```
package hwk4;

/** This is the BST ADT.  It should contain methods that allow it to
 *  insert new nodes, delete nodes, search, etc.  You’ll be adding
 *  code to this class for this hwk.
 *
 * @author Kaito
 * @version May 29th 2025, Spring 2025
 */
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.llink,newNode);
                 return subroot;
         }
     }

     /**
      * inserts recursively. Use this in your JUnit tests to
      * build a starting tree correctly
      *
      * @param newString String to insert
      */
     public void recursiveInsert(String newString){
         BSTNode newNode = new BSTNode(newString);
         root = recursiveInsert(root, newNode);
     }
```

```
/**
 * WARNING: CRAPPY METHOD!  I wish I had toString...
 *
 * prints a parenthesized version of the tree rooted
     * at subroot using an inorder tree traversal algorithm.
     *
 * @param subroot subroot of tree to print
 */
private void print(BSTNode subroot)
{
    if (subroot != null) {     // nothing to print if subtree empty
      System.out.print("(");
      print(subroot.llink);
      System.out.print("  " + subroot + "  ");
      print(subroot.rlink);
      System.out.print(")");
    }
}


/**
 *  WARNING: CRAPPY METHOD!  I wish I had toString...
 *
 *  prints a parenthesized version of this tree that shows
 *  the subtree structure.  Every non-empty subtree is
     *  encased in parentheses. Example: (( A ) B ( C )) means
 *  B is the parent of A (left kid) and C (right kid).
 */
public void print()
{
    print(root);
    System.out.println();
}

/**
 * Inserts value into the tree.
 * The iterative version of insert.
 *
 * @param newString the string to insert
 */
 public void insert(String newString)
 {
     BSTNode newNode = new BSTNode(newString);
     if (root == null)
     {
         root = newNode;
         return;
     }
```

```
        BSTNode current = root;
        BSTNode parent = null;

        while (current != null)
        {
            parent = current;
            if (newString.compareTo(current.data) < 0)
            {
                current = current.llink;
            }
            else
            {
                current = current.rlink;
            }
        }

        if (newString.compareTo(parent.data) < 0)
        {
            parent.llink = newNode;
        }
        else
        {
            parent.rlink = newNode;
        }
    }

    /**
     * Searches for a value in the tree in recursive.
     * * @param key the string to search for
     * @return true if the value is found, false otherwise
     */
    public boolean search(String target)
    {
        return searchRecursive(root, target);
    }
    private boolean searchRecursive(BSTNode subroot, String target)
    {
        if (subroot == null)
        {
            return false;
        }
        if (target.equals(subroot.data))
        {
            return true;
        }
        else if (target.compareTo(subroot.data) < 0)
        {
            return searchRecursive(subroot.llink, target); // Search left subtree
        }
        else
```

```
          {
              return searchRecursive(subroot.rlink, target); // Search right subtree
          }
      }

      /**
       * return a string representation of the tree using inorder traversal.
       * @return a string representation of the tree
       *
       */
      public String toString()
      {
          return toStringRecursive(root);
      }
      private String toStringRecursive(BSTNode subroot)
      {
          if (subroot == null)
          {
          return "";
          }
          String ans = "";
          ans += ’(’;
          ans += toStringRecursive(subroot.llink);
          ans += " " + subroot.data + " ";
          ans += toStringRecursive(subroot.rlink);
          ans += ’)’;
          return ans;
      }

      /**
       * Return the number of nodes in the tree.
       * @return the number of nodes in the tree
       */
      public int size()
      {
          return sizeRecursive(root);
      }
      private int sizeRecursive(BSTNode subroot)
      {
          if (subroot == null)
          {
              return 0;
          }
          return 1 + sizeRecursive(subroot.llink) + sizeRecursive(subroot.rlink);
      }
}
```

```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;
        }
}
```

```
package hwk4;

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;

/**
 *
 * This class tests the Divide class.
 *
 * @author Kaito Nguyen
 * @version May 29th 2025, Spring 2025
 */

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

    BinarySearchTree tree;
    @Before
    public void setUp() throws Exception
    {
        tree = new BinarySearchTree();
        tree.recursiveInsert("D");
        tree.recursiveInsert("B");
        tree.recursiveInsert("F");
        tree.recursiveInsert("A");
        tree.recursiveInsert("C");
        tree.recursiveInsert("E");
        tree.recursiveInsert("G");
    }
    @After
    public void tearDown() throws Exception
    {
        tree = null;
    }

    @Test
    public void testrecursiveInsert()
    {
        // Test inserting a new node
        tree.recursiveInsert("H");
        assertTrue(tree.search("H"));
    }

    @Test
    public void testSearch()
    {
        // Test searching for existing nodes
        assertTrue(tree.search("A"));
        assertTrue(tree.search("B"));
        assertTrue(tree.search("C"));
        assertTrue(tree.search("D"));
        assertTrue(tree.search("E"));
        assertTrue(tree.search("F"));
        assertTrue(tree.search("G"));

        // Test searching for a non-existing node
        assertFalse(tree.search("X"));
    }

    @Test
    public void testPrint()
    {
        // Test printing the tree structure
        String expectedOutput = "((A)B(C))D((E)F(G(H)))";
        assertEquals(expectedOutput, tree.print());
    }

    @Test
    public void testInsert()
    {
        // Test inserting a new node
        tree.insert("I");
        assertTrue(tree.search("I"));
    }

    @Test
    public void testToString()
    {
        // Test the toString method
        String expectedOutput = "((A)B(C))D((E)F(G(H(I))))";
        assertEquals(expectedOutput, tree.toString());
    }

    @Test
    public void testSize()
    {
        assertEquals(9, tree.size());

        // Test size after inserting a new node
        tree.insert("Z");
        assertEquals(10, tree.size());
    }
}
```