

Assignment Four

Shannon Cordoni

Shannon.Cordoni@Marist.edu

November 20, 2021

1 PROBLEM ONE: GRAPHS AND DATA TREES

1.1 THE DATA STRUCTURE

Given multiple text files we were to create different graph representations and a binary search tree. First taking the list of strings our job was to create an algorithm to go through this list and add them to a binary search tree to then later traverse through to search for 42 items. Then taking the list of graph instructions we were to create vertices and edges to form an undirected graph in the form of a matrix, adjacency list, and linked objects.

1.2 MAIN CLASS

1.2.1 DESCRIPTION

This class is where most of our work is done, it contains multiple scanners to read in our multiple files. These files include the *magicitems.txt* file, the *magicitems - find - in - bst.txt* file, and *graphs1.txt* file. Using these files we input each line of the *graphs1.txt* into an instruction array to be split up into a 2 dimensional array to access each index. We then use this split array to create the matrix, adjacency list, and linked object representation for each graph. The *magicitems.txt* file was also read into an array to be passed to the *insertTree* method to put each line into the binary search tree, this tree could then be traversed through to search for specific items. The methods contained in this class preform all these operations and more.

```
1  /*
2  *
3  *
4  * Assignment 4
5  * Due Date and Time: 11/19/21 before 12:00am
6  * Purpose: to implement graph and tree data structures, and to understand the performance of
7  *their traversals.
8  * Input: The user will be inputting a file containing a list of edges and vertices.
9  * Output: The program will output graph and tree data structures.
10 * @author Shannon Cordoni
11 *
```

```

12  */
13
14  import java.io.File;
15  import java.io.FileNotFoundException;
16  import java.util.ArrayList;
17  import java.util.Arrays;
18  import java.util.*;
19  import java.util.Scanner;
20
21
22
23  public class Assignment4Cordoni {
24
25      //Declare keyboard
26      static Scanner keyboard = new Scanner(System.in);
27
28      public static void main(String[] args) {
29
30          //Declare and initialize variables
31          String line;
32
33          //For binary search tree
34          String[] wordarray = new String[666];
35
36          String[] instructionarray = new String[375];
37          String[] graph1array = new String[20];
38          String[] graph2array = new String[38];
39          String[] graph3array = new String[127];
40          String[] graph4array = new String[138];
41          String[] graph5array = new String[48];
42
43          String[] searcharray = new String[42];
44          String[][] splitinstructionarray = new String[375][8];
45          String[][] graph1splitarray = new String[20][8];
46          String[][] graph2splitarray = new String[38][8];
47          String[][] graph3splitarray = new String[127][8];
48          String[][] graph4splitarray = new String[138][8];
49          String[][] graph5splitarray = new String[48][8];
50          TreeCordoni tree = new TreeCordoni();
51
52          int numberOfLookupComparisons = 0;
53          int averagenumberOfLookupComparisons = 0;
54
55
56          //Reads in the magic items file for the binary search tree
57          //create new file object
58          File myFile = new File("magicitems.txt");
59
60          try
61          {
62              //create scanner
63              Scanner input = new Scanner(myFile);
64              line = null;
65
66              int i = 0;
67
68              //while there are more lines in the file it inputs them into a word array
69              while(input.hasNext())
70              {
71                  //Input into array
72                  wordarray[i] = input.nextLine();
73                  i++;
74              }//while
75
76              input.close();

```

```

77
78     }//try
79
80     //error for file not found
81     catch(FileNotFoundException ex)
82     {
83         System.out.println("Failed to find file: " + myFile.getAbsolutePath());
84     }//catch
85
86     //Error in case of a null pointer exception
87     catch(NullPointerException ex)
88     {
89         System.out.println("Null pointer exception.");
90         System.out.println(ex.getMessage());
91     }//catch
92
93     //General error message
94     catch(Exception ex)
95     {
96         System.out.println("Something went wrong");
97         ex.printStackTrace();
98     }//catch
99
100    //Reads in the graph file to create graphs
101    //create new file object
102    File myFile1 = new File("graphs1.txt");
103
104    try
105    {
106        //create scanner
107        Scanner input = new Scanner(myFile1);
108        line = null;
109
110        int i = 0;
111
112        while(input.hasNext()){
113
114            instructionarray[i] = input.nextLine();
115            i++;
116        }
117
118
119        input.close();
120
121    }//try
122
123    //error for file not found
124    catch(FileNotFoundException ex)
125    {
126        System.out.println("Failed to find file: " + myFile.getAbsolutePath());
127    }//catch
128
129    //Error in case of a null pointer exception
130    catch(NullPointerException ex)
131    {
132        System.out.println("Null pointer exception.");
133        System.out.println(ex.getMessage());
134    }//catch
135
136    //General error message
137    catch(Exception ex)
138    {
139        System.out.println("Something went wrong");
140        ex.printStackTrace();
141    }//catch

```

```

142
143
144 //Reads in the magic items to find in the binary search tree
145 //create new file object
146 File myFile2 = new File("magicitems-find-in-bst.txt");
147
148 try
149 {
150     //create scanner
151     Scanner input = new Scanner(myFile2);
152     line = null;
153
154     int i = 0;
155
156     //while there are more lines it inputs them into an instruction array
157     while(input.hasNext())
158     {
159         //Input into array
160         searcharray[i] = input.nextLine();
161         i++;
162     }//while
163
164     input.close();
165
166 }//try
167
168 //error for file not found
169 catch(FileNotFoundException ex)
170 {
171     System.out.println("Failed to find file: " + myFile.getAbsolutePath());
172 }//catch
173
174 //Error in case of a null pointer exception
175 catch(NullPointerException ex)
176 {
177     System.out.println("Null pointer exception.");
178     System.out.println(ex.getMessage());
179 }//catch
180
181 //General error message
182 catch(Exception ex)
183 {
184     System.out.println("Something went wrong");
185     ex.printStackTrace();
186 }//catch
187
188 // i know this is not exactly how I was supposed to split up the text
189 //file, but this was the only way I could get it to work with everything
190
191 //Print to check array
192 for (int i = 0; i < instructionarray.length; i++){
193     // System.out.println(instructionarray[i]);
194 }//for
195
196 //Put instructions into graph 1 array
197 for (int i = 0; i < graph1array.length; i++){
198     graph1array[i] = instructionarray[i];
199 }//for
200
201 //Put instructions into graph 2 array
202 for (int i = 0; i < graph2array.length; i++){
203     graph2array[i] = instructionarray[i + 22];
204 }//for
205
206 //Put instructions into graph 3 array

```

```

207     for (int i = 0; i < graph3array.length; i++){
208         graph3array[i] = instructionarray[i + 61];
209     }//for
210
211     //Put instructions into graph 4 array
212     for (int i = 0; i < graph4array.length; i++){
213         graph4array[i] = instructionarray[i + 189];
214     }//for
215
216     //Put instructions into graph 5 array
217     for (int i = 0; i < graph5array.length; i++){
218         graph5array[i] = instructionarray[i + 327];
219     }//for
220
221     //split up into 2D array
222     for (int i = 0; i < splitinstructionarray.length; i++){
223         for( int j = 0; j < splitinstructionarray[i].length; j++){
224             splitinstructionarray[i] = instructionarray[i].split(" ");
225         }//for j
226     }//for
227
228     //split up into 2D array for graph 1
229     for (int i = 0; i < graph1splitarray.length; i++){
230         for( int j = 0; j < graph1splitarray[i].length; j++){
231             graph1splitarray[i] = graph1array[i].split(" ");
232         }//for j
233     }//for
234
235     //split up into 2D array for graph 2
236     for (int i = 0; i < graph2splitarray.length; i++){
237         for( int j = 0; j < graph2splitarray[i].length; j++){
238             graph2splitarray[i] = graph2array[i].split(" ");
239         }//for j
240     }//for
241
242     //split up into 2D array for graph 3
243     for (int i = 0; i < graph3splitarray.length; i++){
244         for( int j = 0; j < graph3splitarray[i].length; j++){
245             graph3splitarray[i] = graph3array[i].split(" ");
246         }//for j
247     }//for
248
249     //split up into 2D array for graph 4
250     for (int i = 0; i < graph4splitarray.length; i++){
251         for( int j = 0; j < graph4splitarray[i].length; j++){
252             graph4splitarray[i] = graph4array[i].split(" ");
253         }//for j
254     }//for
255
256     //split up into 2D array for graph 5
257     for (int i = 0; i < graph5splitarray.length; i++){
258         for( int j = 0; j < graph5splitarray[i].length; j++){
259             graph5splitarray[i] = graph5array[i].split(" ");
260         }//for j
261     }//for
262
263     //Graphs!!
264
265     System.out.println("Graph 1");
266
267     //making the matrix
268     makeMatrix(graph1splitarray);
269
270     //make the adjacency list
271     makeAdjacencyList(graph1splitarray);

```

```

272
273 //make linked list
274 makeLinkedObjects(graph1splitarray);
275
276 //breadth first traversal
277 breadthTraversal(makeLinkedObjects(graph1splitarray));
278
279 //reset process status
280 reboot(makeLinkedObjects(graph1splitarray));
281
282 System.out.println(" ");
283 System.out.println("Depth Traversal");
284
285 //depth first traversal
286 depthTraversal(makeLinkedObjects(graph1splitarray));
287
288
289 System.out.println(" ");
290 System.out.println("Graph 2");
291
292 //making the matrix
293 makeMatrix(graph2splitarray);
294
295 //make the adjacency list
296 makeAdjacencyList(graph2splitarray);
297
298 //make linked list
299 makeLinkedObjects(graph2splitarray);
300
301 //breadth first traversal
302 breadthTraversal(makeLinkedObjects(graph2splitarray));
303
304 //reset process status
305 reboot(makeLinkedObjects(graph2splitarray));
306
307 System.out.println(" ");
308 System.out.println("Depth Traversal");
309
310 //depth first traversal
311 depthTraversal(makeLinkedObjects(graph2splitarray));
312
313
314 System.out.println(" ");
315 System.out.println("Graph 3");
316
317 //making the matrix
318 makeMatrix(graph3splitarray);
319
320 //make the adjacency list
321 makeAdjacencyList(graph3splitarray);
322
323 //make linked list
324 makeLinkedObjects(graph3splitarray);
325
326 //breadth first traversal
327 breadthTraversal(makeLinkedObjects(graph3splitarray));
328
329 //reset process status
330 reboot(makeLinkedObjects(graph3splitarray));
331
332 System.out.println(" ");
333 System.out.println("Depth Traversal");
334
335 //depth first traversal
336 depthTraversal(makeLinkedObjects(graph3splitarray));

```

```

337
338
339 System.out.println(" ");
340 System.out.println("Graph 4");
341
342 //making the matrix
343 makeMatrix(graph4splitarray);
344
345 //make the adjacency list
346 makeAdjacencyList(graph4splitarray);
347
348 //make linked list
349 makeLinkedObjects(graph4splitarray);
350
351 //breadth first traversal
352 breadthTraversal(makeLinkedObjects(graph4splitarray));
353
354 //reset process status
355 reboot(makeLinkedObjects(graph4splitarray));
356
357 System.out.println(" ");
358 System.out.println("Depth Traversal");
359
360 //depth first traversal
361 depthTraversal(makeLinkedObjects(graph4splitarray));
362
363
364 System.out.println(" ");
365 System.out.println("Graph 5");
366
367 //making the matrix
368 makeMatrix(graph5splitarray);
369
370 //make the adjacency list
371 makeAdjacencyList(graph5splitarray);
372
373 //make linked list
374 makeLinkedObjects(graph5splitarray);
375
376 //breadth first traversal
377 breadthTraversal(makeLinkedObjects(graph5splitarray));
378
379 //reset process status
380 reboot(makeLinkedObjects(graph5splitarray));
381
382 System.out.println(" ");
383 System.out.println("Depth Traversal");
384
385 //depth first traversal
386 depthTraversal(makeLinkedObjects(graph5splitarray));
387
388
389
390 //Binary Search Trees!!
391
392 //I know this does not print out the binary search tree correctly but if
393 // you uncomment line 854 it looks as though different numbers should be returned
394 //for the comparison.
395
396 System.out.println(" ");
397 System.out.println(" ");
398 System.out.println(" Insert the Magic Items into the Tree ");
399
400
401 //insert the word array into the tree

```

```

402     for (int i = 0; i < wordarray.length; i++){
403         insertTree(tree, wordarray[i]);
404     }//for
405
406     System.out.println(" ");
407     System.out.println(" ");
408     System.out.println(" Search for the Magic Items in the Tree ");
409
410     int comparisons = 0;
411
412     //Search for the 42 magic items
413     for (int i = 0; i < searcharray.length; i++){
414
415         System.out.println("Number of Comparisons: " + searchTree(tree.getRoot(),
416             searcharray[i], comparisons));
417         numberOfLookupComparisons = numberOfLookupComparisons +
418             searchTree(tree.getRoot(), searcharray[i], comparisons);
419
420     }//for
421
422     //get the average lookup comparisons
423     averagenumberOfLookupComparisons = numberOfLookupComparisons/searcharray.length;
424
425     System.out.println("Average lookup: " + averagenumberOfLookupComparisons);
426
427 }//main
428
429 //This method creates the matrix of the undirected graph
430 public static void makeMatrix(String[][] instructions) {
431
432     //instantiate matrix
433     int length = 1;
434     int height = 1;
435
436     for (int i = 0; i < instructions.length; i++){
437
438         for(int j = 0; j < instructions[i].length; j++){
439
440             //skip comment line
441             if(instructions[i][j].compareToIgnoreCase("--")==0){
442                 System.out.println(" ");
443             }//if
444
445             //increment length and heigh to get matrix dimensions
446             else if (instructions[i][j].compareToIgnoreCase("vertex")==0){
447                 length++;
448                 height++;
449
450             }//else if
451
452         }//for j
453
454     }//for i
455
456     //create matrix
457     int[][] matrix = new int[length][height];
458
459     //System.out.println("length: " + length);
460     //System.out.println("height: " + height);
461
462     //loop through to add value at correct matrix location
463     for (int i = 0; i < instructions.length; i++){
464
465         for(int j = 0; j < instructions[i].length; j++){

```



```

467         if (instructions[i][j].compareToIgnoreCase("edge")==0){
468
469             //grab index 2 make it length and grab index 4 and make it height
470             //System.out.println("index 2: " + Integer.valueOf(instructions[i][j+1]));
471
472             //grab index 4 make it length and grab index 2 and make it height
473             //System.out.println("index 4: " + Integer.valueOf(instructions[i][j+3]));
474
475             matrix[Integer.valueOf(instructions[i][j + 1])][
476                 Integer.valueOf(instructions[i][j + 3])] = 1;
477
478             matrix[Integer.valueOf(instructions[i][j + 3])][
479                 Integer.valueOf(instructions[i][j + 1])] = 1;
480
481             }//if
482
483         }
484     }
485
486     //print out the matrix
487     for (int i = 0; i < matrix.length; i++) {
488         for (int j = 0; j < matrix[i].length; j++) {
489
490             System.out.print(matrix[i][j] + " ");
491
492             }//for j
493
494             System.out.println();
495
496         }//for i
497     }//make Matrix
498
499     //This method creates the adjacency list of the undirected graph
500     public static void makeAdjacencyList(String[][] instructions) {
501
502         System.out.println(" ");
503
504         int height = 1;
505
506         for (int i = 0; i < instructions.length; i++){
507
508             for(int j = 0; j < instructions[i].length; j++){
509
510                 //skip comment line
511                 if(instructions[i][j].compareToIgnoreCase("--")==0){
512                     System.out.println(" ");
513                 }//if
514
515                 //increment height to create arraylist
516                 else if (instructions[i][j].compareToIgnoreCase("vertex")==0){
517                     height++;
518                 }//else
519
520             }//for j
521         }//for i
522
523         //create arraylist
524         ArrayList<ArrayList<Integer>> adjlist = new ArrayList<>(height);
525
526         //add arraylist at each index
527         for(int i=0; i < height; i++) {
528             adjlist.add(new ArrayList());
529         }//for
530
531         //add neighbors to arraylist

```

```

532     for (int i = 0; i < instructions.length; i++){
533
534         for(int j = 0; j < instructions[i].length; j++){
535
536             if (instructions[i][j].compareToIgnoreCase("edge")==0){
537
538                 //grab index 2 and add 4 to arraylist
539                 //System.out.println(instructions[i][j + 1]);
540
541                 adjlist.get(Integer.parseInt(instructions[i][j + 1])).add
542                 (Integer.parseInt(instructions[i][j + 3]));
543
544                 //grab index 4 and add 2 to arraylist
545                 //System.out.println(instructions[i][j + 3]);
546
547                 adjlist.get(Integer.parseInt(instructions[i][j + 3])).add
548                 (Integer.parseInt(instructions[i][j + 1]));
549
550             }//else
551
552         }//for j
553     }//for i
554
555     //print out arraylist
556     for (int i = 0; i < instructions.length; i++){
557
558         for(int j = 0; j < instructions[i].length; j++){
559
560             if (instructions[i][j].compareToIgnoreCase("vertex")==0){
561
562                 System.out.println "[" + instructions[i][j + 1] + "]" +
563                 adjlist.get(Integer.parseInt(instructions[i][j + 1]));
564
565             }//else
566
567         }//for j
568     }//for i
569
570     //make adjacency list
571
572     //This method creates the linked objects of the undirected graph
573     public static VertexCordoni makeLinkedObjects(String[][] instructions) {
574
575
576         int index = 0;
577         VertexCordoni[] vertexlist;
578
579         //increment index to create vertex array
580         for (int i = 0; i < instructions.length; i++){
581
582             if (instructions[i][0].compareToIgnoreCase("add")==0){
583
584                 if(instructions[i][1].compareToIgnoreCase("vertex")==0){
585
586                     //System.out.println("Id: " + instructions[i][2]);
587                     //System.out.println("Id get: " + vertex.getId());
588                     index++;
589                 }//if
590
591             }//else if
592
593         }//for i
594
595         //create vertex array
596         vertexlist = new VertexCordoni[index];

```

```

597
598     int j = 0;
599
600     //create neighbor array
601     for (int i = 0; i < instructions.length; i++){
602
603         if (instructions[i][0].compareToIgnoreCase("add")==0){
604
605             //create new vertex and set id to add to vertex array
606             if(instructions[i][1].compareToIgnoreCase("vertex")==0){
607
608                 VertexCordoni vertex = new VertexCordoni();
609
610                 vertex.setId(instructions[i][2]);
611
612                 vertexlist[j] = vertex;
613                 //System.out.println(vertexlist[j]);
614                 j++;
615
616             }
617
618             //add edge to neighbor array
619             else if(instructions[i][1].compareToIgnoreCase("edge")==0){
620
621                 for(int k = 0; k < vertexlist.length; k++){
622                     // System.out.println(vertexlist[k].getId());
623
624                     //if the vertex is in the vertex array then add new edge
625                     if(vertexlist[k].getId().compareToIgnoreCase(instructions[i][2])==0){
626                         //System.out.println("hello1");
627
628                         for(int l = 0; l < vertexlist.length; l++){
629                             //System.out.println("hello2");
630
631                             //if the 2nd vertex is in the vertex array then add the first
632                             //vertex to their neighbor array
633                             if (vertexlist[l].getId().compareToIgnoreCase
634                                 (instructions[i][4])==0){
635
636                                 vertexlist[k].neighbors.add(vertexlist[l]);
637                                 //System.out.println("k" + vertexlist[k].getId());
638
639                                 vertexlist[l].neighbors.add(vertexlist[k]);
640                                 //System.out.println("l" +vertexlist[l].getId());
641
642                             }
643                         }
644                     }
645                 }
646             }
647         }
648     }
649
650     //print neighbor array size to check
651     for(int i = 0; i < vertexlist.length; i++){
652         //System.out.println("size " + vertexlist[i].neighbors.size());
653     }
654
655 }
656
657
658
659
660
661

```

```

662         //return
663         return vertexlist[0];
664
665     }//make linked objects
666
667
668
669     //Searching far and wide!
670
671     //this method preforms the breadth traversal
672     public static void breadthTraversal( VertexCordoni vertex) {
673
674         System.out.println(" ");
675         System.out.println("Breadth Traversal");
676
677         QueueVertexCordoni thequeue = new QueueVertexCordoni();
678
679         VertexCordoni currentvertex;
680
681         thequeue.enqueue(vertex);
682
683         vertex.setProcessStatus(true);
684
685         while(!(thequeue.isEmpty())){
686
687             currentvertex = thequeue.dequeue();
688
689             System.out.println("Id " + currentvertex.getId());
690
691
692             for(int i = 0 ; i < currentvertex.neighbors.size() ; i++){
693
694                 if ( currentvertex.neighbors.get(i).getProcessStatus() == false){
695
696                     thequeue.enqueue(currentvertex.neighbors.get(i));
697                     currentvertex.neighbors.get(i).setProcessStatus(true);
698
699                 }//if
700             }//for
701
702
703
704         }//while
705
706     }//breadth Traversal
707
708     //this method resets the process status for depth traversal
709     public static void reboot( VertexCordoni vertex) {
710
711         System.out.println(" ");
712         System.out.println("Reset Processed Status for Depth Traversal");
713
714         QueueVertexCordoni thequeue = new QueueVertexCordoni();
715
716         VertexCordoni currentvertex;
717
718         thequeue.enqueue(vertex);
719
720         vertex.setProcessStatus(false);
721
722         while(!(thequeue.isEmpty())){
723
724             currentvertex = thequeue.dequeue();
725
726

```

```

727         for(int i = 0 ; i < currentvertex.neighbors.size() ; i++){
728
729             if ( currentvertex.neighbors.get(i).getProcessStatus() == true){
730
731                 thequeue.enqueue(currentvertex.neighbors.get(i));
732                 currentvertex.neighbors.get(i).setProcessStatus(false);
733
734             }//if
735         }//for
736
737
738     }//while
739
740 }//reboot
741
742 //this method preforms the depth traversal
743 public static void depthTraversal(VertexCordoni vertex) {
744
745     if((vertex.getProcessStatus() == false)){
746
747         System.out.println("Id: " + vertex.getId());
748         vertex.setProcessStatus(true);
749
750     }//if
751
752     //System.out.println("Size: " + vertex.neighbors.size());
753
754     for(int i = 0; i < vertex.neighbors.size(); i++){
755
756         if(vertex.neighbors.get(i).getProcessStatus() == false){
757             depthTraversal(vertex.neighbors.get(i));
758         }//if
759
760     }//for
761
762 }//depth Traversal
763
764
765
766 //lets make the trees!
767
768 //This method inserts the nodes into the tres
769 public static void insertTree(TreeCordoni tree, String word) {
770
771     TreeCordoni newnode = new TreeCordoni();
772
773     newnode.setData(word);
774
775     TreeCordoni trailing = null;
776
777     //sets current to the tree root
778     TreeCordoni current = tree.getRoot();
779
780     //while the root is not null continue down the tree
781     while (current != null){
782
783         trailing = current;
784
785         if(newnode.getData().compareToIgnoreCase(current.getData()) < 0){
786
787             current = current.getLeft();
788             //System.out.println("L ");
789
790         }//if
791

```

```

792         else{
793             current = current.getRight();
794             //System.out.println("R ");
795         }//else
796     }//while
797
800     newnode.setParent(trailing);
801
802     //if trailing is null then set the new node to the root
803     if(trailing == null){
804         tree.setRoot(newnode);
805         System.out.println("Root: " + newnode.getData());
806     }//if
807
808     //else we set the new node in the tree
809     else{
810         if(newnode.getData().compareToIgnoreCase(trailing.getData()) < 0){
811             trailing.setLeft(newnode);
812             System.out.println("L ");
813
814             //to print!
815             printTree(tree.getRoot());
816         }//if
817
818         else{
819             trailing.setRight(newnode);
820             System.out.println("R ");
821
822             //to print!
823             printTree(tree.getRoot());
824         }//else
825     }//else
826
827     }//insertTree
828
829     //This method prints the tree (kind of)
830     public static void printTree(TreeCordoni root) {
831         if (root != null){
832             root.setRoot(root.getLeft());
833             printTree(root.getRoot());
834
835             System.out.println(root.getData() + " ");
836
837             root.setRoot(root.getRight());
838             printTree(root.getRoot());
839         }//id
840     }//print tree
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856

```

```

857 //This method searches the tree for the 42 items
858 public static int searchTree(TreeCordoni root, String target, int comparisons) {
859
860     //if the root is null or equal to the target then return
861     if((root == null) || (root.getData().compareToIgnoreCase(target)==0)){
862         comparisons++;
863     }//if
864
865     //else we continue down the tree recursively to find the target
866     else{
867
868         if(target.compareToIgnoreCase(root.getData()) < 0){
869
870             comparisons++;
871             System.out.println("L");
872             searchTree(root.getLeft(), target, comparisons);
873
874         }//if
875
876         else{
877             comparisons++;
878             System.out.println("R");
879             searchTree(root.getRight(), target, comparisons);
880
881         }//else
882     }//else
883
884     //if you uncomment this line it does show different numbers for compare
885     //but i am not sure why it is not returning them correctly
886     //System.out.println(comparisons);
887     return comparisons;
888
889 }//searchTree
890
891 }//Assignment4Cordoni
892

```

1.2.2 DESCRIPTION OF MAIN CODE

The main class above consists of different methods to help create graphs and their representations, along with binary search trees. The good parts of the code first include the file sections. While reading the different *txt* files we input each line into arrays for each file. For the Graph representations we create different arrays for each graph so that we can more easily create the graph representations. Along with passing each line of the word array from the *magicitems.txt* file into the binary search tree to be added. Then to keep everything out of the main method, different methods were used to help organize the code better. These methods include the *makeMatrix*, *makeAdjacencyList*, *makeLinkedObjects*, *breadthTraversal*, *reboot*, *depthTraversal*, *insertTree*, *printTree*, and *searchTree* method.

The *makeMatrix* method takes in a 2 dimension instruction array, and goes through the array line by line. First we initialize a height and length for the matrix to be set later once we find all the vertices for the graph. Then we go through the instruction array, if the line starts with a "-" we know to skip that line because it is a comment. If the line contains the word "vertex" we then know that we have to increment the height and length by one because we are adding a new vertex to the graph. If the line then contains the word "edge" we then know to grab the 2nd and 4th indexes of each line so that we can input a "1" at that height/length in the matrix.

The *makeAdjacencyList* method takes in a 2 dimension instruction array, and goes through the array line by line. First we initialize a height to be set later once we find all the vertices for the graph. Then we go through the instruction array, if the line starts with a "-" we know to skip that line because it is a comment. If the line contains the word "vertex" we then know that we have to increment the height by one because we are adding a new vertex to the graph. Then we initialize a 2 dimensional array list, of the size of our variable height that we created before, to store the adjacency list in. If the line then contains the word "edge" we then know to grab the 2nd and 4th indexes of each line so that we can input them into the adjacency list. First we take the 2nd index as the "from" vertex and find that in the height of our adjacency list. Then we take index 4 as our "to" vertex and add it to the array list at the index of the "from" vertex and vise versa.

The *makeLinkedObjects* method takes in a 2 dimension instruction array, and goes through the array line by line. First we initialize a index to be set later once we find all the vertices for the graph and an array to hold all the vertices. Then we go through the instruction array, if the line starts with a "-" we know to skip that line because it is a comment. If the line contains the word "vertex" we then know that we have to increment the index by one because we are adding a new vertex to the graph. Then we initialize an array, of size index. We then go through the array again, and if the line contains the word "vertex", we then grab the vertex id from the instruction array and add it to the array of vertices. Then to add the edge we loop through the vertex array until we find the vertex that matches our "from" vertex, and then we loop through the vertex array again until we find our "to" vertex, and then we add each of them to their opposing neighbor array. The "from" vertex would have the "to" vertex added to its neighbor array, and the "to" vertex would have the "from" vertex added to its neighbor array.

The *breadthTraversal* method takes in a starting vertex of the graph and adds it to a queue, then it sets its processed status to true. Using a *while* loop, while the queue is not empty, we dequeue the queue and set this new vertex to the current vertex, and then we loop through its neighbor array adding each vertex to the queue, and setting their processed status to true. Upon dequeue-ing the queue we print out the id of the current vertex to show where we are in our traversal.

The *reboot* method takes in a 2 dimension instruction array, and follows the same path as breadth traversal except where breadth traversal sets the processed status to true, here we set it to false so that we can undo the breadth traversal and implement the depth traversal.

The *depthTraversal* method takes in a starting vertex and if it has not been processed we print out its Id and set its processed status to true. We then loop through the neighbor array and recursively call depth traversal to process each vertex in the graph.

The *insertTree* method takes in a tree, and a string containing a line from the word array. We then initialize a new tree node, a trailing tree node, and a current tree node. We set the string to be a new node in the tree, and we set current to equal the root of the tree passed in. While current is not null, we traverse through the tree getting either the next left or right node depending on where the new node falls. If current is null then we move on and check trailing to see if it is null, if it is then we set the new node to be the root of the tree. If trailing is not null then we move on and see if the new node comes before or after the trailing node in the alphabet. If it comes before then we set the node left of trailing to the new node, if it comes before we set the node right of trailing to the new node. Upon completion we print out the binary search tree to see our progress in creating the tree.

The *printTree* method recursively calls itself to print its root data going down the left nodes of the binary search tree, upon completion it prints the root node of the entire tree, and then it goes down the right nodes of the binary search tree printing out the node data along the way.

The *searchTree* method takes in the root of the tree, the target string to look for and the number of comparisons to keep track of. We then look to see if the root is null or if the root of the tree is equal to the target and no searching has to be done. Otherwise we check to see if the target comes before or after the root in the alphabet and if so then we continue down the left side of the tree recursively until we find it, and if not then we continue down the right side of the tree recursively until we find it.

1.3 TREE CLASS

1.3.1 DESCRIPTION

For the creation of the binary search tree, we needed to create a tree class to represent the root of the tree and its pointers so that we could determine the path of the binary search tree. This class helped with adding each index of the word array to the binary search tree, and later on when we went back through the tree searching for specific items.

```
1  /*
2  *
3  * Assignment 4
4  * Due Date and Time: 11/19/21 before 12:00am
5  * Purpose: to implement graph and tree data structures, and to understand the performance of
6  *their traversals.
7  * Input: The user will be inputting a file containing a list of edges and vertices.
8  * Output: The program will output graph and tree data structures.
9  * @author Shannon Cordoni
10 *
11 */
12
13 public class TreeCordoni
14 {
15     /**
16      * Instance Variables
17      */
18     private String myData;
19     private TreeCordoni myNext;
20     private TreeCordoni myRoot;
21     private TreeCordoni myRight;
22     private TreeCordoni myLeft;
23     private TreeCordoni myParent;
24
25
26     /**
27      * The default Constructor for TreeCordoni
28      */
29     public TreeCordoni()
30     {
31         myData = new String();
32         myRoot = null;
33         myLeft = null;
34         myRight = null;
35         myParent = null;
36         myNext = null;
37     } //Node Cordoni
38
39     /**
40      * The full constructor for TreeCordoni
41      * @param newData the incoming data of the item
```

```

42     */
43     public TreeCordoni(String newData)
44     {
45         myData = newData;
46         myRoot = null;
47         myLeft = null;
48         myRight = null;
49         myParent = null;
50         myNext = null;
51
52     } //NodeCordoni
53
54
55     /**
56     * the setter for the item data
57     * @param newData the incoming data of the item
58     */
59     public void setData(String newData)
60     {myData = newData;} //set data
61
62     /**
63     * The getter for the item data
64     * @return the incoming data of the item
65     */
66     public String getData()
67     {return myData;} //get data
68
69     /**
70     * The setter for the next
71     * @param NewNext the incoming data
72     */
73     public void setNext(TreeCordoni newNext)
74     {myNext = newNext;} //set Next
75
76     /**
77     * the getter for the next
78     * @return the incoming data
79     */
80     public TreeCordoni getNext()
81     { return myNext;} //get Next
82
83
84     /**
85     * The setter for the root
86     * @param NewRoot the incoming data
87     */
88     public void setRoot(TreeCordoni newroot)
89     {myRoot = newroot;} //set Root
90
91     /**
92     * the getter for the root
93     * @return the incoming data
94     */
95     public TreeCordoni getRoot()
96     { return myRoot;} //get Root
97
98     /**
99     * The setter for the left tree
100    * @param NewLeft the incoming data
101    */
102    public void setLeft(TreeCordoni newLeft)
103    {myLeft = newLeft;} //set Left
104
105    /**
106    * the getter for the Left

```

```

107     * @return the incoming data
108     */
109     public TreeCordoni getLeft()
110     { return myLeft; } //get Left
111
112
113     /**
114     * The setter for the Right
115     * @param NewRight the incoming data
116     */
117     public void setRight(TreeCordoni newRight)
118     { myLeft = newRight; } //set Right
119
120     /**
121     * the getter for the Right
122     * @return the incoming data
123     */
124     public TreeCordoni getRight()
125     { return myRight; } //get Right
126
127     /**
128     * The setter for the parent
129     * @param NewParent the incoming data
130     */
131     public void setParent(TreeCordoni newParent)
132     { myParent = newParent; } //set Parent
133
134     /**
135     * the getter for the parent
136     * @return the incoming data
137     */
138     public TreeCordoni getParent()
139     { return myParent; } //get parent
140
141 } //Tree Cordoni

```

1.3.2 DESCRIPTION OF TREE CODE

This code for the Tree Class was created by previous knowledge working with the Node Class. Using the same set up each Tree was given a root, left, right, parent, and next node, along with a string to contain the word. Getters and setters were created for each to make the insert and search methods called in the main class run more smoothly.

1.4 NODE CLASS

1.4.1 DESCRIPTION

For each element in the word array a node was created to represent the word. This was so that the creation of the binary search tree would run smoothly and so that each node would be linked to the next one.

```

1  /*
2  *
3  * Assignment 4
4  * Due Date and Time: 11/19/21 before 12:00am
5  * Purpose: to implement graph and tree data structures, and to understand the performance of
6  * their traversals.
7  * Input: The user will be inputting a file containing a list of edges and vertices.
8  * Output: The program will output graph and tree data structures.
9  * @author Shannon Cordoni
10 *
11 */

```

```

12
13 public class NodeCordoni
14 {
15     /**
16      * Instance Variable for word data and node
17      */
18     private String myData;
19     private NodeCordoni myNext;
20
21     /**
22      * The default Constructor for NodeCordoni
23      */
24     public NodeCordoni()
25     {
26         myData = new String();
27         myNext= null;
28     } //Node Cordoni
29
30     /**
31      * The full constructor for NodeCordoni
32      * @param newData the incoming data of the item
33      */
34     public NodeCordoni(String newData)
35     {
36         myData = newData;
37         myNext = null;
38     } //NodeCordoni
39
40     /**
41      * the setter for the item data
42      * @param newData the incoming data of the item
43      */
44     public void setData(String newData)
45     {myData = newData;} //set data
46
47     /**
48      * The getter for the item data
49      * @return the incoming data of the item
50      */
51     public String getData()
52     {return myData;} //get data
53
54     /**
55      * The setter for the node
56      * @param NewNext the incoming node data
57      */
58     public void setNext(NodeCordoni newNext)
59     {myNext = newNext;} //set Node
60
61     /**
62      * the getter for the node
63      * @return the incoming node data
64      */
65     public NodeCordoni getNext()
66     { return myNext;} //get node
67
68 } //NodeCordoni

```

1.4.2 DESCRIPTION OF NODE CODE

This code for the Node Class was created by in class lessons but also previous knowledge from Software Development 1. Using the same set up each node was created so that it consisted of a string and a myNext

linking each node to the next. Getters and setters were created for both the nodes themselves and the data inside of them so that we would be able to call *node.getNext()*, *node.setNext()*, *node.getData()*, and *node.setData()* in the main class and to make working the binary search tree run more smoothly.

1.5 VERTEX CLASS

1.5.1 DESCRIPTION

For each vertex in the graph an actual vertex object had to be created to hold it. This class creates the vertex object to be represented in the graph. This was so that the creation of the graph representations could run more smoothly and so that the edges could be added to help represent the graphs.

```
1  /*
2  *
3  * Assignment 4
4  * Due Date and Time: 11/19/21 before 12:00am
5  * Purpose: to implement graph and tree data structures, and to understand the performance of
6  * their traversals.
7  * Input: The user will be inputting a file containing a list of edges and vertices.
8  * Output: The program will output graph and tree data structures.
9  * @author Shannon Cordoni
10 *
11 */
12
13 import java.io.BufferedReader;
14 import java.io.FileReader;
15 import java.util.Arrays;
16 import java.util.ArrayList;
17
18 public class VertexCordoni
19 {
20     /**
21     * Declare Variables
22     */
23     private String myId;
24     private boolean myIsProcessed;
25     public ArrayList <VertexCordoni> neighbors= new ArrayList <VertexCordoni>();
26     private VertexCordoni myNext;
27
28
29     /**
30     * The default Constructor for VertexCordoni
31     */
32     public VertexCordoni()
33     {
34         myId = new String();
35         myIsProcessed = false;
36         myNext = null;
37     } // vertex Cordoni
38
39     /**
40     * The full constructor for VertexCordoni
41     * @param newData the incoming data
42     */
43     public VertexCordoni(String newData)
44     {
45         myId = newData;
46         myIsProcessed = false;
47         myNext = null;
48     }
49 } // NodeCordoni
50
51 /**
```

```

52      *
53      * the setter for the vertex id
54      * @param newId the incoming data of the vertex
55      */
56      public void setId(String newId)
57      {myId = newId;} //set data
58
59      /**
60      * The getter for the vertex id
61      * @return the incoming data of the vertex
62      */
63      public String getId()
64      {return myId;} //get data
65
66
67      /**
68      *
69      * the setter for the next vertex i
70      * @param newNext the incoming data of the vertex
71      */
72      public void setNext(VertexCordoni newNext)
73      {myNext = newNext;} //set data
74
75      /**
76      * The getter for the vertex
77      * @return the incoming data of the vertex
78      */
79      public VertexCordoni getNext()
80      {return myNext;} //get data
81
82      /**
83      * The setter for the process status
84      * @param newIsProcessed the incoming process status
85      */
86      public void setProcessStatus(boolean newIsProcessed)
87      {myIsProcessed = newIsProcessed;} //set Node
88
89      /**
90      * the getter for the process status
91      * @return the incoming process status
92      */
93      public boolean getProcessStatus()
94      { return myIsProcessed;} //get node
95
96      //This checks to see if the neighbor array is empty
97      public boolean isEmpty()
98      {
99          boolean empty = false;
100
101          if(neighbors == null)
102          {
103              empty = true;
104          } //if
105          return empty;
106      } //empty
107
108
109  } //Vertex Cordoni

```

1.5.2 DESCRIPTION OF VERTEX CODE

This code for the Vertex Class was created by previous knowledge working with the Node Class. Using the same set up each Vertex was given an Id, process status, neighbor arraylist, and next. Getters and setters

were created for each to make the creation of the vertexes and edges run more smoothly, but also to help in the breadth and depth traversal of the linked object representation of the graphs.

1.6 QUEUE VERTEX CLASS

1.6.1 DESCRIPTION

For each vertex in the graph an actual vertex object had to be created to hold it. Then to perform the breadth first traversal through these vertexes, a queue had to be made to keep track of where we were in our traversal. Using a queue, this helped the breadth first traversal run successfully.

```
1  /*
2  *
3  *
4  * Assignment 4
5  * Due Date and Time: 11/19/21 before 12:00am
6  * Purpose: to implement graph and tree data structures, and to understand the performance of
7  * their traversals.
8  * Input: The user will be inputting a file containing a list of edges and vertices.
9  * Output: The program will output graph and tree data structures.
10 * @author Shannon Cordoni
11 *
12 */
13
14 public class QueueVertexCordoni {
15
16     private VertexCordoni myHead;
17     private VertexCordoni myTail;
18
19     //This method adds a vertex to the queue, it does so by adding it to the end of
20     //the queue
21     public void enqueue(VertexCordoni newVertex)
22     {
23         //this sets a temp variable to hold the current tail node
24         VertexCordoni oldTail = myTail;
25
26         //this sets the tail to be a new node and its data to be the new vertex
27         myTail = newVertex;
28
29         //This checks to see if the queue is empty
30         //if it is not empty then the old tail is set to now point to the new Node
31         if (!isEmpty()){
32             oldTail.setNext(myTail);
33         }//if
34
35         //if the queue is empty then all variables are the same because there
36         //is nothing in the queue. Then the head and tail pointer would be pointing
37         //to the same thing.
38         else{
39             myHead = myTail;
40         }//else
41
42     }//enqueue
43
44     //This method removes a vertex from the queue
45     public VertexCordoni dequeue()
46     {
47         //This sets the temp variable to null so that it can be set later.
48         VertexCordoni answer = null;
49
50         //If the queue is not empty then it will remove the first vertex from
51         //the queue
52         if (!isEmpty())
```

```

53     {
54         //This sets the temp variable to the first vertex in the list and
55         //then sets the new head pointer to the second vertex in the queue
56         answer = myHead;
57         myHead = myHead.getNext();
58
59         //if the queue is empty then the head is null
60         if(isEmpty()){
61             myHead = null;
62         }//if
63     }//if
64
65     else{
66         System.out.println("The Queue is empty");
67     }
68     return answer;
69 }//dequeue
70
71 //This checks to see if the queue is empty
72 public boolean isEmpty()
73 {
74     boolean empty = false;
75
76     if(myHead == null)
77     {
78         empty = true;
79     }//if
80     return empty;
81 }//empty
82
83 }//QueueCordoni

```

1.6.2 DESCRIPTION OF QUEUE VERTEX CODE

This code for the Queue Vertex Class was created by previous knowledge working with the Node Class, the good parts of the Queue class involve the different methods created, such as *enqueue*, *dequeue*, and *isEmpty*.

The *enqueue* method takes in a new vertex and adds it into the queue. It does this by first creating a temp variable so that we do not lose the current tail pointer of the queue. We then set the tail pointer to be a new vertex. It then checks to see if the queue is empty, if it is not empty then it takes the new vertex and adds it to the queue by setting the temp variable or the old tail to now point to the new vertex. If the queue is empty then that means that the head, and tail would be pointing to or signifying the same vertex.

The *dequeue* method creates a temp variable *answer* which is the vertex we hope to remove from the queue. It then checks to see if the queue is empty, if it is empty then we cannot remove anything from an empty queue. If it is not empty then we can set the temp variable to the head or front of the queue and then set the new head to be the next vertex in line and return the temp variable.

The *isEmpty* method checks whether or not to see if the queue is empty, it does this by looking to see if the head of the list is null, due to the fact that if there is something in the queue then there is always a head to the queue being that queues are first in first out.

1.7 OVERALL:

Overall, these Graph and Tree representations were successful in implementation. To go through each traversal and the BST lookup we can create a table for better data understanding, this table will show each and their asymptotic running time:

Breadth Traversal	Depth Traversal	BST Lookups
$O(V + E)$	$O(V + E)$	$O(\log(n))$

The table above shows a quick understanding of the methods used here. To go into more detail Breadth First Traversal and Depth First Traversal each have an asymptotic running time of $O(|V| + |E|)$, this is because each edge and vertex will be processed once while we traverse through the graph. Once we process a vertex in the graph, we do not pass through it again on our traversal. Binary Search Tree Lookup has an asymptotic running time of $O(\log(n))$ with \log being of base 2. This means that as n increases the number of operations stays the same until n doubles. From the opposite direction, when we start looking through a binary search tree we start at the root, if our target falls below the root then we have eliminated the top half of the tree to look through, and can continue on in our search. Overall, we were mostly successful in our implementation of this assignment.