Answer of 1

The expected running time of BSTSort is O(nlogn). The running time when unbalanced is $O(n^2)$. This is due to insertion on worst case (sorted sequence) taking O(n) for each element.

The BSTSort is faster than SelectionSort, BubbleSort and InsertionSort as they all have $O(n^2)$ running time. BSTSort is slower than MergeSort and MergeSortPlus as they operate on O(nlogn) running time. BSTSort is $\Omega(nlogn)$ while MergeSort and MergeSortPlus are O(nlogn).

```
package sortroutines;

import runtime.Sorter;

import java.util.ArrayList;
import java.util.List;

public class BSTSort extends Sorter  {


    @Override
    public int[] sort(int[] arr) {
        // initially insert to node
        for(var x: arr){
            insert(x);
        }

        ArrayList<Integer> list = new ArrayList<>();
        printTree(list);

        return list.stream()
                .mapToInt(Integer::intValue)  // Converts
Integer to int
                .toArray();
    }

    /** The tree root. */
    private Node root;

    // start with an empty tree
    public BSTSort() {
        root = null;
```

```
    }

    /**
     * Prints the values in the nodes of the tree in sorted
order.
     */
    public void printTree(List<Integer> list) {
        if (root == null)
            return;
        else
            printTree(root,list);
    }

    private void printTree(Node t, List<Integer> list) {
        if (t != null) {
            printTree(t.left, list);

            list.add(t.element);

            printTree(t.right, list);
        }
    }

    public void printTree() {
        if (root == null)
            System.out.println("[]");
        else
            printTree(root);
    }

    private void printTree(Node t) {
        if (t != null) {
            printTree(t.left);
            System.out.println(t.element);
            printTree(t.right);
        }
    }

    // //// find methods
    public boolean find(Integer x) {
        if (x == null)
            return false;
        return find(x, root);
    }

    private boolean find(Integer x, Node n) {
        if (n == null)
```

```
            return false;
        if (n != null && n.element.equals(x))
            return true;
        return (x.compareTo(n.element) < 0) ? find(x, n.left)
                                            : find(x, n.right);
    }

    // returns null if root is null
    public Integer findMax() {
        return findMax(root).element;

    }

    // node will be stored in position 0; parent will be stored
in position 1
    private Node[] findNodeWithParent(Integer x) {
        if (x == null)
            return null;
        return findNodeWithParent(x, root, null);
    }

    private Node[] findNodeWithParent(Integer x, Node n, Node
parent) {
        if (n == null)
            return null;
        Node[] nodes = new Node[2];
        if (n != null && n.element.equals(x)) {
            nodes[0] = n;
            nodes[1] = parent;
            return nodes;
        }
        if (x.compareTo(n.element) < 0) {
            return findNodeWithParent(x, n.left, n);
        } else {
            return findNodeWithParent(x, n.right, n);
        }
    }

    // returns the Node with max value in the tree determined by
Node node
    private Node findMax(Node node) {
        Node n = node;
        while (n != null) {
            if (n.right == null) {
                return n;
            } else {
                n = n.right;
```

```
            }
        }
        return null;
    }

    // returns null if root is null
    public Integer findMin() {
        return findMin(root).element;

    }

    private Node findMin(Node node) {
        Node n = node;
        while (n != null) {
            if (n.left == null) {
                return n;
            } else {
                n = n.left;
            }
        }
        return null;

    }

    // /////////// delete methods

    public boolean delete(Integer x) {
        Node[] toDeleteAndParent = findNodeWithParent(x);
        if (toDeleteAndParent != null) {
            Node node = toDeleteAndParent[0];
            Node parent = toDeleteAndParent[1];

            //node to delete has two children
            if (node.left != null && node.right != null) {
                return deleteNodeTwoChildren(node, parent);

                //node to delete is a leaf node
            } else if (node.left == null && node.right == null)
{
                return deleteLeaf(node, parent);

                //node to delete has just one child
            } else { // exactly one of these is not null
                return deleteNodeOneChild(node, parent);
            }
        }
        return false;
```

```
    }

    private boolean deleteNodeOneChild(Node n, Node parent) {
        Node child = (n.right == null) ? n.left : n.right;
        if (parent == null) { // root is node to be deleted; it
has one child;
            // this child now becomes the root
            root = child;
        } else {
            if (parent.right == n)
                parent.right = child;
            else if (parent.left == n)
                parent.left = child;
            else {
                throw new RuntimeException(
                        "Unable to locate node to be deleted in
relation to its parent");
            }
            n = null;
        }
        return true;
    }

    private boolean deleteNodeTwoChildren(Node n, Node parent) {
        Node rightChild = n.right;
        Node minBelowRight = findMin(rightChild);
        Integer minBelowRightElement = minBelowRight.element;
        delete(minBelowRight.element);
        n.element = minBelowRightElement;
        return true;
    }

    private boolean deleteLeaf(Node n, Node parent) {
        if (parent != null) { // node is root in that case
            if (parent.left == n) {
                parent.left = null;
            } else if (parent.right == n) {
                parent.right = null;
            }
            n = null;
        } else { // Node n is the root; make tree empty
            root = null;
        }
        return true;
    }

    public boolean isLeaf(Integer x) {
```

```
    Node n = findNodeWithParent(x)[0];
    return isLeafNode(n);
}

private boolean isLeafNode(Node n) {
    if (n == null)
        return false;
    return n.left == null && n.right == null;
}

// /////insertion methods

public void insert(Integer x) {
    if (root == null) {
        root = new Node(x, null, null);
    } else {
        Node n = root;
        boolean inserted = false;
        while (!inserted) {
            if (x.compareTo(n.element) < 0) {
                // space found on the left
                if (n.left == null) {
                    n.left = new Node(x, null, null);
                    inserted = true;
                } else {
                    n = n.left;
                }
            }

            else if (x.compareTo(n.element) > 0) {
                // space found on the right
                if (n.right == null) {
                    n.right = new Node(x, null, null);
                    inserted = true;
                } else {
                    n = n.right;
                }
            } else {
                inserted = true;
            }

        }

    }
}

// ///////// testing
```

```
    public static void main(String[] args) {
        BSTSort bst = new BSTSort();
        for (int i = 15; i >= 0; --i) {
            bst.insert(new Integer(2 * i));
            bst.insert(new Integer(2 * i - 5));
        }
        bst.printTree();
        System.out.println("Is 24 in the tree? " + bst.find(new
Integer(24)));
        System.out.println("Is 27 in the tree? " + bst.find(new
Integer(27)));

        System.out.println("Min: " + bst.findMin());
        System.out.println("Is -5 a leaf? " + bst.isLeaf(-5));
        bst.delete(-5);
        bst.printTree();
        // bst2
        BSTSort bst2 = new BSTSort();
        System.out.println("\n\nNew tree:\n");
        populate(bst2);
        bst2.printTree();
        // delete a leaf
        bst2.delete(150);
        System.out.println("\nAfter deleting 150...\n");
        bst2.printTree();
        // delete node with one child
        bst2.delete(75);
        System.out.println("\nAfter deleting 75...\n");
        bst2.printTree();
        // delete node with two children
        bst2.delete(37);
        System.out.println("\nAfter deleting 37...\n");
        bst2.printTree();

    }

    private static void populate(BSTSort tree) {
        tree.insert(50);
        tree.insert(25);
        tree.insert(75);
        tree.insert(12);
        tree.insert(37);
        tree.insert(28);
        tree.insert(100);
        tree.insert(150);
        tree.insert(48);
```

```
        tree.insert(45);
        tree.insert(43);
    }

    // //////// Node class

    private class Node {

        /////////////// Constructors

        @SuppressWarnings("unused")
        Node(Integer theElement) {
            this(theElement, null, null);
        }

        Node(Integer element, Node left, Node right) {
            this.element = element;
            this.left = left;
            this.right = right;
        }

        private Integer element; // The data in the node
        private Node left; // Left child
        private Node right; // Right child
    }
}
```

Answer of 2

```
class Solution {
    public boolean isBalanced(TreeNode root) {
        return height(root) != -1;
    }

    public int height(TreeNode node){
        if(node == null){
            return 0;
        }

        int leftHeight = height(node.left);
        int rightHeight = height(node.right);

        if(leftHeight == -1 || rightHeight == -1 ||
Math.abs(leftHeight - rightHeight) > 1){
            return -1;
        }
```

```
        return Math.max(leftHeight, rightHeight) + 1;
    }
}
```

Answer of 3

```
class Solution {
    public int goodNodes(TreeNode root) {
        return goodNodes(root, Integer.MIN_VALUE);
    }

    public int goodNodes(TreeNode root, int maximum){
        if(root == null){
            return 0;
        }

        int good = root.val >= maximum? 1:0;
        maximum = Math.max(maximum, root.val);

        return good + goodNodes(root.left, maximum) +
goodNodes(root.right, maximum);
    }
}
```

Answer of 4

```
class Solution {
    public TreeNode trimBST(TreeNode root, int low, int high) {
        if (root == null) {
            return root;
        }

        root.left = trimBST(root.left, low, high);
        root.right = trimBST(root.right, low, high);

        if (root.val < low) {
            return root.right;
        } else if (root.val > high) {
            return root.left;
```

```
        } else {
            return root;
        }
    }
}
```