```
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.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.TreeSet;
import java.util.Iterator;

import java.util.stream.Collectors;

/**
 * Provides an implementation of the WordLadderGame interface.
 *
 * @author Caleb St.Germain (CHS0043@auburn.edu)
 * @author Dean Hendrix (dh@auburn.edu)
 * @version 2020-11-10
 */
public class Doublets implements WordLadderGame {

    // The word list used to validate words.
    // Must be instantiated and populated in the constructor.
    ////////////////////////////////////////////////////////////////////////
/
    // DECLARE A FIELD NAMED lexicon HERE. THIS FIELD IS USED TO STORE ALL
THE //
    // WORDS IN THE WORD LIST. YOU CAN CREATE YOUR OWN COLLECTION FOR
THIS     //
    // PURPOSE OF YOU CAN USE ONE OF THE JCF COLLECTIONS. SUGGESTED
CHOICES    //
    // ARE TreeSet (a red-black tree) OR HashSet (a closed addressed
hash      //
    // table with chaining).
    ////////////////////////////////////////////////////////////////////////
/
    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 {
            ////////////////////////////////////
            // INSTANTIATE lexicon OBJECT HERE  //
            ////////////////////////////////////
            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);
        }
    }


    ////////////////////////////////////////////////////////////
    // ADD IMPLEMENTATIONS FOR ALL WordLadderGame METHODS HERE  //
    ////////////////////////////////////////////////////////////

    public int getHammingDistance(String str1, String str2) {
        if (str1.length() != str2.length()) {
            return -1;
        }
        str1 = str1.toLowerCase();
        str2 = str2.toLowerCase();
        int dist = 0;
        for (int i = 0; i < str1.length(); i++) {
            if (str1.charAt(i) != str2.charAt(i)) {
                dist++;
            }
        }
        return dist;
    }

    public List<String> getMinLadder(String start, String end) {
        start = start.toLowerCase();
        end = end.toLowerCase();
        ArrayList<String> backwards = new ArrayList<String>();
        List<String> minLadder = new ArrayList<String>();
        if (start.equals(end)) {
            minLadder.add(start);
            return minLadder;
        }
        if (getHammingDistance(start, end) == -1) {
            return EMPTY_LADDER;
        }
        if(isWord(start) && isWord(end)) {
            backwards = bfs(start, end);
        }
        if (backwards.isEmpty()) {
            return EMPTY_LADDER;
        }
        for (int i = backwards.size() -1; i >= 0; i--) {
            minLadder.add(backwards.get(i));
        }
        return minLadder;
    }
    private ArrayList<String> bfs(String start, String end) {
        Deque<Node> queue = new ArrayDeque<Node>();
        HashSet<String> visited = new HashSet<String>();
        ArrayList<String> backwards = new ArrayList<String>();
        visited.add(start);
        queue.addLast(new Node(start, null));
        Node endNode = new Node(end, null);
        outerloop:

        while (!queue.isEmpty()) {
            Node n = queue.removeFirst();
            String word = n.word;
            List<String> neighbors = getNeighbors(word);
```

```
            for (String neighbor : neighbors) {
                if(!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.addLast(new Node(neighbor, n));
                    if (neighbor.equals(end)) {
                        endNode.predecessor = n;
                        break outerloop;
                    }
                }
            }
        }
        if(endNode.predecessor == null)
        {
            return backwards;
        }
        Node m = endNode;
        while (m != null) {
            backwards.add(m.word);
            m = m.predecessor;
        }
        return backwards;
    }

    /**
     * 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<String>();
        Iterator<String> itr = lexicon.iterator();
        while (itr.hasNext()) {
            String word2 = itr.next();
            if (getHammingDistance(word, word2) == 1) {
                neighbors.add(word2);
            }
        }
        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) {
        str = str.toLowerCase();
        if (lexicon.contains(str)) {
            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;
    }


    /**
     * Constructs a node for linking positions together.
     */
    private class Node {
        String word;
        Node predecessor;

        public Node(String s, Node pred) {
            word = s;
            predecessor = pred;
        }
    }
}
```