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

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

import java.util.stream.Collectors;

/**
 * Provides an implementation of the WordLadderGame interface. The lexicon
 * is stored as a TreeSet of Strings.
 *
 * @author Sean Silva (scs0042@auburn.edu)
 * @author Dean Hendrix (dh@auburn.edu)
 * @version 2017-11-15
 */
public class Doublets implements WordLadderGame {

   // The word list used to validate words.
   // Must be instantiated and populated in the constructor.
   private TreeSet<String> lexicon;
   List<String> EMPTY_LADDER = new ArrayList<>();

   /**
    * 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();
            lexicon.add(str.toLowerCase());
            s.nextLine();
         }
         in.close();
      } catch (java.io.IOException e) {
         System.err.println("Error reading from InputStream.");
         System.exit(1);
      }
   }

   /**
    * 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
    */

   @Override
   public int getHammingDistance(String str1, String str2) {
      if (str1.length() != str2.length()) {
         return -1;
      }
      char[] char1 = str1.toLowerCase().toCharArray();
      char[] char2 = str2.toLowerCase().toCharArray();
      int boz = 0;
      for (int i = 0; i < str1.length(); i++) {
         if (char1[i] != char2[i]) {
            boz++;
         }
      }
      return boz;
   }

   /**
    * 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) {
      List<String> result = new ArrayList<String>();

      if (start.equals(end)) {
         result.add(start);
         return result;
      }
      else if (start.length() != end.length()) {
         return EMPTY_LADDER;
      }
      else if (!isWord(start) || !isWord(end)) {
         return EMPTY_LADDER;
      }

      TreeSet<String> one = new TreeSet<>();
      Deque<String> stack = new ArrayDeque<>();

      stack.addLast(start);
      one.add(start);
      while (!stack.isEmpty()) {

         String current = stack.peekLast();
         if (current.equals(end)) {
            break;
         }
         List<String> neighbors1 = getNeighbors(current);
         List<String> neighbors = new ArrayList<>();

         for (String word : neighbors1) {
            if (!one.contains(word)) {
               neighbors.add(word);
            }
         }
         if (!neighbors.isEmpty()) {
            stack.addLast(neighbors.get(0));
            one.add(neighbors.get(0));
         }
         else {
            stack.removeLast();
         }
      }
      result.addAll(stack);
      return result;
   }


  /**
   * 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) {
      List<String> ladder = new ArrayList<String>();
      if (start.equals(end)) {
         ladder.add(start);
         return ladder;
      }
      else if (start.length() != end.length()) {
         return EMPTY_LADDER;
      }
      else if (!isWord(start) || !isWord(end)) {
         return EMPTY_LADDER;
      }

      Deque<Node> q = new ArrayDeque<>();
      TreeSet<String> one = new TreeSet<>();

      one.add(start);
      q.addLast(new Node(start, null));
      while (!q.isEmpty()) {

         Node n = q.removeFirst();
         String position = n.position;

         for (String neighbor1 : getNeighbors(position)) {
            if (!one.contains(neighbor1)) {
               one.add(neighbor1);
               q.addLast(new Node(neighbor1, n));
            }
            if (neighbor1.equals(end)) {

               Node m = q.removeLast();

               while (m != null) {
                  ladder.add(0, m.position);
                  m = m.predecessor;
               }
               return ladder;
            }
         }
      }
      return EMPTY_LADDER;
   }

   /**
    * 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
    */

   @Override
   public List<String> getNeighbors(String word) {
      List<String> neighbors = new ArrayList<String>();
      for (String string : lexicon) {
         if (this.getHammingDistance(word, string) == 1) {
            neighbors.add(string);
         }
      }
      return neighbors;
   }

   /**
    * Returns the total number of words in the current lexicon.
    *
    * @return number of words in the lexicon
    */

   @Override
   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
    */

   @Override
   public boolean isWord(String str) {
      if (lexicon.contains(str.toLowerCase())) {
         return true;
      }
      return false;
   }

   /**
    * 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) {
      String word1 = "";
      String word2 = "";
```

```
      if (sequence.isEmpty()) {
         return false;
      }
      for (int i = 0; i < sequence.size() - 1; i ++) {
         word1 = sequence.get(i);
         word2 = sequence.get(i + 1);
         if (!isWord(word1) || !isWord(word2)) {
            return false;
         }
         if (getHammingDistance(word1, word2) != 1) {
            return false;
         }
      }
      return true;
   }

   private class Node {
      String position;
      Node predecessor;

      public Node(String p, Node pred) {
         position = p;
         predecessor = pred;
      }
   }
}
```