```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;

import java.util.Arrays;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.TreeSet;

//remove this!
import java.util.Iterator;
import java.io.*;

/**
 * Doublets.java
 * Provides an implementation of the WordLadderGame interface. The lexicon
 * is stored as a TreeSet of Strings.
 *
 * @author Will Hendrix (you@auburn.edu)
 * @author Dean Hendrix (dh@auburn.edu)
 * @version 2017-04-28
 */
public class Doublets implements WordLadderGame {

   ////////////////////////////////////////
   // DON'T CHANGE THE FOLLOWING TWO FIELDS. //
   ////////////////////////////////////////

   // A word ladder with no words. Used as the return value for the ladder methods
   // below when no ladder exists.
   List<String> EMPTY_LADDER = new ArrayList<>();

   // The word list used to validate words.
   // Must be instantiated and populated in the constructor.
   TreeSet<String> lexicon;
   TreeSet<String> visited;


   /**
    * Instantiates a new instance of Doublets with the lexicon populated with
    * the strings in the provided InputStream. The InputStream can be formatted
    * in different ways as long as the first string on each line is a word to be
    * stored in the lexicon.
    */
   public Doublets(InputStream in) {
      try {
         lexicon = new TreeSet<String>();
         Scanner s =
            new Scanner(new BufferedReader(new InputStreamReader(in)));
         while (s.hasNext()) {
            String str = s.next();
            ////////////////////////////////////////////
            // Add code here to store str in the lexicon. //
            ////////////////////////////////////////////
            lexicon.add(str.toLowerCase()); // check if this is ok!
            s.nextLine();
         }
         in.close();
      }
      catch (java.io.IOException e) {
         System.err.println("Error reading from InputStream.");
         System.exit(1);
      }
   }

   ////////////////////////////////////////////////////////////////////////////
```

```
// Fill in implementations of all the WordLadderGame interface methods here. //
////////////////////////////////////////////////////////////////////////////////

/**
 * Returns the Hamming distance between two strings, str1 and str2. The
 * Hamming distance between two strings of equal length is defined as the
 * number of positions at which the corresponding symbols are different. The
 * Hamming distance is undefined if the strings have different length, and
 * this method returns -1 in that case. See the following link for
 * reference: https://en.wikipedia.org/wiki/Hamming_distance
 *
 * @param  str1 the first string
 * @param  str2 the second string
 * @return      the Hamming distance between str1 and str2 if they are the
 *                  same length, -1 otherwise
 */
public int getHammingDistance(String str1, String str2) {
   int hammingDistance = 0;

   // if the strings are different legnths
   if (str1.length() != str2.length()) {
      return -1;
   }

   for (int i = 0; i < str1.length(); i++) {
      char a = str1.charAt(i);
      char b = str2.charAt(i);

      if (a != b) {
         hammingDistance++;
      }
   }
   return hammingDistance;
}

/**
 * Returns a word ladder from start to end. If multiple word ladders exist,
 * no guarantee is made regarding which one is returned. If no word ladder exists,
 * this method returns an empty list.
 *
 * Depth-first search with backtracking must be used in all implementing classes.
 *
 * @param  start  the starting word
 * @param  end    the ending word
 * @return        a word ladder from start to end
 */
public List<String> getLadder(String start, String end) {
   Deque<Node> stack = new ArrayDeque<>();
   visited = new TreeSet<String>();
   List<String> result = new ArrayList<String>();

   // if lexicon contains both
   if (!lexicon.contains(start) || !lexicon.contains(end)) {
      return EMPTY_LADDER;
   }

   // if the start and end are the same
   if (start.equals(end)) {
      result.add(end);
      return result;
   }

   // if the lengths are not the same
   if (start.length() != end.length()) {
      return EMPTY_LADDER;
   }

   // create word ladder
   visit(start);
   stack.addLast(new Node(start));
   while (!stack.isEmpty()) {
```

```
      Node current = stack.removeLast();
      for (String i : getNeighbors(current.element)) {
         if (!visited.contains(i)) {
            visit(i);
            //current is previous of neighbor
            Node x = new Node(i);
            x.prev = current;
            stack.addFirst(x);
            if (i.equals(end)) {
               while (x != null) {
                  result.add(x.element);
                  x = x.prev;
               }
            }
         }
      }
   }

   // reverse order
   Deque<String> revStack = new ArrayDeque<>(); // stack
   List<String> reverseResult = new ArrayList<String>();

   for (String i : result) {
      revStack.push(i);
   }
   for (String i : revStack) {
      reverseResult.add(revStack.pop());
   }
   return Arrays.<String>asList(reverseResult.toArray(new String[]{}));
}

/**
 * Returns a minimum-length word ladder from start to end. If multiple
 * minimum-length word ladders exist, no guarantee is made regarding which
 * one is returned. If no word ladder exists, this method returns an empty
 * list.
 *
 * Breadth-first search must be used in all implementing classes.
 *
 * @param  start  the starting word
 * @param  end    the ending word
 * @return        a minimum length word ladder from start to end
 */
public List<String> getMinLadder(String start, String end) {
   Deque<Node> stack = new ArrayDeque<>();
   visited = new TreeSet<String>();
   List<String> result = new ArrayList<String>();

   // if lexicon contains both
   if (!lexicon.contains(start) || !lexicon.contains(end)) {
      return EMPTY_LADDER;
   }

   // if the start and end are the same
   if (start.equals(end)) {
      result.add(end);
      return result;
   }

   // if the lengths are not the same
   if (start.length() != end.length()) {
      return EMPTY_LADDER;
   }

   // create word ladder
   visit(start);
   stack.addLast(new Node(start));
   while (!stack.isEmpty()) {
      Node current = stack.removeLast();
      for (String i : getNeighbors(current.element)) {
         if (!visited.contains(i)) {
```

```
            visit(i);
            //current is previous of neighbor
            Node x = new Node(i);
            x.prev = current;
            stack.addFirst(x);
            if (i.equals(end)) {
               while (x != null) {
                  result.add(x.element);
                  x = x.prev;
               }
            }
         }
      }
   }

   // reverse order
   Deque<String> revStack = new ArrayDeque<>(); // stack
   List<String> reverseResult = new ArrayList<String>();

   for (String i : result) {
      revStack.push(i);
   }
   for (String i : revStack) {
      reverseResult.add(revStack.pop());
   }
   return Arrays.<String>asList(reverseResult.toArray(new String[]{}));
}

/**
 * Returns all the words that have a Hamming distance of one relative to the
 * given word.
 *
 * @param  word the given word
 * @return      the neighbors of the given word
 */
public List<String> getNeighbors(String word) {
   List<String> neighbors = new ArrayList<>();

   for (int i = 0; i < word.length(); i++) {
      // letters must reset to the original word
      // at the start of this loop
      char[] letters = word.toCharArray();
      for (char ch = 'a'; ch <= 'z'; ch++) {
         letters[i] = ch;
         String s = new String(letters);
         if (lexicon.contains(s) && !neighbors.contains(s) && !s.equals(word)) {
            neighbors.add(s);
         }
      }
   }
   return neighbors;
}

/**
  * Returns the total number of words in the current lexicon.
  *
  * @return number of words in the lexicon
  */
public int getWordCount() {
   return lexicon.size();
}

/**
  * Checks to see if the given string is a word.
  *
  * @param  str the string to check
  * @return     true if str is a word, false otherwise
  */
public boolean isWord(String str) {
   return lexicon.contains(str);
}
```

```
 /**
  * Checks to see if the given sequence of strings is a valid word ladder.
  *
  * @param  sequence the given sequence of strings
  * @return          true if the given sequence is a valid word ladder,
  *                       false otherwise
  */
 public boolean isWordLadder(List<String> sequence) {
    int index = 0;
    for (int i = 0; i < sequence.size() - 1; i++) {
       String current = sequence.get(i);
       String next =  sequence.get(i + 1);

       if (!lexicon.contains(current) || !lexicon.contains(next)) {
          return false;
       }

       if (getHammingDistance(current, next) == 1) {
          index++;
       }
    }
    return index == sequence.size() - 1;
 }

//////////////////////////
// PRIVATE NODE CLASS. //
//////////////////////////

 /**
  * Defines a node class for a doubly-linked list.
  */
 class Node {
    /** the value stored in this node. */
    String element;;
    /** a reference to the node before this node. */
    Node prev;

    /**
     * Instantiate an empty node.
     */
    public Node() {
       element = null;
       prev = null;
    }

    /**
     * Instantiate a node that containts element
     * and with no node before or after it.
     */
    public Node(String e) {
       element = e;
       prev = null;
    }
 }

////////////////////////////
// PRIVATE HELPER METHODS. //
////////////////////////////

/**
 * Mark this valid position as having been visited.
 *
 * @param String str.
 */
 private void visit(String str) {
    visited.add(str);
 }

/**
 * Mark this valid position as having been visited.
```

```
   *
   * @param Node n.
   */
   private void visit(Node n) {
      visited.add(n.element);
   }

} // close class
```