Kurzeinführung: AVL-Bäume

Thierry Steiner
Manthos Takidis
Daniel Hottinger

2001-05-16

Inhaltsverzeichnis

Zusammenfassung

Aus Zeitmangel beim Zusammenstellen dieser Arbeit und der daraus folgenden Unvollständigkeit ist dieses Dokument *nicht* zur veröffentlichung bestimmt

1 Bäume

Bäume sind verallgemeinerte Listenstrukturen. Statt nur einen Nachfolger haben die Knoten mehrere Nachfolger, sogenannte Söhne. Dabei bestimmt die Ordnung die maximale Anzahl der Söhne. Bäume der Ordnung $d > 2$ nennt man Vielwegbäume, solche der Ordnung $d = 2$ Binärbäume. Knoten, die keine Söhne haben, werden als Blätter bezeichnet; alle anderen Knoten nennt man innere Knoten.

Wir werden uns in den nächsten Beispielen mit Binärbäumen beschäftigen. Dazu gehören auch elementare Operationen wie Einfügen, Entfernen, ... eines Knotens.

1.1 Suchbäume

Ein binärer Suchbaum ist ein Binärbaum, dessen Knoten `p` jeweils mit einem Schlüsselwert $key(p)$ versehen sind, so dass *für alle inneren Knoten* p gilt:

- ist w ein Knoten im linken Teilbaum von p, dann ist $key(w) < key(p)$
- ist u ein Knoten im rechten Teilbaum von p, dann ist $key(u) > key(p)$

Ein Beispiel ist in Abbildung 1 dargestellt. Den Unterschied zu zu einem gewöhnlichen binären Baum zeigt Abbildung 2.

1.1.1 Operationen auf Suchbäumen

`Search(p,x)` Die Suche eines Schlüsselwerts x erfolgt durch den Funktionsaufruf $Search(p, x)$, wobei p der Wurzelknoten ist[1]. Wir verfahren nach folgender Methode:

- starte in der Wurzel
- sei p der aktuelle Knoten
  - ist $key(p) > x$, dann suche x im linken Teilbaum von x (falls möglich $\Rightarrow p \neq NIL$)
  - ist $key(p) < x$, dann suche x im rechten Teilbaum von x (falls möglich $\Rightarrow p \neq NIL$)
  - ist $key(p) = x$, dann beende die Suche.

Der Algorithmus $Search(x, p)$ ist so formuliert, dass ein Knoten z zurückgegeben wird:

- Für eine erfolgreiche Suche ist $key(z) = x$.
- Für eine erfolglose Suche zeigt z auf NIL.

[1] Für den leeren Baum ist die Suche sofort "erfolglos" zu beenden

Die Rückgabe des Knotens z wird für das Löschen bzw. Einfügen benötigt.

`Insert(VAR p, x)` Um einen Schlüssel in einem Suchbaum einzufügen, suchen wir zunächst nach dem einzufügenden Schlüssel x im gegebenen Baum. Der nichtgefundene Schlüssel x wird dann an der erwartenden Position p unter den Blättern eingefügt. D.h. wir ersetzen das Blatt durch einen inneren Knoten mit dem einzufügenden Schlüssel x als Wert und zwei Blätter als Söhnen. So entsteht wieder ein Suchbaum. Abbildung 3 veranschaulicht dies.

`Delete(p,x)` Beim Löschen eines Schlüsselwerts x, der als Markierung eines Knotens v gefunden wurde, sind drei Fälle zu unterscheiden:

- Ist v ein Blatt, dann wird v einfach gelöscht.
- Wenn v genau einen Sohn w hat, dann kann man wie folgt vorgehen:
  - ist v die Wurzel, dann wird v gelöscht und w zur Wurzel gemacht.
  - ist u der Vater von v, dann kann man v löschen und die Kante (u, v) durch die Kante (u, w) ersetzen.

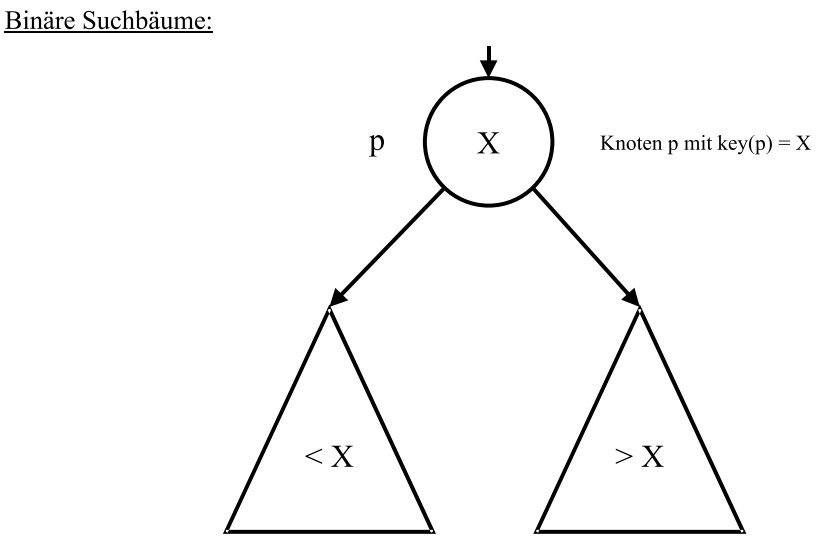

Abbildung 1: Ein binärer Suchbaum

Beispiel

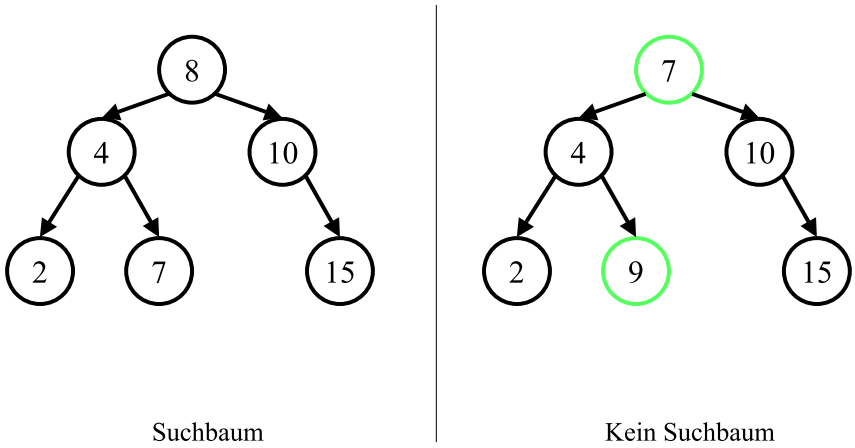

Abbildung 2: Bei einem Suchbaum haben die linken Knoten einen kleineren Wert, als die rechten

- Wir nehmen an, dass v zwei Söhne hat. Wir bestimmen zunächst denjenigen Knoten w im linken Teilbaum von v, der mit dem nächst kleineren Schlüsselwert markiert ist. Dieser ist der grösste Schlüsselwert im linken Teilbaum von v und lässt sich durch die Suche des Schlüsselwertes $x = key(v)$ im linken Teilbaum von v bestimmen. Die Markierung $key(w)$ wird dann zur Markierung von v gemacht und der Knoten w gelöscht. Zu beachten: Der Knoten w hat keinen rechten Sohn, kann also gemäss Fall 1 oder 2 gelöscht werden. Tatsächlich gelöscht wird also stets nur solche Knoten, die höchstens einen Sohn haben.

Im wesentlichen wird für alle drei Operationen der Suchbaum auf einem Pfad von der Wurzel bis zu einem Blatt durchlaufen. Damit ergeben sich folgende Kosten:

- Die Grundoperationen Suchen, Löschen und Einfügen in einen binären Suchbaum T lassen sich in Zeit $\Theta(\text{Höhe}(T)) = O(n)$ durchführen. Dabei ist n die Anzahl der Schlüsselwerte (Knoten) in T.
- $[\log n] \leq \text{Höhe}(T) \leq n - 1$

Einfügen von Suchbäumen:

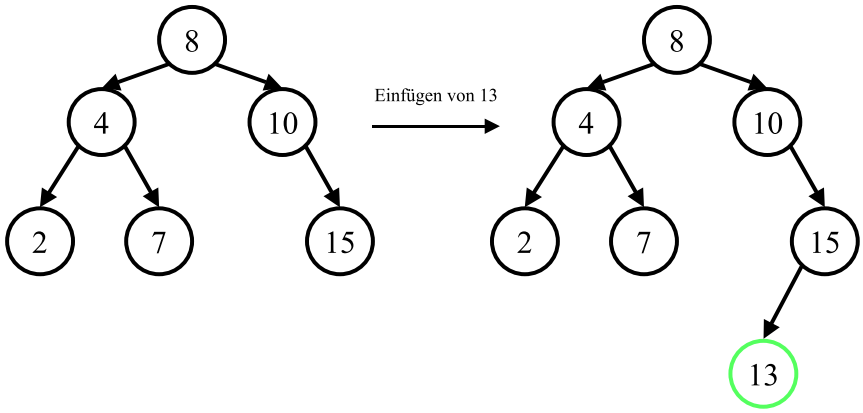

Abbildung 3: Einfügen in einen Suchbaum

1.2 Balancierte Binärbäume

Um Extremfälle, in denen Suchbäume zu einer linearen Liste degenerieren, zu verhindern, wendet man Balancierungskriterien an, die garantieren, dass selbst im schlimmsten Fall die Höhe des Suchbaumes in $O(\log n)$ liegt. Wir betrachten höhenbalancierte Suchbäume, kurz `AVL`-Bäume.

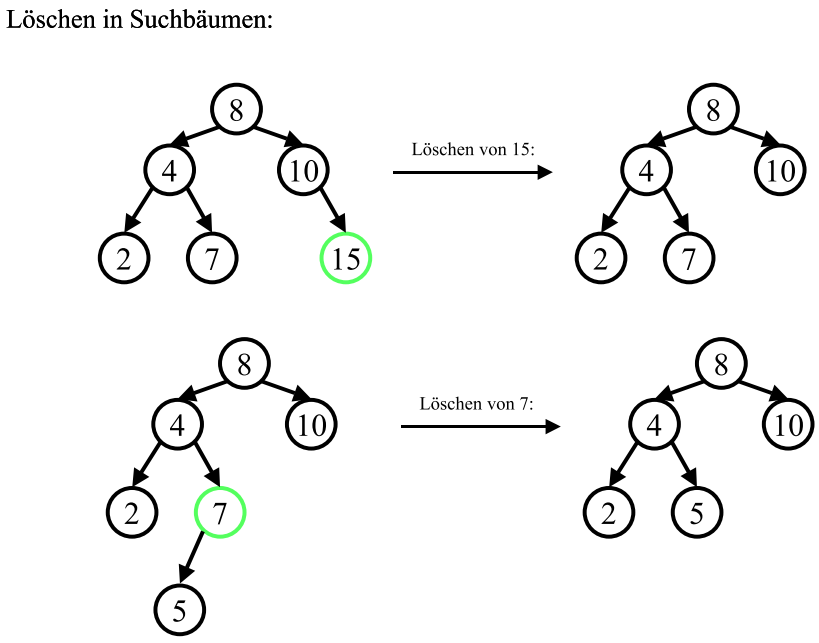

Abbildung 4: Löschen von Knoten

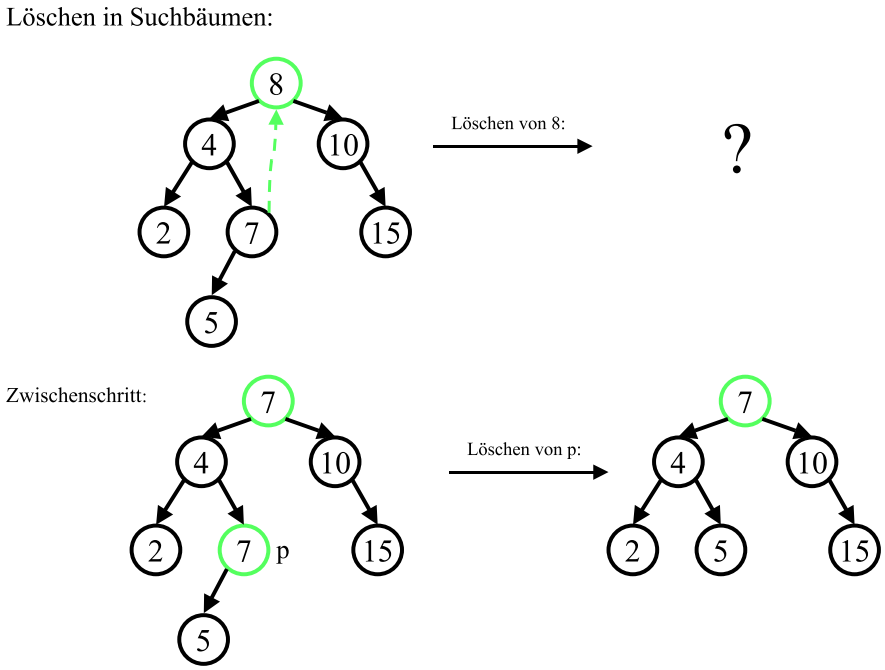

Abbildung 5: Löschen von Knoten(2)

1.2.1 AVL-Bäume

Sei T ein binärer Suchbaum und p ein Knoten in T. Weiter sei T_l der linke und T_r der rechte Teilbaum des Knotens p. Die Balance in Knoten p ist der Wert:

$$\text{bal}(p) = \text{Höhe}(T_l) - \text{Höhe}(T_r) \tag{1}$$

Ein `AVL`-Baum ist ein binärer Suchbaum T mit Knoten p, genau dann, wenn gilt:

$$\forall p : \text{bal}(p) \in \{-1, 0, 1\} \tag{2}$$

`AVL`-Bäume werden wie Suchbäume implementiert, wobei für jeden Knoten p die Balance in Knoten p als zusätzliche Komponente gespeichert wird. Abbildung 6 veranschaulicht dies.

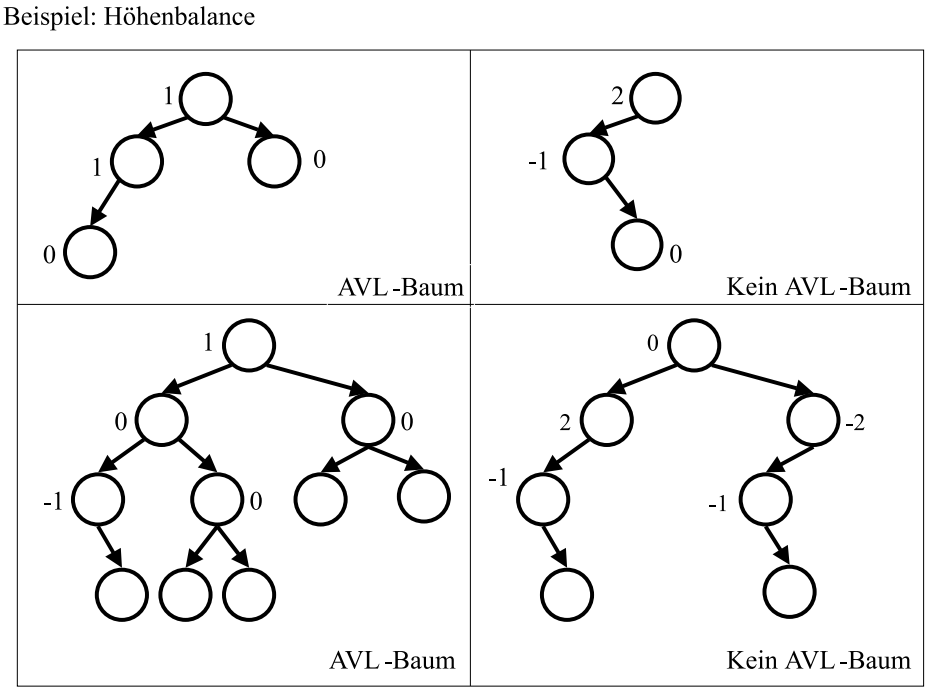

Abbildung 6: Höhenbalance

Für `AVL`-Bäume T mit n Knoten gilt (ohne Beweis):

$$\text{Höhe}(T) = O(\log n) \tag{3}$$

Genauer: $\text{Höhe}(T) \leq 1.44 \log n$

1.2.2 Operationen auf `AVL`-Bäumen

Zunächst können Suchen, Löschen und Einfügen wie in gewöhnlichen Suchbäumen durchgeführt werden. Das Einfügen bzw. Löschen eines Elements in einen

AVL-Baum kann dessen Struktur verändern, sodass die Balancierungskriterien nicht mehr stimmen. Deshalb ist zu prüfen, ob Rebalancierungsmassnahmen notwendig sind oder nicht. Zur Rebalancierung werden

- die einfache Rotation (kurz Rotation genannt)
- die Doppelrotation verwendet.

Rotation: (im Bottom-Up-Verfahren)

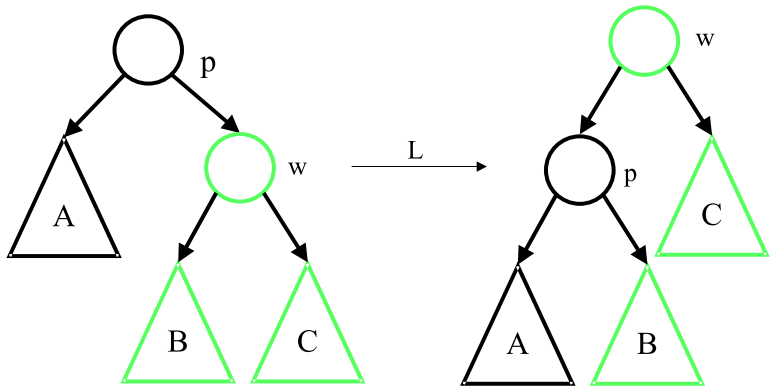

Abbildung 7: Rotationen

Die Rotation und die Doppelrotation können nach links oder nach rechts ausgeführt werden. Die Rotation sowie die Doppelrotation an einem Knoten v kann in konstanter Zeit ausgeführt werden. Wir erläutern hier nur den Fall einer Rotation bzw. Doppelrotation nach links, die nach dem Einfügen im rechten Teilbaum oder Löschen im linken Teilbaum erforderlich sein kann.

Man beachte, dass eine Rotation oder Doppelrotation nur an solchen Knoten v durchgeführt wird, für die die Teilbäume von v der AVL-Bedingungen genügen. Dies kann dadurch sichergestellt werden, in dem die AVL-Bedingungen in Bottom-Up-Manier wiederhergestellt wird. Das heisst, dass wir starten mit dem tiefsten Knoten v, an dem die AVL-Balance gestört ist.

- Ist $bal(v) = -2$ und $bal(right(v)) \in \{-1, 0\}$, dann ist eine einfache Rotation nach links auszuführen.

- Ist $bal(v) = 2$ und $bal(left(v)) \in \{1, 0\}$, dann ist eine einfache Rotation nach rechts auszuführen.
- Ist $bal(v) = -2$ und $bal(right(v)) = 1$, dann ist eine Doppelrotation nach links auszuführen.
- Ist $bal(v) = 2$ und $bal(left(v)) = -1$, dann ist eine Doppelrotation nach rechts auszuführen.

Nach dem eigentlichen Einfügen bzw. Löschen, wie in gewöhnlichen Suchbäumen, muss die Balanceinformation berichtigt werden und eventuelle rebalanciert werden.

Eine Beispielimplementierung könnte wie folgt aussehen:

```
PROCEDURE BalanceNode(node: PAVLNode): PAVLNode;
BEGIN
  IF node^.balance < LESS THEN
    IF node^.left^.balance > EQUAL THEN
      node^.left := RotLeft(node^.left);
    END;
    node := RotRight(node);
  ELSIF node^.balance > MORE THEN
```

Doppelrotation:(im Bottom-Up-Verfahren)

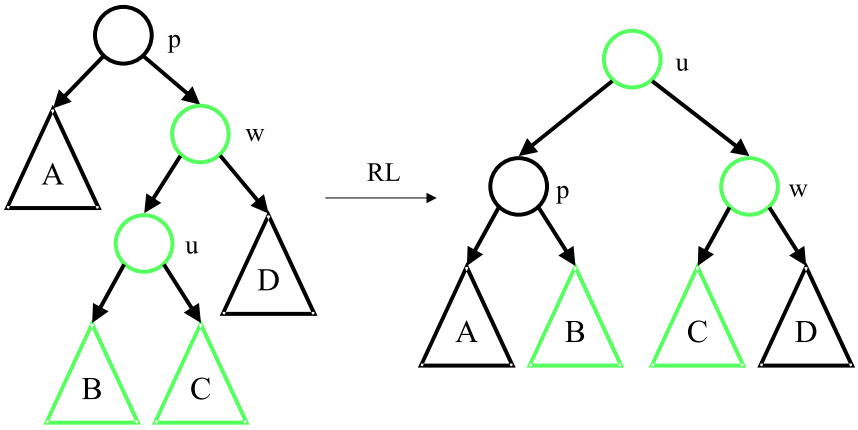

Abbildung 8: Doppelrotationen

```
    IF node^.right^.balance < EQUAL THEN
      node^.right := RotRight(node^.right); (* siehe Beispiel *)
    END;
    node := RotLeft(node);
  END;
  RETURN node;
END BalanceNode;
```

1.3 Implementierung in Oberon

Die folgende Beispielimplementierung hält sich nicht immer an die verwendeten Bezeichnungen in diesem Dokument. Sie ist aber voll funktionsfähig und soll der veranschaulichung dienen.

```
(**************************************
 * Description:
 *   Implementiert einen AVL-Baum
 *   in Oberon.
 * Author:
 *   Daniel Hottinger <hodaniel@iiic.ethz.ch>
 *   Department of Computer Science, ETH Zurich
 *   SS 2001
 * Licence: GNU GPL v2 or later
 * Created: Mon May 14 20:16:37 CEST 2001
 * Last update: none
 * Changes:
 **************************************)

MODULE AVLTree;

IMPORT
  Out;

CONST
  LESS  = -1;
  EQUAL =  0;
  MORE  =  1;

TYPE
  PKey* = POINTER TO TKey;
  TKey* = RECORD
  END;

  PValue* = POINTER TO TValue;
```

```
  TValue* = RECORD
  END;

  TCompareFunc* = PROCEDURE (key1, key2: PKey): LONGINT;
  TPrintProc = PROCEDURE(key: PKey);

  PAVLNode = POINTER TO TAVLNode;
  TAVLNode = RECORD
    balance: LONGINT;
    left, right: PAVLNode;
    key: PKey;
    value: PValue;
  END;

  PAVLTree* = POINTER TO TAVLTree;
  TAVLTree* = RECORD
    root: PAVLNode;
    cmp: TCompareFunc;
  END;

(**************************************
 * Speicherverwaltung
 **************************************)

PROCEDURE NewNode(key: PKey; value: PValue): PAVLNode;
VAR
  node: PAVLNode;
BEGIN
  NEW(node);
  node^.balance := 0;
  node^.left := NIL;
  node^.right := NIL;
  node^.key := key;
  node^.value := value;
  RETURN node;
END NewNode;

(* One day we'll reuse these nodes, *)
(* so oberon can do less wrong      *)
PROCEDURE DestroyNode(VAR node: PAVLNode);
BEGIN
  IF node # NIL THEN
    DestroyNode(node^.left);
    DestroyNode(node^.right);
```

```
  END;
  (* Hope the oberon garbage collector works *)
  node := NIL;
END DestroyNode;

PROCEDURE NewTree*(cmp: TCompareFunc): PAVLTree;
VAR
  tree: PAVLTree;
BEGIN
  NEW(tree);
  tree^.root := NIL;
  tree^.cmp := cmp;
  RETURN tree;
END NewTree;

PROCEDURE DestroyTree*(VAR tree: PAVLTree);
BEGIN
  IF tree # NIL THEN
    DestroyNode(tree^.root);
    tree := NIL;
  END;
END DestroyTree;

(****************************************
 * Rotationen
 ****************************************)

(****************************************
 * n(l,r(rl,rr)) =>
 * r(n(l,rl),rr)
 *
 *   n                r
 *  / `_            _.' \
 * l    r    =>    n     rr
 *     / \         / \
 *    rl  rr      l   rl
 ****************************************)
PROCEDURE RotLeft(node: PAVLNode): PAVLNode;
VAR
  left, right: PAVLNode;
  abal, bbal: LONGINT;
BEGIN
  left := node^.left;
  right := node^.right;
```

```
  node^.right := right^.left;
  right^.left := node;

  abal := node^.balance;
  bbal := right^.balance;

  IF bbal <= EQUAL THEN
    IF abal >= MORE THEN
      right^.balance := bbal - 1;
    ELSE
      right^.balance := abal + bbal - 2;
    END;
    node^.balance := abal - 1;
  ELSE
    IF abal <= bbal THEN
      right^.balance := abal - 2;
    ELSE
      right^.balance := bbal - 1;
    END;
    node^.balance := abal - bbal - 1;
  END;
  RETURN right;
END RotLeft;

(*****************************************
 * n(l(ll,lr),r) =>
 * l(ll,n(lr,r))
 *
 *       n        l
 *    _.' \      / '_
 *   l     r => ll   n
 *  / \             / \
 * ll  lr          lr  r
 *****************************************)
PROCEDURE RotRight(node: PAVLNode): PAVLNode;
VAR
  left, right: PAVLNode;
  abal, bbal: LONGINT;
BEGIN
  left := node^.left;
  right := node^.right;

  node^.left := left^.right;
```

```
  left^.right := node;

  abal := node^.balance;
  bbal := left^.balance;

  IF bbal <= EQUAL THEN
    IF abal < bbal THEN
      left^.balance := bbal + 1;
    ELSE
      left^.balance := abal + 2;
    END;
    node^.balance := abal - bbal + 1;
  ELSE
    IF abal <= LESS THEN
      left^.balance := bbal + 1;
    ELSE
      left^.balance := abal + bbal + 2;
    END;
    node^.balance := abal + 1;
  END;
  RETURN left;
END RotRight;

(****************************************
 * Ausbalancieren
 ****************************************)

(****************************************
 * Beispiel: (Doppelrotation)
 * 2(1,6(5(4),7)) =>
 * 2(1,5(4,6(,7))) =>
 * 5(2(1,4),6(,7))
 *
 *   2             2
 *  / `-._        / `_            5
 * 1      6      1    5          / \
 *       / \  =>     / \    =>  2   6
 *      5   7       4   6       ^    \
 *     /                 \     1 4    7
 *    4                   7
 ****************************************)
PROCEDURE BalanceNode(node: PAVLNode): PAVLNode;
BEGIN
  IF node^.balance < LESS THEN
```

```
    IF node^.left^.balance > EQUAL THEN
      node^.left := RotLeft(node^.left);
    END;
    node := RotRight(node);
  ELSIF node^.balance > MORE THEN
    IF node^.right^.balance < EQUAL THEN
      node^.right := RotRight(node^.right); (* siehe Beispiel *)
    END;
    node := RotLeft(node);
  END;
  RETURN node;
END BalanceNode;

(* Called after a node of node^.left was removed *)
PROCEDURE RestoreLeftBalance(node: PAVLNode; oldbalance: LONGINT): PAVLNode;
BEGIN
  IF node^.left = NIL THEN
    INC(node^.balance);
  ELSIF (node^.left^.balance # oldbalance) & (node^.left^.balance = 0) THEN
    (* left tree shrunk *)
    INC(node^.balance);
  END;
  IF node^.balance > MORE THEN
    RETURN BalanceNode(node);
  ELSE
    RETURN node;
  END;
END RestoreLeftBalance;

(* Called after a node of node^.right was removed *)
PROCEDURE RestoreRightBalance(node: PAVLNode; oldbalance: LONGINT): PAVLNode;
BEGIN
  IF node^.right = NIL THEN
    DEC(node^.balance);
  ELSIF (node^.right^.balance # oldbalance) & (node^.right^.balance = 0) THEN
    (* right tree shrunk *)
    DEC(node^.balance);
  END;
  IF node^.balance > LESS THEN
    RETURN BalanceNode(node);
  ELSE
    RETURN node;
  END;
END RestoreRightBalance;
```

```
PROCEDURE RemoveNodeMostLeft(node: PAVLNode; VAR leftmost: PAVLNode): PAVLNode;
VAR
  oldbalance: LONGINT;
BEGIN
  IF node^.left = NIL THEN
    leftmost := node;
    RETURN node^.right;
  END;

  oldbalance := node^.left^.balance;
  node^.left := RemoveNodeMostLeft(node^.left, leftmost);
  RETURN RestoreLeftBalance(node, oldbalance);
END RemoveNodeMostLeft;

(**************************************
 * grundlegende Operationen
 **************************************)

PROCEDURE InsertNode(node: PAVLNode; cmp: TCompareFunc; key: PKey;
  value: PValue; VAR inserted: BOOLEAN): PAVLNode;
VAR
  relation: LONGINT;
  oldbalance: LONGINT;
BEGIN
  IF node = NIL THEN
    inserted := TRUE;
    RETURN NewNode(key, value);
  END;

  relation := cmp(key, node^.key);
  IF relation = EQUAL THEN
    (* Don't insert dublicate key/value *)
    inserted := FALSE;
    RETURN node;
  ELSIF relation = LESS THEN
    IF node^.left # NIL THEN
      oldbalance := node^.left^.balance;
      node^.left := InsertNode(node^.left, cmp, key, value, inserted);
      IF (oldbalance # node^.left^.balance) & (node^.left^.balance # 0) THEN
(* Tree has grown *)
DEC(node^.balance);
      END;
    ELSE
```

```
      inserted := TRUE;
      node^.left := NewNode(key, value);
      DEC(node^.balance);
    END;
  ELSIF relation = MORE THEN
    IF node^.right # NIL THEN
      oldbalance := node^.right^.balance;
      node^.right := InsertNode(node^.right, cmp, key, value, inserted);
      IF (oldbalance # node^.right^.balance) & (node^.right^.balance # 0) THEN
(* Tree has grown *)
INC(node^.balance);
      END;
    ELSE
      inserted := TRUE;
      node^.right := NewNode(key, value);
      INC(node^.balance);
    END;
  END;

  IF inserted THEN
    IF ABS(node^.balance) > 1 THEN
      node := BalanceNode(node);
    END;
  END;

  RETURN node;

END InsertNode;

PROCEDURE Insert*(tree: PAVLTree; key: PKey; value: PValue);
VAR
  inserted: BOOLEAN;
BEGIN
  IF tree # NIL THEN
    inserted := FALSE;
    tree^.root := InsertNode(tree^.root, tree^.cmp, key, value, inserted);
  END;
END Insert;

(**************************************
 * Beispiel:
 * n(l(:,:),r(rl(rll,rlr),rr)) =>
 * n(l(:,:),r(rl(,rlr),rr)) =>
 * rll(l(:,:),r(rl(,rlr),rr))
```

```
 *    n                    n                  rll
 *   / `--...___          / `-..__           /   `-..__
 *  l           r        l        r         l          r
 *  ^      __.-' \   =>  ^   __.-' \   =>  ^     __.-' \
 * : :    rl      rr    : : rl      rr    : :   rl      rr
 *      _'  \                \                  \
 *     rll   rlr              rlr                rlr
 ***************************************)
PROCEDURE RemoveNode(node: PAVLNode; cmp: TCompareFunc; key: PKey): PAVLNode;
VAR
  relation, oldbalance: LONGINT;
  garbage, newroot: PAVLNode;
BEGIN
  IF node = NIL THEN
    RETURN NIL;
  END;

  relation := cmp(key, node^.key);
  IF relation = EQUAL THEN
    garbage := node;
    IF node^.right = NIL THEN
      node := node^.left;
    ELSE
      oldbalance := node^.right^.balance;
      (* new right node is the leftmost of the right tree *)
      (* Beispiel *)
      node^.right := RemoveNodeMostLeft(node^.right, newroot);
      newroot^.left := node^.left;
      newroot^.right := node^.right;
      newroot^.balance := node^.balance;
      node := RestoreRightBalance(newroot, oldbalance);
    END;
    (* free *only* the removed node *)
    garbage^.right := NIL;
    garbage^.left := NIL;
    DestroyNode(garbage);
  ELSIF relation = LESS THEN
    IF node^.left # NIL THEN
      oldbalance := node^.left^.balance;
      node^.left := RemoveNode(node^.left, cmp, key);
      node := RestoreLeftBalance(node, oldbalance);
    END;
  ELSIF relation = MORE THEN
    IF node^.right # NIL THEN
```

```
      oldbalance := node^.right^.balance;
      node^.right := RemoveNode(node^.right, cmp, key);
      node := RestoreRightBalance(node, oldbalance);
    END;
  END;

  RETURN node;

END RemoveNode;

PROCEDURE Remove*(tree: PAVLTree; key: PKey);
BEGIN
  IF tree # NIL THEN
    tree^.root := RemoveNode(tree^.root, tree^.cmp, key);
  END;
END Remove;

(**************************************
 * Debug
 **************************************)
PROCEDURE Traverse(node: PAVLNode; print: TPrintProc);
BEGIN
  IF node # NIL THEN
    print(node^.key);
    IF (node^.left # NIL) OR (node^.right # NIL) THEN
      Out.String("(");
      Traverse(node^.left, print);
      IF node^.right # NIL  THEN
Out.String(",");
      END;
      Traverse(node^.right, print);
      Out.String(")");
    END;
  END;
END Traverse;

PROCEDURE Dump*(tree: PAVLTree; print: TPrintProc);
BEGIN
  Traverse(tree^.root, print);
END Dump;

END AVLTree.
```