CMPT 435 - Fall 2021 - Dr. Labouseur

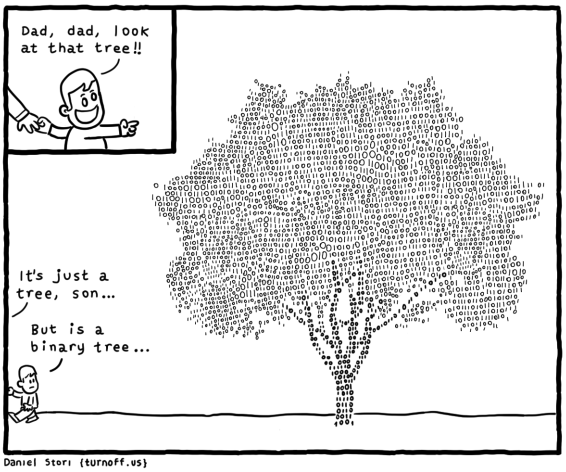
Assignment Four - Graphs and Binary Trees

Augusto Gonzalez-Bonorino
augusto.gonzalezbonorino1@marist.edu

November 19, 2021

1 Description of program

The program I will discuss in the following sections is concerned with the implementation of a Binary Tree class and, arguably one of the most important mathematical objects, Graphs which were discovered by Leonhard Euler in 1752. This document is organized in the following manner: 1) Description of the program; 2) Explanation and implementation of a Vertex; 3) Explanation and implementation of a Graph in java; 4) Manipulating and Displaying the Graph; 5) Explanation and implementation of Binary Tree; 6) Walk-through over Main class; 7) Results and conclusion; 8) Further thoughts.



2 Vertex

Graphs are composed of nodes, vertices, that can be connected to other vertices by edges. The vertex is the object who holds the data we are working with - an integer corresponding to the vertex's id in this case - a collection of neighbours (vertices connected by an edge and one-hop distance apart) and a "switch" to check if it was processed or not to avoid double checking vertices. Here is how my implementation in Java looks like:

```
public class VertexGonzalezBonorino {

  private int myId;

  private ArrayList<VertexGonzalezBonorino> myNeighbours;

  private VertexGonzalezBonorino myNext;

  private boolean processed;

  // constructor

  public VertexGonzalezBonorino(int newId)
    {

      myId = newId;
      myNext = null;
      processed = false;
      myNeighbours = new ArrayList<>();

    } // VertexGonzalezBonorino
```

Note that I store the neighbours of each vertex on an ArrayList which, even though is a bit slower, allows me to freely modify the size of the list unlike regular Arrays. Moreover, *myNext* is nothing else than a pointer which will be used to create a linked list of the nodes of the graph that will allow us to traverse it with a pair of algorithms. Keep reading to find out which algorithms. Besides the usual getters and setters I have defined 5 methods that are worth analyzing. Here is my code:

```
public ArrayList<VertexGonzalezBonorino> getNeighbours()
    {
      return myNeighbours;

    } // getNeighbours

  public void addNeighbour(VertexGonzalezBonorino neighbour)
    {
      myNeighbours.add(neighbour);

    } // addNeighbour

  public boolean isProcessed()
    {
```

```
15      return processed;
16
17    } // isProcessed
18
19  public void process()
20    {
21      processed = true;
22
23    } // process
24
25  public void unprocess()
26    {
27      processed = false;
28
29    } // unprocess
30
31
32    public void addEdge(VertexGonzalezBonorino newNeighbour)
33      {
34          myNeighbours.add(newNeighbour);
35
36      } // addEdge
```

The first two methods in the listing above give me the capability of accessing the vertex's neighbours and adding new ones. The function *isProcessed()* returns if the vertex has been visited (i.e. processed) previously, the following function processes the vertex and the third function unprocesses it, a useful helper method I included to implement multiple graphs sequentially. Lastly, the method *addEdge* connects to nodes to each other (both ways because we are implementing an undirected graph) by adding the input vertex to the current vertex's list of neighbours.

3 Undirected Graphs

A graph is a set of vertices and a collection of edges that each connect a pair of vertices. Undirected graphs are a special type of Graph for which the relations between pairs of vertices are symmetric, meaning that the edges have no direction and thus a connection between two vertices flows both ways. Graphs have applications all around, from modelling neural and social networks to communication or mechanical circuits. Therefore, it is a great thing that we have learnt how to implement and leverage such a powerful data structure. An undirected graph can be displayed in multiple ways, for this program we will employ the following three: Adjacency list, Matrix form and as a linked list. Before discussing these methods in more detail, let's take a look at the building blocks of a Graph.

```
1 public class GraphGonzalezBonorino {
2
3   private int numVertices;
4
5   public ArrayList<VertexGonzalezBonorino> myVertices;
```

```
6
7   /**
8    * Constructor
9    */
10  public GraphGonzalezBonorino()
11    {
12      myVertices = new ArrayList<>();
13
14    } // GraphGonzalezBonorino
15
16  /**
17   * Add vertex to graph given vertex object and id
18   * @param newVertex
19   * @param id
20   */
21  public void addVertex(int id)
22    {
23
24      VertexGonzalezBonorino newVertex = new VertexGonzalezBonorino
    (id);
25      myVertices.add(newVertex);
26
27      numVertices++;
28
29    } // addVertex
```

The main component to observe is the ArrayList that contains the vertices in the Graph. It is a list of Vertex objects which correspond to each of the nodes in the Graph. I also opted for initializing a variable to keep track of the number of vertices which, even though it does not have much functionality for this project, I believe might be useful if we want to keep track of the size of our Graph. lastly, the *addVertex* method, as the name suggests, simply adds a new Vertex to our graph by appending it to our *myVertices* ArrayList.

Now that we have the skeleton of our Graph and a way of adding vertices, we need a method to populate a graph given a list of commands. This will be useful for this particular project because we must read the instructions to create each graph from a text file. Each instruction is a String with the instruction and the id to assign to that vertex. Thus, as I will explain when we look at the Main class, we can store the instructions in a list of Strings and populate our graph based on those commands. Here is my method to populate our graph:

```
1  public void populateGraph(ArrayList<String> list)
2    {
3
4          //iterate through arrayList
5          for(int i = 0; i < list.size(); i++)
6          {
7              //if line is empty or a comment, print it out
8              if(list.get(i).equals("") || list.get(i).substring(0,
     2).equals("--"))
9              {
10                 System.out.println(list.get(i));
11             }
12
```

```
13            //if line wants us to add vertex...
14            else if(list.get(i).contains("vertex"))
15            {
16                //take the last string, convert to an int and add a
     vertex with this id
17                int id = Integer.parseInt(list.get(i).substring(
    list.get(i).lastIndexOf(" ") + 1));
18
19                addVertex(id);
20
21            }
22
23            //if line wants us to add an edge...
24            else if(list.get(i).contains("edge"))
25            {
26                //first id is first string after last instance of
    the letter 'e', until before the '-', convert to an int
27                int id1 = Integer.parseInt(list.get(i).substring(
    list.get(i).lastIndexOf("e") + 2,
28                        list.get(i).indexOf("-") - 1));
29                //second id is from instance of '-' till end of
    string, convert to int
30                int id2 = Integer.parseInt(list.get(i).substring(
    list.get(i).indexOf("-") + 2));
31
32                //if the first node id is a 0, we take the id's as
    they are
33                if(myVertices.get(0).getId() == 0)
34                {
35                  myVertices.get(id1).addEdge(myVertices.get(id2));
36                  myVertices.get(id2).addEdge(myVertices.get(id1));
37                }
38
39                //if the first node is not a 0 (it will be 1), we
    need to subtract one
40                // as ArrayLists start at 0 but our id is 1
41                else
42                {
43                  myVertices.get(id1 - 1).addEdge(myVertices.get(
    id2 - 1));
44                  myVertices.get(id2 - 1).addEdge(myVertices.get(
    id1 - 1));
45                }
46
47            }
48
49        } // for loop
50
51    } // populateGraph
```

It may look complicated, maybe even intimidating at first, but fear not because all we are doing here is parsing the text file to identify the instruction and the ids of the respective vertices. First, we look at the first line in the list of Strings we receive as input and check if it is a comment (characterized by a double hyphen "–") or a blank line. If so, we print it out to offer the user a description of the graph that will be implemented. Next, we check if we are being told

to add a vertex or to add edges between vertices, by checking if the respective sub-string is found in the current String, and execute it. Now, adding a vertex is straightforward but adding an edge not so much so I it will be beneficial to dive a little deeper. First we parse the String to extract the ids specified in the instruction. Then, if the we are looking at the vertex with id 0 we simply connect it to the vertex with *id2*. If it is not the first vertex, we must account for the index count that commences at 0 by subtracting 1 from the given indices. Voila! We have a fresh graph we can now play around with. Let's take a look at some of the methods we have implemented.

4 Displaying the Graph

As previously mentioned, we will implement three distinct methods for displaying the contents of our graph: Adjacency List, Adjacency Matrix and Linked List. An adjacency list is a function that maps each vertex to its neighbours. There are many ways in which it can be implemented such as using a Hash map, for example. I opted for implementing it in a way that I found more intuitive and easier to implement given my lack of experience with Java libraries. Let's take a look at the code first and then I will walk the reader through each component:

```
1  public void printAdjacencyList(GraphGonzalezBonorino graph)
2        {
3            //if first node starts at 1...
4            if(graph.myVertices.get(0).getId() == 1)
5              {
6                   //iterate through nodes
7                   for (int i = 0; i < graph.myVertices.size(); i++)
8                     {
9                         //print out i + 1 (since i is 0 but graph
     starts at  1) and print out the neighbors
10                        System.out.print("[" + (i + 1) + "] ");
11                        for (int j = 0; j < graph.myVertices.get(i).
     getNeighbours().size(); j++)
12                          {
13                              System.out.print(graph.myVertices.get(i
     ).getNeighbours().get(j).getId() + " ");
14
15                          } // inner for loop
16
17                        System.out.println();
18
19                    } // outer for loop
20
21             } // if
22
23           //else (graph starts at 0)
24           else
25             {
26                //same thing but we can print i directly
```

```
27                for (int i = 0; i < graph.myVertices.size(); i++)
28                  {
29                       System.out.print("[" + i + "] ");
30                       for (int j = 0; j < graph.myVertices.get(i).
    getNeighbours().size(); j++)
31                         {
32                              System.out.print(graph.myVertices.get(i
    ).getNeighbours().get(j).getId() + " ");
33
34                         } // inner for loop
35
36                       System.out.println();
37
38                  } // outer for loop
39
40            } // else
41
42      } // printAdjacencyList
```

I consider two possible cases: If the vertex id equals 1 or if it equals 0 (zero). Recall that Java employs indices starting at 0, instead of 1 as in mathematics. Thus, if the vertex id is one we must take this into account by simply adding 1 to each id. This is the purpose of the if statement. Nevertheless, if the vertex's id equals 0 then we do not need to alter the value of the index. The algorithm to print out the Adjacency list is equivalent in both cases. First, we iterate over each vertex in the graph. Second, print out its id inside brackets. Finally, iterate over the neighbours if the current vertex and print their respective ids on the same line.

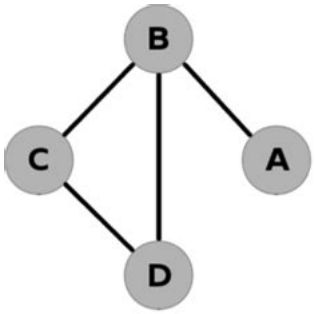
The second way of displaying the contents of our Graph is an Adjacency matrix, which is simply a square matrix containing 0s and 1s where 1 represents a connection between two vertices (i.e. they are neighbours). So, how does this look like in Java? Let's take a look at it.

```
1     /**
2      * Method to print the Matrix representation of the graph
3      * @param GraphGonzalezBonorino object
4      */
5     public void printMatrix(GraphGonzalezBonorino graph)
6       {
7           int[][] matrix = new int[graph.myVertices.size()][graph.
    myVertices.size()];
8
9           //iterate through rows of matrix
10          for(int i = 0; i <graph.myVertices.size(); i++)
11          {
12              //iterate through columns of matrix
13              for(int j = 0; j < graph.myVertices.size(); j++)
14                {
15                    //iterate through neighbors list
16                    for(int k = 0; k < graph.myVertices.get(i).
    getNeighbours().size(); k++)
17                      {
18                          //if first node starts at 0...
19                          if(graph.myVertices.get(0).getId() == 0)
```

```
                            {
                                //we check neighbor id directly to
    j and if they are equal, set matrix[i][j] to 1
                                if(graph.myVertices.get(i).
    getNeighbours().get(k).getId() == j)
                                  {
                                      matrix[i][j] = 1;
                                      break;

                                  } // if

                            } // if


                          else
                            {
                                //we check neighbor id to j + 1
    because matrices start at 0 while our graph starts at 1
                                if(graph.myVertices.get(i).
    getNeighbours().get(k).getId() == j + 1)
                                  {
                                      matrix[i][j] = 1;
                                      break;

                                  } // if

                            } // else

                          matrix[i][j] = 0;

                      } // third loop

                } // second loop

          } // first loop

          //print out matrix
          for(int i = 0; i < graph.myVertices.size(); i++)
            {
                for(int j = 0; j < graph.myVertices.size(); j++)
                  {
                      System.out.print(matrix[i][j] + " ");

                  } // inner for loop

                System.out.println();

            } // outer for loop

      } // printMatrix
```

First, we initialize a matrix of dimensions equal to the size of the graph (which is equal to its number of vertices). Then, we perform an n^2 operation to iterate through each row and column. Next, we look at each vertex in the graph and get its neighbours. Here is where the magic happens. We want to append a 1 to the matrix at every cell that "links" two vertices. For this, we check if the

id of the neighbours of the current vertex equals the id of any of the vertices in the columns. If they are equal, append a 1 otherwise append a 0. Note that we must account for Java's zero index, just like we did for Adjacency List. This might be a bit overwhelming, it was for me, thus I believe it will be beneficial to illustrate the aforementioned idea with a simple example.



a

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

b

A 4x4 matrix is depicted in the image, with vertices A, B, C, D. Think of these letters as our vertices' ids. So, we observe that A is only connected to B and so there is a 1 at the cell that "links" them. Moreover, B is connected to all other vertices but itself, and so there are 1s in every cell but the one linking B to itself. This is the exact same procedure we are employing on our *printMatrix* method. I hope that this simple example helped clarify any possible doubts.

Finally, we can display our Graph as a Linked List and traverse said list with two common methods: Depth-First-Traversal (DFT) and Breath-First-Traversal (BFT). A DFT, as the name suggests, explores the first child of the root and then each children of the root's first child before exploring its siblings. Picking the first child of the root and exploring all of its respective children is what we mean by "Depth First". With this intuition in our head let's explore my implementation in Java:

```
1     /**
2      * Depth First Search traversal of a linked list implementation
       of the graph
3      * @param vertex
4      */
5     public void depthFirstSearch(VertexGonzalezBonorino vertex)
6       {
7         if (vertex.isProcessed() == false)
8           {
9             System.out.println("Vertex id: " + vertex.getId());
10            vertex.process();
11
12          } // if
13
14        for (int i = 0; i < vertex.getNeighbours().size(); i++)
15          {
16            if (vertex.getNeighbours().get(i).isProcessed() ==
    false)
```

```
                depthFirstSearch(vertex.getNeighbours().get(i));

            } // for loop

        } // depthFirstSearch
```

First, we make sure the current vertex has not being processed already. We are lazy we do not want to be checking things more than once. Then, the vertex's id is printed out and marked as processed. Next, we perform the same operations for each of the vertex's neighbours by recursively calling our DFT method. Alternatively, we could have leverage a Stack to keep track of the vertices' ids and then print them in order. But, recursion is way cooler, and efficient, than a simple Stack so I decided to challenge myself and implement it recursively.

BFT is a bit trickier. Instead of traversing "vertically" we move "horizontally", and I encapsulate these terms in quotes because a Graph may be oriented in many other ways but I believe that deepness is easily related with vertical movement and so hopefully it gives the reader a more intuitive understanding of the underlying idea behind these methods. Hans on now. Recall I mentioned that an alternative implementation of DFT utilizes a Stack, which makes sense given the vertical movement we employ. Now we are trying to go horizontally. To achieve this we can leverage a Queue data structure. So, my BFT implementation has two components. One, the Queue class we are all familiar with:

```
    /**
 *
 * @author Augusto Gonzalez Bonorino <br><br>
 *
 * This is the class definition for QueueGonzalezBonorino
 */
public class QueueGonzalezBonorino {

  /**
   * Instance variable for the queue's tail
   */
  private VertexGonzalezBonorino myTail;

  /**
   * Instance variable for the queue's head
   */
  private VertexGonzalezBonorino myHead;

  /**
   * Default constructor for QueueGonzalezBonorino
   */
  public QueueGonzalezBonorino() {

    myTail = null;
  } // constructor

  /**
   * Method to check if Queue is empty
   * @return boolean true if empty false otherwise
```

```
   */
  public boolean isEmpty() {

    if (myHead == null)
      return true;
    else
      return false;
  } // check if empty

  /**
   * Method to add elements to the Queue
   * @param element
   */
  public void enqueue(VertexGonzalezBonorino newVertex) {

    VertexGonzalezBonorino oldTail = myTail;
    myTail = newVertex;
    myTail.setNext(null);

    if ( isEmpty() )
      myHead = myTail;
    else
      oldTail.setNext(myTail);
  } //enqueue

  /**
   * Method to delete and element from the Queue
   * @return element extracted
   */
  public VertexGonzalezBonorino dequeue() {

    VertexGonzalezBonorino ans = null;

    if ( !isEmpty() )
    {

      ans = myHead;
      myHead = myHead.getNext();

      if (isEmpty())
        myTail = null;

    } //if

    else
      System.out.println("Queue is empty");

    // return -777 if not found
    return ans;

  } //dequeue

} // QueueGonzalezBonorino
```

Two, the BFT method inside our Graph class. First, we enqueue the vertex to a Queue and process the vertex. Then, while the Queue is not empty we dequeue the first vertex, print its id, and enqueue all of its neighbours. Next we employ

a similar procedure as for DFT by dequeuing each neighbour, processing it, enqueing its neighbours and so on until every vertex has been processed. Here is how my implementation in Java looks like:

```
1   * Breadth First Search traversal of a linked list implmentation
     of the graph with a Queue
2    * @param vertex
3    */
4   public void breadthFirstSearch(VertexGonzalezBonorino vertex)
5     {
6       QueueGonzalezBonorino queue = new QueueGonzalezBonorino();
7       VertexGonzalezBonorino currVertex;
8
9       queue.enqueue(vertex);
10
11      vertex.process();
12
13      while ( !(queue.isEmpty()) )
14        {
15          currVertex = queue.dequeue();
16          System.out.println(currVertex.getId());
17
18          for (int i = 0; i < currVertex.getNeighbours().size();
    i++)
19            {
20              if ( currVertex.getNeighbours().get(i).isProcessed
    () == false )
21                {
22                  queue.enqueue(currVertex.getNeighbours().get(i)
    );
23                  currVertex.getNeighbours().get(i).process();
24
25                } // if
26
27            } // for loop
28
29        } // while loop
30
31    } // breadthFirstSearch
```

5 Binary Tree

A binary tree is a rooted tree that is also an ordered tree (a.k.a. plane tree) in which every node has at most two children. It is a powerful data structure that allows us the search elements very fast, by leveraging a sort of Divide and Conquer approach. The searching algorithm employed in this project is Binary Search, which exploits the ordered characteristic of the Binary tree by filtering out half the elements on every comparison. Thus, its asymptotic complexity is $O(logn)$. To implement the Binary Tree we first need a node class that holds the data and possesses pointers to go left or right, which I conveniently named *treeNode*. It looks like this:

```
   public class treeNode
   {
     /**
      * Instance variable to hold our data
      */
     public String myData;

     /**
      * Instance variable to point to node on the left
      */
     public treeNode myLeft;

     /**
      * Instance variable to point to node on the right
      */
     public treeNode myRight;

     /**
      * Default constructor
      * @param newData
      */
     public treeNode(String newData)
       {
         myData = newData;
         myLeft = null;
         myRight = null;

       } // Node constructor

   } // treeNode
```

Next, our Binary Tree simply contains a root *treeNode* and methods to get the number of comparisons made per look up, insert elements to the tree, print its elements in-order and to perform binary search. Here is how each of those methods look like:

```
     /**
  * Instance variable for tree's root
  */
 public treeNode myRoot;

 /**
  * Variable to keep count of comparisons
  */
 private static float BSTcomparisons;

 /**
  * Constructor to initialize root of the tree
  */
 public BSTGonzalezBonorino()
   {
     myRoot = null;

   } // Tree constructor

 /**
```

```
21    * Method to get number of comparisons
22    * @return BSTcomparisons
23    */
24   public float getComps()
25     {
26       return BSTcomparisons;
27
28     } // getComps
29
30   /**
31    * Method to reset the comparisons count
32    */
33   public void resetCount()
34     {
35       BSTcomparisons = 0;
36
37     } // resetCount
38
39   /**
40    * Recursive method to insert new data
41    * @param newData
42    */
43     public void insert(String newData)
44       {
45           myRoot = insert(myRoot, newData);
46
47       } // insert
48
49     /**
50      * Method to find correct index for given data and insert it to
       the tree
51      * @param root
52      * @param newData
53      * @return treeNode
54      */
55     public treeNode insert(treeNode root, String newData)
56       {
57           // Base Case: root is null or not
58           if (root == null)
59             {
60                 // Insert the new data, if root is null.
61                 root = new treeNode(newData);
62                 // return the current root to his sub tree
63                 return root;
64             } // if
65
66           // Check if root is greater than or equal to root
67           else if (root.myData.compareToIgnoreCase(newData) >= 0)
68             {
69                 // if root is greater than or equal newData then go
       left
70                 root.myLeft = insert(root.myLeft, newData);
71                 System.out.print("L, ");
72
73             } // else if
74
75           else
```

```
76              {
77                  // if root is less than newData then go right
78                  root.myRight = insert(root.myRight, newData);
79                  System.out.print("R, ");
80
81              } // else
82
83          return root;
84        }// insert
85
86      /**
87       * Binary Search method to find a given target
88       * @param root
89       * @param target
90       * @return target if found
91       */
92      public String search(treeNode root, String target)
93      {
94        String ans = " ";
95
96        BSTcomparisons++;
97        if (root == null || root.myData.compareToIgnoreCase(target)
    == 0)
98          {
99            ans = root.myData;
100
101         } // if
102
103       else
104         {
105
106           if ( target.compareToIgnoreCase(root.myData) < 0)
107             {
108               search(root.myLeft, target);
109               System.out.print("L, ");
110
111             } // if
112
113           else
114             {
115               search(root.myRight, target);
116               System.out.print("R, ");
117
118             } // else
119
120         } // else
121
122       return ans;
123
124     } // search
125
126
127     /**
128      * traverse Binary Tree in order
129      */
130     public void inOrder()
131       {
```

```
132         inOrder(myRoot);
133
134       } // inOrder Recursive
135
136     /**
137      * Method to print out elements in the tree in-order
138      * @param node
139      */
140     private void inOrder(treeNode node)
141       {
142
143         if (node == null)
144           {
145             return;
146
147           } // if
148
149         inOrder(node.myLeft);
150         System.out.println(node.myData);
151         inOrder(node.myRight);
152
153       } // inOrder
```

Binary search works in the following manner: if the root node is null (not found in the tree) or equal to the target (meaning the target is the root) then return the root; else check if the target is less than or greater than the value of the root node; if it is smaller then cut the tree in half and move to the left by recursively calling binary search with the left pointer of *myRoot*, if it is greater move right instead. We proceed in this manner until the target value is found or the whole tree has been traversed meaning that the value was not found.

The insert method is defined recursively as well. On one hand, if the root of the tree is empty then initialize a new *treeNode* with the input data and assign it to the root of our tree. On the other hand, if the root's value is greater than the input data the method calls itself and adds the vertex to the left of the root, or to the right if the root's value is less than the input data.

We added a convenient functionality to our graph class that keeps track of the path taken to insert, and search, a given vertex. This is simply done by printing out "L" if it went left or "R" if it moved to the right, as you can observe by the print method inside the respective functions.

Lastly, *inOrder* is a simple variation of a depth-first traversal that prints out the contents of our graph in order. It accomplishes this by visiting the left nodes first, then the root, and finally the right nodes. Since a Binary tree has all the values less than the root to the left, and those greater than the root to the right, this method works perfectly to print out the magic elements of our magic list in alphabetical order.

6 Main

Main is where all the cool stuff takes place. So far we have reviewed in detail the inner workings of each data structure that we needed to complete this project. But, how does everything come together to give life to our Graph and Magic Tree? Well, let's go by steps. The first thing is to read in all the data from the given text files, this is simply achieved by leveraging Java's *BufferedReader* as follows:

```
1         import java.util.*;
2 import java.io.*;
3 /**
4  *
5  * @author Augusto Gonzalez Bonorino <br>
6  *
7  * Assignment4GonzalezBonorino <br>
8  * Due Date and Time: 11/19/21 <br><br>
9  *
10 * Purpose: Implement graph and tree data structures, and to
       understand the performance of their traversals <br><br>
11 *
12 * Input: A text file containing data describing multiple
       undirected graphs and another one containing the Strings to
       populate BST.
13 *
14 * Output: Matrix and Adjacency list version of undirected graphs,
       plus number of comparisons per lookup. <br><br>
15 *
16 *
17 * Certification of Authenticity: <br>
18 *
19 * I certify that this assignment is entirely my own work. <br>
20 */
21 public class MainGonzalezBonorino {
22
23   private static String MAGIC_NAME = "magicitems.txt";
24   private static String GRAPH_NAME = "graphs1.txt";
25   private static String BST_NAME = "magicitems-find-in-bst.txt";
26
27   public static void main(String[] args) {
28
29
30
31     ArrayList<String> myMagicList = new ArrayList<String>();
32     ArrayList<String> myGraphList = new ArrayList<String>();
33     ArrayList<String> myBSTList = new ArrayList<String>();
34
35
36     String tempStringG = null;
37     String tempMagicString = null;
38     String tempBSTString = null;
39
40
41     String ans = "\nSome suggestions: \n"
42         + "\n* Check that the name of the file was typed correctly"
```

```
        + "\n* Make sure that you are not missing any information
    in your item description in the file"
        + "\n* Make sure you are not entering more or less items
    than specified";

    try
      {

        BufferedReader inputG = new BufferedReader(new FileReader(
    GRAPH_NAME));

        while ( (tempStringG = inputG.readLine()) != null )
          {
            myGraphList.add(tempStringG);

          } // while

        BufferedReader inputMagic = new BufferedReader(new
    FileReader(MAGIC_NAME));

        while( (tempMagicString = inputMagic.readLine()) != null)
          {
            myMagicList.add(tempMagicString);

          } // while

        BufferedReader inputBST = new BufferedReader(new FileReader
    (BST_NAME));

        while ( (tempBSTString = inputBST.readLine()) != null )
          {
            myBSTList.add(tempBSTString);

          } // while
```

These are operations we are very familiar with so I will refrain from going into further detail. Next, we parse the commands to create and populate each of the five graphs we have been tasked with implementing. I have done this in the following manner:

```
        System.out.println(" ");
    System.out.println("**** GRAPH 1 ****");
    System.out.println(" ");

    ArrayList<String> commands1 = new ArrayList<>();

        // take lines from file up until first blank line
        for(int i = 0; i < myGraphList.indexOf(""); i++){
          commands1.add(myGraphList.get(i));
        }

        // create first graph

        GraphGonzalezBonorino graph1 = new GraphGonzalezBonorino();
        graph1.populateGraph(commands1);

```

```
17          System.out.println(" ");
18          System.out.println("**** Matrix form ****");
19          //graph1.printGraph(graph1);
20          graph1.printMatrix(graph1);
21
22          System.out.println(" ");
23          System.out.println("**** Adjancency List ****");
24          graph1.printAdjacencyList(graph1);
25
26          System.out.println(" ");
27          System.out.println("**** Depth First Search ****");
28          graph1.depthFirstSearch(graph1.myVertices.get(0));
29
30          // vertices have been process with DFS so we need to set
        them all to false again
31
32          for(int i = 0; i < graph1.myVertices.size(); i++)
33          {
34              graph1.myVertices.get(i).unprocess();
35
36          } // for loop
```

We read each line until reaching a blank line (which means that the following lines correspond to commands for a new graph), and use our method *populateGraph* that was previously explained to, well, populate the graph, duh. Once we have populated our graph we make use of each of the methods we have just went over to print the Adjacency Matrix, Adjacency List, Depth-First-traversal and Breath-First-Traversal, respectively. The code is straightforward. The only detail worth explaining is the for loop after the DFT. Recall that for the traversals we process each vertex as we visit them. To perform BFT we need to unprocess every vertex otherwise it the program will see that everything has been processed already and will not perform any comparisons. This is what the for loop is for: loop over each vertex and unprocess it. Then, we proceed on a similar fashion to initialize, populate and print out each of the necessary graphs.

The final component of Main is implementing our binary tree. For this, we initialize a graph object, populate it with all the Strings in "magicitems.txt", print its contents in order using the method we discussed in the previous section, search for the elements given in "magicitems-find-in-bst.txt", and lastly compute the average number of comparisons.

7 Results and conclusion

In this documentation we have analyzed my implementation in Java of various data structures and algorithms, but we have not discussed their performance yet. The asymptotic running time of Depth-First-Traversal and Breadth-First-Traversal is $O(\mid V \mid + \mid E \mid)$, where $\mid V \mid$ and $\mid E \mid$ denote the cardinality of Vertices and Edges (i.e. number of vertices and edges), respectively. This is so because in the worst case scenario we would have to go through every vertex

and edge in the graph once. Lastly, the asysmptotic performance of Binary Search Tree is $O(h)$ where h denotes the height of the tree. Since in our case our tree is perfectly balanced we calculate h to equal $logn$ and so the worst case scenario for BST is $O(logn)$, pretty fast. The running time of BST can be double checked by counting the comparisons made per lookup and computing the average, which turns out to be 10.64 for a list of 666 elements. Note that $\log_2 666 = 9.4$ so our calculation seem correct.

8 Further thoughts

As always there are many components of this program that could be improved. For example, parsing the file should have a more general structure in order for it to be able to take variations of the text file, as of now I am manually computing the indices of each blank line to be able to get the distinct list of commands which is not great software craftsmanship. Moreover, I could re-write my Adjacency List method to implement a Hash map instead of the nested for loop, which runs on $O(n^2)$.

Graphs and Trees are extremely useful data structures and have applications in multiple different disciplines. I have learnt a lot while working on this project, not even mentioning the absurd amount of code that I have to write to implement each component. I finally feel like I am grasping the main idea of software development and project management in Java. As I grow my skills toolbox and my time at Marist College approaches an end, I am faced with a decision of staying in academia or working in the industry. Here is how I encode my decisions inside my head after learning about graphs and trees: