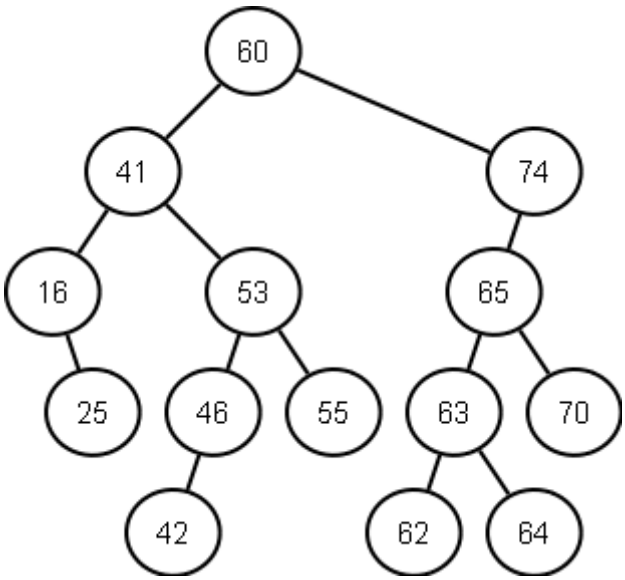
Binary Search Trees

Search trees with only one element per node and at most two children are a heckuva lot simpler to understand and work with that they've become probably the most popular search tree there is. And they have some cool operations that can be used to make searching even better.

CONTENTS

Definition

A **binary search tree** is a binary tree in which every node holds a value >= every value in its left subtree and <= every value in its right subtree.

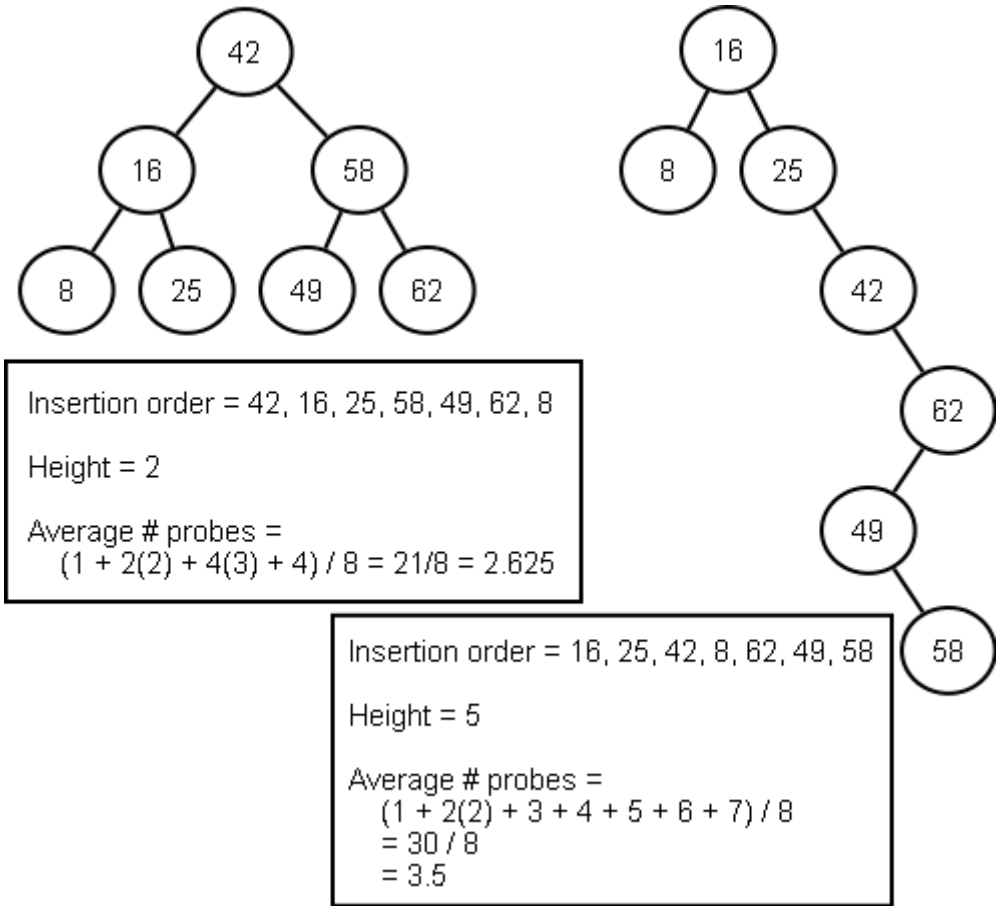


Overview

BSTs can be used for

- **Sorting**: Just traverse inorder!
- **Searching** (implementing sets and dictionaries): while searching in a linked list is Θ(n), searching a randomly generated BSTs is, on average, Θ(log n).

However, not every BST gives logarithmic search performance:



To deal with the problem of "degenerate" (tall and thin) trees, we define rotation operations.

How They Work

Lookup

To find a value V into tree T,

- If the tree T is empty, the lookup failed. If V is at the root, you found it. If V < T.root, look for V in T.left, otherwise look in T.right.

Insertion

To insert a value V into tree T,

- If the tree T is empty, give it a new root node with V, otherwise if V < T.root, insert V into T.left, otherwise insert V into T.right.

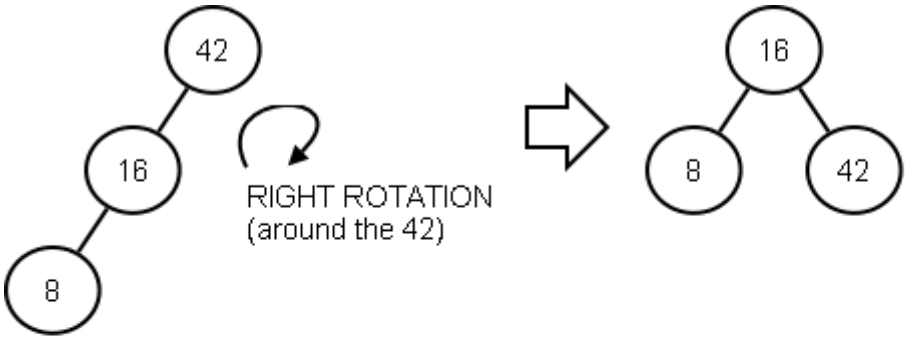
Deletion

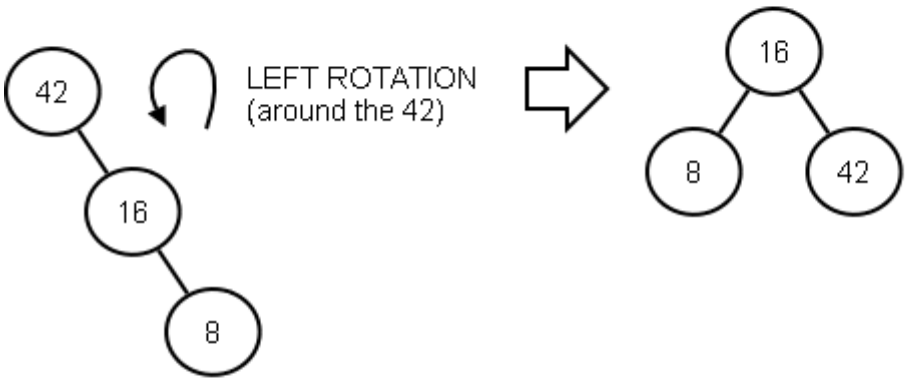
To delete the value V from tree T,

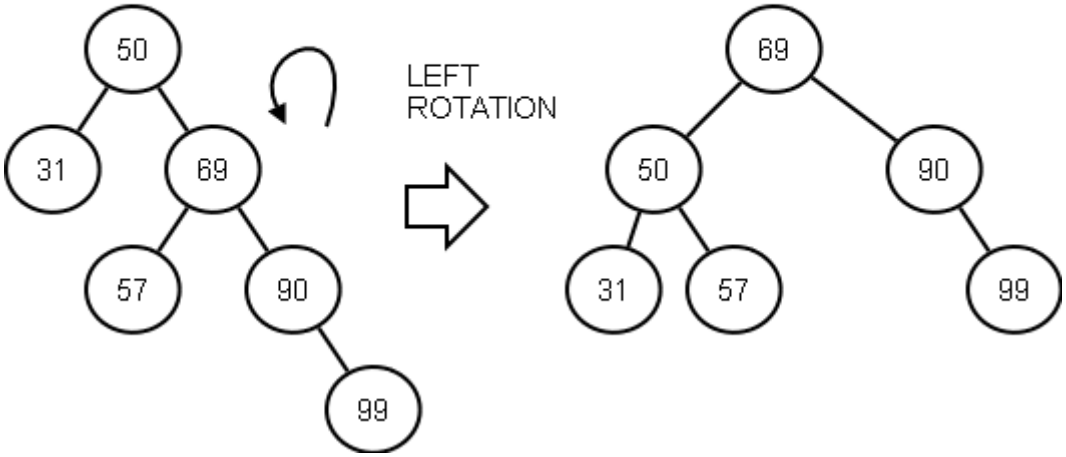
- Let *d* be the node to be deleted (the one containing V). If *d* has an empty subtree then point the link to d to the other subtree. Otherwise *r* be the rightmost node in *d*'s left subtree. Replace the contents of *d* with the contents of *r* then delete *r*.

Rotation

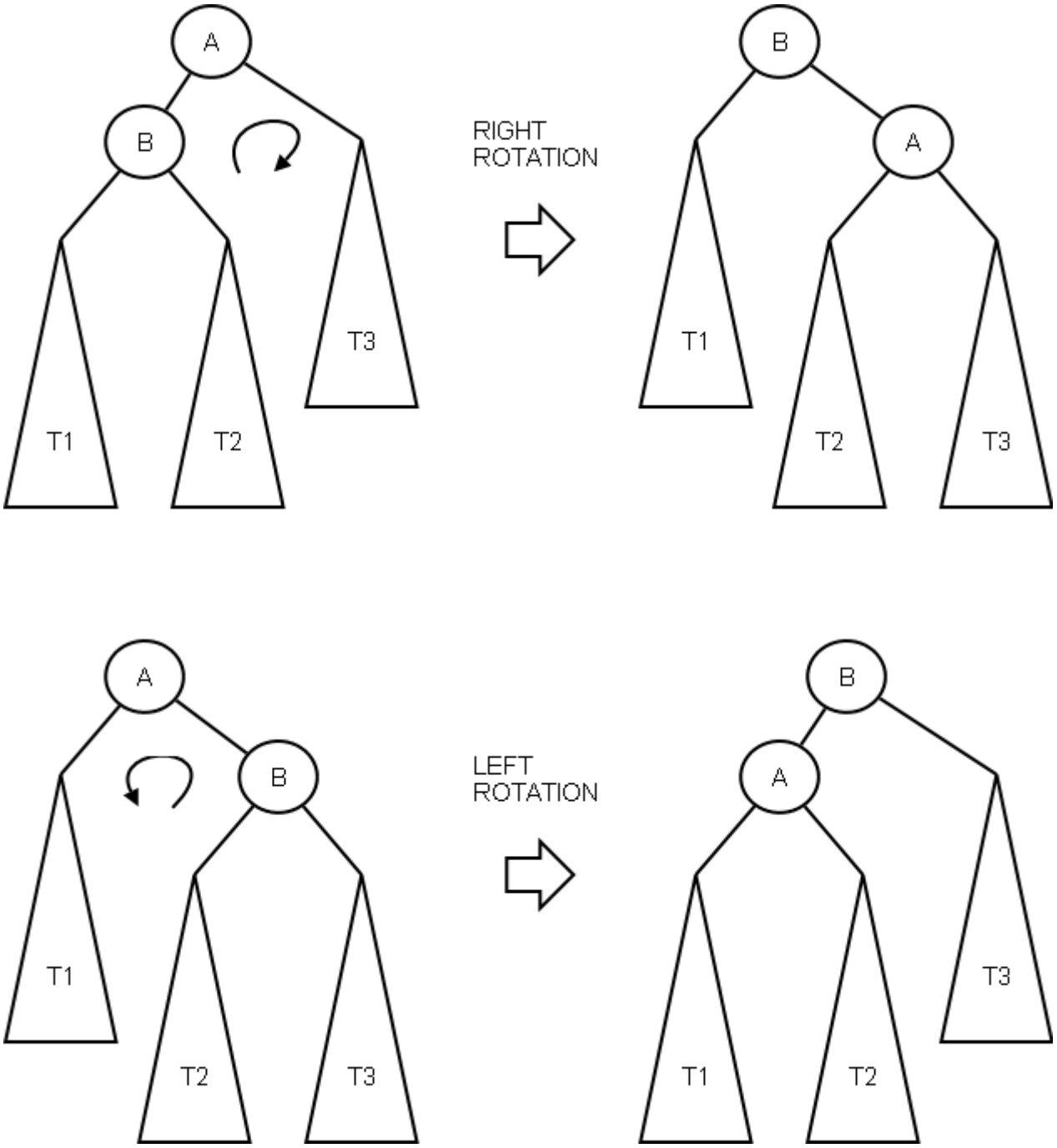
Rotations usually make a tree shorter


42
16
8
RIGHT ROTATION
(around the 42)
16
8
42


42
16
8
LEFT ROTATION
(around the 42)
16
8
42


50
31
69
57
90
99
LEFT
ROTATION
69
50
90
31
57
99

In general



There are many different strategies for applying rotations. Generally you'll want to rotate after any insertion or deletion that causes a tree to get "too tall" — **AVL trees** and **Red-Black trees** do this. Another approach is to constantly rotate after every insertion, deletion, and lookup to so as to bubble the most likely to be looked up elements near the top. **Splay trees** do this.

An Implementation

We don't need an interface for Binary Search Trees, because they are just wrappers around objects of the BinaryTreeNode interface we made before. Note that we need a comparator, though, since BSTs are inherently ordered things.

BinarySearchTree.java

```
import java.util.Comparator;

/**
 * A binary search tree class with insertion, removal and lookup. A comparator
 * is used to order the items in the tree. All tree items must be distinct
 * according to the comparator. If no comparator is supplied the natural order
 * of tree elements is used.
 */
public class BinarySearchTree<E> {

    /**
     * Root of the tree.
     */
    protected BinaryTreeNode<E> root = null;

    /**
     * Comparator used to order the items in the tree. If null, the natural order of
     * the items will be used.
     */
    private Comparator<E> comparator;

    /**
     * Constructs an empty BST that can only accept Comparables as items.
     */
    public BinarySearchTree() {
        this(null);
    }

    /**
     * Constructs a BST that orders its items according to the given comparator.
     */
    public BinarySearchTree(Comparator<E> c) {
        comparator = c;
    }

    /**
     * Returns whether or not the tree contains an object with the given value.
     */
    public boolean contains(E data) {
        return nodeContaining(data) != null;
    }

    /**
     * Adds a single data item to the tree. If there is already an item in the tree
     * that compares equal to the item being inserted, it is "overwritten" by the
     * new item.
     */
    public void add(E data) {
        if (root == null) {
            root = new LinkedBinaryTreeNode<E>(data);
        }
        BinaryTreeNode<E> n = root;
        while (true) {
            int comparisonResult = compare(data, n.getData());
            if (comparisonResult == 0) {
                n.setData(data);
```

```
                return;
            } else if (comparisonResult < 0) {
                if (n.getLeft() == null) {
                    n.setLeft(new LinkedBinaryTreeNode<E>(data));
                    return;
                }
                n = n.getLeft();
            } else { // comparisonResult > 0
                if (n.getRight() == null) {
                    n.setRight(new LinkedBinaryTreeNode<E>(data));
                    return;
                }
                n = n.getRight();
            }
        }
    }

    /**
     * Removes the node containing the given value. Does nothing if there is no such
     * node.
     */
    public void remove(E data) {
        BinaryTreeNode<E> node = nodeContaining(data);
        if (node == null) {
            // No such object, do nothing.
            return;
        } else if (node.getLeft() != null && node.getRight() != null) {
            // Node has two children, we cannot delete it. Copy
            // predecessor data here and get ready to delete predecessor.
            BinaryTreeNode<E> predecessor = predecessor(node);
            node.setData(predecessor.getData());
            node = predecessor;
        }
        // At this point node has zero or one child
        BinaryTreeNode<E> pullUp = (node.getLeft() == null) ? node.getRight() : node.getLeft();
        if (node == root) {
            setRoot(pullUp);
        } else if (node.getParent().getLeft() == node) {
            node.getParent().setLeft(pullUp);
        } else {
            node.getParent().setRight(pullUp);
        }
    }

    // Best to put the comparison code in a single place so that we don't have
    // to check for comparators and cast all over the place.

    protected int compare(E x, E y) {
        if (comparator == null) {
            return ((Comparable<E>) x).compareTo(y);
        } else {
            return comparator.compare(x, y);
        }
    }

    // Methods relating to nodes, not part of public interface.

    /**
     * Returns the root of the tree.
     */
    protected BinaryTreeNode getRoot() {
        return root;
    }

    /**
```

```
     * Makes the given node the new root of the tree.
     */
    protected void setRoot(BinaryTreeNode<E> node) {
        if (node != null) {
            node.removeFromParent();
        }
        root = node;
    }

    /**
     * Rotates left around the given node.
     */
    protected void rotateLeft(BinaryTreeNode<E> n) {
        if (n.getRight() == null) {
            return;
        }
        BinaryTreeNode<E> oldRight = n.getRight();
        n.setRight(oldRight.getLeft());
        if (n.getParent() == null) {
            root = oldRight;
        } else if (n.getParent().getLeft() == n) {
            n.getParent().setLeft(oldRight);
        } else {
            n.getParent().setRight(oldRight);
        }
        oldRight.setLeft(n);
    }

    /**
     * Rotates right around the given node.
     */
    protected void rotateRight(BinaryTreeNode<E> n) {
        if (n.getLeft() == null) {
            return;
        }
        BinaryTreeNode<E> oldLeft = n.getLeft();
        n.setLeft(oldLeft.getRight());
        if (n.getParent() == null) {
            root = oldLeft;
        } else if (n.getParent().getLeft() == n) {
            n.getParent().setLeft(oldLeft);
        } else {
            n.getParent().setRight(oldLeft);
        }
        oldLeft.setRight(n);
    }

    /**
     * Returns the rightmost node in the left subtree.
     */
    protected BinaryTreeNode<E> predecessor(BinaryTreeNode<E> node) {
        BinaryTreeNode<E> n = node.getLeft();
        if (n != null) {
            while (n.getRight() != null) {
                n = n.getRight();
            }
        }
        return n;
    }

    /**
     * A special helper method that returns the node containing an object that
     * compares equal to the given object. This is used in both contains and remove.
     */
    protected BinaryTreeNode<E> nodeContaining(E data) {
```

```
            for (BinaryTreeNode<E> n = root; n != null;) {
                int comparisonResult = compare(data, n.getData());
                if (comparisonResult == 0) {
                    return n;
                } else if (comparisonResult < 0) {
                    n = n.getLeft();
                } else {
                    n = n.getRight();
                }
            }
            return null;
        }
    }
```

***Exercise**: In this implementation, I've hardcoded the fact that these trees are made up of LinkedBinaryTreeNodes. This is actually a Bad Thing. Rewrite this so that clients can create nodes of any class implementing BinaryTreeNode they desire.*

A Viewer

For fun, I made my own viewer application. No animation, though:

BinarySearchTreeViewer.java

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.border.BevelBorder;

/**
 * A little application that lets you interactively manipulate a binary search
 * tree.
 */
public class BinarySearchTreeViewer extends JFrame {
    BinarySearchTree<String> tree = new BinarySearchTree<String>();

    JFrame frame = new JFrame("Binary Search Tree");
    JTextField valueField = new JTextField(40);
    JPanel buttonPanel = new JPanel();
    BinaryTreePanel panel = new BinaryTreePanel(null, 40, 40);
    JScrollPane displayArea = new JScrollPane();
    JLabel messageLine = new JLabel();

    /**
```

```
     * An operation encapsulates a button and its action. The constructor will
     * create a button, add it to a button panel, and register itself as a listener
     * for the button. The listener first reads inputs from a textfield, then calls
     * a subclass-supplied method with those inputs, then displays the resulting
     * tree in the display area.
     */
    private abstract class Operation implements ActionListener {
        public Operation(String label) {
            JButton button = new JButton(label);
            buttonPanel.add(button);
            button.addActionListener(this);
        }

        public void actionPerformed(ActionEvent event) {
            String value = valueField.getText();
            messageLine.setText("");
            execute(value);
            // Update the picture and return the focus to the text
            // field. Select all the text in the textfield so it
            // can easily be overwritten.
            panel.setTree(tree.getRoot());
            valueField.requestFocus();
            valueField.selectAll();
        }

        protected abstract void execute(String value);
    }

    /**
     * Constructs a viewer, laying out all the components in a very nice way, and
     * constructs and registers all the operation objects.
     */
    public BinarySearchTreeViewer() {
        JPanel valuePanel = new JPanel();
        valuePanel.add(new JLabel("Value: "));
        valuePanel.add(valueField);

        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new GridLayout(0, 1));
        controlPanel.add(valuePanel);
        controlPanel.add(buttonPanel);

        // NOTE: Hardcoded preferred size! Fix this in the exercises.
        panel.setPreferredSize(new Dimension(2048, 2048));
        panel.setBackground(Color.white);
        panel.setBorder(new BevelBorder(BevelBorder.LOWERED));
        displayArea.setViewportView(panel);

        frame.setBackground(Color.lightGray);
        frame.getContentPane().add(controlPanel, "North");
        frame.getContentPane().add(displayArea, "Center");
        frame.getContentPane().add(messageLine, "South");
        frame.pack();

        new Operation("Add") {
            protected void execute(String value) {
                tree.add(value);
            }
        };
        new Operation("Add All") {
            protected void execute(String value) {
                for (String s : value.split("\\s+"))
                    tree.add(s);
            }
        };
```

```
            new Operation("Lookup") {
                protected void execute(String value) {
                    messageLine.setText("The value \"" + value + "\" is " + (tree.contains(value) ? "" : "not ") +
                }
            };
            new Operation("Remove") {
                protected void execute(String value) {
                    tree.remove(value);
                }
            };
            new Operation("Rotate Left") {
                protected void execute(String value) {
                    BinaryTreeNode<String> n = tree.nodeContaining(value);
                    if (n != null && n.getRight() != null)
                        tree.rotateLeft(n);
                }
            };
            new Operation("Rotate Right") {
                protected void execute(String value) {
                    BinaryTreeNode<String> n = tree.nodeContaining(value);
                    if (n != null && n.getLeft() != null)
                        tree.rotateRight(n);
                }
            };
        }

        /**
         * Makes an application whose main window is a BinarySearchTreeViewer.
         */
        public static void main(String[] args) {
            BinarySearchTreeViewer viewer = new BinarySearchTreeViewer();
            viewer.frame.setSize(540, 480);
            viewer.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            viewer.frame.setVisible(true);
        }
    }
```