```java
package hwk4;

/**
 * Binary Search Tree Abstract Data Type (ADT)
 *
 * @author Neil Daterao
 * @version 2/29/2024
 */
public class BinarySearchTree
{
    private BSTNode root;

    public BinarySearchTree() {
      root=null;
    }

    /**
    * inserts recursively.  I include this one so you can
    * make your own trees in the testing class
    *
    * @param subroot inserts into subtree rooted at subroot
    * @param newNode node to insert
    * @return the BST rooted at subroot that has newNode inserted
    */
    private BSTNode recursiveInsert(BSTNode subroot, BSTNode newNode) {
      if (subroot == null) {
        return newNode;
      }
      else if (newNode.data.compareTo(subroot.data) > 0) {
        subroot.rlink = recursiveInsert(subroot.rlink,newNode);
        return subroot;
      }
      else {  // newNode.data smaller than subroot.data, so newNode goes on left
        subroot.llink = recursiveInsert(subroot.
```

```
34 llink,newNode);
35         return subroot;
36       }
37     }
38 
39     /**
40     * inserts recursively. Use this in your JUnit
   tests to
41     * build a starting tree correctly
42     *
43     * @param newString String to insert
44     */
45     public void recursiveInsert(String newString){
46       BSTNode newNode = new BSTNode(newString);
47       root = recursiveInsert(root, newNode);
48     }
49 
50 
51     /**
52     * Private recursive function to build the string
    version of the BST.
53     *
54     * @param subroot subroot of tree to print
55     */
56     private String print(BSTNode subroot)
57     {
58       StringBuilder builder = new StringBuilder();
59 
60       if (subroot != null) {
61           builder.append("(");
62           builder.append(print(subroot.llink));
63           builder.append(" " + subroot + " ");
64           builder.append(print(subroot.rlink));
65           builder.append(")");
66       }
67 
68       return builder.toString();
69     }
70 
71 
72     /**
```

```
 73      *  Prints a parenthesized version of this tree that shows
 74      *  the subtree structure.  Every non-empty subtree is
 75      *  encased in parentheses. Example: (( A ) B ( C )) means
 76      *  B is the parent of A (left kid) and C ( right kid).
 77      * @return String version of Binary Search Tree.
 78      */
 79     public String toString()
 80     {
 81
 82       return print(root);
 83
 84     }
 85
 86     /**
 87      *
 88      * @return number of data items (nodes) in the tree
 89      */
 90     public int size() {
 91         return size(root);
 92     }
 93
 94     /**
 95      * Private recursive method to return size of BST
 96      * @param subRoot
 97      * @return
 98      */
 99     private int size(BSTNode subRoot) {
100       if (subRoot == null) { return 0; }
101       else { return 1 + size(subRoot.llink) + size(subRoot.rlink);  }
102
103     }
104
105
```

```
106     /**
107      *
108      * @param target value to search for
109      * @return true if target is in BST and false
    if not
110      */
111     public boolean search(String target) {
112         return search(target, root);
113     }
114 
115 
116     /**
117      * Private recursive method to search for
    target given a subroot in a BST
118      * @param target
119      * @param subRoot
120      * @return true if target is in BST and false
    if not
121      */
122     private boolean search(String target, BSTNode
    subRoot) {
123         if (subRoot == null) { return false; }
124         else if (subRoot.data.equals(target)) {
    return true; }
125 
126         else if (subRoot.data.compareTo(target) > 0
     ){
127             return search(target, subRoot.llink);
128         }
129         else { return search(target, subRoot.rlink
    ); }
130     }
131 
132 
133 
134     /**
135      * Method that inserts value into a Binary
    Search Tree
136      * @param value String value
137      */
138     public void insert(String value){
```

```
139       BSTNode newNode = new BSTNode(value);
140       if (root == null) { root = newNode;  }
141
142       else {
143
144         BSTNode parent = null;
145         BSTNode runner = root;
146
147         while (runner != null) {
148           parent = runner;
149           if (runner.data.compareTo(value) > 0) {
150             runner = runner.llink;
151           }
152           else {
153             runner = runner.rlink;
154           }
155         }
156
157         if (parent.data.compareTo(value) > 0) {
    parent.llink = newNode;  }
158         else { parent.rlink = newNode; }
159
160       }
161
162     }
163
164
165
166 }
167
```

```java
package test;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import hwk4.*;

/**
 * Testing class for Binary Search Tree Abstract Data Type (ADT)
 *
 * @author Neil Daterao
 * @version 2/29/2024
 */
public class BinarySearchTreeTester {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    private BinarySearchTree bst;
    private BinarySearchTree largeBst;

    @Before
    public void setUp() throws Exception {
        bst = new BinarySearchTree();
    }

    @After
    public void tearDown() throws Exception {
        bst = null;
    }

    private void createLargeBST() {
        largeBst = new BinarySearchTree();
        largeBst.insert("D");
        largeBst.insert("H");
        largeBst.insert("C");
```

```
41          largeBst.insert("J");
42          largeBst.insert("M");
43          largeBst.insert("E");
44          largeBst.insert("A");
45          largeBst.insert("F");
46          largeBst.insert("G");
47          largeBst.insert("K");
48          largeBst.insert("I");
49          largeBst.insert("L");
50
51      }
52
53
54      @Test
55      public void testToString() {
56          bst.recursiveInsert("B");
57          bst.recursiveInsert("A");
58          bst.recursiveInsert("C");
59          assertEquals("should be (( A ) B ( C ))",
    "(( A ) B ( C ))", bst.toString());
60      }
61
62      @Test
63      public void testToStringEmptyTree() {
64          assertEquals("", bst.toString());
65      }
66
67      @Test
68      public void testToStringOneNode() {
69          bst.insert("B");
70          assertEquals("( B )", bst.toString());
71      }
72
73      @Test
74      public void testToStringMultipleNodes() {
75          bst.insert("B");
76          bst.insert("A");
77          bst.insert("C");
78          bst.insert("E");
79          bst.insert("D");
80          assertEquals("(( A ) B ( C (( D ) E ))",
```

```
 80 bst.toString());
 81     }
 82
 83     @Test
 84     public void testSizeEmptyTree() {
 85         assertEquals(0, bst.size());
 86     }
 87
 88     @Test
 89     public void testSizeOneNode() {
 90         bst.insert("B");
 91         assertEquals(1, bst.size());
 92     }
 93
 94     @Test
 95     public void testSizeMultipleNodes() {
 96         bst.insert("B");
 97         bst.insert("A");
 98         bst.insert("C");
 99         assertEquals(3, bst.size());
100
101         bst.insert("E");
102         bst.insert("D");
103         assertEquals(5, bst.size());
104     }
105
106     @Test
107     public void testInsert_M() {
108         bst.insert("M");
109         assertEquals("( M )", bst.toString());
110     }
111
112     @Test
113     public void testInsert_MV() {
114         bst.insert("M");
115         bst.insert("V");
116         assertEquals("( M ( V ))", bst.toString());
117     }
118
119     @Test
120     public void testInsert_MVP() {
```

```
121         bst.insert("M");
122         bst.insert("V");
123         bst.insert("P");
124         assertEquals("( M (( P ) V ))", bst.
    toString());
125     }
126 
127     @Test
128     public void testInsert_MVPDN() {
129         bst.insert("M");
130         bst.insert("V");
131         bst.insert("P");
132         bst.insert("D");
133         bst.insert("N");
134         assertEquals("(( D ) M ((( N ) P ) V ))",
    bst.toString());
135     }
136 
137     @Test
138     public void testInsert_MVPNDF() {
139         bst.insert("M");
140         bst.insert("V");
141         bst.insert("P");
142         bst.insert("D");
143         bst.insert("N");
144         bst.insert("F");
145         assertEquals("(( D ( F )) M ((( N ) P ) V
     ))", bst.toString());
146     }
147 
148     @Test
149     public void testInsert_MVPDNFB() {
150         bst.insert("M");
151         bst.insert("V");
152         bst.insert("P");
153         bst.insert("D");
154         bst.insert("N");
155         bst.insert("F");
156         bst.insert("B");
157         assertEquals("((( B ) D ( F )) M ((( N ) P
     ) V ))", bst.toString());
```

```
158     }
159
160     @Test
161     public void testInsert_MVPDNFBC() {
162         bst.insert("M");
163         bst.insert("V");
164         bst.insert("P");
165         bst.insert("D");
166         bst.insert("N");
167         bst.insert("F");
168         bst.insert("B");
169         bst.insert("C");
170         assertEquals("((( B ( C )) D ( F )) M ((( N
    ) P ) V ))", bst.toString());
171     }
172
173     @Test
174     public void testInsert_MVPDNFBCQR() {
175         bst.insert("M");
176         bst.insert("V");
177         bst.insert("P");
178         bst.insert("D");
179         bst.insert("N");
180         bst.insert("F");
181         bst.insert("B");
182         bst.insert("C");
183         bst.insert("Q");
184         bst.insert("R");
185         assertEquals("((( B ( C )) D ( F )) M ((( N
    ) P ( Q ( R ))) V ))", bst.toString());
186     }
187
188     @Test
189     public void testSearchExistingValue() {
190         createLargeBST();
191         assertTrue("Value 'D' should exist in the
    tree", largeBst.search("D"));
192     }
193
194     @Test
195     public void testSearchNonExistingValue() {
```

```
196         createLargeBST();
197         assertFalse("Value 'X' should not exist in
    the tree", largeBst.search("X"));
198     }
199 
200     @Test
201     public void testSearchRootValue() {
202         createLargeBST();
203         assertTrue("Root value 'H' should exist in
    the tree", largeBst.search("H"));
204     }
205 
206     @Test
207     public void testSearchLeafs() {
208         createLargeBST();
209         assertTrue("Last node value 'A' should
    exist in the tree", largeBst.search("A"));
210         assertTrue("Last node value 'C' should
    exist in the tree", largeBst.search("C"));
211         assertTrue("Last node value 'M' should
    exist in the tree", largeBst.search("M"));
212     }
213 
214     @Test
215     public void testSearchMiddleNodeValue() {
216         createLargeBST();
217         assertTrue("Middle node value 'L' should
    exist in the tree", largeBst.search("L"));
218     }
219 
220     @Test
221     public void testSearchEmptyTree() {
222         assertFalse("Empty Tree", bst.search("A"));
223 
224     }
225 
226 
227 
228 
229 
230 }
```

231
232
233

```java
package hwk4;

/** A not-very-reusable node class, since it only holds a String.
 *  But good enough for this hwk.
 *
 * @author Chris Fernandes
 * @version 2/27/24
 */
public class BSTNode {

    public String data;
    public BSTNode llink;
    public BSTNode rlink;

    /**
     * non-default constructor
     * @param newKey string that node will hold
     */
    public BSTNode(String newKey)
    {
        data = newKey;
        llink = null;
        rlink = null;
    }

    /**
     * returns key as printable string
     */
    public String toString()
    {
        return data;
    }
}

```