

# Assignment 2 - Code Evaluation

Mason Nakamura

October 2021

## 1 Node Class

```
1 public class Node {  
2     String name;  
3     Node left = null;  
4     Node right = null;  
5 }
```

The Node class is used only in the Binary\_Search\_Tree class.

## 2 Vertex Class

```
1 import java.util.ArrayList;  
2  
3 public class Vertex {  
4     Vertex next;  
5     int label;  
6     int connecting_vertex;  
7     int origin_vertex;  
8     ArrayList<Vertex> neighbors;  
9 }
```

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

```
1 public class Queue {
2     Vertex head, tail;
3
4     public boolean isEmpty(){
5         return(head == null);
6     }
7
8     public void enqueue(Vertex s){
9         Vertex oldTail = tail;
10        tail = new Vertex();
11        tail = s;
12        tail.next = null;
13        if(isEmpty()){
14            head = tail;
15        }else{
16            oldTail.next = tail;
17        }
18    }
19
20    public Vertex dequeue(){
21        Vertex retval;
22        if(!isEmpty()){
23            retval = head;
24            head = head.next;
25            if(isEmpty()){
26                tail = null;
27            }
28        }else{
29            retval = null;
30        }
31        return retval;
32    }
33
34 }
```

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

## 4 Outputs Class

```
1 import java.util.ArrayList;
2 public class Outputs {
3     /**
4      * This method is used to print all adjacency matrices given the lines in the
5      * file
6      * @param lines an array of every line in the file
7      */
8     public void adjacency_matrix(String[] lines) {
9         // use these indices throughout for loops
10        int index_start;
11        int index_end = 0;
12        // for printing purposes
13        int graph_id = 0;
14        // for vertex starting index
15        boolean indexIs0 = false;
16        while (index_end <= lines.length) {
17            graph_id += 1;
18            index_start = index_end;
19            // don't need rows and columns since adjacency matrix is symmetric ==>
20            rows = columns
21            int rows_columns = 0;
22            // get indices for for loop for edges
23            for (int i = index_start; i < lines.length; i++) {
24                String line = lines[i];
25                index_start += 1;
26                // case for line is blank;
27                if (line.isBlank()) {
28                    // skip the line
29                    continue;
30                }
31                // case for line is a comment
32                if (line.charAt(1) == '-' & line.charAt(0) == '-') {
33                    // skip the line
34                    continue;
35                }
36                // make an array of the words from the line
37                String[] words = line.split(" ");
38                // case for declaring new graph
39                if (words[0].equals("new")) {
40                    // print graph_id
41                    System.out.println("Graph Number: " + graph_id);
42                    //skip the line
43                    continue;
44                }
45                // case for adding vertex
46                if (words[0].equals("add") & words[1].equals("vertex")) {
47                    rows_columns += 1;
48                    if (words[2].equals("0")) {
49                        indexIs0 = true;
50                    }
51                    continue;
52                }
53                //check if hit edge case
54                if (words[0].equals("add") & words[1].equals("edge")) {
55                    index_start -= 1;
56                    break;
57                }
58            }
59
60            // create instance of matrix given number of vertices declared
61            index_end = index_start + 1;
62            int[][] matrix = new int[rows_columns][rows_columns];
63            // use index_start to continue in the for loop
64            for (int i = index_start; i < index_end; i++) {
65                // case if reached to end of file/(array of lines)
66                if (lines.length < index_end) {
67                    break;
68                }
69                String line = lines[i];
70                String[] words = line.split(" ");
71                // check if line is blank; if so, move to next graph
```

```

72         if (line.isBlank()) {
73             break;
74         }
75         // case for adding edge
76         if (words[0].equals("add") & words[1].equals("edge")) {
77             index_end += 1;
78             // use Integer.parseInt to convert string to int
79
80             if (indexIs0) { // if index starts at 0 don't subtract 1
81                 matrix[Integer.parseInt(words[2])][Integer.parseInt(words[4])
82                     ] = 1;
83                 // do it twice since undirected
84                 matrix[Integer.parseInt(words[4])][Integer.parseInt(words[2])
85                     ] = 1;
86             } else { // subtract 1 to keep indices same
87                 matrix[Integer.parseInt(words[2]) - 1][Integer.parseInt(words
88                     [4]) - 1] = 1;
89                 // do it twice since undirected
90                 matrix[Integer.parseInt(words[4]) - 1][Integer.parseInt(words
91                     [2]) - 1] = 1;
92             }
93         }
94     }
95
96     for (int i = 0; i < matrix.length; i++) {
97         for (int j = 0; j < matrix[i].length; j++) {
98             System.out.print(matrix[i][j] + " ");
99         }
100         System.out.println();
101     }
102
103     public void adjacency_list(String[] lines) {
104         // use these indices throughout for loops
105         int index_start;
106         int index_end = 0;
107         // for printing purposes
108         int graph_id = 0;
109         // for vertex starting index
110         boolean indexIs0 = false;
111         while (index_end <= lines.length) {
112             graph_id += 1;
113             index_start = index_end;
114             int adj_list_length = 0;
115             // get indices for for loop for edges
116             for (int i = index_start; i < lines.length; i++) {
117                 String line = lines[i];
118                 index_start += 1;
119                 // case for line is blank;
120                 if (line.isBlank()) {
121                     // skip the line
122                     continue;
123                 }
124                 // case for line is a comment
125                 if (line.charAt(1) == '-' & line.charAt(0) == '-') {
126                     // skip the line
127                     continue;
128                 }
129                 // make an array of the words from the line
130                 String[] words = line.split(" ");
131                 // case for declaring new graph
132                 if (words[0].equals("new")) {
133                     // print graph_id
134                     System.out.println("Graph Number: " + graph_id);
135                     // skip the line
136                     continue;
137                 }
138                 // case for adding vertex
139                 if (words[0].equals("add") & words[1].equals("vertex")) {
140                     adj_list_length += 1;
141                     if (words[2].equals("0")) {
142                         indexIs0 = true;
143                     }
144                     continue;

```

```

144     }
145     //check if hit edge case
146     if (words[0].equals("add") & words[1].equals("edge")) {
147         // subtract an index since we previously added it but didn't need
148         // to since we entered the edge case
149         index_start -= 1;
150         break;
151     }
152 }
153
154 // create instance of array given number of vertices declared
155 index_end = index_start + 1;
156 //make array of vertex objects
157 Vertex[] array = new Vertex[adj_list_length];
158 // check starting index
159
160 // create first vertex objects in array
161 if (indexIs0) {
162     for (int i = 0; i < adj_list_length; i++) {
163         // from Vertex class
164         Vertex vertex = new Vertex();
165         vertex.label = i;
166         vertex.next = null;
167         array[i] = vertex;
168     }
169 } else { // shift index by 1
170     for (int i = 1; i < adj_list_length + 1; i++) {
171         Vertex vertex = new Vertex();
172         vertex.label = i;
173         vertex.next = null;
174         array[i - 1] = vertex;
175     }
176 }
177
178 // use index_start to continue in the for loop
179 for (int i = index_start; i < index_end; i++) {
180     // case if reached to end of file/(array of lines)
181     if (lines.length < index_end) {
182         break;
183     }
184     String line = lines[i];
185     String[] words = line.split(" ");
186     // check if line is blank; if so, move to next graph
187     if (line.isBlank()) {
188         break;
189     }
190
191     // case for adding edge
192     if (words[0].equals("add") & words[1].equals("edge")) {
193         index_end += 1;
194         // you need to define vertex1 and vertex2 since not doing so
195         // creates a pointer infinitely pointing to itself
196         if (indexIs0) { // if index starts at 0 don't subtract 1
197             Vertex vertex1 = new Vertex();
198             // use Integer.parseInt to convert string to int
199             vertex1.origin_vertex = Integer.parseInt(words[2]);
200             vertex1.connecting_vertex = Integer.parseInt(words[4]);
201             vertex1.next = null;
202             Vertex head = array[vertex1.origin_vertex];
203             while (head.next != null) {
204                 head = head.next;
205             }
206             head.next = vertex1;
207
208             Vertex vertex2 = new Vertex();
209             vertex2.origin_vertex = Integer.parseInt(words[2]);
210             vertex2.connecting_vertex = Integer.parseInt(words[4]);
211             vertex2.next = null;
212             // do it twice since undirected
213             Vertex head1 = array[vertex2.connecting_vertex];
214             while (head1.next != null) {
215                 head1 = head1.next;
216             }
217             head1.next = vertex2;
218         } else { // subtract 1 to keep indices same

```

```

218         Vertex vertex1 = new Vertex();
219         vertex1.origin_vertex = Integer.parseInt(words[2]);
220         vertex1.connecting_vertex = Integer.parseInt(words[4]);
221         vertex1.next = null;
222         Vertex head = array[vertex1.origin_vertex - 1];
223         while (head.next != null) {
224             head = head.next;
225         }
226         head.next = vertex1;
227
228         // do it twice since undirected
229         Vertex vertex2 = new Vertex();
230         vertex2.origin_vertex = Integer.parseInt(words[2]);
231         vertex2.connecting_vertex = Integer.parseInt(words[4]);
232         vertex2.next = null;
233         Vertex head1 = array[vertex2.connecting_vertex - 1];
234         while (head1.next != null) {
235             head1 = head1.next;
236         }
237         head1.next = vertex2;
238     }
239 }
240
241 }
242
243
244 for (Vertex vertex : array) {
245     System.out.print("[ " + vertex.label + "]" + " ");
246     if (vertex.next != null) {
247         Vertex temp_vertex = vertex.next;
248         //check if we print connecting_vertex or origin_vertex
249         while (temp_vertex != null) {
250             if (temp_vertex.connecting_vertex != vertex.label) {
251                 System.out.print(temp_vertex.connecting_vertex + " ");
252             }
253             if (temp_vertex.origin_vertex != vertex.label) {
254                 System.out.print(temp_vertex.origin_vertex + " ");
255             }
256             temp_vertex = temp_vertex.next;
257         }
258     }
259     System.out.println();
260 }
261 }
262 }
263
264 public void linked_objects(String[] lines) {
265     // use these indices throughout for loops
266     int index_start;
267     int index_end = 0;
268     // for printing purposes
269     int graph_id = 0;
270     // for vertex starting index
271     boolean indexIs0 = false;
272     // Read the lines up to add edge case
273     while (index_end <= lines.length) {
274         graph_id += 1;
275         index_start = index_end;
276         int adj_list_length = 0;
277         // get indices for for loop for edges
278         for (int i = index_start; i < lines.length; i++) {
279             String line = lines[i];
280             index_start += 1;
281             // case for line is blank;
282             if (line.isBlank()) {
283                 // skip the line
284                 continue;
285             }
286             // case for line is a comment
287             if (line.charAt(1) == '-' & line.charAt(0) == '-') {
288                 // skip the line
289                 continue;
290             }
291             // make an array of the words from the line
292             String[] words = line.split(" ");
293             // case for declaring new graph

```

```

294         if (words[0].equals("new")) {
295             // print graph_id
296             System.out.println("Graph Number: " + graph_id);
297             //skip the line
298             continue;
299         }
300         // case for adding vertex
301         if (words[0].equals("add") & words[1].equals("vertex")) {
302             adj_list_length += 1;
303             if (words[2].equals("0")) {
304                 indexIs0 = true;
305             }
306             continue;
307         }
308         //check if hit edge case
309         if (words[0].equals("add") & words[1].equals("edge")) {
310             // subtract an index since we previously added it but didn't need
311             // to since we entered the edge case
312             index_start -= 1;
313             break;
314         }
315     }
316
317     // create instance of array given number of vertices declared
318     index_end = index_start + 1;
319     //make array of vertex objects
320     Vertex[] array = new Vertex[adj_list_length];
321     // check starting index
322
323     // create first vertex objects in array given index
324     if (indexIs0) {
325         for (int i = 0; i < adj_list_length; i++) {
326             // from Vertex class
327             Vertex vertex = new Vertex();
328             vertex.connecting_vertex = i;
329             vertex.label = i;
330             vertex.neighbors = new ArrayList<Vertex>();
331             array[i] = vertex;
332         }
333     }
334     else { // shift index by 1
335         for (int i = 1; i < adj_list_length + 1; i++) {
336             Vertex vertex = new Vertex();
337             vertex.connecting_vertex = i;
338             vertex.label = i;
339             vertex.neighbors = new ArrayList<Vertex>();
340             array[i - 1] = vertex;
341         }
342     }
343
344     // use index_start to continue in the for loop
345     for (int i = index_start; i < index_end; i++) {
346         // case if reached to end of file/(array of lines)
347         if (lines.length < index_end) {
348             break;
349         }
350         String line = lines[i];
351         String[] words = line.split(" ");
352         // check if line is blank; if so, move to next graph
353         if (line.isBlank()) {
354             break;
355         }
356
357         // case for adding edge
358         if (words[0].equals("add") & words[1].equals("edge")) {
359             index_end += 1;
360             // you need to define vertex1 and vertex2 since not doing so
361             // creates a pointer infinitely pointing to itself
362             if (indexIs0) { // if index starts at 0 don't subtract 1
363                 Vertex vertex1 = new Vertex();
364                 vertex1.origin_vertex = Integer.parseInt(words[2]);
365                 vertex1.connecting_vertex = Integer.parseInt(words[4]);
366                 vertex1.label = vertex1.connecting_vertex;
367                 vertex1.neighbors = array[vertex1.connecting_vertex].
368                     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

```
1      import java.util.ArrayList;
2
3      public class Search {
4          ArrayList<Integer> processed = new ArrayList<>();
5
6          public void depth_first(Vertex v, Vertex[] array) {
7              if (!processed.contains(v.label)) {
8                  System.out.print(v.label + " ");
9                  processed.add(v.label);
10             }
11             for (Vertex neighbor : v.neighbors) {
12                 if (!processed.contains(neighbor.label)) {
13                     depth_first(neighbor, array);
14                 }
15             }
16             for (Vertex vertex: array){
17                 if (!processed.contains(vertex.label)){
18                     if (!processed.contains(vertex.label)) {
19                         System.out.print(vertex.label + " ");
20                         processed.add(vertex.label);
21                     }
22                     for (Vertex neighbor : vertex.neighbors) {
23                         if (!processed.contains(neighbor.label)) {
24                             depth_first(neighbor, array);
25                         }
26                     }
27                 }
28             }
29         }
30
31
32         public void breadth_first (Vertex v, Vertex[] array){
33             Queue queue = new Queue();
34             queue.enqueue(v);
35             processed.add(v.label);
36             while (!queue.isEmpty()) {
37                 Vertex vertex = queue.dequeue();
38                 System.out.print(vertex.label + " ");
39                 for (Vertex neighbor: vertex.neighbors) {
40                     if (!processed.contains(neighbor.label)) {
41                         queue.enqueue(neighbor);
42                         processed.add(neighbor.label);
43                     }
44                 }
45             }
46             for (Vertex vertex: array){
47                 if (!processed.contains(vertex.label)){
48                     Queue queue1 = new Queue();
49                     queue1.enqueue(vertex);
50                     processed.add(vertex.label);
51                     while (!queue1.isEmpty()) {
52                         Vertex vertex1 = queue1.dequeue();
53                         System.out.print(vertex1.label + " ");
54                         for (Vertex neighbor: vertex1.neighbors) {
55                             if (!processed.contains(neighbor.label)) {
56                                 queue1.enqueue(neighbor);
57                                 processed.add(neighbor.label);
58                             }
59                         }
60                     }
61                 }
62             }
63         }
64     }
65 }
```

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 first 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

```
1 public class Binary_Search_Tree {
2     int i = 0;
3     int comparisons;
4     Node first_root;
5     public Node populateBST(Node root, String word){
6         // if there is no root to begin with or it hits a terminal node
7         if (root == null){
8             Node node = new Node();
9             node.name = word;
10            return node;
11        }
12        // go down the left side if word <= root.name
13        else if (word.compareTo(root.name) >= 0){
14            root.left = populateBST(root.left, word);
15            System.out.print(" " + "L");
16        }
17        else{ // go down right side if word > root.name
18            root.right = populateBST(root.right, word);
19            System.out.print(" " + "R");
20        }
21        return root;
22    }
23
24    public Node makeBST(String[] words)
25    {
26        Node root = null;
27        boolean gotroot = false;
28        // go through all words and populate BST using method populateBST
29        for (String word: words) {
30            if (!gotroot){
31                gotroot = true;
32                Node temp_first_root = new Node();
33                temp_first_root.name = word;
34                first_root = temp_first_root;
35            }
36            System.out.println(word + ": ");
37            root = populateBST(root, word);
38            System.out.println();
39        }
40        // if there are no words
41        return root;
42    }
43
44    public void inorder(Node root)
45    {
46        if (root == null) {
47            return;
48        }
49
50        // go to the left child
51        inorder(root.left);
52        // print the current root name
53        System.out.println(root.name);
54        // go to the right child
55        inorder(root.right);
56    }
57
58    public Node search(Node root, String target){
59        if (root.name.equals(target) | root == null){
60            comparisons += 1;
61            return root;
62        }
63
64        if (target.compareTo(root.name) >= 0){
65            comparisons += 1;
66            System.out.print(" " + "L");
67            return search(root.left, target);
68        }
69    }
70 }
```

```

71     else{
72         comparisons += 1;
73         System.out.print(" " + "R");
74         return search(root.right, target);
75     }
76 }
77 }
78 }

```

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

```

1  import java.io.*;
2  import java.util.Arrays;
3  import java.util.Random;
4
5  public class Test {
6
7
8
9      public static void main(String[] args) {
10         String[] lines = {};
11         // Read line by line the txt file using File reader
12         String fileName = "Assignment 4/graphs1.txt"; //REMEMBER TO NOT HARDCODE
13         File file = new File(fileName);
14         try {
15             FileReader fr = new FileReader(file);
16             BufferedReader br = new BufferedReader(fr);
17             String line;
18             while ((line = br.readLine()) != null) {
19                 // add strings from txt file line by line into array words
20                 lines = Arrays.copyOf(lines, lines.length + 1); //extends memory
21                 lines[lines.length - 1] = line; //adds line to extra memory
22             }
23         } catch (FileNotFoundException e) {
24             System.out.println("An error occurred.");
25             e.printStackTrace();
26         } catch (IOException e) {
27             e.printStackTrace();
28         }
29
30         String[] copy_graphs0 = lines.clone();
31
32         Outputs out = new Outputs();
33         // outputs matrices
34         System.out.println("ADJACENCY MATRICES: ");
35         out.adjacency_matrix(copy_graphs0);
36         // outputs adjacency lists
37         System.out.println("ADJACENCY LISTS: ");
38         out.adjacency_list(copy_graphs0);
39         System.out.println("LINKED OBJECTS: ");
40         out.linked_objects(copy_graphs0);
41
42
43         // Binary Search Tree population and traversal
44         String[] lines1 = {};
45         // Read line by line the txt file using File reader
46         String fileName1 = "Assignment 4/magicitems.txt"; //REMEMBER TO NOT HARDCODE
47         File file1 = new File(fileName1);
48         try {
49             FileReader fr = new FileReader(file1);
50             BufferedReader br = new BufferedReader(fr);
51             String line;
52             while ((line = br.readLine()) != null) {
53                 // add strings from txt file line by line into array words

```

```

54         lines1 = Arrays.copyOf(lines1, lines1.length + 1); //extends memory
55         lines1[lines1.length - 1] = line; //adds line to extra memory
56     }
57 } catch (FileNotFoundException e) {
58     System.out.println("An error occurred.");
59     e.printStackTrace();
60 } catch (IOException e) {
61     e.printStackTrace();
62 }
63
64 String[] lines2 = {};
65 // Read line by line the txt file using File reader
66 String fileName2 = "Assignment 4/magicitems-find-in-bst.txt"; //REMEMBER TO
    NOT HARDCODE
67 File file2 = new File(fileName2);
68 try {
69     FileReader fr = new FileReader(file2);
70     BufferedReader br = new BufferedReader(fr);
71     String line;
72     while ((line = br.readLine()) != null) {
73         // add strings from txt file line by line into array words
74         lines2 = Arrays.copyOf(lines2, lines2.length + 1); //extends memory
75         lines2[lines2.length - 1] = line; //adds line to extra memory
76     }
77 } catch (FileNotFoundException e) {
78     System.out.println("An error occurred.");
79     e.printStackTrace();
80 } catch (IOException e) {
81     e.printStackTrace();
82 }
83 final int FILELENGTH = lines2.length;
84
85 // copy_words2 -> magicitems-find-in-bst.txt
86 // copy_words1 -> magicitems.txt
87 String[] copy_words2 = lines2.clone();
88 String[] copy_words1 = lines1.clone();
89
90 Binary_Search_Tree bst = new Binary_Search_Tree();
91 System.out.println("BINARY SEARCH TREE POPULATING: ");
92 Node root = bst.makeBST(copy_words1);
93 System.out.println("BINARY SEARCH TREE IN-ORDER TRAVERSAL: ");
94 bst.inorder(root);
95 System.out.println("BINARY SEARCH TREE SEARCHING FOR ITEMS W/ COMPARISONS: ")
    ;
96 float total_comparisons = 0;
97 for (String i: copy_words2){
98     System.out.print(i + " ");
99     bst.search(root, i);
100    System.out.println();
101    System.out.println("Comparisons: " + bst.comparisons);
102    total_comparisons += bst.comparisons;
103    bst.comparisons = 0;
104 }
105 System.out.println("AVERAGE NUMBER OF COMPARISONS: " + total_comparisons/
    FILELENGTH);
106 }
107 }

```

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.