```java
package proj5;
/**
 *
 *  Theo Scola
 *  Ver: 06/4/2025
 *  I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 * accordance with the Union College Honor Code and the course syllabus
 *  Deque: Creates an ADT that has two accessible sides, left and right.
 *      The only accessible items in the deque are the first elements in the left and right ends.
 *      Left and right ends can add a new element or remove the current element.
 */


public class Deque {

    /**
     * Invariant:
     * 1. Capacity represents the capacity of the Deque, only changed by trimToSize and ensureCapacity
     * 2. DLL represents how the Data is held in a doublyLinkedList.
     *      It should only be accessed when adding, removing, or getting elements
     */

    private int capacity;
    private DoublyLinkedList DLL;

    private final int INITIAL_CAPACITY = 10;

    /**
     * Creates a new deque with initial capacity 10.
     */
    public Deque() {
        DLL = new DoublyLinkedList();
        capacity = INITIAL_CAPACITY;
```

```
34      }
35
36      /**
37       * Creates a new deque.
38       *
39       * @param initialCapacity the initial capacity of
    the deque.
40       */
41      public Deque(int initialCapacity) {
42          DLL = new DoublyLinkedList();
43          capacity = initialCapacity;
44      }
45
46      /**
47       * inserts a String on the left of the deque.
48       * If this cannot be done because the deque has
49       * reached its capacity, the deque will
50       * expand to twice its current capacity plus 1
51       * to accommodate the new entry.
52       *
53       * @param value the String to add.
54       */
55      public void addLeft(String value) {
56          if (size() == getCapacity()) {
57              int expandedSize = DLL.getLength() * 2 +
   1;
58              ensureCapacity(expandedSize);
59          }
60          DLL.insertAtHead(value);
61      }
62
63      /**
64       * inserts a String on the right of the deque.
65       * If this cannot be done because the deque has
66       * reached its capacity, the deque will
67       * expand to twice its current capacity plus 1
68       * to accommodate the new entry.
69       *
70       * @param value the String to add.
71       */
72      public void addRight(String value) {
```

```
 73          if (size() == getCapacity()) {
 74              int expandedSize = DLL.getLength() * 2
    + 1;
 75              ensureCapacity(expandedSize);
 76          }
 77          DLL.insertAtTail(value);
 78      }
 79
 80      /**
 81       * Remove and return the left item from the
   deque.  Return
 82       * null if the deque is empty.
 83       */
 84      public String removeLeft() {
 85          if (!(isEmpty())) {
 86              String temp = leftMost();
 87              DLL.removeHead();
 88              return temp;
 89          } else {
 90              return null;
 91          }
 92      }
 93
 94      /**
 95       * Remove and return the right item from the
   deque.  Return
 96       * null if the deque is empty.
 97       */
 98      public String removeRight() {
 99          if (!(isEmpty())) {
100              String temp = rightMost();
101              DLL.removeTail();
102              return temp;
103          } else {
104              return null;
105          }
106      }
107
108      /**
109       * Places the contents of another deque in order
    at the right
```

```
110      * end of this deque.  For example, if this
    deque is {A,B} and
111      * the other deque is {B,C,D}, addAll will
    change this deque
112      * to be {A,B,B,C,D}.
113      * If adding all elements of the other deque
    would exceed the
114      * capacity of this deque, the capacity is
    changed to make
115      * exactly enough room for all the elements to
    be added.
116      * Postcondition: NO SIDE EFFECTS!  the other
    deque should be left
117      * unchanged.
118      *
119      * @param otherDeque the deque whose contents
    should be added.
120      */
121     public void addAll(Deque otherDeque) {
122         ensureCapacity(size() + otherDeque.size());
123         Deque tempDeque = otherDeque.clone();
124         while (!tempDeque.isEmpty()) {
125             addRight(tempDeque.removeLeft());
126         }
127     }
128
129     /**
130      * Make a copy of this deque.  Changes to the
    copy
131      * do not affect the current deque, and vice
    versa.
132      * Postcondition: NO SIDE EFFECTS!  This deque's
     leftmost
133      * and rightmost elements should remain
    unchanged.
134      *
135      * @return the copy of this deque.
136      */
137     public Deque clone() {
138         Deque newDeque = new Deque(getCapacity());
139         for (int i = 0; i < size(); i++) {
```

```
140             String removed = removeLeft();
141             newDeque.addRight(removed);
142             addRight(removed);
143         }
144         return newDeque;
145     }
146 
147 
148     /**
149      * Change the capacity of this deque to
    minCapacity
150      * if it doesn't already have that much capacity
    . Does
151      * nothing if current capacity is already >=
    minCapacity.
152      *
153      * @param minCapacity the minimum capacity that
    the deque
154      *                    should now have.
155      */
156     public void ensureCapacity(int minCapacity) {
157         capacity = getCapacity();
158         if (capacity < minCapacity) {
159             capacity = minCapacity;
160         }
161     }
162 
163     /**
164      * Getter for the amount of data this deque can
    potentially hold.
165      *
166      * @return the capacity of the deque.
167      */
168     public int getCapacity() {
169         return capacity;
170     }
171 
172     /**
173      * Returns the leftmost element in the deque
    without
174      * altering the deque itself.
```

```
175      *
176      * @return the leftmost element in the deque, or
177      * null if deque is empty
178      */
179     public String leftMost() {
180         int head = 0;
181         return DLL.getIthData(head);
182     }
183 
184     /**
185      * Returns the rightmost element in the deque
    without
186      * altering the deque itself.
187      *
188      * @return the rightmost element in the deque,
    or
189      * null if deque is empty
190      */
191     public String rightMost() {
192         int tail = size() - 1;
193         return DLL.getIthData(tail);
194     }
195 
196 
197     /**
198      * Getter for the amount of data this deque
    currently holds.
199      *
200      * @return the number of elements stored in the
    deque.
201      */
202     public int size() {
203         return DLL.getLength();
204     }
205 
206 
207     /**
208      * Reduce the current capacity to its actual
    size, so that it has
209      * capacity to store only the elements currently
     stored.
```

```
210      */
211     public void trimToSize() {
212         capacity = size();
213     }
214 
215     /**
216      * Produce a left-to-right string representation of this deque.
217      * For example, a deque of three elmts with capacity 5 with "A" as its leftmost elmt,
218      * "C" as its rightmost elmt, and "B" in the middle would produce this string:
219      * {A, B, C} (capacity = 5)
220      * The string you create should be formatted like the above example,
221      * with a comma and space following each element, no comma following the
222      * last element, and all on a single line. An empty deque
223      * should give back "{}" followed by its capacity.
224      *
225      * @return a string representation of this deque.
226      */
227     public String toString() {
228         String toReturn = "{";
229         Deque clone = clone();
230         String[] stringList = new String[clone.size()];
231         int lastElementIndex = stringList.length - 1;
232         for (int i = 0; i < stringList.length; i++) {
233             stringList[i] = (String)clone.removeLeft();
234             if (i < lastElementIndex) {
235                 toReturn += stringList[i] + ", ";
236             }
237         }
238         if (stringList.length != 0) {
```

```
239
240             toReturn += stringList[lastElementIndex
    ];
241         }
242         toReturn += "} " + "(capacity = " +
    getCapacity() + ")";
243
244         return toReturn;
245
246     }
247
248     /**
249      * Checks whether another deque is equal to this
     one.  To be
250      * considered equal, both deques must have
    exactly the same
251      * elements in the same order.  The capacity can
     differ.
252      * <p>
253      * NO SIDE EFFECTS!  Both this deque and the
    other deque
254      * must be the same at the end of this method as
     they were
255      * at the start of this method.
256      *
257      * @param other the other Deque with which to
    compare
258      * @return true iff the other Deque is equal to
    this one. Else false.
259      */
260     public boolean equals(Deque other) {
261         boolean equals = true;
262         Deque thisClone = clone();
263         Deque otherClone = other.clone();
264         int thisSize = thisClone.size();
265         int otherSize = otherClone.size();
266         if (thisSize != otherSize) {
267             return false;
268         }
269         for (int i = 0; i < thisSize; i++) {
270             if (!(thisClone.removeLeft().equals(
```

```
otherClone.removeLeft())) {
                    equals = false;
                }
            }
            return equals;
        }

        /**
         * @return true if deque empty, else false
         */
        public boolean isEmpty() {
            if (size() == 0) {
                return true;
            } else {
                return false;
            }
        }

        /**
         * empty the deque
         */
        public void clear() {
            DLL = new DoublyLinkedList();
        }
}

```

```java
package proj5;

/** BSTNode class.
 *
 * @author Theo Scola
 * @version 6/2/25
 */
public class BSTNode<E extends Comparable>{

    public E data;
    public BSTNode<E> llink;
    public BSTNode<E> rlink;

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

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

    /**
     *CompareTo method
     * @param otherNode the object to be compared.
     * @return pos if this greater than other, neg if this less than other, 0 if equal
     */
    public int compareTo(Object otherNode)
    {
        Object otherData = otherNode;
```

```
41          E thisData = data;
42          return thisData.compareTo(otherData);
43      }
44
45      /**
46       * Checks to see if this BSTNode is a leaf or not
47       * @return true if yes, false if no
48       */
49      public boolean isLeaf()
50      {
51          if(llink ==  null && rlink == null)
52          {
53              return true;
54          }
55          else{
56              return false;
57          }
58      }
59  }
60
```

```java
package proj5;
/**
 * ThesaurusEntry Test Class
 *
 * Author: Theo Scola
 * Ver: 6/5/25
 *
 */
import org.junit.*;
import org.junit.After;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

public class TETests {
    ThesaurusEntry TE1;
    ThesaurusEntry TE2;

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

    @After
    public void tearDown(){
        TE1 = null;
        TE2 = null;
    }

    @Test
    public void testCompareTo0()
    {
        TE1 = new ThesaurusEntry("happy");
        TE2 = new ThesaurusEntry("sad");
        assertTrue("testing compareTo returns
  correctly",  TE1.compareTo(TE2) < 0);
    }

    @Test
    public void testCompareTo1()
    {
        StringBag bag1 = new StringBag();
        bag1.insert("joyful");
        TE1 = new ThesaurusEntry("happy", bag1);
```

```
41          TE2 = new ThesaurusEntry("sad");
42          assertTrue("testing compareTo returns
   correctly",  TE1.compareTo(TE2) < 0);
43      }
44
45      @Test
46      public void testCompareTo2()
47      {
48          StringBag bag1 = new StringBag();
49          bag1.insert("joyful");
50          StringBag bag2 = new StringBag();
51          bag2.insert("glum");
52          TE1 = new ThesaurusEntry("happy", bag1);
53          TE2 = new ThesaurusEntry("sad", bag2);
54          assertTrue("testing compareTo returns
   correctly",  TE2.compareTo(TE1) > 0);
55      }
56
57      @Test
58      public void testCompareTo3()
59      {
60          TE1 = new ThesaurusEntry("zinc");
61          String word = "gang";
62          assertTrue("testing compareTo when inputting
   a string", TE1.compareTo((Object)word) > 0);
63      }
64
65      @Test
66      public void testToString0()
67      {
68          StringBag bag1 = new StringBag();
69          bag1.insert("sandwich");
70          TE1 = new ThesaurusEntry("burger", bag1);
71          assertEquals("testing toString", "burger - {
   sandwich}", TE1.toString());
72      }
73
74  }
75
```

```java
package proj5;
/**
 * WordCounterEntry class:
 * Author: Theo Scola
 * Ver: 6/4/2025
 */
public class WCEntry implements Comparable {
    /**
     * INVARIANT:
     * This represents the data that is put into the WordCounter
     * word: individual word that is found within text files
     * frequency: amount of times that word appears in the text file
     */
    private String word;
    private int frequency;

    /**
     * Non-Default constructor
     *
     * @param newKey word with associated frequency
     */
    public WCEntry(String newKey) {
        word = newKey;
        frequency = 1;
    }

    /**
     * Getter for the frequency of this entry's word
     *
     * @return frequency
     */
    public int getFrequency() {
        return frequency;
    }

    /**
     * getter for word
     * @return word
```

```
39      */
40     public String getWord()
41     {
42         return word;
43     }
44 
45     /**
46      * CompareTo method
47      * @param other the object to be compared.
48      * @return pos if this greater than other, neg if
    other greater than this, 0 if equal
49      */
50     public int compareTo(Object other) {
51         if (other.getClass() == WCEntry.class) {
52             String otherWord = ((WCEntry)other).
   getWord();
53             return word.compareTo(otherWord);
54         } else if (other.getClass() == String.class
   ) {
55             return word.compareTo((String)other);
56         }
57         else{
58             return -1;
59         }
60     }
61 
62     /**
63      * Setter for frequency, increments by 1
64      */
65     public void incrementFrequency()
66     {
67         frequency++;
68     }
69 
70     /**
71      * toString
72      * ex:
73      * are: 2
74      * @return string representation of WCEntry
75      */
76     public String toString()
```

```
77      {
78          return word + ": " + frequency;
79      }
80  }
81
```

```java
package proj5;

/**
 * WordCounter Test Class
 *
 * Author: Theo Scola
 * Ver: 6/5/25
 *
 */
import org.junit.*;
import org.junit.After;
import org.junit.Before;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

public class WCTests {
    WordCounter WC;
    @Rule
    public Timeout timeout = Timeout.millis(100);

    @Before
    public void setUp() throws Exception{
        WC = new WordCounter();
    }

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

    @Test
    public void testConstruct()
    {
        assertNotNull("test constructor works", WC);
    }

    @Test
    public void testFindFrequencies()
    {
        WC.findFrequencies("src/apartment.txt");
        System.out.println(WC);
```

```
42      }
43
44      @Test
45      public void testGetFrequencies0()
46      {
47          WC.findFrequencies("src/apartment.txt");
48          assertEquals("testing get frequencies returns
    correct num",
49                  7, WC.getFrequency("was"));
50      }
51
52      @Test
53      public void testGetFrequencies1()
54      {
55          WC.findFrequencies("src/apartment.txt");
56          assertEquals("testing get frequencies with
   word not in WC", 0, WC.getFrequency("awesome"));
57      }
58
59      @Test
60      public void testGetFrequencies2()
61      {
62          assertEquals("testing get frequencies with
   empty WC", 0, WC.getFrequency("awesome"));
63      }
64  }
65
```

```java
package proj5;

/**
 * BinarySearchTree Test Class
 * Author: Theo Scola
 * Ver: 6/5/25
 *
 */
import org.junit.*;
import org.junit.After;
import org.junit.Before;
import org.junit.rules.Timeout;

import static org.junit.Assert.*;


public class BSTTests {
    private BinarySearchTree BST;
    @Rule
    public Timeout timeout = Timeout.millis(100);

    @Before
    public void setUp() throws Exception {
        BST = new BinarySearchTree<>();
    }

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

    @Test
    public void testSize0()
    {
        assertEquals("Testing size when size is 0", 0
, BST.size());
    }

    @Test
    public void testSize1()
    {
```

```
41          BST.insert("A");
42          assertEquals("Testing size when size is 1", 1
    , BST.size());
43      }
44
45      @Test
46      public void testSize2()
47      {
48          BST.insert("B");
49          BST.insert("A");
50          BST.insert("C");
51          assertEquals("Testing size when size is more
    than 1", 3, BST.size());
52      }
53
54      @Test
55      public void testSearch0(){
56          assertFalse("Testing search with no elements
    returns false", BST.search("A"));
57      }
58
59      @Test
60      public void testSearch1()
61      {
62          BST.insert("B");
63          BST.insert("A");
64          BST.insert("C");
65          assertTrue("Testing search with correct
    element returns true", BST.search("A"));
66      }
67
68      @Test
69      public void testSearch3()
70      {
71          BST.insert("B");
72          BST.insert("A");
73          BST.insert("C");
74          assertFalse("Testing search with incorrect
    returns false", BST.search("D"));
75      }
76
```

```
 77     @Test
 78     public void testToString0()
 79     {
 80         assertEquals("Testing toString with no
    elements", "", BST.toString());
 81     }
 82 
 83     @Test
 84     public void testToString1()
 85     {
 86         BST.insert("A");
 87         assertEquals("Testing toString with 1
    element", "A\n", BST.toString());
 88     }
 89 
 90     @Test
 91     public void testToString2()
 92     {
 93         BST.insert("B");
 94         BST.insert("A");
 95         BST.insert("C");
 96         assertEquals("Testing toString with 3
    elements", "A\nB\nC\n", BST.toString());
 97     }
 98 
 99     @Test
100     public void testInsert0()
101     {
102         BST.insert("A");
103         assertEquals("Testing inserting with no
    elements work", "A\n", BST.toString());
104     }
105 
106     @Test
107     public void testInsert1()
108     {
109         BST.insert("B");
110         BST.insert("A");
111         BST.insert("C");
112         assertEquals("Testing insert correctly
    inserts 3 elements","A\nB\nC\n", BST.toString());
```

```
113     }
114
115     @Test
116     public void testSearchNonLeafInRightSubtree()
117     {
118         BST.insert("M");
119         BST.insert("V");
120         BST.insert("P");
121         BST.insert("D");
122         BST.insert("N");
123         BST.insert("F");
124         BST.insert("B");
125         BST.insert("C");
126         BST.insert("Q");
127         BST.insert("R");
128         assertTrue("Testing search for Q returns
    true", BST.search("Q"));
129     }
130
131     @Test
132     public void testDelete1()
133     {
134        BST.insert("A");
135        BST.delete("A");
136        assertEquals("Testing delete works on leaf",
    "",BST.toString());
137     }
138
139     @Test
140     public void testDelete2()
141     {
142         BST.insert("A");
143         BST.delete("B");
144         assertEquals("Testing delete does not delete
     incorrect value", "A\n", BST.toString());
145     }
146
147     @Test
148     public void testDelete3()
149     {
150         BST.insert("A");
```

```
151         BST.insert("B");
152         BST.insert("C");
153         BST.delete("B");
154         assertEquals("Testing delete with value with
     right child", "A\nC\n", BST.toString());
155     }
156 
157     @Test
158     public void testDelete4()
159     {
160         BST.insert("A");
161         BST.insert("C");
162         BST.insert("B");
163         BST.delete("C");
164         assertEquals("Test delete with value with
    left child", "A\nB\n", BST.toString());
165     }
166 
167     @Test
168     public void testDelete5()
169     {
170         BST.insert("M");
171         BST.insert("D");
172         BST.insert("F");
173         BST.insert("B");
174         BST.insert("C");
175         BST.delete("D");
176         assertEquals("Test delete with value with
    two children", "B\nC\nF\nM\n", BST.toString());
177     }
178 
179     @Test
180     public void testGrabData0()
181     {
182         BST.insert("B");
183         BST.insert("A");
184         BST.insert("C");
185         assertEquals("testing grab data", "A", BST.
    grabData("A"));
186     }
187 
```

```
188 }
189
```

```java
package proj5;
/**
 * The ListNode class is more data-specific than the DoublyLinkedList class.  It
 * details what a single node looks like.  This node has one data field holding
 * a string along with pointers to the node in front of it and the node behind it.
 * This is the only class where I'll let you use public instance variables.
 *
 * @author Chris Fernandes
 * @version 4/28/25
 */
public class ListNode
{
    public String data;
    public ListNode next;
    public ListNode prev;

    /**
     * non-default constructor
     * @param newData int to hold
     */
    public ListNode(String newData) {
        data = newData;
        next = null;
        prev = null;
    }

    /**
     * get node contents in string form
     * @return data in string format
     */
    public String toString() {
        return data;
    }

}

```

```java
package proj5;

import java.util.Random;

/**
 * StringBag ADT
 * Author: Theo Scola
 * Ver: 6/5/2025
 */
public class StringBag {
    /**
     * INVARIANT:
     * deque represents the underlying ADT that will hold all the data in Bag.
     * All of deque's code/Invariant can be seen in the attatched Deque class
     */
    private Deque deque;

    /**
     * Default Constructor
     */
    public StringBag()
    {
        deque = new Deque();
    }

    /**
     * inserts a new String into the bag
     * @param value to insert
     */
    public void insert(String value)
    {
        deque.addRight(value);
    }

    /**
     * grabs(but doesn't remove) a random value
     * @return grabbed value
     */
    public String grab()
```

```
40      {
41          if(!isEmpty())
42          {
43              shuffle();
44          }
45          return deque.leftMost();
46      }
47
48      /**
49       * removes a random value
50       * @return removed value
51       */
52      public String remove()
53      {
54          if(!isEmpty())
55          {
56              shuffle();
57          }
58          return deque.removeLeft();
59      }
60
61      /**
62       * Returns the number of items in the bag
63       * @return size
64       */
65      public int size()
66      {
67          return deque.size();
68      }
69
70      /**
71       * Returns a Boolean of if the bag is empty or
   not
72       * @return if bag is empty or not
73       */
74      public boolean isEmpty()
75      {
76          return deque.isEmpty();
77      }
78
79      /**
```

```
 80      * Adds all values from otherBag to this bag,
    does not change otherBag
 81      * @param otherBag bag to add values from
 82      */
 83     public void addAll(StringBag otherBag)
 84     {
 85         StringBag clone = otherBag.clone();
 86         int size = clone.size();
 87         for(int i = 0; i<size; i++)
 88         {
 89             deque.addLeft(clone.remove());
 90         }
 91     }
 92 
 93     /**
 94      * Clone method - clones the current StringBag
    and returns it
 95      * @return clone - the cloned Bag
 96      */
 97     public StringBag clone()
 98     {
 99         Deque cloneDeque = deque.clone();
100         StringBag clone = new StringBag();
101         int size = cloneDeque.size();
102         for(int i = 0; i<size; i++)
103         {
104             clone.insert(cloneDeque.removeLeft());
105         }
106 
107         return clone;
108     }
109 
110     /**
111      * Equals method
112      * @param otherBag otherBag to check if equal
    with this bag
113      * @return true if equal, false if not
114      */
115     public boolean equals(StringBag otherBag)
116     {
117         if(isEmpty() && otherBag.isEmpty())
```

```
118         {
119             return true;
120         }
121         int thisSize = size();
122         int otherSize = otherBag.size();
123         if(thisSize == otherSize) {
124             StringBag thisClone = clone();
125             StringBag otherClone = otherBag.clone();
126             String[] thisArray = new String[thisSize
    ];
127             String[] otherArray = new String[
    otherSize];
128             boolean found = false;
129 
130             for(int i = 0; i<thisSize; i++)
131             {
132                 thisArray[i] = thisClone.remove();
133                 otherArray[i] = otherClone.remove();
134             }
135             for(String thisItem: thisArray)
136             {
137                 found = false;
138                 for(String otherItem: otherArray)
139                 {
140                     if(thisItem != null && otherItem
     != null) {
141                         if (thisItem.equals(
    otherItem)) {
142                             found = true;
143                             thisItem = null;
144                             otherItem = null;
145                         }
146                     }
147                 }
148                 if(found == false)
149                 {
150                     return false;
151                 }
152             }
153             return found;
154         }
```

```
155         else{
156             return false;
157         }
158     }
159 
160     /**
161      * toString method
162      * ex: {A, B, C}
163      * @return string representation of the bag
164      */
165     public String toString()
166     {
167         if(!isEmpty()) {
168             String toReturn = "{";
169             StringBag clone = clone();
170             int size = clone.size();
171             if (size > 1) {
172                 for (int i = 0; i < size - 1; i++) {
173                     toReturn += clone.remove() +
    ", ";
174                 }
175             }
176             toReturn += clone.remove() + "}";
177             return toReturn;
178         }
179         else{
180             return "{}";
181         }
182     }
183 
184 
185     /**
186      * Helper method to randomize the bag
187      */
188     private void shuffle()
189     {
190         Random gen = new Random();
191         int times = gen.nextInt(deque.size());
192         for(int i = 0; i<times; i++)
193         {
194             deque.addRight(deque.removeLeft());
```

```
195             }
196         }
197
198 }
199
```

```java
package proj5;
/**
 * Thesaurus Class: Data structure that holds words and their associated synonyms.
 * You can look up a word and retrieve a synonym for it.
 *
 * Author: Theo Scola
 * Ver: 6/5/2025
 */
public class Thesaurus {
    /**
     * INVARIANT:
     * BST: BinarySearchTree that holds all the data in the Thesaurus,
     *  only changed with insert() and delete().
     */

    private BinarySearchTree<ThesaurusEntry> BST;

    /**
     * Default Constructor
     */
    public Thesaurus(){
        BST = new BinarySearchTree<>();
    }

    /**
     * Non-default constructor
     * @param file file to build thesaurus off of
     */
    public Thesaurus(java.lang.String file){
        BST = new BinarySearchTree<>();
        LineReader LR = new LineReader(file, ",");

        String[] tempArray = LR.getNextLine();
        while(tempArray != null)
        {
            String keyword = tempArray[0];
            String[] syns = new String[tempArray.length-1];
```

```
38              for(int i = 1; i< tempArray.length; i
   ++) {
39                  syns[i-1] = (tempArray[i]);
40                  //using i-1 because tempArray is 1
   larger than syns
41              }
42              insert(keyword, syns);
43              tempArray = LR.getNextLine();
44          }
45      }
46
47      /**
48       * inserts entry and synonyms into thesaurus.
49       * If entry does not exist, it creates one.
50       * If it does exist, it adds the given synonyms
   to the entry's synonym list
51       * @param entry keyword
52       * @param syns synonyms of keyword
53       */
54      public void insert(String entry, String[] syns)
55      {
56          StringBag synonymsBag = new StringBag();
57          for(String synonym: syns)
58          {
59              synonymsBag.insert(synonym);
60          }
61          ThesaurusEntry newEntry = new ThesaurusEntry(
   entry, synonymsBag);
62          if(BST.search(newEntry))
63          {
64              ThesaurusEntry existing = BST.grabData(
   newEntry);
65              existing.addSynonyms(synonymsBag);
66          }
67          else {
68              BST.insert(newEntry);
69          }
70      }
71
72      /**
73       * Deletes an entry from the Thesaurus
```

```
 74      * @param entry keyword to delete
 75      */
 76     public void delete(String entry)
 77     {
 78         ThesaurusEntry toDelete = new ThesaurusEntry
    (entry);
 79         BST.delete(toDelete);
 80     }
 81 
 82     /**
 83      * returns a random synonym for a keyword
 84      * @param keyword to find a synonym for
 85      * @return a random synonym from its connected
    list
 86      */
 87     public String getSynonymFor(String keyword){
 88         ThesaurusEntry toGet = new ThesaurusEntry(
    keyword);
 89         ThesaurusEntry entry = BST.grabData(toGet);
 90         if(entry == null)
 91         {
 92             return "";
 93         }
 94         else{
 95             return entry.getSynonym();
 96         }
 97     }
 98 
 99     /**
100      * toString method:
101      * @return returns a string representation of
    the Thesaurus
102      */
103     public String toString(){
104         return BST.toString();
105     }
106 
107     /**
108      * Checks to see if keyword is in the Thesaurus
109      * @param keyword to search for
110      * @return true if in, false if not
```

```
111      */
112     public boolean contains(String keyword)
113     {
114         ThesaurusEntry toFind = new ThesaurusEntry(
    keyword);
115         return BST.search(toFind);
116     }
117 }
118 
```

```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");}
```

```
35     }
36
37     /** Returns next line of input file as an array
   of strings.
38      *  For a thesaurus input file, index 0 in the
39      *  array will hold the keyword and the remaining
    positions
40      *  will be synonyms of the keyword.  Returns
   null if
41      *  end of file is reached.
42      *
43      * @return line of input as array of strings (
   delimiter is consumed)
44      */
45     public String[] getNextLine()
46     {
47         if (sc.hasNextLine())
48             return sc.nextLine().split(delimiter);
49         else
50             return null;
51     }
52
53     /** Closes this LineReader.
54      *
55      */
56     public void close()
57     {
58         sc.close();
59     }
60 }
```

```java
package proj5;

/**
 * WordCounter Class: class for computing word frequencies from a text file
 *
 * Author: Theo Scola
 * Ver: 6/3/25
 */
public class WordCounter {
    /**
     * INVARIANT:
     * BST: internal representation of the WordCounter
     * Words can only be inserted into the WordCounter through findFrequencies.
     * Frequencies of each word can be returned
     */
    private BinarySearchTree<WCEntry> BST;

    /**
     * Default constructor
     */
    public WordCounter()
    {
        BST = new BinarySearchTree<>();
    }

    /**
     * Computes frequency of each word in given file
     * @param file given file
     */
    public void findFrequencies(String file)
    {
        LineReader LR = new LineReader(file, " ");

        String[] tempArray = LR.getNextLine();
        while(tempArray != null) {
            for (int i = 0; i < tempArray.length; i ++) {
                if(isLetter(tempArray[i].charAt(0
```

```
37 ))) {
38                     tempArray[i] = stripPunctuation(
   tempArray[i]);
39                     WCEntry newEntry = new WCEntry(
   tempArray[i]);
40                     insert(newEntry);
41                 }
42             }
43             tempArray = LR.getNextLine();
44         }
45     }
46 
47     /**
48      * Helper for findFrequencies
49      * @param entry new word to insert
50      */
51     private void insert(WCEntry entry)
52     {
53         if(BST.search(entry))
54         {
55             WCEntry existing = BST.grabData(entry);
56             existing.incrementFrequency();
57         }
58         else{
59             BST.insert(entry);
60         }
61     }
62 
63     /**
64      * Helper for findFrequencies
65      * @param unStripped String to potentially strip
66      * @return stripped or unchanged string
67      */
68     private String stripPunctuation(String unStripped
   )
69     {
70         int length = unStripped.length();
71         char lastChar = unStripped.charAt(length-1);
72         if(isLetter(lastChar))
73         {
74             return unStripped;
```

```
 75          }
 76          else{
 77              String toReturn = "";
 78              for(int i =0 ; i< length-1; i++)
 79              {
 80                  toReturn+=unStripped.charAt(i);
 81              }
 82              return toReturn;
 83          }
 84      }
 85
 86      /**
 87       * Helper for stripPunctuation
 88       * @param c character to check if it is a letter
 89       * @return true if is letter, false if not
 90       */
 91      private boolean isLetter(char c)
 92      {
 93          char[] letters = {'a','b','c','d','e','f','g
    ','h','i','j',
 94                  'k','l','m','n','o','p','q','r','s',
    't','u','v','w','x','y','z'};
 95          char[] capitals = {'A','B','C','D','E','F','
    G','H','I','J',
 96                  'K','L','M','N','O','P','Q','R','S',
    'T','U','V','W','X','Y','Z'};
 97          for(int i = 0; i< letters.length; i++)
 98          {
 99              if(c == letters[i] || c == capitals[i])
100              {
101                  return true;
102              }
103          }
104          return false;
105      }
106
107      /**
108       * Returns the frequency of a given word
109       * @param word word to find frequency of
110       * @return frequency of word
111       */
```

```
112     public int getFrequency(String word){
113         WCEntry toGetFreq = new WCEntry(word);
114         if(BST.search(toGetFreq))
115         {
116             WCEntry entry = BST.grabData(toGetFreq);
117             return entry.getFrequency();
118         }
119         else{
120             return 0;
121         }
122 
123     }
124 
125     /**
126      * toString
127      * For example,
128      *  computer: 3
129      *  jail: 2
130      * @return returns words and their frequencies
    as a printable String
131      */
132     public String toString(){
133         return BST.toString();
134     }
135 }
136 
```

```java
package proj5;
/**
 * WordCounterEntry Test Class
 *
 * Author: Theo Scola
 * Ver: 6/5/25
 *
 */
import org.junit.*;
import org.junit.After;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

public class WCEntryTests {
    WCEntry WCE1;
    WCEntry WCE2;

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

    @After
    public void tearDown(){
        WCE1 = null;
        WCE2 = null;
    }

    @Test
    public void testCompareTo0()
    {
        WCE1 = new WCEntry("happy");
        WCE2 = new WCEntry("sad");
        assertTrue("testing compareTo returns
correctly",  WCE1.compareTo(WCE2) < 0);
    }

    @Test
    public void testCompareTo1()
    {
        WCE1 = new WCEntry("zinc");
        String word = "gang";
        assertTrue("testing compareTo when inputting
```

```
40 a string", WCE1.compareTo((Object)word) > 0);
41     }
42
43     @Test
44     public void testToString0()
45     {
46         WCE1 = new WCEntry("burger");
47         assertEquals("testing toString", "burger: 1"
   , WCE1.toString());
48     }
49
50 }
51
```

```java
package proj5;
/**
 * GrammarChecker Class: Uses a thesaurus and word frequencies to
 *  replace overused words in a text document with random synonyms.
 *
 * Author: Theo Scola
 * Ver: 6/3/25
 *  I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 *  accordance with the Union College Honor Code and the course syllabus.
 *
 */
public class GrammarChecker {
    private Thesaurus T;
    private WordCounter WC;
    private int maximum;


    /**
     * Non-default constructor
     * @param thesaurusFile file to feed to the thesaurus
     * @param threshold maximum frequency of words that the get replaced
     */
    public GrammarChecker(String thesaurusFile, int threshold)
    {
        T = new Thesaurus(thesaurusFile);
        WC = new WordCounter();
        maximum = threshold;
    }

    /**
     * Given a text file, replaces overused words with synonyms.
     * Finished text is printed to the console.
     * @param textfile file to improve
```

```
34      */
35     public void improveGrammar(String textfile)
36     {
37         WC.findFrequencies(textfile);
38         LineReader LR = new LineReader(textfile, " "
   );
39 
40         String[] words = LR.getNextLine();
41         String toPrint = "";
42         String punctuation;
43         while(words != null) {
44             for (int i = 0; i < words.length; i++) {
45                 if(isLetter(words[i].charAt(0))) {
46                     punctuation = "";
47                     char lastChar = words[i].charAt(
   words[i].length()-1);
48                     if(!isLetter(lastChar)) {
49                         punctuation = String.valueOf(
   lastChar);
50                         words[i] = stripPunctuation(
   words[i]);
51                     }
52                     if(WC.getFrequency(words[i]) >
   maximum)
53                     {
54                         if(T.contains(words[i]))
55                         {
56                             words[i] = T.
   getSynonymFor(words[i]);
57                         }
58                         toPrint += words[i] +
   punctuation + " ";
59                     }
60                     else{
61                         toPrint += words[i] +
   punctuation + " ";
62                     }
63                 }
64                 else{
65                     toPrint += words[i] + " ";
66                 }
```

```
 67              }
 68              words = LR.getNextLine();
 69          }
 70          System.out.println(toPrint);
 71
 72      }
 73
 74      /**
 75       * Helper for grammar checker
 76       * @param unStripped String to potentially strip
 77       * @return stripped or unchanged string
 78       */
 79      private String stripPunctuation(String
    unStripped)
 80      {
 81          int length = unStripped.length();
 82          char lastChar = unStripped.charAt(length-1);
 83          if(isLetter(lastChar))
 84          {
 85              return unStripped;
 86          }
 87          else{
 88              String toReturn = "";
 89              for(int i =0 ; i< length-1; i++)
 90              {
 91                  toReturn+=unStripped.charAt(i);
 92              }
 93              return toReturn;
 94          }
 95      }
 96
 97      /**
 98       * Helper for stripPunctuation
 99       * @param c character to check if it is a letter
100       * @return true if is letter, false if not
101       */
102      private boolean isLetter(char c)
103      {
104          char[] letters = {'a','b','c','d','e','f','g
    ','h','i','j',
105                  'k','l','m','n','o','p','q','r','s',
```

```
105 't','u','v','w','x','y','z'};
106         char[] capitals = {'A','B','C','D','E','F','
    G','H','I','J',
107                 'K','L','M','N','O','P','Q','R','S',
    'T','U','V','W','X','Y','Z'};
108         for(int i = 0; i< letters.length; i++)
109         {
110             if(c == letters[i] || c == capitals[i])
111             {
112                 return true;
113             }
114         }
115         return false;
116     }
117 }
118
```

```java
package proj5;
/**
 * StringBag Test Class
 *
 * Author: Theo Scola
 * Ver: 6/5/25
 * Note: can't predict the output of some of the functions,
 * so just checked size updates for some of the functions
 */
import org.junit.*;
import org.junit.After;
import org.junit.Before;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

public class StringBagTests {

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

    @Before
    public void setUp() throws Exception{
        bag = new StringBag();
    }

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

    @Test
    public void testConstruct()
    {
        assertNotNull("testing default constructor constructs a bag",bag);
    }

    @Test
```

```
39      public void testInsert0()
40      {
41          bag.insert("A");
42          assertEquals("testing insert increases size"
    , 1, bag.size());
43      }
44      @Test
45      public void testInsert1()
46      {
47          bag.insert("A");
48          bag.insert("B");
49          assertEquals("testing insert twice works as
    intended", 2, bag.size());
50      }
51
52      @Test
53      public void testGrab0()
54      {
55          assertNull("testing grab when bag is empty
    returns null", bag.grab());
56      }
57      @Test
58      public void testGrab1()
59      {
60          bag.insert("A");
61          assertEquals("testing grab will return an
    item in the bag", "A", bag.grab());
62      }
63
64      @Test
65      public void testGrab2()
66      {
67          bag.insert("A");
68          bag.insert("B");
69          bag.grab();
70          assertEquals("testing grab does not change
    the bag size", 2, bag.size());
71      }
72
73      @Test
74      public void testRemove0()
```

```
 75     {
 76         assertNull("testing remove when empty
    returns null", bag.remove());
 77     }
 78 
 79     @Test
 80     public void testRemove1()
 81     {
 82         bag.insert("A");
 83         assertEquals("testing remove returns correct
     value", "A", bag.remove());
 84     }
 85 
 86     @Test
 87     public void testRemove2()
 88     {
 89         bag.insert("A");
 90         bag.insert("B");
 91         bag.remove();
 92         assertEquals("testing remove changes size",
    1, bag.size());
 93     }
 94 
 95     @Test
 96     public void testEquals0()
 97     {
 98         StringBag bag2 = new StringBag();
 99         assertTrue("testing that two empty bags are
    equal",bag.equals(bag2));
100     }
101 
102     @Test
103     public void testEquals1()
104     {
105         StringBag bag2 = new StringBag();
106         bag.insert("A");
107         bag2.insert("A");
108         assertTrue("testing bags of size 1 are equal
    ", bag.equals(bag2));
109     }
110 
```

```
111     @Test
112     public void testEquals2()
113     {
114         StringBag bag2 = new StringBag();
115         bag.insert("A");
116         bag.insert("B");
117         bag2.insert("A");
118         bag2.insert("B");
119         assertTrue("testing bags of size2 are equal"
    , bag.equals(bag2));
120     }
121 
122     @Test
123     public void testEquals3()
124     {
125         StringBag bag2 = new StringBag();
126         bag.insert("A");
127         bag.insert("B");
128         bag2.insert("A");
129         bag2.insert("C");
130         assertFalse("testing unequal bags are not
    equal", bag.equals(bag2));
131     }
132 
133     @Test
134     public void testClone0()
135     {
136         bag.insert("A");
137         StringBag clone = bag.clone();
138         assertTrue("testing clone is equal to bag",
    bag.equals(clone));
139     }
140     @Test
141     public void testClone1()
142     {
143         bag.insert("A");
144         bag.insert("B");
145         StringBag clone = bag.clone();
146         clone.remove();
147         assertFalse("testing changes to clone don't
    affect bag", bag.equals(clone));
```

```
148     }
149     @Test
150     public void testClone2()
151     {
152         bag.insert("A");
153         bag.insert("B");
154         StringBag clone = bag.clone();
155         bag.remove();
156         assertFalse("testing changes to bag don't
    affect clone", bag.equals(clone));
157 
158     }
159 
160     @Test
161     public void testAddAll0()
162     {
163         StringBag bag2 = new StringBag();
164         bag.insert("A");
165         bag.insert("B");
166         bag2.insert("A");
167         bag2.insert("C");
168         bag.addAll(bag2);
169         assertEquals("testing addAll increases size
    of bag from 2 to 4", 4, bag.size());
170     }
171 
172     @Test
173     public void testAddAll1()
174     {
175         StringBag bag2 = new StringBag();
176         bag.insert("A");
177         bag.insert("B");
178         bag2.insert("A");
179         bag2.insert("C");
180         StringBag clone = bag2.clone();
181         bag.addAll(bag2);
182         assertTrue("testing no changes to bag2 from
    addAll", bag2.equals(clone));
183     }
184 
185     @Test
```

```
186     public void testToString0()
187     {
188         assertEquals("test empty toString", "{}",
    bag.toString());
189     }
190 
191     @Test
192     public void testToString1()
193     {
194         bag.insert("A");
195         assertEquals("testing toString with 1
    element", "{A}", bag.toString());
196     }
197 }
198 
```

```java
package proj5;
/**
 * Thesaurus Entry Class: represents what is inputted into the Thesaurus ADT.
 * Holds data in a String keyword and a StringBag of synonyms.
 * Author: Theo Scola
 * Ver: 6/5/2025
 */
public class ThesaurusEntry implements Comparable{
    /**
     * INVARIANT:
     *  keyword: Main word of the TE. Used for comparing.
     *  sys: StringBag that holds all the synonyms of keyword.(See StringBag Invariant)
     */
    private String keyword;
    private StringBag syns;

    /**
     * Non-Default constructor
     * @param newKey keyword
     * @param newSyns associated synonyms
     */
    public ThesaurusEntry(String newKey, StringBag newSyns)
    {
        keyword = newKey;
        syns = newSyns;
    }

    /**
     * Non-default constructor (for just a keyword)
     * @param newKey keyword
     */
    public ThesaurusEntry(String newKey)
    {
        keyword = newKey;
        syns = null;
    }
```

```
37
38      /**
39       * CompareTo method -- compares strings lexicographically
40       * @param other the object to be compared.
41       * @return pos if this greater than other, neg if other greater than this, 0 if equal
42       */
43      public int compareTo(Object other)
44      {
45          if(other.getClass() == ThesaurusEntry.class)
46          {
47              String otherKey = ((ThesaurusEntry)other).getKey();
48              return keyword.compareTo(otherKey);
49          }
50          else if(other.getClass() == String.class){
51              return keyword.compareTo((String)other);
52          }
53          else {
54              return -1;
55          }
56      }
57
58      /**
59       * Getter for the keyword
60       * @return keyword
61       */
62      private String getKey()
63      {
64          return keyword;
65      }
66
67      /**
68       * toString method
69       * Ex: happy - {joyful, gleeful, ecstatic, etc.}
70       * @return string representation of the Thesaurus entry
71       */
72      public String toString()
73      {
```

```
74          return keyword + " - " + syns.toString();
75      }
76
77      /**
78       * Adds synonyms to an existing bag of synonyms
79       * @param newSyns bag to add
80       */
81      public void addSynonyms(StringBag newSyns)
82      {
83          syns.addAll(newSyns);
84
85      }
86
87      /**
88       * Returns a synonym from the synonym bag
89       * @return random synonym
90       */
91      public String getSynonym()
92      {
93          return syns.grab();
94      }
95  }
96
```

```java
package proj5;
/**
 * Thesaurus Test Class
 *
 * Author: Theo Scola
 * Ver: 6/5/25
 *
 * Note: just checked output for some of the tests rather than assert
 */
import org.junit.*;
import org.junit.After;
import org.junit.Before;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

public class ThesaurusTests {
    Thesaurus T;
    @Rule
    public Timeout timeout = Timeout.millis(1000);

    @Before
    public void setUp() throws Exception{
        T = new Thesaurus();
    }

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

    @Test
    public void testDefaultConstructor()
    {
        assertNotNull("Testing that the constructor makes a Thesaurus", T);
    }

    @Test
    public void testNonDefaultConstructor()
    {
```

```
40          Thesaurus newT = new Thesaurus("src/
    smallThesaurus.txt");
41          assertNotNull("Testing that nondefault
    constructor makes a Thesaurus", newT);
42          System.out.println(newT);
43      }
44
45      @Test
46      public void testGeneral()
47      {
48          //Just making sure that it prints correctly
49          String[] syns1 = new String[2];
50          syns1[0] = "gleeful";
51          syns1[1] = "joyful";
52          T.insert("happy",syns1);
53          String[] syns2 = new String[3];
54          syns2[0] = "glum";
55          syns2[1] = "depressing";
56          syns2[2] = "melancholy";
57          T.insert("sad",syns2);
58          System.out.println(T);
59      }
60
61      @Test
62      public void testDelete0()
63      {
64          String[] syns1 = new String[2];
65          syns1[0] = "gleeful";
66          syns1[1] = "joyful";
67          T.insert("happy",syns1);
68          String[] syns2 = new String[3];
69          syns2[0] = "glum";
70          syns2[1] = "depressing";
71          syns2[2] = "melancholy";
72          T.insert("sad",syns2);
73          T.delete("sad");
74          System.out.println(T);
75      }
76
77      @Test
78      public void testInsertWhenExisting()
```

```
 79     {
 80         String[] syns1 = new String[2];
 81         syns1[0] = "gleeful";
 82         syns1[1] = "joyful";
 83         T.insert("happy",syns1);
 84         String[] syns2 = new String[1];
 85         syns2[0] = "joyous";
 86         T.insert("happy", syns2);
 87         System.out.println(T);
 88     }
 89 
 90     @Test
 91     public void testGetSynonymFor0()
 92     {
 93         assertEquals("testing getSynonym for empty
    thesaurus returns empty String", "",T.getSynonymFor(
    "happy"));
 94     }
 95 
 96     @Test
 97     public void testGetSynonymFor1()
 98     {
 99         String[] syns1 = new String[1];
100         syns1[0] = "gleeful";
101         T.insert("happy",syns1);
102         assertEquals("testing getSynonym returns a
    synonym", "gleeful", T.getSynonymFor("happy"));
103     }
104 
105     @Test
106     public void testGetSynonymFor2()
107     {
108         String[] syns1 = new String[1];
109         syns1[0] = "gleeful";
110         T.insert("happy",syns1);
111         assertEquals("testing getSynonym for a non-
    empty Thesaurus with incorrect entry returns empty
    string"
112                 ,"", T.getSynonymFor("sad"));
113     }
114 
```

```
115 }
116
```

```java
package proj5;

import java.lang.annotation.ElementType;

/**
 * BinarySearchTree class
 * Author: Theo Scola
 * Ver: 6/2/25
 */

public class BinarySearchTree<E extends Comparable>
{
    /**
     * INVARIANT:
     *  root: Original BSTNode in the tree. Essentially is the tree.
     *  BST can be changed using insert and delete. BST can be searched for elements
     *  and the elements internal data can be returned.
     */
    private BSTNode<E> root;

    /**
     * Constructor
     */
    public BinarySearchTree() {
        root = null;
    }

    /**
     * inserts recursively.
     *
     * @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<E> insert(BSTNode<E> subroot, BSTNode<E> newNode) {
```

```
36          if (subroot == null) {
37              return newNode;
38          }
39          else if (newNode.compareTo(subroot.data) > 0
    ) {
40              subroot.rlink = insert(subroot.rlink,
    newNode);
41              return subroot;
42          }
43          else {  // newNode.data smaller than subroot.
    data, so newNode goes on left
44              subroot.llink = insert(subroot.llink,
    newNode);
45              return subroot;
46          }
47      }
48
49      /**
50       * inserts recursively.
51       *
52       * @param value E to insert
53       */
54      public void insert(E value){
55          BSTNode<E> newNode = new BSTNode(value);
56          root = insert(root, newNode);
57      }
58
59      /**
60       * Search for specified element
61       * @param target to find
62       * @return true or false if found or not found
63       */
64      public boolean search(E target)
65      {
66          if(size()>0) {
67              return search(target, root);
68          }
69          else{
70              return false;
71          }
72      }
```

```
 73
 74      /**
 75       *Recursive search for specified element
 76       * @param target specified element
 77       * @param subtree Recursive BSTNode
 78       * @return true or false if found or not found
 79       */
 80      private boolean search(E target, BSTNode<E>
    subtree)
 81      {
 82          if(subtree == null)
 83          {
 84              return false;
 85          }
 86          else if(subtree.compareTo(target) == 0) {
 87              return true;
 88          }
 89          else{
 90              if(subtree.compareTo(target)>0)
 91              {
 92                  return search(target, subtree.llink
    );
 93              }
 94              else{
 95                  return search(target, subtree.rlink
    );
 96              }
 97          }
 98      }
 99
100      /**
101       * toString method
102       * @return a String representation of the BST
103       */
104      public String toString()
105      {
106          return toString(root);
107      }
108
109      /**
110       * Recursive toString
```

```
111      * @param subroot Recursive BSTNode
112      * @return a String representation of the BST
113      */
114     private String toString(BSTNode<E> subroot)
115     {
116         if(subroot == null)
117         {
118             return "";
119         }
120         else{
121             String toReturn = "";
122             toReturn += toString(subroot.llink);
123             toReturn += subroot.toString() + "\n";
124             toReturn += toString(subroot.rlink);
125             return toReturn;
126         }
127     }
128 
129     /**
130      * Returns the size of the BST
131      * @return int representing how many elements
    are in BST
132      */
133     public int size()
134     {
135         return size(root);
136     }
137 
138     /**
139      * Returns the size of the BST
140      * @param subroot Recursive BSTNode
141      * @return int representing how many elements
    are in BST
142      */
143     private int size(BSTNode<E> subroot)
144     {
145         if(subroot == null)
146         {
147             return 0;
148         }
149         else{
```

```
150             int size = 1;
151             size += size(subroot.llink);
152             size += size(subroot.rlink);
153             return size;
154         }
155     }
156 
157     /**
158      * Recursive delete
159      * @param value value to be deleted
160      */
161     public void delete(E value)
162     {
163         root = delete(root, value);
164     }
165 
166     /**
167      * Recursive Delete
168      * @param subroot recursive BSTNode
169      * @param value to be deleted
170      * @return a BSTNode that represents the updated
     tree
171      */
172     private BSTNode<E> delete(BSTNode<E> subroot, E
    value){
173         if(subroot == null)
174         {
175             return null;
176         }
177         else if(subroot.compareTo(value) > 0)
178         {
179             subroot.llink = delete(subroot.llink,
    value);
180             return subroot;
181         }
182         else if(subroot.compareTo(value) < 0) {
183             subroot.rlink = delete(subroot.rlink,
    value);
184             return subroot;
185         }
186         else{
```

```
187             if(subroot.isLeaf())
188             {
189                 return null;
190             }
191             else if(subroot.rlink != null && subroot
    .llink == null)
192             {
193                 return subroot.rlink;
194             }
195             else if(subroot.llink != null && subroot
    .rlink == null)
196             {
197                 return subroot.llink;
198             }
199             else{
200                 BSTNode<E> replacement =
    findSuccessor(subroot.rlink);
201                 subroot.data = replacement.data;
202                 subroot.rlink = delete(subroot.rlink
    , replacement.data);
203                 return subroot;
204             }
205         }
206     }
207 
208     /**
209      * Helper method for Delete.
210      * Finds the successor for the deleted node if
    the node has two children
211      * @param subroot recursive BSTNode
212      * @return the BSTNode for the successor
213      */
214     private BSTNode<E> findSuccessor(BSTNode<E>
    subroot)
215     {
216         if(subroot.llink == null)
217         {
218             return subroot;
219         }
220         else{
221             return findSuccessor(subroot.llink);
```

```
222         }
223     }
224 
225     /**
226      * Recursive method
227      * Essentially Search, but returns the data rather than a boolean
228      * (only called after target has been confirmed to be in the BST,
229      *  don't want to call search again in this method)
230      * @param target value to return
231      * @return data of target
232      */
233     public E grabData(E target)
234     {
235         return grabData(target, root);
236     }
237 
238     /**
239      * Recursive method
240      * Essentially Search, but returns the data rather than a boolean
241      * @param target value to return
242      * @param subtree recursive BSTNode
243      * @return data of target
244      */
245     private E grabData(E target,BSTNode<E> subtree)
246     {
247         if(subtree == null)
248         {
249             return null;
250         }
251         else if(subtree.compareTo(target) == 0) {
252             return subtree.data;
253         }
254         else{
255             if(subtree.compareTo(target)>0)
256             {
257                 return grabData(target, subtree.llink);
```

```
258             }
259             else{
260                 return grabData(target, subtree.
    rlink);
261             }
262         }
263     }
264 
265 
266     /** recursive helper for toStringParen
267      *
268      * @param subroot root of subtree to start at
269      * @return inorder string of elements in this
    subtree
270      */
271     private String toStringParen(BSTNode subroot) {
272       if (subroot == null) // base case
273           return "";
274       else
275           return "(" + toStringParen(subroot.llink
    ) + " " +
276                   subroot.toString() + " " +
    toStringParen(subroot.rlink) + ")";
277     }
278 
279     /**
280      * returns string showing tree structure using
    parentheses, as shown in class
281      */
282     public String toStringParen() {
283       return toStringParen(root);
284     }
285 }
```

```java
package proj5;  // Gradescope needs this.
/**
 * The doubly linked list class gives you access to the beginning and end of a linked
 * list through instance variables firstNode and lastNode.  This class
 * should contain all the methods for general manipulation of linked lists:
 * traversal, insertion, deletion, searching, etc.
 *
 * @author Theo Scola
 * @version 05/10/2025
 */
public class DoublyLinkedList
{
    private int length;          // number of nodes
    private ListNode firstNode;  // pointer to first node
    private ListNode lastNode;  // pointer to last node

    /**
     * Default Constructor
     */
    public DoublyLinkedList()
    {
        length = 0;
        firstNode = null;
        lastNode = null;
    }

    /**
     * Inserts a new ListNode at the beginning of the LinkedList
     * @param newData data of the new ListNode
     */
    public void insertAtHead(String newData)
    {
        ListNode newNode = new ListNode(newData);
        if(!(isEmpty())) {
            firstNode.prev = newNode;
```

```
36              newNode.next = firstNode;
37              firstNode = newNode;
38              length++;
39          }
40          else{
41              firstNode= newNode;
42              lastNode = newNode;
43              length++;
44          }
45      }
46
47      /**
48       * Inserts a new ListNode at the end of the
   DoublyLinkedList
49       * @param newData data for the new ListNode
50       */
51      public void insertAtTail(String newData)
52      {
53          ListNode newNode = new ListNode(newData);
54          if(!(isEmpty()))
55          {
56              lastNode.next = newNode;
57              newNode.prev = lastNode;
58              lastNode = newNode;
59              length++;
60          }
61          else{
62              firstNode = newNode;
63              lastNode = newNode;
64              length++;
65          }
66      }
67
68      /**
69       * Removes the first ListNode in the
   DoublyLinkedList
70       * @return the String of the data in the ListNode
    or null if the List is empty
71       */
72      public String removeHead()
73      {
```

```
 74          if(!(isEmpty()) && getLength() > 1) {
 75              String toRemove = firstNode.data;
 76              firstNode = firstNode.next;
 77              firstNode.prev = null;
 78              length--;
 79              return toRemove;
 80          }
 81          else if(!(isEmpty()))
 82          {
 83              String toRemove = firstNode.data;
 84              firstNode = null;
 85              lastNode = null;
 86              length--;
 87              return toRemove;
 88          }
 89          else{
 90              return null;
 91          }
 92
 93      }
 94
 95      /**
 96       * Removes the ListNode at the end of the
    DoublyLinkedList
 97       * @return the data of the removed node or null
    if the List is empty
 98       */
 99      public String removeTail()
100      {
101         if(!(isEmpty()) && getLength() >1)
102         {
103             String toRemove = lastNode.data;
104             lastNode = lastNode.prev;
105             lastNode.next = null;
106             length--;
107             return toRemove;
108
109         }
110         else if(!(isEmpty()))
111         {
112             String toRemove = lastNode.data;
```

```
113             lastNode = null;
114             firstNode = null;
115             length--;
116             return toRemove;
117         }
118         else{
119             return null;
120         }
121     }
122 
123     /**
124      * search for first occurrence of value and
    return index where found
125      *
126      * @param value String to search for
127      * @return index where value occurs (first node
    is index 0).  Return -1 if value not found.
128      */
129     public int indexOf(String value) {
130         length = getLength();
131         if(!(isEmpty())){
132             ListNode runner = firstNode;
133             for(int i = 0; i<length; i++)
134             {
135                 if(runner.data.equals(value))
136                 {
137                     return i;
138                 }
139                 runner = runner.next;
140             }
141         }
142         return -1;
143     }
144 
145     /**
146      * Returns a string representation of the
    DoublyLinkedList
147      * @return toReturn, a string of the whole list
    formatted "(A,B,C)"
148      */
149     public String toString()
```

```
150     {
151         String toReturn="(";
152         ListNode runner=firstNode;
153         while (runner!=null)
154         {
155             toReturn = toReturn + runner;
156             runner=runner.next;
157             if (runner!=null)
158             {
159                 toReturn = toReturn + ",";
160             }
161         }
162         toReturn = toReturn + ")";
163         return toReturn;
164     }
165 
166     /**
167      * getter for the length of the DoublyLinkedList
168      * @return length, the amount of items in the
    DoublyLinkedList
169      */
170     public int getLength() { return length;}
171 
172     /**
173      *
174      * @return true if DLL empty or false if not
175      */
176     public boolean isEmpty() {return getLength()==0
    ;}
177 
178 
179     /**
180      * getter
181      * @return the string for the node at index,
182      * return null if empty or if index less than
    zero/greater than or equal to length
183      */
184     public String getIthData(int index) {
185         if(!(isEmpty()) && index < getLength() &&
    index >= 0)
186         {
```

```
187             ListNode runner = firstNode;
188             for(int i = 0; i<index; i++)
189             {
190                 runner = runner.next;
191             }
192             return runner.data;
193         }
194         else{
195             return null;
196         }
197     }
198 }
```

```java
package proj5;
/**
 * GrammarChecker Test Class
 *
 * Author: Theo Scola
 * Ver: 6/5/25
 * Note: For the improveGrammar tests,
 * I just print them to check the output rather than try to use an assert function
 */

import org.junit.*;
import org.junit.After;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

public class GrammarCheckerTests {
    GrammarChecker GC;

    @Rule
    public Timeout timeout = Timeout.millis(1000);

    @After
    public void tearDown()
    {
        GC = null;
    }

    @Test
    public void testConstruct()
    {
        GC = new GrammarChecker("src/smallThesaurus.txt", 3);
        assertNotNull("testing constructor constructs", GC);
    }

    @Test
    public void testImproveGrammar0()
    {
        GC = new GrammarChecker("src/smallThesaurus.
```

```
38 txt", 3);
39          GC.improveGrammar("src/apartment.txt");
40      }
41
42      @Test
43      public void testImproveGrammar1()
44      {
45          GC = new GrammarChecker("src/bigThesaurus.txt
   ", 3);
46          GC.improveGrammar("src/apartment.txt");
47      }
48
49      @Test
50      public void testImproveGrammar2()
51      {
52          GC = new GrammarChecker("src/bigThesaurus.txt
   ", 3);
53          GC.improveGrammar("src/lamb.txt");
54      }
55 }
56
```