Assignment Four

Tyler Hernandez
Tyler.Hernandez1@marist.edu

November 18, 2022

This page left blank intentionally.

1 Main Method

```java
import java.util.ArrayList;
public class Main {

    // Driver for Assignment 4.
    public static void main(String[] args) throws Exception {

        Reader reader = new Reader("./graphs1.txt");

        String line = reader.getNextLine();
        String[] words;
        ArrayList<Graph> graphs = new ArrayList<>();

        Graph g = new Graph(); // Initialize so our compiler will stop complaining.

        // Flag to make sure while loop runs one time after reader flags end of file
        boolean readingLastLine = false;

        while (!readingLastLine) { // Until we've reached the end of the file.

            if (reader.endOfFile) {
                readingLastLine = true; // We've reached end of file, we can stop looping.
            }
            // System.out.println(line);

            // Take our next line of text and put each word into an array.
            line = line.replaceAll("\n", "");
            words = line.split(" ");

            // Now, we can determine which action to take strictly based on sentence length!

            if (words.length == 0){
                // Blank line.
            }
            else if (words[0].equals("--")) {
                // Do nothing because this is a comment.
            }
            else if (words.length == 2) {
                // Save previous graph to our list, if it was not the default graph.
                if (!g.isEmpty()){
                    graphs.add(g);
                }

                // NEW GRAPH command
                g = new Graph();
            } else if (words.length == 3) {
                // ADD VERTEX (INT) command
                g.addVertex(Integer.parseInt(words[2]));
            } else if (words.length == 5) {
                // ADD EDGE (INT) - (INT) command
                g.addEdge(Integer.parseInt(words[2]), Integer.parseInt(words[4]));
            }

            // Grab the next line so we can keep going!
            line = reader.getNextLine();
        } // end of while.

        // Since the final graph never reaches our 'save' in while loop.
        graphs.add(g);


        // For each graph,
        for (Graph graph : graphs) {
            System.out.println("---------------------------------------------");
```

```java
            System.out.println("GRAPH: ");

            //  display it's matrix!
            System.out.println("Matrix\n");
            printMatrix(graph.createMatrix());
            System.out.println("---------------------------------------------");

            // display it's adjacency list!
            System.out.println("Adjacency List\n");
            graph.printList();
            System.out.println("---------------------------------------------");

            // Depth First Search.
            System.out.println("Depth First Search\n");
            Graph.DFS(graph.vertices.get(0), graph); // Grab the first vertex held in our graph.
            System.out.println("---------------------------------------------");

            // Reset graph, since all of it's vertices have been processed.
            System.out.println();
            for (Vertex vertex : graph.vertices){
                System.out.print(vertex.id);
                vertex.isProcessed = false;
            }
            System.out.println();

            // Breadth First Search.
            System.out.println("Breadth First Search\n");
            Graph.BFS(graph.vertices.get(0), graph);
            System.out.println("---------------------------------------------");
            System.out.println("\n\n\n");


        }
        System.out.println("\n\n\n");

        // Create a binary search tree!
        BinarySearchTree tree = new BinarySearchTree();

        // Read in our magicitems values.
        reader = new Reader("./magicitems-find-in-bst.txt");
        String magicItem = reader.getNextLine();

        // Reset flag.
        readingLastLine = false;

        // Holds the magic items we will lookup in our tree.
        ArrayList<String> magicItemsList = new ArrayList<>();

        // Populate our BST with magicItems!
        while (!readingLastLine) {

            if (reader.endOfFile) {
                readingLastLine = true; // We've reached end of file, we can stop looping.
            }
            magicItem = magicItem.replace("\n", "");

            System.out.println("Insertion of " + magicItem + " ");
            tree.insert(new TreeNode(magicItem));
            System.out.println();
            System.out.println();
            magicItemsList.add(magicItem);


            magicItem = reader.getNextLine();

```

```java
        }

        // in-order traversal for tree.
        tree.traverseInOrder(tree.root); // Need to give recursive function a starting root.

        int sum = 0;
        // Get the count of comparisons made looking for
        for (String item : magicItemsList){
            System.out.println();
            int comparisons = tree.lookup(item);
            sum += comparisons;
            System.out.print(item + "-->comparison count: " + comparisons);
            System.out.println();
        }
        System.out.println();
        System.out.println();
        System.out.println();
        double average = (sum / magicItemsList.size());

        System.out.println("Average for " + magicItemsList.size() + " items = " + average);


    }




    public static void printMatrix(int[][] matrix) {
        // This helps display the coordinate location.
        System.out.println();
        int length = matrix.length;

        System.out.print("x-");// Display top horizontal indices.
        for (int i=0; i<length; i++) {
            System.out.print(i + "-");
        }
        System.out.println();
        System.out.println("|");

        for (int i=0; i<length; i++) {
            System.out.print(i + "-"); // Display left-side vertical indices.
            for(int x=0; x<matrix[i].length; x++) {
                System.out.print(matrix[i][x] + "-");
            }
            System.out.println();
            System.out.println("|");
        }
        System.out.println();

    }

}
```

1.1 Main Method

There were many milestones with this assignment. Firstly, I demonstrated my ability to parse a text file which had command requesting creation of a graph, and population of it with vertices and edges. Next, I create a matrix representation of the graph, an adjacency list of the graph, and perform depth first and breadth first searches of the graph given the first vertex stored in the graph.

The second half of this assignment involved creation of a binary search tree that held a new list of magic items. With this binary search tree, I demonstrate it's population decisions in the form of L's and R's, as well as count the number of comparisons taken when looking up each item in this tree.

Lastly, I take the average number of comparisons and display it for the user to see. The average I was getting for 42 items was 5, which makes sense because the expected time for this lookup, as well as an in-order traversal, is 'h' where h is the height of the tree, while the worst case scenario would be O(n). This makes sense, because with each iteration of the tree we can either take one of two routes, and are presented with the same option until we reach the bottom of the tree- meaning the time it takes will depend on the height of the tree. And given the worst case data of a tree that is just a linked list, in order to search or traverse this tree we would have to go down by n elements. If it were to be perfectly balanced, however, we could expect a log base 2 (n) time complexity for both traversal and searches!

1.2 Graph

```java
    import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
public class Graph {

    public ArrayList<Vertex> vertices;
    private boolean isEmpty;
    private int highestIdFound;

    public Graph() {
        this.vertices = new ArrayList<>();
        this.isEmpty = true;
    }

    public Vertex findVertexById(int id) {

        // Loop through vertices list to find a vertex with our desired id.
        Vertex v;
        for (int i = 0; i < vertices.size(); i++) {
            v = vertices.get(i);
            if (v.getId() == id) {
                return v;
            }
        }

        // Vertex does not exist.
        return new Vertex(-1);
    }

    // Static version of previous function. Used for searching.
    private static Vertex getVertexById(int id, Graph g) {

        // Loop through vertices list to find a vertex with our desired id.
        Vertex v;
        for (int i = 0; i < g.vertices.size(); i++) {
            v = g.vertices.get(i);
            if (v.getId() == id) {
                return v;
            }
        }

        // Vertex does not exist.
        return new Vertex(-1);
    }

    // Given an id, creates a vertex and adds it to the graph.
    public void addVertex(int id) {
        this.isEmpty = false;

        if (id > this.highestIdFound){
            this.highestIdFound = id;
        }

        if (findVertexById(id).id == id){ // if we can find it's id (as non negative 1)
            System.out.println("The vertex ID " + id +" already exists.");
            return;
        }


        Vertex vertex = new Vertex(id);

        vertices.add(vertex);
    }
```

```
    // Takes in two vertex id's and adds themselves to each others neighbor lists.
    public void addEdge(int vertex1, int vertex2) {
        // Add vertex2 to vertex 1's neighbor list.
        this.findVertexById(vertex1).addNeighbor(vertex2);

        // Add vertex1 to vertex2's neighbor list.
        this.findVertexById(vertex2).addNeighbor(vertex1);
    }

    // Looks at all vertices and edges inside this graph, and returns a matrix representation.
    public int[][] createMatrix(){

        int[][] matrix = new int[this.highestIdFound + 1][this.highestIdFound + 1];

        // Grab each vertex associated with each other and set their coordinates to 1.
        for (Vertex v : this.vertices) {
            int vertexId = v.getId();

            for (int neighborId : v.getNeighbors()){
                matrix[vertexId][neighborId] = 1;
                matrix[neighborId][vertexId] = 1;
            }
        }

        return matrix;
    }

    // Prints out adjacency list representation for each vertex in graph.
    // Initially, I was going to return it similar to createMatrix, however this was
    // a lot easier and more efficient as I do not have to loop once again for display.
    public void printList() {

        // Grab each vertex associated with each other and set their coordinates to 1.
        for (Vertex v : this.vertices) {
            int vertexId = v.getId();

            System.out.println("[" + vertexId + "]" + " ->" + v.getNeighbors().toString());
        }

    }

// In order to retrieve pointer to a vertex given an id, needed to create self as static
// function, as well as getVertexById.

// Depth First Search / Traversal prints out the ID's in order they are processed.
    public static void DFS(Vertex v, Graph g) {

        if (!v.isProcessed){
            System.out.println(v.id);
            v.isProcessed = true;
        }
        for (int neighborId : v.getNeighbors()){
            Vertex neighbor = getVertexById(neighborId, g); // Use neighborId to retrieve a pointer to vert
            if (!neighbor.isProcessed){
                DFS(neighbor, g);
            }
        }
    }

    // Breadth First Search / Traversal prints out the ID's in order they are processed.
    public static void BFS(Vertex v, Graph g) {

        Queue<Vertex> q = new LinkedList<>();

```

```
        q.add(v);
        v.isProcessed = true;

        while (!q.isEmpty()){
            Vertex currentVertex = q.remove();
            System.out.println(currentVertex.getId());

            for (int neighborId : currentVertex.getNeighbors()) {
                //System.out.println("-" + neighborId + "-");
                Vertex neighbor = getVertexById(neighborId, g); // Use neighborId to retrieve a pointer to
                if (!neighbor.isProcessed){
                    q.add(neighbor);
                    neighbor.isProcessed = true;
                }
            }
            //System.out.println("*" + q.toString() + "*");
        }

    }

    public boolean isEmpty(){
        return this.isEmpty;
    }

    public String toString(){
        String retStr = "";
        for (Vertex v : vertices){
            retStr += "   Vertex: " + v.id + "\n";
            retStr += v.toString();
            retStr += "\n\n";
        }
        return retStr;
    }

}
```

Graph utilized our Vertex class, similar to how LinkedLists utilize Node classes. However, there were other functions I made specific to this graph class that turned out to be very useful!

For one, I created a findVertexById function which traverses our list of vertices and returns a pointer to the vertex. This was super useful as it allowed me to add vertices ensuring no duplicates were created, as well as perform breadth and depth first traversals on our graph.

Next, I was able to implement breadth first searches and depth first searches given a graph and starting vertex. This at first gave me trouble as retrieving a pointer to a vertex given an id needed to be static, therefore I decided to make the methods static and take in their own graphs as a parameter.

1.3 Vertex

```
    import java.util.ArrayList;

public class Vertex {

    public String data;

    public int id;
    private ArrayList<Integer> neighbors;
    public boolean isProcessed = false;

    public Vertex(int id) {
        data = "";
        this.id = id;
        this.neighbors = new ArrayList<>();
    }

    // ID.
    public int getId() {
        return this.id;
    }

    // Neighbors.
    public ArrayList<Integer> getNeighbors() {
        return this.neighbors;
    }

    public void addNeighbor(int neighborId) {
        // Do not add neighbors more than once.
        if (this.neighbors.contains(neighborId)) {
            return;
        }
        this.neighbors.add(neighborId);
    }

    public String toString() {
        return "       neighbors:" + this.neighbors;
    }

}
```

This vertex class was quite simple, as it is very similar to my old Node class, however the big difference would be it's function to hold an unlimited amount of neighbors(theoretically), which were used to create pointers! It did this by storing neighbors as mere integer id's, which took advantage of my graph's findVertexById function.

1.4 Binary Search Tree

```
  public class BinarySearchTree {

    public static int comparisons = 0; // will record amt of comparisons for lookup

    // Holds pointer to the root of the tree.
    TreeNode root;
    public BinarySearchTree() {
        this.root = null; // initialize with null value.
    }

    // Checks if BinarySearchTree has a root node, calls real insert method.
    public void insert(TreeNode treeNode) {
        System.out.print("[");
```

```java
        // pretty cool; because the function is head recursive this will give us back our original root.
        this.root = recursiveInsert(this.root, treeNode);
    }

    // Uses recursion to check where treeNode should be inserted into.
    public TreeNode recursiveInsert(TreeNode currentRoot, TreeNode desiredInsertNode) {

        // Once we reach this statement, we can insert our desiredInsertNode into this slot
        // and unravel the recursion.
        if (currentRoot == null) {
            currentRoot = desiredInsertNode;
            System.out.println("]");
            return desiredInsertNode;
        }

        // Determine if we should move left or right, down our tree.
        if (currentRoot.value.compareTo(desiredInsertNode.value) < 0) {
            System.out.print("R, "); // We will be going to the right of our currentRoot.
            currentRoot.right = recursiveInsert(currentRoot.right, desiredInsertNode);
        } else if (currentRoot.value.compareTo(desiredInsertNode.value) > 0) {
            System.out.print("L, "); // We will be going to the left of our currentRoot.
            currentRoot.left = recursiveInsert(currentRoot.left, desiredInsertNode);
        }

        return currentRoot;

    }


    // Recursively performs in-order traversal of tree(alphabetical).
    public void traverseInOrder(TreeNode currentRoot) {
        if (currentRoot != null) {
            // Declare left of current root as the new root to look at.
            traverseInOrder(currentRoot.left);
            System.out.println(currentRoot.value);
            // Declare right of current root as the new root to look at.
            traverseInOrder(currentRoot.right);
        }
    }

    // User's lookup function to return how many iterations of
    public int lookup(String desiredNodeValue) {
        comparisons = 0;
        System.out.print("[");
        this.find(this.root, desiredNodeValue);
        System.out.print("] ");
        return comparisons;

    }


    // Actual searching function to find a desired tree node.
    private TreeNode find(TreeNode currentRoot, String desiredNodeValue) {
        comparisons++;
        // We've either found our TreeNode, or it does not exist.
        if ( currentRoot == null || currentRoot.value.equals(desiredNodeValue)){
            return currentRoot;
        }

        // Desired tree node is greater than(to the right) of current root.
        if ( currentRoot.value.compareTo(desiredNodeValue) < 0 ) {
            System.out.print("R, ");
            return find(currentRoot.right, desiredNodeValue);
        }

```

```
        // otherwise, desired tree node is less than(to the left) of current root.
        else {
            System.out.print("L, ");
            return find(currentRoot.left, desiredNodeValue);
        }
  }

}
public class TreeNode {

    String value;
    TreeNode left;
    TreeNode right;

    public TreeNode(String value) {
        this.value = value;
        left = null;
        right = null;
    }


}
```

This was my implementation of the binary search tree class! It used a small TreeNode class that just holds a string value, and two TreeNode pointers as left and right. This allowed me to create a binary search tree which holds functions such as insertion, traversals, and lookups for the purposes of this assignment. I was able to record the amount of comparisons taken by having the lookup function reset our comparisons counter to 0, then having my recursive function count how many comparisons it's taken throughout each iteration, and lastly printing that same counter once it finishes recurring.

Overall, this assignment was a fun experience to play around with binary search trees and graphs, and it allowed me to understand how these data structures work through experience creating their core functions!