

# Assignment Four – Graphs and Trees

---

Genevieve Anderson  
Genevieve.anderson1@marist.edu

November 19, 2022

## 1 BST Class

In my BST class I build the framework for my binary search tree. I declared that in each instance of a binary search tree being created there would be a root Node, path String, comparison integer, findPath String (for when we are searching for nodes rather than inserting), a findComparisons integer (for when we are counting comparisons when searching rather than inserting), totalComparisons integer, and findTotalComparisons integer (for when we are counting total comparisons in searching rather than inserting) [3-9]. I also initialized all of these variables in my "BST()" constructor [12].

Then, I created my "setRoot()" function [23]. Which is where I declare that the root of the tree will be a new node with a provided value inside.

Then, I created my "insert()" function [28]. This is the algorithm I created for inserting nodes into the tree. In Software Development 1, our final project was to create a binary search tree. When I did the final project, I used recursion for my insertion method so I was able to recycle some of this code from that project! In my function, the node that will be inserted is first compared to see if it is being compared to a null value [29], this would indicate that this is where the node being inserted should be placed. If not, the node being inserted is being compared as "greater than" or "less than" the "current" node we are looking at in the tree [33, 37]. Based on which side of the "current" node that the node being inserted is placed, the path and the comparisons are accounted for each iteration of the recursion. Based on the comparison, recursion of the insert statement is then called to see how far down the node should be placed. After this function is run, all nodes will be in there correct placements.

Next, I created my "find()" function [46]. This is the algorithm I created for finding nodes in the tree after they are inserted. First, the "findComparisons"

are incremented [47] since a comparison is made at the beginning each time. Then, the target that is being searched is first compared to the "current" node to see if they are equal [48]. This would indicate that the target has been found. If not, the target node we are trying to find is then recursively compared to the "current" node as "less than" and "greater than" [51, 55]. Based on what side of the "current" node is being searched, the path is updated accordingly.

Next, I have a few simple functions that are concerned with formatting and calculating results of my binary search tree. I have my "path()" function [63]. This is where I print the path and comparisons associated with the insertion of nodes in the tree. Next, I have my "findPath()" function [71]. This is where I print the path and comparisons associated with the finding of nodes in the tree. Then, I have my "average()" function [79]. This is where I calculate the average. This function can be used for calculating the average for both inserting and finding!

Lastly, I have my "ITW()" function [84]. This is my function for in order traversal of the tree. This function recursively prints all nodes of the tree in ascending order. It first visits the left side of the tree, then prints, then the right side of the tree.

```

1 public class BST {
2
3     Node root;
4     String path;
5     int comparisons;
6     String findPath; // For when we are finding the path of the
7                       // target being searched in the tree
8     int findComparisons; // For when we are finding the target in
9                       // the tree
10    int totalComparisons; // For the average
11    int findTotalComparisons;
12
13    // Constructor
14    public BST() {
15        root = null;
16        path = "";
17        comparisons = 0;
18        findPath = "";
19        findComparisons = 0;
20        totalComparisons = 0;
21        findTotalComparisons = 0;
22    }
23
24    // Function for setting the root
25    public void setRoot(String value) {
26        root = new Node(value);
27    }
28
29    // Function for inserting nodes into the tree
30    public Node insert(Node node, String value) {
31        if (node == null) {
32            return new Node(value); // New node we are inserting
33        }
34    }
35
36    // Function for deleting nodes from the tree
37    public Node delete(Node node, String value) {
38        if (node == null) {
39            return null;
40        }
41        if (node.value == value) {
42            return delete(node.left, value);
43        }
44        if (node.value < value) {
45            node.left = delete(node.left, value);
46        }
47        if (node.value > value) {
48            node.right = delete(node.right, value);
49        }
50        return node;
51    }
52
53    // Function for finding the path of the target
54    public String findPath(Node node, String value) {
55        if (node == null) {
56            return null;
57        }
58        if (node.value == value) {
59            return path;
60        }
61        if (node.value < value) {
62            path = path + "L ";
63            return findPath(node.left, value);
64        }
65        if (node.value > value) {
66            path = path + "R ";
67            return findPath(node.right, value);
68        }
69        return null;
70    }
71
72    // Function for finding the target
73    public boolean findTarget(Node node, String value) {
74        if (node == null) {
75            return false;
76        }
77        if (node.value == value) {
78            return true;
79        }
80        if (node.value < value) {
81            return findTarget(node.left, value);
82        }
83        if (node.value > value) {
84            return findTarget(node.right, value);
85        }
86        return false;
87    }
88
89    // Function for finding the average
90    public double findAverage() {
91        return (double) totalComparisons / comparisons;
92    }
93
94    // Function for finding the total comparisons
95    public int findTotalComparisons() {
96        return totalComparisons;
97    }
98
99    // Function for finding the path of the target
100    public String findPath() {
101        return findPath;
102    }
103
104    // Function for finding the target
105    public boolean findTarget() {
106        return findTarget;
107    }
108
109    // Function for finding the average
110    public double findAverage() {
111        return findAverage;
112    }
113
114    // Function for finding the total comparisons
115    public int findTotalComparisons() {
116        return findTotalComparisons;
117    }
118
119    // Function for finding the path of the target
120    public String findPath() {
121        return findPath;
122    }
123
124    // Function for finding the target
125    public boolean findTarget() {
126        return findTarget;
127    }
128
129    // Function for finding the average
130    public double findAverage() {
131        return findAverage;
132    }
133
134    // Function for finding the total comparisons
135    public int findTotalComparisons() {
136        return findTotalComparisons;
137    }
138
139    // Function for finding the path of the target
140    public String findPath() {
141        return findPath;
142    }
143
144    // Function for finding the target
145    public boolean findTarget() {
146        return findTarget;
147    }
148
149    // Function for finding the average
150    public double findAverage() {
151        return findAverage;
152    }
153
154    // Function for finding the total comparisons
155    public int findTotalComparisons() {
156        return findTotalComparisons;
157    }
158
159    // Function for finding the path of the target
160    public String findPath() {
161        return findPath;
162    }
163
164    // Function for finding the target
165    public boolean findTarget() {
166        return findTarget;
167    }
168
169    // Function for finding the average
170    public double findAverage() {
171        return findAverage;
172    }
173
174    // Function for finding the total comparisons
175    public int findTotalComparisons() {
176        return findTotalComparisons;
177    }
178
179    // Function for finding the path of the target
180    public String findPath() {
181        return findPath;
182    }
183
184    // Function for finding the target
185    public boolean findTarget() {
186        return findTarget;
187    }
188
189    // Function for finding the average
190    public double findAverage() {
191        return findAverage;
192    }
193
194    // Function for finding the total comparisons
195    public int findTotalComparisons() {
196        return findTotalComparisons;
197    }
198
199    // Function for finding the path of the target
200    public String findPath() {
201        return findPath;
202    }
203
204    // Function for finding the target
205    public boolean findTarget() {
206        return findTarget;
207    }
208
209    // Function for finding the average
210    public double findAverage() {
211        return findAverage;
212    }
213
214    // Function for finding the total comparisons
215    public int findTotalComparisons() {
216        return findTotalComparisons;
217    }
218
219    // Function for finding the path of the target
220    public String findPath() {
221        return findPath;
222    }
223
224    // Function for finding the target
225    public boolean findTarget() {
226        return findTarget;
227    }
228
229    // Function for finding the average
230    public double findAverage() {
231        return findAverage;
232    }
233
234    // Function for finding the total comparisons
235    public int findTotalComparisons() {
236        return findTotalComparisons;
237    }
238
239    // Function for finding the path of the target
240    public String findPath() {
241        return findPath;
242    }
243
244    // Function for finding the target
245    public boolean findTarget() {
246        return findTarget;
247    }
248
249    // Function for finding the average
250    public double findAverage() {
251        return findAverage;
252    }
253
254    // Function for finding the total comparisons
255    public int findTotalComparisons() {
256        return findTotalComparisons;
257    }
258
259    // Function for finding the path of the target
260    public String findPath() {
261        return findPath;
262    }
263
264    // Function for finding the target
265    public boolean findTarget() {
266        return findTarget;
267    }
268
269    // Function for finding the average
270    public double findAverage() {
271        return findAverage;
272    }
273
274    // Function for finding the total comparisons
275    public int findTotalComparisons() {
276        return findTotalComparisons;
277    }
278
279    // Function for finding the path of the target
280    public String findPath() {
281        return findPath;
282    }
283
284    // Function for finding the target
285    public boolean findTarget() {
286        return findTarget;
287    }
288
289    // Function for finding the average
290    public double findAverage() {
291        return findAverage;
292    }
293
294    // Function for finding the total comparisons
295    public int findTotalComparisons() {
296        return findTotalComparisons;
297    }
298
299    // Function for finding the path of the target
300    public String findPath() {
301        return findPath;
302    }
303
304    // Function for finding the target
305    public boolean findTarget() {
306        return findTarget;
307    }
308
309    // Function for finding the average
310    public double findAverage() {
311        return findAverage;
312    }
313
314    // Function for finding the total comparisons
315    public int findTotalComparisons() {
316        return findTotalComparisons;
317    }
318
319    // Function for finding the path of the target
320    public String findPath() {
321        return findPath;
322    }
323
324    // Function for finding the target
325    public boolean findTarget() {
326        return findTarget;
327    }
328
329    // Function for finding the average
330    public double findAverage() {
331        return findAverage;
332    }
333
334    // Function for finding the total comparisons
335    public int findTotalComparisons() {
336        return findTotalComparisons;
337    }
338
339    // Function for finding the path of the target
340    public String findPath() {
341        return findPath;
342    }
343
344    // Function for finding the target
345    public boolean findTarget() {
346        return findTarget;
347    }
348
349    // Function for finding the average
350    public double findAverage() {
351        return findAverage;
352    }
353
354    // Function for finding the total comparisons
355    public int findTotalComparisons() {
356        return findTotalComparisons;
357    }
358
359    // Function for finding the path of the target
360    public String findPath() {
361        return findPath;
362    }
363
364    // Function for finding the target
365    public boolean findTarget() {
366        return findTarget;
367    }
368
369    // Function for finding the average
370    public double findAverage() {
371        return findAverage;
372    }
373
374    // Function for finding the total comparisons
375    public int findTotalComparisons() {
376        return findTotalComparisons;
377    }
378
379    // Function for finding the path of the target
380    public String findPath() {
381        return findPath;
382    }
383
384    // Function for finding the target
385    public boolean findTarget() {
386        return findTarget;
387    }
388
389    // Function for finding the average
390    public double findAverage() {
391        return findAverage;
392    }
393
394    // Function for finding the total comparisons
395    public int findTotalComparisons() {
396        return findTotalComparisons;
397    }
398
399    // Function for finding the path of the target
400    public String findPath() {
401        return findPath;
402    }
403
404    // Function for finding the target
405    public boolean findTarget() {
406        return findTarget;
407    }
408
409    // Function for finding the average
410    public double findAverage() {
411        return findAverage;
412    }
413
414    // Function for finding the total comparisons
415    public int findTotalComparisons() {
416        return findTotalComparisons;
417    }
418
419    // Function for finding the path of the target
420    public String findPath() {
421        return findPath;
422    }
423
424    // Function for finding the target
425    public boolean findTarget() {
426        return findTarget;
427    }
428
429    // Function for finding the average
430    public double findAverage() {
431        return findAverage;
432    }
433
434    // Function for finding the total comparisons
435    public int findTotalComparisons() {
436        return findTotalComparisons;
437    }
438
439    // Function for finding the path of the target
440    public String findPath() {
441        return findPath;
442    }
443
444    // Function for finding the target
445    public boolean findTarget() {
446        return findTarget;
447    }
448
449    // Function for finding the average
450    public double findAverage() {
451        return findAverage;
452    }
453
454    // Function for finding the total comparisons
455    public int findTotalComparisons() {
456        return findTotalComparisons;
457    }
458
459    // Function for finding the path of the target
460    public String findPath() {
461        return findPath;
462    }
463
464    // Function for finding the target
465    public boolean findTarget() {
466        return findTarget;
467    }
468
469    // Function for finding the average
470    public double findAverage() {
471        return findAverage;
472    }
473
474    // Function for finding the total comparisons
475   
```

```

32     // Determines which side of tree to place node, will
    compare the value we are trying to insert to the "current" node
33     if (value.compareTo(node.getName()) < 0) { // Less than "
        current" node
34         node.left = insert(node.left, value); // Recursively
        inserts node on the left
35         path += "L, "; // Used when printing path of insertion
36         comparisons++; // Increment comparisons
37     } else { // Greater than "current" node
38         node.right = insert(node.right, value); // Recursively
        inserts node on the right
39         path += "R, ";
40         comparisons++;
41     }
42     return node;
43 }
44
45 // Function for finding nodes in the tree
46 public Node find(Node node, String target) {
47     findComparisons++; // Increment comparisons for find,
    different variable than comparisons for insert
48     if (target.equals(node.getName())) { // When the node has
        been found
49         return node;
50     }
51     else if (target.compareTo(node.getName()) < 0) { //
        Searching left hand side of tree
52         findPath += "L, "; // Used when printing path of search
53         find(node.getLeft(), target); // Recursively finding
        the node on the left
54     }
55     else { // Searching right hand side
56         findPath += "R, ";
57         find(node.getRight(), target);
58     }
59     return null; // If the target is not found
60 }
61
62 // Function for printing the path and comparisons of when they
    are inserted
63 public void path(String name) {
64     System.out.println(name + ": " + path + " and the number of
        comparisons is: " + comparisons);
65     path = "";
66     totalComparisons += comparisons;
67     comparisons = 0;
68 }
69
70 // Function for printing the path and comparisons for when they
    are searched
71 public void findPath(String name) {
72     System.out.println(name + ": " + findPath + " and the
        number of comparisons is: " + findComparisons);
73     findPath = "";
74     findTotalComparisons += findComparisons;
75     findComparisons = 0;
76 }

```

```

77
78 // Function for calculating the average
79 public void average(int avgComparisons, int size) {
80     double average = ((double)(avgComparisons))/size; // double
      for decimal places
81     System.out.println("The average is: " + String.format("%.2f
      ", average));
82 }
83
84 // Function for ITW
85 public void ITW(Node node) {
86     if (node == null) {
87         return;
88     }
89     // Prints nodes in order
90     ITW(node.left);
91     System.out.println(node.getName());
92     ITW(node.right);
93 }
94
95
96 }

```

## 2 Node Class

Now, I have my Node class. This class is referenced for the binary search tree.

I was able to recycle some of this code from previous assignments! However, I needed to additionally initialize the left and right nodes to null [4, 5]. I also needed to add "getLeft()" [19] and "getRight()" [23] to return the left or right nodes in the tree. I also needed to add "setLeft()" [32] and "setRight()" [36] for setting the left or right nodes of the tree.

```
1 // Building the framework for each node
2 public class Node {
3     String name = ""; // Declaring and initializing the name inside
4     // the node
5     Node left = null;
6     Node right = null;
7
8     // Node constructor
9     public Node(String n) { // The first parameter is for the
10        // information the node is holding, the second parameter is for
11        // the pointer
12        this.name = n; // Initializing the name
13        this.left = null;
14        this.right = null;
15    }
16
17    // Getters
18    public String getName() { // Getting the name variable from the
19        // node, returns a string
20        return name; // Returns the information from the variable
21        name
22    }
23
24    public Node getLeft() {
25        return left;
26    }
27
28    public Node getRight() {
29        return right;
30    }
31
32    // Setters
33    public void setName(String n) {
34        name = n;
35    }
36
37    public void setLeft(Node m) {
38        left = m;
39    }
40
41    public void setRight(Node m) {
42        right = m;
43    }
44
45    // toString so that the program prints what is actually inside
```

```
41     the node rather than the object identifier
42     public String toString() {
43         String result = name; // Setting a string equal to what is
44         inside the node
45         return result;
46     }
47 }
```

### 3 Graph class

In my graph class I build the framework for each of my graphs.

First I declare that each instance of a graph will have an vertex list, and adjacency list, and a matrix representation of the graph [5-7]. I then initialize all of these variables in both a constructor [10], and a parameterized constructor [17]. Then, I have my "setGraph()" function [24] which is where I set the vertex list to contain the vertices, the adjacency list and matrix to contain the size of the vertex list plus one to account for the formatting.

Now, I have my "matrix()" function [31]. This is where I actually build the matrix representation for each graph. I used a nested for loop that goes through the rows and columns of each matrix [32,33]. Inside these loops, I have a series of if else statements. First, I account for the row and column labels for the matrix [34, 37]. Then, I am concerned with the inside of the matrix in my last else if statement [40]. Here, I have another series of if else statements that go through each point to see whether or not an edge exists [42, 45]. If there is an edge, the matrix will indicate so by saying "1" instead of "0".

Now, I have my "adjList()" function [54]. This is where I create the adjacency list for each matrix. I use a for loop to go through the vertex list [56]. Then, each vertex is printed with a list of its neighbors.

Now, I have my "printResults()" function [64]. This is where I format and print the results of the graph. For printing the matrix, I use a nested for loop which goes through the the matrix and prints the rows and columns accordingly [65, 67]. Then I simply print the adjacency list [72].

Now, I have my functions for the traversals of the graphs. First, I have my "DFS()" function [76] for the depth first traversal. I used the pseudo code from the class notes to write this algorithm which goes "deep before wide." Before each node is "processed," the depth first search is recursively called [83]. Next, I have my "BFS()" function [89] for the breadth first traversal. Again, I used the pseudocode from the class notes to write this algorithm which goes "wide before deep." This algorithm uses a queue to dequeue and queue the vertices in order to become processed. Lastly, concerning the traversals, I have my "reset-Traversal()" function [106] which simply resets the processed condition and the pointers prior to each traversal.

```
1 import java.util.ArrayList;
2 import java.util.Arrays;
3
4 public class Graph {
5     ArrayList<Vertex> vertexList;
6     int [][] adjList; // 2d array - array of an array
7     int [][] matrix;
8
9     // Constructor
10    public Graph() {
11        vertexList = null;
12        adjList = null;
13        matrix = null;
```

```

14     }
15
16     // Parameterized Constructor
17     public Graph(ArrayList<Vertex> vList) {
18         vertexList = vList;
19         adjList = new int[vertexList.size() + 1][vertexList.size()
20 + 1]; // +1 for formatting of the matrix
21         matrix = new int[vertexList.size() + 1][vertexList.size() +
22 1];
23     }
24
25     // Setting the Graph
26     public void setGraph(ArrayList<Vertex> vertices) {
27         vertexList = vertices;
28         adjList = new int[vertexList.size() + 1][vertexList.size()
29 + 1];
30         matrix = new int[vertexList.size() + 1][vertexList.size() +
31 1];
32     }
33
34     // Function for building matrix
35     public void matrix() {
36         for (int r = 0; r < matrix.length; r++) {
37             for (int c = 0; c < matrix.length; c++) {
38                 if (r == 0 && c != 0) {
39                     matrix[r][c] = c; // for making the list of
40 numbers at the top for formatting
41                 }
42                 else if (c == 0 && r != 0) {
43                     matrix[r][c] = r; // for making the list of
44 numbers at the side for formatting
45                 }
46                 else if (c != 0 && r != 0) { // Inside the matrix
47                     boolean edge = vertexList.get(r - 1).haveEdge(c
48 ); // at each point in the matrix going through to see if it
49 has an edge
50
51                     if (edge == true) {
52                         matrix[r][c] = 1; // If there is an edge
53 the matrix point will be set to 1
54                     }
55                     else {
56                         matrix[r][c] = 0; // If there isnt an edge
57                     }
58                 }
59             }
60         }
61     }
62
63     // Function for adjacency list
64     public String adjList() {
65         String result = "";
66         for (int i = 0; i < vertexList.size(); i++) {
67             Vertex v = vertexList.get(i); // Individual vertex
68             result += "[" + v.getVertexByID() + "]" + Arrays.
69 toString(v.neighbors.toArray()) + "\n"; // Formatting the
70 adjacency list
71         }
72     }

```



```

60         return result;
61     }
62
63     // Function for printing the results of the graph
64     public void printResults() {
65         for (int r = 0; r < matrix[0].length; r++) {
66             System.out.print("[");
67             for (int c = 0; c < matrix[0].length; c++) {
68                 System.out.print(matrix[r][c] + ", ");
69             }
70             System.out.println("]"); // For each line of the matrix
71         }
72         System.out.println(adjList()); // Printing the adjacency
73         list
74     }
75
76     // Function for depth first traversal
77     public void DFS(Vertex v) {
78         if (v.processed == false) {
79             System.out.println(v.getVertexByID());
80             v.processed = true;
81         }
82         for (Vertex n: vertexList) { // for all the vertices in the
83             vertex list
84                 if (n.processed == false) {
85                     DFS(n);
86                 }
87             }
88         }
89
90     // Function for breadth first traversal
91     public void BFS(Vertex v) {
92         Queue queue = new Queue();
93         queue.enqueue(v);
94         v.processed = true;
95         while (queue.isEmpty() == false) {
96             Vertex currentVertex = queue.dequeue();
97             System.out.println(currentVertex.getVertexByID());
98             for (Vertex n: currentVertex.neighbors()) {
99                 if (n.processed == false) {
100                     queue.enqueue(n);
101                     n.processed = true;
102                 }
103             }
104         }
105
106     // Function for resetting prior to each traversale
107     public void resetTraversal() {
108         for (Vertex n: vertexList) {
109             n.processed = false; // Making them all false again
110             n.setNext(null);
111         }
112     }

```

## 4 Queue Class

Here I build the framework for my queue that is utilized in the breadth first traversal. I was able to recycle this code completely from assignment 1! However, in assignment 1 the algorithm was concerned with nodes in a linked list, so I slightly change my code around to deal with vertices in a graph!

```
1 //Building the framework for the queue
2 public class Queue {
3     //I am declaring the head and tail inside the queue, then
4     //initialize them in the constructor
5     Vertex head; //Queue needs both head and tail, whereas the
6     //stack only needed a head
7     Vertex tail;
8
9     //Queue constructor
10    public Queue() {
11        head = null; //Initializing the head
12        tail = null; //Initializing the tail
13    }
14
15    //Function for enqueue
16    public void enqueue(Vertex y) { //Parameter is a Vertex because
17        //we will be a new Vertex being enqueued in the queue
18        if (head == null) { //If the queue was empty, then the if
19            //loop will make it so that the incoming Vertex to the queue will
20            //become both the head and the tail
21            tail = y; //Setting the incoming Vertex to the tail
22            head = tail; //Setting the head to the tail
23        }
24        else { //The else statement is for if there are already
25            //Vertices in the queue
26            tail.setNext(y);
27            tail = y; //Setting the incoming Vertex to the tail
28        }
29    }
30
31    //Function for dequeue
32    public Vertex dequeue() {
33        Vertex oldHead = head; //Creating old head makes it so that
34        //we don't completely lose the Vertex that is getting removed
35        //from the queue
36        head = head.getNext(); //The new head is going to be the
37        //head that was originally next in line
38        if (head == null) { //If the head is null, then the tail
39            //should also be null
40            tail = null; //Doing this because otherwise, the tail
41            //would still be pointing at the Vertex we are returning
42        }
43        return oldHead;
44    }
45
46    //Function for checking if the queue is empty
47    public Boolean isEmpty() {
48        if (head == null) {
49            return true;
50        }
51    }
52 }
```

```
39     }
40     else {
41         return false;
42     }
43 }
44 }
```

## 5 Vertex Class

My Vertex class is a modified node class. First, I declare and initialize the variables for processed, id, neighbors, and next [10-23]. First I have my "add()" function [26] which is for adding vertices to the graph. Then, I have my "isProcessed()" function [31] which is for determining whether or not a node has been processed during the traversals. My "getNext()" function [35] is for returning the "next" node. My "setNext()" function [39] is utilized in my queue class as in assignment 1. My "getVertexByID()" function [44] is utilized whenever a vertex id is needed, such as printing results. My "haveEdge()" function [49] uses a simple for loop and if statement to determine whether of two vertices have an edge by going through the neighbors and seeing if their id's match. Then, my "neighbors()" function [60] returns the neighbors of a vertex. Lastly, I have a "toString()" function [65] which simply converts the object identifier of a vertex to the id we are interested in looking at.

```
1 import java.util.ArrayList;
2
3 public class Vertex {
4     boolean processed;
5     int id;
6     ArrayList<Vertex> neighbors;
7     Vertex next;
8
9     // Constructor
10    public Vertex() {
11        id = 0;
12        processed = false;
13        neighbors = new ArrayList<Vertex>();
14        next = null;
15    }
16
17    // Parameterized Constructor, if we need to make a vertex with
18    // the provided id
19    public Vertex(int id) {
20        this.id = id;
21        processed = false;
22        neighbors = new ArrayList<Vertex>();
23        next = null;
24    }
25
26    // Function for adding edge to vertex
27    public void add(Vertex v) {
28        neighbors.add(v);
29    }
30
31    // Function for changing processed
32    public void isProcessed() {
33        processed = true;
34    }
35
36    public Vertex getNext() { //Returning a node
37        return next; //Next because we are calling the pointer next
38    }
39 }
```

```

37     , returns the node from the variable next
38     }
39     public void setNext(Vertex m) { //"m" is the node that we are
40         going to set next equal to, for the pointer
41         next = m; //I am using m so that the pointer is not null.
42         the pointer will not be null until the end of the linked list.
43     }
44     // Function for getting the id
45     public int getVertexByID() {
46         return id;
47     }
48     // Function for checking edge
49     public boolean haveEdge(int id) {
50         boolean check = false;
51         for (int i = 0; i < neighbors.size(); i++) {
52             if (neighbors.get(i).getVertexByID() == id) {
53                 check = true; // means there is an edge
54             }
55         }
56         return check;
57     }
58
59     // Function to return all neighbors of a vertex
60     public ArrayList<Vertex> neighbors() {
61         return neighbors;
62     }
63
64     // toString for printing results
65     public String toString() {
66         return id + ""; // Converting the value being stored to a
67         string so that the adjList can be printed and formatted
68         correctly
69     }
70 }

```

## 6 Main Program

Lastly, I have my Main class which deals with both the BST algorithm and the graphs algorithms!

First, like in previous assignments, I use try and catch statements for uploading the magic items file [11-24]. I also use this process for the file that contains the magic items to find [27-39].

Then, I create a new instance of a binary search tree [41] and set the root of the tree [42]. I then use a for loop [43] to go through the magic items to insert them in the tree [44] and create their path [45]. Then, I use a for loop [49] to go through the magic items that will be searched [50] and create their path as well [51]. I also print the averages [47,53], and the in order traversal [54]. The running time for both inserting and finding nodes in the binary tree is  $O(n)$  because the algorithms are made up of if else statements.

Then, I move on to creating the graphs. First, I initialize the array list of vertices [57]. Then, I use try and catch statements again to upload the file and parse it as well. I parse the file using a series of if else statements so that the code knows when to add a vertex [66-69], and when to add an edge [70-81]. It is also parsed so that it knows when to start a new graph [82-96].

While the parsing and formatting are taken care of, I create the instances of my graphs. First, I create the first four graphs [83-95]. Then, outside of the if else statements I create the last graph, since this one starts at 0 [99-111].

The running time for both BFS and DFS is  $O(V + E)$ .  $V$  is vertices and  $E$  is edges. Each vertex is only accounted for once and each edge is accounted for once so that is why adding them together yields the running time.

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Arrays;
4 import java.util.Scanner;
5 import java.util.*;
6
7 public class main {
8     public static void main(String[] args) {
9         // BST code
10        String[] magicItems = new String[666];
11        try { //Trying to find the file
12            File file = new File("magicitems.txt");
13            Scanner sc = new Scanner(file);
14            int index = 0;
15
16            while (sc.hasNextLine()) {
17                String item = sc.nextLine();
18                item = item.toUpperCase();
19                magicItems[index++] = item;
20            }
21        }
22        catch (FileNotFoundException e) { // If we cant find the
23            file
24                e.printStackTrace();
25        }
```

```

25
26     String[] magicItemsFind = new String[42];
27     try { //Trying to find the file
28         File file = new File("magicitems-find-in-bst.txt");
29         Scanner sc = new Scanner(file);
30         int index = 0;
31         while (sc.hasNextLine()) {
32             String item = sc.nextLine();
33             item = item.toUpperCase();
34             magicItemsFind[index ++] = item;
35         }
36     }
37     catch (FileNotFoundException e) { // If we cant find the
file
38         e.printStackTrace();
39     }
40
41     BST tree = new BST();
42     tree.setRoot(magicItems[0]); // We can do this because the
first magic item will always be the root
43     for (int i = 0; i < magicItems.length; i++) {
44         tree.insert(tree.root, magicItems[i]);
45         tree.path(magicItems[i]);
46     }
47     tree.average(tree.totalComparisons, magicItems.length);
48     System.out.println("-----Inserting the
magic items done, now searching for the targets
!-----");
49     for (int i = 0; i < magicItemsFind.length; i++) { //
Finding the target list in the bst
50         tree.find(tree.root, magicItemsFind[i]);
51         tree.findPath(magicItemsFind[i]);
52     }
53     tree.average(tree.findTotalComparisons, magicItemsFind.
length);
54     tree.ITW(tree.root);
55
56     // Graph codes now
57     ArrayList<Vertex> vertices = new ArrayList<Vertex>(); //
vertices that will be provided to the graph
58
59     try { //Trying to find the file
60         File file = new File("graphs1.txt");
61         Scanner sc = new Scanner(file);
62
63         while (sc.hasNextLine()) {
64             String item = sc.nextLine();
65             String[] parse = item.split(" ");
66             if (parse[0].equals("add") && parse[1].equals("
vertex")) {
67                 Vertex v = new Vertex(Integer.parseInt(parse
[2])); // Getting the vertex number and creating a new vertex
based off of it
68                 vertices.add(v); // Adding the vertex to the
array list of all the vertices
69             }
70             else if (parse[0].equals("add") && parse[1].equals(

```

```

71     "edge")) {
72         int firstVertex = Integer.parseInt(parse[2]);
73         // First vertex the edge will be connected to
74         int secondVertex = Integer.parseInt(parse[4]);
75         // Second vertex the edge will be connected to
76         if (firstVertex == 0) {
77             vertices.get(firstVertex).add(vertices.get(
78                 secondVertex)); // Correlating the edges to the vertices array
79             // of all the vertices
80             vertices.get(secondVertex).add(vertices.get(
81                 firstVertex)); // Same as above, but vice versa so they are
82             // connected both ways
83         }
84         else {
85             vertices.get(firstVertex - 1).add(vertices.
86                 get(secondVertex - 1)); // Correlating the edges to the
87             // vertices array of all the vertices
88             vertices.get(secondVertex - 1).add(vertices
89                 .get(firstVertex - 1)); // Same as above, but vice versa so
90             // they are connected both ways
91         }
92     }
93     else if (parse[0].equals("new") && parse[1].equals(
94         "graph")) {
95         Graph graph = new Graph();
96         graph.setGraph(vertices);
97         graph.matrix(); // Matrix
98         graph.adjList(); // Adjacency list
99         graph.printResults(); // Results
100         if (vertices.isEmpty() == false) { //
101             Traversals
102                 System.out.println("
103                 -----Below is the depth first
104                 traversal-----");
105                 graph.DFS(vertices.get(0));
106                 graph.resetTraversal(); // Resetting
107                 System.out.println("
108                 -----Below is the breadth first
109                 traversal-----");
110                 graph.BFS(vertices.get(0));
111             }
112             vertices.clear(); // So that the graphs aren't
113             // added on top of each other
114         }
115         // For the last graph
116         Graph graph = new Graph();
117         graph.setGraph(vertices);
118         graph.matrix();
119         graph.adjList();
120         graph.printResults();
121         if (vertices.isEmpty() == false) {
122             System.out.println("
123             -----Below is the depth first
124             traversal-----");
125             graph.DFS(vertices.get(0));

```



```
107         graph.resetTraversal();
108         System.out.println("
-----Below is the breadth first
traversal-----");
109         graph.BFS(vertices.get(0));
110     }
111     vertices.clear();
112 }
113 catch (FileNotFoundException e) { // If we cant find the
file
114     e.printStackTrace();
115 }
116 }
117 }
```