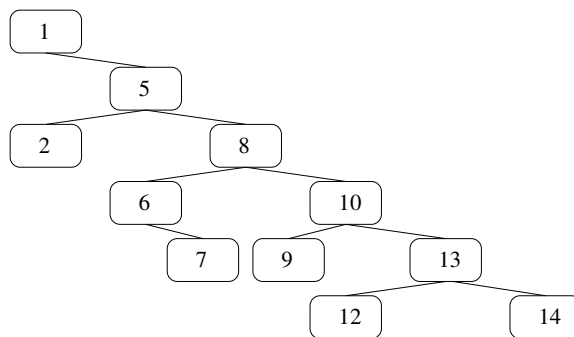


Algorithmen und Datenstrukturen 1

Serie 5

1. a) Hier der binäre Suchbaum:

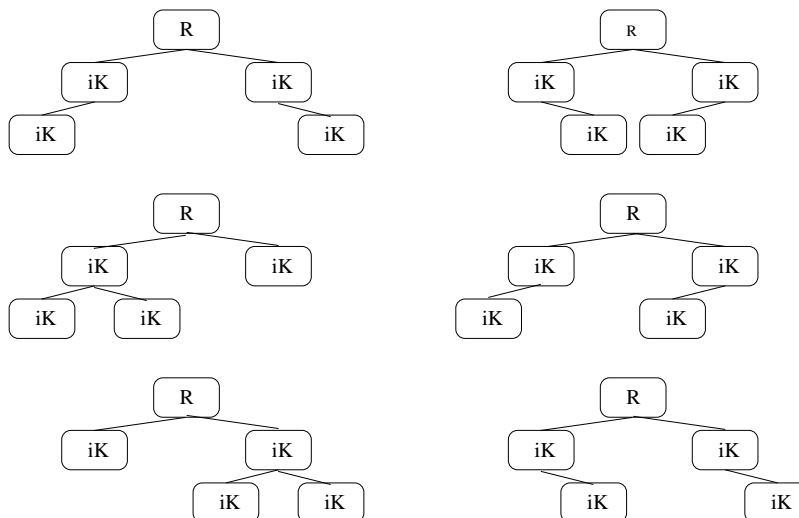


b) Die durchschnittliche Anzahl der besuchten Knoten beträgt 4. Dies ergibt sich aus der Summe aller besuchten Knoten (44) und der Anzahl der Knoten (11).

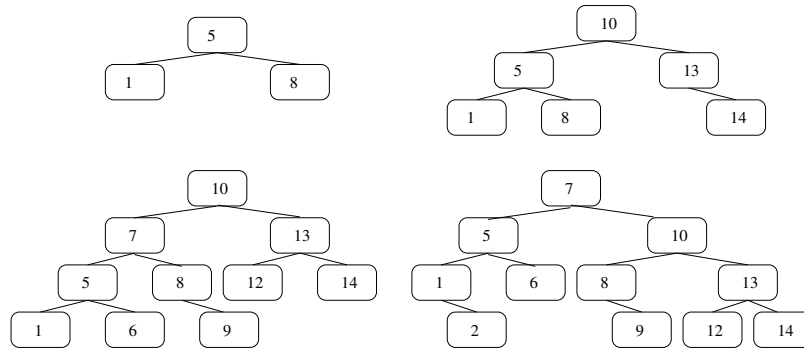
1	5	8	10	13	14	12	6	7	9	2
1	2	3	4	5	6	6	4	5	5	3

c) Da der linke Teilbaum komplett leer ist gilt $k = h$ und somit $k = 5$.

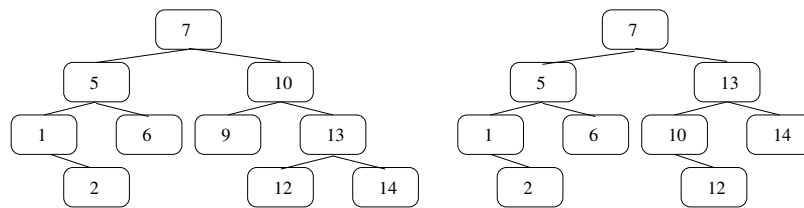
d) Hier alle möglichen Variationen mit vier inneren Knoten und einer Höhe von maximal drei:



2. a) Hier die einzelnen AVL-Bäume nach dem Einfügen ...



b) ... und hier nach dem Löschen:



3. a) Mindestanzahl von Blättern in einem AVL Baum:

Höhe	mind. Blattanzahl
6	13
7	21
8	34

b) Anzahl der Blätter = $fib(h + 1)$; wobei es sich um die Fibonacci-Zahl von x handelt.

4. Hier der Quellcode. Der Einfachheit halber habe ich BinaryDic nicht beerbt sondern die Methoden übernommen:

```
/**
 * Klasse die einen AVL Baum implementiert
 * geschrieben im Rahmen von ADS1, Serie 5, Aufgabe 4
 *
 * @author Arne Brutschy
 */
public class AVLBaum {

    private static int RIGHT = -1;
    private static int LEFT = 1;

    // inline Klasse die einen einzelnen Knoten im Baum repräsentiert
    private class IntTreeNode {
        IntTreeNode parent;
        IntTreeNode left;
        IntTreeNode right;
        Integer value;
        int balance;

        public IntTreeNode(IntTreeNode parent, Integer value) {
            this.parent = parent;
            this.value = value;
            balance = 0;
        }

        // liefert den Sohn einer bestimmten Seite zurueck
        public IntTreeNode getChild(int side) {
            if (side == LEFT)
                return left;
            else if (side == RIGHT)
                return right;
            else
                return this;
        }

        // auf welcher Seite meines Vaterknotens bin ich?
        public int getSide() {
            return (parent.left == this) ? LEFT : RIGHT;
        }
    }

    private IntTreeNode root;
    private int elements;

    /**
```

```
* Erzeugt einen bin"aren Suchbaum mit leerer Wurzel
*/
public AVLBaum() {
    super();
}

/**
 * Erzeugt einen bin"aren Baum mit Inhalt
 * Die Elemente werden als integer Array uebergeben und
 * werden in Reihenfolge eingefuegt
 * @param intArray Elemente des Baumes
 */
public AVLBaum(int[] intArray) {
    super();

    for (int i = 0; i < intArray.length; i++)
        insert(new Integer(intArray[i]));
}

/**
 * Fuegt ein Integer-Objekt in den Baum ein
 * @param x Integer Objekt das eingefuegt werden soll
 */
public void insert(Integer x) {
    // gibts es schon eine Wurzel?
    if (root != null) {
        int res = 0;
        IntTreeNode last = null;
        IntTreeNode current = root;

        // wir suchen bis wir an einem Blatt sind
        while (current != null) {
            last = current;
            // vergleiche mit Wert in momentanen Knoten
            res = x.compareTo(current.value);
            // kleiner -> linker Sohn
            if (res < 0)
                current = current.left;
            // groesser -> rechter Sohn
            else if (res > 0)
                current = current.right;
            // ex. schon -> raus
            else
                return;
        }

        // Wert an den richtigen Sohn anfüegen
        if (res < 0) {
            last.left = new IntTreeNode(last, x);
        }
    }
}
```

```
        rebalTree(last, LEFT);
    } else {
        last.right = new IntTreeNode(last, x);
        rebalTree(last, RIGHT);
    }

    // Wurzel existiert nicht, wird neu angelegt
} else
    root = new IntTreeNode(null, x);

elements++;
}

/**
 * F"uhrt die Rebalancierung des Baumes durch
 * @param node  der naechste Vater am eingefuegten Knoten
 * @param side  auf welcher Seite der Knoten eingefuegt wurde
 */
public void rebalTree(IntTreeNode node, int side)
{
    // bis zur Wurzel ausbalancieren
    for(; node != null; node = node.parent)
    {
        // wie sieht's aus mit der balance
        if(node.balance != side)
            node.balance = node.balance + side;
        else
            node = rotate(node);

        // sind in balance -> raus
        if(node.balance == 0 )
            break;

        side = node.getSide();
    }
}

/**
 * F"uhrt eine Rotation durch
 * bzw bestimmt welche Art von Rotation benoetigt ist
 * @param node Knoten um den rotiert wird
 * @return neuer root
 */
public IntTreeNode rotate(IntTreeNode node)
{
    int side = node.balance;
    IntTreeNode child = node.getChild(side);

    // brauchen wir eine doppelrotation?
```

```

    if(child.balance == -side)
    {
        IntTreeNode grand = child.getChild(-side);
        // ja, nach welcher Seite?
        if (grand.balance == -side) {
            grand.balance = 0;
            child.balance = side;
            node.balance = 0;
        } else if(grand.balance == side) {
            grand.balance = 0;
            child.balance = 0;
            node.balance = -side;
        }
        // all fine
    } else {
        node.balance = 0;
        child.balance = 0;
    }
    rotate(child, side);
    // einfache Rotation
} else {
    if(child.balance == side) {
        node.balance = 0;
        child.balance = 0;
    } else if(child.balance == 0) {
        node.balance = side;
        child.balance = -side;
    }
}
// rotiere
node = rotate(node, -side);

return node;
}

/**
 * F"uhrt eine Rotation nach einer bestimmten Seite aus
 * @param node
 * @param side
 * @return
 */
public IntTreeNode rotate(IntTreeNode node, int side)
{
    IntTreeNode parent = node.parent;           // kann null sein
    IntTreeNode child = node.getChild(-side);   // nicht null
    IntTreeNode grand = child.getChild(side);   // kann null sein

    // Zeiger umaendern
    link(node, -side, grand);
    link(parent, node.getSide(), child);
}

```

```
        link(child, side, node);
        if(node == root)
            root = child;
        return child;
    }

    /**
     * Setzt die Zeiger bei einer Rotation neu
     * @param parent
     * @param side
     * @param child
     */
    public void link(IntTreeNode parent, int side, IntTreeNode child)
    {
        if(child != null)
            child.parent = parent;
        if(parent != null)
            if (side == LEFT)
                parent.left = child;
            else
                parent.right = child;
        else
            root = child;
    }

    /**
     * Sucht ein Integer Objekt im Baum
     * @param x zu suchendes Integer Objekt
     * @return gesuchtes Objekt wenn gefunden, null wenn nicht gefunden
     */
    public Integer search(Integer x) {
        int res = 0;
        IntTreeNode current = root;
        StringBuffer out = new StringBuffer();

        // wir suchen bis wir den Wert gefunden haben oder an einem Blatt sind
        while (current != null) {
            out.append(fixedLengthString(current.value.intValue()));
            // vergleiche mit Wert in momentanen Knoten
            res = x.compareTo(current.value);
            // kleiner -> linker Sohn
            if (res < 0)
                current = current.left;
            // groesser -> rechter Sohn
            else if (res > 0)
                current = current.right;
            // gefunden -> raus
            else {
                System.out.println(out);
            }
        }
    }
}
```

```
        return x;
    }
}

System.out.println("Nicht gefunden!");
return null;
}

/**
 * Fuehrt eine Breitensuche aus und gibt die erhaltenden Knoten mit ihren
 * linken und rechten S"ohnen als String zuruck
 * @return Auflistung Knoten Breitensuche
 */
public String toString() {
    int rpos = 0;          // leseposition im Stack
    int wpos = 0;          // schreibposition im Stack

    // der Stack in dem die zu bearbeitenden Knoten gespeichert werden
    IntTreeNode stack[] = new IntTreeNode[elements];
    StringBuffer out = new StringBuffer("Knoten Lsohn Rsohn\n");

    // wir starten mit der Wurzel
    stack[rpos] = root;

    while (rpos < elements) {
        // momentanen Knoten ausgeben
        out.append(fixedLengthString(stack[rpos].value.intValue()));

        // linker Sohn in den Stack (wenn ex) und Ausgabe erzeugen
        if (stack[rpos].left != null) {
            stack[++wpos] = stack[rpos].left;
            out.append(fixedLengthString(stack[rpos].left.value.intValue()));
        } // linker Sohn ex. nicht, fuege Spaces ein
        } else
            out.append("      ");
        // rechter Sohn in den Stack und ausgabe
        if (stack[rpos].right != null) {
            stack[++wpos] = stack[rpos].right;
            out.append(fixedLengthString(stack[rpos].right.value.intValue()));
        }

        // naechsten Knoten lesen
        rpos++;
        out.append("\n");
    }

    return out.toString();
}
}
```

```
// kleine Hilfsfunktion die ein int auf einen String
// gleichbleibender Laenge formatiert
private static String fixedLengthString(int i) {
    String out = "    "+i;
    return out.substring(out.length() - 6);
}

// Hauptfunktion
public static void main(String argv[]) {
    int vars[] = {74, 27, 60, 26, 12, 2, 50, 25, 75, 80, 81, 79, 15};
    AVLBaum btree = new AVLBaum(vars);

    System.out.println("Breitensuche:");
    System.out.println(btree.toString());

    System.out.println("Suche nach 15:");
    btree.search(new Integer(15));

    System.out.println("Suche nach 27:");
    btree.search(new Integer(27));

    System.out.println("Suche nach 75:");
    btree.search(new Integer(75));

    System.out.println("Suche nach 100:");
    btree.search(new Integer(100));
}
}
```