by eamon in May 2015

Chapter 3 Binary Tree

Binary Tree DFS template

```
/*
Template 1 Traverse
*/
public class Solution {
  public void traverse(TreeNode root) {
    if (root == null) {
      return
    }
    traverse(root.left);
    traverse(root.right);
  }
}
/*
Template 2 Divide & Conquer
*/
public class Solution {
  public ResultType traversal(TreeNode root) {
    // null or leaf
    if (root == null) {
      // do something and return
    }

    // Divide
    ResultType left = traversal(root.left);
    ResultType right = traversal(root.right);
    // Conquer
    ResultType result = Merge from left and right
    return result;
  }
}
```

Binary Tree BFS Template

```
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List rst = new ArrayList();

        if (root == null) {
            return rst;
        }

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<Integer>();
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode head = queue.poll();
                level.add(head.val);
                if (head.left != null) {
                    queue.offer(head.left);
                }
                if (head.right != null) {
                    queue.offer(head.right);
                }
            }
            rst.add(level);
        }
        return rst;
    }
}
```

Binary Tree Preorder Traversal Total
Given a binary tree, return the preorder traversal of its nodes' values.
For example:
Given binary tree {1,#,2,3},

```
   1
   \
    2
   /
   3
```

return [1,2,3].
Answer:

```
/*
 *    1
 *   / \
 *  2   3
 * / \   \
 * 4  5   6
 *
 */
public class BTPreorderSol {
            /*
             * test
             */
            public static void BinaryTreePreorderTraversal(TreeNode root) {
                        if (root == null) {
                                    return;
                        }
                        System.out.print(root.val + "");
                        BinaryTreePreorderTraversal(root.left);
                        BinaryTreePreorderTraversal(root.right);
            }
            /*
             * Non-Recursion
             */
```

```
public static List<Integer> preorderTraversal0(TreeNode root) {
    Stack<TreeNode> stack = new Stack<TreeNode>();
    List<Integer> rst = new ArrayList<Integer>();
    if (root == null) {
        return rst;
    }
    stack.push(root);
    while (!stack.empty()) {
        TreeNode node = stack.pop();
        rst.add(node.val);
        if (node.right != null) {
            stack.push(node.right);
        }
        if (node.left != null) {
            stack.push(node.left);
        }
    }
    return rst;
}
/*
 * Traverse
 */
public static List<Integer> preorderTraversal1(TreeNode root) {
    List<Integer> rst = new ArrayList<Integer>();
    traverse(root, rst);
    return rst;
}

private static void traverse(TreeNode root, List<Integer> rst) {
    if (root == null) {
        return;
    }
    rst.add(root.val);
    traverse(root.left, rst);
    traverse(root.right, rst);
}
/*
 * Divide & Conquer
 */
public static List<Integer> preorderTraversal2(TreeNode root) {
    List<Integer> rst = new ArrayList<Integer>();
    if (root == null) {
        return rst;
    }
    // Divide
    List<Integer> left = preorderTraversal2(root.left);
    List<Integer> right = preorderTraversal2(root.right);
    // Conquer
    rst.add(root.val);
```

```
		rst.addAll(left);
		rst.addAll(right);
		return rst;
	}

	private static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		public TreeNode(int val) {
			this.val = val;
			left = null;
			right = null;
		}
	}

	public static void main(String[] args) {
		TreeNode r1 = new TreeNode(1);
		TreeNode r2 = new TreeNode(2);
		TreeNode r3 = new TreeNode(3);
		TreeNode r4 = new TreeNode(4);
		TreeNode r5 = new TreeNode(5);
		TreeNode r6 = new TreeNode(6);
		r1.left = r2;
		r1.right = r3;
		r2.left = r4;
		r2.right = r5;
		r3.right = r6;

		System.out.print("test: ");
		BinaryTreePreorderTraversal(r1);

		List<Integer> preorder0 = new ArrayList<Integer>();
		preorder0 = preorderTraversal0(r1);
		System.out.print("\n" + "preorder0: ");
		for (int i = 0; i < preorder0.size(); i++) {
			System.out.print(preorder0.get(i) + "");
		}

		List<Integer> preorder1 = new ArrayList<Integer>();
		preorder1 = preorderTraversal1(r1);
		System.out.print("\n" + "preorder1: ");
		for (int i = 0; i < preorder1.size(); i++) {
			System.out.print(preorder1.get(i) + "");
		}

		List<Integer> preorder2 = new ArrayList<Integer>();
		preorder2 = preorderTraversal1(r1);
		System.out.print("\n" + "preorder2: ");
```

```
            for (int i = 0; i < preorder2.size(); i++) {
                System.out.print(preorder2.get(i) + "");
            }
        }
}
/*
 * outputs:
 * test: 124536
 * preorder0: 124536
 * preorder1: 124536
 * preorder2: 124536
 */
```

Binary Tree Inorder Traversal
Given a binary tree, return the inorder traversal of its nodes' values.
For example:
Given binary tree {1,#,2,3},

```
  1
  \
   2
  /
  3
```

return [1,3,2].
Note: Recursive solution is trivial, could you do it iteratively?
Answer:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    // recursion
    public List<Integer> inorderTraversal1(TreeNode root) {
        List<Integer> rst = new ArrayList<Integer>();
        rec(root, rst);
        return rst;
    }

    public void rec(TreeNode root, List<Integer> rst) {
        if (root == null) {
            return;
        }

        rec(root.left, rst);
        rst.add(root.val);
        rec(root.right, rst);
    }
    // iteration
    public List<Integer> inorderTraversal2(TreeNode root) {
        List<Integer> rst = new ArrayList<Integer>();
        if (root == null) {
            return rst;
        }

        Stack<TreeNode> s = new Stack<TreeNode>();
        TreeNode cur = root;
        while (true) {
            while (cur != null) {
                s.push(cur);
                cur = cur.left;
            }
            if (s.isEmpty()) {
                break;
            }
            cur = s.pop();
            rst.add(cur.val);
            cur = cur.right;
        }
        return rst;
    }
}
```

Binary Tree Postorder Traversal
Given a binary tree, return the postorder traversal of its nodes' values.
For example:
Given binary tree {1,#,2,3},

```
   1
    \
     2
    /
   3
```

return [3,2,1].
Answer:

```
/*
从左到右的后序与从右到左的前序的逆序是一样的, 用另外一个栈进行翻转即可
123 -> 132 -> 231
*/
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    // recursion
    public List<Integer> postorderTraversal1(TreeNode root) {
        List<Integer> rst = new ArrayList<Integer>();
        if (root == null) {
            return rst;
        }
        dfs(root, rst);
        return rst;
    }

    public void dfs(TreeNode root, List<Integer> rst) {
        if (root == null) {
            return;
        }

        dfs(root.left, rst);
        dfs(root.right, rst);
        rst.add(root.val);
    }
    // iteration
    public List<Integer> postorderTraversal2(TreeNode root) {
        List<Integer> rst = new ArrayList<Integer>();
        if (root == null) {
            return rst;
        }

        Stack<TreeNode> s = new Stack<TreeNode>();
        Stack<Integer> out = new Stack<Integer>();
        s.push(root);
        while (!s.isEmpty()) {
            TreeNode cur = s.pop();
            out.push(cur.val);
            if (cur.left != null) {
                s.push(cur.left);
            }
            if (cur.right != null) {
                s.push(cur.right);
            }
        }

        while (!out.isEmpty()) {
            rst.add(out.pop());
        }
        return rst;
    }
}
```

Construct Binary Tree from Inorder and Postorder Traversal
Given inorder and postorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.
Answer:

```
/*
使用递归的思想，先找到根节点（它就是post order最后一个），然后再在inorder中找到它，以
确定左子树的node个数,
然后分别确定左子树右子树的左右边界，就是左右子树的划分关系
e.g.
{4, 5, 2, 7, 8, 1, 3}
inorder: 7 5 8 | 4 | 1 2 3
postorder: 7 8 5 | 1 3 2 | 4
*/
public class ConstructBT1Sol {
	public static TreeNode buildTree(int[] inorder, int[] postorder) {
		if (inorder == null || postorder == null) {
			return null;
		}

		return dfs(inorder, postorder, 0, inorder.length - 1, 0, postorder.length - 1);
	}

	public static TreeNode dfs(int[] inorder, int[] postorder, int inL, int inR, int postL, int postR)
{
		if (inL > inR) {
			return null;
		}
		// create the root node
		TreeNode root = new TreeNode(postorder[postR]);
		// find the position of the root node in the inorder
		int pos = 0;
		for (; pos <= inR; pos++) {
			if (inorder[pos] == postorder[postR]) {
				break;
			}
		}
		int leftNum = pos - inL;

		root.left = dfs(inorder, postorder, inL, pos - 1, postL, postL + leftNum - 1);
		root.right = dfs(inorder, postorder, pos + 1, inR, postL + leftNum, postR - 1);
		return root;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		int[] inorder = {2, 1, 3};
		int[] postorder = {2, 3, 1};

		print(buildTree(inorder, postorder));
	}

	public static void print(TreeNode root) {
		if (root == null) {
			return;
		}

		System.out.print(root.val + " ");
		print(root.left);
		print(root.right);
	}
}
```

Construct Binary Tree from Preorder and Inorder Traversal
Given preorder and inorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.
Answer:

```
/*
1. Find the root node from the preorder.(it is the first node.)
2. Try to find the position of the root in the inorder. Then we can get the number of nodes in the
left tree.
3. 递归调用，构造左子树和右子树。
*/
public class ConstructBT2Sol {
	public static TreeNode buildTree(int[] preorder, int[] inorder) {
		if (preorder == null || inorder == null || preorder.length == 0 || inorder.length == 0)
{
			return null;
		}

		return dfs(preorder, inorder, 0, preorder.length - 1, 0, inorder.length - 1);
	}

	public static TreeNode dfs(int[] preorder, int[] inorder, int preL, int preR, int inL, int inR) {
		if (preL > preR) {
			return null;
		}
		TreeNode root = new TreeNode(preorder[preL]);
		int position = 0;
		for (; position <= inR; position++) {
			if (inorder[position] == preorder[preL]) {
				break;
			}
		}
		int leftNum = position - inL;

		root.left = dfs(preorder, inorder, preL + 1, preL + leftNum, inL, position - 1);
		root.right = dfs(preorder, inorder, preL + leftNum + 1, preR, position + 1, inR);
		return root;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		int[] preorder = {1, 2, 3};
		int[] inorder = {2, 1, 3};
		print(buildTree(preorder, inorder));
	}

	public static void print(TreeNode root) {
		if (root == null) {
			return;
		}

		System.out.print(root.val + " ");
		print(root.left);
		print(root.right);
	}
}
```

Maximum Depth of Binary Tree
Given a binary tree, find its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.
Answer:

```
public class MaxDepthofBTSol {
	public static int maxDepth1(TreeNode root) {
		if (root == null) {
			return 0;
		}
		int left = maxDepth1(root.left);
		int right = maxDepth1(root.right);
		return Math.max(left, right) + 1;
	}

	public static int maxDepth2(TreeNode root) {
		if (root == null) {
			return 0;
		}

		TreeNode dummy = new TreeNode(0);
		Queue<TreeNode> q = new LinkedList<TreeNode>();
		q.offer(root);
		q.offer(dummy);
		int depth = 0;
		while (!q.isEmpty()) {
			TreeNode cur = q.poll();
			if (cur == dummy) {
				depth++;
				if (!q.isEmpty()) {
					q.offer(dummy);
				}
			}
			if (cur.left != null) {
				q.offer(cur.left);
			}
			if (cur.right != null) {
				q.offer(cur.right);
			}
		}
		return depth;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		public TreeNode (int x) {
			val = x;
		}
	}

	public static void main(String arg[]) {
		TreeNode t1 = new TreeNode(1);
		TreeNode t2 = new TreeNode(2);
		TreeNode t3 = new TreeNode(3);
		TreeNode t4 = new TreeNode(4);
		t1.left = t2;
		t1.right = t3;
		t2.left = t4;
		System.out.print(maxDepth1(t1));
		System.out.print("\n");
		System.out.print(maxDepth2(t1));
	}
}
/*
outputs:
3
3
*/
```

Minimum Depth of Binary Tree
Given a binary tree, find its minimum depth.
The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.
Answer:

```
public class MinDepthOfBTSol {
	// recursion
	public static int minDepth1(TreeNode root) {
		if (root == null) {
			return 0;
		}

		return dfs(root);
	}

	public static int dfs(TreeNode root) {
		if (root == null) {
			return Integer.MAX_VALUE;
		}

		if (root.left == null && root.right == null) {
			return 1;
		}

		return Math.min(dfs(root.left), dfs(root.right)) + 1;
	}
	// level traversal
	public static int minDepth2(TreeNode root) {
		if (root == null) {
			return 0;
		}

		int level = 0;
		Queue<TreeNode> q = new LinkedList<TreeNode>();
		q.offer(root);
		while (!q.isEmpty()) {
			int size = q.size();
			level++;
```

```
				for (int i = 0; i < size; i++) {
					TreeNode cur = q.poll();
					if (cur.left == null && cur.right == null) {
						return level;
					}
					if (cur.left != null) {
						q.offer(cur.left);
					}
					if (cur.right != null) {
						q.offer(cur.right);
					}
				}
			}
			return 0;
		}

		public static class TreeNode {
			int val;
			TreeNode left;
			TreeNode right;
			public TreeNode (int x) {
				val = x;
			}
		}

		public static TreeNode myTree() {
			TreeNode t1 = new TreeNode(1);
			TreeNode t2 = new TreeNode(2);
			TreeNode t3 = new TreeNode(3);
			TreeNode t4 = new TreeNode(4);
			t1.left = t2;
			t1.right = t3;
			t2.left = t4;
			return t1;
		}

		public static void main(String arg[]) {
			System.out.print(minDepth1(myTree()));
			System.out.print("\n");
			System.out.print(minDepth2(myTree()));
		}
}
/*
 * outputs:
 * 2
 * 2
 */
```

Balance binary tree
Given a binary tree, determine if it is height-balanced.
For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.
Answer:

```
public class BalanceBTSol {
	public static boolean isBalanced(TreeNode root) {
		return Helper(root) != -1;
	}

	private static int Helper(TreeNode root) {
		if (root == null) {
			return 0;
		}

		int left = Helper(root.left);
		int right = Helper(root.right);
		if (left == -1 || right == -1 || Math.abs(left - right) > 1) {
			return -1;
		}
		return Math.max(left, right) + 1;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode (int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		TreeNode t1 = new TreeNode(1);
		TreeNode t2 = new TreeNode(2);
		TreeNode t3 = new TreeNode(3);
		TreeNode t4 = new TreeNode(5);
		TreeNode t5 = new TreeNode(4);
		t1.left = t2;
		t1.right = t3;
		t2.left = t4;
		t4.right = t5;
		boolean rst = isBalanced(t1);
		System.out.print(rst);
	}
}
/*
 * output: false
 */
```

Binary Tree Maximum Path Sum
Given a binary tree, find the maximum path sum.
The path may start and end at any node in the tree.
For example:
Given the below binary tree,

```
       1
      / \
     2   3
```

Return 6.
Answer:

```
/*
计算树的最长path有2种情况：
1. 通过根的path.
  (1)如果左子树从左树根到任何一个Node的path大于零，可以链到root上
  (2)如果右子树从右树根到任何一个Node的path大于零，可以链到root上
2. 不通过根的path. 这个可以取左子树及右子树的path的最大值。
所以创建一个inner class:
记录2个值：
1. 本树的最大path。
2. 本树从根节点出发到任何一个节点的最大path.
注意，当root == null,以上2个值都要置为Integer_MIN_VALUE; 因为没有节点可取的时候，是不存在solution的。以免干扰递归的计算
if any of the path of left and right is below 0, don't add it.
注意，这里不可以把Math.max(left.maxSingle, right.maxSingle) 与root.val加起来，会有可能越界
may left.maxSingle and right.maxSingle are below 0
*/
public class BTMaxPathSumSol {
	public static int maxPathSum(TreeNode root) {
		return dfs(root).max;
	}

	public static class ReturnType {
		int maxSingle;
		int max;
		ReturnType(int maxSingle, int max) {
			this.max = max;
			this.maxSingle = maxSingle;
		}
	}

	public static ReturnType dfs(TreeNode root) {
		ReturnType ret = new ReturnType(Integer.MIN_VALUE, Integer.MIN_VALUE);
		if (root == null) {
			return ret;
		}

		ReturnType leftR = dfs(root.left);
		ReturnType rightR = dfs(root.right);
```

```
        // case1
        int cross = root.val;
        cross += Math.max(0, leftR.maxSingle);
        cross += Math.max(0, rightR.maxSingle);
        // case2
        int maxSingle = Math.max(leftR.maxSingle, rightR.maxSingle);
        maxSingle = Math.max(maxSingle, 0);
        maxSingle += root.val;
        ret.maxSingle = maxSingle;
        ret.max = Math.max(leftR.max, rightR.max);

        ret.max = Math.max(ret.max, cross);
        return ret;
    }

    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode(int x) {
            val = x;
        }
    }

    public static void main(String[] args) {
        TreeNode t1 = new TreeNode(1);
        TreeNode t2 = new TreeNode(2);
        TreeNode t3 = new TreeNode(3);
        t1.left = t2;
        t1.right = t3;
        int rst = maxPathSum(t1);
        System.out.print(rst);
    }
}
```

Lowest Common Ancestor
Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of the two nodes.
The lowest common ancestor is the node with largest depth which is the ancestor of both nodes.
Example

```
    4
  /   \
 3     7
     /   \
    5     6
```

For 3 and 5, the LCA is 4.
For 5 and 6, the LCA is 7.
For 6 and 7, the LCA is 7.

Answer:
```
public class LCASol {
    public static TreeNode lowestCommonAncestor1(TreeNode root, TreeNode A, TreeNode
B) {
        ArrayList<TreeNode> list1 = getPath2Root(A);
        ArrayList<TreeNode> list2 = getPath2Root(B);
        int i, j;
        for (i = list1.size() - 1, j = list2.size() - 1; i >= 0 && j >= 0; i--, j--) {
            if (list1.get(i) != list2.get(j)) {
                return list1.get(i).parent;
            }
        }
        return list1.get(i + 1);
    }

    private static ArrayList<TreeNode> getPath2Root(TreeNode node) {
        ArrayList<TreeNode> list = new ArrayList<TreeNode>();
        while (node != null) {
            list.add(node);
            node = node.parent;
        }
        return list;
    }
    // Divide & Conquer
    public static TreeNode lowestCommonAncestor2(TreeNode root, TreeNode A, TreeNode
B) {
        if (root == null || root == A || root == B) {
            return root;
        }
        TreeNode left = lowestCommonAncestor2(root.left, A, B);
        TreeNode right = lowestCommonAncestor2(root.right, A, B);
        if (left != null && right != null) {
            return root;
        }
        if (left != null) {
            return left;
        }
        if (right != null) {
            return right;
        }
        return null;
    }

    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode parent;
        TreeNode(int x) {
```

```
                        val = x;
                }
        }

        public static void main(String[] args) {
                TreeNode t1 = new TreeNode(4);
                TreeNode t2 = new TreeNode(3);
                TreeNode t3 = new TreeNode(7);
                TreeNode t4 = new TreeNode(5);
                TreeNode t5 = new TreeNode(6);
                t1.left = t2;
                t1.right = t3;
                t3.left = t4;
                t3.right = t5;
                t5.parent = t3;
                t4.parent = t3;
                t3.parent = t1;
                t2.parent = t1;
                t1.parent = null;
                TreeNode rst1 = lowestCommonAncestor1(t1, t2, t4);
                System.out.print(rst1.val);
                System.out.print("\n");
                TreeNode rst2 = lowestCommonAncestor1(t1, t4, t5);
                System.out.print(rst2.val);
                System.out.print("\n");
                TreeNode rst3 = lowestCommonAncestor1(t1, t5, t3);
                System.out.print(rst3.val);
                System.out.print("\n");
                TreeNode rst4 = lowestCommonAncestor2(t1, t2, t4);
                System.out.print(rst4.val);
                System.out.print("\n");
                TreeNode rst5 = lowestCommonAncestor2(t1, t4, t5);
                System.out.print(rst5.val);
                System.out.print("\n");
                TreeNode rst6 = lowestCommonAncestor2(t1, t5, t3);
                System.out.print(rst6.val);
        }
}
```

Binary Tree Level Order Traversal
Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).
For example:
Given binary tree {3,9,20,#,#,15,7},

```
   3
  / \
 9  20
   /  \
  15   7
```

return its level order traversal as:
```
[
  [3],
  [9,20],
  [15,7]
]
```
Answer:
```
public class BTLevelOrderTSol {
	public static List<List<Integer>> levelOrder1(TreeNode root) {
		List rst = new ArrayList();
		if (root == null) {
			return rst;
		}

		Queue<TreeNode> queue = new LinkedList<TreeNode>();
		queue.offer(root);
		while (!queue.isEmpty()) {
			List<Integer> level = new ArrayList<Integer>();
			int size = queue.size();
			for (int i = 0; i < size; i++) {
				TreeNode head = queue.poll();
				level.add(head.val);
				if (head.left != null) {
					queue.offer(head.left);
				}
				if (head.right != null) {
					queue.offer(head.right);
				}
			}
			rst.add(level);
		}
		return rst;
	}

	public static List<List<Integer>> levelOrder2(TreeNode root) {
		List<List<Integer>> rst = new ArrayList<List<Integer>>();
		levelVisit(root, 0, rst);
		return rst;
	}

	public static void levelVisit(TreeNode root, int level, List<List<Integer>> rst) {
		if (root == null) {
			return;
		}

		if (level >= rst.size()) {
			rst.add(new ArrayList<Integer>());
		}
```

```
                rst.get(level).add(root.val);
                levelVisit(root.left, level + 1, rst);
                levelVisit(root.right, level + 1, rst);
        }

        public static class TreeNode {
                int val;
                TreeNode left;
                TreeNode right;
                TreeNode(int x) {
                        val = x;
                }
        }

        public static void main(String arg[]) {
                TreeNode t1 = new TreeNode(3);
                TreeNode t2 = new TreeNode(9);
                TreeNode t3 = new TreeNode(20);
                TreeNode t4 = new TreeNode(15);
                TreeNode t5 = new TreeNode(7);
                t1.left = t2;
                t1.right = t3;
                t3.left = t4;
                t3.right = t5;
                System.out.print(levelOrder1(t1));
                System.out.print("\n");
                System.out.print(levelOrder2(t1));
        }
}
/*
 * output: [[3], [9, 20], [15, 7]]
 */
```

Binary Tree Level Order Traversal II
Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).
For example:
Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
    /  \
   15   7
```

return its bottom-up level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

by eamon in May 2015

Answer:
```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List rst = new ArrayList();
        if (root == null) {
            return rst;
        }

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<Integer>();
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode head = queue.poll();
                level.add(head.val);
                if (head.left != null) {
                    queue.offer(head.left);
                }
                if (head.right != null) {
                    queue.offer(head.right);
                }
            }
            rst.add(level);
        }
        Collections.reverse(rst);
        return rst;
    }
}
```

Binary Tree Zigzag Level Order Traversal
Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).
For example:
Given binary tree {3,9,20,#,#,15,7},
```
    3
   / \
  9  20
    /  \
   15   7
```

return its zigzag level order traversal as:
```
[
  [3],
  [20,9],
  [15,7]
]
```
Answer:
```
public class BTZLevelOrderTSol {
	public static List<List<Integer>> zigzaglevelOrder(TreeNode root) {
		List rst = new ArrayList();
		if (root == null) {
			return rst;
		}

		Queue<TreeNode> queue = new LinkedList<TreeNode>();
		queue.offer(root);
		int j = 1;
		while (!queue.isEmpty()) {
			List<Integer> level = new ArrayList<Integer>();
			int size = queue.size();
			int i;
			for (i = 0; i < size; i++) {
				TreeNode head =queue.poll();
				level.add(head.val);
				if (head.left != null) {
					queue.offer(head.left);
				}
				if (head.right != null) {
					queue.offer(head.right);
				}
			}
			if ((j - 1) % 2 == 0) {
				j++;
			} else {
				j++;
				Collections.reverse(level);
			}
			rst.add(level);
		}
		return rst;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
```

```
        }

        public static void main(String[] args) {
                TreeNode t1 = new TreeNode(3);
                TreeNode t2 = new TreeNode(9);
                TreeNode t3 = new TreeNode(20);
                TreeNode t4 = new TreeNode(15);
                TreeNode t5 = new TreeNode(7);
                t1.left = t2;
                t1.right = t3;
                t3.left = t4;
                t3.right = t5;
                List rst = new ArrayList();
                rst = zigzaglevelOrder(t1);
                System.out.print(rst);
        }
}
```

Binary Tree Right Side View
Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.
For example:
Given the following binary tree,

```
   1         <---
 /   \
2     3       <---
 \     \
  5     4     <---
```

You should return [1, 3, 4].
Answer:

```
public class BTRightSideViewSol {
        public static List<Integer> rightSideView(TreeNode root) {
                List<Integer> rst = new ArrayList<Integer>();
                if (root == null) {
                        return rst;
                }

                Queue<TreeNode> queue = new LinkedList<TreeNode>();
                queue.offer(root);
                while (!queue.isEmpty()) {
                        int size = queue.size();
                        for (int i = 0; i < size; i++) {
                                TreeNode head = queue.poll();
                                if (i == 0) {
                                        rst.add(head.val);
                                }
                                if (head.right != null) {
                                        queue.offer(head.right);
                                }
```

```
                    if (head.left != null) {
                        queue.offer(head.left);
                    }
                }
            }
            return rst;
        }

        public static class TreeNode {
            int val;
            TreeNode left;
            TreeNode right;
            TreeNode(int x) {
                val = x;
            }
        }

        public static TreeNode myTree() {
            TreeNode t1 = new TreeNode(1);
            TreeNode t2 = new TreeNode(2);
            TreeNode t3 = new TreeNode(3);
            TreeNode t4 = new TreeNode(4);
            TreeNode t5 = new TreeNode(5);
            t1.left = t2;
            t1.right = t3;
            t2.right = t5;
            t3.right = t4;
            return t1;
        }

        public static void main(String[] args) {
            System.out.print(rightSideView(myTree()));
        }
}
```

Same Tree
Given two binary trees, write a function to check if they are equal or not.
Two binary trees are considered equal if they are structurally identical and the nodes have the same value.
Answer:

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

by eamon in May 2015

```
public class Solution {
    public boolean isSameTree1(TreeNode p, TreeNode q) {
        if (p == null && q == null) {
            return true;
        } else if (p == null || q == null) {
            return false;
        } else if (p.val == q.val) {
            if (isSameTree(p.left, q.left) && isSameTree(p.right, q.right)) {
                return true;
            } else {
                return false;
            }
        }
        return false;
    }

    public boolean isSameTree2(TreeNode p, TreeNode q) {
        if (p == null && q == null) {
            return true;
        }
        if (p == null || q == null) {
            return false;
        }
        Stack<TreeNode> s1 = new Stack<TreeNode>();
        Stack<TreeNode> s2 = new Stack<TreeNode>();
        TreeNode cur1 = p;
        TreeNode cur2 = q;
        while (true) {
            while (cur1 != null && cur2 != null) {
                s1.push(cur1);
                s2.push(cur2);
                cur1 = cur1.left;
                cur2 = cur2.left;
            }
            if (cur1 != null || cur2 != null) {
                return false;
            }
            if (s1.isEmpty() && s2.isEmpty()) {
                break;
            }
            cur1 = s1.pop();
            cur2 = s2.pop();
            if (cur1.val != cur2.val) {
                return false;
            }
            cur1 = cur1.right;
            cur2 = cur2.right;
        }
        return true;
```

```
   }
}
```

Symmetric Tree
Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).
For example, this binary tree is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

Note:
Bonus points if you could solve it both recursively and iteratively.
Answer:

```
public class SymmetricTreeSol {
	// recursion
	public static boolean isSymmetric1(TreeNode root) {
		return isMirror1(root.left, root.right);
	}

	public static boolean isMirror1(TreeNode root1, TreeNode root2) {
		if (root1 == null && root2 == null) {
			return true;
		} else if (root1 == null || root2 == null) {
			return false;
		}
		return root1.val == root2.val && isMirror1(root1.left, root2.right) &&
isMirror1(root1.right, root2.left);
	}
	// iteration
	public static boolean isSymmetric2(TreeNode root) {
		if (root == null) {
			return true;
		}

		return isMirror2(root.left, root.right);
	}

	public static boolean isMirror2(TreeNode root1, TreeNode root2) {
		if (root1 == null && root2 == null) {
			return true;
		}
		if (root1 == null || root2 == null) {
```

```
            return false;
        }
        Stack<TreeNode> s1 = new Stack<TreeNode>();
        Stack<TreeNode> s2 = new Stack<TreeNode>();
        while (!s1.isEmpty() && !s2.isEmpty()) {
            TreeNode cur1 = s1.pop();
            TreeNode cur2 = s2.pop();
            if (cur1.val != cur2.val) {
                return false;
            }
            if (cur1.left != null && cur2.right != null) {
                s1.push(cur1.left);
                s2.push(cur2.right);
            } else if (!(cur1.left == null && cur2.right == null)) {
                return false;
            }
            if (cur1.right != null && cur2.left != null) {
                s1.push(cur1.right);
                s2.push(cur2.left);
            } else if (!(cur1.right == null && cur2.left == null)) {
                return false;
            }
        }
        return true;
    }

    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode(int x) {
            val = x;
        }
    }

    public static TreeNode myTree() {
        TreeNode t1 = new TreeNode(1);
        TreeNode t2 = new TreeNode(2);
        TreeNode t3 = new TreeNode(2);
        TreeNode t4 = new TreeNode(3);
        TreeNode t5 = new TreeNode(4);
        TreeNode t6 = new TreeNode(4);
        TreeNode t7 = new TreeNode(3);
        t1.left = t2;
        t1.right = t3;
        t2.left = t4;
        t2.right = t5;
        t3.left = t6;
        t3.right = t7;
```

```
		return t1;
	}

	public static void main(String[] args) {
		System.out.print(isSymmetric1(myTree()));
		System.out.print("\n");
		System.out.print(isSymmetric2(myTree()));
	}
}
```

Flatten Binary Tree to Linked List
Given a binary tree, flatten it to a linked list in-place.
For example,
Given

```
        1
       / \
      2   5
     / \   \
    3   4   6
```

The flattened tree should look like:

```
   1
    \
     2
      \
       3
        \
         4
          \
           5
            \
             6
```

Hints:
If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.
Answer:

```
/*
only init the root tree, can connect the left tree or the right tree
*/
public class FlattenBTToLLSol {
	public static void flatten(TreeNode root) {
		dfs(root);
	}

	public static TreeNode dfs(TreeNode root) {
		if (root == null) {
			return null;
		}

		TreeNode left = root.left;
		TreeNode right = root.right;
		root.left = null;
		root.right = null;
		TreeNode tail = root;
		if (left != null) {
			tail.right = left;
			tail = dfs(left);
		}
		if (right != null) {
			tail.right = right;
			tail = dfs(right);
		}
		return tail;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		TreeNode t1 = new TreeNode(1);
		TreeNode t2 = new TreeNode(2);
		TreeNode t3 = new TreeNode(3);
		TreeNode t4 = new TreeNode(4);
		TreeNode t5 = new TreeNode(5);
		TreeNode t6 = new TreeNode(6);
		t1.left = t2;
		t1.right = t5;
		t2.left = t3;
		t2.right = t4;
		t5.right = t6;
		TreeNode rst = new TreeNode(0);
		flatten(t1);
		print(t1);
	}

	public static void print(TreeNode root) {
		if (root == null) {
			return;
		}
		System.out.print(root.val + " ");
		print(root.right);
	}
}
```

Validate Binary Search Tree
Given a binary tree, determine if it is a valid binary search tree (BST).
Assume a BST is defined as follows:
The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.
Answer:

```
/*
recursive reasoning
dfs(5)
left = dfs(3)
 left = dfs(2)
  left = dfs(1)
   left(null): return ret = new ReturnType(Integer.MAX_VALUE, Integer.MIN_VALUE, true);
   right(null): return ret = new ReturnType(Integer.MAX_VALUE, Integer.MIN_VALUE, true);
   from the return, we can know:
   left.isBST = true, right.isBST = true;
   left.min = Integer.MAX_VALUE, right.min = Integer.MAX_VALUE;
   left.max = Integer.MIN_VALUE, right.max = Integer.MIN_VALUE;
  continue
  return new ReturnType(Math.min(1, Integer.MAX_VALUE), Math.max(1, Integer.MIN_VALUE),
true);
 continue
 left(1): return new ReturnType(Math.min(1, Integer.MAX_VALUE), Math.max(1,
Integer.MIN_VALUE), true);
 right(null): return new ReturnType(Integer.MAX_VALUE, Integer.MIN_VALUE, true);
 left.min = 1, right.min = Integer.MAX_VALUE
 left.max = 1 < 2, right.max = Integer.MIN_VALUE
 root.right == null
 return new ReturnType(Math.min(2, 1), Math.max(2, Integer.MIN_VALUE), true);
...
right = dfs(8)
*/
public class ValidateBSTSol {
	// inorder traversal
	public static boolean isValidBST1(TreeNode root) {
		if (root == null) {
			return true;
		}

		Stack<TreeNode> s = new Stack<TreeNode>();
		TreeNode cur = root;
		TreeNode pre = null;
		while (true) {
			while (cur != null) {
				s.push(cur);
				cur = cur.left;
			}
```

```
                if (s.isEmpty()) {
                    break;
                }
                cur = s.pop();
                if (pre != null && pre.val >= cur.val) {
                    return false;
                }
                pre = cur;
                cur = cur.right;
            }
            return true;
        }
        // low-up bounds
        public static boolean isValidBST2(TreeNode root) {
            if (root == null) {
                return true;
            }

            return dfs(root, Long.MIN_VALUE, Long.MAX_VALUE);
        }

        public static boolean dfs(TreeNode root, long low, long up) {
            if (root == null) {
                return true;
            }
            if (root.val >= up || root.val <= low) {
                return false;
            }
            return dfs(root.left, low, root.val) && dfs(root.right, root.val, up);
        }
        // recursive
        public static boolean isValidBST3(TreeNode root) {
            if (root == null) {
                return true;
            }

            return dfs(root).isBST;
        }

        public static class ReturnType {
            int min;
            int max;
            boolean isBST;
            public ReturnType (int min, int max, boolean isBST) {
                this.min = min;
                this.max = max;
                this.isBST = isBST;
            }
        }
```

```
	public static ReturnType dfs(TreeNode root) {
		ReturnType ret = new ReturnType(Integer.MAX_VALUE, Integer.MIN_VALUE, 
true);
		if (root == null) {
			return ret;
		}

		ReturnType left = dfs(root.left);
		ReturnType right = dfs(root.right);
		if (!left.isBST || !right.isBST) {
			ret.isBST = false;
			return ret;
		}
		if (root.left != null && root.val <= left.max) {
			ret.isBST = false;
			return ret;
		}
		if (root.right != null && root.val >= right.min) {
			ret.isBST = false;
			return ret;
		}
		return new ReturnType(Math.min(root.val, left.min), Math.max(root.val, 
right.max), true);
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		TreeNode t1 = new TreeNode(5);
		TreeNode t2 = new TreeNode(3);
		TreeNode t3 = new TreeNode(2);
		TreeNode t4 = new TreeNode(4);
		TreeNode t5 = new TreeNode(1);
		TreeNode t6 = new TreeNode(8);
		TreeNode t7 = new TreeNode(6);
		TreeNode t8 = new TreeNode(9);
		TreeNode t9 = new TreeNode(7);
		t1.left = t2;
		t1.right = t6;
		t2.left = t3;
		t2.right = t4;
		t3.left = t5;
		t6.left = t7;
		t6.right = t8;
		t7.right = t9;
		System.out.print(isValidBST1(t1));
		System.out.print("\n");
		System.out.print(isValidBST2(t1));
		System.out.print("\n");
		System.out.print(isValidBST3(t1));
	}
}
```

Insert Node in Binary Search Tree
Answer:

```
public class InsertNodeInBSTSol {
	public static TreeNode insertNode1(TreeNode root, TreeNode node) {
		if (root == null) {
			root = node;
			return root;
		}

		TreeNode tmp = root;
		TreeNode last = null;
		while (tmp != null) {
			last = tmp;
			if (tmp.val > node.val) {
				tmp = tmp.left;
			} else {
				tmp = tmp.right;
			}
		}
		if (last != null) {
			if (last.val > node.val) {
				last.left = node;
			} else {
				last.right = node;
			}
		}
		return root;
	}
	// recursive
	public static TreeNode insertNode2(TreeNode root, TreeNode node) {
		if (root == null) {
			return node;
		}

		if (root.val > node.val) {
			root.left = insertNode2(root.left, node);
		} else {
```

```
                root.right = insertNode2(root.right, node);
            }
            return root;
        }

        public static class TreeNode {
            int val;
            TreeNode left;
            TreeNode right;
            TreeNode(int x) {
                val = x;
            }
        }

        public static void main(String[] args) {
            TreeNode t1 = new TreeNode(6);
            TreeNode t2 = new TreeNode(5);
            TreeNode t3 = new TreeNode(9);
            TreeNode t4 = new TreeNode(7);
            TreeNode t5 = new TreeNode(8);
            t1.left = t2;
            t1.right = t3;
            t3.left = t4;
            print(t1);
            System.out.print("\n");
            insertNode1(t1, t5);
            print(t1);
            System.out.print("\n");
            t4.right = null;
            print(t1);
            System.out.print("\n");
            insertNode2(t1, t5);
            print(t1);
        }

        public static void print(TreeNode root) {
            if (root == null) {
                return;
            }

            System.out.print(root.val + " ");
            print(root.left);
            print(root.right);
        }
}
```

Search Range in Binary Search Tree
Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree. Find all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Return all the keys in ascending order.
Example
For example, if k1 = 10 and k2 = 22, then your function should print 12, 20 and 22.

```
       20
     /     \
   8        22
  /   \
4      12
```

Answer:

```
public class SearchRangeBSTSol {
        private static ArrayList<Integer> rst;

        public static ArrayList<Integer> searchRange(TreeNode root, int k1, int k2) {
      rst = new ArrayList<Integer>();
      helper(root, k1, k2);
      return rst;
   }

        private static void helper(TreeNode root, int k1, int k2) {
                if (root == null) {
                        return;
                }

                if (root.val > k1) {
                        helper(root.left, k1, k2);
                }
                if (root.val >= k1 && root.val <= k2) {
                        rst.add(root.val);
                }
                if (root.val < k2) {
                        helper(root.right, k1, k2);
                }
        }

        public static class TreeNode {
                int val;
                TreeNode left;
                TreeNode right;
                TreeNode(int x) {
                        val = x;
                }
        }

        private static TreeNode myTree() {
                TreeNode t1 = new TreeNode(20);
                TreeNode t2 = new TreeNode(8);
```

```
		TreeNode t3 = new TreeNode(22);
		TreeNode t4 = new TreeNode(4);
		TreeNode t5 = new TreeNode(12);
		t1.left = t2;
		t1.right = t3;
		t2.left = t4;
		t2.right = t5;
		return t1;
	}

	public static void main(String[] args) {
		searchRange(myTree(), 12, 22);
		System.out.print(rst);
	}
}
```

Binary Search Tree Iterator
Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.
Calling next() will return the next smallest number in the BST.
Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.
Answer:

```
public class BSTIteratorSol {
	// recursion
	public static class BSTIterator1 {
		ArrayList<TreeNode> list;
		int index;

		public BSTIterator1(TreeNode root) {
			list = new ArrayList<TreeNode>();
			dfs(root, list);
			index = 0;
		}
		/** @return whether we have a next smallest number */
		public boolean hasNext() {
			if (index < list.size()) {
				return true;
			}
			return false;
		}
		/** @return the next smallest number */
		public int next() {
			return list.get(index++).val;
		}

		public void dfs(TreeNode root, ArrayList<TreeNode> ret) {
			if (root == null) {
				return;
```

```
                }

                dfs(root.left, ret);
                ret.add(root);
                dfs(root.right, ret);
            }
        }
        // iteration
        public static class BSTIterator2 {
            ArrayList<TreeNode> list;
            int index;

            public BSTIterator2(TreeNode root) {
                list = new ArrayList<TreeNode>();
                iterator(root, list);
                index = 0;
            }

            public boolean hasNext() {
                if (index < list.size()) {
                    return true;
                }
                return false;
            }

            public int next() {
                return list.get(index++).val;
            }

            public void iterator(TreeNode root, ArrayList<TreeNode> ret) {
                if (root == null) {
                    return;
                }

                Stack<TreeNode> s = new Stack<TreeNode>();
                TreeNode cur = root;
                while (true) {
                    while (cur != null) {
                        s.push(cur);
                        cur = cur.left;
                    }
                    if (s.isEmpty()) {
                        break;
                    }
                    cur = s.pop();
                    ret.add(cur);
                    cur = cur.right;
                }
            }
```

```
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static TreeNode myTree() {
		TreeNode t1 = new TreeNode(6);
		TreeNode t2 = new TreeNode(5);
		TreeNode t3 = new TreeNode(9);
		TreeNode t4 = new TreeNode(7);
		t1.left = t2;
		t1.right = t3;
		t3.left = t4;
		return t1;
	}

	public static void main(String[] args) {
		BSTIterator1 i = new BSTIterator1(myTree());
		while (i.hasNext()) {
			System.out.print(i.next() + " ");
		}
		System.out.print("\n");
		BSTIterator2 j = new BSTIterator2(myTree());
		while (j.hasNext()) {
			System.out.print(j.next() + " ");
		}
	}
}
/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = new BSTIterator(root);
 * while (i.hasNext()) v[f()] = i.next();
 */
/*
 *outputs:
 *5 6 7 9
 *5 6 7 9
 */
```

Remove Node in Binary Search Tree
Given a root of Binary Search Tree with unique value for each node. Remove the node with given value. If there is no such a node with given value in the binary search tree, do nothing. You should keep the tree still a binary search tree after removal.

by eamon in May 2015

Example
Given binary search tree:
```
        5
      /   \
    3        6
 /   \
2      4
```
Remove 3, you can either return:
```
        5
      /   \
    2        6
     \
       4
```
or :
```
        5
      /   \
    4        6
 /
2
```
Answer:
```
public class RemoveNodeBSTSol {
	public static TreeNode removeNode(TreeNode root, int value) {
		TreeNode dummy = new TreeNode(0);
		dummy.left = root;
		TreeNode parent = findNode(dummy, root, value);
		TreeNode node;
		if (parent.left != null && parent.left.val == value) {
			node = parent.left;
		} else if (parent.right != null && parent.right.val == value) {
			node = parent.right;
		} else {
			return dummy.left;
		}
		deleteNode(parent, node);
		return dummy.left;
	}

	private static TreeNode findNode(TreeNode parent, TreeNode node, int value) {
		if (node == null) {
			return parent;
		}
		if (node.val == value) {
			return parent;
		}
		if (value < node.val) {
			return findNode(node, node.left, value);
		} else {
			return findNode(node, node.right, value);
		}
	}

	private static void deleteNode(TreeNode parent, TreeNode node) {
		if (node.right == null) {
			if (parent.left == node) {
				parent.left = node.left;
			} else {
				parent.right = node.left;
			}
		} else {
			TreeNode tmp = node.right;
			TreeNode father = node;
			while (tmp.left != null) {
				father = tmp;
				tmp = tmp.left;
			}
			if (father.left == tmp) {
				father.left = tmp.right;
			} else {
				father.right = tmp.right;
			}
			if (parent.left == node) {
				parent.left = tmp;
			} else {
				parent.right = tmp;
			}
			tmp.left = node.left;
			tmp.right = node.right;
		}
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		TreeNode t1 = new TreeNode(5);
		TreeNode t2 = new TreeNode(3);
		TreeNode t3 = new TreeNode(6);
		TreeNode t4 = new TreeNode(2);
		TreeNode t5 = new TreeNode(4);
		t1.left = t2;
		t1.right = t3;
		t2.left = t4;
		t2.right = t5;
		print(t1);
		System.out.print("\n");
		removeNode(t1, 3);
		print(t1);
	}

	public static void print(TreeNode root) {
		if (root == null) {
			return;
		}

		System.out.print(root.val + " ");
		print(root.left);
		print(root.right);
	}
}
/*
 *outputs:
 *5 3 2 4 6
 *5 4 2 6
 */
```

Convert Sorted List to Binary Search Tree
Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.
Answer:

```
/*
method 1
每次遍历当前list, 找到中间的节点, 建立root, 分别使用递归建立左树以及右树, 并将左右树挂在root之下,
建立root次数为N, 每次遍历最多N次, so the worst case is N^2
*/
public class sortedListToBSTSol {
	// method 1
	public static TreeNode sortedListToBST1(ListNode head) {
		ListNode fast = head;
		ListNode slow = head;
		ListNode pre = head;
		if (head == null) {
			return null;
		}
		TreeNode root = null;
		if (head.next == null) {
			root = new TreeNode(head.val);
			root.left = null;
			root.right = null;
			return root;
		}
```

```
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            pre = slow;
            slow = slow.next;
        }
        pre.next = null;
        TreeNode left = sortedListToBST1(head);
        TreeNode right = sortedListToBST1(slow.next);
        root = new TreeNode(slow.val);
        root.left = left;
        root.right = right;
        return root;
    }
    // Bottom-up
    public static TreeNode sortedListToBST2(ListNode head) {
        int size;
        cur = head;
        size = getListLength(head);
        return helper2(size);
    }

    private static ListNode cur;

    private static int getListLength(ListNode head) {
        int size = 0;
        while (head != null) {
            size++;
            head = head.next;
        }
        return size;
    }

    public static TreeNode helper2(int size) {
        if (size <= 0) {
            return null;
        }

        TreeNode left = helper2(size / 2);
        TreeNode root = new TreeNode(cur.val);
        cur = cur.next;
        TreeNode right = helper2(size - 1 - size / 2);
        root.left = left;
        root.right = right;
        return root;
    }

    public static class ListNode {
        int val;
        ListNode next;
```

```
		ListNode(int x) {
			val = x;
		}
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}

	public static ListNode myList() {
		ListNode n1 = new ListNode(1);
		ListNode n2 = new ListNode(2);
		ListNode n3 = new ListNode(3);
		n1.next = n2;
		n2.next = n3;
		n3.next = null;
		return n1;
	}

	public static void main(String[] args) {
		print(sortedListToBST1(myList()));
		System.out.print("\n");
		print(sortedListToBST2(myList()));
	}

	public static void print(TreeNode root) {
		if (root == null) {
			return;
		}

		print(root.left);
		System.out.print(root.val + " ");
		print(root.right);
	}
}
```

Convert Sorted Array to Binary Search Tree
Given an array where elements are sorted in ascending order, convert it to a height balanced BST.
Answer:

```
/*
the difference between LinkedList and Array is getting the Nth element value
*/
```

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        if (nums == null) {
            return null;
        }

        return buildTree(nums, 0, nums.length - 1);
    }

    private TreeNode buildTree(int[] num,  int start, int end) {
        if (start > end) {
            return null;
        }

        TreeNode node = new TreeNode(num[(start + end) / 2]);
        node.left = buildTree(num, start, (start + end) / 2 - 1);
        node.right = buildTree(num, (start + end) / 2 + 1, end);
        return node;
    }
}
```

Recover Binary Search Tree
Two elements of a binary search tree (BST) are swapped by mistake.
Recover the tree without changing its structure.
Note:
A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?
Answer:

```
/*
         8 (4)
        / \
       3  10
      / \   \
     1   6  14
        /\  /
        4 7 13
       (8)
```

Find the place which the order is wrong.
inorder
For example: 1 3 4 6 7 8 10 13 14

```
Wrong order: 1 3 8 6 7 4 10 13 14
FIND:            ___
Then we find:        ___
8, 6是错误的序列, 但是, 7, 4也是错误的序列。
因为8, 6前面的序列是正确的, 所以8, 6一定是后面的序列交换来的。
而后面的是比较大的数字, 也就是说8一定是被交换过来的。而7, 4中也应该是小的数字4是前面交
换过来的。
*/
/*
inOrder(3)
 inOrder(1)
  pre = 1
 pre = 1 < 3
 pre = 3
 inOrder(6)
  inOrder(8)
   pre = 3 < 8
   pre = 8
  pre = 8 > 6
  first = 8
  second = 6
  pre = 6
  inOrder(7)
   pre = 6 < 7
   pre = 7
pre = 7 > 4
second = 4
inOrder(10)
 pre = 7 < 10
 pre = 10
 inOrder(14)
  inOrder(13)
   pre = 10 < 13
   pre = 13
*/
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void recoverTree(TreeNode root) {
        inOrder(root);
        int tmp = first.val;
        first.val = second.val;
        second.val = tmp;
    }

    TreeNode pre = null;
    TreeNode first = null;
    TreeNode second = null;

    public void inOrder(TreeNode root) {
        if (root == null) {
            return;
        }

        inOrder(root.left);
        if (pre != null && pre.val > root.val) {
            if (first == null) {
                first = pre;
                second = root;
            } else {
                second = root;
            }
        }
        pre = root;
        inOrder(root.right);
    }
}
```

Unique Binary Search Trees
Given n, how many structurally unique BST's (binary search trees) that store values 1...n?
For example,
Given n = 3, there are a total of 5 unique BST's.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

Answer:

```
/*
The case for 3 elements example
left nodes possibilities * right nodes possibilities, 0 - n total n + 1,
Count[3] = Count[0] * Count[2] (1 as root) 3, 2 > 1, so there is no possible for 3, 2 in left, so 0
           + Count[1] * Count[1] (2 as root)
           + Count[2] * Count[0] (3 as root) 2, 1 < 3, so there is no possible for 2, 1 in right, so 0
*/
public class Solution {
    public int numTrees(int n) {
        int[] count = new int[n + 1];
        count[0] = 1;
        count[1] = 1;
        for (int i = 2; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                count[i] += count[j] * count[i - j - 1];
            }
        }
        return count[n];
    }
}
```

Unique Binary Search Trees II
Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.
For example,
Given n = 3, your program should return all 5 unique BST's shown below.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

Answer:

```
/*
使用递归来做。
1. 先定义递归的参数为左边界、右边界，即1到n.
2. 考虑从left, 到right 这n个数字中选取一个作为根，余下的使用递归来构造左右子树。
3. 当无解时，应该返回一个null树，这样构造树的时候，我们会比较方便，不会出现左边解为
空，或是右边解为空的情况。
4. 如果说左子树有n种组合，右子树有m种组合，那最终的组合数就是n*m. 把这所有的组合组装
起来即可
求出左右子树的所有的可能,
将左右子树的所有的可能性全部组装起来,
先创建根节点,
将组合出来的树加到结果集合中,
null也是一种解，也需要把它加上去。这样在组装左右子树的时候，不会出现左边没有解的情
况，或是右边没有解的情况
*/
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<TreeNode> generateTrees(int n) {
        return dfs(1, n);
    }

    public List<TreeNode> dfs(int left, int right) {
        List<TreeNode> ret = new ArrayList<TreeNode>();
        if (left > right) {
            ret.add(null);
            return ret;
        }
        for (int i = left; i <= right; i++) {
            List<TreeNode> lTree = dfs(left, i - 1);
            List<TreeNode> rTree = dfs(i + 1, right);
            for (TreeNode nodeL : lTree) {
                for (TreeNode nodeR : rTree) {
                    TreeNode root = new TreeNode(i);
                    root.left = nodeL;
                    root.right = nodeR;
                    ret.add(root);
                }
            }
        }
        return ret;
    }
}
```

add some general questions

```
/*
 * A perfect binary tree is a binary tree in which all leaves have the same depth or same level.
 * A full binary tree is a tree in which every node in the tree has either 0 or 2 children.
 * A complete binary tree is a binary tree in which every level, except possibly the last, is
completely filled, and all nodes are as far left as possible.
 * 树的深度是从根节点开始（其深度为1）自顶向下逐层累加的，而高度是从叶节点开始（其高度
为1）自底向上逐层累加的。虽然树的深度和高度一样，但是具体到树的某个节点，其深度和高
度是不一样的。非根非叶结点的深度是从根节点数到它的，高度是从叶节点数到它的。
 *   complete              full
 * k 2^(h - 1) <= k < 2^h - 1 k = 2^h - 1
 * h h = logk + 1          h = log(k + 1)
 */
public class TreeSol {
        public static int getNodeNumRec(TreeNode root) {
                if (root == null) {
                        return 0;
                }

                return getNodeNumRec(root.left) + getNodeNumRec(root.right) + 1;
        }

        public static int getNodeNum(TreeNode root) {
                if (root == null) {
                        return 0;
                }

                Queue<TreeNode> q = new LinkedList<TreeNode>();
                q.offer(root);
                int cnt = 0;
```

```
		while (!q.isEmpty()) {
			TreeNode node = q.poll();
			if (node.left != null) {
				q.offer(node.left);
			}
			if (node.right != null) {
				q.offer(node.right);
			}
			cnt++;
		}
		return cnt;
	}

	public static int getNodeNumKthLevelRec(TreeNode root, int k) {
		if (root == null || k <= 0) {
			return 0;
		}
		if (k == 1) {
			return 1;
		}

		return getNodeNumKthLevelRec(root.left, k - 1) +
getNodeNumKthLevelRec(root.right, k - 1);
	}

	public static int getNodeNumKthLevel(TreeNode root, int k) {
		if (root == null || k <= 0) {
			return 0;
		}
		int level = 0;
		Queue<TreeNode> q = new LinkedList<TreeNode>();
		q.offer(root);
		TreeNode dummy = new TreeNode(0);
		int cnt = 0;
		q.offer(dummy);
		while (!q.isEmpty()) {
			TreeNode node = q.poll();
			if (node == dummy) {
				level++;
				if (level == k) {
					return cnt;
				}
				cnt = 0;
				if (q.isEmpty()) {
					break;
				}
				q.offer(dummy);
				continue;
			}
```

```
                    cnt++;
                    if (node.left != null) {
                            q.offer(node.left);
                    }
                    if (node.right != null) {
                            q.offer(node.right);
                    }
            }
            return 0;
 }

    public static int getNodeNumLeaf(TreeNode root) {
            if (root == null) {
                    return 0;
            }

            int cnt = 0;
            Stack<TreeNode> s = new Stack<TreeNode>();
            TreeNode cur = root;
            while (true) {
                    while (cur != null) {
                            s.push(cur);
                            cur = cur.left;
                    }
                    if (s.isEmpty()) {
                            break;
                    }
                    cur = s.pop();
                    if (cur.left == null && cur.right == null) {
                            cnt++;
                    }
                    cur = cur.right;
            }
            return cnt;
    }

    public static int getMaxDistanceRec(TreeNode root) {
            return helper1(root).maxDistance;
    }

    public static ResultType helper1(TreeNode root) {
            ResultType ret = new ResultType(-1, -1);
            if (root == null) {
                    return ret;
            }
            ResultType left = helper1(root.left);
            ResultType right = helper1(root.right);
            ret.depth = Math.max(left.depth, right.depth) + 1;
            int cross = left.depth + right.depth + 2;
            ret.maxDistance = Math.max(left.maxDistance, right.maxDistance);
            ret.maxDistance = Math.max(ret.maxDistance, cross);
            return ret;
    }

    private static class ResultType {
            int depth;
            int maxDistance;
            public ResultType(int depth, int maxDistance) {
                    this.depth = depth;
                    this.maxDistance = maxDistance;
            }
    }

    public static boolean isCompleteBTRec(TreeNode root) {
            return helper2(root).isComplete;
    }

    public static ReturnType2 helper2(TreeNode root) {
        ReturnType2         ret = new ReturnType2(true, true, -1);
        if (root == null) {
            return ret;
        }
        ReturnType2 left = helper2(root.left);
        ReturnType2 right = helper2(root.right);
        ret.height = Math.max(left.height, right.height) + 1;
        ret.isPerfect = false;
        if (left.isPerfect && right.isPerfect && left.height == right.height) {
            ret.isPerfect = true;
        }
        ret.isComplete = ret.isPerfect
                    || (left.isComplete && right.isPerfect && left.height == right.height + 1)
                    || (left.isPerfect && right.isComplete && left.height == right.height);
        return ret;
    }

    private static class ReturnType2 {
            boolean isComplete;
            boolean isPerfect;
            int height;
            ReturnType2(boolean isComplete, boolean isPerfect, int height) {
                    this.isComplete = isComplete;
                    this.isPerfect = isPerfect;
                    this.height = height;
            }
    }

    public static int findLongest1(TreeNode root) {
            if (root == null) {
```

```
                        return 0;
                }

                TreeNode l = root;
                int cntL = 1;
                while (l.left != null) {
                        cntL++;
                        l = l.left;
                }
                TreeNode r = root;
                int cntR = 1;
                while (r.right != null) {
                        cntR++;
                        r = r.right;
                }
                int lmax = findLongest1(root.left);
                int rmax = findLongest1(root.right);
                int max = Math.max(lmax, rmax);
                max = Math.max(max, cntR);
                max = Math.max(max, cntL);
                return max;

        }

        public static int findLongest2(TreeNode root) {
                int[] maxVal = new int[1];
                maxVal[0] = -1;
                helper3(root, maxVal);
                return maxVal[0];
        }

        private static int[] helper3(TreeNode root, int[] maxVal) {
                int[] ret = new int[2];
                if (root == null) {
                        ret[0] = 0;
                        ret[1] = 0;
                        return ret;
                }
                ret[0] = helper3(root.left, maxVal)[0] + 1;
                ret[1] = helper3(root.right, maxVal)[1] + 1;
                maxVal[0] = Math.max(maxVal[0], ret[0]);
                maxVal[0] = Math.max(maxVal[0], ret[1]);
                return ret;
        }

        public static class TreeNode {
                int val;
                TreeNode left;
                TreeNode right;
```

```
		TreeNode(int x) {
			val = x;
		}
	}

	public static TreeNode myTree() {
		TreeNode t1 = new TreeNode(1);
		TreeNode t2 = new TreeNode(2);
		TreeNode t3 = new TreeNode(3);
		TreeNode t4 = new TreeNode(4);
		t1.left = t2;
		t1.right = t3;
		t3.left = t4;
		return t1;
	}

	public static void main(String[] args) {
		System.out.print(getNodeNumRec(myTree()));
		System.out.print("\n");
		System.out.print(getNodeNum(myTree()));
		System.out.print("\n");
		System.out.print(getNodeNumKthLevelRec(myTree(), 2));
		System.out.print("\n");
		System.out.print(getNodeNumKthLevel(myTree(), 2));
		System.out.print("\n");
		System.out.print(getNodeNumLeaf(myTree()));
		System.out.print("\n");
		System.out.print(getMaxDistanceRec(myTree()));
		System.out.print("\n");
		System.out.print(isCompleteBTRec(myTree()));
		System.out.print("\n");
		System.out.print(findLongest1(myTree()));
		System.out.print("\n");
		System.out.print(findLongest2(myTree()));
	}
}
/*
 * outputs:
 * 4
 * 4
 * 2
 * 2
 * 2
 * 3
 * false
 * 2
 * 2
 */
```

```
public class GenerateMirrorSol {
	public static TreeNode mirrorRec(TreeNode root) {
		if (root == null) {
			return null;
		}
		TreeNode tmp = root.right;
		root.right = mirrorRec(root.left);
		root.left = mirrorRec(tmp);
		return root;
	}

	public static TreeNode mirror(TreeNode root) {
		if (root == null) {
			return null;
		}
		Stack<TreeNode> s= new Stack<TreeNode>();
		s.push(root);
		while (!s.isEmpty()) {
			TreeNode cur = s.pop();
			TreeNode tmp = cur.left;
			cur.left = cur.right;
			cur.right = tmp;
			if (cur.right != null) {
				s.push(cur.right);
			}
			if (cur.left != null) {
				s.push(cur.left);
			}
		}
		return root;
	}

	public static TreeNode mirrorCopyRec(TreeNode root) {
		if (root == null) {
			return null;
		}
		TreeNode rootCopy = new TreeNode(root.val);
		rootCopy.left = mirrorCopyRec(root.right);
		rootCopy.right = mirrorCopyRec(root.left);
		return rootCopy;
	}

	public static TreeNode mirrorCopy(TreeNode root) {
		if (root == null) {
			return null;
		}
		Stack<TreeNode> s = new Stack<TreeNode>();
		Stack<TreeNode> sCopy = new Stack<TreeNode>();
		s.push(root);
		TreeNode rootCopy = new TreeNode(root.val);
		sCopy.push(rootCopy);
		while (!s.isEmpty()) {
			TreeNode cur = s.pop();
			TreeNode curCopy = sCopy.pop();
			if (cur.right != null) {
				TreeNode leftCopy = new TreeNode(cur.right.val);
				curCopy.left = leftCopy;
				s.push(cur.right);
				sCopy.push(curCopy.left);
			}
			if (cur.left != null) {
				TreeNode rightCopy = new TreeNode(cur.left.val);
				curCopy.right= rightCopy;
				s.push(cur.left);
				sCopy.push(curCopy.right);
			}
		}
		return rootCopy;
	}
}

public class LCABstSol {
	public static TreeNode LCABstRec(TreeNode root, TreeNode node1, TreeNode node2) {
		if (root == null || node1 == null || node2 == null) {
			return null;
		}
		if (root == node1 || root == node2) {
			return root;
		}
		int min = Math.min(node1.val, node2.val);
		int max = Math.max(node1.val, node2.val);
		if (root.val > max) {
			return LCABstRec(root.left, node1, node2);
		} else if (root.val < min) {
			return LCABstRec(root.right, node1, node2);
		}
		return root;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}
	public static void main(String[] args) {
		TreeNode t1 = new TreeNode(1);
		TreeNode t2 = new TreeNode(2);
		TreeNode t3 = new TreeNode(3);
		t2.left = t1;
		t2.right = t3;
		System.out.print(LCABstRec(t2, t1, t3).val);
	}
}

/*
 * ret[0] left pointer
 * ret[1] right pointer
 */
public class BST2LLSol {
	// recursion
	public static TreeNode convertBST2LL1(TreeNode root) {
		return helper(root)[0];
	}
	public static TreeNode[] helper(TreeNode root) {
		TreeNode[] ret = new TreeNode[2];
		ret[0] = null;
		ret[1] = null;
		if (root == null) {
			return ret;
		}
		if (root.left != null) {
			TreeNode[] left = helper(root.left);
			left[1].right = root;
			root.left = left[1];
			ret[0] = left[0];
		} else {
			ret[0] = root;
		}
		if (root.right != null) {
			TreeNode[] right = helper(root.right);
			right[0].left = root;
			root.right = right[0];
			ret[1] = right[1];
		} else {
			ret[1] = root;
		}
		return ret;
	}
	// iteration inorder
	public static TreeNode convertBST2LL2(TreeNode root) {
		if (root == null) {
			return null;
		}
		TreeNode pre = null;
		Stack<TreeNode> s = new Stack<TreeNode>();
		TreeNode cur = root;
		TreeNode head = null;
		while (true) {
			while (cur != null) {
				s.push(cur);
				cur = cur.left;
			}
			if (s.isEmpty()) {
				break;
			}
			cur = s.pop();
			if (head == null) {
				head = cur;
			}
			cur.left = pre;
			if (pre != null) {
				pre.right = cur;
			}
			cur = cur.right;
			pre = cur;
		}
		return root;
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}
	public static TreeNode myTree() {
		TreeNode t1 = new TreeNode(1);
		TreeNode t2 = new TreeNode(2);
		TreeNode t3 = new TreeNode(3);
		t2.left = t1;
		t2.right = t3;
		return t2;
	}

	public static void main(String[] args) {
		System.out.print(convertBST2LL1(myTree()));
		System.out.print("\n");
		//System.out.print(convertBST2LL2(myTree()));
	}
}
```