

AVL-Baum-Implementierung

```
/*
 * File: AVLTreeTest.java
 */

/* Knoten fuer ALV Baum */
class AVLNode {
    int content;          // Inhalt, hier integer
    byte balance;        // fuer Werte -2, -1, 0, 1, +2
    AVLNode left;        // linker Nachfolger
    AVLNode right;       // rechter Nachfolger
    AVLNode (int c){     // Konstruktor fuer neuen Knoten
        content = c;     // uebergabener Inhalt
        balance = 0;     // Balance ausgeglichen
        left = right = null; // erst mal keine Nachfolger
    }
} //class AVLNode

/* ALV Baum
root zeigt auf Wurzel des Baumes
grown merkt sich, ob die letzte Einfuegeoperation den Baum
wachsen liess
shrunk merkt sich, ob die letzte Loeschoperation den Baum
schrumpfen liess
found gibt an, ob die letzte Einfuege- oder Loeschoperation
einen Knoten antraf, der das Element schon enthielt
HINWEIS: statt der Variablen grown und shrunk koennte man
auch leicht mit nur einer Variablen (changed)
auskommen, die angibt, ob die letzte Einfuege- oder
Loeschoperation die Hoehe des Baumes geaendert hat.
*/
class AVLTree {
    AVLNode root;       // Die Wurzel des Baumes
```

```

boolean grown, shrunk, found; // Hilfsvariablen fuer die Methoden
AVLTree () { // Konstruktor fuer leeren Baum
    root = null; // Wurzel erst mal leer
    grown = shrunk = found = false; // nicht gewachsen, geschrumpft
} // oder gefunden

boolean search (int c) { // Suche nach c im Baum
    AVLNode n = root; // Start bei Wurzel
    while (n != null) { // falls null: fertig
        if (c == n.content) return true; // falls hier: gefunden
        if (c < n.content) n = n.left; // sonst: suche in
        else n = n.right; // Teilbaeumen
    }
    return false; // nicht gefunden
}

void balanceError (AVLNode n) { System.out.println (
    "Fehler im Balance-Wert: "+ n.balance+" Inhalt: "+n.content );
}

boolean insert (int c) { // Fuege c im Baum ein
    found = false; // default: nicht gefunden
    grown = true; // default: unterster TB wird wachsen
    root = insert (root, c); // rekursives Einfuegen
    return !found; // erfolgreich, falls nicht gefunden
}

AVLNode insert (AVLNode n, int c){ // Fuege c in Teilbaum ein
    if (n == null) return new AVLNode (c); // neuer innerer Knoten
    if (c == n.content) { found = true; // gefunden (also nicht einfuegen)
        grown = false; // und nicht gewachsen
    } else { // in Teilbaeumen versuchen
        if (c < n.content) { // betrachte linken Teilbaum
            n.left = insert (n.left, c); // rekursives Einfuegen
            if (grown) n.balance--; // Balance neigt sich nach links
        } else { // betrachte rechten Teilbaum
            n.right = insert (n.right, c); // rekursives Einfuegen
            if (grown) n.balance++; // Balance neigt sich nach rechts
        }
    }
}

```

```

    switch (n.balance) {
        case -2: if (n.left.balance == +1) rotateLeft (n.left); // D-Rot
                rotateRight (n);
                grown = false; break; // nicht mehr gewachsen!
        case -1: break;
        case 0:  grown = false; break; // nicht gewachsen!
        case +1: break;
        case +2: if (n.right.balance == -1) rotateRight (n.right); // D-Rot
                rotateLeft (n);
                grown = false; break; // nicht mehr gewachsen!
        default: balanceError (n); break;
    } // switch (n.balance)
}
return n;
} //AVLNode insert (AVLNode n, int c)
void rotateRight (AVLNode n) { // einfache Rotation nach rechts
    AVLNode m = n.left;
    int cc = n.content;
    n.content = m.content;
    m.content = cc;
    n.left = m.left;
    m.left = m.right;
    m.right = n.right;
    n.right = m;
    int bm = 1 + Math.max (-m.balance, 0) + n.balance;
    int bn = 1 + m.balance + Math.max (0, bm);
    n.balance = (byte)bn;
    m.balance = (byte)bm;
} //void rotateRight (AVLNode n)
void rotateLeft (AVLNode n) { // einfache Rotation nach links
    AVLNode m = n.right;
    int cc = n.content;
    n.content = m.content;
    m.content = cc;
    n.right = m.right;

```

```

    m.right = m.left;
    m.left = n.left;
    n.left = m;
    int bm = -(1 + Math.max (+m.balance, 0) - n.balance);
    int bn = -(1 - m.balance + Math.max (0, -bm));
    n.balance = (byte)bn;
    m.balance = (byte)bm;
} //void rotateLeft (AVLNode n)
int height (){ // Hoehe
    int h = 0; // zunaechst: 0
    AVLNode n = root; // beginne bei Wurzel
    while (n != null) { h++; // zaehle diesen Knoten
        if (n.balance > 0) { // betrachte nur kleineren Teilbaum
            h += n.balance;
            n = n.left;
        } else {
            h -= n.balance;
            n = n.right;
        }
    }
    return h;
}
int I (){ // interne Pfadlaenge
    return I (root, 0);
}
int I (AVLNode n, int h){
    if (n == null) return 0;
    else { h++; return h + I (n.left, h) + I (n.right, h);
    }
}
boolean delete (int c){ // Loesche c im Baum
    found = shrunk = true; // vermute: wird gefunden und schrumpfen
    root = delete (root, c); // der Teilbaum kann sich aendern!
    return found; // erfolgreich, falls gefunden
}

```

```

AVLNode delete (AVLNode n, int c){
    if (n == null) { found = shrunk = false;
                    return n;
    }
    if (c == n.content) {
        if (n.left == null) return n.right; // rechter Nachfolger
        if (n.right == null) return n.left; // linker Nachfolger
        n.content = c = minValue (n.right); // loesche symm. Nachfolger
    }
    if (c < n.content) { // betrachte linken Teilbaum
        n.left = delete (n.left, c); // versuche links zu loeschen
        if (shrunk) n.balance++; // Balance neigt sich nach rechts
    } else { // betrachte rechten Teilbaum
        n.right = delete (n.right, c); // versuche rechts zu loeschen
        if (shrunk) n.balance--; // Balance neigt sich nach links
    }
    switch (n.balance) {
        case -2: switch (n.left.balance) {
                    case +1: rotateLeft (n.left); break; // D-Rot
                    case 0: shrunk = false; break;
                    case -1: break;
                    default: balanceError (n.left); break;
                }
                rotateRight (n); break;
        case -1: shrunk = false; break;
        case 0: break;
        case +1: shrunk = false; break;
        case +2: switch (n.right.balance) {
                    case -1: rotateRight (n.right); break; // D-Rot
                    case 0: shrunk = false; break;
                    case +1: break;
                    default: balanceError (n.right); break;
                }
                rotateLeft (n); break;
        default: balanceError (n); break;
    }
}

```

```

    } // switch (n.balance)
    return n;
} //AVLNode delete (AVLNode n, int c)
// bestimmt kleinsten Wert im Teilbaum von n
int minValue (AVLNode n) {
    while (n.left != null) n = n.left;
    return n.content;
}
// Hauptreihenfolge; WLR
void preOrder (){
    preOrder (root);
    System.out.println ();
}
void preOrder (AVLNode n){
    if (n == null) return;
    System.out.print (n.content+"("+n.balance+" " );
    preOrder (n.left);
    preOrder (n.right);
}
// Nebenreihenfolge; LRW
void postOrder (){
    postOrder (root);
    System.out.println ();
}
void postOrder (AVLNode n){
    if (n == null) return;
    postOrder (n.left);
    postOrder (n.right);
    System.out.print (n.content+" ");
}
// Symmetrische Reihenfolge; LWR; sortiert;
void inOrder (){
    inOrder (root);
    System.out.println ();
}

```

```

void inOrder (AVLNode n){
    if (n == null) return;
    inOrder (n.left);
    System.out.print (n.content+" ");
    inOrder (n.right);
}
void print (){
    print (root, 0);
    System.out.println ();
}
void print (AVLNode n, int indent){
    if (n == null) return;
    print (n.right, indent+1);
    for (int i=0; i < indent; i++) System.out.print ("  ");
    System.out.println (n.content+" (" +n.balance+""));
    print (n.left, indent+1);
}
} //class AVLTree

public class AVLTreeTest {
    static AVLTree T;
    public static void main(String args[]){
        System.out.println ("AVLTreeTest");
        System.out.println ("Leerer Baum T:");
        T = new AVLTree ();
        T.print ();
        System.out.println ("Hoehe: "+T.height());
        System.out.println ("Interne Padlaenge: "+T.I ());
        System.out.print ("inOrder:  "); T.inOrder ();

        System.out.println ("Baum mit Schluesseln 0 .. 10000:");
        for (int i = 0; i <= 10000; i++) T.insert (i);
        System.out.println ("Hoehe: "+T.height());
        System.out.println ("Interne Padlaenge: "+T.I ());
    }
}

```

```

    System.out.println ("Kann 11000 geloescht werden? "+T.delete (11000));
    for (int i = 11000; i > 64; i--) T.delete (i);
    System.out.println ("Kann 64 geloescht werden? "+T.delete (64));
    System.out.println ("Baum nach Loeschungen der Schluessel 11000 .. 64:");
    T.print ();
    System.out.println ("Hoehe: "+T.height());
    System.out.println ("Interne Padlaenge: "+T.I ());
    System.out.print ("inOrder:  "); T.inOrder ();
}
}

```

```

// Vor dem Uebersetzen: setup java
// Uebersetzen:          javac AVLTreeTest.java
// Laufenlassen:       java  AVLTreeTest
//
// Und das kommt dann heraus:
// *****
//
// AVLTreeTest
// Leerer Baum T:
//
// Hoehe: 0
// Interne Padlaenge: 0
// inOrder:
// Baum mit Schluesseln 0 .. 10000:
// Hoehe: 14
// Interne Padlaenge: 123645
// Kann 11000 geloescht werden? false
// Kann 64 geloescht werden? true
// Baum nach Loeschungen der Schluessel 11000 .. 64:
//           63 (-1)
//           62 (0)
//           61 (1)
//           60 (0)
//           59 (1)

```

// 58 (0)
// 57 (0)
// 56 (0)
// 55 (1)
// 54 (0)
// 53 (0)
// 52 (0)
// 51 (0)
// 50 (0)
// 49 (0)
// 48 (0)
// 47 (1)
// 46 (0)
// 45 (0)
// 44 (0)
// 43 (0)
// 42 (0)
// 41 (0)
// 40 (0)
// 39 (0)
// 38 (0)
// 37 (0)
// 36 (0)
// 35 (0)
// 34 (0)
// 33 (0)
// 32 (0)
// 31 (1)
// 30 (0)
// 29 (0)
// 28 (0)
// 27 (0)
// 26 (0)
// 25 (0)
// 24 (0)

```

//      23 (0)
//          22 (0)
//      21 (0)
//          20 (0)
//      19 (0)
//          18 (0)
//      17 (0)
//          16 (0)
//  15 (0)
//          14 (0)
//      13 (0)
//          12 (0)
//      11 (0)
//          10 (0)
//      9 (0)
//      8 (0)
//  7 (0)
//          6 (0)
//      5 (0)
//          4 (0)
//      3 (0)
//          2 (0)
//      1 (0)
//      0 (0)
//
// Hoehe: 7
// Interne Padlaenge: 328
// inOrder:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
//          26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
//          49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

```