```java
package proj5;

/**
 * Run the program to check the output of grammar checker
 */
public class Client {
    public static void main(String[] arg){
        System.out.println("Original:");
        System.out.println();
        GrammarChecker test = new GrammarChecker("src/smallThesaurus.txt", 10
);
        test.improveGrammar("src/grammarTest.txt");
        System.out.println();
        System.out.println("Change threshold should replace all:");
        System.out.println();
        GrammarChecker test2 = new GrammarChecker("src/smallThesaurus.txt", 8
);
        test2.improveGrammar("src/grammarTest.txt");
        System.out.println();
        System.out.println("Output of two given text:");
        GrammarChecker apartment = new GrammarChecker("src/smallThesaurus.txt
", 2);
        apartment.improveGrammar("src/apartment.txt");
        System.out.println();
        GrammarChecker lamb = new GrammarChecker("src/bigThesaurus.txt", 3);
        lamb.improveGrammar("src/lamb.txt");
        System.out.println();
        System.out.println("Test upperCase: output should maintain the upper
case whether they are improved grammar");
        System.out.println();
        GrammarChecker test3 = new GrammarChecker("src/bigThesaurus.txt",3);
        test3.improveGrammar("src/upperCase.txt");
        System.out.println();
        System.out.println("Test punctuation: output should still include
punctuation like comma, period");
        System.out.println();
        GrammarChecker test4 = new GrammarChecker("src/smallThesaurus.txt",3
);
        test4.improveGrammar("src/punctuationTest.txt");

    }
}
```

```java
package proj5;

/**
 * A generic BSTNode in a BST with instance variable key as the value of the
node and left, right link of that node
 * @param <T>
 */
public class BSTNode<T> {
    public T key;
    public BSTNode<T> llink;
    public BSTNode<T> rlink;

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

    }

    /**
     * check if the node is the leaf (last item with no children) in a BST
     * @return true if both left and right link point to null; else false
     */
    public boolean isLeaf() {
        return this.llink == null && this.rlink == null;
    }

    /**
     * check if the node has only one child on the right
     * @return true if the left link points to null but the right link points
 to another node; else false
     */
    public boolean hasRightChildOnly() {
        return this.llink == null && this.rlink != null;
    }


    /**
     * check if the node has only one child on the left
     * @return true if the right link points to null but the left link points
 to another node; else false
     */

    public boolean hasLeftChildOnly() {
        return this.llink != null && this.rlink == null;
    }


    /**
```

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

```java
package proj5;
/**
 * @author: Emma Vu
 * @version: 5/29/2020
 * JUnit test class.  Use these tests as models for your own.
 */
import org.junit.*;


import org.junit.rules.Timeout;

import static org.junit.Assert.*;



public class BSTTest {
    @Rule

    public Timeout timeout = Timeout.millis(100);

    private BinarySearchTree<String> bst;

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

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

    @Test
    public void testSearchEmpty(){
        assertNull(bst.search("A"));
    }

    @Test
    public void testSearchAtRoot(){
        bst.insert("A");
        assertEquals("A",bst.search("A"));
    }

    @Test
    public void testSearchNonExist(){
        bst.insert("A");
        bst.insert("B");
        bst.insert("C");
        assertNull(bst.search("D"));
    }

    @Test
    public void testSearchLeft(){
```

```
    bst.insert("B");
    bst.insert("A");
    bst.insert("C");
    assertEquals("A",bst.search("A"));
}

@Test
public void testSearchRight(){
    bst.insert("B");
    bst.insert("A");
    bst.insert("C");
    assertEquals("C",bst.search("C"));
}

@Test
public void testDeleteEmpty(){
    bst.delete("A");
    assertEquals("",bst.toString());
}
@Test
public void testDeleteAtRoot(){
    bst.insert("A");
    bst.delete("A");
    assertEquals("",bst.toString());
}
@Test
public void testDeleteNonExist(){
    bst.insert("A");
    bst.insert("B");
    bst.insert("C");
    bst.delete("D");
    assertEquals("( A ( B ( C )))",bst.toString());
}

@Test
public void testDeleteLeaf(){
    bst.insert("B");
    bst.insert("A");
    bst.insert("C");
    bst.insert("E");
    bst.insert("D");
    bst.insert("F");
    bst.delete("D");
    assertEquals("(( A ) B ( C ( E ( F ))))",bst.toString());


}

@Test
public void testDeleteOneChild(){
    bst.insert("B");
    bst.insert("A");
    bst.insert("C");
```

```
        bst.insert("E");
        bst.insert("D");
        bst.insert("F");
        bst.delete("C");
        assertEquals("(( A ) B (( D ) E ( F )))",bst.toString());
    }

    @Test
    public void testDeleteTwoChildren(){
        bst.insert("B");
        bst.insert("A");
        bst.insert("C");
        bst.insert("E");
        bst.insert("D");
        bst.insert("F");
        bst.delete("E");
       assertEquals("(( A ) B ( C (( D ) F )))",bst.toString());
    }

    @Test
    public void testToStringOrder(){
        bst.insert("B");
        bst.insert("A");
        bst.insert("C");
        bst.insert("E");
        bst.insert("D");
        bst.insert("F");
        assertEquals("A\nB\nC\nD\nE\nF\n",bst.toStringOrder());
    }

    @Test
    public void testToStringEmpty(){
        assertEquals("",bst.toString());
    }

    @Test
    public void testToStringOrderEmpty(){
        assertEquals("",bst.toStringOrder());
    }




}
```

```java
package proj5;

/**
 * Data structure that holds words and their associated synonyms. You can
look up a word and retrieve a synonym for it.
 * The thesaurus gets searched for each overused word found.
 * Thesaurus class for holding all of the entries and their synonyms
 * CLASS SPECIFICATION: This class is implemented using a BST holding
ThesaurusInfo
 * Each ThesaurusInfo represents an entry and its correspond synonyms.
 */
public class Thesaurus {
    private BinarySearchTree<ThesaurusInfo> thesaurus;
    private LineReader reader;
    private String[] currentLine;

    //default constructor to create an empty thesaurus
    public Thesaurus(){
        thesaurus = new BinarySearchTree<ThesaurusInfo>();
        reader = null;
        currentLine = null;
        buildThesaurus();

    }

    /**
     * non-default constructor
     * Builds a thesaurus from a text file. Each line of the text file is a
comma-separated list of synonymous words.
     * The first word in each line should be the thesaurus entry.
     * The remaining words on that line are the list of synonyms for the
entry.
     * @param file path to comma-delimited text file
     */
    public Thesaurus(String file){
        thesaurus = new BinarySearchTree<ThesaurusInfo>();
        reader = new LineReader(file,",");
        currentLine = reader.getNextLine();
        buildThesaurus();

    }

    /**
     * A private helper method to build the thesaurus by inserting the first
word as a new entry and copying the rest
     * as synonyms. Insert in the thesaurus line by line
     */
    private void buildThesaurus(){
        while(currentLine!=null){
            thesaurus.insert(new ThesaurusInfo(currentLine[0]));
            insert(currentLine[0],copy(currentLine,1,currentLine.length));
            currentLine = reader.getNextLine();
        }
```

```
    }



    /**
     * inserts entry and synonyms into thesaurus.
     * If entry does not exist, it creates one.
     * If it does exist, it adds the given synonyms to the entry's synonym
list
     * @param entry keyword to be added
     * @param syns array of synonyms for keyword entry
     */
    public void insert(String entry, String[] syns){
        ThesaurusInfo entryInfo = new ThesaurusInfo(entry);

        if(!isInThesaurus(entry)){
            thesaurus.insert(entryInfo);
            addSynsOfExistEntryToThesaurus(entry, syns);

        }
        else{

            addSynsOfExistEntryToThesaurus(entry, syns);

        }



    }

    /**
     * private helper method to add the synonyms of an exist entry to the
thesaurus
     * @param entry the keyword in the thesaurus
     * @param syns the list of the synonyms to add to the thesaurus
     */

    private void addSynsOfExistEntryToThesaurus(String entry, String [] syns
){
        ThesaurusInfo content = thesaurus.search(new ThesaurusInfo(entry));
        int count = 0;
        for(int i = 0; i < syns.length; i++){
            content.addSyns(syns[count]);
            count++;
        }

    }

    /**
     * removes entry (and its associated synonym list) from this thesaurus.
If entry does not exist, do nothing.
     * @param entry word to remove
     */
```

```
    public void delete(String entry){
        thesaurus.delete(new ThesaurusInfo(entry));

    }

    /**
     * Gets a random synonym for the given keyword. If keyword does not exist
, return the empty string
     * @param keyword word to find a synonym for
     * @return a random synonym from the synonym list of that word, or empty
string if keyword doesn't exist.
     */
    public String getSynonymFor(String keyword){
        ThesaurusInfo synsList = thesaurus.search(new ThesaurusInfo(keyword
));
        if(!isInThesaurus(keyword)){
            return "";
        }
        else {
            return synsList.getSyns();
        }

    }

    /**
     * Each keyword and synonym list should be on its own line.
     * The format of each line is: <keyword> - {<syn1>, <syn2>, ..., <synN>}
     * For example,
     * happy - {glad, content, joyful}
     * jump - {leap, bound}
     * The thesaurus keywords will be in alphabetical order. The order of the
 synonym list words is arbitrary.
     * @return this thesaurus as a printable string
     */
    public String toString(){
        return thesaurus.toStringOrder();
    }

    /**
     * check whether a keyword is in the thesaurus.
     * @param entry the keyword to check
     * @return true if the keyword is in the thesaurus. Else, false
     */
    public boolean isInThesaurus(String entry){
        if(thesaurus == null){
            return false;
        }
        ThesaurusInfo content = thesaurus.search(new ThesaurusInfo(entry));
        if(content == null){
            return false;
        } else return true;
    }
```

```
    /**
     * private helper method to copy a certain part of an array to a new
array from index start to index end excluding
     * @param original the original array want to copy from
     * @param start the starting index
     * @param end the ending index
     * @return the new array with certain elements similar to the original
array with corresponding indexes
     */
    private String[] copy(String[] original, int start, int end){
        int length = end - start;
        String[] newArray = new String[length];
        int count=0;
        for(int i=start;i<end;i++){
            newArray[count]=original[i];
            count++;
        }
        return newArray;


    }
}
```

```java
package proj5;

/**
 * This Bag class uses both generics and interfaces!
 * Reason: you get to leave types unspecified AND get control
 * over what is baggable and what is not.
 * change to Comparable Bag to holder comparable wrapper
 *
 * @author Chris Fernandes
 * @version 5/3/12
 * @updated version 6/4/2020
 *
 ************************************************************************/
public class GenericBag<T extends Comparable> {

    private int size;           //# of elements in bag
    private Comparable[] contents;
    private final int INITIAL_CAPACITY = 10;

    /**
     * non default constructor of a generic bag. If the capacity is negative
, set the capacity = INITIAL_CAPACITY = 10
     * @param capacity
     */
    public GenericBag(int capacity)
    {
        if(capacity < 0){
            capacity = INITIAL_CAPACITY;
        }
        contents = new Comparable[capacity];
        size=0;
    }

    /**
     * default constructor of a generic bag with default capacity = 10
     */
    public GenericBag(){
        contents = new Comparable[INITIAL_CAPACITY];
        size = 0;
    }

    /**
     * add item to bag
     * @param value the item to add
     */
    public void add(T value)
    {
        if(size() == capacity())
                this.growDouble();

        contents[size]=value;
        size++;
    }
```

```
/**
 * remove item from bag
 * @param value the item to remove
 */
public void remove(T value)
{

    int found_position = find(value);
    if (found_position>-1)
    {
        // move last element to fill the hole
        contents[found_position]=contents[size-1];
        size--;
    }
}

/**
 * does bag contain item?
 * @param value item to search for
 * @return true if bag contains item, else false
 */
public boolean contains(T value)
{
    if (find(value) == -1)
        return false;
    else
        return true;
}

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

/**
 * empty bag of contents
 */
public void clear()
{
    size=0;
}

/**
 *
 * @return number of items in bag
 */
```

```
    public int size()
    {
        return size;
    }

    /**
     * return bag contents as printable string
     */
    public String toString()
    {
        String answer = "{";
        int currentSize=this.size();
        for(int i=0; i < (currentSize - 1); i++)  // take care of all but
last one
        {
            answer = answer + contents[i] + ", ";
        }
        if (currentSize>0)  // as long as not empty
        answer = answer + contents[(currentSize - 1)];
        answer+= "}";
        return answer;
    }

    /**
     * Getter for bag capacity
     * @return how many items bag can hold
     */
    public int capacity()
    {
        return contents.length;
    }

    /**
     * remove a random element from the bag
     * @return the random element removed
     */
    public T removeRandom()
    {
        T toReturn = grabRandom();
        remove(toReturn);
        return toReturn;
    }

    /**
     * grab a random element from the bag
     * @return the element grabbed
     */
    @SuppressWarnings("unchecked")
    public T grabRandom()
    {
        int rand = (int)(Math.random()*this.size());
        T toReturn = (T)contents[rand];
        return toReturn;
```

```
}

/**
 *
 * @param otherBag another bag of the same type of item
 * @return true if this bag has same elements as otherBag.
 * Order doesn't matter.
 */
public boolean equals(GenericBag<T> otherBag)
{
    if (this.size()!=otherBag.size())
        return false;
    else {
        GenericBag<T> thisCopy = this.clone();
        GenericBag<T> otherCopy = otherBag.clone();
        while (!thisCopy.isEmpty()) {
            T someElement = (T) thisCopy.removeRandom();
            if (!otherCopy.contains(someElement))
                return false;
            else
                otherCopy.remove(someElement);
        }
        return true;
    }
}

/**
 * @return exact copy of this bag.  Changes to copy
 * do not affect the original, and vice versa.
 */
public GenericBag<T> clone()
{
    GenericBag<T> newBag = new GenericBag<T>(this.capacity());
    // since you can't make new E instances directly,
    // I'll let you use the array's clone method here.
    newBag.contents = this.contents.clone();
    newBag.size = this.size();
    return newBag;
}

/**
 * makes bag capacity equal to number of items in bag
 */
public void trimToSize()
{
    int currentSize = this.size();
    Comparable[] newContents = new Comparable[currentSize];
    for (int i=0; i<currentSize; i++) {
        newContents[i]=this.get(i);
    }
    contents=newContents;
}
```

```java
/** make new bag contain all elements from this
 *  bag and otherBag.  Return the new bag. this Bag
 *  and otherBag are not altered in the process.
 * @param otherBag the other bag
 * @return the union of this bag and otherBag
 */
public GenericBag<T> union(GenericBag<T> otherBag)
{
    GenericBag<T> newBag = this.clone();
    GenericBag<T> temp = otherBag.clone();
    while (!temp.isEmpty()) {
        newBag.add(temp.removeRandom());
    }
    return newBag;
}

/**
 * return array index where item can be found. Return -1 if not found.
 * @param value the item to search for
 * @return -1 if not found.  Else, the array index where first one found.
 */
private int find(T value)
{
    int currentSize=this.size();
    for (int i=0; i<currentSize; i++)
    {
        if (this.contents[i].equals(value))
            return i;
    }
    return -1;
}

/**
 * double the size of the internal array
 */
private void growDouble()
{
    Comparable[] newArray;
    newArray = new Comparable[(contents.length) * 2];
    int oldSize = this.size();
    for(int i=0; i < oldSize; i++)
    {
        newArray[i] = contents[i];
    }
    this.contents = newArray;
}

/**
 * get the element at index i
 * @param i index of internal array where sought item is stored
 * @return the sought item
 */
@SuppressWarnings("unchecked")
```

```
    private T get(int i)
    {
        return (T)contents[i];
    }
}
```

```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();
    }
}
```

```java
package proj5;

/**
 * class for computing word frequencies from a text file
 * For the word counter, you should build the frequency BST in one pass of
the input file. For each
 * word, search for it in the BST. If you find it, increase its frequency
counter by 1. If you don't find
 * it, insert the brand new word into the tree.
 * This class is implemented using a BST holding FrequencyInfo
 * Each FrequencyInfo represents a string and its correspond frequency.
 */
public class WordCounter {

    private BinarySearchTree<FrequencyInfo> counter;
    private LineReader reader;
    private String[] currentLine;

    public WordCounter(){

        counter = new BinarySearchTree<FrequencyInfo>();
        reader = null;
        currentLine = null;
    }

    /**
     * PRECONDITION: file cannot be empty, or else there would be error
     * Computes frequency of each word in given file
     * SPECIFICATION: Each FrequencyInfo in the tree represents a string and
its correspond frequency.
     * Thus each time we see an old word, we check if there is a
FrequencyInfo in the tree
     * holding that word first, before incrementing the correspond frequency.
     * Computes frequency of each word in given file
     * @param file path to file, such as "src/input.txt"
     *
     *
     */
    public void findFrequencies(String file){

        reader = new LineReader(file, " ");
        currentLine = reader.getNextLine();
        while(currentLine!=null) {
            for (String word : currentLine) {
                String res = keepOnlyLetter(word);
                if(!res.equals("")) {
                    FrequencyInfo content = counter.search(new FrequencyInfo(
res));
                    if (content == null){
                        counter.insert(new FrequencyInfo(res));

                        content = counter.search(new FrequencyInfo(res));
                        content.increaseFrequency();
```

```
                }
                else{
                    content.increaseFrequency();
                }
            }

        }
        currentLine = reader.getNextLine();
    }
}

/**
 * returns the frequency of the given word
 * @param word string to get the frequency of
 * @return the number of times word appears in the input file
 */
public int getFrequency(String word){
    FrequencyInfo frequency = counter.search(new FrequencyInfo(word));
    if(frequency == null){
        return 0;
    }
    return frequency.getFrequency();

}

/**
 * Each word/frequency pair should be on a separate line, and the format
of each line should be <word>: <frequency>
 * For example,
 * are: 3
 * bacon: 2
 * Words should be in alphabetical order.
 * @return words and their frequencies as a printable String.
 */
public String toString(){
    return counter.toStringOrder();
}

/**
 * PRECONDITION: s is not null
 * handle syntax and keep only letter
 * @param s the String to handle
 * @return the handled String stripped of all non letter
 */
private String keepOnlyLetter(String s){
    String finalS= "";
    s=s.toLowerCase();
    for(int i=0;i<s.length();i++){
        char c=s.charAt(i);
        if(Character.isLetter(c)){
            finalS += c;
        }
    }
```

```
        return finalS;
    }


}
```

```java
package proj5;
/**
 * @author Emma Vu
 * @version 5/30/2020
 * A wrapper that contains the content of a word counter (like a node in a
Word Counter BST)
 * CLASS INVARIANT: This class has two instance variable
 * which holds the value of a String, and a correspond occurence in variable
 * count.
 * WHEN COMPARING TWO FREQUENCY INFO, WE ONLY CARE AND COMPARE THE STRING
VALUE
 * NOT THE INT COUNT
 * (because if two Frequency Info has the same value, then its occurence is
by default
 * equal. The class invariant of the compareTo method to be the result of
comparing the
 * two instance variable String value)
 */
public class FrequencyInfo implements Comparable {
    private int frequency;
    private String entry;

    public FrequencyInfo(String keyword){
        entry = keyword;
        frequency = 0;
    }
    /**
     * determine if this FrequencyInfo is greater, less, or equal other
FrequencyInfo
     * look at class invariant for the definition of comparing two
FrequencyInfo.
     * @param other the other FrequencyInfo
     * @return 1 if greater, -1 if smaller, 0 if equal
     */
    public int compareTo(Object other){   return this.entry.compareTo(((
FrequencyInfo) other).entry);}
    /**
     * get the occurence
     * @return count, representing the occurence
     */
    public int getFrequency(){
        return frequency;
    }
    /**
     * increment the occurence by one, adding 1 to instance count
     */
    public void increaseFrequency(){
        frequency ++;
    }
    /**
     * Return the String representing the FrequencyInfo, which is the value
     * following a comma and the count value
     */
```

```
    public String toString(){
        return entry + ": " + frequency;
    }
}
```

```java
package proj5;

/**
 * StringContent is the "wrapper" that will let your bag hold strings
 * It's something that's comparable by a bag
 * Class Invariant : comparing two String content is the same as comparing
two content (unboxing)
 * which justify for the compare method
 *
 * @author Chris Fernandes
 * @version 10/18/08
 */
public class StringContent implements Comparable
{
    private String content;

    /**
     * non default constructor to create a new content from a new string
     * @param newString
     */
    public StringContent(String newString)
    {
        content=newString;
    }

    /**
     * Check if this String content and other are equal.
     * @param other the other object to check
     * @return true if they are the same object or if the contents are equal.
     * If they are not the same class or the parameter is null or the
contents are not equal, return false
     */
    public boolean equals(Object other)
    {
        // if the *pointers* are the same, then by golly it must be the same
object!
        if (this == other)
            return true;

        // if the parameter is null or the two objects are not instances of
the same class,
        // they can't be equal
        else if (other == null || this.getClass() != other.getClass())
            return false;

        //Since this class is what the bag will use if it wants
        //to hold strings, we'll use the equals() method in the
        //String class to check for equality
        else if ((this.content).equals(((StringContent)other).content))
            return true;

        else return false;
    }
```

```
    /**
     * The printable version
     * @return the toString version, which is the content
     */
    //since this is already a string, just return it!
    public String toString()
    {
        return content;
    }

    /**
     * Comparing this StringContent with other
     * @param other the other StringContent to compare to
     * @return 1 if this String's value is greater than the other, -1 if less
, 0 otherwise
     * Precondition: Object must be a StringContent
     */

    public int compareTo(Object other){
        return this.content.compareTo(((StringContent)other).content);
    }

    /**
     * Get the content
     * @return the content, which is a string
     */
    public String getContent(){
        return content;
    }
}
```

```java
package proj5;
/**
 * @author Emma Vu
 * @version 6/4/2020
 * A wrapper that contains the content of a thesaurus (like a node in a
Thesaurus BST)
 * CLASS INVARIANT: This class has two instance variable
 * which holds the value of a String, and a correspond synonyms list in a bag
 * synonyms.
 * WHEN COMPARING TWO ThesaurusInfo, WE ONLY CARE AND COMPARE THE STRING
VALUE
 * NOT THE synonyms list
 * (because if two ThesaurusInfo has the same String entry, then its synonyms
 is by default
 * equal. The class invariant of the compareTo method to be the result of
comparing the
 * two instance variable String entry, ignoring the synonyms)
 */


public class ThesaurusInfo implements Comparable {
    private final int INITIAL_CAPACITY = 10;
    private String entry;
    private GenericBag<StringContent> syns;

    /**
     * Non default constructor to create the content of the thesaurus with a
keyword and a synonym list
     * @param keyword
     */

    public ThesaurusInfo(String keyword){
        entry = keyword;
        syns = new GenericBag<StringContent>(INITIAL_CAPACITY);
    }

    /**
     * determine if this ThesaurusInfo is greater, less, or equal other
ThesaurusInfo
     * look at class invariant for the definition of comparing two
ThesaurusInfo.
     * @param other the other ThesaurusInfo
     * @return 1 if greater, -1 if smaller, 0 if equal
     */

    public int compareTo(Object other){
        return this.entry.compareTo(((ThesaurusInfo) other).entry);
    }

    /**
     * get the entry (keyword) in the thesaurus
     * @return the keyword
     */
```

```
    public String getEntry() {
        return entry;
    }

    /**
     * get a random synonyms of entry
     * @return a random element of synonyms
     */
    public String getSyns(){
        if(!syns.isEmpty()){
            return syns.grabRandom().getContent();
        }
        else return "";
    }

    /**
     * add the new value to synonyms
     * @param value the value to be added
     */
    public void addSyns(String value){
        StringContent valueContent = new StringContent(value);
        syns.add(valueContent);
    }
    /**
     * Return the string value of the ThesaurusInfo, which is represented by
an
     * entry, following by the synonyms list.
     */

    public String toString(){
        return entry + " - " + syns.toString();

    }

}
```

```java
package proj5;

import org.junit.Test;

import static org.junit.Assert.*;

/**
 * test the functionality of Thesaurus
 *
 * @author: Emma Vu
 * @version: 6/5/2020
 */
public class ThesaurusTest {


    private Thesaurus makeDefaultThesaurus() {
        Thesaurus thesaurus = new Thesaurus();
        return thesaurus;
    }

    private Thesaurus makeNonDefaultThesaurus(String file) {
        Thesaurus thesaurus = new Thesaurus(file);
        return thesaurus;
    }

    @Test
    public void testToStringEmpty(){
        Thesaurus empty = makeDefaultThesaurus();
        assertEquals(empty.toString(),"");
    }

    @Test
    public void testToStringAndConstructor(){
        Thesaurus empty = makeNonDefaultThesaurus("src/thesaurusText.txt");
        assertEquals(empty.toString(),"blue - {cyan, azure, cobalt, navy,
aquamarine}\n" +
                "hi - {hello, sup, cia, chao}\n");
    }

    @Test
    public void testInsertNotIn(){
        Thesaurus empty = makeNonDefaultThesaurus("src/thesaurusText.txt");
        empty.insert("tiger",new String[]{"cat","lion","panther"});
        assertEquals(empty.toString(),"blue - {cyan, azure, cobalt, navy,
aquamarine}\n" +
                "hi - {hello, sup, cia, chao}\n" +
                "tiger - {cat, lion, panther}\n");
    }
    @Test
    public void testInsertIn(){
        Thesaurus empty = makeNonDefaultThesaurus("src/thesaurusText.txt");
        empty.insert("blue",new String[]{"xanh"});
        assertEquals(empty.toString(),"blue - {cyan, azure, cobalt, navy,
```

```
aquamarine, xanh}\n" +
                "hi - {hello, sup, cia, chao}\n");
    }

    @Test
    public void testIsInThesaurusEmpty(){
        Thesaurus empty = makeDefaultThesaurus();
        assertFalse(empty.isInThesaurus("hi"));
    }

    @Test
    public void testIsInThesaurus(){
        Thesaurus nonEmpty = makeNonDefaultThesaurus("src/thesaurusText.txt"
);
        assertTrue(nonEmpty.isInThesaurus("blue"));
    }

    @Test
    public void testGetSynonymExist(){
        Thesaurus nonEmpty = makeNonDefaultThesaurus("src/thesaurusText.txt"
);
        assertNotEquals("blue",nonEmpty.getSynonymFor("blue"));
    }

    @Test
    public void testGetSynonymNonExist(){
        Thesaurus nonEmpty = makeNonDefaultThesaurus("src/thesaurusText.txt"
);
        assertEquals("", nonEmpty.getSynonymFor("aabc"));
    }

    @Test
    public void testDeleteExist(){
        Thesaurus nonEmpty = makeNonDefaultThesaurus("src/thesaurusText.txt"
);
        nonEmpty.delete("blue");
        assertEquals(nonEmpty.toString(),"hi - {hello, sup, cia, chao}\n");
    }

    @Test
    public void testDeleteNonExist(){
        Thesaurus nonEmpty = makeNonDefaultThesaurus("src/thesaurusText.txt"
);
        nonEmpty.delete("green");
        assertEquals(nonEmpty.toString(),"blue - {cyan, azure, cobalt, navy,
aquamarine}\n" +
                "hi - {hello, sup, cia, chao}\n");    }



}
```

```java
package proj5;
/**
 * @author: Emma Vu
 * @version: 6/6/2020
 * JUnit test class.  Use these tests as models for your own.
 */
import org.junit.*;


import org.junit.rules.Timeout;

import static org.junit.Assert.*;

public class GenericBagTest {

    @Rule

    public Timeout timeout = Timeout.millis(100);

    private GenericBag<StringContent> genericBag;

    @Before

    public void setUp() throws Exception {
        genericBag = new GenericBag<StringContent>();
    }

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


    @Test
    public void testDefaultConstructor(){
        assertEquals("{}",genericBag.toString());
        assertEquals(10,genericBag.capacity());
        assertEquals(0,genericBag.size());
    }

    @Test
    public void testNonDefaultConstructorValidCap(){
        genericBag = new GenericBag<StringContent>(4);
        assertEquals(4,genericBag.capacity());
        assertEquals("{}",genericBag.toString());

    }

    @Test
    public void testNonDefaultConstructorInvalidCap(){
        genericBag = new GenericBag<>(-2);
        assertEquals(10,genericBag.capacity());
        assertEquals("{}",genericBag.toString());
```

```
}

@Test
public void testToStringNonEmpty(){
    StringContent sc = new StringContent("A");
    genericBag.add(sc);
    assertEquals("{A}",genericBag.toString());
    genericBag.add(new StringContent("B"));
    assertEquals("{A, B}",genericBag.toString());
}

@Test
public void testAddSizeEqualCap(){
    StringContent sc1 = new StringContent("A");
    StringContent sc2 = new StringContent("B");
    StringContent sc3 = new StringContent("C");
    genericBag = new GenericBag<>(2);
    genericBag.add(sc1);
    genericBag.add(sc2);
    assertEquals(genericBag.size(),genericBag.capacity());
    genericBag.add(sc3);
    assertEquals(3,genericBag.size());
    assertEquals(4,genericBag.capacity());
}

@Test
public void testAddSizeNotEqualCap(){
    genericBag = new GenericBag<>(3);
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    assertEquals("{A, B}",genericBag.toString());
    assertNotEquals(genericBag.size(),genericBag.capacity());
}

@Test
public void testAddSame(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    assertEquals("{A, A, A, A}",genericBag.toString());
    assertEquals(4,genericBag.size());
    assertEquals(10,genericBag.capacity());
}

@Test
public void testRemoveEmpty(){
    genericBag.remove(new StringContent("A"));
    assertTrue(genericBag.isEmpty());
}

@Test
public void testRemoveOne(){
```

```
    genericBag.add(new StringContent("B"));
    genericBag.remove(new StringContent("B"));
    assertTrue(genericBag.isEmpty());
}

@Test
public void testRemoveInvalid(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));
    genericBag.add(new StringContent("D"));
    genericBag.remove(new StringContent("E"));
    assertEquals("{A, B, C, D}",genericBag.toString());
}

@Test
public void testRemoveMultiple(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));
    genericBag.add(new StringContent("D"));
    genericBag.add(new StringContent("E"));
    genericBag.remove(new StringContent("D"));
    genericBag.remove(new StringContent("C"));
    genericBag.remove(new StringContent("B"));
    assertEquals("{A, E}",genericBag.toString());
}

@Test
public void testContainEmpty(){
    assertFalse(genericBag.contains(new StringContent("A")));
}

@Test
public void testContainNonExist(){
    genericBag.add(new StringContent("A"));
    assertFalse(genericBag.contains(new StringContent("B")));
}

@Test
public void testContainExist(){
    genericBag.add(new StringContent("B"));
    assertTrue(genericBag.contains(new StringContent("B")));
}

@Test
public void testIsEmptyEmpty(){
    assertTrue(genericBag.isEmpty());
}

@Test
public void testIsEmptyNonEmpty(){
```

```
    genericBag.add(new StringContent("A"));
    assertFalse(genericBag.isEmpty());
}

@Test
public void testClearEmpty(){
    genericBag.clear();
    assertTrue(genericBag.isEmpty());
}

@Test
public void testClearNonEmpty(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.clear();
    assertTrue(genericBag.isEmpty());
}

@Test
public void testSizeEmpty(){
    assertEquals(0,genericBag.size());
}

@Test
public void testSizeNonEmpty(){
    genericBag.add(new StringContent("A"));
    assertEquals(1,genericBag.size());
}

@Test
public void testRemoveRandomEmpty(){
    genericBag.removeRandom();
    assertNull(genericBag.removeRandom());
    assertEquals(0,genericBag.size());
}

@Test
public void testRemoveRandomOne(){
    genericBag.add(new StringContent("A"));
    genericBag.removeRandom();
    assertEquals("A",genericBag.removeRandom().toString());
    assertEquals(0,genericBag.size());
}

@Test
public void testRemoveRandomMultiple(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));
    genericBag.add(new StringContent("D"));
    genericBag.removeRandom();
    assertEquals(3,genericBag.size());
```

```
}

@Test
public void testRemoveRandomSame(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.removeRandom();
    assertEquals(2,genericBag.size());
    assertEquals("A",genericBag.removeRandom().toString());


}

@Test
public void testGrabRandomEmpty(){
    genericBag.grabRandom();
    assertNull(genericBag.grabRandom());
    assertEquals(0,genericBag.size());
}

@Test
public void testGrabRandomOne(){
    genericBag.add(new StringContent("A"));
    genericBag.grabRandom();
    assertEquals("A",genericBag.grabRandom().toString());
    assertEquals(1,genericBag.size());
}

@Test
public void testGrabRandomMultiple(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));
    genericBag.add(new StringContent("D"));
    genericBag.grabRandom();
    assertEquals(4,genericBag.size());
}

@Test
public void testGrabRandomSame(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("A"));
    genericBag.grabRandom();
    assertEquals("A",genericBag.grabRandom().toString());
    assertEquals(3,genericBag.size());
}

@Test
public void testTrimToSizeEqual(){
    genericBag = new GenericBag<>(3);
    genericBag.add(new StringContent("A"));
```

```
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));
    genericBag.trimToSize();
    assertEquals(genericBag.size(),genericBag.capacity());
}

@Test
public void testTrimToSizeEmpty(){
    genericBag.trimToSize();
    assertEquals(0,genericBag.capacity());
}

@Test
public void testTrimToSizeLess(){
    genericBag = new GenericBag<>(2);
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));
    genericBag.trimToSize();
    assertEquals(3,genericBag.capacity());
}

@Test
public void testTrimToSizeMore(){
    genericBag = new GenericBag<>(5);
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.trimToSize();
    assertEquals(2,genericBag.capacity());
}

@Test

public void test_Clone_EmptyBag(){
    GenericBag<StringContent> gb = genericBag.clone();

    assertEquals(10,gb.capacity());
    assertEquals(0,gb.size());
    String expected = "{}";
    assertEquals(expected,gb.toString());

}

@Test

public void test_Clone_Bag_DifferentObjects(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));
    GenericBag<StringContent> gb = genericBag.clone();
    assertNotSame(genericBag,gb);
    genericBag.add(new StringContent("D"));
    assertNotEquals(genericBag.toString(),gb.toString());
```

```
    gb = genericBag.clone();
    gb.add(new StringContent("D"));
    assertNotSame(genericBag,gb);

}

@Test

public void test_Clone_Identical(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = genericBag.clone();
    assertTrue(genericBag.equals(gb));

}

@Test

public void test_Equals_DifferentCap(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = new GenericBag<>(45);

    gb.add(new StringContent("A"));
    gb.add(new StringContent("B"));
    gb.add(new StringContent("C"));
    assertTrue(genericBag.equals(gb));


}

@Test

public void test_Equals_TwoEmpty_Sequences(){

    GenericBag<StringContent> gb = new GenericBag<>(12);
    assertTrue(genericBag.equals(gb));


}

@Test

public void test_Equals_EmptyVsNonEmpty(){

    GenericBag<StringContent> gb = new GenericBag<>();
    gb.add(new StringContent("A"));
    gb.add(new StringContent("B"));
```

```
    gb.add(new StringContent("C"));

    assertFalse(genericBag.equals(gb));

}

@Test


public void test_Equals_NonEmptyVsEmpty(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = new GenericBag<>();

    assertFalse(genericBag.equals(gb));

}


@Test

public void test_Equals_DifferentSize(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = new GenericBag<>(45);

    gb.add(new StringContent("A"));
    gb.add(new StringContent("B"));

    assertFalse(genericBag.equals(gb));

}

@Test


public void test_Equals_DifferentElements(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = new GenericBag<>(45);

    gb.add(new StringContent("A"));
    gb.add(new StringContent("B"));
    gb.add(new StringContent("D"));
    assertFalse(genericBag.equals(gb));

}
```

```
@Test

public void test_Equals_Reflexivity(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    assertTrue(genericBag.equals(genericBag));

}

@Test

public void test_Equals_Clone(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = genericBag.clone();
    assertTrue(genericBag.equals(gb));


}

@Test


public void test_Equals_Symmetry(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = new GenericBag<>(45);

    gb.add(new StringContent("A"));
    gb.add(new StringContent("B"));
    gb.add(new StringContent("C"));
    assertTrue(genericBag.equals(gb));
    assertTrue(gb.equals(genericBag));

}

@Test
public void testUnionBothEmpty(){
    GenericBag<StringContent> gb = new GenericBag<>();
    GenericBag<StringContent> union = genericBag.union(gb);
    assertTrue(genericBag.isEmpty());
    assertTrue(gb.isEmpty());
    assertTrue(union.isEmpty());
}
```

```
@Test
public void testUnionEmptyVsNonEmpty(){
    GenericBag<StringContent> gb = new GenericBag<>(45);

    gb.add(new StringContent("A"));
    gb.add(new StringContent("B"));
    gb.add(new StringContent("C"));

    GenericBag<StringContent> union = genericBag.union(gb);
    assertEquals("{}", genericBag.toString());
    assertEquals("{A, B, C}", gb.toString());
    assertEquals(3,union.size());


}

@Test
public void testUnionNonEmptyVsEmpty(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = new GenericBag<>(45);

    GenericBag<StringContent> union = genericBag.union(gb);
    assertEquals("{A, B, C}",genericBag.toString());
    assertTrue(gb.isEmpty());
    assertEquals(3,union.size());
}

@Test
public void testUnionNonEmptyVsNonEmpty(){
    genericBag.add(new StringContent("A"));
    genericBag.add(new StringContent("B"));
    genericBag.add(new StringContent("C"));

    GenericBag<StringContent> gb = new GenericBag<>(45);

    gb.add(new StringContent("D"));
    gb.add(new StringContent("E"));
    gb.add(new StringContent("F"));

    GenericBag<StringContent> union = genericBag.union(gb);
    assertEquals(6,union.size());
    assertEquals("{A, B, C}",genericBag.toString());
    assertEquals("{D, E, F}",gb.toString());

}
```

```
}
```

```java
package proj5;

/**
 * Uses a thesaurus and word frequencies to replace overused words in a text
document with random synonyms.
 */
public class GrammarChecker {
    private Thesaurus thesaurus;
    private int threshold;
    private WordCounter wordCounter;
    private LineReader reader;
    private String[] currentLine;

    /**
     * Non-default constructor.
     * Builds a thesaurus out of the given comma-separated file and sets the
threshold for overused words
     * @param thesaurusFile path to comma-separated file used to build a
thesaurus
     * @param threshold a word is considered "overused"
     * if it appears more than (but not equal to) this many times in a text
document
     */
    public GrammarChecker(String thesaurusFile, int threshold){

        thesaurus = new Thesaurus(thesaurusFile);
        this.threshold = threshold;
        wordCounter = new WordCounter();
        reader = null;
        currentLine = null;
    }

    /**
     * Given a text file, replaces overused words with synonyms. Finished
text is printed to the console.
     * @param textfile file with original text
     */
    public void improveGrammar(String textfile){

        wordCounter.findFrequencies(textfile);
        reader= new LineReader(textfile," ");
        currentLine= reader.getNextLine();

        String res="";

        while (currentLine!=null){
            for(String s:currentLine){
                if(isOnlyLetters(s)) {
                    res += handleString(s, wordCounter);
                }
                else{
                    res += s + " ";
                }
```

```
            }


            currentLine = reader.getNextLine();

            if(currentLine!=null){
                res+="\n";
            }
        }
        System.out.println(res);
    }

    /**
     * Check if a String contains only letters
     * @param toCheck the String to check
     * @return true if the String contains only letter, false otherwise.
     */
    private boolean isOnlyLetters(String toCheck){
        for(int i=0;i<toCheck.length()-1;i++){
            char c=toCheck.charAt(i);
            if(!Character.isLetter(c)){
                return false;
            }

        }
        return true;
    }



    /**
     * Handle a word in the input file:
     * 1. Check to see if the word is in upper case. If it is, make the word
lower to check for frequency
     * 2. Strip the last part of the word if it is not a letter
     * to check for frequency with word counter
     * 3. Check if the word need replace has the frequency is higher than
     * threshold. If it is higher, get the synonyms of that word
     * 4. Change the word back to original state
     * by putting the non-letter characters back if any and by making the
word upper case again
     * @param toHandle the word to handle
     * @param wc the counter to count for frequency
     * @return the newly handled word with space for a new word
     */
    private String handleString(String toHandle, WordCounter wc){
        boolean hasUppercase = !toHandle.equals(toHandle.toLowerCase());
        toHandle=toHandle.toLowerCase();

        char last = toHandle.charAt(toHandle.length() - 1);
        if (!Character.isLetter(last)){
            toHandle = toHandle.substring(0, toHandle.length() - 1);
        }
```

```
        int freq = wc.getFrequency(toHandle);
        String replace = toHandle;
        if (freq > threshold && thesaurus.isInThesaurus(toHandle)) {
            replace = thesaurus.getSynonymFor(toHandle);
        }

        if (Character.isLetter(last)) {
            replace += "";
        }
        else{
            replace += last;
        }
        if (hasUppercase)
            replace = (replace.charAt(0) + "").toUpperCase() + replace.
substring(1, replace.length());
        return  replace + " ";
    }
}
```

```java
package proj5;

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

/**
 * test the functionality of WordCounter
 * @author: Emma Vu
 * @version: 6/5/2020
 */
public class WordCounterTest {

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

    private WordCounter wc;

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

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

    @Test
    public void testToStringEmpty(){

        assertEquals(wc.toString(),"");
    }

    @Test
    public void testFindFrequencyEmpty(){

        wc.findFrequencies("src/emptyText.txt");
        assertEquals(wc.toString(),"");
    }

    @Test
    public void testFindFrequency(){

        wc.findFrequencies("src/wordCountText.txt");
        assertEquals(wc.toString(),"a: 7\nb: 5\n");
    }
    @Test
    public void testGetFrequencyIn(){
```

```
        wc.findFrequencies("src/wordCountText.txt");
        assertEquals(7, wc.getFrequency("a"));
    }

    @Test
    public void testGetFrequencyNotIn(){

        wc.findFrequencies("src/wordCountText.txt");
        assertEquals(0, wc.getFrequency("c"));
    }


}
```

```java
package proj5;
/** This is the Generic BST ADT.
 * @author Emma Vu
 * @version 5/30/2020
 * Class Invariant: The position of the element in the BinarySearchTree is determined by
 * comparing the value of each Node, which correspond to different compareTo function
 * (compareTo of Integer is different than compareTo of String)
 * Note that Comparable is a built-in Java interface, there's no need to write a interface class
 */
public class BinarySearchTree<T extends Comparable> {
    private BSTNode<T> root;
    public BinarySearchTree(){
        root = null;
    }

    /**
     * inserts newNode into tree rooted at startingNode.
     * Returns root of that tree with newNode inserted.
     *
     * @param startingNode
     * @param newNode
     * @return root of tree with node inserted
     */
    private BSTNode<T> insert(BSTNode<T> startingNode, BSTNode<T> newNode) {
        if (startingNode == null) {
            return newNode;
        }
        // pretend that key has compareTo method
        else if (startingNode.key.compareTo(newNode.key) > 0) {
            // newNode goes on left
            startingNode.llink = insert(startingNode.llink,newNode);
            return startingNode;
        }
        else {
            // newNode goes on right
            startingNode.rlink = insert(startingNode.rlink,newNode);
            return startingNode;
        }
    }

    /**
     * inserts an int into BST
     * @param data to insert
     */
    public void insert(T data) {
        BSTNode<T> newNode = new BSTNode<T>(data);
        root=insert(root,newNode);
    }

    /**
```

```
     * Search for the element of type T in a BSTNode,
     * if the element is in the tree, return the data in that BSTNode
position
     * if the element is not in the tree, return null
     * @param rootTree the BSTNode we are searching
     * @param keyword the element to search for
     * @return the data of BSTNode that position if the element is in the
Node, null otherwise
     */

    private T search(BSTNode<T> rootTree,T keyword){
        if(rootTree==null){
            return null;
        }
        else{
            if(rootTree.key.compareTo(keyword) > 0) {
                return search(rootTree.llink,keyword);
            }
            else if(rootTree.key.compareTo(keyword) < 0)
                return search(rootTree.rlink, keyword);
            else
                return rootTree.key;
        }

    }

    /**
     * Search for the keyword in the tree
     * return the keyword in the tree if keyword is in the tree
     * null otherwise
     * @param keyword the keyword to search for
     * @return the keyword in the tree, if keyword is in the tree, null
otherwise
     */

    public T search(T keyword){
        return search(root,keyword);
    }

    /**
     * private helper method to find the min value by going to the leftest
link in a BST
     * @param rootTree
     * @return
     */
    private T minValue(BSTNode<T> rootTree)
    {
        T minv = rootTree.key;
        while (rootTree.llink != null)
        {
            minv = rootTree.llink.key;
            rootTree = rootTree.llink;
        }
```

```
        return minv;
    }

    /**
     * Delete Node holding the victim if the victim is in the Node, null
otherwise
     * @param subroot the BSTNode to search for the victim
     * @param victim the victim we are searching for
     * @return the BSTNode holding the target if victim is in subroot, null
otherwise
     * POST CONDITION: the BSTNode holding the victim is deleted from the
subroot BSTNode
     */
    private BSTNode<T> delete(BSTNode<T> subroot, T victim){
        if(subroot == null){
            return subroot;
        }
        else if(subroot.key.compareTo(victim) > 0){
            subroot.llink = delete(subroot.llink,victim);
        }
        else if(subroot.key.compareTo(victim) < 0){
            subroot.rlink = delete(subroot.rlink,victim);
        }
        else{
            if(subroot.isLeaf()){
                return null;
            }
            else if(subroot.hasLeftChildOnly()){
                return subroot.llink;
            }
            else if(subroot.hasRightChildOnly()){
                return subroot.rlink;
            }
            else{
                // node with two children: Get the inorder successor (
smallest
                // in the right subtree)
                subroot.key = minValue(subroot.rlink);

                // Delete the inorder successor
                subroot.rlink = delete(subroot.rlink, subroot.key);
            }
        }
        return subroot;
    }

    /**
     * Delete key from the tree
     * @param key The key to delete
     * POSTCONDITION: nothing happens if key is not in the tree, if key is in
 the tree, it will be deleted/replaced
     */
```

```
    public void delete(T key)
    {
        root = delete(root, key);
    }
    /**
     * Return the String that represent a Node
     * @param subroot the Node to get the String
     * @return The String representing the Node
     */
    private String toString(BSTNode<T> subroot){
        String printStr = "";
        if(subroot != null){
            printStr = "(" + toString(subroot.llink) + " " + subroot.toString
() + " " + toString(subroot.rlink) + ")";
        }
        return printStr;
    }
    /**
     * return the String
     * @return a String representing the tree
     */
    public String toString(){
        return toString(root);
    }

    /**
     * Return the String that represent a Node in correct increasing
alphabetical order
     * @param subroot the Node to get the String
     * @return The String representing the Node
     */
    private String toStringOrder(BSTNode subroot){
        String printStr = "";
        if (subroot!= null) {

            printStr = toStringOrder(subroot.llink) + subroot + "\n" +
toStringOrder(subroot.rlink);
        }
        return  printStr;
    }

    /**
     * return the String of the elements in the tree in correct increasing
alphabetical order
     * @return a String representing the element in increasing order.
     */
    public String toStringOrder()
    {
        if (root == null){
            return "";
        }
        else {
            String printStr = toStringOrder(root);
```

```
            String returnedString = printStr.substring(0, printStr.length
() - 1);
            return  returnedString + "\n";
        }
    }



}
```