

CS21120 Word Ladder Assignment Document

Module Code:

**CS21120 Program Design, Data Structures and
Algorithms**

Author:

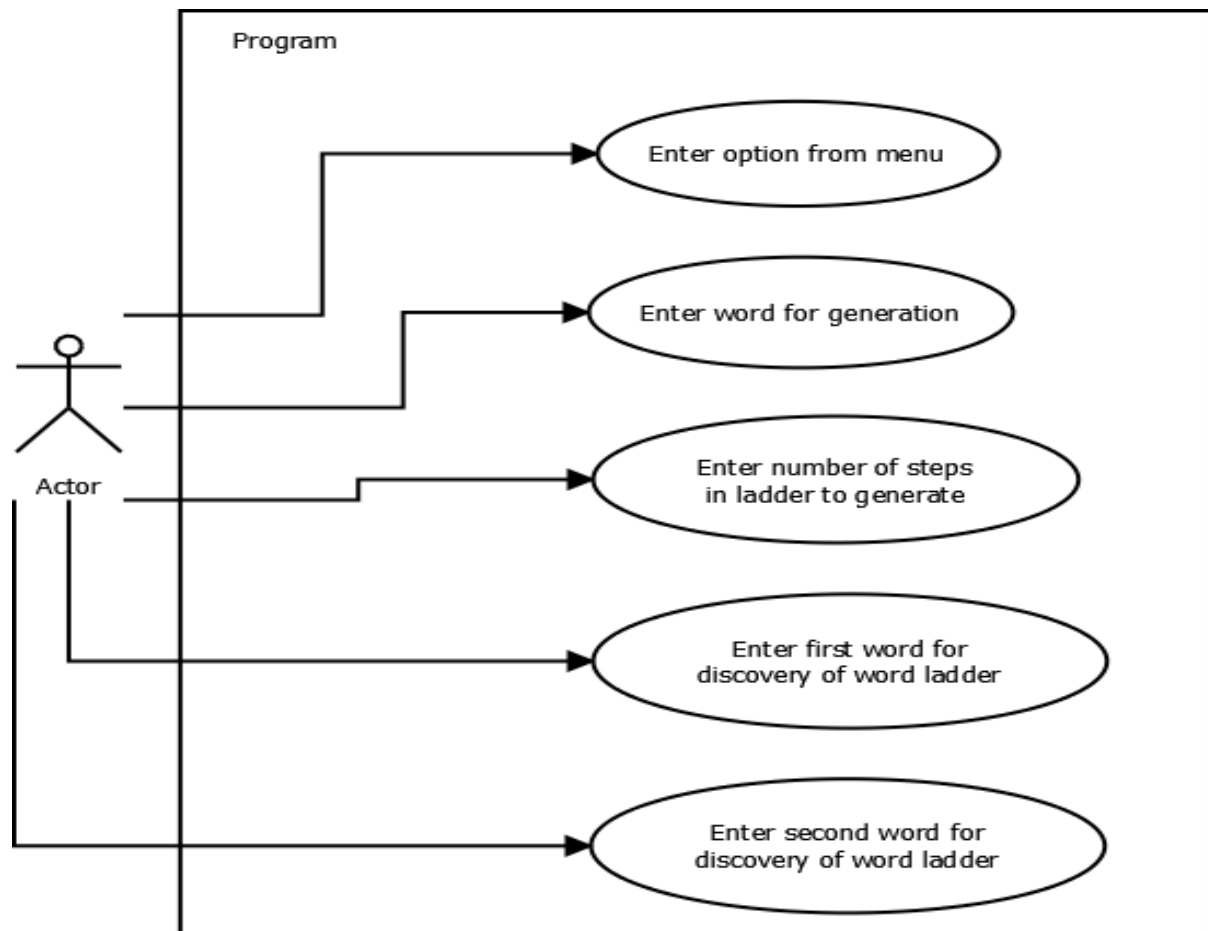
Chris Savill – chs17@aber.ac.uk

Table of Contents

Use Case	4
Class Diagram	5
Design	6
Graph Class	6
Vertex Class	6
WordReader	6
Menu	6
WordLadderDriver	6
Justification of Algorithms	7
Generation Algorithm	7
Discovery Algorithm	7
Pseudo-code	8
Initial start-up	8
Depth-Limited Search (DLS) Algorithm for Generation:	8
Breadth-First Search (BFS) Algorithm for Discovery:	9
Testing	10
Menu Testing	10
Launch Screen:	10
Selecting Option 1 – Generation	10
Selecting Option 2 – Discovery	11
Selecting Option 3 – Exit	11
Selecting an invalid option	11
Generation Testing	12
'test' with '5'	12
'bed' with '462' (limit at 461)	13
'worker' with '50'	14
Discover Testing	14
'head' with 'foot'	15
'print' with 'light'	15
'shine' with 'rhyme'	16
More Tests:	16
Java Source Code	18
Graph	18

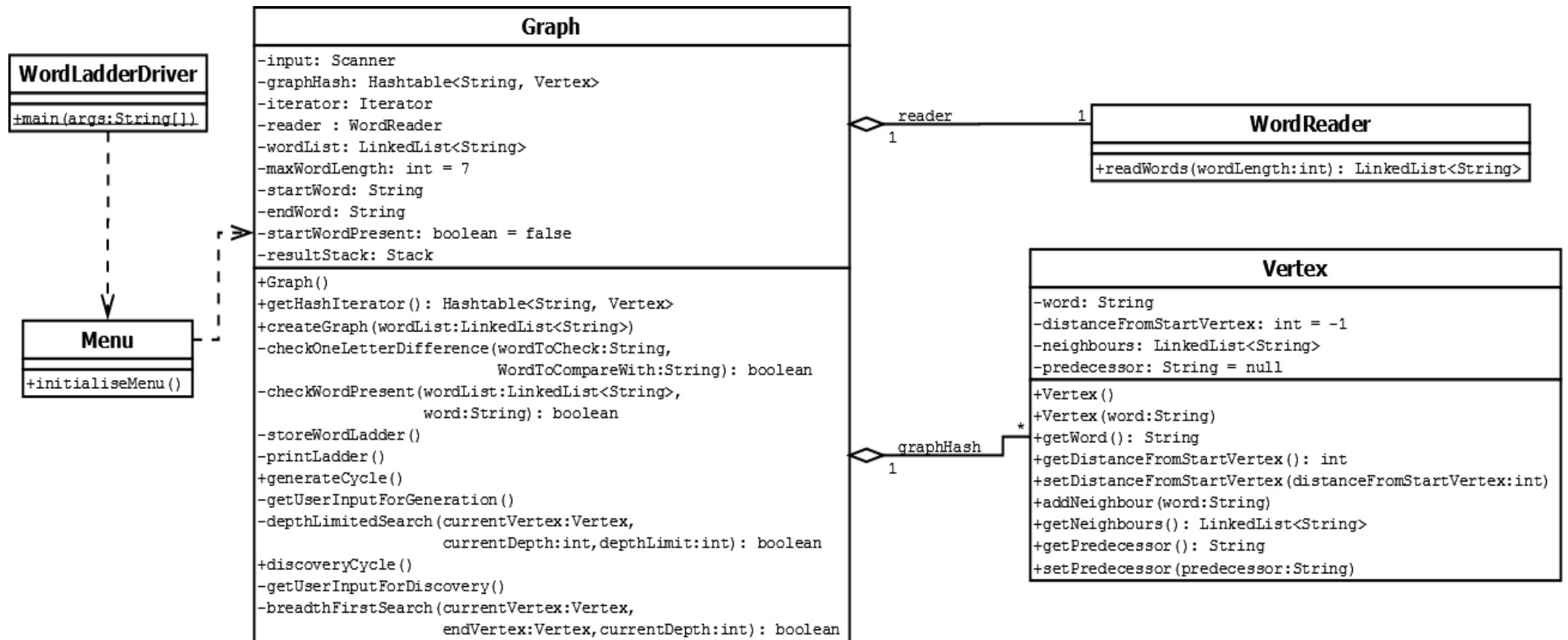
Vertex.....	28
WordReader.....	31
Menu.....	34
WordLadderDriver	35

Use Case



The purpose of the above use case diagram is to illustrate the interactions that the user will have with the program. The user will be able to enter in their option at the launch menu, enter a word to generate a ladder for, enter the depth of the ladder they want to try and generate and enter in a start and end word for the discovery of a word ladder.

Class Diagram



The class diagram above shows how the classes are structured and how they are linked to each other. The program was designed to enforce low-coupling and high-cohesion. By having the Graph class encapsulate all of the methods related to graph generation, traversal of the graph using search algorithms enforces high-cohesion. By having only two other classes (Vertex and WordReader) linked to the Graph class, low coupling is enforced as well. The reason the other two classes are separate is because the WordReader class contains the method relating to file input so there was no need to encapsulate it within the Graph class where it was not relevant. The Vertex class is kept separate because it has its own properties and methods related to vertices so it makes sense to encapsulate those together in the Vertex class.

Design

Below are the class descriptions, refer to class diagram above where necessary. Justifications of design are given for each class where appropriate.

Graph Class

To accomplish the task at hand guidance was given that a graph data structure consisting of a hash table and vertices would be the best method. The hash table data structure would be implemented using the Hashtable.java class. It would use the words of type String as keys and vertices would be the data that the keys were hashed to.

As the class diagram indicates, the Graph class contains methods relating to setting up and creating the graph as well as methods to run the search algorithms for the two main functions of the program.

Vertex Class

The vertex class was designed to act as nodes/vertices of the graph class. The class contains four variables:

1. 'word' which is a string to store the word to which the vertex refers to;
2. 'distanceFromStartVertex' which is an int to label how far from the start word the vertex lies (initialised to -1 to mark as unexplored);
3. 'neighbours' which is a linked list of strings to store all of the words that have only a one letter difference;
4. 'predecessor' which is a string to store the word of the vertex that the vertex was expanded from (initialised to null as there is no predecessor when generated).

Having the linked list of neighbours essentially creates a network of words which represents a graph data structure, along with the hash table to access the graph's vertices.

WordReader

The WordReader class as shown by the class diagram is used by the Graph class for reading in the word lists from the data files provided. The list created is then used to add to the hash table and create new vertices from each word in the word list to create the graph.

Menu

This class is just a simple menu class that gives the user three options, generation, discovery and exit.

WordLadderDriver

As indicated by the class diagram this class just contains the main method, it is the first class launched on start-up and just creates an instance of the Menu class and calls the initialiseMenu method.

Justification of Algorithms

Generation Algorithm

For the generation part of the program it seemed apparent that a Depth-Limited Search (DLS) would be the most appropriate algorithm to use. This is because the user would enter in the number of steps in the ladder which is basically the depth at which to go down. As a Depth-Limited Search algorithm acts like a Depth-First Search (DFS) but with a limit on the depth, it made sense just to use a Depth-Limited Search instead of a Depth-First Search. As soon as the depth limit was reached the resulting ladder would have been generated successfully. It must be referenced that the book 'Artificial Intelligence: A Modern Approach' Third Edition aided me in the decision and implementation of the Depth-Limited Search algorithm for the generation part of the program.

Reference- Russel, S, Norvig, P (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. New Jersey: PEARSON. 85-88.

Discovery Algorithm

For the discovery part of the program it was hinted that Dijkstra's Algorithm would be one of the best ways to go (at least for an uninformed search). However as Dijkstra's Algorithm relies on having a weighted graph (of which the graph implemented here is not), there is no point in using it. Without the priority queue due to a weighted graph the algorithm would act exactly like a Breadth-First Search (BFS), this is why a Breadth-First Search was chosen for the discovery algorithm. The Breadth-First Search algorithm is complete and is guaranteed to find the shortest word ladder/path to a solution as it checks the shallowest vertices/nodes first then the next depth below etc. Breadth-First Search is the most efficient uninformed search algorithm to use. There may be a better and more optimal/efficient informed heuristic search algorithm but due to time constraints it was a good idea to get a simpler algorithm working first. It must be referenced that the book 'Artificial Intelligence: A Modern Approach' Third Edition aided me in the decision and implementation of the Breadth-First Search algorithm for the discovery part of the program.

Reference - Russel, S, Norvig, P (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. New Jersey: PEARSON. 81-83.

Pseudo-code

Initial start-up

Involves:

- Displaying the menu to the user;
- Getting their option input:
- Checking if their selected option is valid
 - If not get input again and recheck.
 - If yes run the corresponding code.

Display "Welcome to the Word Ladder Generator"

While (selected option is not equal to the exit option) {

Prompt a menu to run either the generation function, discovery function or exit function.

Read user's input for option.

Switch (selected option) {

Case 1 (Generation function):

Run generation function

Break out of switch case

Case 2 (Discovery function):

Run discovery function

Break out of switch-case

Case 3 (Exit):

Display "Exiting program"

Break out of switch-case

Default:

Display "Invalid option selected, please select a valid option"

Break out of switch case

}

}

Depth-Limited Search (DLS) Algorithm for Generation:

Recursive DLS-

- Set current word being looked at as the word passed in (start word if at beginning);
- Set distance of current word to current depth (0 for start word);
- Check if current word's distance from start word is higher than the current depth;
 - If yes, return false.
- If no, check if current word is at the depth limit;
 - If yes, return true.
- If no, for each neighbour word unexplored, if any (distance less than 0):
 - Set predecessor of neighbour word to the current word;
 - Check if a call to the recursive DLS method returns true to a result being found (passing in the neighbour word, the current depth + 1 and the depth limit);

- If result found equals true, return true.
- If no result found through any path to the depth required, return false (failure).

Breadth-First Search (BFS) Algorithm for Discovery:

Breadth-First Search-

- Create new frontier queue;
- Set distance from start word to current depth (0 if at beginning/start word);
- Add word to frontier queue;
- While frontier queue is not empty:
 - If current word matches end word, return true.
 - Else set current word to word at the front of the frontier queue and remove front of the queue;
 - For every neighbour of current word unexplored, if any (distance less than 0):
 - Add neighbour to back of frontier queue;
 - Set distance from start word of neighbour to the current depth + 1;
 - Set the predecessor of the neighbour to the current word;

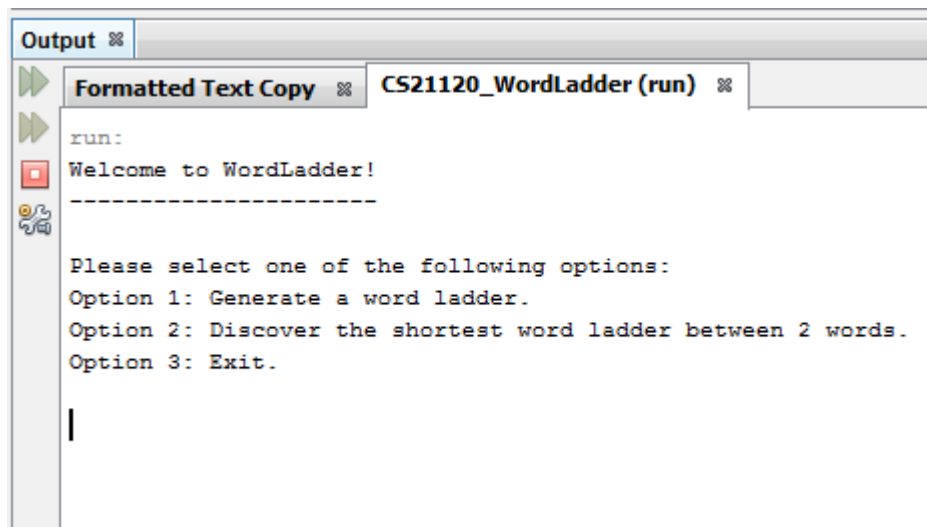
Testing

The testing approach adopted involved JUnit testing where possible (mainly the Vertex class) and then testing of the actual algorithms through running the program and taking screenshots of the outputs. If there were more time, more extensive JUnit tests would have been created to ensure the program is robust.

Menu Testing

Here is a simple screenshot walk through of selecting the three options available to the user as well as what happens if the user enters invalid options:

Launch Screen:



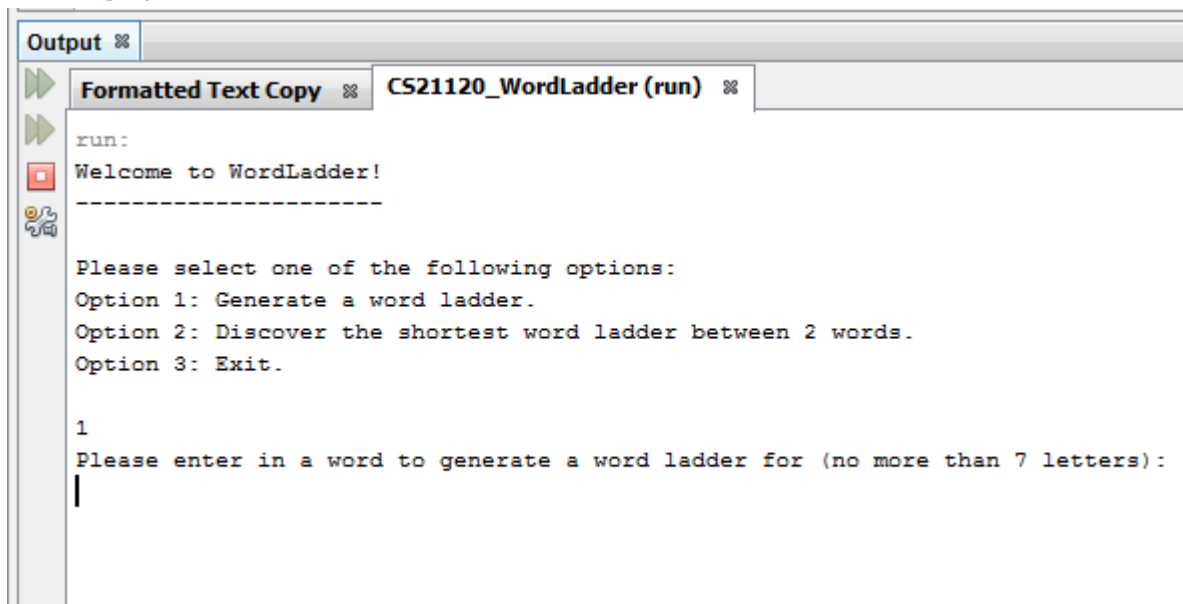
The screenshot shows an IDE's output window with two tabs: 'Formatted Text Copy' and 'CS21120_WordLadder (run)'. The 'run' tab is active, displaying the program's output. The text is as follows:

```
run:
Welcome to WordLadder!
-----

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

|
```

Selecting Option 1 – Generation



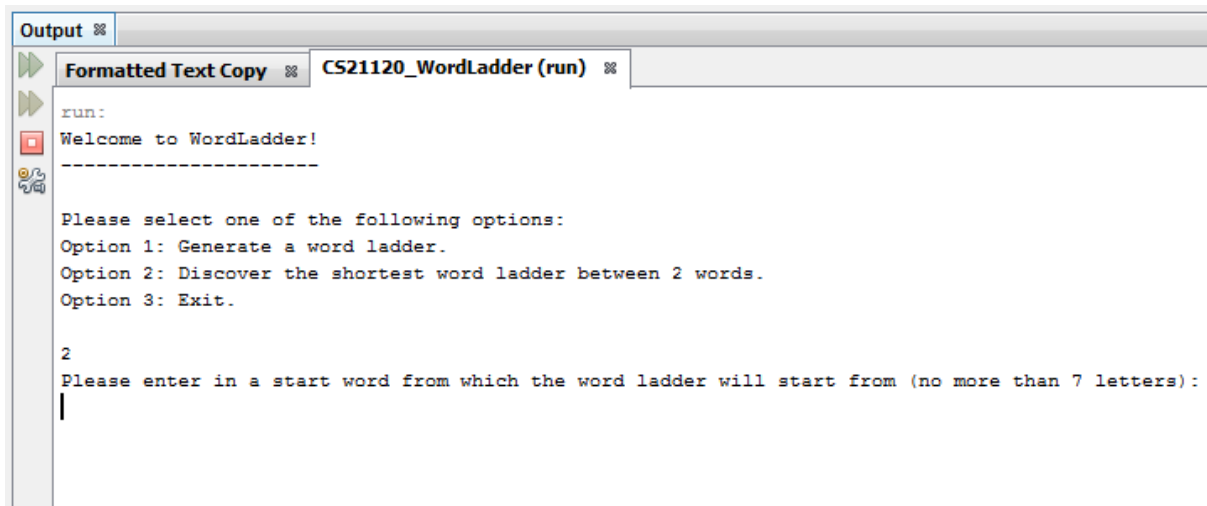
This screenshot shows the same IDE output window as the previous one, but with additional input and output. The user has entered '1' to select Option 1. The program's response is as follows:

```
run:
Welcome to WordLadder!
-----

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

1
Please enter in a word to generate a word ladder for (no more than 7 letters):
|
```

Selecting Option 2 – Discovery

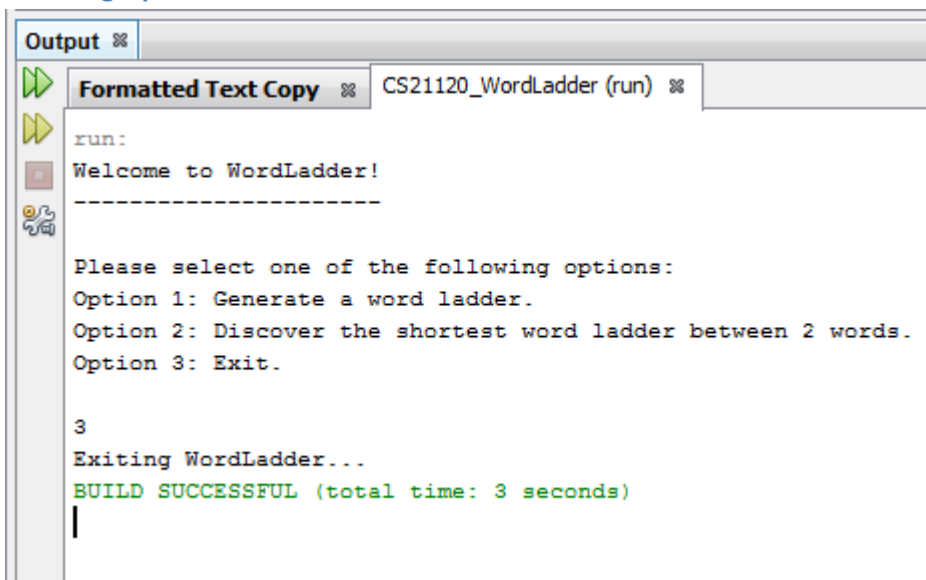


```
Output %
Formatted Text Copy % CS21120_WordLadder (run) %
run:
Welcome to WordLadder!
-----

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

2
Please enter in a start word from which the word ladder will start from (no more than 7 letters):
|
```

Selecting Option 3 – Exit

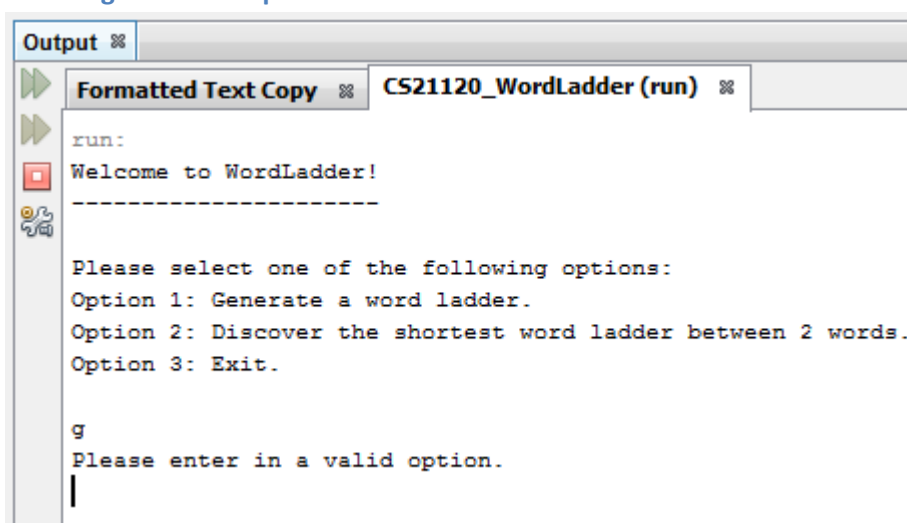


```
Output %
Formatted Text Copy % CS21120_WordLadder (run) %
run:
Welcome to WordLadder!
-----

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

3
Exiting WordLadder...
BUILD SUCCESSFUL (total time: 3 seconds)
|
```

Selecting an invalid option



```
Output %
Formatted Text Copy % CS21120_WordLadder (run) %
run:
Welcome to WordLadder!
-----

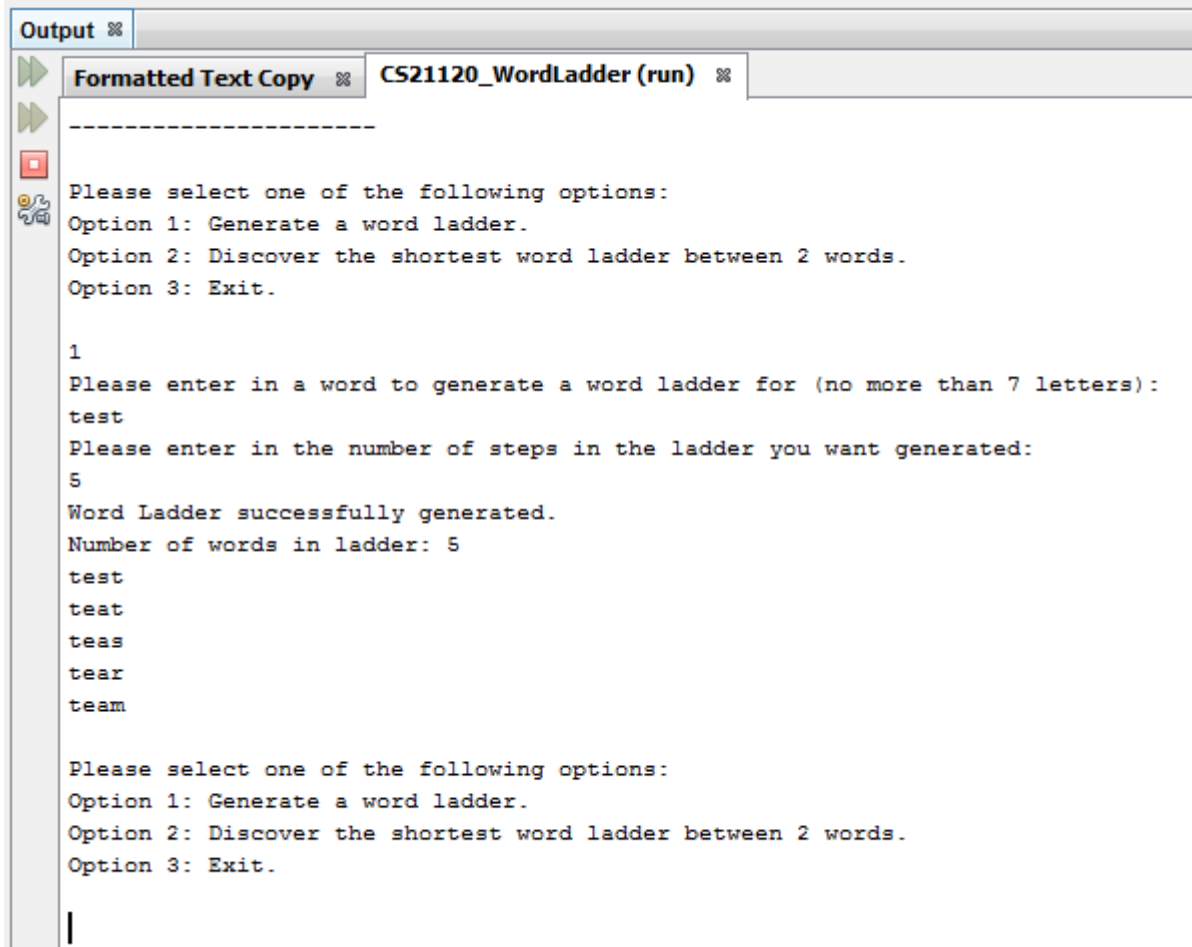
Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

g
Please enter in a valid option.
|
```

Generation Testing

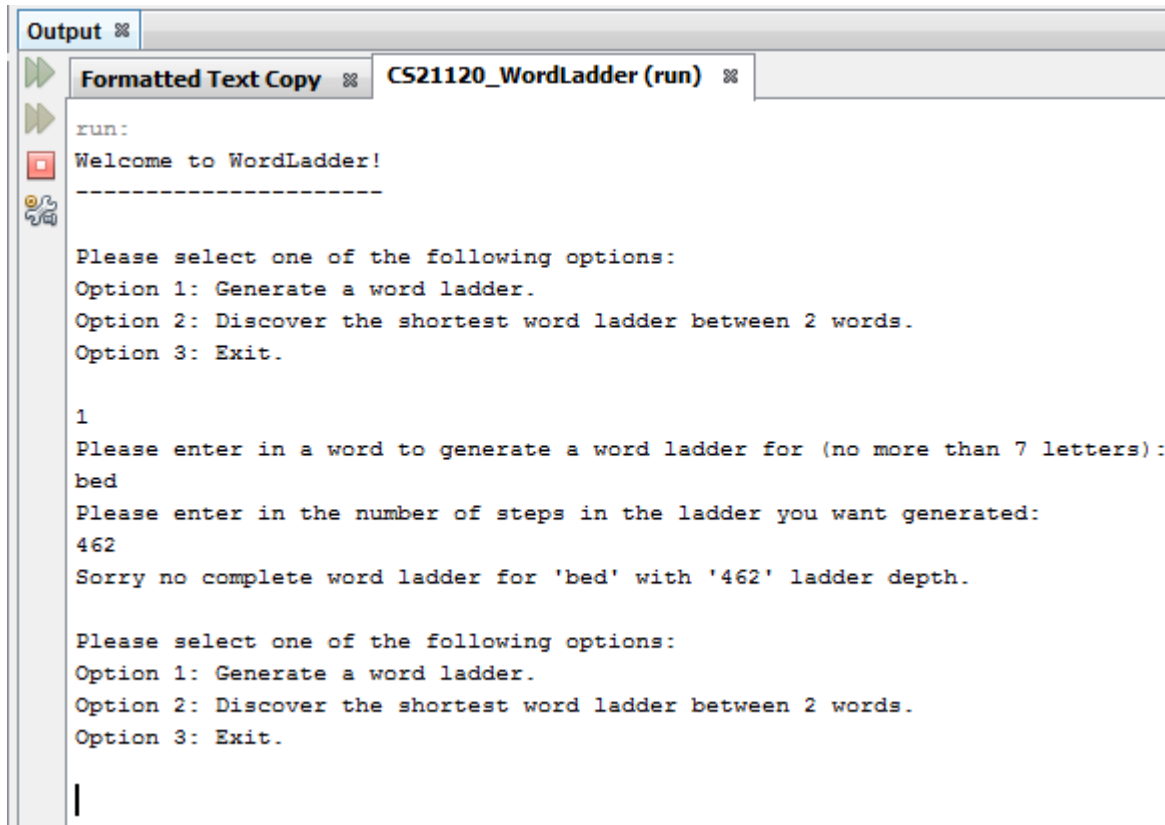
Here are three tests on the generation functionality with three different sets of input:

‘test’ with ‘5’



```
-----  
Please select one of the following options:  
Option 1: Generate a word ladder.  
Option 2: Discover the shortest word ladder between 2 words.  
Option 3: Exit.  
  
1  
Please enter in a word to generate a word ladder for (no more than 7 letters):  
test  
Please enter in the number of steps in the ladder you want generated:  
5  
Word Ladder successfully generated.  
Number of words in ladder: 5  
test  
teat  
teas  
tear  
team  
  
Please select one of the following options:  
Option 1: Generate a word ladder.  
Option 2: Discover the shortest word ladder between 2 words.  
Option 3: Exit.  
  
|
```

'bed' with '462' (limit at 461)



```
run:
Welcome to WordLadder!
-----

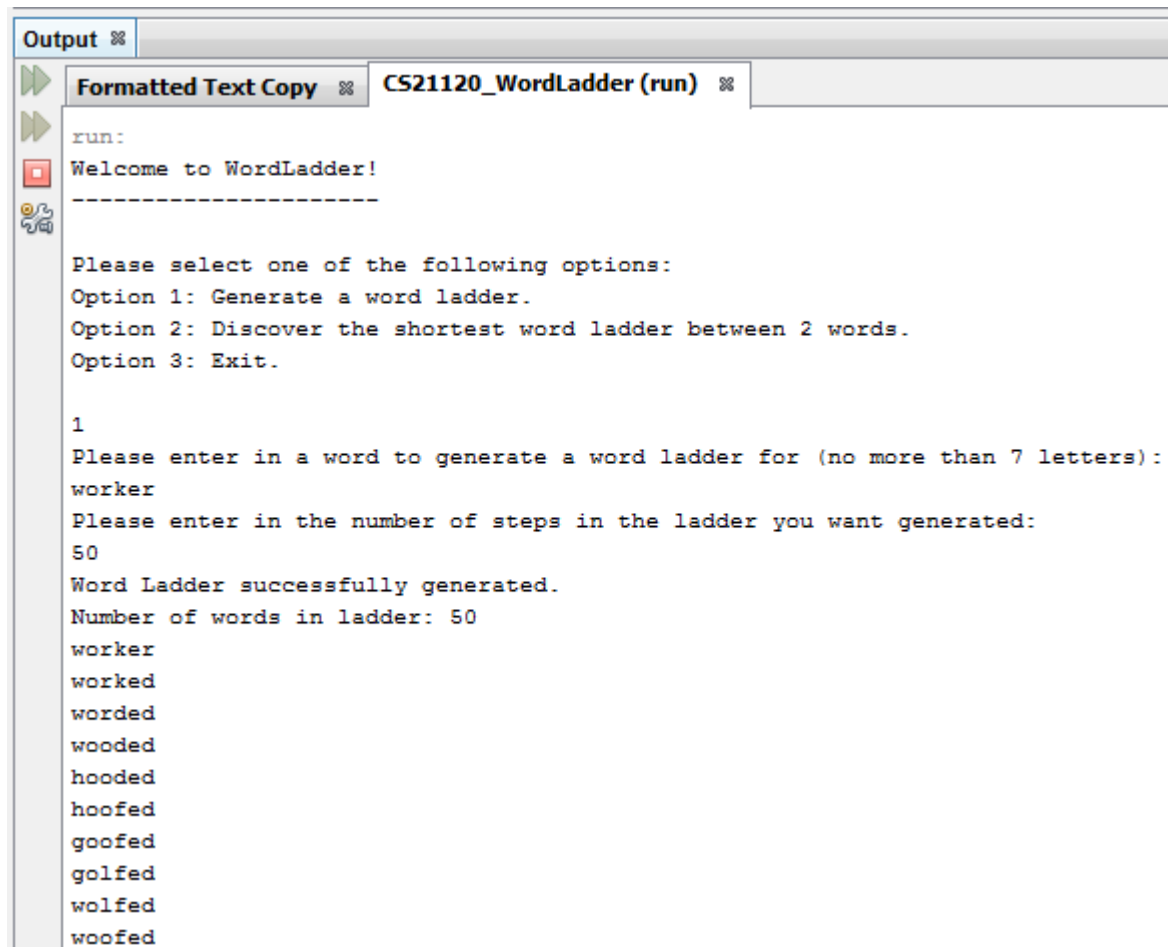
Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

1
Please enter in a word to generate a word ladder for (no more than 7 letters):
bed
Please enter in the number of steps in the ladder you want generated:
462
Sorry no complete word ladder for 'bed' with '462' ladder depth.

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

1
```

'worker' with '50'



The screenshot shows a Java IDE with an 'Output' window. The window has two tabs: 'Formatted Text Copy' and 'CS21120_WordLadder (run)'. The 'CS21120_WordLadder (run)' tab is active, displaying the program's output. The output starts with 'run:' followed by 'Welcome to WordLadder!' and a dashed line. It then prompts the user to select an option from three choices: 'Option 1: Generate a word ladder.', 'Option 2: Discover the shortest word ladder between 2 words.', and 'Option 3: Exit.'. The user enters '1'. Next, it prompts for a word to generate a word ladder for (no more than 7 letters), and the user enters 'worker'. Then, it prompts for the number of steps in the ladder, and the user enters '50'. The program then outputs 'Word Ladder successfully generated.' and 'Number of words in ladder: 50'. Finally, it lists the words in the ladder: 'worker', 'worked', 'worded', 'wooded', 'hooded', 'hoofed', 'goofed', 'golfed', 'wolfed', and 'woofed'.

```
run:
Welcome to WordLadder!
-----

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

1
Please enter in a word to generate a word ladder for (no more than 7 letters):
worker
Please enter in the number of steps in the ladder you want generated:
50
Word Ladder successfully generated.
Number of words in ladder: 50
worker
worked
worded
wooded
hooded
hoofed
goofed
golfed
wolfed
woofed
```

... down to 'beaten'.

Discover Testing

Here are three tests on the discovery functionality with three different sets of input:

'head' with 'foot'

```
Output %
Formatted Text Copy % CS21120_WordLadder (run) %
Welcome to WordLadder!
-----
Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

2
Please enter in a start word from which the word ladder will start from (no more than 7 letters):
head
Please enter in the target word to ladder to (same length as the start word):
foot
Word Ladder successfully generated.
Number of words in ladder: 6
head
bead
beat
boat
boot
foot

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.
```

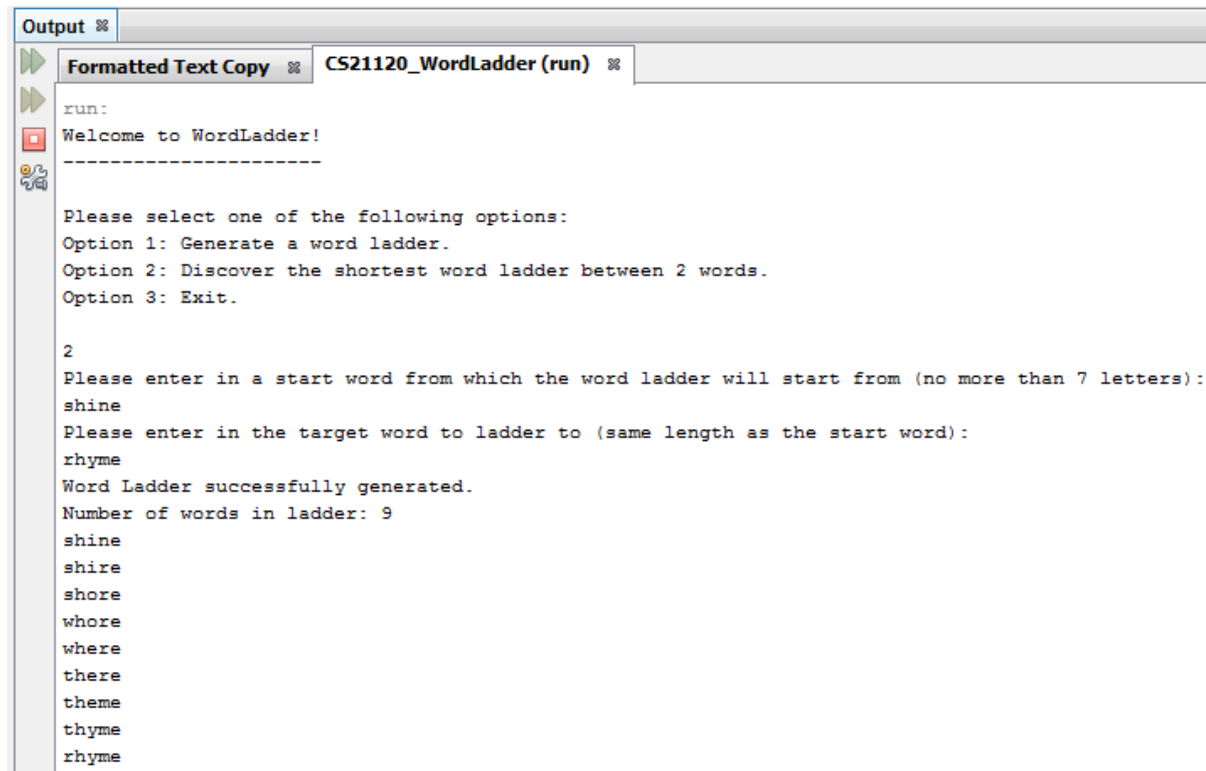
'print' with 'light'

```
Output %
Formatted Text Copy % CS21120_WordLadder (run) %
run:
Welcome to WordLadder!
-----
Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

2
Please enter in a start word from which the word ladder will start from (no more than 7 letters):
print
Please enter in the target word to ladder to (same length as the start word):
light
Sorry no complete word ladder between 'print' and 'light'.

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.
```

'shine' with 'rhyme'



```

Output
Formatted Text Copy CS21120_WordLadder (run)
run:
Welcome to WordLadder!
-----

Please select one of the following options:
Option 1: Generate a word ladder.
Option 2: Discover the shortest word ladder between 2 words.
Option 3: Exit.

2
Please enter in a start word from which the word ladder will start from (no more than 7 letters):
shine
Please enter in the target word to ladder to (same length as the start word):
rhyme
Word Ladder successfully generated.
Number of words in ladder: 9
shine
shire
shore
whore
where
there
theme
thyme
rhyme

```

More Tests

Test Number	Function	Input 1	Input 2	Result
1	Generation	test	2	test teat
2	Generation	fight	10	fight eight bight bigot begot beget begat began begin begun
3	Generation	sincere	5	Sorry no complete word ladder for 'sincere' with '5' ladder depth.
4	Generation	flights	3	flights alights

				blights
5	Generation	on	4	on an ah ad
6	Discovery	dog	bed	dog bog beg bed
7	Discovery	crate	night	Sorry no complete word ladder between 'crate' and 'night'.
8	Discovery	soft	wrap	soft soot coot coop crop crap wrap
9	Discovery	justice	freight	Sorry no complete word ladder between 'justice' and 'freight'.
10	Discovery	jumper	joiner	jumper jumped dumped damped camped carped carded corded corned coined joined joiner

Java Source Code

Graph

```
package aber.dcs.cs21120.chs17.WordLadder.dataStructures;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Map;
import java.util.Scanner;
import java.util.Stack;

/**
 * CS21120 WordLadder class that contains methods to create a graph and traverse
 * the graph to find word ladders
 *
 * @author Chris Savill - chs17
 */
public class Graph {
    ////////////////////////////////////////////////// Variables ///////////////////////////////////

    /**
     * Scanner class used for retrieving user input
     */
    private Scanner input = new Scanner(System.in);

    /**
     * Hash table class used to store the unique words in the key and vertexes
     * as the data
     */
    private Hashtable<String, Vertex> graphHash;

    /**
     * Iterator class used to iterate through the hash table
     */
    private Iterator<Map.Entry<String, Vertex>> iterator;

    /**
     * WordReader class used for reading in words from a data file
     */
}
```

```
*/
private WordReader reader;
/**
 * LinkedList of type String used to store all the words read in by the
 * WordReader class
 */
private LinkedList<String> wordList;
/**
 * int primitive used to set a max word length on words being used in the
 * word ladder, can be changed later if additional word data files generated
 * for other lengths
 */
private int maxWordLength = 7;
/**
 * String class used to store the startWord
 */
private String startWord;
/**
 * String class used to store the endWord
 */
private String endWord;
/**
 * boolean primitive used to determine whether or not the start word is
 * present in the data file being scanned
 */
private boolean startWordPresent = false;
/**
 * Stack class used for storing the resulting words in the word ladder
 */
private Stack resultStack;

//////////////////// Constructors //////////////////////
/**
 * Default constructor that just initialises the hash table
 */
public Graph() {
    graphHash = new Hashtable();
    reader = new WordReader();
}
```

```

    resultStack = new Stack();
}

////////// Methods //////////
/**
 * Method that returns the iterator for the hash table
 *
 * @return Returns the iterator for the graphHash hash table
 */
public Iterator getHashIterator() {
    iterator = graphHash.entrySet().iterator();
    return iterator;
}

/**
 * Method that builds the graph using the word list passed in
 *
 * @param wordList The LinkedList of type String from which a graph is built
 * from
 */
public void createGraph() {
    String key;
    Vertex vertex;
    Vertex newVertex;
    Iterator<Map.Entry<String, Vertex>> iterator;

    for (int counter = 0; counter < wordList.size(); counter++) {
        graphHash.put(wordList.get(counter), newVertex = new Vertex(wordList.get(counter))); //Creates a new vertex,
initialising it with the word in the word list at the index of the counter
        iterator = getHashIterator(); //Retrieves a new iterator for the hash table within the graph class

        while (iterator.hasNext()) {
            key = iterator.next().getKey(); //Assigns the next key from to the iterator to the key variable
            vertex = graphHash.get(key); //Assigns the vertex in the hash table referenced from the key to the vertex
variable
            if (checkOneLetterDifference(newVertex.getWord(), vertex.getWord()) == true) {
                vertex.addNeighbour(newVertex.getWord()); //Adds newVertex word to the neighbour adjacency list of the
current vertex it is being compared to
            }
        }
    }
}

```

```

        newVertex.addNeighbour(vertex.getWord()); //Adds the current vertex word to the neighbour adjacency list of
the newVertex it is being compared to

```

```

    }
}

/**
 * Method to check if the two words being passed in only have a one letter
 * difference
 *
 * @param wordToCheck A String to compare with the wordToCompareWith String
 * @param wordToCompareWith A String to compare with the wordToCheck String
 * @return Returns true or false depending on whether or not the two words
 * only have a one letter difference
 */
private boolean checkOneLetterDifference(String wordToCheck, String wordToCompareWith) {
    boolean oneLetterDifference = false;
    int numberOfLettersMatched = 0;

    for (int counter = 0; counter < wordToCheck.length(); counter++) {
        if (wordToCheck.charAt(counter) == wordToCompareWith.charAt(counter)) {
            numberOfLettersMatched++;
        }
    }

    if (numberOfLettersMatched == wordToCheck.length() - 1) {
        oneLetterDifference = true;
    }
    return oneLetterDifference;
}

/**
 * Method that checks if the word passed in is present in the word list
 * supplied
 *
 * @param wordList The LinkedList of type String that contains the list of
 * words for comparison

```

```
* @param word The String to compare with the wordList LinkedList
* @return Returns true or false depending on whether or not the word passed
* in is present in the word list supplied
*/
private boolean checkWordPresent(LinkedList<String> wordList, String word) {
    boolean wordPresent = false;
    for (int counter = 0; counter < wordList.size() && wordPresent == false; counter++) {
        if (wordList.get(counter).equals(word)) {
            wordPresent = true;
        }
    }
    return wordPresent;
}

/**
 * Method that sets off the word ladder generation cycle, first by calling a
 * method that gets the word to ladder from the user, then calls a method
 * that creates a new graph, then calls the search algorithm, evaluates its
 * result and acts accordingly
 */
public void generateCycle() {
    int ladderDepth = 0;
    ladderDepth = getUserInputForGeneration();
    createGraph();

    if (depthLimitedSearch(graphHash.get(startWord), 0, ladderDepth - 1) == true) { //Evaluates the result of the
recursiveDepthLimitedSearchForGeneration method
        storeWordLadder();
        printLadder();
    } else {
        System.out.println("Sorry no complete word ladder for '" + startWord + "' with '" + ladderDepth + "' ladder
depth.");
    }
}

/**
 * Method that cycles through getting the user to input a valid word for the
 * word ladder generation and checks if it exists in the data files supplied
```

```

    * and then gets the user to input the depth of the ladder they want
    */
    private int getUserInputForGeneration() {
        while (startWordPresent == false) {
            startWord = "WordTooLong"; // "WordTooLong" used as it has more than 7 letters and to initialise word ready for while
loop condition checking
            while (startWord.length() > maxWordLength) {
                System.out.println("Please enter in a word to generate a word ladder for (no more than 7 letters): ");
                startWord = input.next();
            }

            wordList = reader.readWords(startWord.length());

            if (checkWordPresent(wordList, startWord) == true) { //Evaluates if the word chosen exists in the appropriate word
data file
                startWordPresent = true;
            } else {
                System.out.println("Word is not present in file, please try another word.");
            }
        }
        System.out.println("Please enter in the number of steps in the ladder you want generated: ");

        while (!input.hasNextInt()) {
            System.out.println("Please enter in a valid option.");
            input.next();
        }

        return input.nextInt();
    }

    /**
     * Depth-Limited Search (DLS) algorithm to find the word ladder for a word
     * up to a certain depth
     *
     * @param currentVertex The current vertex being evaluated to see if goal
     * state has been met.
     * @param currentDepth The current depth at which the current vertex lies
     * at.

```

```

* @param depthLimit The maximum depth that the search will go to; the goal
* state
* @return Returns True or false based on whether the goal state has been
* found.
*/
private boolean depthLimitedSearch(Vertex currentVertex, int currentDepth, int depthLimit) {
    boolean resultFound = false;
    if (currentVertex.getDistanceFromStartVertex() < 0) {
        currentVertex.setDistanceFromStartVertex(currentDepth);
    }

    if (currentVertex.getDistanceFromStartVertex() > currentDepth) {
        return false;
    } else {
        if (currentVertex.getDistanceFromStartVertex() == depthLimit) {
            endWord = currentVertex.getWord();
            return true;
        } else {
            for (String neighbour : graphHash.get(currentVertex.getWord()).getNeighbours()) {
                if (graphHash.get(neighbour).getDistanceFromStartVertex() < 0) {
                    graphHash.get(neighbour).setPredecessor(currentVertex.getWord()); //Sets the predecessor/parent vertex
of the neighbour/child vertex to the current vertex
                    resultFound = depthLimitedSearch(graphHash.get(neighbour), currentDepth + 1, depthLimit);
                    if (resultFound == true) {
                        return true;
                    }
                }
            }
        }
    }
    return false;
}

/**
 * Method that sets off the word ladder discovery cycle, first by calling a
 * method that gets the words to ladder between from the user, then calls a
 * method that creates a new graph, then calls the search algorithm,
 * evaluates its result and acts accordingly

```



```

    */
    public void discoveryCycle() {
        getUserInputForDiscovery();
        createGraph();

        if (breadthFirstSearch(graphHash.get(startWord), graphHash.get(endWord), 0) == true) { //Evaluates the result of the
iterativeDeepeningSearchForDiscovery method
            storeWordLadder();
            printLadder();
        } else {
            System.out.println("Sorry no complete word ladder between '" + startWord + "' and '" + endWord + "'.");
        }
    }

    /**
     * Method that cycles through getting the user to input two valid words for
     * the word ladder discovery and checks if they exist in the data files
     * supplied
     */
    private void getUserInputForDiscovery() {
        boolean endWordPresent = false;

        while (startWordPresent == false) {
            startWord = "WordTooLong"; // "WordTooLong" used as it has more than 7 letters and to initialise word ready for while
loop condition checking
            endWord = "WordTooLong";
            while (startWord.length() > maxWordLength) {
                System.out.println("Please enter in a start word from which the word ladder will start from (no more than 7
letters): ");
                startWord = input.next();
            }

            wordList = reader.readWords(startWord.length());

            if (checkWordPresent(wordList, startWord) == true) { //Evaluates if the word chosen exists in the appropriate word
data file
                startWordPresent = true;
            } else {

```

```

        System.out.println("Start word is not present in file, please try another word.");
    }
}

while (endWordPresent == false) {
    while (endWord.length() != startWord.length()) {
        System.out.println("Please enter in the target word to ladder to (same length as the start word): ");
        endWord = input.next();
    }

    if (checkWordPresent(wordList, endWord) == true) { //Evaluates if the word chosen exists in the appropriate word
data file
        endWordPresent = true;
    } else {
        endWord = "WordTooLong";
        System.out.println("Target word is not present in file, please try another word.");
    }
}

}

/**
 * Breadth-First Search (BFS) algorithm to find the shortest word ladder
 * between two words
 *
 * @param currentVertex The current vertex/word being analysed
 * @param endVertex The target vertex/word/goal state
 * @param currentDepth The current depth in the graph
 * @return Returns true if word ladder has been found, false if not
 */
private boolean breadthFirstSearch(Vertex currentVertex, Vertex endVertex, int currentDepth) {
    LinkedList<String> frontierQueue = new LinkedList<String>();
    currentVertex.setDistanceFromStartVertex(currentDepth); //Sets distance from start vertex to the current depth, if it is
the start vertex, distance would be 0
    frontierQueue.add(currentVertex.getWord()); //Adds the current vertex to the queue

    while (!frontierQueue.isEmpty()) { //Evaluates if the frontierQueue queue is not empty
        if (currentVertex.getWord().equals(endVertex.getWord())) { //Checks if goal state has been met
            endWord = currentVertex.getWord();

```

```

        return true;
    } else {
        currentVertex = graphHash.get(frontierQueue.peek().toString()); //Sets the current vertex to the vertex at the
front of the queue
        frontierQueue.remove(); //Removes the current vertex from the frontierQueue queue, (counted as explored)

        for (String neighbour : graphHash.get(currentVertex.getWord()).getNeighbours()) {
            if (graphHash.get(neighbour).getDistanceFromStartVertex() < 0) { //Evaluates if the vertexes have been
explored

                frontierQueue.add(graphHash.get(neighbour).getWord()); //Adds neighbour/child vertex to end of queue
                graphHash.get(neighbour).setDistanceFromStartVertex(currentDepth + 1); //Sets the distance from start
vertex to the next depth level
                graphHash.get(neighbour).setPredecessor(currentVertex.getWord()); //Sets the predecessor/parent vertex
of the neighbour/child vertex to the current vertex
            }
        }
    }
}

return false; //Return false if no result found. If false is returned at the top, a failure to find the result has
occurred
}

/**
 * Method that stacks the path/word ladder between the two words, works
 * backwards from the goal state using the predecessor variable
 *
 * @param endWord The target vertex/word/goal state
 */
private void storeWordLadder() {
    String currentWord = endWord;
    resultStack.push(endWord); //Adds the goal state word to the result stack

    while (graphHash.get(currentWord).getPredecessor() != null) { //Loops until hit start vertex as the start vertex would
have no predecessor so would be null
        resultStack.push(graphHash.get(currentWord).getPredecessor()); //Adds predecessor to result stack
        currentWord = graphHash.get(currentWord).getPredecessor(); //Sets the current word to the predecessor vertex
    }
}
}

```

```

/**
 * Method to print the resulting ladder from the result stack
 */
private void printLadder() {
    System.out.println("Word Ladder successfully generated.");
    System.out.println("Number of words in ladder: " + resultStack.size());
    while (!resultStack.isEmpty()) {
        System.out.println(resultStack.pop().toString()); //Prints out the word ladder stack if successful
    }
}
}

```

Vertex

```

package aber.dcs.cs21120.chs17.WordLadder.dataStructures;

import java.util.LinkedList;

/**
 * CS21120 WordLadder class to represent a vertex/node for use within the graph
 *
 * @author Chris Savill - chs17
 */
public class Vertex {
    ////////////////////////////////////////////////// Variables ///////////////////////////////////

    /**
     * String class used to store the word within the vertex
     */
    private String word;

    /**
     * int primitive used to store the distance of the vertex from the start
     * vertex, defaults to -1 to represent that the vertex is unexplored
     */
    private int distanceFromStartVertex = -1;

    /**
     * LinkedList of type String used as an adjacency list for the vertex of all
     * the neighbours

```

```
*/
private LinkedList<String> neighbours;
/**
 * String class used to store the predecessor of the vertex. Defaults to
 * null as no predecessor is assigned at first.
 */
private String predecessor = null;

////////// Constructors //////////
/**
 * Default constructor
 */
public Vertex() {
}

/**
 * Constructor used for initialising the word and neighbours
 *
 * @param word the String to initialise the vertex word variable
 */
public Vertex(String word) {
    this.word = word;
    neighbours = new LinkedList();
}

////////// Methods //////////
/**
 * Method to return the word assigned to the vertex
 *
 * @return Returns word
 */
public String getWord() {
    return this.word;
}

/**
 * Method to return the distance that this vertex is from the start vertex
 *
```

```
* @return Returns distanceFromStartVertex
*/
public int getDistanceFromStartVertex() {
    return this.distanceFromStartVertex;
}

/**
 * Method to set the value of the distance that this vertex is from the
 * start vertex
 *
 * @param distanceFromStartVertex The int value to initialise the vertex
 * distanceFromStartVertex variable
 */
public void setDistanceFromStartVertex(int distanceFromStartVertex) {
    this.distanceFromStartVertex = distanceFromStartVertex;
}

/**
 * Method to add a new neighbour to the adjacency list (LinkedList) of
 * neighbours
 *
 * @param word The string to add to the adjacency list of the vertex
 */
public void addNeighbour(String word) {
    neighbours.add(word);
}

/**
 * Method to return the LinkedList of neighbours
 *
 * @return Returns neighbours
 */
public LinkedList<String> getNeighbours() {
    return this.neighbours;
}

/**
 * Method to return the predecessor of the vertex
```

```

    *
    * @return Returns predecessor
    */
    public String getPredecessor() {
        return this.predecessor;
    }

    /**
     * Method to set the predecessor of the vertex
     *
     * @param predecessor The String to initialise the vertex predecessor
     * variable
     */
    public void setPredecessor(String predecessor) {
        this.predecessor = predecessor;
    }
}

```

WordReader

```

package aber.dcs.cs21120.chs17.WordLadder.dataStructures;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.LinkedList;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Class that contains various methods to read words from a file
 *
 * @author Chris Savill - chs17
 */
public class WordReader {

    /////////////////////////////////////////////////// Methods ///////////////////////////////////
}

```

```
* Method that reads the words of a file into a LinkedList of String type
*
* @param wordLength int value of the word length for use with defining the
* data file to be used
* @return Returns the new LinkedList of String type containing the word
* list
*/
public LinkedList<String> readWords(int wordLength) {
    int numberOfWords = 0;
    int numberOfRelevantWords = 0;
    String wordRead;
    boolean wordAlreadyStored;

    try {
        BufferedReader fileToBeRead = new BufferedReader(new FileReader("WordsOfLength" + wordLength + ".dat"));

        while (fileToBeRead.readLine() != null) { //Loops to the end of the file
            numberOfWords++;
        }

        fileToBeRead.close();

    } catch (IOException ex) {
        Logger.getLogger(WordReader.class.getName()).log(Level.SEVERE, null, ex);
        System.out.println("File could not be accessed.");
    }

    try {
        BufferedReader fileToBeRead = new BufferedReader(new FileReader("WordsOfLength" + wordLength + ".dat"));
        LinkedList<String> wordStore = new LinkedList<String>();

        for (int counter = 0; counter < numberOfWords; counter++) {
            if (numberOfRelevantWords == 0) {
                try {
                    wordRead = fileToBeRead.readLine();
                    if (wordRead.length() == wordLength) {
                        wordStore.add(wordRead); //Adds new word to the word store LinkedList
                    }
                }
            }
        }
    }
}
```



```

        } catch (IOException ex) {
            Logger.getLogger(WordReader.class.getName()).log(Level.SEVERE, null, ex);
        }
    } else {
        wordAlreadyStored = false;
        try {
            wordRead = fileToBeRead.readLine();
            if (wordRead.length() == wordLength){
                for (int counter2 = 0; counter2 < numberOfRelevantWords || wordAlreadyStored == true; counter2++)

                    if (wordStore.get(counter2).equals(wordRead)) { //Evaluates if word is already contained in the
word store LinkedList

                        wordAlreadyStored = true;
                    }
                }
            if (wordAlreadyStored == false) {
                wordStore.add(wordRead);
                numberOfRelevantWords++;
            }
        }
    } catch (IOException ex) {
        Logger.getLogger(WordReader.class.getName()).log(Level.SEVERE, null, ex);
    }
}
}
try {
    fileToBeRead.close();
    return wordStore;
} catch (IOException ex) {
    Logger.getLogger(WordReader.class.getName()).log(Level.SEVERE, null, ex);
}
} catch (IOException ex) {
    Logger.getLogger(WordReader.class.getName()).log(Level.SEVERE, null, ex);
}
return null;
}
}

```

Menu

```

package aber.dcs.cs21120.chs17.WordLadder.wordLadderFunctions;

import aber.dcs.cs21120.chs17.WordLadder.dataStructures.Graph;
import java.util.Scanner;

/**
 * CS21120 WordLadder menu launcher class
 *
 * @author Chris Savill - chs17
 */
public class Menu {
    ////////////////////////////////////////////////// Methods ///////////////////////////////////

    /**
     * Method to launch a menu
     */
    public void initialiseMenu() {
        Graph graph;
        int menuChoice = 0;
        Scanner input = new Scanner(System.in);

        System.out.println(
            "Welcome to WordLadder!");
        System.out.println(
            "-----");

        do {
            System.out.println("\nPlease select one of the following options: ");
            System.out.println("Option 1: Generate a word ladder.");
            System.out.println("Option 2: Discover the shortest word ladder between 2
words.");
            System.out.println("Option 3: Exit.\n");

            while (!input.hasNextInt()) {
                System.out.println("Please enter in a valid option.");
                input.next();
            }

            menuChoice = input.nextInt();

            switch (menuChoice) {
                case 1:
                    graph = new Graph();
                    graph.generateCycle();
                    break;
                case 2:
                    graph = new Graph();
                    graph.discoveryCycle();
                    break;
                case 3:
                    System.out.println("Exiting WordLadder...");
                    break;
                default:
                    System.out.println("Invalid option selected, please select a valid
option.");
            }
        } while (menuChoice != 3);
    }
}

```

```
        }  
    } while (menuChoice  
        != 3);  
}  
}
```

WordLadderDriver

```
package aber.dcs.cs21120.chs17.WordLadder.wordLadderFunctions;  
  
/**  
 * CS21120 WordLadder project main class  
 *  
 * @author Chris Savill - chs17  
 */  
public class WordLadderDriver {  
  
    /**  
     * Main method  
     *  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        Menu mainMenu = new Menu();  
        mainMenu.initialiseMenu();  
    }  
}
```