1 Node Classes

These Node classes are used to make the BST and the graphs. The BST uses the left and right attributes to find the next node to insert and to find a node in the tree. Graphs uses the neighbors attributes to find which nodes the current node is connected to.

```
import java.util.ArrayList;

public class TreeNode {
    String val = null;
    TreeNode left= null;
    TreeNode right =null;
}


public class GraphNode {
    String id;
    ArrayList<GraphNode> neighbors;
    boolean processed = false;
}
```

2 Making the BST

Making the Binary Search Tree is as simple as iterating through the tree until we find the spot where there is no node and a node can be placed in that spot. This function uses the TreeNode class to make the tree. It starts with a root node which is initialized outside of the while loop. The while loop will iterate through the BST until it finds an unoccupied spot for a node. Inserting a node in a BST is logarithmic.

```
public static TreeNode makeBST(String[] findItems, String[] magicItems) {

        TreeNode root = new TreeNode();
        root.val = magicItems[0];
        TreeNode tempRoot = root;
        // Goes through the whole array
        for (int i = 1; i < magicItems.length; i++) {
            // inserted is automatically false until its in the BST.
            boolean inserted = false;
            TreeNode nextNode = new TreeNode();
            nextNode.val = magicItems[i];

            while (!inserted) {
                // This will check to see if the current item is less than the root value.
                if (magicItems[i].compareToIgnoreCase(root.val) <= 0) {
                    // if root.left is null then it will put the current magic item there.
                    if (root.left == null) {
                        System.out.print("L ");
                        root.left = nextNode;
                        root = tempRoot;
                        inserted = true;
                    } else {
                        // if root.left isn't null, we traverse to the left.
                        root = root.left;
                    }
                } else {
                    // if root.right is null then it will put the current magic item there.
                    if (root.right == null) {
                        System.out.print("R ");
                        root.right = nextNode;
                        root = tempRoot;
                        inserted = true;
                    }
                    // if root.right isn't null then we traverse to the right of the current node
                    else {
```

```
                    root = root.right;

                }
            }
        }

    }

    return root;
}
```

3 Searching the BST

Searching the BST is also done in logarithmic time. This search that I'm using is recursive. There is also an iterative solution to this problem. The function starts by getting called in the main function with the root node and the search element as arguments. The element being searched for is compared against the root and if it's smaller, the same function is called with root.left. If the element being searched for is bigger than root. the same function is called with root.right.

```
public static void searchBST(TreeNode root, String findElement) {

    if (findElement.equals(root.val)) {
        System.out.println("Compares:" + currentCount);
        currentCount = 0;
    } else {

        // This goes to the left side of the tree if the element is less than the root
        // Node
        if (findElement.compareToIgnoreCase(root.val) <= 0) {
            countCo += 1;
            currentCount += 1;
            System.out.print("L ");
            root = root.left;
            // Recursive search on the current Node.
            searchBST(root, findElement);

        }
        // otherwise this recursively goes to the right side of the tree until we find
        // the element
        else {
            countCo += 1;
            currentCount += 1;
            System.out.print("R ");
            root = root.right;
            searchBST(root, findElement);
        }
    }
}
```

4 In order traversal

The In order Travsersal is pretty simple also. The function is given a root node and will print all of the nodes from the left, center and right respectively.

```
// In Order traversal of a binary Tree. Binary Search Trees happen to be Binary
    // Trees.
    public static void inOrderTraversal(TreeNode root) {
        if (root == null) {
            return;
        }
        // This will go all the way to the left side of the tree and up.
        inOrderTraversal(root.left);
```

```
        // This will print the current node
        System.out.println(root.val + " ");

        // This will print the right side of the current node
        inOrderTraversal(root.right);
    }
```

5 Depth first Traversal

Depth First traversal is just how it sounds. It will go to the maximum depth of a node and print out all of the nodes recursively in that path.

```
public static void DepthFirst(GraphNode startNode) {
        if (startNode.id == null) {
            return;
        } else {
            if (!startNode.processed) {
                System.out.println(startNode.id);
                startNode.processed = true;

            }

            startNode.neighbors.forEach(action -> {
                if (!action.processed) {
                    DepthFirst(startNode);
                }
            });

        }
    }
```

6 Breadth first Traversal

Breadth first Traversal is also known as level order traversal because the algorithm goes through the tree at a level order first. For example, It will print out the root node first. Then it will print out all the nodes on the second level, third level, etc. Respectively. Breadth first is used to solve many problems.

```
public static void BreadthFirst(GraphNode node) {
        Queue<GraphNode> q = new LinkedList<>();
        node.processed = true;
        q.add(node);
        while (q.size() > 0) {
            GraphNode cv = q.remove();
            System.out.println(cv.id);
            cv.neighbors.forEach(action -> {
                if (!action.processed) {
                    action.processed = true;
                    q.add(action);

                }
            });
        }
    }
```

7 Making Graphs

It looks like there is a lot going on in this function, but there really isn't. I'm reading the graph file and checking to see if the line says "new graph","vertex", or "edge". If the new line is "new graph", The function will print out the matrix, graph and adjacency list. If the new line is "new vertex", the function is going to initialize a GraphNode with the corresponding id, initialize a new matrix row or create a new key in the hashmap. If the new line is "new edge", I'm going to add the neighbors in the graph, add the 1's in the matrix and add the edge to the key.

```
public static void makeGraphs(String[] graphArray) {

        //All of the datastructures used to store the nodes
        HashMap<String, List<String>> newGraph = new HashMap<String, List<String>>();
        ArrayList<int[]> matrix = new ArrayList<int[]>();
        GraphNode graph = new GraphNode();
        ArrayList<GraphNode> listOfNodes = new ArrayList<GraphNode>();


        //This goes through the graphs array which contains the contents of the txt file
        for (int i = 0; i < graphArray.length; i++) {

            //splits the line by spaces to find the numbers
            String findArray[] = graphArray[i].split(" ");

            if (graphArray[i].equals("new graph") || i == graphArray.length - 1) {



            // Printing and display the elements in ArrayList

                System.out.println(" ");
                //This prints out the Current adjacency List
                newGraph.entrySet().forEach(entry -> {
                    System.out.println("key:" + entry.getKey() + " Value:" + entry.getValue());
                });
                System.out.println(" ");

                //prints out the breadth First and depth First traversal.
                if (listOfNodes.size() != 0) {

                    BreadthFirst(listOfNodes.get(0),listOfNodes);
                    DepthFirst(listOfNodes.get(0));
                }

                //This prints out the current matrix
                for (int o = 0; o < matrix.size(); o ++){
                    System.out.println(" ");
                    for (int k = 0; k < matrix.get(0).length; k ++){
                        System.out.print(matrix.get(o)[k]);
                    }
                }
                System.out.println(" ");
                //Clear the adjacency list, matrix and nodes.
                matrix = new ArrayList<int[]>();
                newGraph.clear();
                listOfNodes.clear();
                matrixCount =0;
                flag = 0;


            }

            else {

                if (findArray[0].equals("add")) {

                    if (findArray[1].equals("vertex")) {
                        newGraph.put(findArray[2], new ArrayList<String>());

                        matrixCount +=1;


                        //creates a new node in the graph
```

```
                    graph = new GraphNode();
                    graph.id = findArray[2];
                    graph.neighbors = new ArrayList<GraphNode>();
                    listOfNodes.add(graph);

                } else {
                    // This adds both the adjacencys to the dictionary.
                    newGraph.get(findArray[2]).add(findArray[4]);
                    newGraph.get(findArray[4]).add(findArray[2]);

                    if (flag ==0){
                        flag = 1;
                        for(int j = 0; j < matrixCount + 1; j++ ){
                            matrix.add(new int[matrixCount + 1] );
                        }
                    }
                    //These will add the number one to the correct positions in the matrix
        matrix.get(Integer.parseInt(findArray[2]))[Integer.parseInt(findArray[4])] = 1;
        matrix.get(Integer.parseInt(findArray[4]))[Integer.parseInt(findArray[2])] = 1;
                    //Creating 2 new graph nodes. Each for the 2 elements.
                    GraphNode newNodeInGraph = new GraphNode();
                    newNodeInGraph.id = findArray[2];
                    newNodeInGraph.neighbors = new ArrayList<GraphNode>();

                    GraphNode anotherNodeInGraph = new GraphNode();
                    anotherNodeInGraph.id = findArray[4];
                    anotherNodeInGraph.neighbors = new ArrayList<GraphNode>();

                    //Finds each of the nodes in the file and inserts their neighbors.
                    listOfNodes.forEach(action -> {
                        if (action.id.equals(findArray[4])) {
                            action.neighbors.add(newNodeInGraph);

                        }
                        if (action.id.equals(findArray[2])) {
                            action.neighbors.add(anotherNodeInGraph);
                        }

                    });

                }

            }

        }

    }
}
```

8 Number of comparisons

Depth First	Depth First	BST Search
N/A	N/A	9
O(—V— + —E—)	O(—V— + —E—	log n

Binary Search is a logarithmic function because it splits the total amount by half every time. The average look up time in a tree of 666 is 9. It was split in half 9 times to find the element it was searching for.

Depth first Search and Breadth first Search are both the cardinality of the number of edges and the cardinality of the number of vertices combined. It isn't enough to say these algorithms are o(n) because they also have edges that need to be traversed too.

The cardinality is the part to make sure there are no duplicates.