot (r(t) - r(x)) + 1$.

Zum Beweis betrachten wir zunächst den Fall, daß x bereits Schlüssel der Wurzel ist. Dann wird nur auf x zugegriffen und weiter keine Operation ausgeführt. Die tatsächliche Zeit stimmt also mit der amortisierten überein; beide haben den Wert 1 und das Zugriffs-Lemma gilt in diesem Fall, da sich $r(t)$ und $r(x)$ in diesem Fall nicht ändern. Wir können also annehmen, daß wenigstens eine Rotation ausgeführt wird. Für jede im Zuge der Ausführung von $Splay(t,x)$ durchgeführte *zig*-, *zig-zig*- und *zig-zag*-Operation, die einen Knoten p betrifft, betrachten wir die Größe $s(p)$ und den Rang $r(p)$ unmittelbar vor und die Größe $s'(p)$ und den Rang $r'(p)$ unmittelbar nach Ausführung einer dieser Operationen. Wir werden zeigen, daß jede *zig-zig*- oder *zig-zag*-Operation für p in amortisierter Zeit von höchstens $3(r'(p)-r(p))$ und jede *zig*-Operation in amortisierter Zeit höchstens $3(r'(p)-r(p))+1$ ausführbar ist. Nehmen wir einmal an, wir hätten das bereits bewiesen und sei $r^{(i)}(x)$ der Rang von x nach Ausführen der i-ten von insgesamt k *zig-zig*-, *zig-zag*- oder *zig*-Operationen. (Genau die letzte Operation ist eine *zig*-Operation.) Dann ergibt sich als amortisierte Zeit zur Ausführung von $Splay(t,x)$ insgesamt die folgende obere Schranke:

$$\begin{aligned} & 3(r^{(1)}(x)-r(x)) \\ + \quad & 3(r^{(2)}(x)-r^{(1)}(x)) \\ & \vdots \\ + \quad & 3(r^{(k)}(x)-r^{(k-1)}(x))+1 \\ = \quad & 3(r^{(k)}(x)-r(x))+1. \end{aligned}$$

Weil x durch die k Operationen zur Wurzel gewandert ist, ist $r^{(k)}(x)=r(t)$ und damit das Zugriffs-Lemma bewiesen. Wir müssen daher nur noch die amortisierten Kosten jeder einzelnen Operation abschätzen. Dazu betrachten wir jeden der drei Fälle getrennt.

Fall 1 [*zig*] Dann ist $q=\varphi p$ die Wurzel. Es wird eine Rotation ausgeführt. Die tatsächlichen Kosten der *zig*-Operation sind also 1. Es können durch die Rotation höchstens die Ränge von p und q geändert worden sein. Die amortisierten Kosten am_{zig} der *zig*-Operation sind daher:

$$\begin{aligned} am_{zig} &= 1+(r'(p)+r'(q))-(r(p)+r(q)) \\ &= 1+r'(q)-r(p), \text{ da } r'(p)=r(q) \\ &\leq 1+r'(p)-r(p), \text{ da } r'(p)\geq r'(q) \\ &\leq 1+3(r'(p)-r(p)), \text{ da } r'(p)\geq r(p) \end{aligned}$$

Bevor wir die nächsten beiden Fälle behandeln, formulieren wir einen Hilfssatz, den wir dabei verwenden.

Hilfssatz 5.1 *Sind a und b positive Zahlen und gilt $a+b\leq c$, so folgt* $\log_2 a+\log_2 b\leq 2\log_2 c-2$.

Zum Beweis des Hilfssatzes gehen wir aus von der bekannten Tatsache, daß das geometrische Mittel zweier positiver Zahlen niemals größer als das arithmetische ist:

$$\begin{aligned}\sqrt{ab} &\leq (a+b)/2, \text{ also nach Voraussetzung}\\ \sqrt{ab} &\leq \frac{c}{2}\end{aligned}$$

Quadrieren und Logarithmieren ergibt sofort die gewünschte Behauptung.

Kehren wir nun zum Beweis des Zugriffs-Lemmas zurück und behandeln die restlichen zwei Fälle.

Fall 2 [*zig-zag*] Sei $q = \varphi p$ und $r = \varphi\varphi p$. Eine auf p ausgeführte *zig-zag*-Operation hat tatsächliche Kosten 2, weil zwei Rotationen ausgeführt werden. Es können sich höchstens die Ränge von p, q und r ändern. Ferner ist $r'(p) = r(r)$. Also gilt für die amortisierten Kosten

$$\begin{aligned}am_{zig-zag} &= 2 + (r'(p) + r'(q) + r'(r)) - (r(p) + r(q) + r(r))\\ &= 2 + r'(q) + r'(r) - r(p) - r(q)\end{aligned}$$

Nun ist $r(q) \geq r(p)$, weil p vor Ausführung der *zig-zag*-Operation Sohn von q war. Daher folgt

$$am_{zig-zag} \leq 2 + r'(q) + r'(r) - 2r(p) \qquad (*)$$

Um die Abschätzung für $r'(q) + r'(r)$ zu erhalten, betrachten wir noch einmal die Abbildung, in der die *zig-zag*-Operation veranschaulicht wird. Daraus entnehmen wir, daß gilt $s'(q) + s'(r) \leq s'(p)$. Die Definition des Ranges und der oben angegebene Hilfssatz liefern damit $r'(q) + r'(r) \leq 2r'(p) - 2$. Setzt man das in $(*)$ ein, erhält man

$$\begin{aligned}am_{zig-zag} &\leq 2(r'(p) - r(p))\\ &\leq 3(r'(p) - r(p)), \text{ da } r'(p) \geq r(p).\end{aligned}$$

Fall 3 [*zig-zig*] Sei wieder $q = \varphi p$ und $r = \varphi\varphi p$. Eine auf p ausgeführte *zig-zig*-Operation hat tatsächliche Kosten 2, weil zwei Rotationen ausgeführt werden. Genau wie im vorigen Falle folgt zunächst:

$$am_{zig-zig} = 2 + r'(q) + r'(r) - r(p) - r(q)$$

Da vor Ausführung der *zig-zig*-Operationen p Sohn von q und nachher q Sohn von p ist, folgt $r(p) \leq r(q)$ und $r'(p) \geq r'(q)$. Daher gilt

$$am_{zig-zig} \leq 2 + r'(p) + r'(r) - 2r(p)$$

Diese letzte Summe ist kleiner oder gleich $3(r'(p) - r(p))$ genau dann, wenn

$$r(p) + r'(r) \leq 2r'(p) - 2 \qquad (**)$$

ist. Zum Nachweis von $(**)$ betrachten wir noch einmal die Abbildung, die die *zig-zig*-Operation veranschaulicht. Daraus entnimmt man, daß gilt $s(p) + s'(r) \leq s'(p)$. Mit Hilfe des oben angegebenen Hilfssatzes und der Definition der Ränge erhält man daraus sofort die gewünschte Ungleichung $(**)$. Damit ist das Zugriffs-Lemma bewiesen. □

Eine genaue Betrachtung der im Beweis des Zugriffs-Lemmas benutzten Argumentation zeigt folgendes: Nur im Fall 3 (der zig-zig-Operation) ist die Abschätzung der amortisierten Kosten scharf. Sie wird überhaupt erst dadurch möglich, daß hier die strikte „bottom-up-Rotations-Strategie" der Move-to-root-Heuristik lokal durchbrochen wird.

Wir ziehen eine erste Folgerung aus dem Zugriffs-Lemma.

Satz 5.1 *Das Ausführen einer beliebigen Folge von m Wörterbuchoperationen, in der höchstens N mal die Operation Einfügen vorkommt und die mit dem anfangs leeren Splay-Baum beginnt, benötigt höchstens $O(m \cdot \log N)$ Zeit.*

Zum Beweis wählen wir sämtliche Gewichte gleich 1 und erhalten als amortisierte Kosten einer Splay-Operation $Splay(t,x)$ die Schranke $3 \cdot (r(t) - r(x)) + 1$. Weil in diesem Fall für jeden im Verlauf der Operationsfolge erzeugten Baum $s(t) \leq N$ gilt und jede Wörterbuchoperation höchstens ein konstantes Vielfaches der Kosten der Splay-Operation verursacht, folgt die Behauptung. □

Wir haben bereits darauf hingewiesen, daß die Splay-Operation auf einen beliebigen binären Suchbaum anwendbar ist. Das Zugriffs-Lemma erlaubt es, die amortisierten Kosten einer Splay-Operation und damit auch die amortisierten Kosten einer Zugriffs-(Such-)Operation abzuschätzen. Wegen

$$t = a + (\phi_{\text{vorher}} - \phi_{\text{nachher}})$$

kann man die realen Kosten abschätzen, wenn man die durch die Operation bedingte Veränderung des Kontostandes kennt.

Eine auf einem beliebigen Baum mit N Knoten ausgeführte Splay- (oder Such-) Operation wird den Kontostand in der Regel verringern. Die maximal mögliche Abnahme des Kontostandes und der damit zur Ausführung der Operation neben den amortisierten Kosten maximal vom Konto zu entnehmende Geldbetrag kann leicht abgeschätzt werden. Ist $W = \sum_{i=1}^{N} w_i$ die Summe aller Gewichte der im Baum gespeicherten Schlüssel, so ändert sich durch die Splay-Operation für jeden einzelnen Schlüssel i mit Gewicht w_i der Rang $r(i)$ *vor* Ausführung und $r'(i)$ *nach* Ausführung der Splay-Operation höchstens um den Betrag

$$r(i) - r'(i) \leq \log W - \log w_i.$$

Also kann die Gesamtveränderung des Kontostandes wie folgt abgeschätzt werden:

$$\begin{aligned} \phi_{\text{vorher}} - \phi_{\text{nachher}} &\leq \sum_{i=1}^{N} (\log W - \log w_i) \\ &= \sum_{i=1}^{N} \log \frac{W}{w_i}. \end{aligned}$$

Dieselbe Überlegung gilt auch für eine Folge von m Zugriffs-Operationen: Die zur Ausführung der m Operationen erforderlichen wirklichen Kosten $\sum_{l=1}^{m} t_l$ ist die Summe der amortisierten Kosten $\sum_{l=1}^{m} a_l$ plus die Gesamtveränderung des Kontos $\phi_0 - \phi_m$ vor und nach Ausführung der Operationsfolge. Die Gesamtveränderung des Kontos kann wie oben gezeigt durch $\sum_{i=1}^{N} \log(W/w_i)$ abgeschätzt werden.

Wählt man nun wieder $w_i = 1$ für jedes i, so ergibt sich zunächst als amortisierte Zeit für jeden Zugriff auf einen Schlüssel x die Schranke $3 \cdot (r(t) - r(x)) + 1 \leq 3 \cdot \log_2 N + 1$ aus dem Zugriffs-Lemma. Ferner ist die Gesamtveränderung des Kontos durch m Zugriffsoperationen höchstens $\sum_{i=1}^{N} \log(W/w_i) = N \cdot \log N$.

Damit erhält man sofort folgenden Satz.

Satz 5.2 *Führt man für einen beliebigen binären Suchbaum mit N Schlüsseln m-mal die Operation* Suchen *aus, so ist die dafür insgesamt benötigte Zeit von der Größenordnung* $O((N+m)\log N + m)$.

Man beachte, daß eine einzelne Such-Operation sehr wohl $\Omega(N)$ Schritte kosten kann, z.B. dann, wenn man mit einem zu einer linearen Liste „degenerierten" Baum mit Höhe N startet und auf den Schlüssel mit größtem Abstand zur Wurzel zugreift. Aus Satz 5.2 folgt jedoch, daß für jede genügend lange Folge von Zugriffsoperationen, d.h. falls $m = \Omega(N)$, die pro Operation im Mittel über die Operationsfolge erforderliche Zeit durch $O(\log N)$ beschränkt bleibt. Das ist weniger als man für balancierte Bäume erreicht hat, aber mehr als für natürliche Suchbäume gilt. Erkauft wird dieses Verhalten dadurch, daß anders als für natürliche Suchbäume oder balancierte Bäume jede Zugriffs-Operation nach dem für Splay-Bäume definierten Verfahren die Struktur des Baumes verändert (falls nicht gerade auf die Wurzel zugegriffen wird): Jeder Zugriff „verbessert" den Baum in dem Sinne, daß künftige Suchoperationen beschleunigt werden. Genauer kann das durch folgenden Satz ausgedrückt werden.

Satz 5.3 *Führt man für einen beliebigen binären Suchbaum mit N Schlüsseln insgesamt m-mal die Operation* Suchen *aus, so daß dabei auf Schlüssel i $q(i)$-mal zugegriffen wird, so ist die dafür insgesamt benötigte Zeit von der Größenordnung*

$$O(m + \sum_{i=1}^{N} q(i) \log(\frac{m}{q(i)})).$$

Zum Beweis wählen wir als Gewicht des Schlüssels i den Wert $w_i = q(i)/m$ und damit $W = \sum_{i=1}^{N} w_i = 1$ und $\sum_{i=1}^{N} q(i) = m$. Dann folgt aus dem Zugriffs-Lemma für die amortisierten Kosten eines Zugriffs auf einen beliebigen Schlüssel i die obere Schranke

$$\begin{aligned} 3 \cdot (r(t) - r(i)) + 1 &\leq 3 \cdot (\log W - \log w_i) + 1 \\ &= 3 \cdot (\log_2 1 - \log_2 \frac{q(i)}{m}) + 1 \\ &= 3 \cdot \log_2 (\frac{m}{q(i)}) + 1. \end{aligned}$$

Die gesamten amortisierten Zugriffskosten sind also höchstens von der Größenordnung

$$\sum_{i=1}^{N} q(i) \cdot (3 \log_2 (\frac{m}{q(i)}) + 1) = O(m + \sum_{i=1}^{N} q(i) \log(\frac{m}{q(i)})).$$

Da sich durch eine einzelne Zugriffsoperation auf Schlüssel i der Kontostand höchstens um $\log W - \log w_i$ verändern kann, ergibt sich als Gesamtveränderung nach m Operationen höchstens der Betrag

$$\sum_{i=1}^{N} q(i) \cdot \log(\frac{W}{w_i}) = \sum_{i=1}^{N} q(i) \log(\frac{m}{q(i)}).$$

Damit folgt die Behauptung des Satzes. □

Wir vergleichen das Ergebnis mit den Suchkosten eines optimalen Suchbaumes, also eines Suchbaumes, der die minimalen Suchkosten unter allen (statischen) Suchbäumen für N Schlüssel hat, so daß mit der Häufigkeit $q(i)$ auf Schlüssel i zugegriffen wird und $\sum_{i=1}^{N} q(i) = m$ ist. Die Suchkosten eines jeden Suchbaumes sind definiert durch

$$\sum_{i=1}^{N} q(i)(\text{Tiefe}(i)+1) = \sum_{i=1}^{N} q(i) + \sum_{i=1}^{N} q(i)\text{Tiefe}(i).$$

Dabei ist $\mathit{Tiefe}(i)$ der Abstand des Schlüssels i von der Wurzel des Baumes.

Mit Hilfe von Argumenten aus der Informationstheorie kann man nun zeigen, daß in einem optimalen Suchbaum die Tiefe eines Schlüssels i, auf den mit der relativen Häufigkeit $q(i)/m$-mal zugegriffen wird, wenigstens von der Größenordnung $\log(m/q(i))$ sein muß. D.h. es werden in einem solchen Baum zwar Schlüssel, auf die häufiger zugegriffen wird, näher bei der Wurzel sein können, als solche, auf die seltener zugegriffen wird. Dennoch müssen die Schlüssel aufgrund der Binärstruktur den angegebenen Mindestabstand zur Wurzel haben. Aus diesen Überlegungen folgt, daß Splay-Bäume sich „von selbst“ optimalen Suchbäumen anpassen: Obwohl die Zugriffshäufigkeiten nicht bekannt sind, sorgt das Splay-Verfahren dafür, daß durch Zugriffsoperationen Suchbäume entstehen, deren Suchkosten sich von denen entsprechender optimaler Suchbäume (für bekannte Zugriffshäufigkeiten) nur um einen konstanten Faktor unterscheiden. Damit haben Splay-Bäume eine Eigenschaft, die völlig analog ist zu selbstanordnenden linearen Listen, die nach der Move-to-front-Regel manipuliert werden, vgl. hierzu Abschnitt 3.3.

5.5 B-Bäume

Ohne es explizit zu sagen, sind wir in den Abschnitten 5.1 und 5.2 davon ausgegangen, daß die als natürliche oder balancierte Bäume strukturierten Datenmengen vollständig im Hauptspeicher Platz finden. Nicht selten hat man es aber mit Datenmengen zu tun, die nicht mehr im Hauptspeicher des jeweils vorhandenen Rechners gehalten werden können. Sie müssen dann auf sogenannten Hintergrundspeichern, wie Magnetbändern, Magnetplatten oder Disketten, abgelegt werden. Nur die jeweils aktuell etwa für eine Änderungsoperation benötigten Daten werden bei Bedarf vom Hintergrundspeicher in den Hauptspeicher geladen. Man spricht in diesem Fall üblicherweise von Dateien und faßt die Menge der Dienstprogramme zur Handhabung von Dateien zu einem Dateiverwaltungssystem zusammen. Wenn man eine Datei wie eine Internspeicherstruktur, also etwa als AVL-Baum, strukturiert und die Knoten dieses Baumes mehr oder weniger beliebig auf der Magnetplatte, der Diskette oder einem anderen Hintergrundspeicherme-

dium ablegt, so wird man im allgemeinen keineswegs ähnlich effizient suchen, einfügen und entfernen können wie bei interner Speicherung der Datei. Denn zwischen interner Speicherung und Speicherung auf Hintergrundspeichern bestehen grundlegende Unterschiede, die wir zunächst genauer erläutern wollen. Als Ergebnis unserer Überlegungen wird sich ergeben, daß eine spezielle Art von Vielwegbäumen, sogenannte B-Bäume, eine für auf Hintergrundspeichern abgelegte Dateien gut geeignete Organisationsform sind.

Eine Datei besteht aus einzelnen Datensätzen. Die Datei der Studenten an der Universität Freiburg besteht beispielsweise aus in einzelne Felder unterteilten Sätzen, die alle für die Universitätsverwaltung relevanten Daten über die jeweiligen Studenten enthalten. Jedes Feld hat eine bestimmte Bedeutung. Man nennt es daher auch *Attribut*.

Beispiel: Studentendatei

Felder	:	Feld 1	Feld 2	Feld 3	Feld 4
Attribute	:	Matr.Nr.	Name	Fach	Semester
Sätze	:	(4711,	Elvira Schön,	Chemie,	14)
		(007,	Hubert Stahl,	Mikrosystemtechnik,	3)
		(1010,	Monika Bit,	Informatik,	1)

Ein Satzfeld, das zur Identifizierung eines Satzes in einer Operation dient, wird auch *Satzschlüssel* genannt. Wir setzen (wie bisher stets) voraus, daß die Sätze über einen *ganzzahligen* Schlüssel identifiziert werden können. Im Beispiel der Studentendatei kann die Matrikelnummer als Schlüssel genommen werden. Da wir annehmen, daß die Datei auf einem Hintergrundspeicher abgelegt ist, stellt sich natürlich die Frage, woher das Dateiverwaltungssystem weiß, wo ein Satz mit gegebenem Schlüssel auf dem Hintergrundspeicher zu finden ist.

Wir setzen voraus, daß der zur Verfügung stehende Hintergrundspeicher ein Medium mit *direktem Zugriff* ist (z.B. eine Magnetplatte oder Diskette, aber kein Magnetband, das nur *sequentiellen Zugriff* erlaubt). Damit ist folgendes gemeint. Die Oberfläche der Magnetplatte oder Diskette ist durch konzentrische Kreise in *Spuren* und durch Kreisausschnitte in *Sektoren* geteilt. Hierdurch ist die Magnetplatte oder Diskette in direkt adressierbare *Blöcke* gegliedert. Die *Adresse eines Blocks* ist durch seine Spur- und Sektornummer gegeben. Wir nehmen an, daß in jedem Block ein oder mehrere Sätze der Datei gespeichert werden können. Der Dateiverwaltung steht nun permanent eine Tabelle im Hauptspeicher zur Verfügung, in der niedergelegt ist, unter welcher Blockadresse ein durch seinen Schlüssel identifizierter Satz zu finden ist. Diese Tabelle ist ein vollständiges Inhaltsverzeichnis der auf der Magnetplatte oder Diskette abgelegten Datei und wird als *Indextabelle* (kurz: *Index*) bezeichnet. Erhält die Dateiverwaltung etwa den Auftrag, einen Satz mit bestimmtem Schlüssel zu holen, durchsucht sie den Index, um die Blockadresse des Satzes mit diesem Schlüssel festzustellen; die Blockadresse wird dann zur Positionierung des Schreib-Lesekopfes benutzt und der Block in den Hauptspeicher geladen. Das Suchen im Index geht relativ schnell, da es im Hauptspeicher stattfindet und der Index beispielsweise als geordneter Binärbaum organisiert sein kann. Das Positionieren des Schreib-Lesekopfes auf eine bestimmte Blockadresse und das *Laden*, d.h. das Übertragen eines Blocks oder mehrerer aufeinanderfolgender Blöcke vom Hintergrundspeicher benötigt jedoch um Größenordnungen (bis zu 10000

mal) mehr Zeit als eine Suche nach einem Schlüssel im Hauptspeicher. Schwierig wird es nun, wenn der Index so groß ist, daß er nicht im Hauptspeicher Platz hat. Denn dann müssen offenbar Teile des Index wie die Datei selbst auf dem Hintergrundspeicher gehalten werden; nur ein Teil des Index ist im Hauptspeicher *resident*. Dann kann folgender Fall eintreten. Der Benutzer fordert den Zugriff auf einen Satz, dessen Schlüssel aber gerade nicht im residenten Teil des Index zu finden ist. Dann müssen Teile des auf dem Hintergrundspeicher befindlichen Index in den Hauptspeicher geholt werden. Dabei ist es natürlich wünschenswert, nur die richtigen Teile laden zu müssen. In jedem Fall sollte die Anzahl der erforderlichen Hintergrundspeicherzugriffe klein sein, weil sie erhebliche Zeit beanspruchen. Eine gute Möglichkeit zur Lösung dieser Probleme ist es, den Index hierarchisch als Baum, eben als B-Baum zu organisieren.

Dazu denkt man sich den gesamten Index in einzelne *Seiten* unterteilt. Jede Seite enthält eine bestimmte Anzahl von Indexelementen. Die Seiten sind zusammenhängend auf der Magnetplatte oder der Diskette gespeichert. Die Größe der Seiten ist so gewählt, daß mit einem Platten- (oder Disketten-) zugriff genau eine Seite in den Hauptspeicher geladen werden kann. So kann die Seitengröße beispielsweise der Blockgröße entsprechen. Dann kann der gesamte Index auch als Folge von Blöcken angesehen werden, in denen die Seiten des Index gespeichert sind. Jede Seite enthält aber nicht nur einen Teil des Index, sondern darüber hinaus Zusatzinformationen, aus denen das Dateiverwaltungssystem ermitteln kann, welche Seite neu in den Hauptspeicher zu laden ist, wenn der gesuchte Schlüssel nicht in dem gerade residenten Teil des Index zu finden ist. Diese Zusatzinformationen sind natürlich ebenfalls Blockadressen und damit *Zeiger* auf andere Teile des Index. Da in Abhängigkeit vom gesuchten, aber nicht gefundenen Schlüssel auf verschiedene Seiten verzweigt werden kann, ist es ganz natürlich, sich den Index hierarchisch aufgebaut als einen Vielwegbaum vorzustellen. Die Knoten entsprechen den Seiten; jeder Knoten enthält Schlüssel und Zeiger auf weitere Knoten. Durch zusätzliche Forderungen an die Struktur dieser Bäume sorgt man dafür, daß sich die typischen Wörterbuchoperationen, d.h. das Suchen, Einfügen und Entfernen von Schlüsseln (genauer: von durch ihre Schlüssel identifizierten Datensätzen) effizient ausführen lassen. Damit ist die den B-Bäumen zugrunde liegende Idee grob skizziert. Zur präzisen Definition sehen wir zunächst von der bei der Speicherung von Schlüsseln einzuhaltenden Anordnung der Schlüssel untereinander ab und beschreiben nur die B-Bäume charakterisierenden strukturellen Eigenschaften.

Ein *B-Baum der Ordnung m* ist ein Baum mit folgenden Eigenschaften:

(1) Alle Blätter haben die gleiche Tiefe.

(2) Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat wenigstens $\lceil m/2 \rceil$ Söhne.

(3) Die Wurzel hat wenigstens 2 Söhne.

(4) Jeder Knoten hat höchstens m Söhne.

(5) Jeder Knoten mit i Söhnen hat $i - 1$ Schlüssel.

Bemerkung: Die Terminologie im Zusammenhang mit B-Bäumen ist in der Literatur nicht ganz einheitlich. Man spricht manchmal von B-Bäumen der Ordnung k und fordert statt der zweiten Bedingung, daß jeder innere Knoten außer der Wurzel wenigstens $k+1$ Söhne haben muß, und statt der vierten Bedingung, daß jeder Knoten höchstens $2k+1$ Söhne haben darf. Wir haben die Terminologie von D. Knuth [89] übernommen, da sie zu dem zu Beginn dieses Kapitels eingeführten Begriff der *Ordnung* eines Baumes paßt.

B-Bäume der Ordnung 3 heißen auch 2-3-Bäume; ganz allgemein könnte man B-Bäume der Ordnung m in sinnvoller Weise auch $\lceil m/2 \rceil$-m-Bäume nennen, weil jeder innere Knoten mit Ausnahme der Wurzel mindestens $\lceil m/2 \rceil$ und höchstens m Söhne hat.

Deuten wir einen Schlüssel einfach durch einen Punkt an, so zeigt Abbildung 5.44 das Beispiel eines 2-3-Baumes, also eines B-Baumes der Ordnung 3.

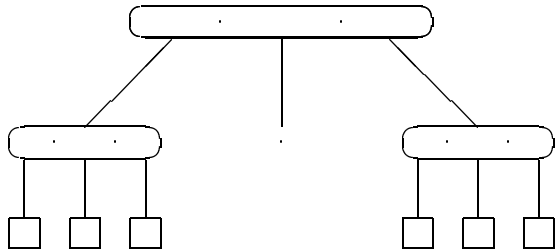


Abbildung 5.44

Dieser Baum hat sieben Schlüssel und acht Blätter. Die Anzahl der Blätter ist also um 1 größer als die Anzahl der Schlüssel. Das ist natürlich kein Zufall, sondern eine einfache Folgerung aus den die Struktur von B-Bäumen bestimmenden Bedingungen (1) – (5). Das beweist man durch Induktion über die Höhe von B-Bäumen. Hat der Baum die Höhe 1, so besteht er aus der Wurzel und k Blättern mit $2 \leq k \leq m$. Er muß dann wegen Bedingung 5. $k-1$ Schlüssel haben. Sind $t_1, \ldots, t_l$, $2 \leq l \leq m$, die l Teilbäume gleicher Höhe h eines B-Baumes mit Höhe $h+1$ und jeweils $n_1, \ldots, n_l$ Blättern und nach Induktionsvoraussetzung jeweils $(n_1-1), \ldots, (n_l-1)$ Schlüsseln, so muß die Wurzel wegen Bedingung 5. $l-1$ Schlüssel haben. Der Baum hat damit wiederum insgesamt $\sum_{i=1}^{l} n_i$ Blätter und $\sum_{i=1}^{l}(n_i-1)+l-1 = \sum_{i=1}^{l} n_i - 1$ Schlüssel gespeichert.

Um die Anzahl der in einem B-Baum mit gegebener Höhe h gespeicherten Schlüssel abzuschätzen, genügt es also, die Anzahl seiner Blätter abzuschätzen. Ein B-Baum der Ordnung m mit gegebener Höhe h hat die minimale Blattzahl, wenn seine Wurzel nur 2 und jeder andere innere Knoten nur $\lceil m/2 \rceil$ Söhne hat. Daher ist die minimale Blattzahl

$$N_{min} = 2 \cdot \lceil \frac{m}{2} \rceil^{h-1}.$$

Die Blattzahl wird maximal, wenn jeder innere Knoten die maximal mögliche Anzahl m von Söhnen hat. Daher ist die maximale Blattzahl

$$N_{max} = m^h.$$

Ist umgekehrt ein B-Baum mit N Schlüsseln gegeben, so hat er $(N+1)$ Blätter. Hat der Baum die Höhe h, so muß gelten:

$$N_{min} = 2 \cdot \lceil \frac{m}{2} \rceil^{h-1} \leq (N+1) \leq m^h = N_{max}$$

Also:

$$h \leq 1 + \log_{\lceil \frac{m}{2} \rceil}(\frac{N+1}{2}) \quad \text{und} \quad h \geq \log_m(N+1).$$

Wir haben also wieder die für eine Klasse balancierter Bäume typische Eigenschaft, daß die Höhe eines B-Baumes logarithmisch in der Anzahl der gespeicherten Schlüssel beschränkt ist. Da die Ordnung m eines B-Baumes üblicherweise etwa bei 100 bis 200 liegt, sind B-Bäume besonders niedrig. Ist etwa $m = 199$, so haben B-Bäume mit bis zu 1999999 Schlüsseln höchstens die Höhe 4.

Wir haben bisher nichts über die Anordnung der Schlüssel in den Knoten eines B-Baumes vorausgesetzt. Für das Suchen, Einfügen und Entfernen von Schlüsseln ist sie natürlich von ausschlaggebender Bedeutung.

Ist p ein innerer Knoten eines B-Baumes der Ordnung m, so hat p l Schlüssel und $(l+1)$ Söhne, $\lceil m/2 \rceil \leq l+1 \leq m$. Es ist zweckmäßig, sich vorzustellen, daß die l Schlüssel $s_1, \ldots, s_l$ und die $(l+1)$ Zeiger $p_0, \ldots, p_l$ auf die Söhne von p wie in Abbildung 5.45 innerhalb des Knotens p angeordnet sind.

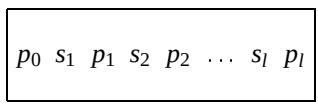


Abbildung 5.45

Dem Schlüssel s_i werden die Zeiger p_{i-1} und p_i zugeordnet, wobei p_{i-1} ein Zeiger auf den $(i-1)$-ten und p_i ein Zeiger auf den i-ten Sohn von p ist; der i-te Sohn von p (bzw. der $(i-1)$-te Sohn) ist die Wurzel des Teilbaums T_{p_i} (bzw. $T_{p_{i-1}}$).

Das Knotenformat eines B-Baumes der Ordnung m kann also in Pascal wie folgt vereinbart werden:

```
const
    m = {Ordnung des B-Baumes};
  type
    Knotenzeiger = ↑Knoten;
    Knoten = record
               {Sohnzahl} l : 0 .. m;
```

```
    {Schlüssel} s : array [1 .. m] of integer;
    {Sohn} p : array [0 .. m] of Knotenzeiger
end;
```

Man verlangt nun zusätzlich zu den bereits angegebenen Bedingungen (1) – (5) die folgende Anordnung der Schlüssel:

(6) Für jeden Knoten p mit l Schlüsseln $s_1, \ldots, s_l$ und $(l+1)$ Söhnen $p_0, \ldots, p_l$ ($\lceil m/2 \rceil \leq l+1 \leq m$) gilt: Für jedes i, $1 \leq i \leq l$, sind alle Schlüssel in $T_{p_{i-1}}$ kleiner als s_i, und s_i wiederum ist kleiner als alle Schlüssel in T_{p_i}.

Das ist die natürliche Erweiterung der von binären Suchbäumen wohlbekannten Ordnungsbeziehung auf Vielwegbäume. (Natürlich haben wir auch hier wieder stillschweigend vorausgesetzt, daß sämtliche Schlüssel paarweise verschieden sind.)

Das Beispiel in Abbildung 5.46 zeigt einen B-Baum der Ordnung 3, der die Schlüsselmenge $\{1, 3, 5, 6, 7, 12, 15\}$ speichert.

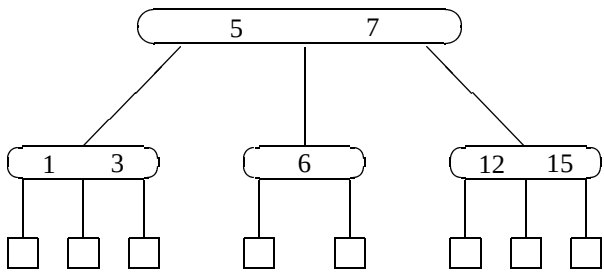


Abbildung 5.46

5.5.1 Suchen, Einfügen und Entfernen in B-Bäumen

Das *Suchen* nach einem Schlüssel x in einem B-Baum der Ordnung m kann als natürliche Verallgemeinerung des von binären Suchbäumen bekannten Verfahrens aufgefaßt werden. Man beginnt bei der Wurzel und stellt zunächst fest, ob der gesuchte Schlüssel x einer der im gerade betrachteten Knoten p gespeicherten Schlüssel $s_1, \ldots, s_l$, $1 \leq l \leq m-1$, ist. Ist das nicht der Fall, so bestimmt man das kleinste i, $1 \leq i \leq l$, für das $x < s_i$ ist, falls es ein solches i gibt; sonst ist $x > s_l$. Im ersten Fall setzt man die Suche bei dem Knoten fort, auf den der Zeiger p_{i-1} zeigt; im letzten Fall folgt man dem Zeiger p_l. Das wird solange fortgesetzt, bis man den gesuchten Schlüssel gefunden hat oder die Suche in einem Blatt erfolglos endet. Es ist klar, daß man im schlechtesten Fall höchstens alle Knoten auf einem Pfad von der Wurzel zu einem Blatt betrachten muß.

Wir lassen offen, wie die Suche nach x innerhalb eines Knotens p mit den Schlüsseln $s_1, \ldots, s_l$ und den Zeigern $p_0, \ldots, p_l$ erfolgt. Um dasjenige i zu finden, für das $x = s_i$ gilt, bzw. das kleinste i zu bestimmen, für das $x < s_i$ ist, bzw. festzustellen, daß $x > s_l$ ist, kann man beispielsweise sowohl lineares als auch binäres Suchen verwenden. Da diese Suche in jedem Fall im Internspeicher stattfindet, beeinflußt sie die Effizienz des gesamten Suchverfahrens weit weniger als die Anzahl der Knoten, die betrachtet werden müssen, die ja unmittelbar mit der Zahl der bei der Suche nach x erforderlichen Hintergrundspeicherzugriffe zusammenhängt.

Um einen neuen Schlüssel x in einen B-Baum *einzufügen*, sucht man zunächst im Baum nach x. Da x im Baum noch nicht vorkommt, endet die Suche erfolglos in einem Blatt, das die erwartete Position des Schlüssels x repräsentiert. Sei der Knoten p der Vater dieses Blattes. Der Knoten p habe die Schlüssel $s_1, \ldots, s_l$ gespeichert, und die Suche nach x ende beim Blatt, auf das der Zeiger p_i zeigt, $0 \leq i \leq l$. Dann sind zwei Fälle möglich:

Fall 1: Der Knoten p hat noch nicht die maximal zulässige Anzahl $m-1$ von Schlüsseln gespeichert. In diesem Fall fügt man x in p zwischen s_i und s_{i+1} ein (bzw. vor s_1, falls $i = 0$, und nach s_l, falls $i = l$), schafft ein neues Blatt, und nimmt in p einen neuen Zeiger auf dieses Blatt auf. Der Einfügevorgang (vgl. Abbildung 5.47) ist damit beendet.

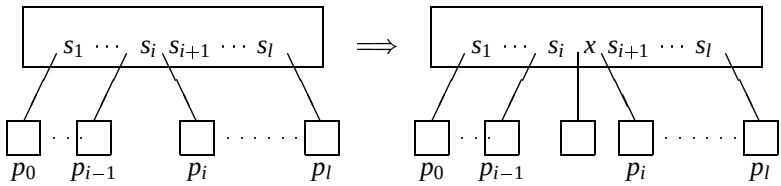
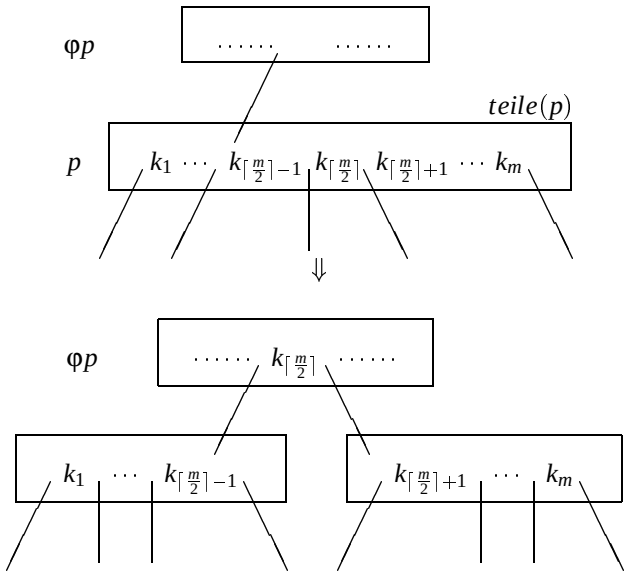


Abbildung 5.47

Fall 2: Der Knoten p hat bereits die maximal zulässige Anzahl $m-1$ von Schlüsseln gespeichert. In diesem Fall ordnen wir den Schlüssel x seiner Größe entsprechend unter die $m-1$ Schlüssel von p ein, schaffen, wie vorher im Fall 1, ein neues Blatt und teilen nun den zu großen Knoten mit m Schlüsseln und $m+1$ Blättern als Söhne in der Mitte auf. D.h.: Sind $k_1, \ldots, k_m$ die Schlüssel in aufsteigender Reihenfolge (also die in p zuvor bereits gespeicherten $m-1$ Schlüssel und der neu eingefügte Schlüssel x), so bildet man zwei neue Knoten, die jeweils die Schlüssel $k_1, \ldots, k_{\lceil m/2 \rceil - 1}$ und $k_{\lceil m/2 \rceil + 1}, \ldots, k_m$ enthalten, und fügt den mittleren Schlüssel $k_{\lceil m/2 \rceil}$ auf dieselbe Weise in den Vater des Knotens p ein. Dieses *Teilen* eines *überlaufenden* Knotens wird solange rekursiv längs eines Pfades zurück von den Blättern zur Wurzel wiederholt, bis ein Knoten erreicht ist, der noch nicht die Maximalzahl von Schlüsseln gespeichert hat, oder bis die Wurzel erreicht ist. Muß die Wurzel geteilt werden, so schafft man eine neue Wurzel, die die durch Teilung entstehenden Knoten als Söhne und den vor der Teilung mittleren Schlüssel als einzigen Schlüssel hat. Der Vorgang des Teilens eines überlaufenden Knotens ist in Abbildung 5.48 dargestellt.



und $teile(\varphi p)$, falls φp (nach Einfügen von $k_{\lceil \frac{m}{2} \rceil}$) m Schlüssel hat

Abbildung 5.48

Es ist klar, daß man im ungünstigsten Fall dem Suchpfad von den Blättern zurück zur Wurzel folgen und jeden Knoten auf diesem Pfad teilen muß. Daraus ergibt sich sofort, daß das Einfügen eines neuen Schlüssels in einen B-Baum der Ordnung m mit N Schlüsseln (und $N+1$ Blättern) in $O(\log_{\lceil m/2 \rceil}(N+1))$ Schritten ausführbar ist.

Wir verfolgen ein Beispiel und fügen in den in Abbildung 5.46 gezeigten B-Baum der Ordnung 3 den Schlüssel 14 ein. Dazu zeigen wir die Situation in den Abbildungen 5.49–5.51 jeweils unmittelbar vor der Teilung eines Knotens; ein überlaufender, also zu teilender Knoten ist jeweils durch einen $*$ markiert.

Zum *Entfernen* eines Schlüssels aus einem B-Baum der Ordnung m geht man umgekehrt vor. Man sucht den Schlüssel im Baum, entfernt ihn und verschmilzt gegebenenfalls einen Knoten mit einem Bruder, wenn er nach Entfernen eines Schlüssels *unterläuft*, also weniger als $\lceil \frac{m}{2} \rceil - 1$ Schlüssel gespeichert hat. Ein Unterlauf der Wurzel, die ja nur einen Schlüssel gespeichert haben muß, bedeutet natürlich, daß die Wurzel keinen Schlüssel mehr gespeichert und nur noch einen einzigen Sohn hat. Man kann dann die Wurzel entfernen und den einzigen Sohn zur neuen Wurzel machen. Wir überlassen die Ausführung der Details dem interessierten Leser und weisen lediglich auf die Ähnlichkeit zum Entfernen von Schlüsseln aus 1-2-Bruder-Bäumen hin. Wie dort muß man das Entfernen eines Schlüssels eines inneren Knotens aus einem B-Baum zunächst auf das Entfernen eines Schlüssels unmittelbar oberhalb der Blätter reduzieren. Dann wird man den Fall, daß man zum Auffüllen eines unterlaufenden Knotens einen Schlüssel von einem Bruder dieses Knotens borgen kann, anders behandeln als den Fall, daß ein unterlaufender Knoten nur (unmittelbare) Brüder hat, die die Minimalzahl von

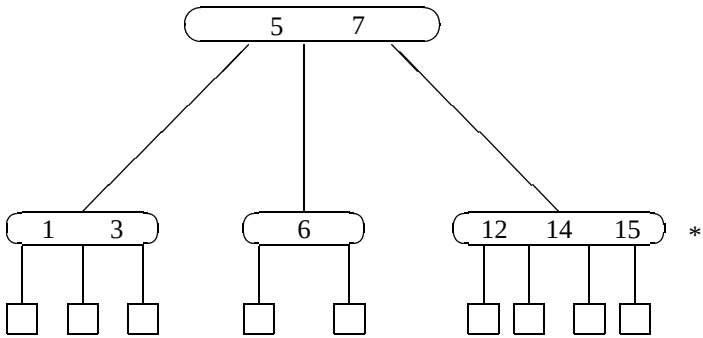


Abbildung 5.49

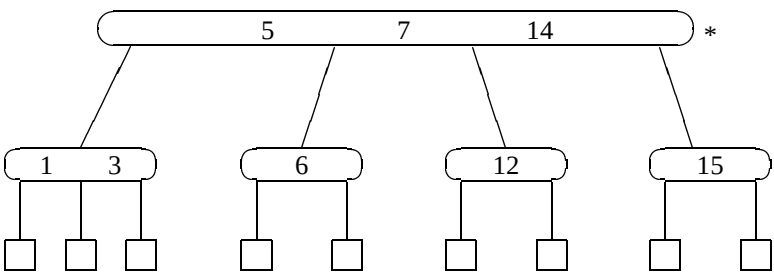


Abbildung 5.50

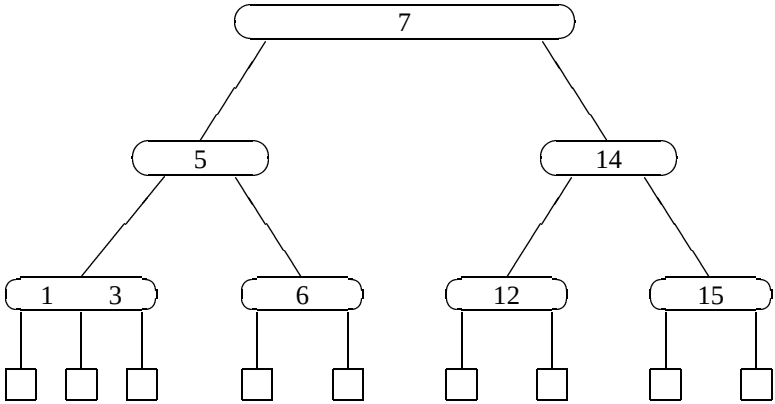


Abbildung 5.51

Schlüsseln gespeichert haben. In diesem Fall kann der Knoten mit einem Bruder verschmolzen werden. Es ist nicht schwer zu sehen, daß das Entfernen eines Schlüssels aus einem B-Baum der Ordnung m mit N Schlüsseln stets in $O(\log_{\lceil m/2 \rceil}(N+1))$ Schritten ausführbar ist.

B-Bäume sind also eine weitere Möglichkeit zur Implementation von Wörterbüchern, die es gestattet, jede der drei Operationen Suchen, Einfügen und Entfernen von Schlüsseln in logarithmischer Zeit in der Anzahl der Schlüssel auszuführen. Das Verhalten im Mittel ist besser. Wie im Falle von 1-2-Bruder-Bäumen gilt auch hier, daß die Gesamtzahl der ausgeführten Knotenteilungen für eine Folge iterierter Einfügungen linear mit der Anzahl der insgesamt erzeugten Knoten zusammenhängt. Weil ein B-Baum der Ordnung m, der N Schlüssel gespeichert hat, höchstens

$$\frac{N-1}{\lceil \frac{m}{2} \rceil - 1} + 1$$

innere Knoten haben kann, ist die mittlere Anzahl von Teilungsoperationen konstant, wenn man über eine Folge von N Einfügeoperationen in den anfangs leeren Baum mittelt, obwohl natürlich eine einzelne Einfügeoperation $\Omega(\log_{\lceil m/2 \rceil} N)$ Knotenteilungen erfordern kann.

Erwartungswerte für die in einem Knoten gespeicherte Schlüsselzahl, also für die *Speicherplatzausnutzung* eines B-Baumes der Ordnung m und weitere das Einfügeverfahren charakterisierende Parameter kann man mit Hilfe der Fringe-Analysetechnik berechnen (vgl. [196]). Es ergibt sich, daß man (unabhängig von m) eine Speicherplatzausnutzung von $ln\ 2 \approx 69\%$ erwarten kann, wenn man eine zufällig gewählte Folge von N Schlüsseln in den anfangs leeren B-Baum der Ordnung m einfügt, d.h. die Knoten des entstehenden B-Baumes sind nur zu gut $2/3$ gefüllt.

Fügt man Schlüssel in auf- oder absteigend sortierter Reihenfolge in den anfangs leeren B-Baum ein, entstehen B-Bäume mit besonders schlechter Speicherplatzausnutzung. Die Knoten sind (in allen Fällen, in denen $N = 2 \cdot \lceil \frac{m}{2} \rceil^h$ ist) *minimal* gefüllt, d.h. die Wurzel hat nur einen und jeder andere innere Knoten nur $\lceil \frac{m}{2} \rceil - 1$ Schlüssel. B-Bäume verhalten sich also gerade anders als 1-2-Bruder-Bäume: B-Bäume werden besonders dünn, 1-2-Bruder-Bäume aber besonders dicht, wenn man Schlüssel in auf- oder absteigend sortierter Reihenfolge einfügt.

Es gibt verschiedene Vorschläge, die schlechte Speicherplatzausnutzung von B-Bäumen zu verhindern. Man kann (wie bei 1-2-Bruder-Bäumen) zunächst die unmittelbaren oder gar alle Brüder eines überlaufenden Knotens daraufhin untersuchen, ob man ihnen nicht Schlüssel abgeben kann, bevor man den Knoten teilt und den mittleren Schlüssel und damit eventuell auch das Überlaufproblem auf das nächsthöhere Niveau verschiebt (vgl. hierzu [33]). Andere Vorschläge zielen darauf ab, für eine Folge bereits sortierter Schlüssel B-Bäume nicht durch iteriertes Einfügen in den anfangs leeren Baum zu erzeugen, sondern möglichst optimale Anfangsstrukturen zu erzeugen in der Hoffnung, daß nachfolgende Einfügungen oder Entfernungen von Schlüsseln den Baum höchstens allmählich, d.h. für eine große Zahl solcher Operationen, stark vom Optimum abweichen lassen.

5.6 Weitere Klassen

Neben den in 5.2 und 5.5 genannten Beispielen für Klassen balancierter Bäume findet man in der Literatur zahlreiche weitere Vorschläge. Allen Klassen gemeinsam ist die Eigenschaft, daß durch die jeweils geforderte Balance-Bedingung eine Klasse von Bäumen definiert wird, deren Höhe logarithmisch in der Knotenzahl bleibt. Sonst werden aber sehr unterschiedliche Ziele verfolgt. Wir geben zunächst eine grobe Übersicht und besprechen dann zwei Aspekte genauer.

5.6.1 Übersicht

Dichte Bäume

Wie wir bereits gesehen haben, besitzen Bruder-Bäume und B-Bäume im allgemeinen mehr Knoten als zur Speicherung einer Menge von Schlüsseln unbedingt notwendig ist. Man kann mit Hilfe der Technik der Fringe-Analyse zeigen, daß man in beiden Fällen eine Speicherplatzausnutzung von etwa 70% für „zufällig“ erzeugte Bäume erwarten kann. Verschiedene Vorschläge zielen darauf ab, *dichte* balancierte Bäume zu erhalten, die vollständigen Bäumen nahekommen. D.h. sie sollen geringe Höhe und keine „überflüssigen“ Knoten haben, aber natürlich soll das Einfügen und Entfernen von Schlüsseln immer noch in logarithmischer Schrittzahl ausführbar sein.

Es ist intuitiv klar, wie man das erreichen kann. Man bezieht in die Umstrukturierungen immer größere Umgebungen (Nachbarn von Knoten auf demselben Niveau, größere „Verwandtschaften“ von Knoten auch auf verschiedenen Niveaus) in die Betrachtungen ein. Das Einfüge- oder Entferne-Problem wird erst dann rekursiv — analog zu Bruder-Bäumen und B-Bäumen — auf das nächsthöhere Niveau verschoben, wenn es sich in der fixierten größeren Umgebung nicht lösen läßt. Die Arbeiten [33, 118] zeigen, daß man auf diese Weise vollständigen Bäumen beliebig nahekommen kann und asymptotisch eine Speicherplatzausnutzung von 100% erreicht. Natürlich hängt die Komplexität der zum Rebalancieren erforderlichen Umstrukturierungsalgorithmen von der Größe der jeweils betrachteten Umgebung ab. Je mehr Brüder oder Nachbarn eines Knotens man in die Betrachtung einbezieht, umso komplizierter werden die Einfüge- und Entferne-Verfahren. Andererseits werden aber die (durch iteriertes Einfügen) erzeugten Bäume auch immer dichter.

Reduktion der Balanceinformation

AVL-Bäume haben gegenüber gewichtsbalancierten Bäumen den großen Vorteil, daß die an jedem Knoten zur Sicherung der AVL-Ausgeglichenheit zu speichernde und zu überprüfende Balanceinformation sehr klein ist. Es genügt, sich einen von drei möglichen Werten 0, 1 oder -1 an jedem Knoten für die Höhendifferenz zwischen linkem und rechtem Teilbaum zu merken. An jedem Knoten eines gewichtsbalancierten Baumes muß man dagegen das Gewicht des gesamten Teilbaumes dieses Knotens, also eine prinzipiell nicht beschränkte Information mitführen. Es hat eine ganze Reihe von schließlich auch erfolgreichen Versuchen gegeben, „einseitig“ höhenbalancierte Bäume und Algorithmen mit logarithmischer Schrittzahl zum Einfügen und Entfernen von Schlüsseln für solche Bäume zu finden. Ein einseitig, z.B. linksseitig höhenbalancierter Binärbaum ist dabei charakterisiert durch die Eigenschaft, daß für jeden Knoten p des Baumes gilt: Die Höhen der beiden Teilbäume von p sind entweder gleich oder aber der linke Teilbaum von p ist um 1 höher als der rechte. Zur Speicherung der Höhendifferenz reicht also ein Bit an jedem Knoten aus. In [79] wurde ein in $O(\log^2 n)$ Schritten ausführbarer Einfügealgorithmus und in [200] ein in logarithmischer Schrittzahl, d.h. in $O(\log n)$ Schritten ausführbares Verfahren zum Entfernen von Schlüsseln für einseitig höhenbalancierte Bäume angegeben.

Man kann zu solchen Verfahren auch auf dem „Umweg“ über einseitige Bruderbäume kommen. Zunächst wird die Bedingung an die Verteilung der unären und binären Knoten in Bruderbäumen wie folgt verschärft. Wir verlangen, daß jeder unäre Knoten einen *rechten* Bruder haben soll mit zwei Söhnen. Für so definierte Klasse von *Rechts-Bruder-Bäumen* kann man Verfahren zum Einfügen und Entfernen von Schlüsseln angeben, deren Laufzeit logarithmisch in der Knotenzahl ist (vgl. dazu [137]). Bruder-Bäume kann man als „expandierte“ höhenbalancierte Bäume und umgekehrt höhenbalancierte Bäume als durch Zusammenziehen unärer Knoten mit ihren jeweils einzigen Söhnen entstehende kontrahierte Bruder-Bäume auffassen (vgl. [138]). In [156] wird dieser Zusammenhang ausgenutzt und ein in logarithmischer Schrittzahl ausführbares Einfügeverfahren für einseitig höhenbalancierte Bäume angegeben. Auch in diesen Fällen kann man beobachten, daß eine Verschärfung der Balancebedingung dazu führt, daß die Update-Verfahren komplizierter werden.

Wege zur Vereinheitlichung

Die große Vielfalt der in der Literatur zu findenden Klassen balancierter Bäume macht es schwer, die verschiedenen Klassen miteinander zu vergleichen. Man möchte ferner nicht für jede neue Variante einer Balancebedingung, also für jede neue Forderung an die statische Struktur von Bäumen, entsprechende Einfüge- und Entferne-Verfahren jedesmal neu erfinden. Es hat daher nicht an Versuchen gefehlt, möglichst viele Klassen balancierter Bäume in einem einheitlichen Rahmen zu behandeln. Zwei Vorschläge sind in diesem Zusammenhang bemerkenswert, die *Rot-schwarz-Bäume* von Guibas und Sedgewick [70] und das *Schichtenmodell* von van Leeuwen und Overmars [106]. Rot-schwarz-Bäume erlauben es, AVL-Bäume, B-Bäume und viele andere Klassen balancierter Bäume einheitlich zu repräsentieren und zu implementieren. Ein Rot-schwarz-Baum ist ein Binärbaum, dessen Kanten entweder rot oder schwarz sind. Die roten (auch: horizontalen) Kanten dienen dazu, Knoten mit mehr als zwei Nachfolgern, wie sie etwa in B-Bäumen vorkommen, binär zu repräsentieren; die schwarzen Kanten entsprechen den sonst üblichen Kanten zwischen Vätern und Söhnen. Knoten der Ordnung 3 und 4 kann man in diesem Rahmen wie in Abbildung 5.52 repräsentieren.

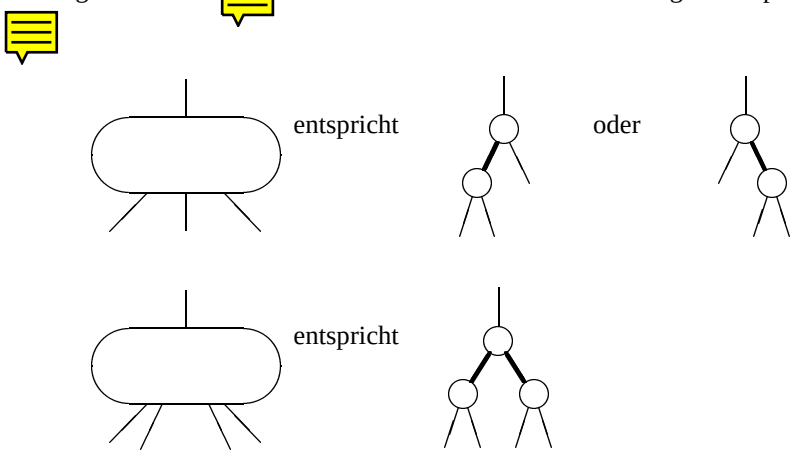


Abbildung 5.52: Rote Kanten sind dick, schwarze dünn gezeichnet.

Als Balancierungsbedingung wird dann verlangt, daß alle Pfade von der Wurzel zu einem Blatt dieselbe Anzahl von schwarzen Kanten haben — dabei werden nur die Kanten zwischen inneren Knoten gezählt. (Das entspricht offenbar der von B-Bäumen und Bruder-Bäumen bekannten Bedingung, daß alle Blätter denselben Abstand zur Wurzel haben müssen.) Weitere Balancebedingungen hängen davon ab, welche Baumklasse in diesem Rahmen repräsentiert werden soll. Will man etwa die Klasse der *2-3-4-Bäume* (das sind Bäume, bei denen jeder innere Knoten 2, 3 oder 4 Söhne hat) im Rahmen der Rot-schwarz-Bäume repräsentieren, so wird zusätzlich verlangt, daß kein Pfad von einem inneren Knoten zu einem Blatt zwei aufeinanderfolgende rote Kanten haben darf. Damit sind in einem 2-3-4-Baum nur die „roten“ Teilbäume aus Abbildung 5.53 möglich.

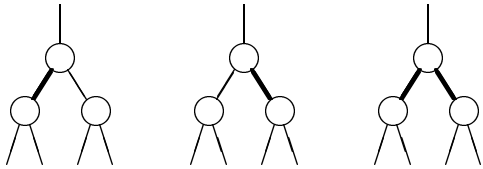


Abbildung 5.53

Ein neuer Knoten wird stets an der erwarteten Position unter den Blättern mit einer roten Kante angefügt. Dadurch kann es vorkommen, daß zwei rote Kanten aufeinanderfolgen. In einem solchen Fall wird eine *Rotation* oder ein *Farbwechsel* ausgeführt, ein Prozeß, der sich rekursiv bis zur Wurzel fortsetzen kann. Wir geben je ein Beispiel für diese Operationen an (siehe Abbildung 5.54); die nicht angegebenen symmetrischen Fälle sind analog zu behandeln.

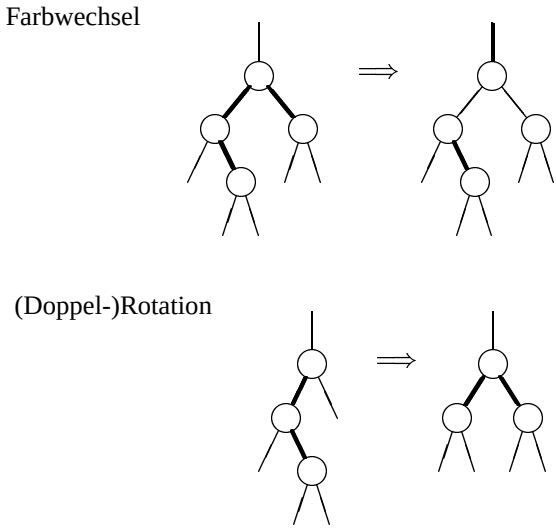
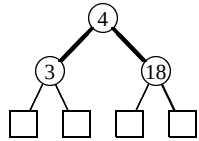


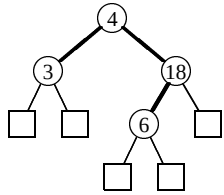
Abbildung 5.54

Wir zeigen am Beispiel der Schlüsselfolge 4, 3, 18, 6, 17, 10, 9, 11, wie mit Hilfe dieser Operationen 2-3-4-Bäumen entsprechende Rot-schwarz-Bäume erzeugt werden können.

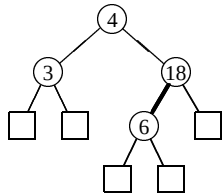
Nach Einfügen der Schlüssel 4, 3, 18 in den anfangs leeren Baum entsteht:



Einfügen des Schlüssels 6 an der erwarteten Position unter den Blättern ergibt zunächst:

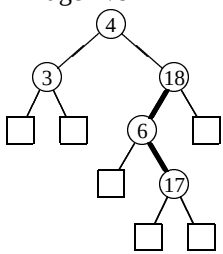


Ein Farbwechsel liefert den zulässigen Baum:

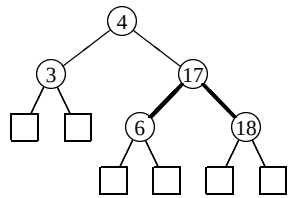


Wir geben die weitere Operationsfolge kurz an:

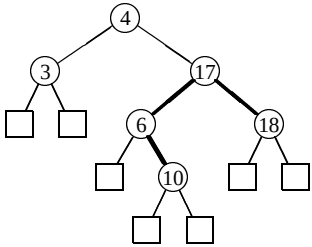
Einfügen von 17



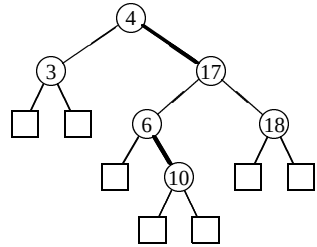
Rotation

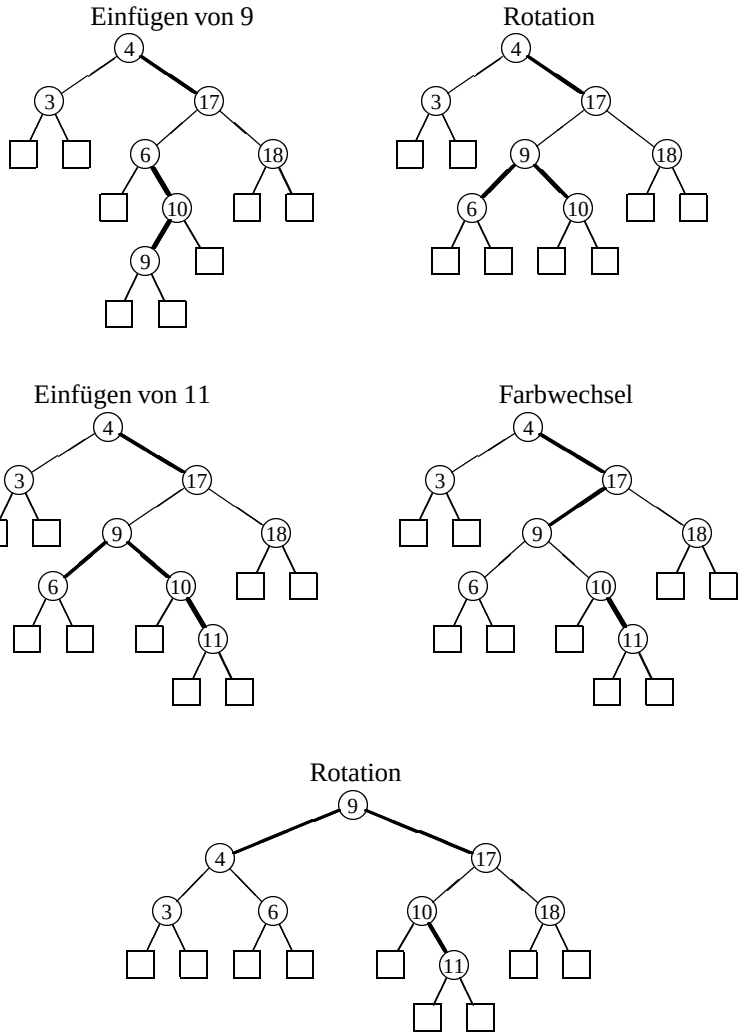


Einfügen von 10



Farbwechsel





Es ist nicht schwer zu sehen, daß die Operationen Farbwechsel und Rotation ausreichen, um aus einem gültigen, einem 2-3-4-Baum entsprechenden Rot-schwarz-Baum wieder einen solchen Baum zu machen, wenn man einen neuen Knoten wie beschrieben einfügt.

AVL-Bäume lassen sich als spezielle Bäume dieser Art auffassen, wenn man ihre Kanten richtig färbt. Definieren wir als Höhe eines Knotens die Länge des längsten Pfades von dem Knoten zu einem Blatt. Dann färbt man genau diejenigen Kanten rot, die von Knoten mit gerader Höhe zu Knoten mit ungerader Höhe führen. Es ist leicht zu zeigen, daß dadurch ein AVL-Baum zu einem speziellen gültigen 2-3-4-Baum in Rot-schwarz-Repräsentation wird.

Auch andere Klassen balancierter Bäume lassen sich in diesem Rahmen darstellen. Auf welche Weise eine Darstellung durch Rot-schwarz-Bäume möglich ist, muß man sich aber in jedem Fall gesondert überlegen.

Im nächsten Abschnitt stellen wir eine Variante des Schichtenmodells von van Leeuwen und Overmars [106] vor, das auf spezielle Bedürfnisse (konstante Zahl struktureller Änderungen pro Update und Entkopplung von Updates und Rebalancierung) zugeschnitten ist. Das Schichtenmodell ist ein Rahmen zur statischen Definition von Klassen balancierter Bäume. Man sorgt wie im Fall von höhen- oder gewichtsbalancierten Bäumen durch geeignete Strukturbedingungen dafür, daß Bäume mit N Blättern stets eine Höhe haben, die in $O(\log N)$ liegt.

Für die in [106] definierten Klassen balancierter Bäume ist leicht zu sehen, daß nicht jeder zur jeweiligen Klasse gehörender Baum durch iteriertes Einfügen von Schlüsseln in den anfangs leeren Baum erzeugt werden kann. Ob und gegebenenfalls welche Unterschiede zwischen einer statisch definierten Klasse von balancierten Bäumen und der Klasse aller Bäume bestehen, die durch Ausführen von Einfüge- oder Entferne-Operationen aus gegebenen Anfangsbäumen gewonnen werden können, ist für viele Klassen balancierter Bäume und zugehöriger Update-Verfahren noch offen (vgl. hierzu [141]).

5.6.2 Konstante Umstrukturierungskosten und relaxiertes Balancieren

Die bisher dargestellten Verfahren zum Ausgleichen von Bäumen nach dem Einfügen oder Entfernen eines Schlüssels in einem balancierten Suchbaum führen im schlechtesten Fall eine logarithmische Zahl struktureller Änderungen durch. Es kann vorkommen, daß man für sämtliche Knoten längs eines Pfades von den Blättern zur Wurzel Rotationen durchführen oder Knoten spalten bzw. verschmelzen muß. Wir stellen jetzt eine Klasse balancierter Bäume vor, die sich nach jeder Einfüge- oder Entferne-Operation durch endlich viele (höchstens drei) Rotationen wieder ausgleichen lassen. Eine Klasse von Bäumen dieser Art und Update-Verfahren für diese Klasse wurden erstmals von Olivié angegeben [135]. Einen anderen Vorschlag findet man in [181].

Außer dieser Eigenschaft, daß pro Update nur *konstante Umstrukturierungskosten* erforderlich sind, haben die in diesem Abschnitt definierten Bäume eine weitere, bemerkenswerte Eigenschaft: Sie eigenen sich besonders gut für den Einsatz in *Mehrbenutzerumgebungen* oder Situationen, wo plötzlich eine sehr große Zahl von Updates erledigt werden muß, so daß möglicherweise nicht genug Zeit ist, um die erforderlichen Umstrukturierungen sogleich nach jeder einzelnen Update-Operation durchzuführen.

Ohne es explizit zu fordern, sind wir nämlich bisher stets stillschweigend davon ausgegangen, daß die drei Wörterbuchoperationen Suchen, Einfügen und Entfernen von Schlüsseln in Bäumen strikt nacheinander ausgeführt werden. Die jeweils nächste Operation darf erst begonnen werden, wenn die jeweils vorangehende vollständig abgeschlossen ist. Insbesondere dann, wenn mehrere Benutzer gleichzeitig konkurrierend auf eine als Baum strukturierte Menge von Daten zugreifen können, möchte man aber auch mehrere Such-, Einfüge- und Entferne-Prozesse gleichzeitig (englisch: concurrent) ausführen können. Solange nur Suchoperationen ausgeführt werden, gibt es dabei wenig Probleme. Denn so können durchaus mehrere Suchprozesse auf denselben Knoten zugreifen (Man muß die jeweils betrachteten Knoten nur für Schreibprozesse sperren). Man kann sich also eine Menge parallel ablaufender Suchprozesse in einem

Suchbaum vorstellen als einen Strom von voneinander unabhängigen, von oben (von der Wurzel) nach unten (zu den Blättern) verlaufenden Einzelprozessen, die sich nicht gegenseitig stören. Die nach dem Einfügen oder Entfernen von Schlüsseln insbesondere bei balancierten Bäumen durchgeführten Strukturänderungen können jedoch dazu führen, daß begonnene und noch nicht beendete Suchprozesse falsche Ergebnisse liefern. Es kann ferner vorkommen, daß parallel ablaufende strukturelle Änderungen nach einer Einfüge- oder Entferne-Operation sich gegenseitig stören. Wir erläutern dies an einem einfachen Beispiel. Nehmen wir an, in einem AVL-Baum wird eine Suche nach einem Schlüssel k begonnen, bevor eine vorangehende Einfüge- oder Entferne-Operation vollständig abgeschlossen wurde, die unter anderem eine Rotation bei einem Knoten q zur Wiederherstellung der AVL-Ausgeglichenheit ausführt, vgl. Abbildung 5.55.

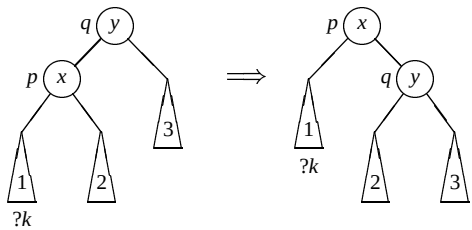


Abbildung 5.55

Nehmen wir an, der Prozeß des Suchens nach dem Schlüssel k sei auf dem Weg von der Wurzel abwärts beim Knoten q angelangt. Ein Schlüsselvergleich ergibt, daß nunmehr der linke Sohn von q betrachtet werden muß. Nehmen wir an, daß jetzt eine Rotation bei q ausgeführt wird, bevor der Suchprozeß fortgesetzt wird. Es folgt, daß die Suche nach k möglicherweise im falschen Teilbaum fortgesetzt wird.

Ähnliche Probleme treten bei nahezu allen Balancierungsverfahren auf. Es können sogar dann falsche Ergebnisse geliefert werden, wenn auf eine Balancierung verzichtet wird, wie bei natürlichen Bäumen. Entfernt man den Schlüssel eines inneren Knotens aus einem solchen Baum, muß er zunächst durch seinen symmetrischen Vorgänger oder Nachfolger ersetzt werden. „Überholt“ nun ein Such-Prozeß einen Entferne-Prozeß an einem solchen Knoten, bevor die Schlüssel ausgetauscht wurden, kann eine Suche falsch dirigiert werden. Wie wir weiter unten erläutern, kann man die beim Entfernen von Schlüsseln auftretenden Probleme aber dadurch umgehen, daß man Blattsuchbäume verwendet.

Es gibt verschiedene Vorschläge in der Literatur, ein reibungsloses, korrektes Miteinander verschiedener Such-, Einfüge- und Entferne-Prozesse sicherzustellen. Wir nennen einige Ansätze.

Sperrstrategien

Knoten, die von einer begonnenen, aber noch nicht abgeschlossenen Umstrukturierungsmaßnahme betroffen sein könnten, werden für nachfolgende Prozesse vorsorglich

gesperrt. Das Verfolgen einer naiven Sperrstrategie kann allerdings leicht dazu führen, daß etwa die Wurzel eines Baumes gesperrt werden muß und damit ein paralleles Abarbeiten mehrerer Prozesse praktisch unmöglich wird. Man findet jedoch in der Literatur eine große Zahl besserer, aber auch komplexerer Sperrstrategien.

Reine Top-down-Update-Verfahren

Es sind Update-Verfahren entwickelt worden, die wie Suchprozesse niemals bereits inspizierte und verlassene Knoten beeinflussen können. Statt also beispielsweise in einem B-Baum nach dem Einfügen eines Schlüssels einen Suchpfad von unten nach oben zurückzulaufen und dabei, falls nötig, überlaufende Knoten zu spalten, geht man so vor: Bereits bei der Suche nach einem neu einzufügenden Schlüssel werden „kritische", d.h. die maximal mögliche Schlüsselzahl enthaltende Knoten vorsorglich gespalten. Man spart damit das Zurücklaufen längs des Suchpfades und kann gefahrlos mehrere Prozesse gleichzeitig ablaufen lassen. Es genügt, die jeweils gerade betrachteten oder zu spaltenden Knoten zu sperren, um eine konsistente Bearbeitung zu sichern.

Die Reduktion des Entfernens von Schlüsseln innerer Knoten auf das Entfernen des symmetrischen Nachfolgers oder Vorgängers kann es erfordern, Zeiger auf den Knoten weit oben im Baum stehenzulassen („dangling pointer"), die später erneut inspiziert werden müssen. Um ein reines Top-down-Vorgehen zu ermöglichen, betrachtet man daher Blattsuchbäume und wählt die „Wegweiser" an den inneren Knoten so, daß sie auch nach dem Entfernen von Schlüsseln der Blätter stehenbleiben können, ohne daß nachfolgende Suchoperationen falsch geleitet werden.

Umstrukturierung als Hintergrundprozeß

Die nach dem Einfügen oder Entfernen von Schlüsseln in balancierten Suchbäumen unter Umständen erforderlichen Umstrukturierungen werden von den Update-Operationen abgekoppelt und als getrennte, im Hintergrund ablaufende, lokale, strukturelle Änderungsoperationen implementiert. Es wird also darauf verzichtet, nach jeder Einfüge- oder Entferne-Operation einen das jeweilige Balancierungskriterium erfüllenden Suchbaum wiederherzustellen. Vielmehr wird eine Anzahl von Umstrukturierungsprozessen generiert, die konkurrierend zu den eigentlichen Update-Operationen ausgeführt werden. Erst wenn alle diese Prozesse vollständig beendet sind, muß wieder ein balancierter Suchbaum vorliegen.

Man spricht in diesem Fall von *relaxiertem Balancieren*. Statt zu fordern, daß die Balance-Bedingung unmittelbar nach jeder Update-Operation wiederhergestellt wird, können die Umstrukturierungsoperationen nach Belieben zurückgestellt und nach Bedarf oder Möglichkeit mit den Such- und Update-Operationen verschränkt ausgeführt werden. In der Literatur findet man zahlreiche Vorschläge für relaxiertes Balancieren (vgl. z.B. [86] [98] [99] [132] [133]). Wir beschreiben jetzt eine besonders einfache und elegante Lösung aus [74].

Stratifizierte Bäume

Stratifizierte Bäume sind Blattsuchbäume, die aus verschiedenen *Schichten* (auch *Straßen* genannt) bestehen. Als Balancebedingung wird gefordert, daß alle Blätter denselben Abstand zur Wurzel haben müssen, wenn man nur die Anzahl der Straßen zählt. Sei nun Z die in Abbildung 5.56 gezeigte Menge von vier Binärbäumen mit den Höhen 1 und 2. Dann ist die Klasse der *Z-stratifizierten Bäume* die kleinste Klasse von Bäumen,

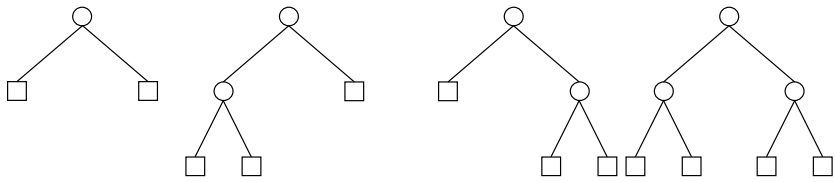


Abbildung 5.56: Menge Z von stratifizierten Bäumen

die man wie folgt erhält:

1. Jeder Baum aus Z ist Z-stratifiziert.
2. Sei ein Z-stratifizierter Baum gegeben. Ersetzt man jedes Blatt des Baumes durch einen Baum aus Z, so ist das Ergebnis wieder ein Z-stratifizierter Baum.

Z-stratifizierte Bäume können daher schematisch wie in Abbildung 5.57 dargestellt werden. Man beachte, daß die Zerlegung eines gegebenen Binärbaumes in Straßen, die zeigt, daß der Baum Z-stratifiziert ist, nicht eindeutig sein muß. Wir sehen also Bäume mit verschiedenen Zerlegungen als verschieden an und denken uns die Zerlegung stets explizit gegeben. Eine Möglichkeit zur Repräsentation der Straßengrenzen ist, die Knoten unterhalb und oberhalb einer Straßengrenze unterschiedlich einzufärben. Es ist nicht schwer zu sehen, daß die soeben definierte Klasse der Z-stratifizierten Bäume identisch ist mit der Klasse der symmetrischen binären B-Bäume [12], der Klasse der halb-balancierten Bäume von Olivié [134] und der Klasse der Rot-schwarz Bäume von Guibas und Sedgewick [70], wenn man die jeweiligen Update-Verfahren nicht berücksichtigt. Ferner ist klar, daß die Höhe eines Z-stratifizierten Baumes mit N Blättern (gemessen in der Anzahl der Kanten eines längsten Pfades von der Wurzel zu einem Blatt) von der Größenordnung $O(\log N)$ ist.

Wir beschreiben jetzt die Update-Verfahren, also das Einfügen und Entfernen von Schlüsseln für Z-stratifizierte Bäume. Den Umstrukturierungsoperationen, die nach einer Einfügung oder Entfernung eines Schlüssels ausgeführt werden müssen, liegt folgende Idee zugrunde:

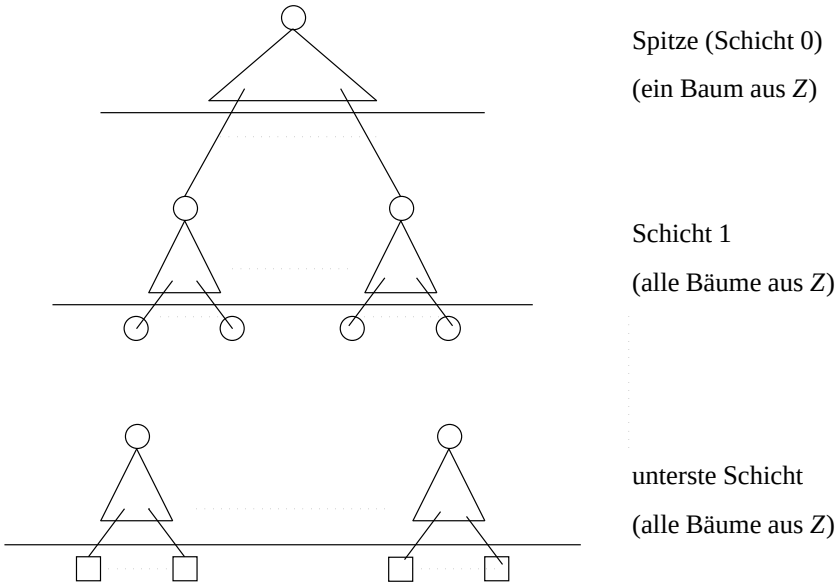


Abbildung 5.57: Struktur eines Z-stratifizierten Baumes

Es wird entweder eine auf die lokale Umgebung eines Knotens beschränkte strukturelle Änderung durchgeführt, oder das Umstrukturierungsproblem wird ohne jede Strukturänderung auf das nächst höhere Niveau, das heißt auf die nächste Straße verschoben. Unter Strukturveränderung verstehen wir dabei stets nur die Änderung von Zeigern; Farbänderungen, also jede lokale Verschiebung einer Straßengrenze, zählen nicht. Dieser Unterschied ist gerechtfertigt durch die bereits oben erläuterte Tatsache, daß es nicht erforderlich ist, Knoten in einer Mehrbenutzerumgebung zu sperren, wenn sich lediglich ihre Farbe ändert. Denn eine Farbänderung kann niemals eine Suchoperation in die falsche Richtung leiten.

Einfügen in Z-stratifizierte Bäume

Um einen neuen Schlüssel in einen Z-stratifizierten Suchbaum einzufügen, bestimmen wir zunächst seine Position unter den Blättern und ersetzen das Blatt, bei dem die erfolglose Suche endet, durch einen inneren Knoten mit zwei Blättern. Diese zwei Blätter speichern jetzt den alten Schlüssel, wo die Suche endete, und den neu eingefügten Schlüssel. Beachte, daß der so entstandene Baum jetzt kein Z-stratifizierter Suchbaum mehr ist, weil ein innerer Knoten unmittelbar unter der untersten Straßengrenze auftritt. Um das zu korrigieren und die Balancebedingung wiederherzustellen, versehen wir diesen Knoten mit einer Push-up-Marke (siehe Abbildung 5.58). Die Aufgabe, die wir für einen Knoten mit einer Push-up-Marke lösen müssen, ist, ihn über die Straßengrenze hinüber zu schieben, unterhalb der er auftritt. (Diese Aufgabe nennen wir auch eine Push-up-Forderung.) Dabei müssen wir darauf achten, daß die Z-stratifizierte Struktur des Baumes wiederhergestellt wird. Zugleich wollen wir erreichen, daß nur eine konstante Anzahl struktureller Änderungen ausgeführt wird. Daher gehen wir so vor, daß

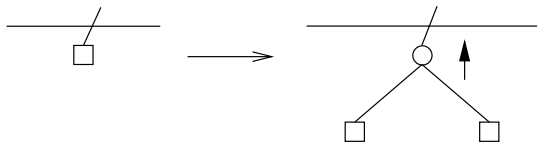


Abbildung 5.58: Einfügen eines neuen Schlüssels mit Setzen einer Push-up-Marke

das Beseitigen einer Push-up-Marke aus einer Bewegung des Knotens mit der Marke über die Straßengrenze hinweg besteht und

1. entweder zu einer strukturellen Änderung führt, die nur ein paar Knoten auf der gerade betrachteten Straße betrifft, und Halt oder

2. zu einer Push-up-Forderung führt für einen Knoten, der unmittelbar unterhalb der Grenze zur nächsthöheren Straße auftritt, aber zu keiner strukturellen Änderung.

Wir unterscheiden also zwei Fälle zur Behandlung von Knoten mit einer Push-up-Marke:

Fall 1 [Es gibt genug Platz in der nächsthöheren Schicht]

Dieser Fall liegt vor, wenn der Knoten mit der Push-up-Marke an einem Baum aus der Menge Z hängt, der nicht die maximale Anzahl von vier Blättern hat. In diesem Fall kann man durch Ausführen von höchstens zwei Rotationen (Einfach- oder Doppelrotation) den Baum aus Z durch einen anderen mit einem zusätzlichen Blatt ersetzen, alle Teilbäume in der gleichen Reihenfolge wieder anhängen und so die Balancebedingung wiederherstellen. Abbildung 5.59 zeigt die in diesem Fall erforderlichen Strukturänderungen. Dabei sind alle symmetrischen Fälle weggelassen.

Fall 2 [Es gibt nicht genug Platz auf der nächsthöheren Schicht]

Dieser Fall liegt vor, wenn der Knoten mit der Push-up-Marke ein Blatt eines vollständigen Binärbaumes der Höhe 2 ist. Denn nun kann man die Push-up-Forderung nicht durch eine lokale Strukturänderung auf der nächsthöheren Schicht erledigen. Also verschieben wir in diesem Fall die Push-up-Forderung rekursiv auf die nächsthöhere Schicht, indem wir die Marke einfach an die Wurzel dieses vollständigen Binärbaums der Höhe 2 auf der nächsthöheren Schicht heften und den Knoten, der vorher die Push-up-Marke hatte, über die Straßengrenze hinaufziehen, ohne eine Strukturänderung durchzuführen. Abbildung 5.60 zeigt eine der vier Möglichkeiten, wo der Knoten mit der Push-up-Marke vorkommen kann. Wir nehmen stillschweigend an, daß eine neue Schicht und eine neue Spitze eingefügt werden, sobald eine Push-up-Marke die Wurzel des ursprünglich gegebenen Z-stratifizierten Baumes erreicht hat. Z-stratifizierte Bäume wachsen also an der Wurzel durch Abspalten eines Knotens von einem Baum, der einen Knoten mehr hat als der Baum mit Höhe 2 und der maximalen Blattzahl 4.

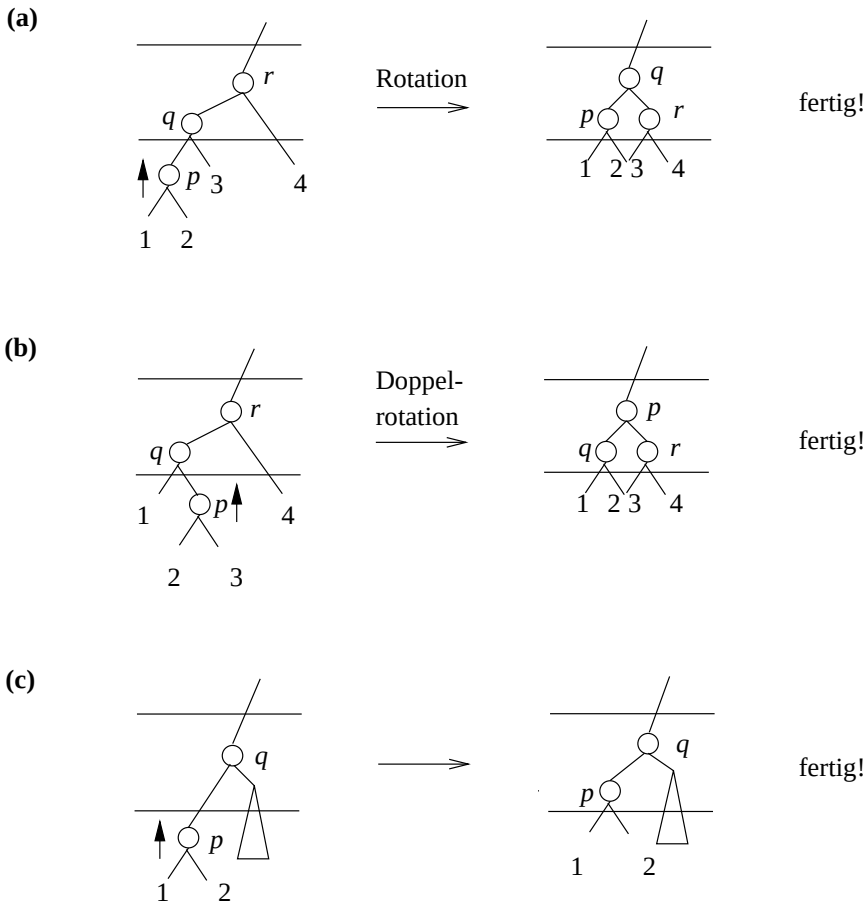


Abbildung 5.59: Lokale Umstrukurierungen bei einer Push-up-Forderung

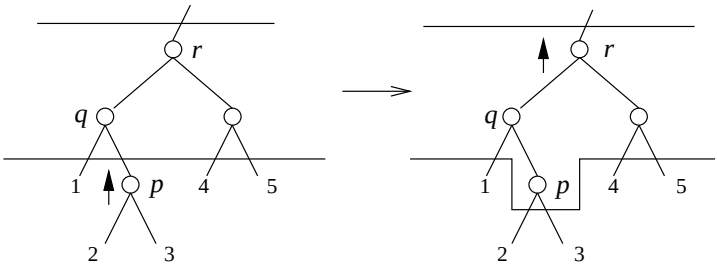


Abbildung 5.60: Rekursive Verschiebung einer Push-up-Forderung zum nächstshöheren Niveau

Wie wir gesehen haben, kann eine einzelne Einfügung zu einer Push-up-Forderung für einen Knoten führen, der unmittelbar unterhalb der untersten Straßengrenze auftritt. Das Erfüllen dieser Push-up-Forderung kann entweder zu einer Reihe weiterer Push-up-Forderungen für Knoten führen, die auf dem Suchpfad liegen und unmittelbar unterhalb der Grenzen zu nächsthöheren Schichten auftreten, ohne eine Strukturänderung durchführen zu müssen, oder aber zu einer lokalen Strukturänderung und Halt. Dabei besteht die Strukturänderung in dem Ersetzen eines Baumes aus der Menge Z von Straßenbäumen durch einen anderen. Sie wird realisiert durch eine Einfach- oder Doppelrotation. Damit dürfte klar sein, daß eine Push-up-Forderung stets durch eine konstante Zahl struktureller Änderungen erfüllt werden kann.

Wir beschreiben jetzt, wie eine *Folge von Einfügungen* behandelt wird, so daß es nicht erforderlich ist, den Baum nach jeder einzelnen Einfügung umzustrukturieren (Dabei lassen wir natürlich zu, daß der Baum zwischenzeitlich nicht mehr Z-stratifiziert ist). Zunächst beobachten wir, daß Push-up-Forderungen akkumuliert werden können und im Baum konkurrierend aufsteigen können so lange nur gesichert ist, daß keine zwei Push-up-Forderungen denselben Straßenbaum betreffen. Falls also mehrere Push-up-Marken an Knoten angebracht sind, die vom selben Straßenbaum über eine Straßengrenze herunterhängen, behandeln wir sie einfach nacheinander in beliebiger Reihenfolge wie oben beschrieben. Sobald eine Push-up-Forderung verschwunden ist (durch eine Strukturänderung oder durch rekursives Hinaufschieben auf die nächsthöhere Schicht), können wir bereits damit beginnen, die nächste Push-up-Forderung zu erfüllen. Abbildung 5.61 zeigt an einem Beispiel, wie hier vorzugehen ist.

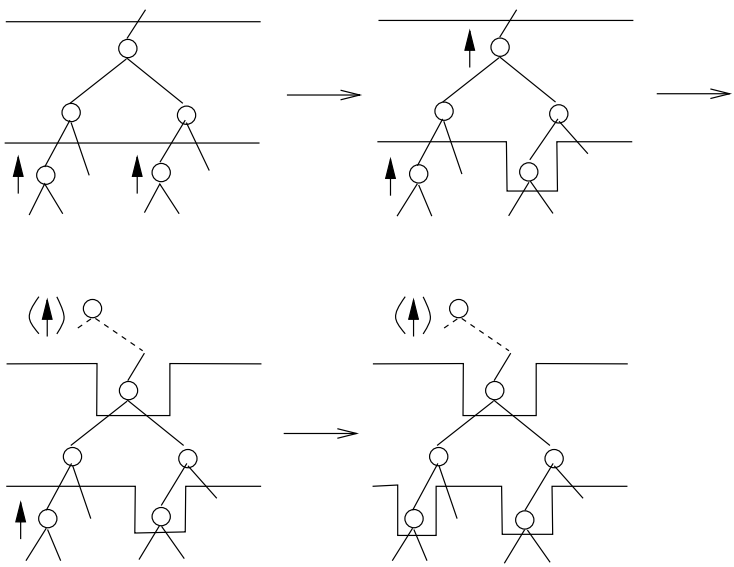


Abbildung 5.61

Dies löst das Problem, wie man Folgen von Einfügungen behandeln kann, die sämtlich verschiedene Blätter des ursprünglich gegebenen Z-stratifizierten Baumes betreffen. Wir fügen einfach an jeden neu erzeugten internen Knoten unterhalb der untersten Straßengrenze eine Push-up-Marke an. Nun sehen wir, daß wir dasselbe auch in dem Falle tun können, daß eine Einfügung in ein Blatt fällt, das nicht Blatt des ursprünglich gegebenen Z-stratifizierten Baumes ist, sondern ein Blatt, das durch eine frühere Einfügung erzeugt worden ist. Das heißt, wir können Push-up-Forderungen für Knoten, die unter der untersten Straßengrenze auftreten, einfach akkumulieren und wie vorher erledigen. Wir erfüllen sie in der Weise, daß wir stets Knoten unmittelbar unterhalb der untersten Straßengrenze des ursprünglich gegebenen Z-stratifizierten Baumes zuerst behandeln (Diese Bedingung gilt zum Beispiel, wenn wir die Push-up-Forderungen in derselben Reihenfolge erfüllen, in der wir die Knoten eingefügt haben.)

In dieser Weise kann also eine Folge von Einfügungen zu einem Wachstum des ursprünglich gegebenen Z-stratifizierten Baumes unterhalb der untersten Straßengrenze führen, das vergleichbar ist mit dem Wachstum eines natürlichen Suchbaumes. Jeder neu erzeugte Knoten hat eine Push-up-Marke. Die Push-up-Forderungen werden, wie oben beschrieben, von oben nach unten, aber sonst in beliebiger Reihenfolge erledigt. Sind alle Push-up-Forderungen erfüllt, ist der resultierende Baum wieder ein Z-stratifizierter Suchbaum. Abbildung 5.62 zeigt schematisch das Bild eines Z-stratifizierten Baumes nach einer Reihe von Einfügungen mit noch nicht erfüllten Push-up-Forderungen.

Entfernen aus Z-stratifizierten Bäumen

Um einen Schlüssel aus einem Z-stratifizierten Suchbaum zu entfernen, suchen wir ihn zunächst unter den Blättern und versehen das Blatt mit einer Löschmarke „$\times$“. Eine Löschmarke kann entweder unmittelbar beseitigt werden durch eine Strukturänderung in der Umgebung des betroffenen Blattes auf der untersten Schicht, oder aber sie führt dazu, daß der Bruder des entfernten Blattes mit einer Pull-down-Marke (Pull-down-Forderung) versehen wird. Denn eine an einem Blatt eines Baumes aus Z mit drei oder vier Blättern angebrachte Löschmarke kann leicht dadurch entfernt werden, daß man den Baum aus Z durch einen Baum ersetzt, der ein Blatt (und einen inneren Knoten) weniger hat. Hat allerdings ein Blatt eines Baumes aus Z mit nur zwei Blättern eine Löschmarke, so kann man nach Entfernen des Blattes die Balancebedingung nicht direkt wiederherstellen. Vielmehr führt das Beseitigen der Löschmarke zu einer Pull-down-Forderung „$\downarrow$“. Das ist in Abbildung 5.63 erläutert, in der alle symmetrischen Fälle weggelassen wurden.

Hat ein Knoten (also anfangs der Bruder des entfernten Blattes) eine Pull-down-Marke, so befindet er sich selbst unmittelbar unter einer Straßengrenze und sein Vater zwei Straßen oberhalb der Straße, auf der er selbst auftritt. Das ist natürlich ein Verstoß gegen die Z-Stratifiziertheit des Baumes. Um diesen Verstoß zu beheben, müssen wir den Vater des Knotens mit der Pull-down-Marke eine Straße hinunterziehen und zugleich dafür sorgen, daß die Schichtenstruktur des Baumes durch eine konstante Anzahl struktureller Änderungen wiederhergestellt wird. Das Beseitigen einer Pull-down-Marke besteht also in einer Bewegung eines Knotens über eine Straßengrenze nach unten hinweg und

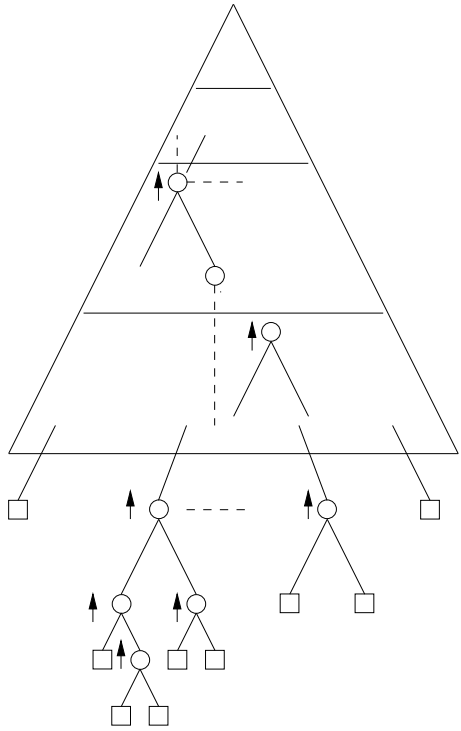


Abbildung 5.62: *Z*-stratifizierter Baum nach einer Reihe von Einfügungen mit noch nicht erfüllten Push-up-Forderungen

1. entweder einer lokalen strukturellen Änderung des *Z*-stratifizierten Baumes in der Schicht, in der der Vater des Knotens mit der Pull-down-Marke anschließend vorkommt und Halt

2. oder aber in einer rekursiven Verschiebung der Pull-down-Marke zum Vater des Knotens und keiner strukturellen Änderung im Baum.

Wir unterscheiden also wieder zwei Fälle je nachdem, wieviele Knoten in der unmittelbaren Verwandtschaft des Knotens *v* mit der Pull-down-Marke vorkommen.

Fall 1 [Es gibt genug Knoten in der Umgebung des Knotens *v* mit der Pull-down-Marke, vgl. Abbildung 5.64]

In diesem Fall kann die Pull-down-Forderung durch eine strukturelle Änderung, die nur wenige Knoten in der Umgebung des Knotens *v* betrifft, erledigt werden.

Um festzustellen, ob *Fall 1* vorliegt, inspizieren wir zunächst den Brudern *w* von *v*. *w* kann auf derselben Schicht wie sein Vater *p* auftreten oder eine Schicht darunter. (Beachte, daß *v* genau zwei Schichten unterhalb von *p* liegt.)

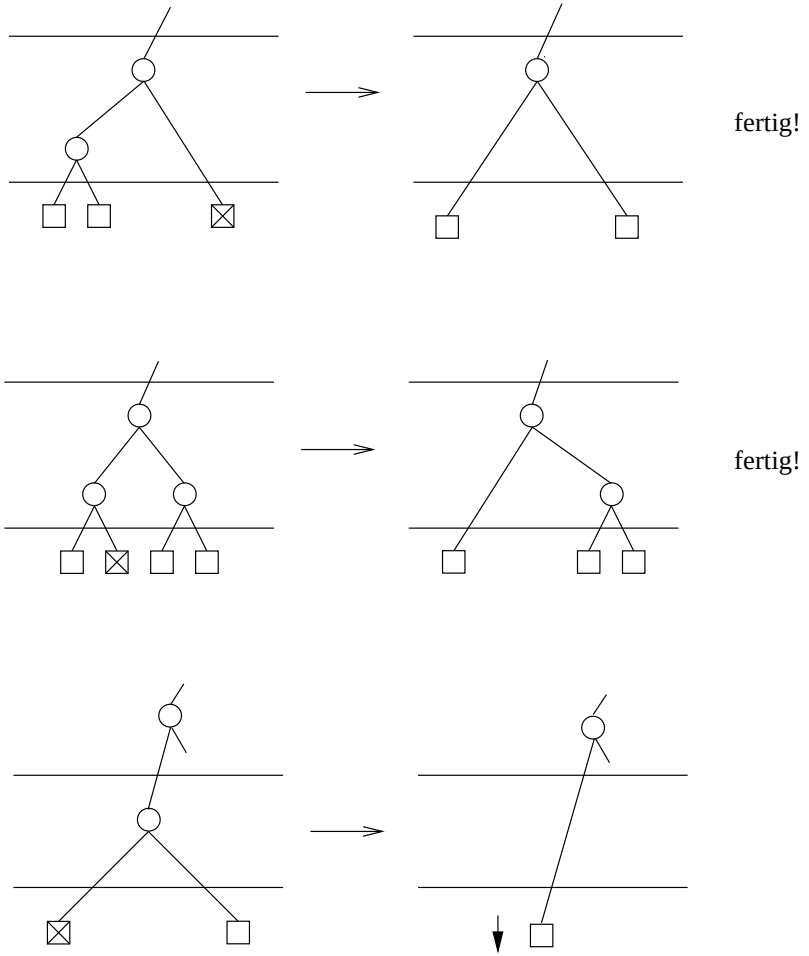


Abbildung 5.63: Löschen eines Schlüssels mit Setzen einer Pull-Down-Marke

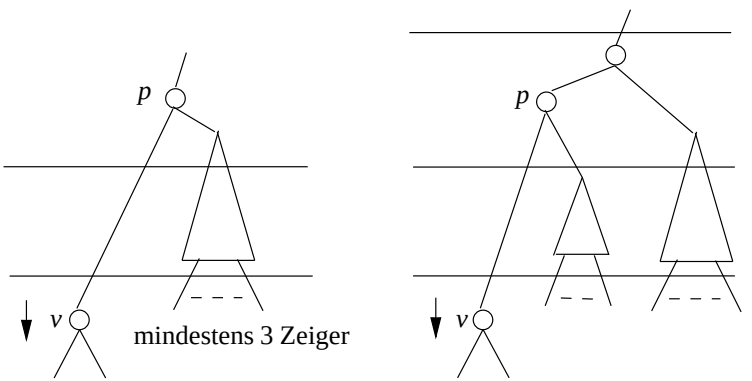


Abbildung 5.64: Der Knoten mit der Pull-down-Marke hat genug Knoten in seiner Umgebung

Wir betrachten zunächst den Fall, daß p und w in der gleichen Schicht liegen. Dann wissen wir, daß außer dem Zeiger, der p und v miteinander verbindet, wenigstens vier weitere Zeiger die Straßengrenze schneiden, unterhalb derer v liegt. Also ist es auf jeden Fall möglich, den Teilbaum mit Wurzel p durch einen neuen Straßenbaum aus Z zu ersetzen und die Teilbäume unterhalb von w so neu zu verteilen, daß v einen Vater auf der zwischen v und p liegenden Schicht erhält und die Z-stratifizierte Baumstruktur wiederhergestellt wird.

Um die Fallunterscheidung zu vereinfachen und die mehrfache Behandlung ähnlicher Fälle zu vermeiden, zeigen wir allerdings nicht explizit, wie in diesem Falle der Baum umzustrukturieren ist. Vielmehr führen wir die folgende Transformation durch, die den hier vorliegenden Fall auf einen anderen Fall reduziert, der ebenfalls unter *Fall 1* subsumierbar ist: Führe eine einfache Rotation bei p aus wie in Abbildung 5.65 (d) zu sehen, in der wieder alle symmetrischen Fälle weggelassen wurden. Man beachte, daß als Ergebnis dieser Rotation p einen Sohn auf der nächsten und den anderen Sohn v zwei Schichten unter seiner eigenen Schicht hat. Ferner treten p und der Vater von p auf der gleichen Schicht auf.

Wir können jetzt also annehmen, daß p und w auf verschiedenen Schichten auftreten. Das heißt, ein Sohn v von p ist zwei und der andere Sohn w von p eine Schicht unterhalb von p. Der Knoten w kann keinen, einen oder zwei Söhne auf derselben Schicht haben. Die letzteren beiden Fälle lassen sich unter *Fall 1* subsumieren und wie in Abbildung 5.65 (a) und (b) gezeigt behandeln, wobei wieder alle symmetrischen Fälle weggelassen wurden.

Im Falle, daß w keinen Sohn auf derselben Schicht hat, schauen wir nach oben zum Vater q von p. q kann auf derselben Schicht wie p auftreten. Dies ist ebenfalls eine Situation, die unter *Fall 1* subsumiert wird. Denn es ist in diesem Falle möglich, q den Sohn p wegzunehmen, so daß q dennoch Wurzel eines Straßenbaumes oberhalb von p bleibt, wie in Abbildung 5.65 (c) zu sehen.

Der einzige Fall, der nicht unter *Fall 1* subsumierbar ist, ist also eine Situation, in der der auf der Schicht unter der Schicht von p auftretende Knoten w keinen Sohn auf derselben Schicht wie w hat und in der p und der Vater q von p auf verschiedenen Schichten auftreten (p und w sind also jeweils Wurzeln von Bäumen aus Z mit der Höhe 1). Diese Situation bezeichnen wir als *Fall 2:*

Fall 2 [Es gibt nicht genügend Knoten in der Umgebung des Knotens v mit einer Pull-down-Marke]

In diesem Fall hat also der Knoten v mit der Pull-down-Marke die minimale Anzahl von Verwandten in seiner Umgebung. Wir können die Pull-down-Forderung daher nicht in der Umgebung von v erledigen. Also verschieben wir die Pull-down-Forderung von v auf den Vater p, indem wir einfach p unter die Straßengrenze oberhalb der p auftritt, hinunterziehen und die Pull-down-Marke bei p anbringen, vgl. Abbildung 5.66.

Man beachte, daß in diesem Fall keinerlei strukturelle Änderung (Änderung von Zeigern) ausgeführt wird. Ferner erfüllt der Knoten p offensichtlich die Invarianz-Bedingung, die wir oben für Knoten mit Pull-down-Marke formuliert haben, nämlich: p tritt unmittelbar unter einer Straßengrenze auf und der Vater von p liegt zwei Straßen oberhalb von p.

Wir nehmen übrigens stillschweigend an, daß eine Schicht an der Spitze des Z-stratifizierten Baumes verschwindet, wenn eine Pull-down-Marke den Sohn v der Wurzel p des Baumes erreicht hat und die Schicht zwischen dem Knoten v und seinem Vater

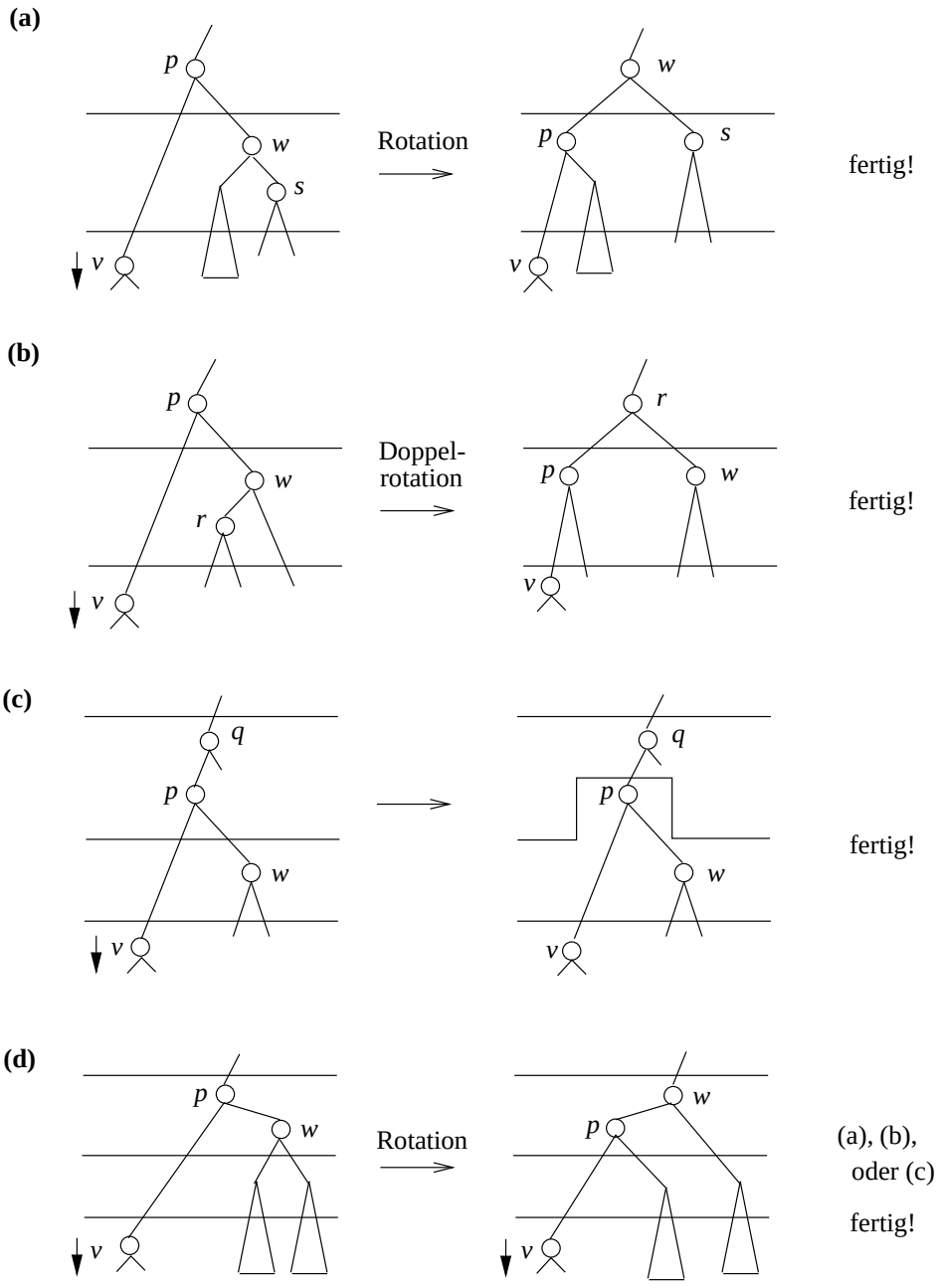


Abbildung 5.65: Lokale Umstrukurierungen bei einer Pull-down-Forderung

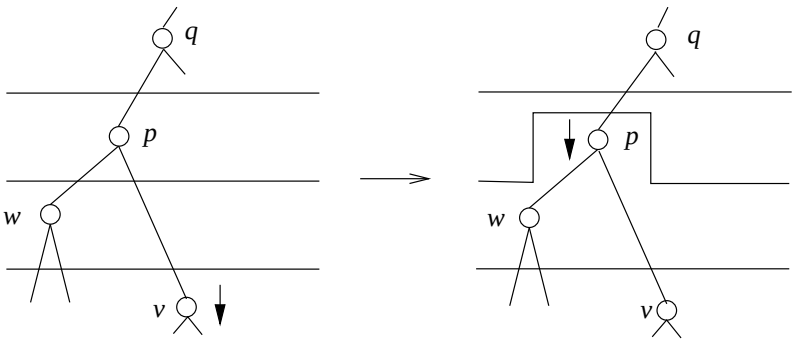


Abbildung 5.66: Rekursive Verschiebung einer Pull-down-Forderung zum nächsthöheren Niveau

p leer geworden ist. Denn in diesem Fall macht das Hinunterschieben des Knotens *p* unter die oberste Straßengrenze diese Grenze überflüssig.

Wie wir gesehen haben, führt eine einzelne Entfernung aus einem *Z*-stratifizierten Baum dazu, daß ein Blatt des Baumes mit einer Löschmarke versehen wird. Die Beseitigung dieser Löschmarke kann entweder unmittelbar durch eine auf die Umgebung dieses Blattes beschränkte strukturelle Änderung auf der untersten Schicht erfolgen, oder aber sie löst eine Pull-down-Forderung für den Bruder des entfernten Blattes aus. Pull-down-Forderungen (also Knoten mit Pull-down-Marken) können in dem Baum hochsteigen durch rekursive Verschiebung auf höhere Schichten, aber ohne strukturelle Änderungen, bis sie schließlich durch eine strukturelle Änderung beseitigt werden, die aber immer nur eine konstante Anzahl von Knoten und Zeigern betrifft.

Wir erläutern jetzt, wie eine *Folge von Entfernungen* in der Weise behandelt werden kann, so daß es nicht erforderlich ist, den Baum direkt nach jeder einzelnen Entfernung wieder umzustrukturieren. Zunächst beobachten wir, daß Entfernungen einfach dadurch akkumuliert werden können, daß man für jede Entfernung ein Blatt mit einer Löschmarke versieht und zunächst nichts weiter tut. Die Löschmarken können nun konkurrierend in beliebiger Reihenfolge beseitigt werden, wie oben beschrieben, so lange nur sichergestellt ist, daß die Beseitigung von mehreren Löschmarken niemals denselben Straßenbaum betrifft. Man muß sie nur nacheinander in beliebiger Reihenfolge behandeln durch die zuvor beschriebenen Rebalancierungsoperationen. Das impliziert insbesondere, daß die Beseitigung einer Löschmarke eines Knotens mit Pull-down-Marke (als Ergebnis einer vorher beseitigten Löschmarke), nicht erfolgen kann, bevor die Pull-down-Marke beseitigt oder im Baum weiter hochgestiegen ist. Beachtet man aber diese Bedingung, so ist gesichert, daß die Beseitigung zweier Löschmarken an den Blättern desselben *Z*-Straßenbaumes immer zu einem korrekten Ergebnis führt: Bevor die zweite Löschmarke beseitigt wird, hat eine Pull-down-Forderung den Vater des betroffenen Blattes eine Schicht hinuntergezogen, vgl. hierzu Abbildung 5.67 für eine graphische Erläuterung.

Kommen als Folge mehrerer beseitigter Löschmarken mehrere Pull-down-Marken an Knoten im Baum vor, so kann man sie stets konfliktfrei mit Hilfe der angegebenen Transformationen entweder beseitigen oder auf die nächsthöhere Schicht verschieben. Solange sie nicht denselben Baum aus *Z* betreffen, können sie sich nämlich nicht stören

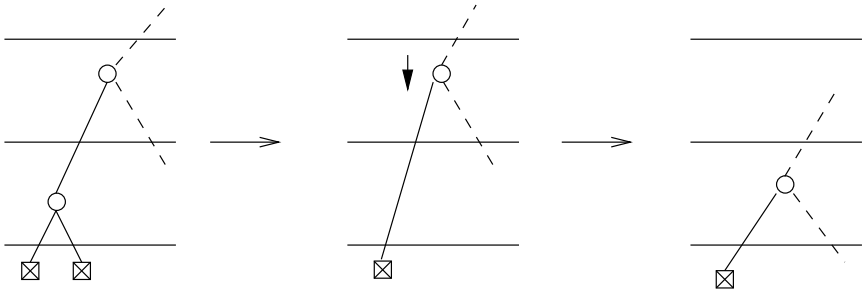


Abbildung 5.67: Beseitigung zweier Löschmarken an den Blättern desselben Z-Straßenbaumes

und man kann sie daher in beliebiger Reihenfolge behandeln. Kommt in der Umgebung eines Knotens mit Pull-down-Marke ein weiterer Knoten mit Pull-down-Marke vor, muß die weiter oben liegende Pull-down-Marke zuerst beseitigt werden. Dieses *Top-down-Vorgehen* zur Beseitigung mehrerer Pull-down-Marken ist immer möglich und korrekt mit Ausnahme eines einzigen Falls: Es kann als Ergebnis des rekursiven Verschiebens mehrerer Pull-down-Marken nach oben vorkommen, daß beide Söhne v und w eines Knotens p eine Pull-down-Marke haben und v und w zwei Schichten unter p liegen. Dann verschiebe man einfach p um eine Schicht nach unten, beseitige die Pull-down-Marken von v und w und bringe eine Pull-down-Marke bei p an, falls p keinen Vater auf seiner Schicht hat; sonst genügt bereits das Hinunterschieben von p, um beide Pull-down-Forderungen zu erfüllen. Das ist graphisch in Abbildung 5.68 gezeigt.

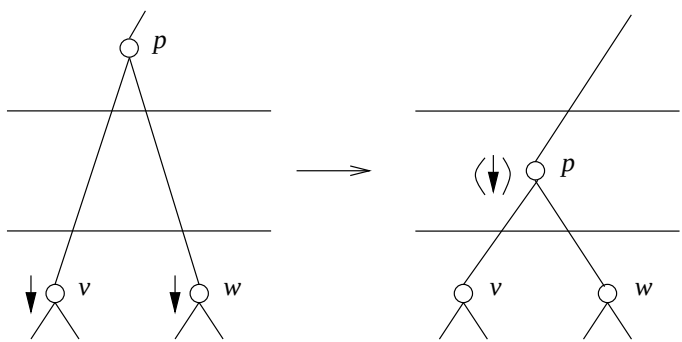


Abbildung 5.68: Gleichzeitiges Beseitigen von zwei Pull-down-Marken

Auf diese Weise wird sichergestellt, daß jede Folge von akkumulierten Entfernungen und die von Ihnen ausgelösten Umstrukturierungsprozesse beliebig verzahnt ausgeführt werden können, ganz genauso, als hätte man sie nacheinander (seriell) ausgeführt. Abbildung 5.69 zeigt schematisch einen nach einer Reihe von Entfernungen und strukturellen Änderungen entstandenen Z-stratifizierten Suchbaum.

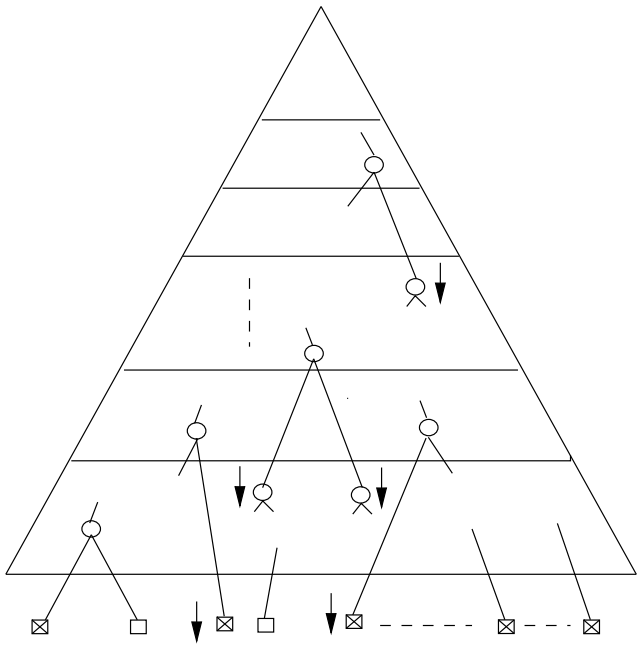


Abbildung 5.69: *Z*-stratifizierter Baum nach einer Reihe von Entfernungen mit noch nicht erfüllten Pull-down-Forderungen

Wie wir gesehen haben, wachsen und schrumpfen *Z*-stratifizierte Suchbäume also an der Wurzel. Neue Schlüssel wandern in den Baum von unten hinein, das heißt über die unterste Straßengrenze. Ebenso werden Schlüssel entfernt, indem man sie an der untersten Straßengrenze aus dem Baum herauszieht. Jetzt können wir erklären, wie beliebig verzahnte *Folgen von Einfügungen, Entfernungen und Umstrukturierungen* ausgeführt werden können.

Wenn eine Einfügung oder Entfernung in ein Blatt fällt, das unmittelbar unter der untersten Straßengrenze liegt, geschieht zunächst nichts Neues mit Ausnahme der Möglichkeit, daß jetzt eine Einfügung in ein Blatt fallen kann, das eine Löschmarke trägt. Es ist klar, wie man dann vorzugehen hat: Beseitige die Löschmarke und füge den Schlüssel an dieser Stelle wieder ein, siehe Abbildung 5.70.

Falls umgekehrt eine Entfernung in ein Blatt fällt, das durch eine frühere Einfügung entstanden und das noch nicht hinaufgewandert ist zur untersten Straßengrenze, kann man das Blatt und den zugehörigen inneren Knoten einfach entfernen und eine Pull-down-Marke beseitigen. Abbildung 5.71 zeigt ein Beispiel für dieses Ereignis.

Abgesehen von diesen geringfügigen Änderungen und Zusätzen ist nichts Neues erforderlich, um sicherzustellen, daß Einfügungen, Entfernungen und Rebalancierungsoperationen (das heißt also das Beseitigen von Push-up-, Lösch- und Pull-down-Marken) nebenläufig und beliebig verzahnt ausgeführt werden können. Man muß im Konfliktfall (wenn mehrere Push-up-, Pull-down- oder Löschmarken an Knoten in der-

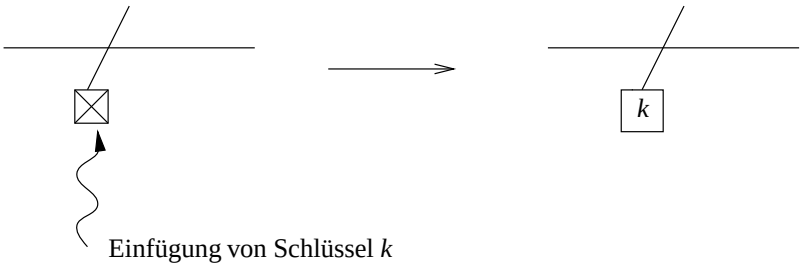


Abbildung 5.70: (Wieder-)Einfügung eines Schlüssels in ein Blatt mit Löschmarke

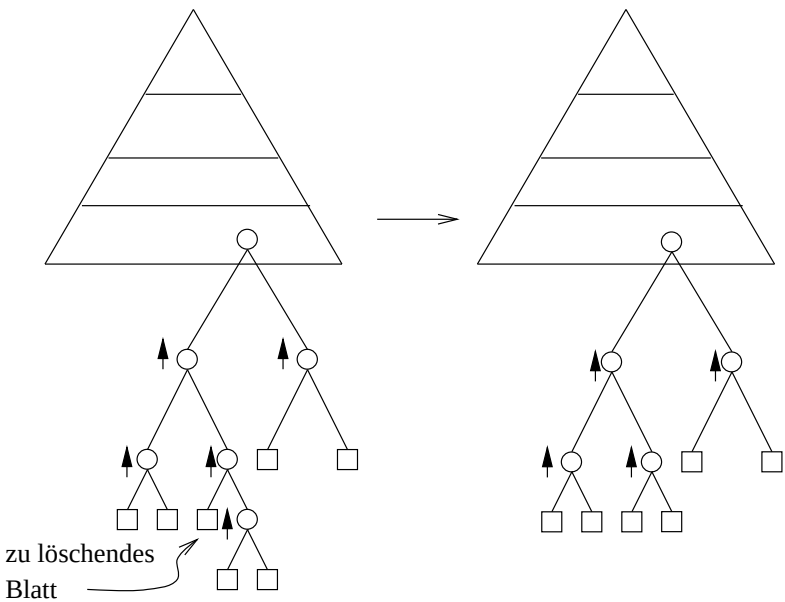


Abbildung 5.71: Entfernung eines durch Einfügung entstandenen Blattes

selben Umgebung vorkommen) nur darauf achten, der Top-down-Strategie zu folgen: Die jeweils weiter oben befindliche Marke muß ggfs. zuerst beseitigt werden. Das ist mit Hilfe der beschriebenen Transformationen immer möglich. Diese Überlegungen können im folgenden Satz zusammengefaßt werden:

Satz 5.4 *Sei T ein Z-stratifizierter Suchbaum, und sei eine beliebig verzahnte Folge von Einfügungen, Entfernungen und Transformationen zur Rebalancierung gegeben, die auf T angewandt wird. Dann ist die Anzahl der strukturellen Änderungen (Änderungen von Zeigern, die erforderlich sind, um die Balancebedingung für T wieder herzustellen, das heißt, um T wieder Z-stratifiziert zu machen) höchstens von der Größenordnung $O(i+d)$, wobei i die Anzahl der Einfügungen und d die Anzahl der Entfernungen ist.*

Wir sehen also, daß man mit derselben Anzahl struktureller Änderungen auskommt, die man auch aufzuwenden hätte, um einen gegebenen Baum jeweils unmittelbar nach einer Update-Operation wieder *Z*-stratifiziert zu machen.

Wir bemerken abschließend noch, daß keinerlei Umstrukturierungsoperationen erforderlich sind, wenn man zunächst eine Reihe von Einfügungen und dann eine Reihe von Entfernungen für einen gegebenen Baum ausführt und am Schluß der Baum wieder seine ursprüngliche Gestalt hat, ohne daß man zwischendurch irgendwelche Rebalancierungs-Operationen begonnen oder erledigt hat. Dies ist ein durchaus wichtiger Unterschied zu anderen in der Literatur vorgeschlagenen Verfahren zum relaxierten Balancieren.

5.6.3 Eindeutig repräsentierte Wörterbücher

Auch wenn eine Klasse von Bäumen durch eine statische Bedingung an die Struktur der Bäume festgelegt ist, kann es immer noch viele verschiedene Bäume in der Klasse geben, die sämtlich die gleiche Menge von Schlüsseln speichern. Wir können beginnend mit dem anfangs leeren Baum eine Reihe von Einfüge- und Entferne-Operationen ausführen, um schließlich einen Baum zu erhalten, der eine bestimmte Menge von Schlüsseln speichert. In der Regel hängt die Struktur dieses Baumes von seiner Entstehungsgeschichte, also von der Reihenfolge der Einfüge- und Entferne-Operationen ab.

Wir wollen jetzt Bäume als spezielle, durch Zeiger verbundene Graphen auffassen, die in ihren Knoten die Schlüssel speichern. Wir nennen ein Wörterbuch *mengen-eindeutig* repräsentiert, wenn jede Menge von Schlüsseln durch genau eine derartige Struktur repräsentiert ist. Bei Mengen-Eindeutigkeit kommt es also auf die Reihenfolge der Operationen, mit der man eine Struktur zur Speicherung einer gegebenen Schlüsselmenge erzeugt, nicht an. Es gibt genau einen Graphen, dessen Knoten die Schlüssel speichern.

Wir nennen ein Wörterbuch *größen-eindeutig* repräsentiert, wenn sogar jede Menge derselben Größe jeweils durch genau eine Struktur repräsentiert wird. Größen-Eindeutigkeit impliziert natürlich Mengen-Eindeutigkeit. Wir verlangen darüberhinaus stets, daß die Knoten des Graphen angeordnet sind und die Schlüssel der Größe nach in den Knoten mit aufsteigender Ordnungsnummer abgelegt sind. Wir bezeichnen diese Eigenschaft auch als *Ordnungs-Eindeutigkeit*.

Das Problem der eindeutigen Repräsentierung von Wörterbüchern besteht in der Suche nach möglichst effizienten Algorithmen zum Suchen, Einfügen und Entfernen von Schlüsseln für Wörterbücher, die mengen- oder größeneindeutig repräsentiert sind.

Ein einfaches Beispiel für eine größen-eindeutige Repräsentierung von Wörterbüchern sind sortierte, verkettet gespeicherte lineare Listen. Die im Abschnitt 5.3.2 beschriebenen randomisierten Suchbäume sind ein Beispiel für eine mengen-eindeutige aber nicht größen-eindeutige Repräsentation von Wörterbüchern. (Dabei unterstellen wir, daß die zur Berechnung der Prioritäten benutzte Hash-Funktion beliebig, aber fest gewählt ist.)

Man kann nun zeigen, daß die Forderung nach mengen- oder größen-eindeutiger Repräsentierung von Wörterbüchern zur Folge hat, daß wenigstens eine der drei Wörterbuchoperationen Suchen, Einfügen und Entfernen von Schlüsseln mehr als $O(\log n)$ Zeit für Strukturen mit n Schlüsseln benötigt. Es wurde erstmals von Snyder in [174] für eine große Klasse von Verfahren zum Suchen, Einfügen und Entfernen von Schlüsseln in Datenstrukturen gezeigt, daß die untere Grenze für den Aufwand zur Ausführung dieser Operationen bei eindeutig repräsentierten Datenstrukturen von der Größenordnung $\Omega(\sqrt{n})$ ist. Es ist also kein Zufall, daß AVL-Bäume, Bruder-Bäume, gewichtsbalancierte Bäume, B-Bäume und all die anderen zuvor genannten Klassen balancierter Bäume keine eindeutig repräsentierten Datenstrukturen sind. Der Wert dieser Aussage hängt natürlich stark von dem in diesem Zusammenhang benutzten Verfahrens- und Aufwandsbegriff ab. Natürlich sollten alle bekannten Verfahren zum Suchen, Einfügen und Entfernen von Schlüsseln in Listen, balancierte und unbalancierte Bäume aller Art darunter subsumierbar sein.

Snyder (vgl. [174]) gibt auch eine von ihm „Qualle“ genannte größen-eindeutige Struktur zur Repräsentation von Wörterbüchern an, die es erlaubt, jede der drei Wörterbuchoperationen in Zeit $O(\sqrt{n})$ auszuführen. Die in den Beweisen für die obere und untere Schranke zugelassenen Operationen stimmen aber nicht überein. Wir werden jetzt eine größen- und ordnungseindeutige Repräsentation von Wörterbüchern angeben, die die von Snyder angegebene untere Schranke im gewissen Sinne unterbietet.

Dazu betrachten wir eine größen- und ordnungseindeutige Repäsentation von Wörterbüchern durch Graphen mit begrenztem Ausgangsgrad (jeder Knoten hat höchstens die Ordnung k, k fest) und nehmen an, daß es für jede Zahl n genau einen Graphen mit n-Knoten gibt. Ferner unterstellen wir, daß die Knoten eines jeden Graphen eine feste Ordnung haben. Die Elemente einer gegebenen Menge von Schlüsseln der Größe n sind dann in den Knoten des Graphen in der Weise gespeichert, daß der i-größte Schlüssel im Knoten mit der Ordnungsnummer i abgelegt ist, für jedes i.

Jede Suche startet bei einem bestimmten Knoten, den wir die *Wurzel* nennen und läuft dann Kanten des Graphen entlang, bis das gesuchte Element in einem Knoten gefunden ist oder die Suche erfolglos endet. Alle Elemente müssen also von der Wurzel aus erreichbar sein. Daraus folgt sofort, daß jeder Knoten mit Ausnahme der Wurzel wenigstens eine in den Knoten hineinführende Kante hat. Die Kosten der Suche sind die Anzahl der bei der Suche durchlaufenen Kanten plus eins.

Wenn man eine Update-Operation ausführt, also eine Einfügung oder Entfernung, darf der Graph durch eine der folgenden Operationen verändert werden: Schaffen oder Entfernen eines Knotens, das Ändern, Hinzufügen oder Entfernen einer den Knoten verlassenden Kante (Zeiger-Änderung), Austauschen von Elementen zwischen zwei Knoten.

Jede dieser Operationen verlangt Kosten der Größenordnung $\Theta(1)$. In diesem Kostenmodell kann man nun die folgende untere Schranke beweisen, vgl. [9].

Satz 5.5 *Für jede größen- und ordnungseindeutige Repräsentation von Wörterbüchern durch Graphen benötigt wenigstens eine der drei Wörterbuchoperationen Zeit* $\Omega\left(n^{1/3}\right)$.

Wir verzichten auf einen Beweis dieses Satzes und zeigen vielmehr eine mit der im Satz behaupteten unteren Schranke übereinstimmende obere Schranke.

Halbdynamische *c*-Ebenen-Sprunglisten

Wir führen zunächst eine Variante der von Snyder in [174] eingeführten Struktur ein, die wir 2-Ebenen-Sprungliste nennen, für die dieselbe $O(\sqrt{n})$ Worst-case-Zeitschranke für alle drei Wörterbuchoperationen gilt. Um die Präsentation von 2-Ebenen-Sprunglisten zu vereinfachen, nehmen wir an, daß $i^2 \leq n < (i+1)^2$ für ein festes i ist. Das heißt, wir nehmen an, daß die Größe n des Wörterbuches nicht beliebig infolge von Einfügungen und Entfernungen schwanken kann, sondern immer zwischen gegebenen Schranken $i^2 \leq n < (i+1)^2$ für ein festes i bleibt. Eine 2-Ebenen-Sprungliste der Größe n besteht nun aus einer doppelt verketteten Liste von n Knoten $1, \ldots, n$. Für jedes p, $1 \leq p < n$ sind also die Knoten p und $p+1$ miteinander durch ein Paar von Zeigern auf Ebene 1 miteinander verknüpft. Wir nennen die Folge der durch Zeiger auf Ebene 1 miteinander verknüpften Knoten auch die 1-Ebenen-Liste. Ferner sind die Knoten 1, $i+1$, $2i+1$, …, $\lfloor n/i \rfloor \cdot i + 1$ miteinander zu einer 2-Ebenen-Liste verknüpft, die wir auch die *oberste* Ebenen-Liste nennen. Der letzte Knoten dieser Liste ist die *Wurzel* der 2-Ebenen-Sprungliste. Abbildung 5.72 zeigt die Struktur einer 2-Ebenen-Sprungliste.

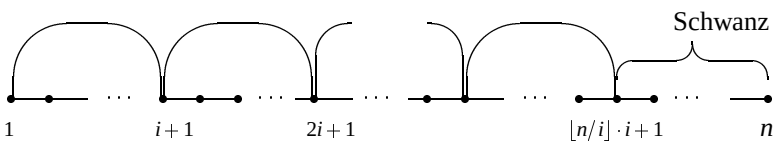


Abbildung 5.72: 2-Ebenen-Sprungliste der Größe n

Wir verlangen, daß die Elemente einer Menge mit n Schlüsseln in aufsteigender Ordnung in den Knoten 1, 2, …, n abgelegt sind. Damit haben wir also eine größen- und ordnungseindeutige Repräsentation von Wörterbüchern.

Nun sollte klar sein, wie man nach einem Schlüssel in einer solchen Struktur sucht und dabei höchstens $2i$ Schlüsselvergleiche ausführt: Man benutze ausgehend von der Wurzel die oberste Ebenen-Liste, um die Folge von höchstens i Knoten zu bestimmen, die den gesuchten Schlüssel enthalten kann und führe anschließend eine lineare Suche durch, indem man den Zeigern auf Ebene 1 folgt. Solange n im Bereich $i^2 \leq n < (i+1)^2$ bleibt, können Updates ebenfalls in Zeit $O(i)$ ausgeführt werden: Bestimme zuerst die Einfüge- oder Entferneposition in der 1-Ebenen-Liste. Das benötigt $O(i)$ Schritte. Dann füge das Element in diese Liste ein oder entferne es daraus. Das ist eine in konstanter Zeit ausführbare Operation. Sie hat zur Folge, daß eine Folge von Knoten auf Ebene

1, die von einem Zeiger auf der obersten Ebene übersprungen wird, entweder zu lang geworden ist (nach einer Einfügung) oder zu kurz (nach einer Entfernung). Also müssen einige Zeiger auf der obersten Ebene um eine Position nach links oder um eine Position nach rechts verschoben werden.

Abbildung 5.73 zeigt ein Beispiel einer Einfügung von Schlüssel 9 in eine 2-Ebenen-Sprungliste der Größe 11, die die Schlüssel $\{2,3,5,7,8,10,11,12,14,17,19\}$ speichert. Beachte, daß das Einfügen die Länge des Schwanzes der 2-Ebenen-Sprungliste um eins verlängert.

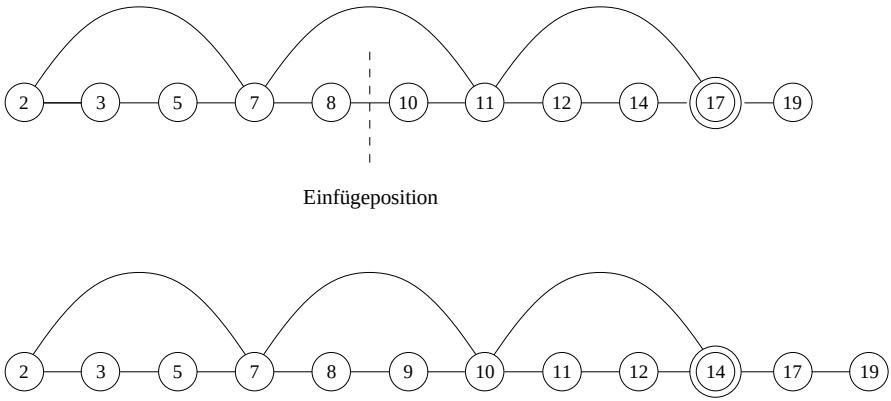


Abbildung 5.73: Einfügung von 9 in eine 2-Ebenen-Sprungliste

Folglich muß die oberste Ebenen-Liste um ein Element verlängert werden, sobald die Länge des Schwanzes i übersteigt. Analog kann eine Entfernung es erfordern, die oberste Ebenen-Liste um ein Element zu verkürzen. Das Adjustieren der obersten Ebenen-Liste nach einer Einfügung oder Entfernung ist aber in jedem Fall in $O(i)$ Schritten im schlechtesten Fall möglich.

So wie wir 2-Ebenen-Sprunglisten eingeführt haben, sind sie nur halbdynamisch, weil wir nicht erlaubt haben, daß ihre Größe n beliebig variieren darf. Es ist aber nicht allzuschwer, sich zu überlegen, daß man die Struktur auch volldynamisch machen kann, ohne daß man ihre wesentlichen Eigenschaften zerstört. Wir verzichten auf eine explizite Darstellung und verweisen dazu auf [9]. Statt dessen führen wir halbdynamische c-Ebenen-Sprunglisten für jedes $c \geq 3$ als natürliche Verallgemeinerung von 2-Ebenen-Sprunglisten ein. Wir nehmen also der Einfachheit halber wieder an, daß $i^c \leq n < (i+1)^c$ für ein festes i ist. Eine c-Ebenen-Sprungliste der Größe n besteht nun aus n Knoten $1, \ldots, n$. Die Knoten sind miteinander verknüpft durch Zeiger, die auf verschiedenen Ebenen verlaufen, nämlich auf unteren Ebenen und auf oberen Ebenen.

Untere Ebenen. Für jedes j, $1 \leq j \leq \lceil c/2 \rceil$, und jedes p, $1 \leq p \leq n - i^{j-1}$, sind die Knoten p und $p + i^{j-1}$ durch ein Paar von Zeigern auf Ebene j miteinander verknüpft.

Obere Ebenen. Für jedes j, $\lceil c/2 \rceil + 1 \leq j \leq c$, sind die Knoten 1, $1 \cdot i^{j-1} + 1$, $2 \cdot i^{j-1} + 1$, $3 \cdot i^{j-1} + 1$, $\ldots$ miteinander verknüpft, wobei höchsten $i^{j-1} - 1$ Knoten in

einem Schwanz übrig bleiben. Der letzte Knoten dieser obersten Ebenen-Liste ist die Wurzel.

Die Knoten einer c-Ebenen-Sprungliste, die durch Zeiger auf Ebene j miteinander verknüpft sind, bilden die Folge der j-Ebenen-Liste. Eine j-Ebenen-Liste hat maximal die Länge $\lfloor n/i^{j-1} \rfloor = O(i^{c-j+1})$. Man beachte den Unterschied zwischen den unteren und oberen Ebenen. In den unteren Ebenen ist jeder Knoten Teil einer j-Ebenen-Liste, während die oberen Ebenen jeweils nur eine j-Ebenen-Liste enthalten, die jede nur einige Knoten einschließen.

Abbildung 5.74 zeigt die Struktur einer 3-Ebenen-Sprungliste der Größe 30 mit zwei unteren und einer obersten Ebenen-Liste. Man beachte, daß eine c-Ebenen-Sprungliste der Größe n einen Speicherbedarf von $O(c \cdot n)$ hat.

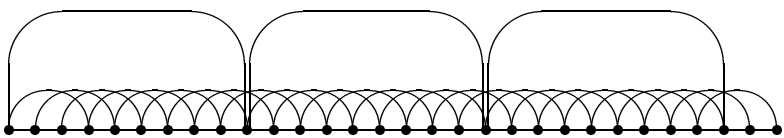


Abbildung 5.74: 3-Ebenen-Sprungliste der Größe 30

Wir verlangen wieder, daß die Schlüssel einer Menge von n Elementen in aufsteigender Reihenfolge in den Knoten $1, \ldots, n$ einer c-Ebenen-Sprungliste der Größe n abgelegt sind. Das ergibt dann eine größen- und ordnungseindeutige Repräsentation von Wörterbüchern.

Um nach einem Schlüssel zu *suchen*, beginnen wir bei der Wurzel in der obersten Ebenen-Liste und bestimmen die Folge von höchsten i^{c-1} Knoten, die den gesuchten Schlüssel enthalten können. Dann folgen wir für jedes $j = c-1, c-2, \ldots, 1$ einer Folge von Zeigern auf Ebene j, um die Position des gesuchten Schlüssels in der j-Ebenen-Liste zu bestimmen, bis wir den gesuchten Schlüssel gefunden haben oder j den Wert 1 bekommen hat und der gesuchte Schlüssel nicht an seiner erwarteten Position in der 1-Ebenen-Liste gefunden wurde. Beachte, daß für jedes j, $c-1 \geq j \geq 1$, die Suche beschränkt ist auf einen Teil der j Ebenen-Liste mit Länge höchstens i. So folgt, daß eine erfolgreiche oder erfolglose Suche in Zeit $O(c \cdot i) = O(c \cdot n^{1/c})$ im schlechtesten Fall ausführbar ist.

In Abbildung 5.75 ist ein möglicher Suchpfad in der 3-Ebenen-Sprungliste von Abbildung 5.74 durch fettgedruckte Zeiger dargestellt.

Um einen Schlüssel in eine c-Ebenen-Sprungliste *einzufügen*, bestimmt man zunächst die erwartete Position des neuen Schlüssels durch eine Suche wie vorher erläutert. Dann fügt man das neue Element in *alle* unteren j-Ebenen-Listen ein, $1 \leq j \leq \lceil c/2 \rceil$. Es werden alle Zeiger auf Ebene j, die über die Einfügeposition hinwegspringen, adjustiert; siehe Abbildung 5.76. Das heißt, eine Einfügeoperation kann aufgefaßt werden als ein gleichzeitiges Einfügen des neuen Elementes in i^{j-1} angeordnete, doppelt verkettete, lineare Listen für alle j, $1 \leq j \leq \lceil c/2 \rceil$. Das benötigt Zeit $O(1 + i + i^2 + \ldots + i^{\lceil c/2 \rceil - 1})$ $= O(i^{\lceil c/2 \rceil - 1})$ insgesamt. Dann müssen die Zeiger aller Knoten in den Listen auf den

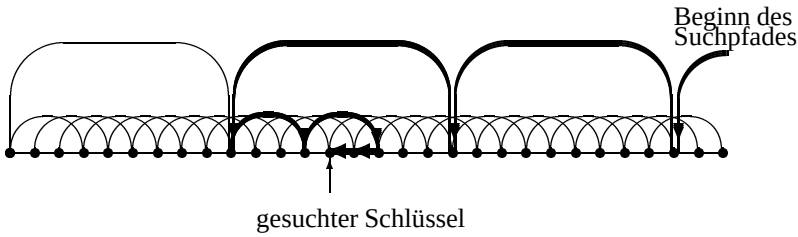


Abbildung 5.75: Beispiel eines möglichen Suchpfades

oberen Ebenen rechts von der Einfügeposition um eine Position nach links verschoben werden. Das benötigt Zeit $O(\sum_{j=\lceil c/2\rceil+1}^{c} n/i^{j-1}) = O(\sum_{j=\lceil c/2\rceil+1}^{c} i^{c-j+1}) = O(i^{\lfloor c/2\rfloor})$ im schlechtesten Fall. Die Gesamtkosten sind also $O(i^{\lceil c/2\rceil-1} + i^{\lfloor c/2\rfloor})$. Das führt zu zwei Fällen, je nachdem ob c gerade oder ungerade ist. Ist c gerade, benötigt das Einfügen Zeit $O(\sqrt{n})$, ist c ungerade, benötigt das Einfügen eines neuen Elementes in eine c-Ebenen-Sprungliste der Größe n Zeit $O(n^{(c-1)/2c})$. In jedem Fall ist die resultierende c-Ebenen-Sprung-Liste eine Liste der Größe $n+1$.

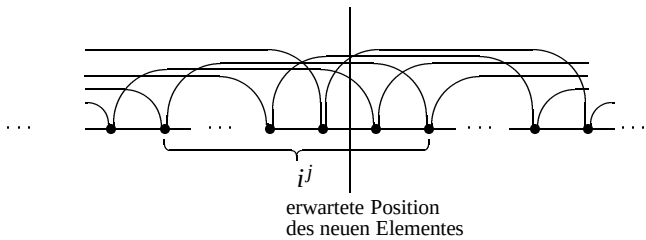


Abbildung 5.76: Auswirkungen durch eine Einfügung in eine j-Ebenen-Liste

Das Entfernen kann in völlig analoger Weise mit den gleichen asymptotischen Kosten durchgeführt werden. Man geht gerade umgekehrt wie beim Einfügen vor.

Auch hier kann man die Struktur volldynamisch machen, also die Beschränkung, daß n stets zwischen i^c und $(i+1)^c$ bleiben muß, fallen lassen. Dazu gibt es allgemeine Techniken, die hier nicht weiter erläutert werden. Insgesamt erhalten wir folgendes Resultat:

Satz 5.6 *Für jedes $c \geq 3$ sind c-Ebenen-Sprunglisten eine größen- und ordnungseindeutige Repräsentation von Wörterbüchern, die Platz $O(c \cdot n)$ beansprucht. Die Wörterbuchoperationen verlangen zu ihrer Ausführung höchstens die folgenden Kosten: Das Suchen ist ausführbar in der Zeit $O(c \cdot n^{1/c})$; Einfügen und Entfernen benötigen Zeit $O(\sqrt{n})$, wenn c gerade ist, und Zeit $O(n^{(c-1)/2c})$, wenn c ungerade ist.*

Wählt man in diesem Satz $c = 3$, erhält man das im Lichte von Snyder's Ergebnis [174] etwas überraschende Resultat, daß in 3-Ebenen-Sprunglisten jede der drei Wörterbuchoperationen in Zeit $O(n^{1/3})$ ausführbar ist.

5.7 Optimale Suchbäume

Suchbäume sind eine Datenstruktur zur Speicherung von Schlüsseln, so daß insbesondere die Such- (oder Zugriffs-)Operation effizient ausführbar ist. Wir haben bisher keinerlei Annahmen über die Zugriffs*häufigkeiten* gemacht und vielmehr darauf geachtet, daß auch die zwei anderen Wörterbuchoperationen, das Einfügen und Entfernen von Schlüsseln, effizient ausführbar sind. In diesem Abschnitt gehen wir davon aus, daß die Schlüsselmenge fest vorgegeben ist und die Zugriffshäufigkeiten sowohl für die Schlüssel, die im Baum vorkommen, als auch für die nicht vorhandenen Objekte im vorhinein bekannt sind. Es wird das Problem diskutiert, wie man unter diesen Annahmen einen „optimalen“, d.h. die Suchkosten minimierenden Suchbaum konstruieren kann. Dazu werden zunächst ein Kostenmaß zur Messung der Suchkosten und der Begriff des optimalen Suchbaumes präzise definiert. Dann werden ein Verfahren zur Konstruktion optimaler Suchbäume angegeben und dessen Laufzeit und Speicherbedarf analysiert.

Im allgemeinen hat man nicht nur Schlüssel k_i, nach denen mit Häufigkeit a_i (erfolgreich) gesucht wird, sondern man nimmt an, daß auch die Häufigkeiten b_j bekannt sind, mit denen nach „nicht vorhandenen“ Objekten im Intervall (k_j, k_{j+1}) erfolglos gesucht wird. Wir gehen also von folgender Situation aus:

$S = \{k_1, \ldots, k_N\}$ Menge von N verschiedenen Schlüsseln, $k_1 < k_2 < \ldots < k_N$.

$a_i =$ (absolute) Häufigkeit, mit der nach k_i gesucht wird, $1 \leq i \leq N$.

$I = (k_0, k_{N+1})$ offenes Intervall aller Schlüssel, nach denen — erfolgreich oder erfolglos — gesucht wird; es gilt $k_0 < k_1$ und $k_N < k_{N+1}$. Typische Werte sind $k_0 = -\infty$ und $k_{N+1} = +\infty$.

$b_j =$ (absolute) Häufigkeit, mit der nach einem $x \in (k_j, k_{j+1})$ gesucht wird, mit $0 \leq j \leq N$.

In einem Suchbaum für S bezüglich I sind die k_i die Werte der inneren Knoten. Die Intervalle zwischen den Schlüsseln werden durch die Blätter repräsentiert. Als Maß für die gesamten Suchkosten eines Baumes nimmt man üblicherweise die *gewichtete Pfadlänge*, die mit Hilfe des Gewichtes eines Baumes definiert ist:

$$W = \sum_i a_i + \sum_j b_j$$

heißt das *Gewicht* des Baumes, und

$$P = \sum_{i=1}^{N} (\text{Tiefe}(k_i) + 1)\, a_i + \sum_{j=0}^{N} \text{Tiefe}(\text{Blatt}(k_j, k_{j+1}))\, b_j$$

heißt *gewichtete Pfadlänge* des Baumes.

Beispiel: Gegeben sei eine Menge von vier Schlüsseln mit folgenden Zugriffshäufigkeiten für die Schlüssel und Intervalle:

$(-\infty,k_1)$	k_1	(k_1,k_2)	k_2	(k_2,k_3)	k_3	(k_3,k_4)	k_4	(k_4,∞)
4	1	0	3	0	3	0	3	10

Ein möglicher Suchbaum für diese Menge ist in Abbildung 5.77 angegeben. Der Baum hat die gewichtete Pfadlänge 48.

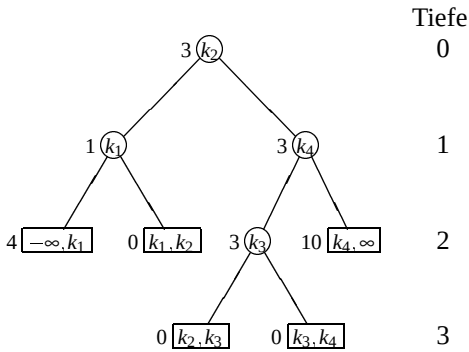


Abbildung 5.77

Die gewichtete Pfadlänge mißt, wieviele Schlüsselvergleiche für die erfolgreichen und erfolglosen Such-Operationen insgesamt ausgeführt werden. Jeden im Baum gespeicherten Schlüssel k_i findet man mit Tiefe$(k_i)+1$ Schlüsselvergleichen wieder. Sucht man nach einem $x \in (k_j,k_{j+1})$, also nach einem Schlüssel, der im Baum nicht vorkommt, muß man bei der üblichen Implementation von Bäumen (Blätter werden durch **nil**-Zeiger in ihren Vätern repräsentiert) genau Tiefe(k_j,k_{j+1}) Schlüsselvergleiche ausführen, um festzustellen, daß x im Baum nicht vorkommt.

Bemerkung: Statt der absoluten Häufigkeiten verwendet man oft auch die relativen Suchhäufigkeiten $\alpha_i = a_i/W$ und $\beta_j = b_j/W$ und betrachtet statt P die *normierte gewichtete Pfadlänge* P/W.

Seien nun N Schlüssel k_i, $1 \le i \le N$, mit Häufigkeiten a_i, $1 \le i \le N$, ein Schlüsselintervall $I = (k_0,k_{N+1})$ mit $k_0 < k_1$ und $k_N < k_{N+1}$ und b_j, $0 \le j \le N$, gegeben. Ein Suchbaum T für $S = \{k_1,\ldots,k_N\}$ bezüglich I heißt *optimal*, wenn seine gewichtete Pfadlänge minimal ist unter allen Suchbäumen für S bezüglich I.

Wir wollen jetzt ein Verfahren zur Konstruktion optimaler Suchbäume angeben. Es beruht wesentlich auf der folgenden Beobachtung: Jeder Teilbaum eines optimalen Suchbaumes ist selbst ein optimaler Suchbaum.

Das folgt unmittelbar aus der folgenden, rekursiven Berechnungsmethode für die gewichtete Pfadlänge. Ist T ein Baum mit linkem Teilbaum T_l und rechtem Teilbaum T_r, so kann man die gewichtete Pfadlänge $P(T)$ des Baumes T wie folgt aus den gewichteten Pfadlängen $P(T_l)$ und $P(T_r)$, den Gewichten der Teilbäume und der Zugriffshäufigkeit für die Wurzel berechnen:

$$\begin{aligned} P(T) &= P(T_l) + \mathit{Gewicht}(T_l) \\ &\quad + \text{Zugriffshäufigkeit der Wurzel} \\ &\quad + P(T_r) + \mathit{Gewicht}(T_r) \\ &= P(T_l) + P(T_r) + \mathit{Gewicht}(T) \qquad (*) \end{aligned}$$

Ist dabei für $S = \{k_1, \ldots, k_N\}$ und $I = (k_0, k_{N+1})$ der Schlüssel an der Wurzel k_l, $1 \leq l \leq N$, so ergibt sich als Schlüsselmenge für den linken Teilbaum $S' = \{k_1, \ldots, k_{l-1}\}$ und als Schlüsselintervall $I' = (k_0, k_l)$; entsprechend ergibt sich für den rechten Teilbaum $S'' = \{k_{l+1}, \ldots, k_N\}$ und $I'' = (k_l, k_{N+1})$.

Falls T ein Blatt ist, gilt natürlich $P(T) = 0$.

Wir teilen nun den gesamten Suchraum $(-\infty, k_1)k_1(k_1, k_2)k_2 \ \ldots \ k_{N-1} \ (k_{N-1}, k_N)$ $k_N(k_N, \infty)$ in immer größere, zusammenhängende Teile ein, für die wir jeweils einen optimalen Suchbaum konstruieren. D.h. wir berechnen größere optimale Teilbäume aus kleineren. Sei

$T(i,j)$ optimaler Suchbaum für $(k_i, k_{i+1})k_{i+1} \ldots k_j(k_j, k_{j+1})$,
$W(i,j)$ das Gewicht von $T(i,j)$, also $W(i,j) = b_i + a_{i+1} + \ldots + a_j + b_j$,
$P(i,j)$ die gewichtete Pfadlänge von $T(i,j)$.

Wegen $(*)$ kann man offenbar den optimalen Suchbaum $T(i,j)$ und seine gewichtete Pfadlänge $P(i,j)$ berechnen, sobald man den Index l der Wurzel von $T(i,j)$ kennt. Das zeigt Abbildung 5.78.

$T(i,j)$, $W(i,j)$, $P(i,j)$ sind definiert für alle $j \geq i$. Falls $j = i$ ist, besteht $T(i,j)$ nur aus dem Blatt (k_i, k_{i+1}). Es gilt:

$$(i) \quad \begin{cases} W(i,i) = b_i = \text{ Häufigkeit, mit der nach } x \in (k_i, k_{i+1}) \\ \qquad\qquad \text{gesucht wird} \qquad (0 \leq i \leq N) \\ W(i,j) = W(i,j-1) + a_j + b_j \qquad (0 \leq i < j \leq N) \end{cases}$$

$$(ii) \quad \begin{cases} P(i,i) = 0 \qquad (0 \leq i \leq N) \\ P(i,j) = W(i,j) + \min\limits_{i<l\leq j}\{P(i,l-1) + P(l,j)\} \qquad (0 \leq i < j \leq N) \end{cases}$$

Sei $r(i,j)$ diejenige Zahl, für die das Minimum angenommen wird, also der Index der Wurzel von $T(i,j)$. Gesucht ist $T(0,N)$; dieser Baum ist durch die Zahlen $r(i,j)$, $0 \leq i < j \leq N$, offenbar völlig bestimmt. Die beiden Gleichungen (i) und (ii) legen unmittelbar ein Verfahren zur (simultanen) Berechnung von $W(i,j)$, $P(i,j)$, $r(i,j)$ für alle i und j mit $0 \leq i \leq j \leq N$ nahe: Definieren wir die *Breite h* eines Baumes als die Anzahl der im Baum gespeicherten Schlüssel, so haben die Bäume $T(i,j)$ die Breite $h = j - i$ für alle i, j mit $0 \leq i \leq j \leq N$. Daher können wir $W(i,j)$, $P(i,j)$, $r(i,j)$ durch Induktion über $h = j - i$ wie folgt berechnen:

Fall 1 $[h = j - i = 0]$

Dann ist $j = i$, also $T(i,i) = \boxed{k_i, k_{i+1}}$, $0 \leq i \leq N$.

Setze $W(i,i) := b_i$, $P(i,i) := 0$, $r(i,i)$ undefiniert.

Fall 2 $[h = j - i = 1]$
Dann ist $j = i+1$, und $T(i,i+1)$ hat den Schlüssel k_{i+1} an der Wurzel, $0 \le i < N$. Abbildung 5.79 zeigt diese Situation.

Setze

$$\begin{aligned} W(i,i+1) &:= W(i,i) + a_{i+1} + W(i+1,i+1), \\ P(i,i+1) &:= W(i,i+1), \\ r(i,i+1) &:= i+1. \end{aligned}$$

Fall 3 $[h = j - i \ge 2]$
Für jedes $i,\ j$ mit $h = j - i \ge 2$, $0 \le i < j \le N$, bestimmen wir dasjenige l (das größte, falls es mehrere gibt), $i < l \le j$, für das $P(i,l-1) + P(l,j)$ minimal wird. Wegen $(l - 1 - i) < h$ und $(j - l) < h$ können wir dabei voraussetzen, daß alle in Frage kommenden Werte $P(i,l-1)$ und $P(l,j)$ bereits bekannt sind.

Setze

$$\begin{aligned} W(i,j) &:= W(i,l-1) + W(l,j) + a_l, \\ P(i,j) &:= P(i,l-1) + P(l,j) + W(i,j), \\ r(i,j) &:= l. \end{aligned}$$

Das beschriebene Verfahren benötigt drei Felder zur Speicherung der Werte von $W(i,j)$, $P(i,j)$, $r(i,j)$; es hat also Platzbedarf $\Theta(N^2)$.

Die Fälle $h = 0$ und $h = 1$ lassen sich in $O(N)$ Schritten erledigen. Zur Ausführung der im Fall $h \ge 2$ angegebenen Operationen reichen $O(N^3)$ Schritte. Um das einzusehen, können wir annehmen, daß alle Gewichte $W(i,j)$ für $0 \le i \le j \le N$ bereits (in höchstens $O(N^2)$ Schritten) berechnet wurden. Dann beschreibt das folgende Programmstück die im Fall 3 auszuführenden Operationen:

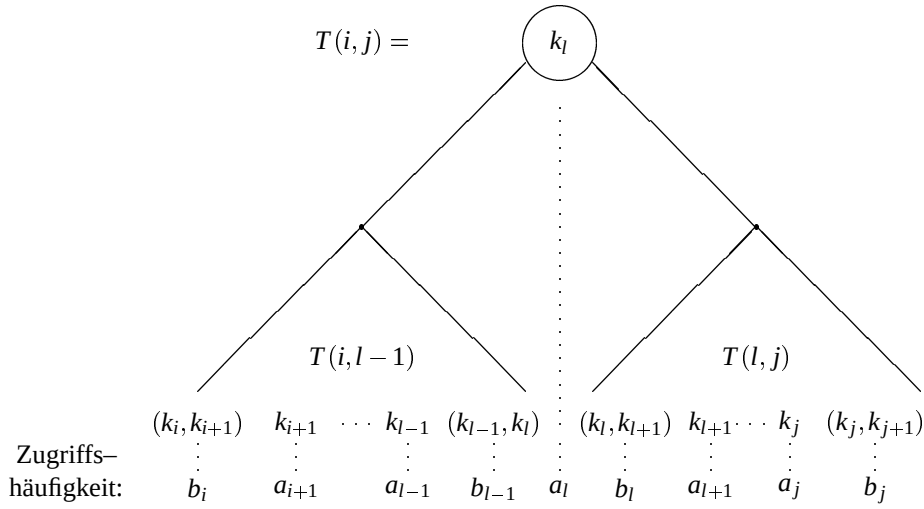


Abbildung 5.78

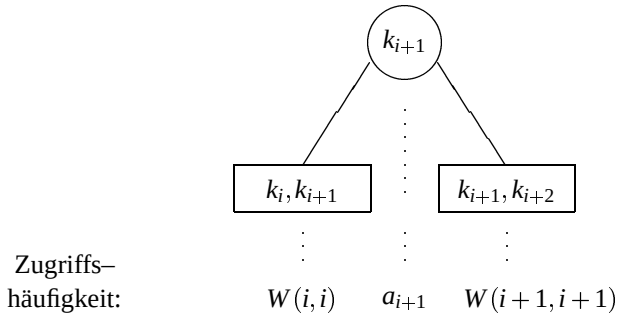


Abbildung 5.79

```
for h := 2 to N do
  for i := 0 to (N − h) do
    begin
      j := i + h;
      finde das (größte) l, i < l ≤ j, für das
        P(i,l−1) + P(l, j) minimal wird;
      P(i, j) := P(i,l−1) + P(l, j) + W(i, j);
      r(i, j) := l
    end
```

In der inneren **for**-Schleife ist das Minimum von $h = j - i$ Elementen zu bestimmen; alle anderen Operationen sind jeweils in konstanter Schrittzahl ausführbar. Das ergibt folgende Abschätzung für die Gesamtschrittzahl:

$$\sum_{h=2}^{N} \sum_{i=0}^{N-h} O(h+1) = \sum_{h=2}^{N} O\left((N-h)(h+1)\right) = O(N^3)$$

Für große N ist das natürlich in vielen Fällen nicht effizient genug. Man erhält eine Verbesserung durch Ausnutzen des sogenannten *Monotonieprinzips*. Man kann zeigen, daß für alle $i,\ j$ mit $0 \le i < j \le N$ gilt:

$$r(i, j-1) \le r(i, j) \le r(i+1, j)$$

(Einen Hinweis auf den Beweis findet man z.B. in [89].)

Dann genügt es, in der inneren **for**-Schleife zum Auffinden desjenigen l, für das $P(i,l-1) + P(l, j)$ minimal wird (bzw. des größten l mit dieser Eigenschaft, wenn es mehrere solche l gibt), l aus dem Bereich $r(i, j-1) \le l \le r(i+1, j)$ zu betrachten. Für festes h ist dann die innere **for**-Schleife in folgender Schrittzahl ausführbar:

$$
\begin{aligned}
& O\left(N-h+\sum_{i=0}^{N-h}(r(i+1,i+h)-r(i,i+h-1)+1)\right) \\
&= O(N-h+r(1,h)-r(0,h-1)+1 \\
&\quad +r(2,h+1)-r(1,h)+1 \\
&\quad +r(3,h+2)-r(2,h+1)+1 \\
&\quad \vdots \\
&\quad +r(N-h+1,N)-r(N-h,N-1)+1) \\
&= O(N-h+N-h+1+(\underbrace{r(N-h+1,N)-r(0,h-1)}_{\leq N})) \\
&= O(N)
\end{aligned}
$$

Damit ist das Verfahren insgesamt in $O(N^2)$ Schritten ausführbar.

Beispiel (Fortsetzung): Wir geben die Belegung der Felder $W(i,j)$, $P(i,j)$, $r(i,j)$, mit $0 \leq i \leq j \leq 4$, für die am Anfang dieses Abschnitts und in Abbildung 5.77 angegebenen Schlüssel, Intervalle und Suchhäufigkeiten an. Zunächst berechnet man die Belegung des Feldes $W(i,j)$, vgl. Tabelle 5.3.

$W(i,j)$

$i\backslash j$	0	1	2	3	4
0	4	5	8	11	24
1		0	3	6	19
2			0	3	16
3				0	13
4					10

Tabelle 5.3

Nun berechnet man der Reihe nach für wachsendes $h := j - i$ die Werte von $P(i,j)$ und $r(i,j)$. Das Ergebnis zeigt Tabelle 5.4. Aus den Werten des Feldes $r(i,j)$ kann man den Suchbaum $T(0,4)$ in Abbildung 5.80 ablesen. Dieser Baum $T(0,4)$ hat die gewichtete Pfadlänge $P(0,4) = 43$.

Leider sind die (absoluten oder relativen) Zugriffshäufigkeiten für eine konkrete Folge von Schlüsseln und Intervallen meistens nicht genau bekannt. Man ist dann auf Schätzungen angewiesen. Für sehr große N — man denke etwa an ein Lexikon mit vielen hunderttausend Einträgen — ist auch ein in $O(N^2)$ Schritten ausführbarer Algorithmus viel zu langsam. Statt einen optimalen Suchbaum nach der beschriebenen Methode

$P(i,j)$

$i\backslash j$	0	1	2	3	4
0	0	5	11	19	43
1		0	3	9	28
2			0	3	19
3				0	13
4					0

$r(i,j)$

$i\backslash j$	0	1	2	3	4
0	–	1	1	2	4
1		–	2	3	4
2			–	3	4
3				–	4
4					–

Tabelle 5.4

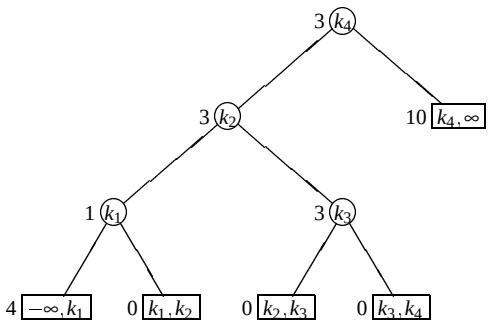


Abbildung 5.80

zu konstruieren, begnügt man sich daher damit, einen „fast optimalen“ Suchbaum möglichst effizient, d.h. möglichst in $O(N)$ Schritten zu konstruieren. Die bekannten Konstruktionsverfahren folgen meistens naheliegenden Strategien, wie z.B. der folgenden: Wähle die Wurzel eines Teilbaums stets so, daß die Summe der Zugriffshäufigkeiten für alle Schlüssel im linken und rechten Teilbaum sich möglichst wenig unterscheidet. Wir verzichten auf eine genauere Diskussion solcher Verfahren und eine präzise Definition des Begriffs „fast optimal“. Der interessierte Leser konsultiere z.B. [123].

5.8 Alphabetische und mehrdimensionale Suchbäume

Außer den bisher vorgestellten Varianten von Bäumen gibt es eine große Zahl weiterer. Wir diskutieren in diesem Abschnitt kurz sogenannte alphabetische Suchbäume (englisch: tries) und mehrdimensionale Suchbäume. In beiden Fällen wird nicht mehr

vorausgesetzt, daß die in einer Baumstruktur zu speichernden Daten durch je einen einzigen, als Einheit zu betrachtenden Schlüssel charakterisiert werden können. Alphabetische Suchbäume benutzen die Darstellung von Schlüsseln als Ziffern- oder Buchstabenfolgen; mehrdimensionale Suchbäume sind Strukturen zur Speicherung von Objekten, wie z.B. Punkten in der Ebene, die sich auf natürliche Weise durch zusammengesetzte Schlüssel, wie z.B. ein Paar kartesischer Koordinaten, charakterisieren lassen.

5.8.1 Tries

Das Wort „Trie“ wird üblicherweise wie das englische Wort „try“ gesprochen, um es vom Wort „tree“ unterscheiden zu können. Es hat seinen Ursprung im Wort re*trie*val, das auf die Verwendung von Tries als Suchstruktur hinweist. Wir erläutern die Tries zugrundeliegende Idee an einem Beispiel. Die Menge der Wörter {wer, weiß, wo, wir, sind} kann durch einen alphabetischen Suchbaum wie in Abbildung 5.81 repräsentiert werden.

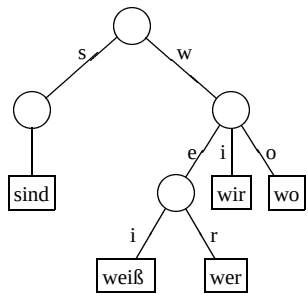


Abbildung 5.81

Wir fassen die Wörter also als Buchstabenfolgen auf, verzweigen genau dort, wo verschiedene Buchstaben unterschieden werden müssen, und erhalten so eine Struktur, in der wir sämtliche Wörter durch buchstabenweises Vergleichen wiederfinden können. Suchen wir etwa nach dem Wort „weiß“, folgen wir zunächst dem Verweis für den ersten Buchstaben, d.h. dem w-Zeiger, dann dem Verweis für den zweiten Buchstaben usw., bis wir bei dem gesuchten Wort angekommen sind. Das Suchen (und Einfügen) in alphabetischen Suchbäumen besteht also darin, im *i*-ten Schritt dem Zeiger für den *i*-ten Buchstaben zu folgen, und das solange zu wiederholen, bis man genügend Buchstaben inspiziert hat, um das gesuchte Wort von allen anderen im alphabetischen Suchbaum zu unterscheiden.

Das genannte Beispiel entspricht insofern nicht dem allgemeinen Fall, als kein Wort Präfix eines anderen ist. Beispielsweise ist es nicht ohne weiteres möglich, das Wort „Wort“ im obigen alphabetischen Suchbaum unterzubringen. Man macht daher zunächst alle Wörter künstlich, durch explizite Berücksichtigung des Leerzeichens, gleich

lang und kann dann *alle* Wörter einer gegebenen Länge h in einem alphabetischen Suchbaum der Höhe h mit n^h Blättern repräsentieren, wobei n die Alphabetgröße bezeichnet.

Soll, wie im angegebenen Beispiel, nur eine sehr kleine Teilmenge des riesigen Universums aller möglichen Schlüssel mit Länge h über dem Alphabet von n Buchstaben repräsentiert werden, wählt man natürlich eine möglichst speicherplatzsparende Implementation. Das bedeutet zweierlei. Erstens werden für großes n nicht alle n möglichen Verzweigungen explizit repräsentiert. Zweitens werden unäre Verweisketten (wie im Beispiel) soweit wie möglich verkürzt.

Besonderes Interesse haben binäre Tries, also alphabetische Suchbäume für Wörter über dem binären Alphabet $\{0, 1\}$ gefunden, weil sie eng mit binären Codes zusammenhängen, die zur Datenkomprimierung Verwendung finden. Offenbar kann man jeden binären Trie als binären Code-Baum für die Werte der Blätter auffassen, wie in folgendem Beispiel (vgl. Abbildung 5.82).

000 codiert „sind“, 10 codiert „wer“ usw. „Wer weiß wo wir sind“ wird also codiert durch 100010111000.

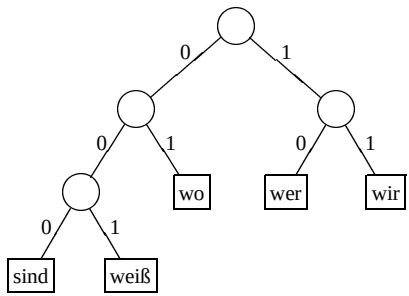


Abbildung 5.82

Ein Code ist dann besonders gut, wenn häufig vorkommende Wörter besonders kurze Codes haben. Es gibt eine Reihe von Verfahren, um für bekannte Häufigkeitsverteilungen in diesem Sinne „optimale“ Codes als binäre Tries zu finden.

5.8.2 Quadranten- und 2d-Bäume

Eine Menge angeordneter Schlüssel kann man auffassen als eine Menge von Punkten auf der Linie. Suchbäume sind also Strukturen zur Speicherung von Punkten im eindimensionalen Raum derart, daß sich die für Punkte typischen Operationen Suchen, Einfügen und Entfernen effizient ausführen lassen. Es gibt eine Reihe von Vorschlägen zur Verallgemeinerung von Suchbäumen. Sie haben zum Ziel, Punkte im 2-, 3-, — allgemein im k-dimensionalen Raum — so abzuspeichern, daß wieder die für Punkte typischen Operationen gut unterstützt werden. Natürlich kann man auch Punkte im k-dimensionalen Raum, $k \geq 2$, in gewöhnlichen Suchbäumen abspeichern. Man bildet

dazu einfach aus den k Koordinaten einen einzigen, den jeweiligen Punkt eindeutig charakterisierenden „Superschlüssel“ und benutzt diesen Schlüssel für die Operationen Suchen, Einfügen und Entfernen. Solange man also keine anderen Operationen ausführen will, besteht keine Notwendigkeit zur Verallgemeinerung von Suchbäumen. Typischerweise möchte man aber für Punkte im k-dimensionalen Raum, $k \geq 2$, auch andere Operationen ausführen. Beispiele für solche Operationen sind:

Bereichsanfrage (englisch: *range query*): Gegeben sei ein k-dimensionaler, rechteckiger Bereich (ein „Hyperrechteck“, wenn $k > 2$ ist). Die Aufgabe besteht darin, alle gespeicherten Punkte zu berichten, die in den Bereich fallen. Dabei wird üblicherweise angenommen, daß die Bereichsgrenzen parallel zu kartesischen Koordinaten sind.

Partielle Bereichssuche (englisch: *partial match query*): Gegeben sind i Koordinatenwerte, $i < k$. Gesucht sind alle gespeicherten Punkte, die für die gegebenen Koordinaten die gegebenen Werte und für die restlichen Koordinaten beliebige Werte haben.

Dies sind Beispiele für typisch geometrische Suchoperationen. Eine gut gewählte Suchstruktur sollte auf geometrische Nachbarschaftsbeziehungen möglichst Rücksicht nehmen, um solche geometrischen Operationen zu unterstützen. Wir besprechen zwei derartige Strukturen für den Fall $k = 2$. Die Verallgemeinerung für $k > 2$ ist offensichtlich. Wir erläutern, wie man eine Menge von Punkten in der Ebene der Reihe nach in einen anfangs leeren Quadranten-Baum bzw. 2d-Baum iteriert einfügt analog zum Einfügen in natürliche Bäume.

Quadranten-Bäume

Seien N Punkte $P_1, P_2, \ldots, P_N$ in der Ebene gegeben. Die Punkte lassen sich wie folgt in einen Baum der Ordnung 4 einfügen. P_1 wird in der Wurzel gespeichert. Ein durch P_1 gelegtes Koordinatenkreuz zerlegt die Ebene in vier Quadranten (vgl. Abbildung 5.83).

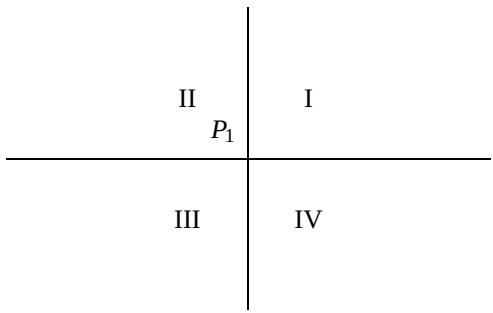


Abbildung 5.83

Die Wurzel erhält vier Zeiger auf Söhne, einen für jeden Quadranten. Der nächste Punkt P_2 wird i-ter Sohn von P_1, wenn P_2 in den i-ten Quadranten bzgl. P_1 fällt. Entsprechend fährt man für die übrigen Punkte fort. D.h. der jeweils nächste Punkt wird i-ter Sohn seines Vaters, wenn er in den i-ten durch den Vater definierten Quadranten

fällt und der Vater nicht bereits einen i-ten Sohn besitzt. Hat der Vater schon einen i-ten Sohn, so wird das Einfügen bei diesem Sohn fortgesetzt.

Betrachten wir als Beispiel die sieben Punkte $A = (7,9)$, $B = (15,14)$, $C = (10,5)$, $D = (3,13)$, $E = (13,6)$, $F = (17,2)$, $G = (3,2)$ in Abbildung 5.84.

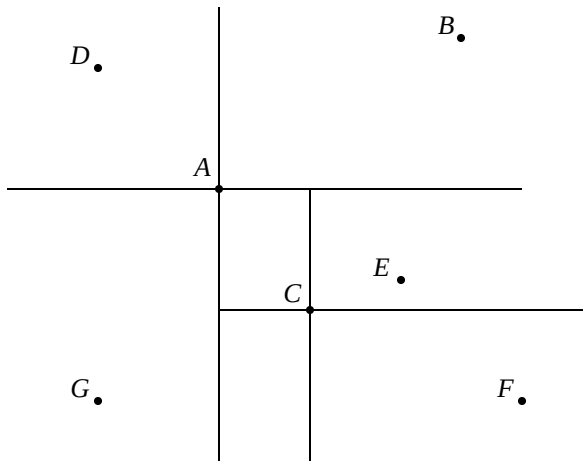


Abbildung 5.84

Fügt man diese Punkte der Reihe nach in den anfangs leeren Quadranten-Baum ein, erhält man Abbildung 5.85.

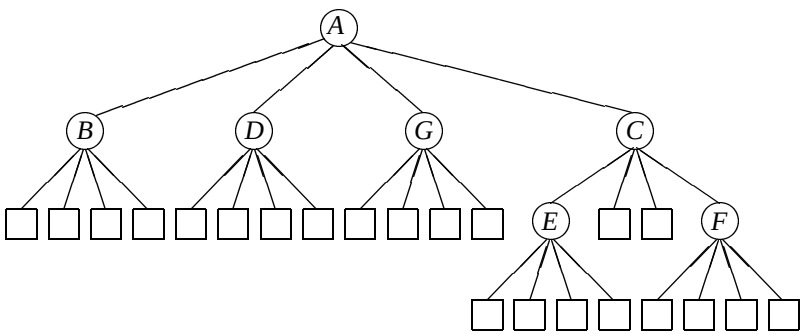


Abbildung 5.85

Es dürfte klar sein, wie man in einem Quadranten-Baum nach Punkten sucht oder weitere Punkte einfügt. (Das Entfernen von Punkten ist offenbar nicht so einfach, es sei denn, der zu entfernende Punkt hat nur Blätter als Söhne.) Zur Bestimmung aller Punkte in einem gegebenen, rechteckigen Bereich beginnt man bei der Wurzel und prüft, ob der dort gespeicherte Punkt im Bereich liegt. Dann setzt man die Bereichssuche bei all den Söhnen fort, deren zugehöriger Quadrant einen nichtleeren Durchschnitt mit dem gegebenen Bereich hat.

2d-Bäume

Wir bauen einen Binärbaum wie einen natürlichen Baum, wobei wir allerdings abwechselnd die *x*- und *y*-Koordinate der Punkte heranziehen, um die Position des jeweils nächsten Punktes im Baum zu bestimmen. Wir erläutern das Verfahren wieder am Beispiel derselben Menge von sieben Punkten in Abbildung 5.86.

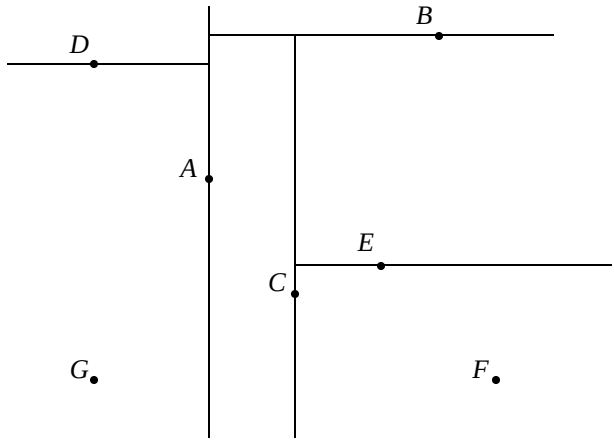


Abbildung 5.86

Beginnt man, zunächst nach *x*, dann nach *y*, dann wieder nach *x* usw. zu unterscheiden, ergibt sich durch iteriertes Einfügen der Punkte $A, \ldots, G$ in den anfangs leeren Baum der 2d-Baum in Abbildung 5.87.

Wieder dürfte unmittelbar klar sein, wie man nach einem Punkt sucht bzw. wie man einen neuen Punkt in einen 2d-Baum einfügt. Das Entfernen von Punkten ist dagegen nicht so einfach. Bereichsanfragen werden offenbar dadurch unterstützt, daß eine Bereichssuche immer dann bei nur einem von zwei Söhnen fortgesetzt werden muß, wenn der Bereich ganz auf einer Seite der durch den Punkt definierten Trennlinie liegt.

Quadranten- und 2d-Bäume ebenso wie zahlreiche andere Strukturen zur mehrdimensionalen Suche sind intensiv studiert worden. Der interessierte Leser möge dazu etwa das Buch [122] konsultieren.

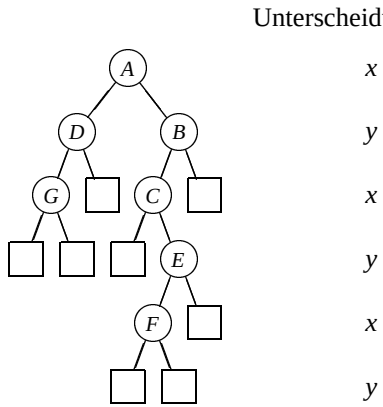


Abbildung 5.87

5.9 Aufgaben

Aufgabe 5.1
Gegeben sei die Folge F von acht Schlüsseln

$$F = 4, 8, 7, 2, 5, 3, 1, 6$$

a) Geben Sie den zu F gehörenden natürlichen Baum an.

b) Welcher Baum entsteht aus dem in a) erzeugten Baum, wenn man den Schlüssel 4 löscht?

c) Geben Sie alle Folgen F' von acht Schlüsseln an, die die Eigenschaft haben, daß der zu F' gehörende natürliche Baum mit dem von F erzeugten übereinstimmt und F' wie folgt beginnt: $F' = 4, 2, 8, 7, \ldots$

Aufgabe 5.2

a) Geben Sie den natürlichen Baum an, der entsteht, wenn man der Reihe nach die Schlüssel $10, 5, 14, 9, 11, 12, 15, 6$ in den anfangs leeren Baum einfügt.

b) Ersetzen Sie in dem bei a) erhaltenen Baum die **nil**-Zeiger durch Verweise auf den symmetrischen Vorgänger (wenn der linke Sohn eines Knotens **nil** ist) bzw. Nachfolger (wenn der rechte Sohn eines Knotens **nil** ist), soweit diese existieren.

c) Welcher Baum entsteht, wenn man Schlüssel 10 entfernt?

Aufgabe 5.3
Die Struktur eines Binärbaumes sei durch folgende Typvereinbarung festgelegt:

```
type Knotenzeiger = ↑knoten;
     knoten = record
                key : integer;
                rechts, links : Knotenzeiger
              end;
```

Ein Baum sei durch einen Zeiger auf die Wurzel und der leere Baum sei durch einen **nil**-Verweis repräsentiert.

Schreiben Sie Funktionen, die die Anzahl der inneren Knoten, die gesamte Pfadlänge (das ist die Summe aller Abstände aller inneren Knoten von der Wurzel, gemessen in der Zahl der Kanten) und die Gesamtanzahl der Blätter berechnet.

Aufgabe 5.4

Binärbäume seien wie in Aufgabe 5.3 vereinbart; jedoch soll jeder Knoten zusätzlich eine Komponente *hoehe* besitzen. Wir nehmen an, daß jeder innere Knoten zwei Söhne besitzt. Beide Zeiger eines externen Knotens haben den Wert **nil**. Jeder Baum bestehe aus mindestens einem (externen) Knoten.

Ergänzen Sie die folgende Definition einer Funktion

```
function tiefstknoten(wurzel : Knotenzeiger ) : Knotenzeiger;
```

in Pascal so, daß für das Argument *wurzel* als Funktionswert ein Zeiger auf einen externen Knoten mit maximaler Tiefe (Endpunkt eines Pfades maximaler Länge) in dem in *wurzel*↑ wurzelnden Binärbaum berechnet wird.

```
function tiefstknoten(wurzel : Knotenzeiger) : Knotenzeiger;
var p : Knotenzeiger;
begin
  markhoehe(wurzel);
  p := wurzel;
  while  ...  do
  ...
  tiefstknoten := p
end
```

Ein Aufruf *markhoehe*(*wurzel*) bewirkt, daß der Komponente *hoehe* jedes Knotens *k* in dem Binärbaum mit Wurzel *wurzel*↑ die Höhe des in *k* wurzelnden Teilbaums als Wert zugewiesen wird.

Aufgabe 5.5

Gegeben sei ein Binärbaum B mit ganzzahligen Schlüsseln. Gegeben sei außerdem ein Schlüssel x. Gesucht ist in B der größte Schlüssel $\leq x$.

a) Geben Sie einen Algorithmus an, der diese Aufgabe in $O(h)$ Schritten löst, wenn h die Höhe von B ist.

b) Setzen Sie die Vereinbarungen von Aufgabe 5.3 voraus und schreiben Sie in Pascal eine vollständige Funktion zu dem in a) entwickelten Algorithmus. Dabei können Sie davon ausgehen, daß für jedes x ein größter im Binärbaum gespeicherter Schlüssel mit Wert $\leq x$ stets vorkommt, da im Binärbaum ein „unechter“ Schlüssel mit Wert $-\infty$ gespeichert ist.

(Hinweis: Verwenden Sie einen Hilfszeiger, der stets am jeweils letzten Knoten stehenbleibt, an dem man beim Hinabsteigen im Baum rechts abgebogen ist.)

Aufgabe 5.6

Ein gefädelter Binärbaum sei durch einen Zeiger auf die Wurzel gegeben. Entwerfen Sie eine Pascal-Prozedur *feinfüge*, die beim Aufruf mit *feinfüge*(*wurzel, k*) den Schlüssel *k* unter Beibehaltung der Fädelung in den Baum einfügt.

Aufgabe 5.7

Gegeben sei die Folge der Schlüssel eines sortierten Binärbaumes in Hauptreihenfolge:

$$20, 15, 5, 18, 17, 16, 25, 22$$

a) Stellen Sie diesen Baum mit Vorgänger- und Nachfolger-Fädelung graphisch dar.

b) Geben die Reihenfolge der Schlüssel in Nebenreihenfolge an.

Aufgabe 5.8

Das Durchlaufen aller Knoten eines Baumes in „umgekehrter Hauptreihenfolge" ist wie folgt definiert:

1. Betrachte die Wurzel.
2. Durchlaufe den rechten Teilbaum der Wurzel in umgekehrter Hauptreihenfolge.
3. Durchlaufe den linken Teilbaum der Wurzel in umgekehrter Hauptreihenfolge.

a) Gegeben sei der Binärbaum aus Abbildung 5.88 mit acht inneren Knoten (Blätter sind durch **nil**-Zeiger repräsentiert). Jeder innere Knoten hat ein unbesetztes Schlüsselfeld. Tragen Sie die Schlüssel $1, 2, \ldots, 8$ so in diesen Baum ein, daß der Schlüssel die Knotennummer in umgekehrter Hauptreihenfolge ist.

b) Das Knotenformat eines Binärbaums sei wie in Aufgabe 5.3 vereinbart.

Ein nichtleerer binärer Baum mit einer festen Anzahl *N* von inneren Knoten sei gegeben durch einen Zeiger auf die Wurzel. Schreiben Sie eine Prozedur

procedure *numeriere* (**var** *wurzel* : *Knotenzeiger*);

die eine „Numerierung" aller inneren Knoten (wie in a) beschrieben) in umgekehrter Hauptreihenfolge vornimmt.

c) Wie kann man (eventuell durch Einführen zusätzlicher Zeiger anstelle von **nil**-Zeigern) die Speicherung von Bäumen analog zur Fädelung für die symmetrische Reihenfolge so ändern, daß man einen Binärbaum in umgekehrter Hauptreihenfolge iterativ durchlaufen kann?

Aufgabe 5.9

Erstellen Sie eine rekursive Pascal-Prozedur *Pfad*(*p* : *Knotenzeiger*; *k* : *integer*), die für einen sortierten Binärbaum mit Zeiger *wurzel* auf die Wurzel beim Aufruf *Pfad*(*wurzel, k*) die Schlüsselwerte auf dem Pfad vom Knoten, der den Suchschlüssel *k* speichert, zur Wurzel in dieser Reihenfolge ausgibt. Es sei bei einem Aufruf *Pfad*(*wurzel, k*) garantiert, daß der Schlüssel *k* im Baum auftritt.

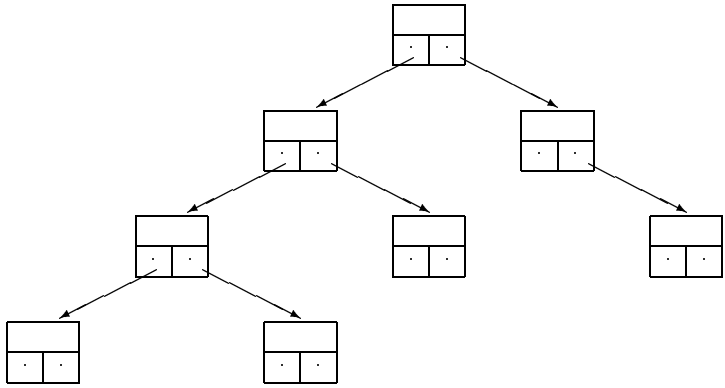


Abbildung 5.88

Aufgabe 5.10

a) Gegeben sei der in Abbildung 5.89 gezeigte Binärbaum mit vier inneren Knoten:

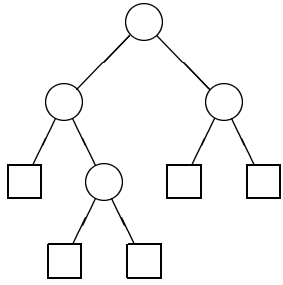


Abbildung 5.89

Geben Sie an, mit welcher Wahrscheinlichkeit dieser Baum durch sukzessives Einfügen der Schlüssel aus der Menge $\{1,2,3,4\}$ in den anfangs leeren natürlichen Baum erzeugt wird, wenn jede Permutation der Schlüssel $1,\ldots,4$ als gleichwahrscheinlich vorausgesetzt wird.

b) Mit welcher Wahrscheinlichkeit kommt der in a) angegebene Baum in der Menge aller strukturell verschiedenen Binärbäume mit vier inneren Knoten vor, wenn jeder sortierte Binärbaum mit Schlüsseln $1, \ldots, 4$ als gleichwahrscheinlich vorausgesetzt wird?

Aufgabe 5.11

Gegeben sei der natürliche Baum aus Abbildung 5.90:

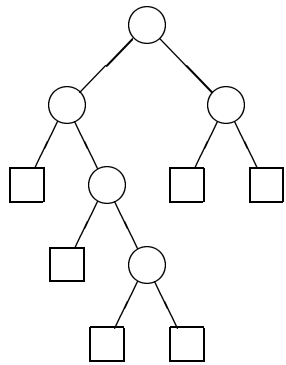


Abbildung 5.90

a) Geben Sie alle Reihenfolgen von Schlüsseln an, die diesen natürlichen Baum erzeugen.

b) Geben Sie alle übrigen strukturell möglichen Bäume mit gleicher Höhe und fünf inneren Knoten an.

Aufgabe 5.12

a) Zeigen Sie, daß der vollständige natürliche Binärbaum mit sieben inneren Knoten von mindestens 49 Permutationen der Zahlen $\{1, \ldots, 7\}$ erzeugt wird bei sukzessivem Einfügen der Schlüssel aus der Menge $\{1, 2, 3, \ldots, 7\}$ in den anfangs leeren Baum.

b) Geben Sie einen natürlichen Baum mit sieben inneren Knoten an, der nur genau einmal erzeugt wird.

Aufgabe 5.13

a) Geben Sie alle natürlichen Bäume mit vier inneren Knoten an, die jeweils von genau einer Permutation der Zahlen $1, \ldots, 4$ erzeugt werden.

b) Geben Sie einen natürlichen Baum mit zehn inneren Knoten an, der von genau zwei Permutationen der Zahlen $1, \ldots, 10$ erzeugt wird, und nennen Sie die Permutationen.

Aufgabe 5.14
Geben Sie den AVL-Baum an, der durch Einfügen der Schlüssel

$$10, 15, 11, 4, 8, 7, 3, 2, 13$$

in den anfangs leeren Baum entsteht.

Aufgabe 5.15

a) Ergänzen Sie die folgende Pascal-Funktionsdefinition so, daß als Funktionswert die Höhe des durch den Zeiger *p* auf die Wurzel gegebenen Baumes geliefert wird.

```
function hoehe (p : Knotenzeiger) : integer;
var l, r : integer;
```

b) Ergänzen Sie die folgende Pascal-Funktionsdefinition so, daß der Wert *true* genau dann geliefert wird, wenn der durch den Zeiger *p* auf die Wurzel gegebene Baum höhenbalanciert ist. Die Funktion *hoehe* darf dabei verwendet werden.

```
function ausgeglichen (p : Knotenzeiger) : boolean;
```

Aufgabe 5.16
Gegeben sei der in Abbildung 5.91 gezeigte 1-2-Bruder-Baum:

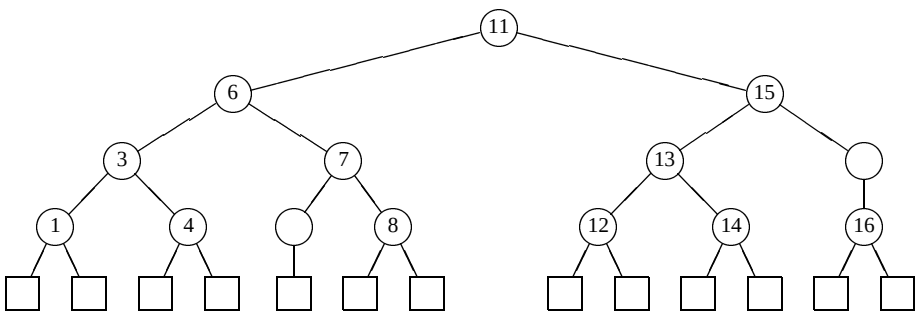


Abbildung 5.91

a) Geben Sie den Baum an, der durch Einfügen des Schlüssels 2 entsteht (mit Zwischenschritten).

b) Geben Sie den Baum an, der durch Entfernen des Schlüssel 11 aus dem ursprünglich gegebenen Baum entsteht.

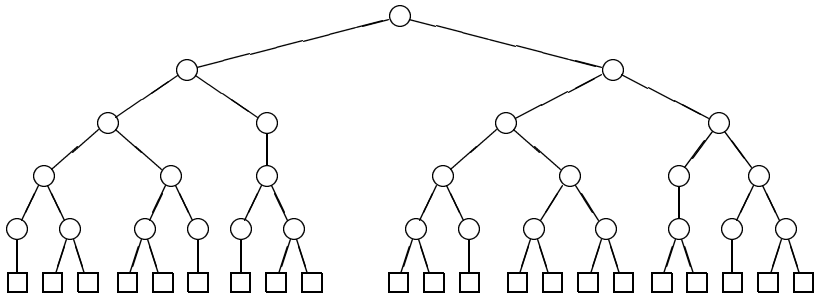


Abbildung 5.92

Aufgabe 5.17

a) Gegeben sei der in Abbildung 5.92 gezeigte Bruder-Baum mit Höhe 5 und 21 Blättern.

Geben Sie eine Position unter den Blättern an, an der eine weitere Einfügung zu einer Umstrukturierung bis zur Wurzel hin und damit zu einem Wachstum der Höhe des Baumes um 1 führt.

b) Welche Eigenschaft muß ein Bruder-Baum haben, so daß eine einzige weitere Einfügung zu einem Wachstum der Höhe führt?

c) Wieviele Blätter muß ein Bruder-Baum mit Höhe h wenigstens haben, damit eine einzige weitere Einfügung an geeigneter Stelle zu einem Bruder-Baum mit Höhe $h+1$ führen kann?

d) Geben Sie für jede Höhe h einen Bruder-Baum mit Höhe h mit minimal möglicher Blattzahl und eine Position unter den Blättern an, so daß eine Einfügung an dieser Stelle zu einem Bruder-Baum mit Höhe $h+1$ führt.

Aufgabe 5.18

a) Geben Sie einen Bruder-Baum der Höhe 4 mit minimal möglicher Blattzahl an.

b) Wieviele Schlüssel muß man mindestens einfügen, damit die Höhe des Baumes um 1 wächst? Wieviele Schlüssel kann man höchstens einfügen, ohne daß der Baum in der Höhe wächst?

c) Geben Sie für den unter a) konstruierten Baum eine längstmögliche Folge von Schlüsseln an, derart, daß der durch ihr sukzessives Einfügen entstehende Baum nicht in der Höhe wächst. (Markieren Sie die Einfügestellen oder geben Sie explizit eine Schlüsselfolge an.)

Aufgabe 5.19

a) Welche beiden Bruder-Bäume entstehen durch iteriertes Einfügen der Schlüssel $1,2,\ldots,7$ und $1,2,\ldots,15$ in den anfangs leeren Baum? Was kann man aufgrund dieser zwei Beispiele für eine aufsteigend sortierte Folge von $N = 2^k - 1$ $(k \geq 1)$ Schlüsseln als Resultat der Einfügung mit Hilfe des Einfügeverfahrens für 1-2-Bruder-Bäume erwarten?

b) Welche Folge von 1-2-Bruder-Bäumen wird erzeugt, wenn man der Reihe nach 7 Schlüssel in *absteigender* Reihenfolge in den anfangs leeren Baum einfügt? Geben Sie die Folge der 7 erzeugten Bäume an.

c) Welche Änderung an dem in Abschnitt 5.2.2 angegebenen Verfahren zum Einfügen von Schlüsseln in 1-2-Bruder-Bäume bewirkt, daß beim iterierten Einfügen von Schlüsseln in absteigender Reihenfolge vollständige Binärbäume erzeugt werden?

Aufgabe 5.20

a) Geben Sie an, welche 1-2-Bruder-Bäume mit fünf Schlüsseln (und sechs Blättern) durch Einfügen von fünf Schlüsseln in den anfangs leeren Baum entstehen können.

b) Mit welcher Wahrscheinlichkeit treten die Bäume aus a) auf, wenn man eine zufällige Folge von fünf Schlüsseln in den anfangs leeren Baum iteriert einfügt? Es wird also angenommen, daß die dem jeweiligen Einfügeschritt vorangehende (erfolglose) Suche nach dem jeweils einzufügenden Schlüssel mit gleicher Wahrscheinlichkeit an jedem der Blätter des Baumes enden kann.

Aufgabe 5.21

Gegeben sei der in Abbildung 5.93 gezeigte 1-2-Bruder-Baum mit drei Schlüsseln (durch Punkte angedeutet) und Höhe 2.

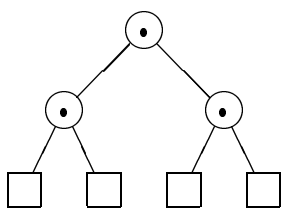


Abbildung 5.93

Geben Sie an, mit welcher Wahrscheinlichkeit daraus ein 1-2-Bruder-Baum mit sieben Schlüsseln und Höhe 4 durch Einfügen weiterer vier Schlüssel entsteht. Dabei wird vorausgesetzt, daß der jeweils nächste einzufügende Schlüssel mit derselben Wahrscheinlichkeit in jedes der Schlüsselintervalle des gegebenen Baumes fällt.

Aufgabe 5.22

Gegeben sei ein zufällig erzeugter 1-2-Bruder-Baum mit N Schlüsseln. Geben Sie die Wahrscheinlichkeit dafür an, daß

a) die Umstrukturierung (mit Hilfe der Prozedur *up*) bereits nach dem ersten Schritt abbricht.

b) die Umstrukturierung wenigstens noch Knoten auf dem zweituntersten Niveau innerer Knoten betrifft.

Aufgabe 5.23

Eine Folge $S = s_1, \ldots, s_N$ von N Schlüsseln ist wie folgt auf Blöcke von je zwei oder drei Schlüsseln aufzuteilen: Man schafft im ersten Schritt den Block s_1, ∞. Dabei ist ∞ ein „Pseudoschlüssel“, der größer als alle in S auftretenden Schlüssel ist. Hat man bereits die Blöcke $B_1, \ldots, B_k$ erzeugt, so ist die in der Reihenfolge der Blöcke und innerhalb der Blöcke von links nach rechts vorkommende Folge von Schlüsseln aufsteigend sortiert. Der nächste Schlüssel s wird jeweils so in diese Folge eingefügt, daß man versucht, ihn in den von links her ersten Block einzufügen, der einen Schlüssel größer als s enthält. Hat dieser Block bereits drei Schlüssel, so zerlegt man ihn in zwei Blöcke mit je zwei Schlüsseln.

Beispiel: $S = 3,2,1,5,4$

$$\boxed{3,\infty} \Longrightarrow_2 \boxed{2,3,\infty} \Longrightarrow_1 \boxed{1,2}\boxed{3,\infty}$$
$$\Longrightarrow_5 \boxed{1,2}\boxed{3,5,\infty} \Longrightarrow_4 \boxed{1,2}\boxed{3,4}\boxed{5,\infty}$$

Berechnen Sie die mittlere Anzahl von Blöcken der Größe 2 und 3 nach N Einfügungen unter der Annahme, daß jede der $N!$ möglichen Anordnungen von N Schlüsseln gleichwahrscheinlich ist.

Aufgabe 5.24

a) Geben Sie die Struktur eines höhenbalancierten Baumes der Höhe 4 an, für den die Wurzelbalance (Verhältnis der Anzahl der Blätter des linken Teilbaums zur Gesamtblätterzahl) möglichst klein ist.

b) Zeigen Sie: Es ist möglich, höhenbalancierte Bäume mit Höhe h anzugeben, für die die Wurzelbalance mit wachsender Höhe h beliebig klein wird.

Aufgabe 5.25

a) Fügen Sie die Punkte $7, 19, 23, 4, 12, 17, 8, 11, 2, 9$ und 13 in einen anfangs leeren B-Baum der Ordnung 3 ein.

b) Entfernen Sie die Punkte 12 und 17.

c) Welchen Aufwand benötigt man zum Entfernen eines Schlüssels im mittleren (schlechtesten) Fall?