http://cslibrary.stanford.edu/110/BinaryTrees.html

1. 二叉查找树的查询某个节点

```
/**
 * [public]Returns true if the given target is in the binary tree.
 * Uses a recursive helper.
 ****************************************
 * [private]Recursive lookup  -- given a node, recur
 * down searching for the given data.
 */
public boolean lookup(int data) {
    return (lookup(root, data));
}
private boolean lookup(Node node, int data) {
    if (node == null) {
        return (false);
    }

    if (data == node.data) {
        return (true);
    } else if (data < node.data) {
        return (lookup(node.left, data));
    } else {
        return (lookup(node.right, data));
    }
}
```

- /** 深度优先遍历算法**/

```
private void traversal(Node node) {
    if (node == null) return;

    //1.前序遍历:root->left->right
    System.out.print(node.data + "  ");
    traversal(node.left);
    traversal(node.right);

    //2.中序遍历:left->root->right
    //traversal(node.left);
```

```
        //System.out.print(node.data + "  ");
        //traversal(node.right);

        //3.后序遍历:left->right->root
        //traversal(node.left);
        //traversal(node.right);
        //System.out.print(node.data + "  ");
    }
```

- 二叉查找树的插入

```
    /**
     *[public]Inserts the given data into the binary tree.
     * Uses a recursive helper.
     ************************************************
     * [private]Recursive insert -- given a node pointer, recur down
    and
     * insert the given data into the tree. Returns the new
     * node pointer (the standard way to communicate
     * a changed pointer back to the caller).
     */
    public void insert(int data) {
        root = insert(root, data);
    }
    private Node insert(Node node, int data) {
        if (node == null) {
            node = new Node(data);
        } else {
            if (data <= node.data) {
                node.left = insert(node.left, data);
            } else {
                node.right = insert(node.right, data);
            }
        }
        return (node); // in any case, return the new pointer to the
    caller
    }
```

- 二叉查找树的节点个数

```
/**
 Returns the number of nodes in the tree.
 Uses a recursive helper that recurs
 down the tree and counts the nodes.
 */
public int size() {
    return(size(root));
}
private int size(Node node) {
    if (node == null) return(0);
    else {
        return(size(node.left) + 1 + size(node.right));
    }
}
```

- 二叉查找树的最大深度

```
/**
 Returns the max root-to-leaf depth of the tree.
 Uses a recursive helper that recurs down to find
 the max depth.
 */
public int maxDepth() {
    return(maxDepth(root));
}
private int maxDepth(Node node) {
    if (node==null) {
        return(0);
    }
    else {
        int lDepth = maxDepth(node.left);
        int rDepth = maxDepth(node.right);

        // use the larger + 1
        return(Math.max(lDepth, rDepth) + 1);
    }
}
```

- 树的最小节点值

```
/** 树的最小节点值
 * Returns the min value in a non-empty binary search tree.
 * Uses a helper method that iterates to the left to find
 * the min value.
 * ***********************************************************
 * Finds the min value in a non-empty binary search tree.
 */
public int minValue() {
    return( minValue(root) );
}
private int minValue(Node node) {
    Node current = node;
    while (current.left != null) {
        current = current.left;
    }

    return(current.data);
}
```

- **遍历二叉树,求解是否有一条路径的值的总和等于sum**

```
/**
 * 遍历二叉树,求解是否有一条路径的值的总和等于sum
 *          5
 *        /  \
 *      4     8
 *     /     / \
 *    11   13  4
 *   /  \       \
 * 7    2        1
 * Root-to-leaf paths:
 * path 1): 5 4 11 7      sum = 27;
 * path 2): 5 4 11 2      sum = 22;
 * path 3): 5 8 13        sum = 21;
 * path 4): 5 8 4 1      sum = 18;
 *
*************************************************************
*
```

```
 * Strategy: subtract the node value from the sum when recurring
down,
 * and check to see if the sum is 0 when you run out of tree.
 * @param sum
 * @return
 */
public boolean hasPathSum(int sum) {
    return hasPathSum(root, sum);
}

boolean hasPathSum(Node node, int sum) {
    // return true if we run out of tree and sum==0
    if (node == null) {
        return(sum == 0);
    }
    else {
        // otherwise check both subtrees
        int subSum = sum - node.data;
        return (hasPathSum(node.left, subSum) ||
hasPathSum(node.right, subSum));
    }
}
```

- *复制二叉树的每一个节点到它的左节点*

```
/** 复制二叉树的每一个节点到它的左节点
 * Changes the tree by inserting a duplicate node
 * on each nodes's .left.
 *
 *
 * So the tree...
 *        2
 *       / \
 *      1   3
 *
 * Is changed to...
 *        2
 *       / \
 *      2   3
```

```
 *     /   /
 *    1   3
 *   /
 *  1
 *
 * Uses a recursive helper to recur over the tree
 * and insert the duplicates.
 */
public void doubleTree() {
    doubleTree(root);
}

private void doubleTree(Node node) {
    Node oldLeft;
    if (node == null) return;

    // do the subtrees
    doubleTree(node.left);
    doubleTree(node.right);

    // duplicate this node to its left
    oldLeft = node.left;
    node.left = new Node(node.data);
    node.left.left = oldLeft;
}
```

- 比较两棵二叉树是否完全相同

```
/**比较两棵二叉树是否完全相同
 *         5
 *       / \
 *      3   6
 *     / \   \
 *    1   2   8
 * Compares the receiver to another tree to
 * see if they are structurally identical.
 ************************************************
 * Recursive helper -- recurs down two trees in parallel,
 * checking to see if they are identical.
```

```
 */
public boolean sameTree(BinaryTree other) {
    return( sameTree(root, other.root) );
}
boolean sameTree(Node a, Node b) {
    // 1. both empty -> true
    if (a==null && b==null) return(true);

    // 2. both non-empty -> compare them
    else if (a!=null && b!=null) {
      return (a.data == b.data &&
              sameTree(a.left, b.left) &&
              sameTree(a.right, b.right));
    }
    // 3. one empty, one not -> false
    else return(false);
}
```

- *判断树是否是二叉查找树方法1*

```
/** 判断树是否是二叉查找树方法1
 * Tests if a tree meets the conditions to be a
 * binary search tree (BST).

**********************************************************************
 * Recursive helper -- checks if a tree is a BST
 * using minValue() and maxValue() (not efficient).
 */
public boolean isBST1() {
    return(isBST(root));
}
private boolean isBST1(Node node) {
    if (node==null) return(true);

    // do the subtrees contain values that do not
    // agree with the node?
    if (node.left!=null && maxValue(node.left) > node.data)
return(false);
    if (node.right!=null && minValue(node.right) <= node.data)
```

```
return(false);

    // check that the subtrees themselves are ok
    return (isBST(node.left) && isBST(node.right));
}
```

- *判断树是否是二叉查找树方法2*

```
/**判断树是否是二叉查找树方法2
 * Tests if a tree meets the conditions to be a
 * binary search tree (BST). Uses the efficient
 * recursive helper.
 ***************************************************************
 * Efficient BST helper -- Given a node, and min and max values,
 * recurs down the tree to verify that it is a BST, and that all
 * its nodes are within the min..max range. Works in O(n) time --
 * visits each node only once.
 */
public boolean isBST2() {
    return( isBST2(root, Integer.MIN_VALUE, Integer.MAX_VALUE) );
}
private boolean isBST2(Node node, int min, int max) {
    if (node==null) {
        return (true);
    }
    else {
        // left should be in range  min...node.data
        boolean leftOk = isBST2(node.left, min, node.data);

        // if the left is not ok, bail out
        if(!leftOk) return (false);

        // right should be in range node.data+1..max
        boolean rightOk = isBST2(node.right, node.data+1, max);

        return (rightOk);
    }
}
```