CS21120 Word Ladder Assignment Document

**Module Code:**

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

**Author:**

**Chris Savill –** [**chs17@aber.ac.uk**](mailto:chs17@aber.ac.uk)

Table of Contents

[Class Diagram 3](#_Toc338793770)

[Design 4](#_Toc338793771)

[Graph Class 4](#_Toc338793772)

[Vertex Class 4](#_Toc338793773)

[UnboundedStack and UnboundedQueue 4](#_Toc338793774)

[LinkedList and Node 5](#_Toc338793775)

[WordReader 5](#_Toc338793776)

[LadderCycler 5](#_Toc338793777)

[GenerateCycler and DiscoveryCycler 5](#_Toc338793778)

[GenerateLadder 5](#_Toc338793779)

[DiscoverLadder 5](#_Toc338793780)

[Menu 5](#_Toc338793781)

[WordLadderDriver 5](#_Toc338793782)

[Description of Algorithms 6](#_Toc338793783)

[Pseudo-code 7](#_Toc338793784)

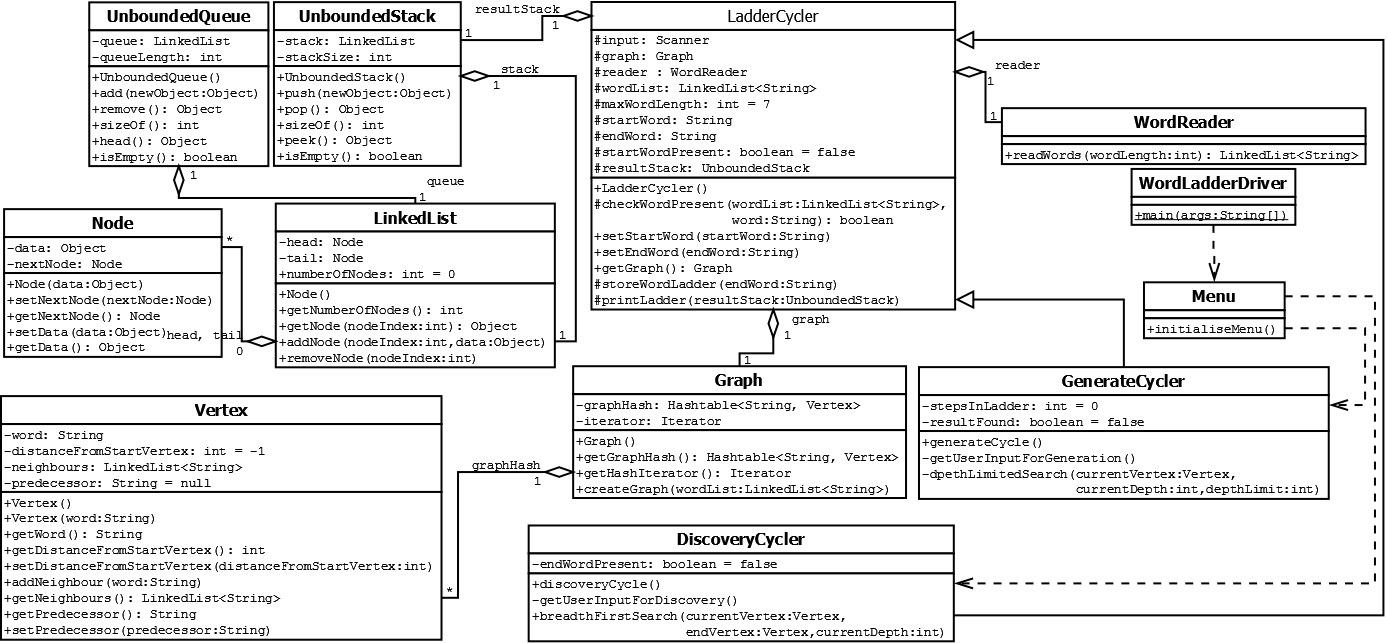
[Initial start-up 7](#_Toc338793785)

[Generation function: 7](#_Toc338793786)

[Java Source Code 8](#_Toc338793787)

[Testing 9](#_Toc338793788)

# Class Diagram



# Design

The design approach for this project was aimed to enforce low-coupling and high-cohesion, however, due to time constraints and problems encountered during the implementation, the program may not be as optimised for low-coupling and high-cohesion as it could be. 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 building/creating the graph as well as returning the hash table an iterator (used for iterating through the hash table) and the default constructor.

## 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.

## UnboundedStack and UnboundedQueue

The UnboundedStack and UnboundedQueue classes were implemented instead of the Stack and Queue classes provided by the java libraries because those classes contain many methods not required by this program. Also as both the UnboundedStack and UnboundedQueue classes were already built when worksheet 5 was completed , no extra work would be required to implement them over the over choices. The two classes used contained what was needed and not much else so it seemed appropriate to implement them over the other choices.

As the class diagram indicates, the UnboundedStack class is used by the DiscoveryCycler and GenerateCycler classes (through the super LadderCycler class) as a stack to store the result/ladder. The UnboundedQueue class is only used by the DiscoverLadder class as a queue data structure for the frontier of vertices/nodes to be expored.

## LinkedList and Node

These two classes were built in worksheet 4 and are used by the UnboundedStack and UnboundedQueue classes so are part of the design and choice of the data structures.

## WordReader

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

## LadderCycler

The LadderCycler abstract class is a super class of the DiscoveryCycler and GenerateCycler classes. This was created because there were many common variables and methods between the two sub classes so it made sense to encapsulate them in this abstract class to reduce code duplication. This also aids in high cohesion as all of the variables and methods relevant are contained in the one class, also to be used by the sub classes. As this class is acting like a hub for all of the main data structures to come together, it allows for all of the other classes to facilitate low coupling as they do not have to all connect to each other.

## GenerateCycler and DiscoveryCycler

These two classes as indicated in the class diagram are sub classes of the LadderCycler super class. They both inherit all of the variables and methods as well as incorporating their own variables and methods related to the generation/discovery functionality. They have methods to launch the generation/discovery cycle including calling other methods to get the user inputs for the purpose of the ladder generation/discovery and the search algorithms.

## 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.

## 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.

# 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 for each class in isolation and then testing of the actual algorithms through running the program and taking screenshots of the outputs.

# 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;  
  
/\*\*  
 \* **@author** Chris Savill - chs17 CS21120 WordLadder class that represents a graph  
 \* data structure  
 \*/  
public class **Graph** {  
 //////////////////////// Variables ///////////////////////////  
  
 /\*\*  
 \* 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;  
  
 //////////////////////// Constructors ///////////////////////////  
 /\*\*  
 \* Default constructor that just initialises the hash table  
 \*/  
 public **Graph**() {  
 graphHash = new Hashtable();  
 }  
  
 //////////////////////// Methods ///////////////////////////  
 /\*\* \* **Method to return the hash table**  
 \*  
 \* **@return** Returns the graphHash hash table  
 \*/  
 public Hashtable<String, Vertex> **getGraphHash**() {  
 return graphHash;  
 }  
  
 /\*\* \* **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**(LinkedList<String> wordList) {  
 String key;  
 Vertex vertex;  
 Vertex newVertex;  
 Iterator<Map.*Entry*<String, Vertex>> iterator;  
  
 for (int counter = 0; counter < wordList.size(); counter++) {  
 getGraphHash().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 = getGraphHash().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 works  
 \* 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;  
 }  
}

## Vertex

package aber.dcs.cs21120.chs17.WordLadder.dataStructures;  
  
import java.util.LinkedList;  
  
/\*\*  
 \* **@author** Chris Savill - chs17 CS21120 WordLadder class to represent a  
 \* vertex/node for use within the graph  
 \*/  
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 it 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;  
 }  
}

## LadderCycler

package aber.dcs.cs21120.chs17.WordLadder.wordLadderFunctions;  
  
import aber.dcs.cs21120.chs17.WordLadder.dataStructures.Graph;  
import aber.dcs.cs21120.chs17.WordLadder.dataStructures.UnboundedStack;  
import java.util.LinkedList;  
import java.util.Scanner;  
  
/\*\*  
 \* **@author** Chris Savill - chs17 CS21120 WordLadder abstract class from which the  
 \* DiscoveryCycler an GenerateCycler extend from  
 \*/  
public abstract class **LadderCycler** {  
  
 //////////////////////// Variables ///////////////////////////  
 /\*\*  
 \* Scanner class used for retrieving user input  
 \*/  
 protected Scanner input = new Scanner(System.*in*);  
 /\*\*  
 \* Graph class used for structuring a graph for use with the search  
 \* algorithm  
 \*/  
 protected Graph graph;  
 /\*\*  
 \* WordReader class used for reading in words from a data file  
 \*/  
 protected WordReader reader;  
 /\*\*  
 \* LinkedList of type String used to store all the words read in by the  
 \* WordReader class  
 \*/  
 protected 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  
 \*/  
 protected int maxWordLength = 7;  
 /\*\*  
 \* String class used to store the startWord  
 \*/  
 protected String startWord;  
 /\*\*  
 \* String class used to store the endWord  
 \*/  
 protected String endWord;  
 /\*\*  
 \* boolean primitive used to determine whether or not the start word is  
 \* present in the data file being scanned  
 \*/  
 protected boolean startWordPresent = false;  
 /\*\*  
 \* Stack class used for storing the resulting words in the word ladder  
 \*/   
 protected UnboundedStack resultStack;  
  
 //////////////////////// Constructors ///////////////////////////  
 /\*\*  
 \* Constructor to initialise variables  
 \*/  
 public **LadderCycler**() {  
 graph = new Graph();  
 reader = new WordReader();  
 wordList = new LinkedList();  
 resultStack = new UnboundedStack();  
 }  
  
 //////////////////////// Methods ///////////////////////////  
 /\*\* \* **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 he word list supplied  
 \*/  
 protected 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 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  
 \*/  
 protected void **storeWordLadder**(String endWord) {  
 String currentWord = endWord;  
 resultStack.push(endWord); //Adds the goal state word to the result stack  
  
 while (graph.getGraphHash().get(currentWord).getPredecessor() != null) { //Loops until hit start vertex as the start vertex would have no predecessor so would be null  
 resultStack.push(graph.getGraphHash().get(currentWord).getPredecessor()); //Adds predecessor to result stack  
 currentWord = graph.getGraphHash().get(currentWord).getPredecessor(); //Sets the current word to the predecessor vertex  
 }  
 }  
  
 /\*\*  
 \* Method to print the resulting ladder from the result stack  
 \*/  
 protected void **printLadder**(UnboundedStack resultStack) {  
 System.*out*.println("Word Ladder successfully generated.");  
 System.*out*.println("Number of words in ladder: " + resultStack.sizeOf());  
 while (!resultStack.isEmpty()) {  
 System.*out*.println(resultStack.pop().toString()); //Prints out the word ladder stack if successful  
 }  
 }  
}

## DiscoveryCycler

package aber.dcs.cs21120.chs17.WordLadder.wordLadderFunctions;  
  
import aber.dcs.cs21120.chs17.WordLadder.dataStructures.UnboundedQueue;  
import aber.dcs.cs21120.chs17.WordLadder.dataStructures.Vertex;  
  
/\*\*  
 \* **@author** Chris Savill - chs17 CS21120 WordLadder class to cycle through the  
 \* process of discovering a word ladder between two words  
 \*/  
public class **DiscoveryCycler** extends LadderCycler {  
 //////////////////////// Variables ///////////////////////////  
  
 /\*\*  
 \* boolean primitive used to store whether or not the end word is present in  
 \* the data file being scanned  
 \*/  
 private boolean endWordPresent = false;  
  
 //////////////////////// Methods ///////////////////////////  
 /\*\*  
 \* 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();  
 graph.createGraph(wordList);  
  
 if (breadthFirstSearch(graph.getGraphHash().get(startWord), graph.getGraphHash().get(endWord), 0) == true) { //Evaluates the result of the iterativeDeepeningSearchForDiscovery method  
 storeWordLadder(endWord);  
 printLadder(resultStack);  
 } 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**() {  
 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 {  
 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) {  
 UnboundedQueue frontier = new UnboundedQueue();  
 currentVertex.setDistanceFromStartVertex(currentDepth); //Sets distance from start vertex to the current depth, if it is the start vertex, distance would be 0  
 frontier.add(currentVertex.getWord()); //Adds the current vertex to the queue  
  
 while (!frontier.isEmpty()) { //Evaluates if the frontier queue is not empty  
 if (currentVertex.getWord().equals(endVertex.getWord())) { //Checks if goal state has been met  
 endWord = currentVertex.getWord();  
 return true;  
 } else {  
 currentVertex = graph.getGraphHash().get(frontier.head().toString()); //Sets the current vertex to the vertex at the front of the queue  
 frontier.remove(); //Removes the current vertex from the frontier queue, (counted as explored)  
  
 for (String neighbour : graph.getGraphHash().get(currentVertex.getWord()).getNeighbours()) {  
 if (graph.getGraphHash().get(neighbour).getDistanceFromStartVertex() < 0) { //Evaluates if the vertexes have been explored  
 frontier.add(graph.getGraphHash().get(neighbour).getWord()); //Adds neighbour/child vertex to end of queue  
 graph.getGraphHash().get(neighbour).setDistanceFromStartVertex(currentDepth + 1); //Sets the distance from start vertex to the next depth level  
 graph.getGraphHash().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  
 }  
}

## GenerateCycler

package aber.dcs.cs21120.chs17.WordLadder.wordLadderFunctions;  
  
import aber.dcs.cs21120.chs17.WordLadder.dataStructures.Vertex;  
  
/\*\*  
 \* **@author** Chris Savill - chs17 CS21120 WordLadder class to cycle through the  
 \* process of generating a word ladder of a certain depth from a word  
 \*/  
public class **GenerateCycler** extends LadderCycler {  
 //////////////////////// Variables ///////////////////////////  
  
 /\*\*  
 \* int primitive used to store the number of steps in the ladder to generate  
 \*/  
 private int stepsInLadder = 0;  
 /\*\*  
 \* boolean primitive used to determine whether or not the result has been  
 \* found  
 \*/  
 private boolean resultFound = false;  
  
 //////////////////////// Methods ///////////////////////////  
 /\*\*  
 \* 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**() {  
 getUserInputForGeneration();  
 graph.createGraph(wordList);  
  
 if (depthLimitedSearch(graph.getGraphHash().get(startWord), 0, stepsInLadder - 1) == true) { //Evaluates the result of the recursiveDepthLimitedSearchForGeneration method  
 storeWordLadder(endWord);  
 printLadder(resultStack);  
 } else {  
 System.*out*.println("Sorry no complete word ladder for '" + startWord + "' with '" + stepsInLadder + "' steps in the ladder.");  
 }  
 }  
  
 /\*\*  
 \* 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 number of steps the want to ladder  
 \*/  
 private void **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();  
 }  
  
 stepsInLadder = 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) {  
 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 : graph.getGraphHash().get(currentVertex.getWord()).getNeighbours()) {  
 if (graph.getGraphHash().get(neighbour).getDistanceFromStartVertex() < 0) {  
 graph.getGraphHash().get(neighbour).setPredecessor(currentVertex.getWord()); //Sets the predecessor/parent vertex of the neighbour/child vertex to the current vertex  
 resultFound = depthLimitedSearch(graph.getGraphHash().get(neighbour), currentDepth + 1, depthLimit);  
 if (resultFound == true) {  
 return true;  
 }  
 }  
 }  
 }  
 }  
 return false;  
 }  
}

## WordReader

package aber.dcs.cs21120.chs17.WordLadder.wordLadderFunctions;  
  
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;  
  
/\*\*  
 \* **@author** Chris Savill - chs17 Class that contains various methods to read  
 \* words from a file  
 \*/  
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);  
 }  
  
 try {  
 BufferedReader fileToBeRead = new BufferedReader(new FileReader("WordsOfLength" + wordLength + ".dat"));  
 LinkedList<String> wordStore = new LinkedList();  
  
 for (int counter = 0; counter < numberOfWords; counter++) {  
 if (numberOfRelevantWords == 0) {  
 try {  
 wordStore.add(fileToBeRead.readLine()); //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();  
  
 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;  
 }  
}

## UnboundedStack

package aber.dcs.cs21120.chs17.WordLadder.dataStructures;  
  
/\*\*  
 \* **@author** Chris Savill - chs17  
 \* Dynamic Stack class  
 \*/  
public class **UnboundedStack** {  
 //////////////////////// Variables ///////////////////////////  
  
 /\*\*  
 \* LinkedList used to allow the stack to change its size dynamically  
 \*/  
 private LinkedList stack;  
 /\*\*  
 \* int primitive used to store the size of the stack  
 \*/  
 private int stackSize = 0;  
  
 //////////////////////// Constructors ///////////////////////////  
 /\*\*  
 \* Default constructor that just initialises the LinkedList  
 \*/  
 public **UnboundedStack**() {  
 stack = new LinkedList();  
 }  
  
 //////////////////////// Methods ///////////////////////////  
 /\*\* \* **Method to add a new node to the top of the stack**  
 \*  
 \* **@param** newObject The object of data to be stored in the node  
 \*/  
 public void **push**(Object newObject) {  
 stack.addNode(0, newObject);  
 stackSize++;  
 }  
  
 /\*\* \* **Method to return the node at the top of the stack**  
 \*  
 \* **@return** Returns the data from the node at the top of the stack  
 \*/  
 public Object **pop**() {  
 Object objectToBeReturned = stack.getNode(0);  
 stack.removeNode(0);  
 stackSize--;  
 return objectToBeReturned;  
 }  
  
 /\*\* \* **Method to return the size of the stack**  
 \*  
 \* **@return** Returns the size of the stack  
 \*/  
 public int **sizeOf**() {  
 return this.stackSize;  
 }  
  
 /\*\* \* **Method to return what is on the top of the stack without removing the** \* **node**  
 \*  
 \* **@return** Returns the data from the node at the top of the stack  
 \*/  
 public Object **peek**() {  
 return stack.getNode(0);  
 }  
  
 /\*\* \* **Method to return true/false depending on if the stack is empty or not**  
 \*  
 \* **@return** Returns a boolean  
 \*/  
 public boolean **isEmpty**() {  
 return stackSize == 0;  
 }  
}

## UnboundedQueue

package aber.dcs.cs21120.chs17.WordLadder.dataStructures;  
  
/\*\*  
 \* **@author** Chris Savill - chs17  
 \* Dynamic Queue class  
 \*/  
public class **UnboundedQueue** {  
 //////////////////////// Variables ///////////////////////////  
  
 /\*\*  
 \* LinkedList used to allow the queue to change its size dynamically  
 \*/  
 private LinkedList queue;  
 /\*\*  
 \* int primitive used to store the size of the stack  
 \*/  
 private int queueLength = 0;  
  
 //////////////////////// Constructors ///////////////////////////  
 /\*\*  
 \* Default constructor that just initialises the LinkedList  
 \*/  
 public **UnboundedQueue**() {  
 queue = new LinkedList();  
 }  
  
 //////////////////////// Methods ///////////////////////////  
 /\*\* \* **Method to add a new node to the front of the queue**  
 \*  
 \* **@param** newObject The object of data to be stored in the node  
 \*/  
 public void **add**(Object newObject) {  
 queue.addNode(queueLength, newObject);  
 queueLength++;  
 }  
  
 /\*\* \* **Method to return the node at the front of the queue**  
 \*  
 \* **@return** Returns the data from the node at the front of the queue  
 \*/  
 public Object **remove**() {  
 Object objectToBeReturned = queue.getNode(0);  
 queue.removeNode(0);  
 queueLength--;  
 return objectToBeReturned;  
 }  
  
 /\*\* \* **Method to return the size of the queue**  
 \*  
 \* **@return** Returns the size of the queue  
 \*/  
 public int **sizeOf**() {  
 return this.queueLength;  
 }  
  
 /\*\* \* **Method to return what is at the front of the queue without removing the** \* **node**  
 \*  
 \* **@return** Returns the data from the node at the front of the queue  
 \*/  
 public Object **head**() {  
 return queue.getNode(0);  
 }  
  
 /\*\* \* **Method to return true/false depending on if the queue is empty or not**  
 \*  
 \* **@return** Returns a boolean  
 \*/  
 public boolean **isEmpty**() {  
 return queueLength == 0;  
 }  
}

## LinkedList

package aber.dcs.cs21120.chs17.WordLadder.dataStructures;  
  
/\*\*  
 \* **@author** Chris Savill - chs17  
 \* LinkedList class that expands dynamically when needed  
 \*/  
public class **LinkedList** {  
 //////////////////////// Variables ///////////////////////////  
  
 /\*\*  
 \* Node class used to store the head of the LinkedList  
 \*/  
 private Node head;  
 /\*\*  
 \* Node class used to store the tail of the LinkedList  
 \*/  
 private Node tail;  
 /\*\*  
 \* int primitive used to store the number of nodes in the LinkedList  
 \*/  
 private int numberOfNodes = 0;  
  
 //////////////////////// Methods ///////////////////////////  
 /\*\* \* **Method to return the number of nodes in the LinkedList**  
 \*  
 \* **@return** Returns the numberOfNodes  
 \*/  
 public int **getNumberOfNodes**() {  
 return this.numberOfNodes;  
 }  
  
 /\*\* \* **Method to return the data in node requested**  
 \*  
 \* **@param** nodeIndex The index of the node to be returned  
 \* **@return** Returns an Object which is the data contained in the node  
 \* requested  
 \*/  
 public Object **getNode**(int nodeIndex) {  
 Node currentNode = head;  
  
 if (nodeIndex == numberOfNodes - 1) {  
 return (Comparable) tail.getData();  
 } else {  
 for (int counter = 1; counter <= nodeIndex; counter++) {  
 currentNode = currentNode.getNextNode();  
 }  
 return (Comparable) currentNode.getData();  
 }  
 }  
  
 /\*\* \* **Method to add a new node to the LinkedList at the position passed in,** \* **initialising the node with data passed in**  
 \*  
 \* **@param** nodeIndex The position at which the node will be added  
 \* **@param** data The data that the new node will contain  
 \*/  
 public void **addNode**(int nodeIndex, Object data) {  
 Node newNode = new Node(data);  
  
 if (numberOfNodes == 0) {  
 head = newNode;  
 tail = newNode;  
 } else if (nodeIndex == 0) {  
 newNode.setNextNode(head);  
 head = newNode;  
 } else if (nodeIndex == numberOfNodes) {  
 tail.setNextNode(newNode);  
 tail = newNode;  
 } else {  
 Node currentNode = head;  
 for (int counter = 0; counter < nodeIndex - 1; counter++) {  
 currentNode = currentNode.getNextNode();  
 }  
 newNode.setNextNode(currentNode.getNextNode());  
 currentNode.setNextNode(newNode);  
 }  
 numberOfNodes++;  
 }  
  
 /\*\* \* **Method to delete a node from the LinkedList**  
 \*  
 \* **@param** nodeIndex The index of the node to be deleted  
 \*/  
 public void **removeNode**(int nodeIndex) {  
 if (1 == numberOfNodes) {  
 head = null;  
 tail = null;  
 } else if (0 == nodeIndex) {  
 head = head.getNextNode();  
 } else {  
 Node currentNode = head;  
 for (int counter = 0; counter < nodeIndex - 1; counter++) {  
 currentNode = currentNode.getNextNode();  
 }  
 Node nextNode = currentNode.getNextNode();  
 currentNode.setNextNode(nextNode.getNextNode());  
 if (tail == nextNode) {  
 tail = currentNode;  
 }  
 }  
 numberOfNodes--;  
 }  
}

## Node

package aber.dcs.cs21120.chs17.WordLadder.dataStructures;  
  
/\*\*  
 \* **@author** Chris Savill - chs17  
 \* Node class for use within LinkedList class  
 \*/  
public class **Node** {  
 //////////////////////// Variables ///////////////////////////  
  
 /\*\*  
 \* Object class used to store any data that the node can contain  
 \*/  
 private Object data;  
 /\*\*  
 \* Node class used to store the next node  
 \*/  
 private Node nextNode;  
  
 //////////////////////// Constructors ///////////////////////////  
 /\*\* \* **Constructor that takes the data object passed in and assigns it to the** \* **data variable**  
 \*  
 \* **@param** data The data to be stored in the node  
 \*/  
 public **Node**(Object data) {  
 this.data = data;  
 }  
  
 /\*\* \* **Method to set the next node of the node**  
 \*  
 \* **@param** nextNode The next node  
 \*/  
 public void **setNextNode**(Node nextNode) {  
 this.nextNode = nextNode;  
 }  
  
 /\*\* \* **Method to return the next node**  
 \*  
 \* **@return** Returns the next node  
 \*/  
 public Node **getNextNode**() {  
 return this.nextNode;  
 }  
  
 /\*\* \* **Method to set the data of the node**  
 \*  
 \* **@param** data The data to be stored in the node  
 \*/  
 public void **setData**(Object data) {  
 this.data = data;  
 }  
  
 /\*\* \* **Method to return the data in the node**  
 \*  
 \* **@return** Returns the data of the node  
 \*/  
 public Object **getData**() {  
 return this.data;  
 }  
}

## Menu

package aber.dcs.cs21120.chs17.WordLadder.wordLadderFunctions;  
  
import java.util.Scanner;  
  
/\*\*  
 \* **@author** Chris Savill - chs17 CS21120 WordLadder menu launcher class  
 \*/  
public class **Menu** {  
 //////////////////////// Methods ///////////////////////////  
  
 /\*\*  
 \* Method to launch a menu  
 \*/  
 public void **initialiseMenu**() {  
  
 int menuChoice = 0;  
 GenerateCycler generateCycler;  
 DiscoveryCycler discoveryCycler;  
 Scanner input = new Scanner(System.*in*);  
  
 System.*out*.println(  
 "Welcome to WordLadder!");  
 System.*out*.println(  
 "----------------------");  
  
 do {  
 System.*out*.println("**\n**Please 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:  
 generateCycler = new GenerateCycler();  
 generateCycler.generateCycle();  
 break;  
 case 2:  
 discoveryCycler = new DiscoveryCycler();  
 discoveryCycler.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);  
 }  
}

## WordLadderDriver

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