Assignment 2 - Code Evaluation

Mason Nakamura

October 2021

1 Node Class

```
public class Node {
    String name;
    Node left = null;
    Node right = null;
}
```

The Node class is used only in the Binary_Search_Tree class.

2 Vertex Class

```
import java.util.ArrayList;

public class Vertex {
    Vertex next;
    int label;
    int connecting_vertex;
    int origin_vertex;
    ArrayList<Vertex> neighbors;
}
```

The Vertex class is used in the Outputs class, Queue class, Search class, and Test class. The next denotes a pointer to the next vertex object. The label denotes the number appended to the vertex (I probably could've combined the Vertex and Node class). The connecting_vertex and origin_vertex both resemble the attributes of an edge (Which I could've separated into a separate class). Finally, the neighbors list is used to determine the neighboring vertices of the vertex object.

3 Queue Class

```
public class Queue {
    Vertex head, tail;

    public boolean isEmpty(){
        return(head == null);
    }

    public void enqueue(Vertex s){
        Vertex oldTail = tail;
        tail = new Vertex();
        tail = s;
        tail.next = null;
        if(isEmpty()){
            head = tail;
        }else{
            oldTail.next = tail;
        }
    }

    public Vertex dequeue(){
        Vertex retval;
        if(!isEmpty()){
            retval = head;
            head = head.next;
            if(isEmpty()){
                tail = null;
            }
        }else{
            retval = null;
        }
        return retval;
    }

}
```

The Queue class was taken from a previous assignment and modified to accommodate for the Vertex type instead of Strings.

4 Outputs Class

```java
import java.util.ArrayList;
public class Outputs {
    /**
     * This method is used to print all adjacency matrices given the lines in the file
     *
     * @param lines an array of every line in the file
     */
    public void adjacency_matrix(String[] lines) {
        // use these indices throughout for loops
        int index_start;
        int index_end = 0;
        // for printing purposes
        int graph_id = 0;
        // for vertex starting index
        boolean indexIs0 = false;
        while (index_end <= lines.length) {
            graph_id += 1;
            index_start = index_end;
            // don't need rows and columns since adjacency matrix is symmetric ==> rows = columns
            int rows_columns = 0;
            // get indices for for loop for edges
            for (int i = index_start; i < lines.length; i++) {
                String line = lines[i];
                index_start += 1;
                // case for line is blank;
                if (line.isBlank()) {
                    // skip the line
                    continue;
                }
                // case for line is a comment
                if (line.charAt(1) == '-' & line.charAt(0) == '-') {
                    // skip the line
                    continue;
                }
                // make an array of the words from the line
                String[] words = line.split(" ");
                // case for declaring new graph
                if (words[0].equals("new")) {
                    // print graph_id
                    System.out.println("Graph Number: " + graph_id);
                    //skip the line
                    continue;
                }
                // case for adding vertex
                if (words[0].equals("add") & words[1].equals("vertex")) {
                    rows_columns += 1;
                    if (words[2].equals("0")) {
                        indexIs0 = true;
                    }
                    continue;
                }
                //check if hit edge case
                if (words[0].equals("add") & words[1].equals("edge")) {
                    index_start -= 1;
                    break;
                }

            }

            // create instance of matrix given number of vertices declared
            index_end = index_start + 1;
            int[][] matrix = new int[rows_columns][rows_columns];
            // use index_start to continue in the for loop
            for (int i = index_start; i < index_end; i++) {
                // case if reached to end of file/(array of lines)
                if (lines.length < index_end) {
                    break;
                }
                String line = lines[i];
                String[] words = line.split(" ");
                // check if line is blank; if so, move to next graph
```

```
                if (line.isBlank()) {
                    break;
                }
                // case for adding edge
                if (words[0].equals("add") & words[1].equals("edge")) {
                    index_end += 1;
                    // use Integer.parseInt to convert string to int

                    if (indexIs0) {// if index starts at 0 don't subtract 1
                        matrix[Integer.parseInt(words[2])][Integer.parseInt(words[4])] = 1;
                        // do it twice since undirected
                        matrix[Integer.parseInt(words[4])][Integer.parseInt(words[2])] = 1;
                    } else {// subtract 1 to keep indices same
                        matrix[Integer.parseInt(words[2]) - 1][Integer.parseInt(words[4]) - 1] = 1;
                        // do it twice since undirected
                        matrix[Integer.parseInt(words[4]) - 1][Integer.parseInt(words[2]) - 1] = 1;
                    }
                }
            }

            for (int i = 0; i < matrix.length; i++) {
                for (int j = 0; j < matrix[i].length; j++) {
                    System.out.print(matrix[i][j] + " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }

    public void adjacency_list(String[] lines) {
        // use these indices throughout for loops
        int index_start;
        int index_end = 0;
        // for printing purposes
        int graph_id = 0;
        // for vertex starting index
        boolean indexIs0 = false;
        while (index_end <= lines.length) {
            graph_id += 1;
            index_start = index_end;
            int adj_list_length = 0;
            // get indices for for loop for edges
            for (int i = index_start; i < lines.length; i++) {
                String line = lines[i];
                index_start += 1;
                // case for line is blank;
                if (line.isBlank()) {
                    // skip the line
                    continue;
                }
                // case for line is a comment
                if (line.charAt(1) == '-' & line.charAt(0) == '-') {
                    // skip the line
                    continue;
                }
                // make an array of the words from the line
                String[] words = line.split(" ");
                // case for declaring new graph
                if (words[0].equals("new")) {
                    // print graph_id
                    System.out.println("Graph Number: " + graph_id);
                    //skip the line
                    continue;
                }
                // case for adding vertex
                if (words[0].equals("add") & words[1].equals("vertex")) {
                    adj_list_length += 1;
                    if (words[2].equals("0")) {
                        indexIs0 = true;
                    }
                    continue;
```

```java
            }
            //check if hit edge case
            if (words[0].equals("add") & words[1].equals("edge")) {
                // subtract an index since we previously added it but didn't need to since we entered the edge case
                index_start -= 1;
                break;
            }

        }

        // create instance of array given number of vertices declared
        index_end = index_start + 1;
        //make array of vertex objects
        Vertex[] array = new Vertex[adj_list_length];
        // check starting index

        // create first vertex objects in array
        if (indexIs0) {
            for (int i = 0; i < adj_list_length; i++) {
                // from Vertex class
                Vertex vertex = new Vertex();
                vertex.label = i;
                vertex.next = null;
                array[i] = vertex;
            }
        } else {// shift index by 1
            for (int i = 1; i < adj_list_length + 1; i++) {
                Vertex vertex = new Vertex();
                vertex.label = i;
                vertex.next = null;
                array[i - 1] = vertex;
            }
        }

        // use index_start to continue in the for loop
        for (int i = index_start; i < index_end; i++) {
            // case if reached to end of file/(array of lines)
            if (lines.length < index_end) {
                break;
            }
            String line = lines[i];
            String[] words = line.split(" ");
            // check if line is blank; if so, move to next graph
            if (line.isBlank()) {
                break;
            }

            // case for adding edge
            if (words[0].equals("add") & words[1].equals("edge")) {
                index_end += 1;
                // you need to define vertex1 and vertex2 since not doing so creates a pointer infinitely pointing to itself
                if (indexIs0) {// if index starts at 0 don't subtract 1
                    Vertex vertex1 = new Vertex();
                    // use Integer.parseInt to convert string to int
                    vertex1.origin_vertex = Integer.parseInt(words[2]);
                    vertex1.connecting_vertex = Integer.parseInt(words[4]);
                    vertex1.next = null;
                    Vertex head = array[vertex1.origin_vertex];
                    while (head.next != null) {
                        head = head.next;
                    }
                    head.next = vertex1;

                    Vertex vertex2 = new Vertex();
                    vertex2.origin_vertex = Integer.parseInt(words[2]);
                    vertex2.connecting_vertex = Integer.parseInt(words[4]);
                    vertex2.next = null;
                    // do it twice since undirected
                    Vertex head1 = array[vertex2.connecting_vertex];
                    while (head1.next != null) {
                        head1 = head1.next;
                    }
                    head1.next = vertex2;
                } else {// subtract 1 to keep indices same
```

```java
                        Vertex vertex1 = new Vertex();
                        vertex1.origin_vertex = Integer.parseInt(words[2]);
                        vertex1.connecting_vertex = Integer.parseInt(words[4]);
                        vertex1.next = null;
                        Vertex head = array[vertex1.origin_vertex - 1];
                        while (head.next != null) {
                            head = head.next;
                        }
                        head.next = vertex1;

                        // do it twice since undirected
                        Vertex vertex2 = new Vertex();
                        vertex2.origin_vertex = Integer.parseInt(words[2]);
                        vertex2.connecting_vertex = Integer.parseInt(words[4]);
                        vertex2.next = null;
                        Vertex head1 = array[vertex2.connecting_vertex - 1];
                        while (head1.next != null) {
                            head1 = head1.next;
                        }
                        head1.next = vertex2;
                    }

                }

            }

            for (Vertex vertex : array) {
                System.out.print("[" + vertex.label + "]" + " ");
                if (vertex.next != null) {
                    Vertex temp_vertex = vertex.next;
                    //check if we print connecting_vertex or origin_vertex
                    while (temp_vertex != null) {
                        if (temp_vertex.connecting_vertex != vertex.label) {
                            System.out.print(temp_vertex.connecting_vertex + " ");
                        }
                        if (temp_vertex.origin_vertex != vertex.label) {
                            System.out.print(temp_vertex.origin_vertex + " ");
                        }
                        temp_vertex = temp_vertex.next;
                    }
                }
                System.out.println();
            }
        }
    }

    public void linked_objects(String[] lines) {
        // use these indices throughout for loops
        int index_start;
        int index_end = 0;
        // for printing purposes
        int graph_id = 0;
        // for vertex starting index
        boolean indexIs0 = false;
        // Read the lines up to add edge case
        while (index_end <= lines.length) {
            graph_id += 1;
            index_start = index_end;
            int adj_list_length = 0;
            // get indices for for loop for edges
            for (int i = index_start; i < lines.length; i++) {
                String line = lines[i];
                index_start += 1;
                // case for line is blank;
                if (line.isBlank()) {
                    // skip the line
                    continue;
                }
                // case for line is a comment
                if (line.charAt(1) == '-' & line.charAt(0) == '-') {
                    // skip the line
                    continue;
                }
                // make an array of the words from the line
                String[] words = line.split(" ");
                // case for declaring new graph
```

```java
            if (words[0].equals("new")) {
                // print graph_id
                System.out.println("Graph Number: " + graph_id);
                //skip the line
                continue;
            }
            // case for adding vertex
            if (words[0].equals("add") & words[1].equals("vertex")) {
                adj_list_length += 1;
                if (words[2].equals("0")) {
                    indexIs0 = true;
                }
                continue;
            }
            //check if hit edge case
            if (words[0].equals("add") & words[1].equals("edge")) {
                // subtract an index since we previously added it but didn't need to since we entered the edge case
                index_start -= 1;
                break;
            }

        }

        // create instance of array given number of vertices declared
        index_end = index_start + 1;
        //make array of vertex objects
        Vertex[] array = new Vertex[adj_list_length];
        // check starting index

        // create first vertex objects in array given index
        if (indexIs0) {
            for (int i = 0; i < adj_list_length; i++) {
                // from Vertex class
                Vertex vertex = new Vertex();
                vertex.connecting_vertex = i;
                vertex.label = i;
                vertex.neighbors = new ArrayList<Vertex>();
                array[i] = vertex;
            }

        } else {// shift index by 1
            for (int i = 1; i < adj_list_length + 1; i++) {
                Vertex vertex = new Vertex();
                vertex.connecting_vertex = i;
                vertex.label = i;
                vertex.neighbors = new ArrayList<Vertex>();
                array[i - 1] = vertex;
            }
        }

        // use index_start to continue in the for loop
        for (int i = index_start; i < index_end; i++) {
            // case if reached to end of file/(array of lines)
            if (lines.length < index_end) {
                break;
            }
            String line = lines[i];
            String[] words = line.split(" ");
            // check if line is blank; if so, move to next graph
            if (line.isBlank()) {
                break;
            }

            // case for adding edge
            if (words[0].equals("add") & words[1].equals("edge")) {
                index_end += 1;
                // you need to define vertex1 and vertex2 since not doing so creates a pointer infinitely pointing to itself
                if (indexIs0) {// if index starts at 0 don't subtract 1
                    Vertex vertex1 = new Vertex();
                    vertex1.origin_vertex = Integer.parseInt(words[2]);
                    vertex1.connecting_vertex = Integer.parseInt(words[4]);
                    vertex1.label = vertex1.connecting_vertex;
                    vertex1.neighbors = array[vertex1.connecting_vertex].neighbors;
```

```
367                         Vertex head = array[vertex1.origin_vertex];
368                         // add vertex to neighbors attribute
369                         head.neighbors.add(vertex1);
370
371                         // do it twice since undirected
372                         Vertex vertex2 = new Vertex();
373                         vertex2.origin_vertex = Integer.parseInt(words[2]);
374                         vertex2.connecting_vertex = Integer.parseInt(words[4]);
375                         vertex2.label = vertex1.origin_vertex;
376                         vertex2.neighbors = array[vertex2.origin_vertex].neighbors;
377                         Vertex head1 = array[vertex2.connecting_vertex];
378                         head1.neighbors.add(vertex2);
379
380                     } else {// subtract 1 to keep indices same
381                         Vertex vertex1 = new Vertex();
382                         vertex1.origin_vertex = Integer.parseInt(words[2]);
383                         vertex1.connecting_vertex = Integer.parseInt(words[4]);
384                         vertex1.label = vertex1.connecting_vertex;
385                         vertex1.neighbors = array[vertex1.connecting_vertex - 1].
                                neighbors;
386                         Vertex head = array[vertex1.origin_vertex - 1];
387                         // add vertex to neighbors attribute
388                         head.neighbors.add(vertex1);
389
390                         // do it twice since undirected
391                         Vertex vertex2 = new Vertex();
392                         vertex2.origin_vertex = Integer.parseInt(words[2]);
393                         vertex2.connecting_vertex = Integer.parseInt(words[4]);
394                         vertex2.label = vertex1.origin_vertex;
395                         vertex2.neighbors = array[vertex2.origin_vertex - 1].
                                neighbors;
396                         Vertex head1 = array[vertex2.connecting_vertex - 1];
397                         head1.neighbors.add(vertex2);
398                     }
399
400                 }
401
402             }
403             Vertex[] copy_vertexes = array.clone();
404             Vertex[] copy_vertexes1 = array.clone();
405             Search search = new Search();
406             Search search1 = new Search();
407             System.out.print("Depth First: ");
408             search1.depth_first(copy_vertexes1[0], array);
409             System.out.println();
410             System.out.print("Breadth First: ");
411             search.breadth_first(copy_vertexes[0], array);
412             System.out.println();
413
414
415
416         }
417     }
418 }
```

This class outputs the data for the matrices, linked objects, and adjacency lists. In line 8, this method (adjacency_matrix) prints our matrices. Lines 10-58 are used to read line-by-line the file of graphs until the edges begin to be added; in which case we move to lines 61-90 which reads in the "adding edges" lines. These lines also create our empty matrix of correct height and length and populates it. From lines 92-99, we print out the matrix.

In line 10, we initialize our starting index and ending index to keep track as where we are in the file. We also initiate our graph id to know which graph we are currently on. In addition, we initiate indexIs0 which keeps track if the starting index is 0 or not. In lines 22 - 56 we have our text processor which takes into account comments initialization of new graphs, and adding vertices.

For the adjacency_list method, we do a very similar process in the adjacency_matrix method, except we initiate an empty array of vertices in lines 162-175 given indexIs0 or not. We then move on to line 192 where we are inserting the edges. In line 196, we initialize our new vertex with relevant attributes. Then we assign a pointer from the original_vertex to the connecting vertex and vice versa to account for an undirected graph. We then print the lists from lines 244 - 257.

Our next method is linked_objects where we also have a similar structure to our other two methods, except we altered the initialization of the array of vertices by adding a neighbors attribute to each vertex in the array. Next, we populated these neighbors arrays from lines 358-398. Later in lines 403 - 412, we output a depth first search and breadth first search on this array of vertices with neighbors, taking into account the starting index and the disconnected graphs in the last graph.

5 Search Class

```
    import java.util.ArrayList;

public class Search {
    ArrayList<Integer> processed = new ArrayList<>();

    public void depth_first(Vertex v, Vertex[] array) {
        if (!processed.contains(v.label)) {
            System.out.print(v.label + " ");
            processed.add(v.label);
        }
        for (Vertex neighbor : v.neighbors) {
            if (!processed.contains(neighbor.label)) {
                depth_first(neighbor, array);
            }
        }
        for (Vertex vertex: array){
            if (!processed.contains(vertex.label)){
                if (!processed.contains(vertex.label)) {
                    System.out.print(vertex.label + " ");
                    processed.add(vertex.label);
                }
                for (Vertex neighbor : vertex.neighbors) {
                    if (!processed.contains(neighbor.label)) {
                        depth_first(neighbor, array);
                    }
                }
            }
        }
    }


        public void breadth_first (Vertex v, Vertex[] array){
            Queue queue = new Queue();
            queue.enqueue(v);
            processed.add(v.label);
            while (!queue.isEmpty()) {
                Vertex vertex = queue.dequeue();
                System.out.print(vertex.label + " ");
                for (Vertex neighbor: vertex.neighbors) {
                    if (!processed.contains(neighbor.label)) {
                        queue.enqueue(neighbor);
                        processed.add(neighbor.label);
                    }
                }
            }
            for (Vertex vertex: array){
                if (!processed.contains(vertex.label)){
                    Queue queue1 = new Queue();
                    queue1.enqueue(vertex);
                    processed.add(vertex.label);
                    while (!queue1.isEmpty()) {
                        Vertex vertex1 = queue1.dequeue();
                        System.out.print(vertex1.label + " ");
                        for (Vertex neighbor: vertex1.neighbors) {
                            if (!processed.contains(neighbor.label)) {
                                queue1.enqueue(neighbor);
                                processed.add(neighbor.label);
                            }
                        }
                    }
                }
                }

        }
    }
```

The Search class takes care of our breadth first and depth first searches in the Output class. In our depth_first method, we do the standard depth first algorithm, then add an extra layer with the for loop to take into account the disconnected graphs cases. We do so similarly in our breadth_first search method. Breadth first traversal and Depth fist traversal both have a time complexity of $O(|V|+|E|)$. For DFS, you are first looking at the vertex in the first if statement then the edges in the next for statement, giving us our expected time complexity. For BFS, the while loop takes care of our time complexity for the vertices while the nested for loop takes care of the edges.

6 Binary_Search_Tree Class

```
public class Binary_Search_Tree {
    int i = 0;
    int comparisons;
    Node first_root;
    public Node populateBST(Node root, String word){
        // if there is no root to begin with or it hits a terminal node
        if (root == null){
            Node node = new Node();
            node.name = word;
            return node;
        }
        // go down the left side if word <= root.name
        else if (word.compareTo(root.name) >= 0){
            root.left = populateBST(root.left, word);
            System.out.print(" " + "L");
        }
        else{ // go down right side if word > root.name
            root.right = populateBST(root.right, word);
            System.out.print(" " + "R");
        }
        return root;
    }

    public Node makeBST(String[] words)
    {
        Node root = null;
        boolean gotroot = false;
        // go through all words and populate BST using method populateBST
        for (String word: words) {
            if (!gotroot){
                gotroot = true;
                Node temp_first_root = new Node();
                temp_first_root.name = word;
                first_root = temp_first_root;
            }
            System.out.println(word + ": ");
            root = populateBST(root, word);
            System.out.println();
        }
        // if there are no words
        return root;
    }

    public void inorder(Node root)
    {
        if (root == null) {
            return;
        }

        // go to the left child
        inorder(root.left);
        // print the current root name
        System.out.println(root.name);
        // go to the right child
        inorder(root.right);
    }

    public Node search(Node root, String target){
        if (root.name.equals(target) | root == null){
            comparisons += 1;
            return root;
        }

        if (target.compareTo(root.name) >= 0){
            comparisons += 1;
            System.out.print(" " + "L");
            return search(root.left, target);

        }
```

```
            else{
                comparisons += 1;
                System.out.print(" " + "R");
                return search(root.right, target);

            }
        }
}
```

The populateBST method populates the binary search tree recursively. The makeBST method uses the populateBST method to create a BST given an array of comparable objects. The inorder method does an in order traversal on the binary search tree and prints the results. The search method searches for a specific word in our already made binary search tree. The search method has an average time complexity of $O(log_2(n))$ given that the tree is balanced since when the tree traverses right or left it essentially removes half of the possible target values recursively down the tree.

7 Class Test

```
import java.io.*;
import java.util.Arrays;
import java.util.Random;

public class Test {



    public static void main(String[] args) {
        String[] lines = {};
        // Read line by line the txt file using File reader
        String fileName = "Assignment 4/graphs1.txt";  //REMEMBER TO NOT HARDCODE
        File file = new File(fileName);
        try {
            FileReader fr = new FileReader(file);
            BufferedReader br = new BufferedReader(fr);
            String line;
            while ((line = br.readLine()) != null) {
                // add strings from txt file line by line into array words
                lines = Arrays.copyOf(lines, lines.length + 1); //extends memory
                lines[lines.length - 1] = line; //adds line to extra memory
            }
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        String[] copy_graphs0 = lines.clone();

        Outputs out = new Outputs();
        // outputs matrices
        System.out.println("ADJACENCY MATRICES: ");
        out.adjacency_matrix(copy_graphs0);
        // outputs adjacency lists
        System.out.println("ADJACENCY LISTS: ");
        out.adjacency_list(copy_graphs0);
        System.out.println("LINKED OBJECTS: ");
        out.linked_objects(copy_graphs0);


        // Binary Search Tree population and traversal
        String[] lines1 = {};
        // Read line by line the txt file using File reader
        String fileName1 = "Assignment 4/magicitems.txt";  //REMEMBER TO NOT HARDCODE
        File file1 = new File(fileName1);
        try {
            FileReader fr = new FileReader(file1);
            BufferedReader br = new BufferedReader(fr);
            String line;
            while ((line = br.readLine()) != null) {
                // add strings from txt file line by line into array words
```

```java
                lines1 = Arrays.copyOf(lines1, lines1.length + 1); //extends memory
                lines1[lines1.length - 1] = line; //adds line to extra memory
            }
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        String[] lines2 = {};
        // Read line by line the txt file using File reader
        String fileName2 = "Assignment 4/magicitems-find-in-bst.txt";  //REMEMBER TO NOT HARDCODE
        File file2 = new File(fileName2);
        try {
            FileReader fr = new FileReader(file2);
            BufferedReader br = new BufferedReader(fr);
            String line;
            while ((line = br.readLine()) != null) {
                // add strings from txt file line by line into array words
                lines2 = Arrays.copyOf(lines2, lines2.length + 1); //extends memory
                lines2[lines2.length - 1] = line; //adds line to extra memory
            }
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        final int FILE_LENGTH = lines2.length;

        // copy_words2 -> magicitems-find-in-bst.txt
        // copy_words1 -> magicitems.txt
        String[] copy_words2 = lines2.clone();
        String[] copy_words1 = lines1.clone();

        Binary_Search_Tree bst = new Binary_Search_Tree();
        System.out.println("BINARY SEARCH TREE POPULATING: ");
        Node root = bst.makeBST(copy_words1);
        System.out.println("BINARY SEARCH TREE IN-ORDER TRAVERSAL: ");
        bst.inorder(root);
        System.out.println("BINARY SEARCH TREE SEARCHING FOR ITEMS W/ COMPARISONS: ");
        float total_comparisons = 0;
        for (String i: copy_words2){
            System.out.print(i + " ");
            bst.search(root, i);
            System.out.println();
            System.out.println("Comparisons: " + bst.comparisons);
            total_comparisons += bst.comparisons;
            bst.comparisons = 0;
        }
        System.out.println("AVERAGE NUMBER OF COMPARISONS: " + total_comparisons/FILE_LENGTH);
    }
}
```

We have 3 file readers in the main method. Although this probably could've been condensed (I will also change the file names before submitting)(Nothing will run if the paths aren't given). We then apply our methods from our other classes for our results. At the end we got an average of about 10.3 comparisons on our binary search tree.