Assignment Four

Shannon Cordoni
Shannon.Cordoni@Marist.edu

November 20, 2021

1 Problem One: Graphs and Data Trees

1.1 The Data Structure

Given multiple text files we were to create different graph representations and a binary search tree. First taking the list of strings our job was to create an algorithm to go through this list and add them to a binary search tree to then later traverse through to search for 42 items. Then taking the list of graph instructions we were to create vertices and edges to form an undirected graph in the form of a matrix, adjacency list, and linked objects.

1.2 Main Class

1.2.1 Description

This class is where most of our work is done, it contains multiple scanners to read in our multiple files. These files include the *magicitems.txt* file, the $magicitems - find - in - bst.txt$ file, and $graphs1.txt$ file. Using these files we input each line of the $graphs1.txt$ into an instruction array to be split up into a 2 dimensional array to access each index. We then use this split array to create the matrix, adjacency list, and linked object representation for each graph. The *magicitems.txt* file was also read into an array to be passed to the *insertTree* method to put each line into the binary search tree, this tree could then be traversed through to search for specific items. The methods contained in this class preform all these operations and more.

```

/*
 *
 * Assignment 4
 * Due Date and Time: 11/19/21 before 12:00am
 * Purpose: to implement graph and tree data structures, and to understand the performance of
 *their traversals.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output graph and tree data structures.
 * @author Shannon Cordoni
 *
```

```java
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.*;
import java.util.Scanner;



public class Assignment4Cordoni {

    //Declare keyboard
    static Scanner keyboard = new Scanner(System.in);

    public static void main(String[] args) {

        //Declare and initialize variables
        String line;

        //For binary search tree
        String[] wordarray = new String[666];

        String[] instructionarray = new String[375];
        String[] graph1array = new String[20];
        String[] graph2array = new String[38];
        String[] graph3array = new String[127];
        String[] graph4array = new String[138];
        String[] graph5array = new String[48];

        String[] searcharray = new String[42];
        String[][] splitinstructionarray = new String[375][8];
        String[][] graph1splitarray = new String[20][8];
        String[][] graph2splitarray = new String[38][8];
        String[][] graph3splitarray = new String[127][8];
        String[][] graph4splitarray = new String[138][8];
        String[][] graph5splitarray = new String[48][8];
        TreeCordoni tree = new TreeCordoni();

        int numberOfLookupComparisons = 0;
        int averagenumberOfLookupComparisons = 0;


        //Reads in the magic items file for the binary search tree
        //create new file object
        File myFile = new File("magicitems.txt");

        try
        {
            //create scanner
            Scanner input = new Scanner(myFile);
            line = null;

            int i = 0;

            //while there are more lines in the file it inputs them into a word array
            while(input.hasNext())
            {
                //Input into array
                wordarray[i] = input.nextLine();
                i++;
            }//while

            input.close();
```

```
        }//try

        //error for file not found
        catch(FileNotFoundException ex)
        {
          System.out.println("Failed to find file: " + myFile.getAbsolutePath());
        }//catch

        //Error in case of a null pointer exception
        catch(NullPointerException ex)
        {
            System.out.println("Null pointer exception.");
            System.out.println(ex.getMessage());
        }//catch

        //General error message
        catch(Exception ex)
        {
            System.out.println("Something went wrong");
            ex.printStackTrace();
        }//catch

        //Reads in the graph file to create graphs
        //create new file object
        File myFile1 = new File("graphs1.txt");

        try
        {
            //create scanner
            Scanner input = new Scanner(myFile1);
            line = null;

            int i = 0;

            while(input.hasNext()){

                instructionarray[i] = input.nextLine();
                i++;
            }


            input.close();

        }//try

        //error for file not found
        catch(FileNotFoundException ex)
        {
          System.out.println("Failed to find file: " + myFile.getAbsolutePath());
        }//catch

        //Error in case of a null pointer exception
        catch(NullPointerException ex)
        {
            System.out.println("Null pointer exception.");
            System.out.println(ex.getMessage());
        }//catch

        //General error message
        catch(Exception ex)
        {
            System.out.println("Something went wrong");
            ex.printStackTrace();
        }//catch
```

```


        //Reads in the magic items to find in the binary search tree
        //create new file object
        File myFile2 = new File("magicitems-find-in-bst.txt");

        try
        {
            //create scanner
            Scanner input = new Scanner(myFile2);
            line = null;

            int i = 0;

             //while there are more lines it inputs them into an instruction array
             while(input.hasNext())
             {
                //Input into array
                searcharray[i] = input.nextLine();
                i++;
             }//while

            input.close();

        }//try

        //error for file not found
        catch(FileNotFoundException ex)
        {
          System.out.println("Failed to find file: " + myFile.getAbsolutePath());
        }//catch

        //Error in case of a null pointer exception
        catch(NullPointerException ex)
        {
            System.out.println("Null pointer exception.");
            System.out.println(ex.getMessage());
        }//catch

        //General error message
        catch(Exception ex)
        {
            System.out.println("Something went wrong");
            ex.printStackTrace();
        }//catch

        // i know this is not exactly how I was supposed to split up the text
        //file, but this was the only way I could get it to work with everything

        //Print to check array
        for (int i = 0; i < instructionarray.length; i++){
           // System.out.println(instructionarray[i]);
        }//for

        //Put instructions into graph 1 array
        for (int i = 0; i < graph1array.length; i++){
            graph1array[i] = instructionarray[i];
        }//for

        //Put instructions into graph 2 array
        for (int i = 0; i < graph2array.length; i++){
            graph2array[i] = instructionarray[i + 22];
        }//for

        //Put instructions into graph 3 array
```

```
        for (int i = 0; i < graph3array.length; i++){
            graph3array[i] = instructionarray[i + 61];
        }//for

        //Put instructions into graph 4 array
        for (int i = 0; i < graph4array.length; i++){
            graph4array[i] = instructionarray[i + 189];
        }//for

        //Put instructions into graph 5 array
        for (int i = 0; i < graph5array.length; i++){
            graph5array[i] = instructionarray[i + 327];
        }//for

        //split up into 2D array
        for (int i = 0; i < splitinstructionarray.length; i++){
            for( int j = 0; j < splitinstructionarray[i].length; j++){
            splitinstructionarray[i] = instructionarray[i].split(" ");
            }//for j
        }//for

        //split up into 2D array for graph 1
        for (int i = 0; i < graph1splitarray.length; i++){
            for( int j = 0; j < graph1splitarray[i].length; j++){
            graph1splitarray[i] = graph1array[i].split(" ");
            }//for j
        }//for

        //split up into 2D array for graph 2
        for (int i = 0; i < graph2splitarray.length; i++){
            for( int j = 0; j < graph2splitarray[i].length; j++){
            graph2splitarray[i] = graph2array[i].split(" ");
            }//for j
        }//for

        //split up into 2D array for graph 3
        for (int i = 0; i < graph3splitarray.length; i++){
            for( int j = 0; j < graph3splitarray[i].length; j++){
            graph3splitarray[i] = graph3array[i].split(" ");
            }//for j
        }//for

        //split up into 2D array for graph 4
        for (int i = 0; i < graph4splitarray.length; i++){
            for( int j = 0; j < graph4splitarray[i].length; j++){
            graph4splitarray[i] = graph4array[i].split(" ");
            }//for j
        }//for

        //split up into 2D array for graph 5
        for (int i = 0; i < graph5splitarray.length; i++){
            for( int j = 0; j < graph5splitarray[i].length; j++){
            graph5splitarray[i] = graph5array[i].split(" ");
            }//for j
        }//for

        //Graphs!!

        System.out.println("Graph 1");

        //making the matrix
        makeMatrix(graph1splitarray);

        //make the adjacency list
        makeAdjacencyList(graph1splitarray);
```

```
        //make linked list
        makeLinkedObjects(graph1splitarray);

        //breadth first traversal
        breadthTraversal(makeLinkedObjects(graph1splitarray));

        //reset process status
        reboot(makeLinkedObjects(graph1splitarray));

        System.out.println(" ");
        System.out.println("Depth Traversal");

        //depth first traversal
        depthTraversal(makeLinkedObjects(graph1splitarray));


        System.out.println(" ");
        System.out.println("Graph 2");

        //making the matrix
        makeMatrix(graph2splitarray);

        //make the adjacency list
        makeAdjacencyList(graph2splitarray);

        //make linked list
        makeLinkedObjects(graph2splitarray);

        //breadth first traversal
        breadthTraversal(makeLinkedObjects(graph2splitarray));

        //reset process status
        reboot(makeLinkedObjects(graph2splitarray));

        System.out.println(" ");
        System.out.println("Depth Traversal");

        //depth first traversal
        depthTraversal(makeLinkedObjects(graph2splitarray));


        System.out.println(" ");
        System.out.println("Graph 3");

        //making the matrix
        makeMatrix(graph3splitarray);

        //make the adjacency list
        makeAdjacencyList(graph3splitarray);

        //make linked list
        makeLinkedObjects(graph3splitarray);

        //breadth first traversal
        breadthTraversal(makeLinkedObjects(graph3splitarray));

        //reset process status
        reboot(makeLinkedObjects(graph3splitarray));

        System.out.println(" ");
        System.out.println("Depth Traversal");

        //depth first traversal
        depthTraversal(makeLinkedObjects(graph3splitarray));
```

```
        System.out.println(" ");
        System.out.println("Graph 4");

        //making the matrix
        makeMatrix(graph4splitarray);

        //make the adjacency list
        makeAdjacencyList(graph4splitarray);

        //make linked list
        makeLinkedObjects(graph4splitarray);

        //breadth first traversal
        breadthTraversal(makeLinkedObjects(graph4splitarray));

        //reset process status
        reboot(makeLinkedObjects(graph4splitarray));

        System.out.println(" ");
        System.out.println("Depth Traversal");

        //depth first traversal
        depthTraversal(makeLinkedObjects(graph4splitarray));


        System.out.println(" ");
        System.out.println("Graph 5");

        //making the matrix
        makeMatrix(graph5splitarray);

        //make the adjacency list
        makeAdjacencyList(graph5splitarray);

        //make linked list
        makeLinkedObjects(graph5splitarray);

        //breadth first traversal
        breadthTraversal(makeLinkedObjects(graph5splitarray));

        //reset process status
        reboot(makeLinkedObjects(graph5splitarray));

        System.out.println(" ");
        System.out.println("Depth Traversal");

        //depth first traversal
        depthTraversal(makeLinkedObjects(graph5splitarray));



        //Binary Search Trees!!

        //I know this does not print out the binary search tree correctly but if
        // you uncomment line 854 it looks as though different numbers should be returned
        //for the comparison.

        System.out.println(" ");
        System.out.println(" ");
        System.out.println(" Insert the Magic Items into the Tree ");


        //insert the word array into the tree
```

```java
        for (int i = 0; i < wordarray.length; i++){
            insertTree(tree, wordarray[i]);
        }//for

        System.out.println(" ");
        System.out.println(" ");
        System.out.println(" Search for the Magic Items in the Tree ");

        int comparisons = 0;

        //Search for the 42 magic items
        for (int i = 0; i < searcharray.length; i++){

            System.out.println("Number of Comparisons: " + searchTree(tree.getRoot(),
                                    searcharray[i], comparisons));
            numberOfLookupComparisons = numberOfLookupComparisons +
            searchTree(tree.getRoot(), searcharray[i], comparisons);

        }//for

        //get the average lookup comparisons
        averagenumberOfLookupComparisons = numberOfLookupComparisons/searcharray.length;

        System.out.println("Average lookup: " + averagenumberOfLookupComparisons);

    }//main

    //This method creates the matrix of the undirected graph
    public static void makeMatrix(String[][] instructions) {

        //instantiate matrix
        int length = 1;
        int height = 1;

        for (int i = 0; i < instructions.length; i++){

            for(int j = 0; j < instructions[i].length; j ++){

                //skip comment line
                if(instructions[i][j].compareToIgnoreCase("--")==0){
                    System.out.println(" ");
                }//if

                //increment length and heigh to get matrix dimensions
                else if (instructions[i][j].compareToIgnoreCase("vertex")==0){
                    length++;
                    height++;

                }//else if

            }//for j

        }//for i

        //create matrix
        int[][] matrix = new int[length][height];

        //System.out.println("length: " + length);
        //System.out.println("height: " + height);

        //loop through to add value at correct matrix location
        for (int i = 0; i < instructions.length; i++){

            for(int j = 0; j < instructions[i].length; j ++){

```

```
                if (instructions[i][j].compareToIgnoreCase("edge")==0){

                    //grab index 2 make it length and grab index 4 and make it height
                    //System.out.println("index 2: "+ Integer.valueOf(instructions[i][j+1]));

                    //grab index 4 make it length and grab index 2 and make it height
                    //System.out.println("index 4: " + Integer.valueOf(instructions[i][j+3]));

                    matrix[Integer.valueOf(instructions[i][j + 1])][
                    Integer.valueOf(instructions[i][j + 3])] = 1;

                    matrix[Integer.valueOf(instructions[i][j + 3])]
                    [Integer.valueOf(instructions[i][j + 1])] = 1;

                }//if

            }
        }

        //print out the matrix
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {

                System.out.print(matrix[i][j] + " ");

            }//for j

            System.out.println();

        }//for i
    }//make Matrix

    //This method creates the adjacency list of the undirected graph
    public static void makeAdjacencyList(String[][] instructions) {

        System.out.println(" ");

        int height = 1;

        for (int i = 0; i < instructions.length; i++){

            for(int j = 0; j < instructions[i].length; j ++){

                //skip comment line
                if(instructions[i][j].compareToIgnoreCase("--")==0){
                    System.out.println(" ");
                }//if

                //increment height to create arraylist
                else if (instructions[i][j].compareToIgnoreCase("vertex")==0){
                    height++;
                }//else

            }//for j
        }//for i

        //create arraylist
        ArrayList<ArrayList<Integer>> adjlist = new ArrayList<>(height);

        //add arraylist at each index
        for(int i=0; i < height; i++) {
            adjlist.add(new ArrayList());
        }//for

        //add neighbors to arraylist
```

```
        for (int i = 0; i < instructions.length; i++){

            for(int j = 0; j < instructions[i].length; j ++){

                if (instructions[i][j].compareToIgnoreCase("edge")==0){

                    //grab index 2 and add 4 to arraylist
                    //System.out.println(instructions[i][j + 1]);

                    adjlist.get(Integer.parseInt(instructions[i][j + 1])).add
                    (Integer.parseInt(instructions[i][j + 3]));

                    //grab index 4 and add 2 to arraylist
                    //System.out.println(instructions[i][j + 3]);

                    adjlist.get(Integer.parseInt(instructions[i][j + 3])).add
                    (Integer.parseInt(instructions[i][j + 1]));

                }//else

            }//for j
        }//for i

        //print out arraylist
        for (int i = 0; i < instructions.length; i++){

            for(int j = 0; j < instructions[i].length; j ++){

                if (instructions[i][j].compareToIgnoreCase("vertex")==0){

                    System.out.println("[" + instructions[i][j + 1] + "]" +
                    adjlist.get(Integer.parseInt(instructions[i][j + 1])));

                }//else

            }//for j
        }//for i

    }//make adjacency list

    //This method creates the linked objects of the undirected graph
    public static VertexCordoni makeLinkedObjects(String[][] instructions) {


        int index = 0;
        VertexCordoni[] vertexlist;

        //increment index to create vertex array
        for (int i = 0; i < instructions.length; i++){

            if (instructions[i][0].compareToIgnoreCase("add")==0){

                if(instructions[i][1].compareToIgnoreCase("vertex")==0){

                    //System.out.println("Id: " + instructions[i][2]);
                    //System.out.println("Id get: " + vertex.getId());
                    index++;
                }//if

            }//else if

        }//for i

        //create vertex array
        vertexlist = new VertexCordoni[index];
```

```
        int j = 0;

        //create neighbor array
        for (int i = 0; i < instructions.length; i++){

            if (instructions[i][0].compareToIgnoreCase("add")==0){

                //create new vertex and set id to add to vertex array
                if(instructions[i][1].compareToIgnoreCase("vertex")==0){

                    VertexCordoni vertex = new VertexCordoni();

                    vertex.setId(instructions[i][2]);

                    vertexlist[j] = vertex;
                    //System.out.println(vertexlist[j]);
                    j++;


                }//if

                //add edge to neighbor array
                else if(instructions[i][1].compareToIgnoreCase("edge")==0){

                    for(int k = 0; k < vertexlist.length; k++){
                       // System.out.println(vertexlist[k].getId());

                        //if the vertex is in the vertex array then add new edge
                        if(vertexlist[k].getId().compareToIgnoreCase(instructions[i][2])==0){
                            //System.out.println("hello1");

                            for(int l = 0; l < vertexlist.length; l++){
                                //System.out.println("hello2");

                                //if the 2nd vertex is in the vertex array then add the first
                                //vertex to their neighbor array
                                if (vertexlist[l].getId().compareToIgnoreCase
                                (instructions[i][4])==0){

                                    vertexlist[k].neighbors.add(vertexlist[l]);
                                    //System.out.println("k" + vertexlist[k].getId());

                                    vertexlist[l].neighbors.add(vertexlist[k]);
                                    //System.out.println("l" +vertexlist[l].getId());

                                }//if

                            }//for

                        }//if

                    }//for

                }//else if

            }//if

        }//for i

        //print neighbor array size to check
        for(int i = 0; i < vertexlist.length; i++){
            //System.out.println("size " + vertexlist[i].neighbors.size());
        }

```

```java

        //return
        return vertexlist[0];

    }//make linked objects


    //Searching far and wide!

    //this method preforms the breadth traversal
    public static void breadthTraversal( VertexCordoni vertex) {

        System.out.println(" ");
        System.out.println("Breadth Traversal");

        QueueVertexCordoni thequeue = new QueueVertexCordoni();

        VertexCordoni currentvertex;

        thequeue.enqueue(vertex);

        vertex.setProcessStatus(true);

        while(!(thequeue.isEmpty())){

            currentvertex = thequeue.dequeue();

            System.out.println("Id " + currentvertex.getId());


            for(int i = 0 ; i < currentvertex.neighbors.size() ; i++){


                if ( currentvertex.neighbors.get(i).getProcessStatus() == false){

                   thequeue.enqueue(currentvertex.neighbors.get(i));
                   currentvertex.neighbors.get(i).setProcessStatus(true);

                }//if
            }//for


        }//while

    }//breadth Traversal

    //this method resets the process status for depth traversal
    public static void reboot( VertexCordoni vertex) {

        System.out.println(" ");
        System.out.println("Reset Processed Status for Depth Traversal");

        QueueVertexCordoni thequeue = new QueueVertexCordoni();

        VertexCordoni currentvertex;

        thequeue.enqueue(vertex);

        vertex.setProcessStatus(false);

        while(!(thequeue.isEmpty())){

            currentvertex = thequeue.dequeue();


```

```
            for(int i = 0 ; i < currentvertex.neighbors.size() ; i++){

                if ( currentvertex.neighbors.get(i).getProcessStatus() == true){

                   thequeue.enqueue(currentvertex.neighbors.get(i));
                   currentvertex.neighbors.get(i).setProcessStatus(false);

                }//if
            }//for


        }//while

    }//reboot

    //this method preforms the depth traversal
    public static void depthTraversal(VertexCordoni vertex) {

        if((vertex.getProcessStatus() == false)){

            System.out.println("Id: " + vertex.getId());
            vertex.setProcessStatus(true);

        }//if

        //System.out.println("Size: " + vertex.neighbors.size());

        for(int i = 0; i < vertex.neighbors.size(); i++){

            if(vertex.neighbors.get(i).getProcessStatus() ==  false){
                depthTraversal(vertex.neighbors.get(i));
            }//if

        }//for

    }//depth Traversal



    //lets make the trees!

    //This method inserts the nodes into the tres
    public static void insertTree(TreeCordoni tree, String word) {

        TreeCordoni newnode = new TreeCordoni();

        newnode.setData(word);

        TreeCordoni trailing = null;

        //sets current to the tree root
        TreeCordoni current = tree.getRoot();

        //while the root is not null continue down the tree
        while (current != null){

            trailing = current;

            if(newnode.getData().compareToIgnoreCase(current.getData()) < 0){

                current = current.getLeft();
                //System.out.println("L ");


            }//if
```

```
            else{

                current = current.getRight();
                //System.out.println("R ");

            }//else

        }//while

        newnode.setParent(trailing);

        //if trailing is null then set the new node to the root
        if(trailing == null){

            tree.setRoot(newnode);
            System.out.println("Root: " + newnode.getData());

        }//if

        //else we set the new node in the tree
        else{

            if(newnode.getData().compareToIgnoreCase(trailing.getData()) < 0){

                trailing.setLeft(newnode);
                System.out.println("L ");

                //to print!
                printTree(tree.getRoot());

            }//if

            else{

                trailing.setRight(newnode);
                System.out.println("R ");

                //to print!
                printTree(tree.getRoot());

            }//else
        }//else


    }//insertTree


    //This method prints the tree (kind of)
    public static void printTree(TreeCordoni root) {

        if (root != null){

            root.setRoot(root.getLeft());
            printTree(root.getRoot());

            System.out.println(root.getData() + " ");

            root.setRoot(root.getRight());
            printTree(root.getRoot());

        }//id

    }//print tree

```

```java
    //This method searches the tree for the 42 items
    public static int searchTree(TreeCordoni root, String target, int comparisons) {

        //if the root is null or equal to the target then return
        if((root == null) || (root.getData().compareToIgnoreCase(target)==0)){
            comparisons++;
        }//if

        //else we continue down the tree recursively to find the target
        else{

            if(target.compareToIgnoreCase(root.getData()) < 0){

                comparisons++;
                System.out.println("L");
                searchTree(root.getLeft(), target, comparisons);

            }//if

            else{
                comparisons++;
                System.out.println("R");
                searchTree(root.getRight(), target, comparisons);

            }//else
        }//else

    //if you uncomment this line it does show different numbers for compare
    //but i am not sure why it is not returning them correctly
    //System.out.println(comparisons);
    return comparisons;

    }//searchTree


}//Assignment4Cordoni
```

1.2.2 Description of Main Code

The main class above consists of different methods to help create graphs and their representations, along with binary search trees. The good parts of the code first include the file sections. While reading the different *txt* files we input each line into arrays for each file. For the Graph representations we create different arrays for each graph so that we can more easily create the graph representations. Along with passing each line of the word array from the *magicitems.txt* file into the binary search tree to be added. Then to keep everything out of the main method, different methods were used to help organize the code better. These methods include the *makeMatrix*, *makeAdjacencyList*, *makeLinkedObjects*, *breadthTraversal*, *reboot*, *depthTraversal*, *insertTree*, *printTree*, and *searchTree* method.

The *makeMatrix* method takes in a 2 dimension instruction array, and goes through the array line by line. First we initialize a height and length for the matrix to be set later once we find all the vertices for the graph. Then we go through the instruction array, if the line starts with a "–" we know to skip that line because it is a comment. If the line contains the word "vertex" we then know that we have to increment the height and length by one because we are adding a new vertex to the graph. If the line then contains the word "edge" we then know to grab the 2nd and 4th indexes of each line so that we can input a "1" at that height/length in the matrix.

The *makeAdjacencyList* method takes in a 2 dimension instruction array, and goes through the array line by line. First we initialize a height to be set later once we find all the vertices for the graph. Then we go through the instruction array, if the line starts with a "−" we know to skip that line because it is a comment. If the line contains the word "vertex" we then know that we have to increment the height by one because we are adding a new vertex to the graph. Then we initialize a 2 dimensional array list, of the size of our variable height that we created before, to store the adjacency list in. If the line then contains the word "edge" we then know to grab the 2nd and 4th indexes of each line so that we can input them into the adjacency list. First we take the 2nd index as the "from" vertex and find that in the height of our adjacency list. Then we take index 4 as our "to" vertex and add it to the array list at the index of the "from" vertex and vise versa.

The *makeLinkedObjects* method takes in a 2 dimension instruction array, and goes through the array line by line. First we initialize a index to be set later once we find all the vertices for the graph and an array to hold all the vertices. Then we go through the instruction array, if the line starts with a "−" we know to skip that line because it is a comment. If the line contains the word "vertex" we then know that we have to increment the index by one because we are adding a new vertex to the graph. Then we initialize an array, of size index. We then go through the array again, and if the line contains the word "vertex", we then grab the vertex id from the instruction array and add it to the array of vertices. Then to add the edge we loop through the vertex array until we find the vertex that matches our "from" vertex, and then we loop through the vertex array again until we find our "to" vertex, and then we add each of them to their opposing neighbor array. The "from" vertex would have the "to" vertex added to its neighbor array, an and the "to" vertex would have the "from" vertex added to its neighbor array.

The *breadthTraversal* method takes in a starting vertex of the graph and adds it to a queue, then it sets its processed status to true. Using a *while* loop, while the queue is not empty, we dequeue the queue and set this new vertex to the current vertex, and then we loop through its neighbor array adding each vertex to the queue, and setting their processed status to true. Upon dequeue-ing the queue we print out the id of the current vertex to show where we are in our traversal.

The *reboot* method takes in a 2 dimension instruction array, and follows the same path as breadth traversal except where breadth traversal sets the processed status to true, here we set it to false so that we can undo the breadth traversal and implement the depth traversal.

The *depthTraversal* method takes in a starting vertex and if it has not been processed we print out its Id and set its processed status to true. We then loop through the neighbor array and recursively call depth traversal to process each vertex in the graph.

The *insertTree* method takes in a tree, and a string containing a line from the word array. We then initialize a new tree node, a trailing tree node, and a current tree node. We set the string to be a new node in the tree, and we set current to equal the root of the tree passed in. While current is not null, we traverse through the tree getting either the next left or right node depending on where the new node falls. If current is null then we move on and check trailing to see if it is null, if it is then we set the new node to be the root of the tree. If trailing is not null then we move on and see if the new node comes before or after the trailing node in the alphabet. If it comes before then we set the node left of trailing to the new node, if it comes before we set the node right of trailing to the new node. Upon completion we print out the binary search tree to see our progress in creating the tree.

The *printTree* method recursively calls itself to print its root data going down the left nodes of the binary search tree, upon completion it prints the root node of the entire tree, and then it goes down the right nodes of the binary search tree printing out the node data along the way.

The *searchTree* method takes in the root of the tree, the target string to look for and the number of comparisons to keep track of. We then look to see if the root is null or if the root of the tree is equal to the target and no searching has to be done. Otherwise we check to see if the target comes before or after the root in the alphabet and if so then we continue down the left side of the tree recursively until we find it, and if not then we continue down the right side of the tree recursively until we find it.

1.3 Tree Class

1.3.1 Description

For the creation of the binary search tree, we needed to create a tree class to represent the root of the tree and it's pointers so that we could determine the path of the binary search tree. This class helped with adding each index of the word array to the binary search tree, and later on when we went back through the tree searching for specific items.

```
/*
 *
 * Assignment 4
 * Due Date and Time: 11/19/21 before 12:00am
 * Purpose: to implement graph and tree data structures, and to understand the performance of
 *their traversals.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output graph and tree data structures.
 * @author Shannon Cordoni
 *
 */

public class TreeCordoni
{
   /**
    * Instance Variables
    */
   private String myData;
   private TreeCordoni myNext;
   private TreeCordoni myRoot;
   private TreeCordoni myRight;
   private TreeCordoni myLeft;
   private TreeCordoni myParent;


   /**
    * The default Constructor for TreeCordoni
    */
   public TreeCordoni()
       {
       myData = new String();
       myRoot = null;
       myLeft = null;
       myRight = null;
       myParent = null;
       myNext= null;
       }//Node Cordoni

    /**
    * The full constructor for TreeCordoni
    * @param newData the incoming data of the item
```

```
     */
    public TreeCordoni(String newData)
    {
         myData = newData;
         myRoot = null;
         myLeft = null;
         myRight = null;
         myParent = null;
         myNext = null;

    }//NodeCordoni


    /**
     * the setter for the item data
     * @param newData the incoming data of the item
     */
    public void setData(String newData)
        {myData = newData;} //set data

    /**
     * The getter for the item data
     * @return the incoming data of the item
     */
    public String getData()
        {return myData;}//get data

    /**
     * The setter for the next
     * @param NewNext the incoming data
     */
    public void setNext(TreeCordoni newNext)
        {myNext = newNext;}//set Next

    /**
     * the getter for the next
     * @return the incoming data
     */
    public TreeCordoni getNext()
        { return myNext;}//get Next


     /**
     * The setter for the root
     * @param NewRoot the incoming data
     */
    public void setRoot(TreeCordoni newroot)
    {myRoot = newroot;}//set Root

     /**
     * the getter for the root
     * @return the incoming data
     */
     public TreeCordoni getRoot()
     { return myRoot;}//get Root

     /**
     * The setter for the left tree
     * @param NewLeft the incoming data
     */
    public void setLeft(TreeCordoni newLeft)
    {myLeft = newLeft;}//set Left

     /**
     * the getter for the Left
```

```
     * @return the incoming data
     */
     public TreeCordoni getLeft()
     { return myLeft;}//get Left


     /**
     * The setter for the Right
     * @param NewRight the incoming data
     */
    public void setRight(TreeCordoni newRight)
    {myLeft = newRight;}//set RIght

     /**
     * the getter for the Right
     * @return the incoming data
     */
     public TreeCordoni getRight()
     { return myRight;}//get Right

      /**
     * The setter for the parent
     * @param NewParent the incoming data
     */
    public void setParent(TreeCordoni newParent)
    {myParent = newParent;}//set Parent

     /**
     * the getter for the parent
     * @return the incoming data
     */
     public TreeCordoni getParent()
     { return myParent;}//get parent

}//Tree Cordoni
```

1.3.2 Description of Tree Code

This code for the Tree Class was created by previous knowledge working with the Node Class. Using the same set up each Tree was given a root, left, right, parent, and next node, along with a string to contain the word. Getters and setters were created for each to make the insert and search methods called in the main class run more smoothly.

1.4 Node Class

1.4.1 Description

For each element in the word array a node was created to represent the word. This was so that the creation of the binary search tree would run smoothly and so that each node would be linked to the next one.

```
/*
 *
 * Assignment 4
 * Due Date and Time: 11/19/21 before 12:00am
 * Purpose: to implement graph and tree data structures, and to understand the performance of
 *their traversals.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output graph and tree data structures.
 * @author Shannon Cordoni
 *
 */
```

```

public class NodeCordoni
{
    /**
     * Instance Variable for word data and node
     */
    private String myData;
    private NodeCordoni myNext;

    /**
     * The default Constructor for NodeCordoni
     */
    public NodeCordoni()
        {
        myData = new String();
        myNext= null;
        }//Node Cordoni

    /**
     * The full constructor for NodeCordoni
     * @param newData the incoming data of the item
     */
    public NodeCordoni(String newData)
        {
        myData = newData;
        myNext = null;
        }//NodeCordoni

    /**
     * the setter for the item data
     * @param newData the incoming data of the item
     */
    public void setData(String newData)
        {myData = newData;} //set data

    /**
     * The getter for the item data
     * @return the incoming data of the item
     */
    public String getData()
        {return myData;}//get data

    /**
     * The setter for the node
     * @param NewNext the incoming node data
     */
    public void setNext(NodeCordoni newNext)
        {myNext = newNext;}//set Node

    /**
     * the getter for the node
     * @return the incoming node data
     */
    public NodeCordoni getNext()
        { return myNext;}//get node

}//NodeCordoni
```

1.4.2 Description of Node Code

This code for the Node Class was created by in class lessons but also previous knowledge from Software Development 1. Using the same set up each node was created so that it consisted of a string and a myNext linking each node to the next. Getters and setters were created for both the nodes themselves and the data inside of them so that we would be able to call $node.getNext()$, $node.setNext()$, $node.getData()$, and $node.setData()$ in the main class and to make working the binary search tree run more smoothly.

1.5 Vertex Class

1.5.1 Description

For each vertex in the graph an actual vertex object had to be created to hold it. This class creates the vertex object to be represented in the graph. This was so that the creation of the graph representations could run more smoothly and so that the edges could be added to help represent the graphs.

```java
/*
 *
 * Assignment 4
 * Due Date and Time: 11/19/21 before 12:00am
 * Purpose: to implement graph and tree data structures, and to understand the performance of
 *their traversals.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output graph and tree data structures.
 * @author Shannon Cordoni
 *
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Arrays;
import java.util.ArrayList;

public class VertexCordoni
{
   /**
    * Declare Variables
    */
    private String myId;
    private boolean myIsProcessed;
    public  ArrayList <VertexCordoni> neighbors= new ArrayList <VertexCordoni>();
    private VertexCordoni myNext;


    /**
    * The default Constructor for VertexCordoni
    */
    public VertexCordoni()
    {
    myId = new String();
    myIsProcessed = false;
    myNext = null;
    }//vertex Cordoni

     /**
    * The full constructor for VertexCordoni
    * @param newData the incoming data
    */
   public VertexCordoni(String newData)
   {
        myId = newData;
        myIsProcessed = false;
        myNext = null;

   }//NodeCordoni

    /**
```

```
      *
     * the setter for the vertex id
     * @param newId the incoming data of the vertex
     */
     public void setId(String newId)
     {myId = newId;} //set data

     /**
     * The getter for the vertex id
     * @return the incoming data of the vertex
     */
     public String getId()
     {return myId;}//get data


     /**
     *
     * the setter for the next vertex i
     * @param newNext the incoming data of the vertex
     */
     public void setNext(VertexCordoni newNext)
     {myNext = newNext;} //set data

     /**
     * The getter for the vertex
     * @return the incoming data of the vertex
     */
     public VertexCordoni getNext()
     {return myNext;}//get data

     /**
     * The setter for the process status
     * @param newIsProcessed the incoming process status
     */
     public void setProcessStatus(boolean newIsProcessed)
     {myIsProcessed = newIsProcessed;}//set Node

     /**
     * the getter for the process status
     * @return the incoming process status
     */
     public boolean getProcessStatus()
     { return myIsProcessed;}//get node

     //This checks to see if the neighbor array is empty
         public boolean isEmpty()
         {
                 boolean empty = false;

                 if(neighbors == null)
                         {
                         empty = true;
                         }//if
                 return empty;
         }//empty


}//Vertex Cordoni
```

1.5.2 Description of Vertex Code

This code for the Vertex Class was created by previous knowledge working with the Node Class. Using the same set up each Vertex was given an Id, process status, neighbor arraylist, and next. Getters and setters were created for each to make the creation of the vertexes and edges run more smoohtly, but also to help in the breadth and depth traversal of the linked object representation of the graphs.

1.6 Queue Vertex Class

1.6.1 Description

For each vertex in the graph an actual vertex object had to be created to hold it. Then to preform the breadth first traversal through these vertexes, a queue had to be made to keep track of where we were in our traversal. Using a queue, this helped the breadth first traversal run successfully.

```

/*
 *
 * Assignment 4
 * Due Date and Time: 11/19/21 before 12:00am
 * Purpose: to implement graph and tree data structures, and to understand the performance of
 *their traversals.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output graph and tree data structures.
 * @author Shannon Cordoni
 *
 */

public class QueueVertexCordoni {

	private VertexCordoni myHead;
	private VertexCordoni myTail;

	//This method adds a vertex to the queue, it does so by adding it to the end of
	//the queue
	public void enqueue(VertexCordoni newVertex)
	{
		//this sets a temp variable to hold the current tail node
		VertexCordoni oldTail = myTail;

		//this sets the tail to be a new node and its data to be the new vertex
		myTail = newVertex;

		//This checks to see if the queue is empty
		//if it is not empty then the old tail is set to now point to the new Node
		if (!isEmpty()){
			oldTail.setNext(myTail);
		}//if

		//if the queue is empty then all variables are the same because there
		//is nothing in the queue. Then the head and tail pointer would be pointing
		//to the same thing.
		else{
			myHead = myTail;
		}//else

	}//enqueue

	//This method removes a vertex from the queue
	public VertexCordoni dequeue()
	{
		//This sets the temp variable to null so that it can be set later.
		VertexCordoni answer = null;

		//If the queue is not empty then it will remove the first vertex from
		//the queue
		if(!isEmpty())
```

```
                {
                        //This sets the temp variable to the first vertex in the list and
                        //then sets the new head pointer to the second vertex in the queue
                        answer = myHead;
                        myHead = myHead.getNext();

                        //if the queue is empty then the head is null
                        if(isEmpty()){
                                myHead = null;
                        }//if
                }//if

                else{
                        System.out.println("The Queue is empty");
                }
                return answer;
        }//dequeue

        //THis checks to see if the queue is empty
        public boolean isEmpty()
        {
                boolean empty = false;

                if(myHead == null)
                        {
                        empty = true;
                        }//if
                return empty;
        }//empty

}//QueueCordoni
```

1.6.2 Description of Queue Vertex Code

This code for the Queue Vertex Class was created by previous knowledge working with the Node Class, the good parts of the Queue class involve the different methods created, such as *enqueue*, *dequeue*, and *isEmpty*.

The *enqueue* method takes in a new vertex and adds it into the queue. It does this by first creating a temp variable so that we do not lose the current tail pointer of the queue. We then set the tail pointer to be a new vertex. It then checks to see if the queue is empty, if it is not empty then it takes the new vertex and adds it to the queue by setting the temp variable or the old tail to now point to the new vertex. If the queue is empty then that means that the head, and tail would be pointing to or signifying the same vertex.

The *dequeue* method creates a temp variable *answer* which is the vertex we hope to remove from the queue. It then checks to see if the queue is empty, if it is empty then we cannot remove anything from an empty queue. If it is not empty then we can set the temp variable to the head or front of the queue and then set the new head to be the next vertex in line and return the temp variable.

The *isEmpty* method checks whether or not to see if the queue is empty, it does this by looking to see if the head of the list is null, due to the fact that if there is something in the queue then there is always a head to the queue being that queues are first in first out.

1.7 Overall:

Overall, these Graph and Tree representations were successful in implementation. To go through each traversal and the BST lookup we can create a table for better data understanding, this table will show each and their asymptotic running time:

Breadth Traversal	Depth Traversal	BST Lookups
O($\|V\| + \|E\|$)	O($\|V\| + \|E\|$)	O($log(n)$)

The table above shows a quick understanding of the methods used here. To go into more detail Breadth First Traversal and Depth First Traversal each have an asymptotic running time of O($|V| + |E|$), this is because each edge and vertex will be processed once while we traverse through the graph. Once we process a vertex in the graph, we do not pass through it again on our traversal. Binary Search Tree Lookup has an asymptotic running time of O($log(n)$) with log being of base 2. This means that as n increases the number of operations stays the same until n doubles. From the opposite direction, when we start looking through a binary search tree we start at the root, if our target falls below the root then we have eliminated the top half of the tree to look through, and can continue on in our search. Overall, we were mostly successful in our implementation of this assignment.