```
package proj5;
/**
 * Thesaurus class allows adding synonyms and checking if a synonym exists in the thesaurus
.
 *
 * @author Kaito Nguyen
 * @version May 31st 2025, Spring 2025
 */

public class Thesaurus {
    private BinarySearchTree<ThesaurusNode> synonyms; // Binary search tree to store synony
ms

    /**
     * Default constructor that initializes an empty thesaurus.
     */
    public Thesaurus() {
        synonyms = new BinarySearchTree<>();
    }

    /**
     * Non default constructor
     */
    public Thesaurus (String fileName) {
        synonyms = new BinarySearchTree<>();
        loadThesaurus(fileName);
    }

    /**
     * Inserts a synonym into the thesaurus.
     *
     * @param synonym the synonym to insert
     */
    public void insert(String key, String[] synonym)  {
        if (synonym == null || synonym.length == 0 || key == null || key.isEmpty()) {
            return;
        }
        ThesaurusNode node = new ThesaurusNode(key, synonym);
        BSTNode<ThesaurusNode> existingNode = synonyms.search(node);
        if (existingNode != null) {
            String[] existingSynonyms = existingNode.element.getSynonyms();
            String[] newSynonyms = new String[existingSynonyms.length + synonym.length];
            for (int i = 0; i < existingSynonyms.length; i++) {
                newSynonyms[i] = existingSynonyms[i];
            }
            for (int i = 0; i < synonym.length; i++) {
                newSynonyms[existingSynonyms.length + i] = synonym[i];
            }
            ThesaurusNode newNode = new ThesaurusNode(key, newSynonyms);
            synonyms.insert(newNode);
        }
        else {
            synonyms.insert(node);
        }
    }

    /**
     * Gets the synonyms for a given word.
     * @param word the word to find synonyms for
     * @return an array of synonyms for the word, or null if no synonyms are found
     */
    private String[] getSynonyms(String word) {
        ThesaurusNode getNode = new ThesaurusNode(word, null);
        BSTNode<ThesaurusNode> node = synonyms.search(getNode);
        if (node != null) {
            return node.element.getSynonyms();
        }
        return null;
    }
```

```java
/**
 * Gets random synonym for a given word.
 * @param word the word to find a random synonym for
 * @return a random synonym for the word, or null if no synonyms are found
 */
public String getSynonymFor(String word) {
    if (word == null || word.isEmpty()) {
        return "";
    }

    String[] synonymsArray = getSynonyms(word);
    if (synonymsArray != null && synonymsArray.length > 0) {
        int randomIndex = (int) (Math.random() * synonymsArray.length);
        return synonymsArray[randomIndex];
    }
    return "";
}

/**
 * Deletes a synonym from the thesaurus.
 * @param key the key of the synonym to delete
 */
public void delete(String key) {
    if (key == null || key.isEmpty()) {
        return;
    }
    ThesaurusNode getNode = new ThesaurusNode(key, null);
    synonyms.delete(getNode);
}

/**
 * toString method to represent the thesaurus.
 * @return a string representation of the thesaurus
 */
public String toString() {
    String ans = "";
    for (String key : synonyms.inOrderTraversal()) {
        ThesaurusNode getNode = new ThesaurusNode(key, null);
        String[] values = synonyms.search(getNode).element.getSynonyms();
        ans += key + " - {";
        for (int i = 0; i < values.length; i++) {
            ans += values[i];
            if (i < values.length - 1) {
                ans += ", ";
            }
        }
        ans += "}\n";
    }
    return ans;
}

/**
 * Checks if the thesaurus is empty.
 * @return true if the thesaurus is empty, false otherwise
 */
public boolean isEmpty() {
    return synonyms.isEmpty();
}

/**
 * Load the thesaurus from a file.
 * @param filePath the path to the thesaurus file
 * @return true if the thesaurus was loaded successfully, false otherwise
 */
public boolean loadThesaurus(String filePath) {
    if (filePath == null || filePath.isEmpty()) {
        throw new IllegalArgumentException("File path cannot be null or empty");
    }
```

```
        LineReader lineReader = new LineReader(filePath, ",");

        String[] words;
        while ((words = lineReader.getNextLine()) != null) {
            if (words.length > 1) {
                String key = words[0];
                String[] synonymsArray = new String[words.length - 1];
                for (int i = 1; i < words.length; i++) {
                    synonymsArray[i - 1] = words[i];
                }
                insert(key, synonymsArray);
            }
        }
        lineReader.close();
        return true;
    }

    /**
     * Return size of the Thesaurus
     * @return int
     */
    public int size() {
        return synonyms.size();
    }
}
```

```java
package proj5;

import org.junit.jupiter.api.Test;

import java.util.Arrays;

import static org.junit.jupiter.api.Assertions.*;

class ThesaurusTest {
    private final Thesaurus thesaurus = new Thesaurus();
    String currentDir = System.getProperty("user.dir");

    @Test
    void testInsert() {
        String[] synonym = {"b", "c", "d"};
        String[] add = {"e", "f", "g"};
        thesaurus.insert("a", synonym);
        assertEquals("a - {b, c, d}\n", thesaurus.toString());
        thesaurus.insert("a", add);
        assertEquals("a - {b, c, d, e, f, g}\n", thesaurus.toString());
        thesaurus.insert("b", add);
        String expected = "a - {b, c, d, e, f, g}\nb - {e, f, g}\n";
        assertEquals(expected, thesaurus.toString());
        thesaurus.insert("", add);
        assertEquals(expected, thesaurus.toString());
    }

    @Test
    void testGetSynonymFor() {
        String[] synonym = {"b", "c", "d"};
        thesaurus.insert("a", synonym);
        String getSynonym = thesaurus.getSynonymFor("a");
        assertTrue(Arrays.asList(synonym).contains(getSynonym));
        String getSynonym2 = thesaurus.getSynonymFor("b");
        assertFalse(Arrays.asList(synonym).contains(getSynonym2));
        String getSynonym3 = thesaurus.getSynonymFor("c");
        assertEquals("", getSynonym3);
    }

    @Test
    void delete() {
        String[] synonym = {"b", "c", "d"};
        thesaurus.insert("a", synonym);
        thesaurus.insert("b", synonym);
        thesaurus.insert("c", synonym);
        thesaurus.insert("d", synonym);
        thesaurus.delete("a");
        String deleteSynonym = "b - {b, c, d}\nc - {b, c, d}\nd - {b, c, d}\n";
        assertEquals(deleteSynonym, thesaurus.toString());
        thesaurus.delete("b");
        thesaurus.delete("c");
        assertEquals("d - {b, c, d}\n", thesaurus.toString());
        thesaurus.delete("c");
        assertEquals("d - {b, c, d}\n", thesaurus.toString());
        thesaurus.delete("");
        assertEquals("d - {b, c, d}\n", thesaurus.toString());
        thesaurus.delete("d");
        assertEquals("", thesaurus.toString());
    }

    @Test
    void testToString() {
        String[] synonym = {"b", "c", "d"};
        thesaurus.insert("a", synonym);
        assertEquals("a - {b, c, d}\n", thesaurus.toString());
    }

    @Test
    void testIsEmpty() {
        String[] synonym = {"b", "c", "d"};
        thesaurus.insert("a", synonym);
        assertFalse(thesaurus.isEmpty());
        thesaurus.delete("a");
        assertTrue(thesaurus.isEmpty());
    }

    @Test
    void testLoadSmallThesaurus() {
        String dir = currentDir + "/src/main/java/proj5/smallThesaurus.txt";
        assertTrue(thesaurus.loadThesaurus(dir));
        assertEquals(11, thesaurus.size());
        thesaurus.delete("terrific");
        assertEquals(10, thesaurus.size());
        thesaurus.delete("ajhsfhidsa");
        assertEquals(10, thesaurus.size());
        thesaurus.delete("grungy");
        assertEquals(9, thesaurus.size());
        thesaurus.delete("grungy");
        assertEquals(9, thesaurus.size());
    }

    @Test
    void testLoadBigThesaurus() {
        String dir = currentDir + "/src/main/java/proj5/bigThesaurus.txt";
        assertTrue(thesaurus.loadThesaurus(dir));
        assertEquals(5006, thesaurus.size());
        thesaurus.delete("terrific");
        assertEquals(5005, thesaurus.size());
        thesaurus.delete("ajhsfhidsa");
        assertEquals(5005, thesaurus.size());
        thesaurus.delete("grungy");
        assertEquals(5004, thesaurus.size());
        thesaurus.delete("fraud");
        String synonym = thesaurus.getSynonymFor("fraud");
        assertEquals(5003, thesaurus.size());
        assertEquals("", synonym);
    }
}
```

```java
package proj5;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class WordCounterTest {
    private final WordCounter wordCounter = new WordCounter();
    String currentDir = System.getProperty("user.dir");

    @Test
    void findFrequenciesAndFrequency() {
        String filePath = currentDir + "/src/main/java/proj5/apartment.txt";
        wordCounter.findFrequencies(filePath);
        assertEquals(5, wordCounter.getFrequency("grungy"));
        assertEquals(7, wordCounter.getFrequency("was"));
        assertNotEquals(3, wordCounter.getFrequency("grungy,"));
        assertEquals(0, wordCounter.getFrequency(""));
        assertEquals(0, wordCounter.getFrequency("$#%^&*("));
    }

    @Test
    void testToString() {
        String filePath = currentDir + "/src/main/java/proj5/apartment.txt";
        wordCounter.findFrequencies(filePath);
        String ans = "--: 1\n" +
                "all: 1\n" +
                "and: 1\n" +
                "apartment: 1\n" +
                "been: 1\n" +
                "could: 2\n" +
                "even: 1\n" +
                "floor: 1\n" +
                "fridge: 1\n" +
                "grass: 1\n" +
                "grungy: 5\n" +
                "have: 1\n" +
                "how: 1\n" +
                "i: 3\n" +
                "insane: 3\n" +
                "is: 1\n" +
                "it: 1\n" +
                "live: 1\n" +
                "outside: 1\n" +
                "so: 1\n" +
                "sofa: 1\n" +
                "that: 2\n" +
                "the: 5\n" +
                "think: 2\n" +
                "this: 1\n" +
                "totally: 1\n" +
                "was: 7\n" +
                "when: 1\n" +
                "where: 1\n" +
                "would: 1\n" +
                "year: 1\n";
        assertEquals(ans, wordCounter.toString());
    }
}
```

```
package proj5;

/**
 * The GrammarChecker class provides functionality to improve grammar
 * @author Kaito Nguyen
 * @version May 31st 2025, Spring 2025
 */

public class GrammarChecker {
    private final int threshold;
    private final Thesaurus thesaurus;

    /**
     * Default constructor that initializes the GrammarChecker with an empty thesaurus.
     */
    public GrammarChecker(String thesaurusFile, int threshold) {
        if (thesaurusFile == null || thesaurusFile.isEmpty()) {
            throw new IllegalArgumentException("Thesaurus file cannot be null or empty");
        }
        thesaurus = new Thesaurus(thesaurusFile);
        this.threshold = threshold;
    }

    /**
     * Formats a word by converting it to lowercase, trimming whitespace, deleting punctuat
ion, etc.
     * @param word the word to format
     * @return the formatted word
     */
    private String formatWord(String word) {
        if (word == null || word.isEmpty()) {
            return word;
        }
        String originalWord = word;
        word = word.toLowerCase();
        word = word.trim();
        word = word.replaceAll("[^a-zA-Z]", "");
        if (word.isEmpty()) {
            return originalWord;
        }
        return word;
    }

    /**
     * Format the first word to be the same as the second word.
     * @param word the word to format
     * @param formattedWord the formatted word to compare against
     * @return the formatted word if it matches the second word
     */
    private String formatWord(String word, String formattedWord) {
        if (word == null || word.isEmpty() || formattedWord == null || formattedWord.isEmpt
y()) {
            return word;
        }
        String formatted = word.toLowerCase();
        if (Character.isUpperCase(formattedWord.charAt(0))) {
            formatted = Character.toUpperCase(formatted.charAt(0)) + formatted.substring(1)
;
        }
        String punctuation = formattedWord.replaceAll("[a-zA-Z]", "");
        if (!punctuation.isEmpty()) {
            formatted += punctuation;
        }
        return formatted;
    }

    /**
     * Improves the grammar of the input file by checking for synonyms and word frequencies
.
```

```
     * * @param textfile the file containing the text to improve
     *
     */
    public String improveGrammarFile(String textfile) {
        if (textfile == null || textfile.isEmpty()) {
            throw new IllegalArgumentException("Text file cannot be null or empty");
        }
        WordCounter wordCounter = new WordCounter();
        wordCounter.findFrequencies(textfile);
        LineReader lineReader = new LineReader(textfile, " ");
        String ans = "";
        String[] words;
        while ((words = lineReader.getNextLine()) != null) {
            for (String word : words) {
                String formattedWord = formatWord(word);
                if (formattedWord.isEmpty()) {
                    ans += word + " ";
                    continue;
                }
                if (wordCounter.getFrequency(formattedWord) > threshold) {
                    String synonym = thesaurus.getSynonymFor(formattedWord);
                    if (synonym != "") {
                        ans += formatWord(synonym, word) + " ";
                    } else {
                        ans += word + " ";
                    }
                } else {
                    ans += word + " ";
                }
            }
            ans += "\n";
        }
        ans = ans.trim();
        lineReader.close();
        return ans;
    }

    /**
     * Improves the grammar of the input file by checking for synonyms and word frequencies
.
     * * @param textfile the file containing the text to improve
     *
     */
    public void improveGrammar(String textfile) {
        String improvedText = improveGrammarFile(textfile);
        System.out.println(improvedText);
    }
}
```

```java
package proj5;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class BinarySearchTreeTest {
    private BinarySearchTree<ThesaurusNode> tree;
    private BinarySearchTree<WordCounterNode> subtree;

    @Test
    void testSize() {
        tree = new BinarySearchTree<>();
        assertEquals(0, tree.size());
    }

    @Test
    void testInsert() {
        tree = new BinarySearchTree<>();
        String key = "key";
        String[] value = {"value", "value2", "value3", "value4", "value5"};
        ThesaurusNode thesaurusNode = new ThesaurusNode(key, value);
        tree.insert(thesaurusNode);
        assertEquals(1, tree.size());

        String key2 = "key2";
        String[] value2 = {"value2", "value3", "value4", "value5"};
        ThesaurusNode thesaurusNode2 = new ThesaurusNode(key2, value2);
        tree.insert(thesaurusNode2);
        assertEquals(2, tree.size());

        String[] value3 = {"value6", "value7", "value8", "value9", "value10"};
        ThesaurusNode thesaurusNode3 = new ThesaurusNode(key, value3);
        tree.insert(thesaurusNode3);
        assertNotEquals(3, tree.size());
        assertEquals(2, tree.size());
    }

    @Test
    void testSearch() {
        tree = new BinarySearchTree<>();
        String key = "key";
        String[] value = {"value", "value2", "value3", "value4", "value5"};
        ThesaurusNode thesaurusNode = new ThesaurusNode(key, value);
        tree.insert(thesaurusNode);
        BSTNode<ThesaurusNode> node = tree.search(thesaurusNode);
        assertNotNull(node);
        assertEquals(node.element.getSynonyms(), value);

        String key2 = "key2";
        String[] value2 = {"value2", "value3", "value4", "value5"};
        ThesaurusNode thesaurusNode2 = new ThesaurusNode(key2, value2);
        tree.insert(thesaurusNode2);
        String key3 = "key3";
        String[] value3 = {"value3", "value4", "value5"};
        String key4 = "key4";
        ThesaurusNode thesaurusNode3 = new ThesaurusNode(key4, value3);
        tree.insert(thesaurusNode3);
        BSTNode<ThesaurusNode> node2 = tree.search(thesaurusNode3);
        assertNotNull(node2);
    }

    @Test
    void testDelete() {
        tree = new BinarySearchTree<>();
        String key = "key";
        String[] value = {"value", "value2", "value3", "value4", "value5"};
        ThesaurusNode thesaurusNode = new ThesaurusNode(key, value);
        tree.insert(thesaurusNode);
        assertEquals(1, tree.size());

        String key2 = "key2";
        String[] value2 = {"value2", "value3", "value4", "value5"};
        ThesaurusNode thesaurusNode2 = new ThesaurusNode(key2, value2);
        tree.insert(thesaurusNode2);
        assertEquals(2, tree.size());

        tree.delete(thesaurusNode);
        assertEquals(1, tree.size());

        tree.delete(thesaurusNode);
        assertEquals(1, tree.size());

        tree.delete(thesaurusNode2);
        assertEquals(0, tree.size());
    }

    @Test
    void testToStringParen() {
        tree = new BinarySearchTree<>();
        ThesaurusNode b = new ThesaurusNode("b", new String[]{"value"});
        tree.insert(b);
        ThesaurusNode c = new ThesaurusNode("c", new String[]{"value2"});
        tree.insert(c);
        ThesaurusNode a = new ThesaurusNode("a", new String[]{"value3"});
        tree.insert(a);
        ThesaurusNode e = new ThesaurusNode("e", new String[]{"value4"});
        tree.insert(e);
        ThesaurusNode d = new ThesaurusNode("d", new String[]{"value5"});
        tree.insert(d);
        ThesaurusNode f = new ThesaurusNode("f", new String[]{"value6"});
        tree.insert(f);
        ThesaurusNode g = new ThesaurusNode("g", new String[]{"value7"});
        tree.insert(g);
        assertEquals("(( a ) b ( c (( d ) e ( f ( g )))))", tree.toStringParen());
    }

    @Test
    void testIsEmpty() {
        tree = new BinarySearchTree<>();
        assertTrue(tree.isEmpty());
        ThesaurusNode thesaurusNode = new ThesaurusNode(null, null);
        tree.insert(thesaurusNode);
        assertFalse(tree.isEmpty());
    }
}
```

```java
package proj5;
/**
 * The BinarySearchTree class represents a binary search tree (BST).
 *
 * @author Kaito Nguyen
 * @version May 31st 2025, Spring 2025
 */

public class BinarySearchTree<E extends Comparable<E>>
{
    private BSTNode<E> root; // root of the binary search tree
    private int size; // number of elements in the tree
    /**
     * Default constructor that initializes an empty binary search tree.
     */
    public BinarySearchTree() {
        root = null;
        size = 0;
    }

    /**
     * Returns the number of elements in this tree.
     *
     * @return the size of the tree
     */
    public int size() {
        return size;
    }

    /**
     * Inserts an element into the binary search tree.
     * @param subroot
     * @param elem the element to insert
     * @param value the value associated with the element
     * @return the new root of the subtree after insertion
     */
    private BSTNode<E> insert(BSTNode<E> subroot,  BSTNode<E> elem) {
        if (subroot == null) {
            size++;
            return elem;
        }
        else if (elem.compareTo(subroot) < 0) {
            subroot.llink = insert(subroot.llink, elem);
        }
        else if (elem.compareTo(subroot) > 0) {
            subroot.rlink = insert(subroot.rlink, elem);
        }
        else {
            subroot.element = elem.element;
        }
        return subroot;
    }

    /**
     * Inserts an element into the binary search tree.
     *
     * @param elem the element to insert
     * @param value the value associated with the element
     */
    public void insert(E elem) {
        if (elem == null) {
            throw new IllegalArgumentException("Element cannot be null");
        }
        BSTNode<E> node = new BSTNode<>(elem);
        root = insert(root, node);
    }

    /**
     * Searches for an element in the binary search tree.
     * @param subroot
     * @param elem the element to search for
     * @return the node containing the element if found, null otherwise
     */
    private BSTNode<E> search(BSTNode<E> subroot, BSTNode<E> elem) {
        if (subroot == null)
            return null;
        else if (elem.compareTo(subroot) < 0)
            return search(subroot.llink, elem);
        else if (elem.compareTo(subroot) > 0)
            return search(subroot.rlink, elem);
        else
            return subroot;
    }

    /**
     * Searches for an element in the binary search tree.
     *
     * @param elem the element to search for
     * @return the node containing the element if found, null otherwise
     */
    public BSTNode<E> search(E elem) {
        if (elem == null) {
            throw new IllegalArgumentException("Element cannot be null");
        }
        BSTNode<E> node = new BSTNode<>(elem);
        return search(root, node);
    }

    /**
     * Finds the minimum element in a subtree rooted at subroot.
     * @param subroot
     * @return the node containing the minimum element
     */
    private BSTNode<E> findMin(BSTNode<E> subroot) {
        if (subroot == null)
            return null;
        else if (subroot.llink == null)
            return subroot;
        else
            return findMin(subroot.llink);
    }

    /**
     * Deletes an element from the binary search tree.
     * @param subroot
     * @param elem the element to delete
     * @return the new root of the subtree after deletion
     */
    private BSTNode<E> delete(BSTNode<E> subroot, BSTNode<E> elem) {
        if (subroot == null)
            return null;
        else if (elem.compareTo(subroot) < 0)
            subroot.llink = delete(subroot.llink, elem);
        else if (elem.compareTo(subroot) > 0)
            subroot.rlink = delete(subroot.rlink, elem);
        else {
            if (subroot.llink == null)  {
                size--;
                return subroot.rlink;
            }
            else if (subroot.rlink == null) {
                size--;
                return subroot.llink;
            }
            else {
                BSTNode<E> minNode = findMin(subroot.rlink);
                subroot.element = minNode.element;
                subroot.rlink = delete(subroot.rlink, minNode);
            }
        }
        return subroot;
    }
    /**
     * Deletes an element from the binary search tree.
     *
     * @param elem the element to delete
     */
    public void delete(E elem) {
        if (elem == null) {
            throw new IllegalArgumentException("Element cannot be null");
        }
        if (search(elem) == null) return;
        BSTNode<E> node = new BSTNode<>(elem);
        root = delete(root, node);
    }

    /** recursive helper for toStringParen
     *
     * @param subroot root of subtree to start at
     * @return inorder string of elements in this subtree
     */
    private String toStringParen(BSTNode<E> subroot) {
          if (subroot == null) // base case
                  return "";
          else
                  return "(" + toStringParen(subroot.llink) + " " +
                  subroot.toString() + " " + toStringParen(subroot.rlink) + ")";
    }

    /**
     * returns string showing tree structure using parentheses, as shown in class
     */
    public String toStringParen() {
          return toStringParen(root);
    }

    /**
     * Returns a string representation of the binary search tree.
     *
     * @return a string representation of the tree
     */
    public String toString() {
        String ans = "";
        for (String elem : inOrderTraversal()) {
            ans += elem + " : " + search((E) elem).element + "\n";
        }
        return ans.trim();
    }

    /**
     * Checks if the tree is empty.
     *
     * @return true if the tree is empty, false otherwise
     */
    public boolean isEmpty() {
        return size == 0;
    }

    /**
     * recursive helper for inOrderTraversal. toStringParen but returns string array
     * @param subroot root of subtree to start at
     * @return string array of elements in this subtree in inorder
     */
    private String[] inOrderTraversal(BSTNode<E> subroot) {
        if (subroot == null) {
            return new String[0];
        }
        String[] left = inOrderTraversal(subroot.llink);
        String[] right = inOrderTraversal(subroot.rlink);
        String[] result = new String[left.length + 1 + right.length];
        for (int i = 0; i < left.length; i++) {
            result[i] = left[i];
        }
        result[left.length] = subroot.toString();
        for (int i = 0; i < right.length; i++) {
            result[left.length + 1 + i] = right[i];
        }
        return result;
    }

    /**
     * inorder traversal of the tree
     * @return string showing elements in inorder
     */
    public String[] inOrderTraversal() {
        return inOrderTraversal(root);
    }
}
```

```java
package proj5;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class GrammarCheckerTest {
    private static final String currentDirectory = System.getProperty("user.dir");
    private static final String TEST_THESAURUS_FILE = currentDirectory + "/src/main/java/pr
oj5/test_thesaurus.txt";
    private static final String TEST_INPUT_FILE = currentDirectory + "/src/main/java/proj5/
test_input.txt";
    private static final String TEST_EMPTY_FILE = currentDirectory + "/src/main/java/proj5/
test_empty.txt";
    private static final String TEST_THESAURUS_NO_MATCH = currentDirectory + "/src/main/jav
a/proj5/test_thesaurus_nomatch.txt";
    private static final String TEST_INPUT_PUNCT_CASE = currentDirectory + "/src/main/java/
proj5/test_punct_case.txt";

    @Test
    void coreReplacementLogic() {
        GrammarChecker checker = new GrammarChecker(TEST_THESAURUS_FILE, 1);
        String output = checker.improveGrammarFile(TEST_INPUT_FILE);
        String[] lines = output.split("\n");

        assertEquals(2, lines.length);

        // Line 1: "This is a very very good test."
        String[] line1Words = lines[0].split(" ");
        assertEquals("This", line1Words[0]);
        assertNotEquals("very", line1Words[3].replaceAll("[^a-zA-Z]", ""));
        assertNotEquals("very", line1Words[4].replaceAll("[^a-zA-Z]", ""));
        assertEquals("good", line1Words[5].replaceAll("[^a-zA-Z]", ""));
        assertTrue(line1Words[6].startsWith("test"));

        // Line 2: "A little little lamb."
        String[] line2Words = lines[1].split(" ");
        assertEquals("A", line2Words[0]);
        assertNotEquals("little", line2Words[1].replaceAll("[^a-zA-Z]", ""));
        assertNotEquals("little", line2Words[2].replaceAll("[^a-zA-Z]", ""));
        assertTrue(line2Words[3].startsWith("lamb"));
    }

    @Test
    void noReplacementsIfThresholdNotMet() {
        GrammarChecker checker = new GrammarChecker(TEST_THESAURUS_FILE, 2);
        String output = checker.improveGrammarFile(TEST_INPUT_FILE);
        String expected = "This is a very very good test. \nA little little lamb.";
        assertEquals(expected, output);
    }

    @Test
    void noReplacementsIfSynonymsNotInThesaurus() {
        GrammarChecker checker = new GrammarChecker(TEST_THESAURUS_NO_MATCH, 1);
        String output = checker.improveGrammarFile(TEST_INPUT_FILE);
        String expected = "This is a very very good test. \nA little little lamb.";
        assertEquals(expected, output);
    }

    @Test
    void emptyInputFileResultsInEmptyOutput() {
        GrammarChecker checker = new GrammarChecker(TEST_THESAURUS_FILE, 1);
        String output = checker.improveGrammarFile(TEST_EMPTY_FILE);
        assertEquals("", output);
    }

    @Test
    void punctuationAndCasePreservation() {
        GrammarChecker checker = new GrammarChecker(TEST_THESAURUS_FILE, 1);
        String output = checker.improveGrammarFile(TEST_INPUT_PUNCT_CASE);
        String[] words = output.split(" ");

        // Original: "Very, very nice. Little little cat."
        // words[0] was "Very,"
        assertNotEquals("Very,", words[0]);
        assertTrue(Character.isUpperCase(words[0].charAt(0)), "Should retain initial capita
l for 'Very,'");
        assertTrue(words[0].endsWith(","), "Should retain comma for 'Very,'");

        // words[1] was "very"
        assertNotEquals("very", words[1].replaceAll("[^a-zA-Z]", ""));

        assertEquals("nice.", words[2]);

        // words[3] was "Little"
        assertNotEquals("Little", words[3].replaceAll("[^a-zA-Z]", ""));
        assertTrue(Character.isUpperCase(words[3].charAt(0)), "Should retain initial capita
l for 'Little'");
        assertFalse(words[3].endsWith("."), "Should not have period if original 'Little' di
dn't");


        // words[4] was "little"
        assertNotEquals("little", words[4].replaceAll("[^a-zA-Z]", ""));

        assertEquals("cat.", words[5]);
    }

    @Test
    void constructor_NullThesaurusFile_ThrowsException() {
        Exception e = assertThrows(IllegalArgumentException.class, () -> new GrammarChecker
(null, 1));
        assertEquals("Thesaurus file cannot be null or empty", e.getMessage());
    }

    @Test
    void constructor_EmptyThesaurusFile_ThrowsException() {
        Exception e = assertThrows(IllegalArgumentException.class, () -> new GrammarChecker
("", 1));
        assertEquals("Thesaurus file cannot be null or empty", e.getMessage());
    }

    @Test
    void improveGrammarFile_NullTextFile_ThrowsException() {
        GrammarChecker checker = new GrammarChecker(TEST_THESAURUS_FILE, 1);
        Exception e = assertThrows(IllegalArgumentException.class, () -> checker.improveGra
mmarFile(null));
        assertEquals("Text file cannot be null or empty", e.getMessage());
    }

    @Test
    void improveGrammarFile_EmptyTextFile_ThrowsException() {
        GrammarChecker checker = new GrammarChecker(TEST_THESAURUS_FILE, 1);
        Exception e = assertThrows(IllegalArgumentException.class, () -> checker.improveGra
mmarFile(""));
        assertEquals("Text file cannot be null or empty", e.getMessage());
    }
}
```

```
package proj5;
/**
 * WordCounterNode class represents a node in a word counting system.
 * @author Kaito Nguyen
 * @version May 31st 2025, Spring 2025
 */

public class WordCounterNode implements Comparable<WordCounterNode> {
    private final String word; // The word being counted
    private int count; // The count of occurrences of the word

    /**
     * Default constructor that initializes the WordCounterNode with a word and a count of
1.
     * @param word the word to be counted
     */
    public WordCounterNode(String word) {
        this.word = word;
        this.count = 1;
    }

    /**
     * Returns the word associated with this WordCounterNode.
     * @return the word
     */
    public String getWord() {
        return word;
    }

    /**
     * Returns the count of occurrences of the word.
     * @return the count
     */
    public int getCount() {
        return count;
    }

    /**
     * Increments the count of occurrences of the word by 1.
     */
    public void incrementCount() {
        count++;
    }

    /**
     * Compares this WordCounterNode with another based on the word.
     * @param other the other WordCounterNode to compare with
     * @return a negative integer, zero, or a positive integer as this node is less than, e
qual to, or greater than the specified node
     */
    @Override
    public int compareTo(WordCounterNode other) {
        return this.word.compareTo(other.word);
    }

    /**
     * Returns a string representation of the WordCounterNode.
     * @return a string representation of the WordCounterNode
     */
    @Override
    public String toString() {
        return word;
    }

}
```

```
package proj5;
/**
 * ThesaurusNode class represents a node in a thesaurus that stores a word and its synonyms
.
 * @author Kaito Nguyen
 * @version May 31st 2025, Spring 2025
 */

public class ThesaurusNode implements Comparable<ThesaurusNode> {
    private String word; // The word for which synonyms are stored
    private String[] synonyms; // Array of synonyms for the word

    /**
     * Default constructor that initializes the ThesaurusNode with a word and its synonyms.
     * @param word the word for which synonyms are stored
     * @param synonyms the array of synonyms for the word
     */
    public ThesaurusNode(String word, String[] synonyms) {
        this.word = word;
        this.synonyms = synonyms;
    }

    /**
     * Returns the word associated with this ThesaurusNode.
     * @return the word
     */
    public String getWord() {
        return word;
    }

    /**
     * Returns the array of synonyms associated with this ThesaurusNode.
     * @return the array of synonyms
     */
    public String[] getSynonyms() {
        return synonyms;
    }

    /**
     * Compares this ThesaurusNode with another based on the word.
     * @param other the other ThesaurusNode to compare with
     * @return a negative integer, zero, or a positive integer as this node is less than, e
qual to, or greater than the specified node
     */
    public int compareTo(ThesaurusNode other) {
        return this.word.compareTo(other.word);
    }

    /**
     * Returns a string representation of the ThesaurusNode.
     * @return a string representation of the ThesaurusNode
     */
    public String toString() {
        return word;
    }
}
```

```java
package proj5;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class WordCounterNodeTest {

    @Test
    void testToString() {
        WordCounterNode wordCounterNode = new WordCounterNode("key");
        for (int i = 0; i < 10; i++) {
            wordCounterNode.incrementCount();
        }
        BSTNode<WordCounterNode> newNode = new BSTNode<>(wordCounterNode);
        assertEquals("key", newNode.toString());
    }

    @Test
    void testCompareTo() {
        WordCounterNode wordCounterNode = new WordCounterNode("key");
        BSTNode<WordCounterNode> newNode = new BSTNode<>(wordCounterNode);

        WordCounterNode wordCounterNode2 = new WordCounterNode("key");
        BSTNode<WordCounterNode> oldNode = new BSTNode<>(wordCounterNode2);
        for (int i = 0; i < 10; i++) {
            wordCounterNode.incrementCount();
        }
        assertEquals(0, newNode.compareTo(oldNode));

        WordCounterNode wordCounterNode3 = new WordCounterNode("key2");
        BSTNode<WordCounterNode> newNode2 = new BSTNode<>(wordCounterNode3);
        assertEquals(1, newNode2.compareTo(oldNode));

        WordCounterNode wordCounterNode4 = new WordCounterNode("key");
        BSTNode<WordCounterNode> newNode3 = new BSTNode<>(wordCounterNode4);
        assertEquals(0, newNode3.compareTo(oldNode));

        WordCounterNode wordCounterNode5 = new WordCounterNode("ke");
        BSTNode<WordCounterNode> newNode4 = new BSTNode<>(wordCounterNode5);
        assertEquals(-1, newNode4.compareTo(newNode));
    }
}
```

```java
package proj5;
/**
 * The BSTNode class represents a node in a binary search tree (BST).
 * Each node contains an element, a link to its left child, and a link to its right child.
 *
 * @author Kaito Nguyen
 * @version May 31st 2025, Spring 2025
 */

public class BSTNode<E extends Comparable<E>> {
    public E element;
    public BSTNode<E> llink;
    public BSTNode<E> rlink;

    /**
     * Default constructor that initializes the node with null links.
     */
    public BSTNode(E elem) {
        this.element = elem;
        this.llink = null;
        this.rlink = null;
    }

    /**
     * Returns a string representation of the element in this node.
     * @return a string representation of the element
     */
    public String toString() {
        return element.toString();
    }

    /**
     * Compares this node with another node based on the element.
     *
     * @param other the other node to compare with
     * @return a negative integer, zero, or a positive integer as this node is less than, e
qual to, or greater than the specified node
     */
    public int compareTo(BSTNode<E> other) {
        return this.element.compareTo(other.element);
    }
}
```

```java
package proj5;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class ThesaurusNodeTest {

    @Test
    void testToString() {
        ThesaurusNode thesaurusNode = new ThesaurusNode("key", new String[]{"value"});
        BSTNode<ThesaurusNode> newNode = new BSTNode<>(thesaurusNode);
        assertEquals("key", newNode.toString());
    }

    @Test
    void testToString2() {
        ThesaurusNode thesaurusNode = new ThesaurusNode("key", new String[]{"value"});
        BSTNode<ThesaurusNode> newNode = new BSTNode<>(thesaurusNode);
        ThesaurusNode thesaurusNode2 = new ThesaurusNode("key", new String[]{"valuue"});
        BSTNode<ThesaurusNode> oldNode = new BSTNode<>(thesaurusNode2);
        assertEquals(newNode.toString(), oldNode.toString());
        assertEquals(oldNode.element.getWord(), newNode.element.getWord());
    }

    @Test
    void testCompareTo() {
        ThesaurusNode thesaurusNode = new ThesaurusNode("key", new String[]{"value"});
        BSTNode<ThesaurusNode> newNode = new BSTNode<>(thesaurusNode);
        ThesaurusNode thesaurusNode2 = new ThesaurusNode("key", new String[]{"valuue"});
        BSTNode<ThesaurusNode> oldNode = new BSTNode<>(thesaurusNode2);
        assertEquals(0, newNode.compareTo(oldNode));

        ThesaurusNode thesaurusNode3 = new ThesaurusNode("key2", new String[]{"value"});
        BSTNode<ThesaurusNode> newNode2 = new BSTNode<>(thesaurusNode3);
        assertEquals(1, newNode2.compareTo(oldNode));

        ThesaurusNode thesaurusNode4 = new ThesaurusNode("key", new String[]{"value2"});
        BSTNode<ThesaurusNode> newNode3 = new BSTNode<>(thesaurusNode4);
        assertEquals(0, newNode3.compareTo(oldNode));

        ThesaurusNode thesaurusNode5 = new ThesaurusNode("ke", new String[]{"value"});
        BSTNode<ThesaurusNode> newNode4 = new BSTNode<>(thesaurusNode5);
        assertEquals(-1, newNode4.compareTo(newNode));
    }
}
```

```
package proj5;
/**
 * WordCounter class allows adding word and checking the count of a word.
 *
 * @author Kaito Nguyen
 * @version May 31st 2025, Spring 2025
 */

public class WordCounter {
    private final BinarySearchTree<WordCounterNode> wordCountTree; // Binary search tree to
 store word counts

    /**
     * Default constructor that initializes an empty word counter.
     */
    public WordCounter() {
        wordCountTree = new BinarySearchTree<>();
    }

    /**
     * Formats a word by converting it to lowercase, trimming whitespace, deleting punctuat
ion, etc.
     * @param word the word to format
     * @return the formatted word
     */
    private String formatWord(String word) {
        if (word == null || word.isEmpty()) {
            return word;
        }
        String originalWord = word; // Store the original word for error messages
        word = word.toLowerCase();
        word = word.trim();
        word = word.replaceAll("[^a-zA-Z]", "");
        if (word.isEmpty()) {
            return originalWord;
        }
        return word;
    }

    private BSTNode<WordCounterNode> search(String word) {
        WordCounterNode wordNode = new WordCounterNode(word);
        return wordCountTree.search(wordNode);
    }

    /**
     * Adds words from file to the word counter.
     *
     * @param word the word to add
     */
    public void findFrequencies(String fileString) {
        LineReader lineReader = new LineReader(fileString, " ");
        String[] words;
        while ((words = lineReader.getNextLine()) != null) {
            for (String word : words) {
                word = formatWord(word);
                if (word != null && !word.isEmpty()) {
                    BSTNode<WordCounterNode> node = search(word);
                    if (node != null) {
                        node.element.incrementCount();
                    } else {
                        WordCounterNode wordCounterNode = new WordCounterNode(word);
                        wordCountTree.insert(wordCounterNode);
                    }
                }
            }
        }
        lineReader.close();
    }
    /**
     * Gets the count of a specific word.
     *
     * @param word the word to get the count for
     * @return the count of the word, or 0 if the word does not exist
     */
    public int getFrequency(String word) {
        if (word == null || word.isEmpty()) {
            return 0;
        }
        BSTNode<WordCounterNode> node = search(word);
        if (node != null) {
            return node.element.getCount();
        }
        return 0;
    }

    /**
     * toString method to return a string representation of the word counter.
     */
    public String toString() {
        String ans = "";
        for (String word : wordCountTree.inOrderTraversal()) {
            ans += word + ": " + search(word).element.getCount() + "\n";
        }
        return ans;
    }
}
```

```java
package proj5;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * The LineReader class lets you read and parse an input file
 * one line at a time.
 *
 * @author Chris Fernandes
 * @version 2/28/18
 */
public class LineReader {

        private Scanner sc;
        private String delimiter;

        /** Constructor.
         *
         * @param file path to file to read.  For example,
         * "src/input.txt".  Use forward slashes to
         * separate directories (folders) and don’t forget the quotes.
         * @param delimiter When reading lines, delimiter is used to parse the tokens.
         * For example, "," should be used for a comma-delimited file (like in
         * a thesaurus). " " should be used where spaces separate words (like
         * in an input file to be grammar-checked).
         */
        public LineReader(String file, String delimiter) {
                try {
                        sc = new Scanner(new File(file));
                        this.delimiter = delimiter;
                } catch (FileNotFoundException e)
                {System.out.println("file not found error");}
        }

        /** Returns next line of input file as an array of strings.
         *  For a thesaurus input file, index 0 in the
         *  array will hold the keyword and the remaining positions
         *  will be synonyms of the keyword.  Returns null if
         *  end of file is reached.
         *
         * @return line of input as array of strings (delimiter is consumed)
         */
        public String[] getNextLine()
        {
                if (sc.hasNextLine())
                        return sc.nextLine().split(delimiter);
                else
                        return null;
        }

        /** Closes this LineReader.
         *
         */
        public void close()
        {
                sc.close();
        }
}
```