Chapter 12

Tree Maps and Tree Sets

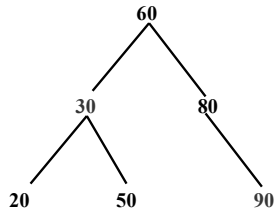
Red-Black Trees

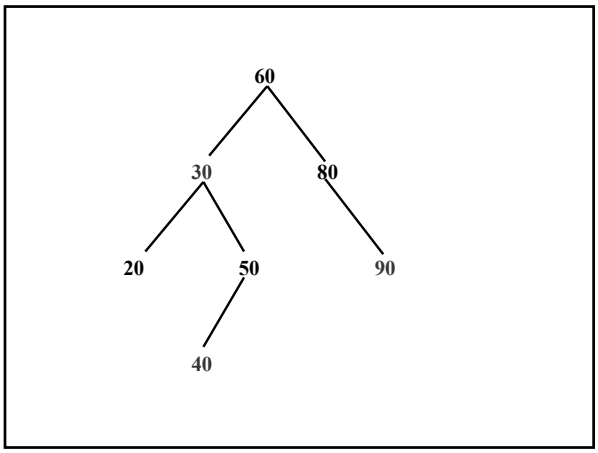
A red-black tree is a balanced binary search tree.

A *red-black tree* is a binary search tree that is empty or in which the root element is colored black, every other element is colored red or black, and

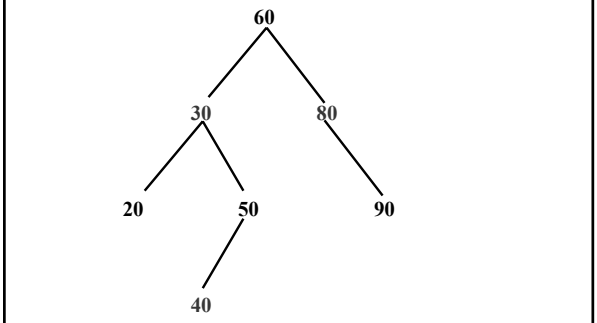
1. **(Red rule) A red element cannot have any red children;**
2. **(Path rule) The number of black elements is the same in any path from the root element to an element with no children or with one child.**

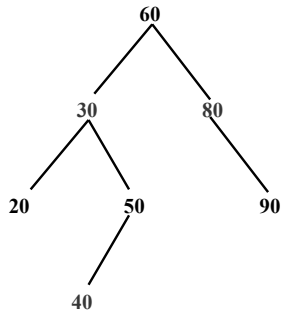
The following are red-black trees:



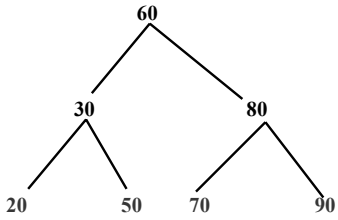


What about this?

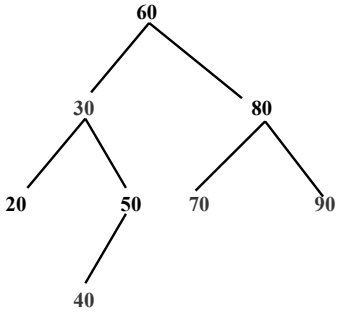




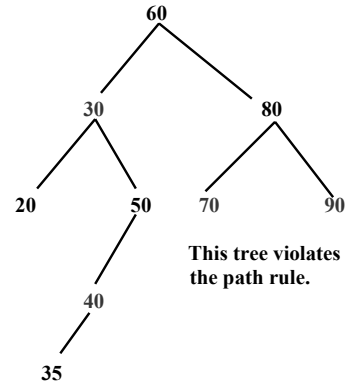
This is not a red-black tree: The path from 60 to 80 has only one black element!



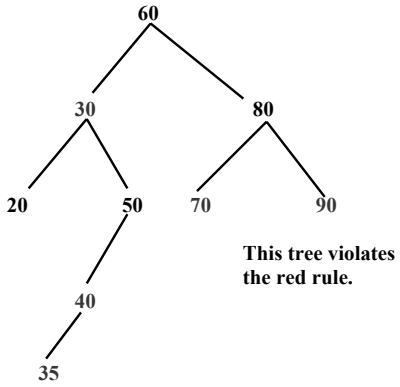
Add 40?



Add 35?

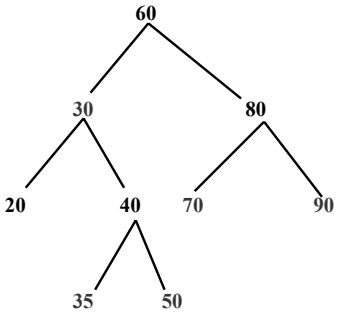


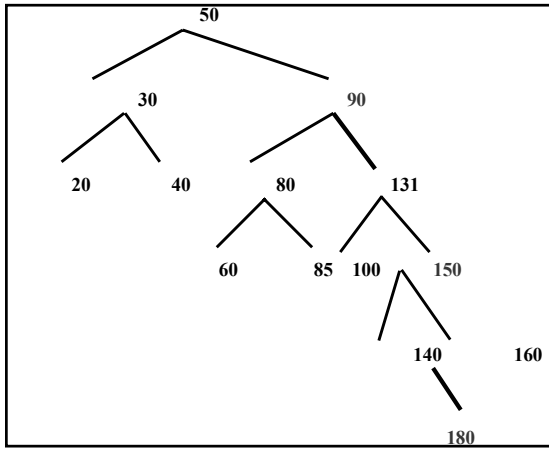
This tree violates the path rule.



This tree violates the red rule.

Rotation to the Rescue!

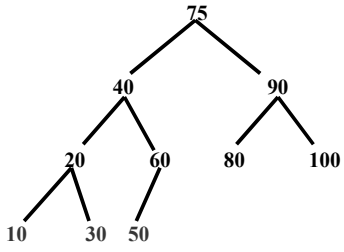




Claim: The height of any red-black tree is logarithmic in *n*.

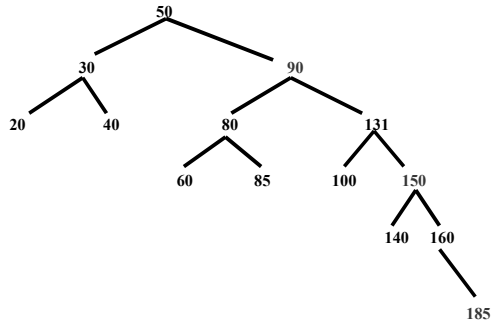
(See Example 2.6 in Appendix 2.)

What is the minimum height of a red-black tree?



Suppose a red-black tree is complete, with all black elements, except for red leaves at the lowest level. Then the height of that tree is, approximately, $\log_2 n$.

What is the maximum height?



Suppose a red-black tree has all black elements, except that one path from the root to a leaf has as many red elements as possible. Then the length of that path, and the height of the tree, will be maximal. The length of that path is approximately twice the minimal height, so the maximum height of a red-black tree will be, roughly, $2\log_2 (n)$.

Group exercise: Give an example of a binary search tree that cannot be colored to make it a red-black tree even though the left and right subtrees have the same height.

A *map* is a collection in which each element has two parts: unique key part and a value part.

For example, we can create a map of students, in which each key is the student ID, and each value is the student's GPA.

Just as with the LinkedList **and** BinarySearchTree **classes, each element is stored in an entry object. The** Map.Entry **class has** getKey() **and** getValue() **methods.**

The Java collection framework's TreeMap class stores a map in a red-black tree, ordered by keys.

```
public class TreeMap<K, V>
        implements SortedMap<K, V>
        extends AbstractMap<K, V>
```

Here are method specifications for several methods in the TreeMap **class:**

Note: The TreeMap **class does not implement the** Collection **interface – because many of the methods are key-value oriented.**

```
/**
 *  Initializes this TreeMap object to be an empty map.
 */
public TreeMap( )
```

Example:

```
TreeMap<String, Double> students =
                new TreeMap<String, Double>( );
```

```
/**
 *  Ensures that there is an element in this TreeMap object
 *  with the specified key&value pair. If this TreeMap
 *  object had an element with the specified key before
 *  this method was called, the previous value associated
 *  with that key has been returned.  Otherwise, null
 *  has been returned.
 *  The worstTime (n) is O (log n).
 *
 *  @param key – the specified key
 *  @param value – the specified value
 *  @return the previous value associated with key, if
 *          there was such a mapping; otherwise, null.
 *
 */
public V put (K key, V value)
```

Examples:

```
students.put ("L00000000", 3.7);
students.put ("L11111111", 2.0);
students.put ("L22222222", 3.5);
students.put ("L44444444", 4.0);
students.put ("L33333333", 3.7);
students.put ("L22222222", 3.8);
```

Now the GPA for L22222222 is 3.8

```
/**
 * Determines if this TreeMap object contains a mapping
 * with a specified key.
 * The worstTime (n) is O (log n).
 *
 * @param key – the specified key
 *
 * @return true – if this TreeMap object contains a mapping
 *              with the specified key; otherwise, false.
 */
public boolean containsKey (Object key)
```

Example:

```
System.out.println (students.containsKey ("L11111111"));
                    // output: true
```

```
/**
 * Determines if this TreeMap object contains a mapping
 * with a specified value.
 * The worstTime (n) is O (n).
 *
 * @param value – the specified value
 *
 * @return true – if this TreeMap object contains a mapping
 *              with the specified value; otherwise, false.
 */
public boolean containsValue (Object value)
```

Example:

```
System.out.println (students.containsValue (3.4));
                    // output: false
```

```
/**
 *  Ensures that there is no mapping in this TreeMap object
 *  with the specified key. If this TreeMap object had such
 *  a mapping before this method was called, the value
 *   has been returned.  Otherwise, null has been returned.
 *  The worstTime (n) is O (log n).
 *
 *  @param key – the specified key
 *
 *  @return the value associated with key, if
 *              there was such a mapping; otherwise, null.
 *
 */
public V remove (Object key)
```

Examples:

```
System.out.println (students.remove ("L22222222"));
                    // output: 3.8

System.out.println (students.remove ("L23456789"));
                    // output: null
```

```
/**
 *  @return a Set view of the mappings in this
 *              TreeMap object.
 */
public Set entrySet( )
```

Example: To print each student whose GPA is above 3.5:

```
for (Map.Entry<String, Double> entry : students.entrySet())
    if (entry.getValue() > 3.5)
        System.out.println (entry);
```

or

```
Iterator<Map.Entry<String, Double>> itr =
                                students.entrySet().iterator();
while (itr.hasNext())
{
    Map.Entry<String, Double> entry = itr.next();
    if (entry.getValue() > 3.5)
        System.out.println (entry);
} // while
```

Here is the output:

L00000000=3.7
L11111111=2.0
L33333333=4.0
L44444444=3.7

Example: To print the ID of each student whose GPA is above 3.5:

```
for (Map.Entry<String, Double> entry : students.entrySet())
    if (entry.getValue() > 3.5)
        System.out.println (entry.getKey());
```

or

```
Iterator<Map.Entry<String, Double>> itr =
                                students.entrySet().iterator();
while (itr.hasNext())
{
    Map.Entry<String, Double> entry = itr.next();
    if (entry.getValue() > 3.5)
        System.out.println (entry.getKey());
} // while
```

Here is the output:

L00000000
L11111111
L33333333
L44444444

Example: To print each GPA that is above 3.5:

```
for (Double gpa : students.values())
    if (gpa > 3.5)
        System.out.println (gpa);
```

or

```
Iterator<Double> itr = students.values().iterator();
while (itr.hasNext())
{
    Double gpa = itr.next();
    if (gpa.doubleValue() > 3.5)
        System.out.println (gpa);
} // while
```

Here is the output;

3.7
4.0
3.7

```
/**
 *  Returns the value associated with a specified key in
 *  this TreeMap object, or null if this TreeMap object has
 *  no mapping with the specified key.
 *  The worstTime (n) is O (log n).
 *
 *  @param key – the specified key
 *
 *  @return the value associated with key, or null if this
 *          TreeMap object has no mapping with this key.
 *
 */
public V get (Object key)
```

Example:

```
System.out.println (students.get ("L11111111"));
System.out.println (students.get ("L22222222"));
```

The output will be

2.0
null

Exercise: Create phoneMap**, a** TreeMap **object in which each key is a 10-digit** Long **(the phone number), and each value is a** String **(the person with that phone number). Insert 3 elements and then print** phoneMap**.**

Illegal: 2222222222
Legal: 2222222222L

The fields in the TreeMap **class**

```
private transient Entry<K, V> root = null;
private transient int size = 0;
private transient int modCount = 0;
private Comparator comparator = null;
```

Recall that transient means that the field itself will not be saved if the instance is serialized (saved to disk).

```
public TreeMap( )
{
    // comparator = null;  key class implements
    // Comparable interface
} // default constructor


public TreeMap (Comparator c)
{
    comparator = c;  // c implements Comparator interface
} // one-parameter constructor
```

Example: Suppose we construct a TreeMap collection; the keys will be of type String and the values will be Integer.

```
TreeMap<String, Integer> myMap =
                        new TreeMap<String, Integer>( );
```

Recall the Comparable **interface:**

```
public interface Comparable
{
    int compareTo(Object obj);
}
```

The int returned by

x.compareTo (y)

Is < 0, if x is less than y;

Is = 0, if x is equal to y;

Is > 0, if x is greater than y.

Since myMap was initialized with the default constructor, the keys are compared by the String class's compareTo method. That performs a lexicographical (≈ alphabetical) comparison:

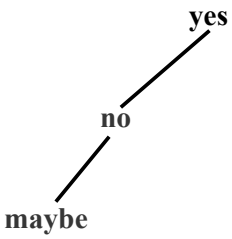
```
myMap.put ("yes", 1);
myMap.put ("no",  1);
myMap.put ("maybe",  1);
myMap.put ("true",  1);
myMap.put ("false",  1);
```

Here, step-by-step, is the red-black tree of keys; Note: When a key is inserted, it is initially colored red.

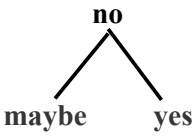
yes

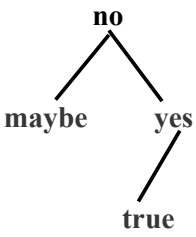
Re-color black because the root must be black.

yes

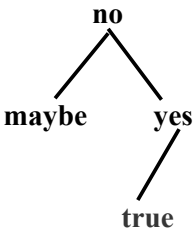


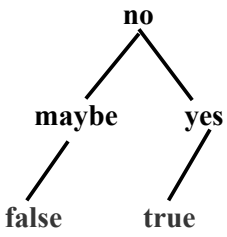
Rotate and re-color:





Re-color “maybe” and “yes” black:






```
System.out.println (myMap.keySet());
```

The output will be:
[false, maybe, no, true, yes]

```
for (String word : myMap.keySet())
    System.out.println (word);
```

The output will be:
false
maybe
no
true
yes

```
Iterator<String> itr = myMap.keySet( ).iterator( );
while (itr.hasNext( ))
    System.out.println (itr.next( ));
```

The output will be:
false
maybe
no
true
yes

Suppose myMap **is a** TreeMap **object with keys of type** String **and values of type** Integer.

1. To print each element:

```
System.out.println (myMap);
System.out.println (myMap.entrySet());
        // KEYS AND VALUES

System.out.println (myMap.keySet());
        // KEYS ONLY

System.out.println (myMap.values());
        // VALUES ONLY
```

2. To iterate through the entries, keys or values without removing from myMap**:**

```
for (Map.Entry<String, Integer> entry : myMap.entrySet())
    …entry.getKey()…entry.getValue()

for (String word : myMap.keySet())
    … word…

for (Integer frequency : myMap.values())
    … frequency …
```

3. To iterate through the elements and, possibly, removing some from myMap**:**

```
Iterator<Map.Entry<String, Integer>> itr =
                    myMap.iterator().entrySet();
```
or
```
Iterator<String> itr = myMap.iterator().keySet();
```
or
```
Iterator<Integer> itr = myMap.iterator().values();
```

```
while (itr.hasNext())
    … itr.next()… // an entry, a key or a value
```

Now we'll look at the Comparator **interface:**

```
public interface Comparator<T>
{
    int compare (T o1, T o2);

    boolean equals (Object obj);

} // interface Comparator
```

The int returned by

compare (x, y)

is < 0, if x **is less than** y**;**

is = 0, if x **is equal to** y**;**

is > 0, if x **is greater than** y**.**

The Comparator **interface allows a user of a class to override how that class performs comparisons. For example, suppose we want to order** String **objects by the length of the string instead of lexicographically.**

Example:

```
public class ByLength implements Comparator<String>
{
    /**
     *  Compares two specified String objects
     *  lexicographically if they have the same length, and
     *  otherwise returns the difference in their lengths.
     *
     *  @param s1 – one of the specified String objects.
     *  @param s2 – the other specified String object.
     *
     *  @return s1.compareTo (s2) if s1 and s2 have the
     *              same length; otherwise, return
     *               s1.length() – s2.length().
     *
     */
```

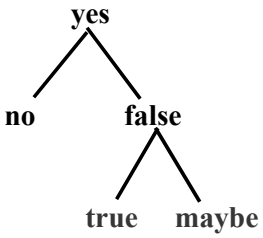
```
    public int compare (String s1, String s2)
    {
        int len1 = s1.length(),
            len2 = s2.length();
        if (len1 == len2)
            return s1.compareTo (s2);
        return len1 – len2;
    } // method compare

} // class ByLength
```

```
TreeMap<String, Integer> yourMap =
    new TreeMap<String, Integer> (new ByLength( ));

yourMap.put ("yes", 1);
yourMap.put ("no", 1);
yourMap.put ("maybe", 1);
yourMap.put ("true",  1);
yourMap.put ("false",  1);
```

Now the ordering is by the length of the keys (but lexicographically for keys with the same length).




```
for (String word : yourMap.keySet())
   if (word.length() > 2)
        System.out.println (word);
```

The output will be:
yes
true
false
maybe

Now, back to the fields and definitions of the TreeMap class.

```
private static final boolean RED = false;

private static final boolean BLACK = true;
```

```
static class Entry<K, V> implements Map.Entry<K, V> {

    K key;
    V value;
    Entry<K, V> left = null;
    Entry<K, V> right = null;
    Entry<K, V> parent;
    boolean color = BLACK;

    // methods such as getKey(), getValue(), toString()

} // class Entry
```

Application of TreeMapS

A Simple Thesaurus

Problem: Given a thesaurus file and words entered from the keyboard, print the synonym(s) of each word entered.

Analysis: A *thesaurus* is a dictionary of synonyms. An error message is printed for each word entered that has no synonyms in the thesaurus file.

Assume the thesaurus file is as follows:

Good enjoyable pleasant nice persistent determined serene tranquil calm

System test (input in blue):

In the input line, please enter the name of the thesaurus file.
thesaurus.dat

In the input line, please enter a word. The sentinel is *.**
HALYCON
The word is not in the thesaurus.

In the input line, please enter a word. The sentinel is *.**
SERENE
The synonyms are: TRANQUIL CALM

In the input line, please enter a word. The sentinel is *.**
***

Design: There will be two classes:

Thesaurus**: To maintain synonym information;**

ThesaurusTester**: To handle input and output.**

```
// Postcondition: this Thesaurus has been initialized.
public Thesaurus( )

// Postcondition: line has been added to this Thesaurus. The
//                worstTime (n) is O (log n).
public void add (String line)

// Postcondition: the LinkedList of synonyms of word has been
//                returned. The worstTime (n) is O (log n).
public LinkedList<String> getSynonyms (String word)
```

The only field is a TreeMap **in which the key is a word and the value is the** LinkedList **of synonyms of the word:**

```
TreeMap<String, LinkedList<String>>
    thesaurusMap;
```

Implementation: The definitions of the constructor and getSynonyms **are one-liners:**

```
public Thesaurus( ) {
    thesaurusMap = new TreeMap
                <String, LinkedList<String>>( );
} // default constructor

public LinkedList<String> getSynonyms (String word) {
    return thesaurusMap.get (word);
} // method getSynonyms
```

The add **method tokenizes the line, makes the first token the key, and stores – as the value – the remaining tokens in a** LinkedList**:**

```
public void add (String line)
{
      LinkedList<String> synonymList =
                  new LinkedList<String>( );

      StringTokenizer st = new StringTokenizer (line);

      String word = st.nextToken();

      while (st.hasMoreTokens())
            synonymList.add (st.nextToken( ));
      thesaurusMap.put (word, synonymList);
} // method add
```

The ThesaurusTester **class will have three methods:**

A default constructor,

constructThesaurus **(from path read in from keyboard)**

printSynonyms **(from keyboard input)**

Fields:

```
protected Thesaurus thesaurus;

protected BufferedReader keyboardReader;
```

```
public ThesaurusTester()
{
      thesaurus = new Thesaurus();
      keyboardReader = new BufferedReader
            (new InputStreamReader (System.in));
} // default constructor
```

```
public void constructThesaurus()
{
      final String FILE_PROMPT =
        "\nPlease enter the path for the thesaurus file: ";
      final String NO_INPUT_FILE_FOUND_MESSAGE =
        "Error: there is no file with that path.\n\n";

      BufferedReader fileReader;

      String inFilePath,
             line;

      boolean pathOK = false;
     while (!pathOK)
     {
            try
            {
```

```
                System.out.print (FILE_PROMPT);
                inFilePath = keyboardReader.readLine();
                fileReader = new BufferedReader (new
                        FileReader (inFilePath));
                pathOK = true;
                while (true) {
                   line = fileReader.readLine();
                   if (line == null)
                       break;
                   thesaurus.add (line);
                } // while not at end of file
            } // try
            catch (IOException e)
            {
               System.out.println (e);
            } // catch
        } // while !pathOK
    } // method constructThesaurus
```

```
public void printSynonyms()
{
      final String SENTINEL = "***";

      final String WORD_PROMPT =
           "\n\nPlease enter the sentinel (" +
           SENTINEL + ") or a word: ";

      final String WORD_NOT_FOUND_MESSAGE =
           "That word does not appear in the thesaurus.";

      final String SYNONYM_MESSAGE =
        "The synonyms of that word are ";

      String word;

      LinkedList<String> synonymList;
```

```
      while (true) {
          try {
            System.out.print (WORD_PROMPT);
            word = keyboardReader.readLine();
            if (word.equals (SENTINEL))
                break;
            synonymList = thesaurus.getSynonyms (word);
            if (synonymList == null)
                System.out.println
                        (WORD_NOT_FOUND_MESSAGE);
             else
                System.out.println (SYNONYM_MESSAGE
                        + synonymList);
          } // try
          catch (IOException e) {
             System.out.println (e);
          } // catch
      } // while
// printSynonyms
```

Time estimates? Let *n* represent the number of lines in the input file and *m* represent the number of words entered from the keyboard.

In constructThesaurus, each of the *n* lines is added to a red-black tree, so worstTime(*n*) is linear-logarithmic in *n*.

Exercise: In printSynonyms**, estimate worstTime(*m*, *n*).**

The TreeSet Class

A *TreeSet* is an ordered Collection **in which duplicate elements are not allowed.**

The TreeSet **class has all of the methods in the** Collection **interface (**add**,** remove**,** size**,** contains**, …) plus** toString **(inherited from** AbstractCollection**) and several constructors.**

```
public class TreeSet<E>
        extends AbstractSet<E>
        implements SortedSet<E>,
        Cloneable,
        java.io.Serializable
{
```

```
public TreeSet( )
    // ASSUMES ELEMENTS ORDERED
    // BY Comparable INTERFACE

public TreeSet (Comparator<? super E> c)
    // ASSUMES ELEMENTS ORDERED
    // BY Comparator c

public TreeSet (Collection<? extends E> c)
    // COPY CONSTRUCTOR; ASSUMES
    // ELEMENTS ORDERED BY
    // Comparable INTERFACE
```

Here are a pair of TreeSet **declarations followed by a few messages. The** ByLength **class was defined earlier.**

```
TreeSet<Integer> tree1 = new TreeSet<Integer>( );
TreeSet<String> tree2 =
            new TreeSet<String> (new ByLength( ));

tree1.add (83);
tree1.add (74);
tree1.add (83);
tree1.add (92);
if (tree1.remove (55))
    System.out.println ("How did 55 get there?");
else
    System.out.println ("size of tree1 = " + tree1.size());
System.out.println (tree1);

tree2.add ("yes");
tree2.add ("no");
tree2.add ("maybe");
tree2.add ("true");
tree2.add ("false");
System.out.println (tree2);
```

The output is:

size of tree1 = 3
[74, 83, 92]
[no, yes, true, false, maybe]

The TreeSet class is implemented with a TreeMap in which all of the values are the same.

```
private transient SortedMap<E, Object> m;
            // The backing Map
private transient Set<E> keySet;
        // The keySet view of the backing Map

// Dummy value to associate with an Object in the
//  backing Map
private static final Object PRESENT = new Object( );
```

Because all of the work is done in the underlying map, the TreeSet methods are one-liners.

For example:

```
/**
 *  Initializes this TreeSet object from a specified
 *  SortedMap object.
 *
 *  @param m – the specified SortedMap object.
 *
 */
private TreeSet (SortedMap<E, Object> m)
{
    this.m = m;                    // OK, so this is a
    keySet = m.keySet( );          // two-liner
} // constructor with map parameter
```

```
/**
 *  Initializes this TreeSet object to be empty.
public TreeSet( )
{
    this (new TreeMap<E, Object>( ));
} // default constructor
```

```
public boolean contains (Object obj)
{
    return m.containsKey (obj);
} // method contains
```

```
public boolean add (E element)
{
    return m.put (element, PRESENT) == null;
} // method add
```

Recall that if m **already has a key equal to** element**, the old value (namely,** PRESENT**) is returned.**

```
/**
 *  Returns the smallest element in this TreeSet object.
 *  The worstTime(n) is O(log n).
 *
 *  @return the smallest element, according to this
 *              TreeSet object's ordering.
 *
 */
public E first( )
{
    return m.firstKey( );
} // method first
```

The TreeSet **class has the same methods as the** BinarySearchTree **class, but is faster, at least in the worst case. For** add**,** remove**, and** contains**, worstTime(*n*) is logarithmic in *n*, versus linear in *n* for the** BinarySearchTree **class.**

TreeSet **Application**

A Spell Checker

Problem: Given a dictionary, in the file supplied by the end-user, and a document, in a file supplied by the end-user, print the words in the document that are not in the dictionary.

Analysis:

1. **The dictionary consists of lower-case words only.**
2. **Each word in the document file consists of letters only – some or all may be in upper-case.**
3. **Each word in the document file is followed by zero or more punctuation symbols followed by any number of blanks and end-of-line markers.**
4. **The dictionary file is in alphabetical order, and will fit in memory. The document file, not necessarily in alphabetical order, will fit in memory if duplicates are excluded.**

Assume the dictionary file consists of
asterisk
do
misspell
not
please
separate

and the document file consists of
Please do not ever mispell asteriks!

System Test:
Please enter the name of the dictionary file: dictionary.dat

Please enter the name of the document file: docfile.dat

The following words are possibly misspelled:
[asteriks, ever, mispell]

Design: As usual, we will separate the spell checker aspects from the input / output aspects.

The SpellChecker **class has four methods:**

```
/**
 *  Initializes this SpellChecker object to be empty.
 */
public SpellChecker ( )
```

```
/**
 *  Inserts a specified word into the dictionary.
 *  The worstTime(n) is O(log n), where n is the number
 *  of words in the dictionary.
 *
 *  @param word – the word to be inserted in the dictionary.
 *
 */
public void addToDictionarySet (String word)
```

```
/**
 *  Inserts the words in a specified line to the document.
 *  The  worstTime(m) is O(log m), where m is the number of
 *  unique words in the document.
 *
 *  @param line – the specified line whose words are to be
 *                inserted into the document.
 *
 */
public void addToDocumentSet (String line)
```

```
/**
 *  Finds each word in the document that is not in the dictionary.
 *  The worstTime(m, n) is O(m log n), where m is the number
 *  of unique words in the document file and n is the number of
 *  words in the dictionary file.
 *
 * @return the LinkedList of words that are in the document but
 *             not in the dictionary.
 */
public LinkedList<String> compare( )
```

The only fields are

```
TreeSet<String> dictionarySet,
                documentSet;
```

Implementation:

The definitions of the default constructor and addToDictionary **are straightforward:**

```
public SpellChecker () {
    dictionarySet = new TreeSet<String>();
    documentSet = new TreeSet<String>();
} // default constructor

public void addToDictionarySet (String word) {
    dictionarySet.add (word);
} // method addToDictionarySet
```

To add the words in a line to documentSet**, the line is tokenized, with delimiters that include punctuation symbols. Each word is lower-cased and inserted in** documentSet **unless the word is already there.**

```
public void addToDocumentSet (String line)
{
        final String DELIMITERS = " \n\r\t;:.,!?()";

        StringTokenizer tokens = new
                StringTokenizer (line, DELIMITERS);

        String word;

        while (tokens.hasMoreTokens( ))
        {
            word = tokens.nextToken( ).toLowerCase();
            documentSet.add (word);
        } // while line has more tokens
    } // method addToDocumentSet
```

Let *m* represent the number of words in the TreeSet **object** documentSet**. When** addToDocumentSet **is called, that method calls the** TreeSet **method** add**, for which worstTime(*m*) is logarithmic in *m*.**
So for the addToDocumentSet **method, worstTime(*m*) is logarithmic in *m*.**

The compare **method iterates through** words**. For each word that is not in** dictionarySet**, the word is appended to** misspelled**, an initially empty** LinkedList**.**

```
public LinkedList<String> compare( )
{
      LinkedList<String> misspelled = new LinkedList<String>( );

      for (String word : documentSet)
            if (!dictionarySet.contains (word))
                  misspelled.add (word);
      } // while
      return misspelled;
} // method compare
```

To iterate through the *m* words in documentSet **takes linear-in-*m* time. Each search of the *n* words in the** TreeSet **object** dictionarySet **takes logarithmic-in-*n* time.**

So for compare**, worstTime(*m*, *n*) is O(*m* log *n*). In fact, worstTime(*m*, *n*) is Θ(*m* log *n*).**

```
/**
 * Initializes this SpellCheckerTester object.
 */
public SpellCheckerTester ()
```

```
/**
 * Reads in and saves a file of a specified type ("dictionary" or
 * "document").
 * The worstTime(t) is O(t log t), where t is the number of lines in
*  the file.
 *
 * @param fileType – a String object representing the type of file,
 *              "dictionary" or "document", to be read in.
 *
 */
public void readFile (String fileType)
```

```
/**
 * Prints the misspelled words – those that are in the document
 * set but not in the dictionary set.
 */
public void printResults()
```

Exercise: Why does the readFile **method have a loop? Why does the** printResults **method *not* have a loop?**