Assignment Four – Graphs and Trees

Genevieve Anderson
Genevieve.anderson1@marist.edu

November 19, 2022

1 BST Class

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

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

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

Next, I created my "find()" function [46]. This is the algorithm I created for finding nodes in the tree after they are inserted. First, the "findComparisons" are incremented [47] since a comparison is made at the beginning each time. Then, the target that is being searched is first compared to the "current" node to see if they are equal [48]. This would indicate that the target has been found. If not, the target node we are trying to find is then recursively compared to the "current" node as "less than" and "greater than" [51, 55]. Based on what side of the "current" node is being searched, the path is updated accordingly.

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

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

```
1 public class BST {
2
3     Node root;
4     String path;
5     int comparisons;
6     String findPath; // For when we are finding the path of the target being searched in the tree
7     int findComparisons; // For when we are finding the target in the tree
8     int totalComparisons; // For the average
9     int findTotalComparisons;
10
11     // Constructor
12     public BST() {
13         root = null;
14         path = "";
15         comparisons = 0;
16         findPath = "";
17         findComparisons = 0;
18         totalComparisons = 0;
19         findTotalComparisons = 0;
20     }
21
22     // Function for setting the root
23     public void setRoot(String value) {
24         root = new Node(value);
25     }
26
27     // Function for inserting nodes into the tree
28     public Node insert(Node node, String value) {
29         if (node == null) {
30             return new Node(value); // New node we are inserting
31         }
```

```java
        // Determines which side of tree to place node, will compare the value we are trying to insert to the "current" node
        if (value.compareTo(node.getName()) < 0) { // Less than "current" node
            node.left = insert(node.left, value); // Recursively inserts node on the left
            path += "L, "; // Used when printing path of insertion
            comparisons++; // Increment comparisons
        } else { // Greater than "current" node
            node.right = insert(node.right, value); // Recursively inserts node on the right
            path += "R, ";
            comparisons++;
        }
        return node;
    }

    // Function for finding nodes in the tree
    public Node find(Node node, String target) {
        findComparisons++; // Increment comparisons for find, different variable than comparisons for insert
        if (target.equals(node.getName())) { // When the node has been found
            return node;
        }
        else if (target.compareTo(node.getName()) < 0) { // Searching left hand side of tree
            findPath += "L, "; // Used when printing path of search
            find(node.getLeft(), target); // Recursively finding the node on the left
        }
        else { // Searching right hand side
            findPath += "R, ";
            find(node.getRight(), target);
        }
        return null; // If the target is not found
    }

    // Function for printing the path and comparisons of when they are inserted
    public void path(String name) {
        System.out.println(name + ": " + path + " and the number of comparisons is: " + comparisons);
        path = "";
        totalComparisons += comparisons;
        comparisons = 0;
    }

    // Function for printing the path and comparisons for when they are searched
    public void findPath(String name) {
        System.out.println(name + ": " + findPath + " and the number of comparisons is: " + findComparisons);
        findPath = "";
        findTotalComparisons += findComparisons;
        findComparisons = 0;
    }
```

```

    // Function for calculating the average
    public void average(int avgComparisons, int size) {
        double average = ((double)(avgComparisons))/size; // dobule for decimal places
        System.out.println("The average is: " + String.format("%.2f", average));
    }

    // Function for ITW
    public void ITW(Node node) {
        if (node == null) {
            return;
        }
        // Prints nodes in order
        ITW(node.left);
        System.out.println(node.getName());
        ITW(node.right);
    }


}
```

2 Node Class

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

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

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

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

3 Graph class

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

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

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

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

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

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

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

```java
    }

    // Parameterized Constructor
    public Graph(ArrayList<Vertex> vList) {
        vertexList = vList;
        adjList = new int[vertexList.size() + 1][vertexList.size() + 1]; // +1 for formatting of the matrix
        matrix = new int[vertexList.size() + 1][vertexList.size() + 1];
    }

    // Setting the Graph
    public void setGraph(ArrayList<Vertex> vertices) {
        vertexList = vertices;
        adjList = new int[vertexList.size() + 1][vertexList.size() + 1];
        matrix = new int[vertexList.size() + 1][vertexList.size() + 1];
    }

    // Function for building matrix
    public void matrix() {
        for (int r = 0; r < matrix.length; r++) {
            for (int c = 0; c < matrix.length; c++) {
                if (r == 0 && c != 0) {
                    matrix[r][c] = c; // for making the list of numbers at the top for formatting
                }
                else if (c == 0 && r!= 0) {
                    matrix[r][c] = r; // for making the list of numbers at the side for formatting
                }
                else if (c != 0 && r != 0) { // Inside the matrix
                    boolean edge = vertexList.get(r - 1).haveEdge(c); // at each point in the matrix going through to see if it has an edge
                    if (edge == true) {
                        matrix[r][c] = 1; // If there is an edge the matrix point will be set to 1
                    }
                    else {
                        matrix[r][c] = 0; // If there isnt an edge
                    }
                }
            }
        }
    }

    // Function for adjacency list
    public String adjList() {
        String result = "";
        for (int i = 0; i< vertexList.size(); i++) {
            Vertex v = vertexList.get(i); // Individual vertex
            result += "[" + v.getVertexByID() + "]" + Arrays.toString(v.neighbors.toArray()) + "\n"; // Formatting the adjacency list
        }
```

```
        return result;
    }

    // Function for printing the results of the graph
    public void printResults() {
        for (int r = 0; r < matrix[0].length; r++) {
            System.out.print("[");
            for (int c = 0; c < matrix[0].length; c++) {
                System.out.print(matrix[r][c] + ", ");
            }
            System.out.println("]"); // For each line of the matrix
        }
        System.out.println(adjList()); // Printing the adjacency list
    }

    // Function for depth first traversal
    public void DFS(Vertex v) {
        if (v.processed == false) {
            System.out.println(v.getVertexByID());
            v.processed = true;
        }
        for (Vertex n: vertexList) { // for all the vertices in the vertex list
            if (n.processed == false) {
                DFS(n);
            }
        }
    }

    // Function for breadth first traversal
    public void BFS(Vertex v) {
        Queue queue = new Queue();
        queue.enqueue(v);
        v.processed = true;
        while (queue.isEmpty() == false) {
            Vertex currentVertex = queue.dequeue();
            System.out.println(currentVertex.getVertexByID());
            for (Vertex n: currentVertex.neighbors()) {
                if (n.processed == false) {
                    queue.enqueue(n);
                    n.processed = true;
                }
            }
        }
    }

    // Function for resetting prior to each traversale
    public void resetTraversal() {
        for (Vertex n: vertexList) {
            n.processed = false; // Making them all false again
            n.setNext(null);
        }
    }
}
```

4 Queue Class

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

```java
//Building the framework for the queue
public class Queue {
    //I am declaring the head and tail inside the queue, then initialize them in the constructor
    Vertex head; //Queue needs both head and tail, whereas the stack only needed a head
    Vertex tail;

    //Queue constructor
    public Queue() {
        head = null; //Initializing the head
        tail = null; //Initializing the tail
    }

    //Function for enqueue
    public void enqueue(Vertex y) { //Parameter is a Vertex because we will be a new Vertex being enqueued in the queue
        if (head == null) { //If the queue was empty, then the if loop will make it so that the incoming Vertex to the queue will become both the head and the tail
            tail = y; //Setting the incoming Vertex to the tail
            head = tail;  //Setting the head to the tail
        }
        else { //The else statement is for if there are already Vertexs in the queue
            tail.setNext(y);
            tail = y; //Setting the incoming Vertex to the tail
        }
    }

    //Function for dequeue
    public Vertex dequeue() {
        Vertex oldHead = head; //Creating old head makes it so that we don't completly lose the Vertex that is getting removed from the queue
        head = head.getNext(); //The new head is goig to be the head that was originally next in line
        if (head == null) { //If the head is null, then the tail should also be null
            tail = null; //Doing this because otherwise, the tail would still be pointing at the Vertex we are returning
        }
        return oldHead;
    }

    //Function for checking is the queue is empty
    public Boolean isEmpty() {
        if (head == null) {
            return true;
```

```
        }
        else {
            return false;
        }
    }
}
```

5 Vertex Class

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

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

```java
    , returns the node from the variable next
    }

    public void setNext(Vertex m) { //"m" is the node that we are going to set next equal to, for the pointer
        next = m; //I am using m so that the pointer is not null. the pointer will not be null until the end of the linked list.
    }

    // Function for getting the id
    public int getVertexByID() {
        return id;
    }

    // Function for checking edge
    public boolean haveEdge(int id) {
        boolean check = false;
        for (int i = 0; i < neighbors.size(); i++) {
            if (neighbors.get(i).getVertexByID() == id) {
                check = true; // means there is an edge
            }
        }
        return check;
    }

    // Function to return all neighbors of a vertex
    public ArrayList<Vertex> neighbors() {
        return neighbors;
    }

    // toString for printing results
    public String toString() {
        return id + ""; // Converting the value being stored to a string so that the adjList can be printed and formatted correctly
    }
}
```

6 Main Program

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

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

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

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

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

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

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

```java

        String[] magicItemsFind = new String[42];
        try { //Trying to find the file
            File file = new File("magicitems-find-in-bst.txt");
            Scanner sc = new Scanner(file);
            int index = 0;
            while (sc.hasNextLine()) {
                String item = sc.nextLine();
                item = item.toUpperCase();
                magicItemsFind[index ++] = item;
            }
        }
        catch (FileNotFoundException e) { // If we cant find the file
            e.printStackTrace();
        }

        BST tree = new BST();
        tree.setRoot(magicItems[0]); // We can do this because the first magic item will always be the root
        for (int i = 0; i < magicItems.length; i++) {
            tree.insert(tree.root, magicItems[i]);
            tree.path(magicItems[i]);
        }
        tree.average(tree.totalComparisons, magicItems.length);
        System.out.println("-----------------------Inserting the magic items done, now searching for the targets !-------------------------");
        for (int i = 0; i < magicItemsFind.length; i++) { // Finding the target list in the bst
            tree.find(tree.root, magicItemsFind[i]);
            tree.findPath(magicItemsFind[i]);
        }
        tree.average(tree.findTotalComparisons, magicItemsFind.length);
        tree.ITW(tree.root);

        // Graph codes now
        ArrayList<Vertex> vertices = new ArrayList<Vertex>(); // vertices that will be provided to the graph

        try { //Trying to find the file
            File file = new File("graphs1.txt");
            Scanner sc = new Scanner(file);

            while (sc.hasNextLine()) {
                String item = sc.nextLine();
                String[] parse = item.split(" ");
                if (parse[0].equals("add") && parse[1].equals("vertex")) {
                    Vertex v = new Vertex(Integer.parseInt(parse[2])); // Getting the vertex number and creating a new vertex based off of it
                    vertices.add(v); // Adding the vertex to the array list of all the vertices
                }
                else if (parse[0].equals("add") && parse[1].equals(
```

```java
"edge")) {
                int firstVertex = Integer.parseInt(parse[2]); // First vertex the edge will be connected to
                int secondVertex = Integer.parseInt(parse[4]); // Second vertex the edge will be connected to
                if (firstVertex == 0) {
                    vertices.get(firstVertex).add(vertices.get(secondVertex)); // Correlating the edges to the vertices array of all the vertices
                    vertices.get(secondVertex).add(vertices.get(firstVertex)); // Same as above, but vice versa so they are connected both ways
                }
                else {
                    vertices.get(firstVertex - 1).add(vertices.get(secondVertex - 1)); // Correlating the edges to the vertices array of all the vertices
                    vertices.get(secondVertex - 1).add(vertices.get(firstVertex - 1)); // Same as above, but vice versa so they are connected both ways
                }
            }
            else if (parse[0].equals("new") && parse[1].equals("graph")) {
                Graph graph = new Graph();
                graph.setGraph(vertices);
                graph.matrix(); // Matrix
                graph.adjList(); // Adjacency list
                graph.printResults(); // Results
                if (vertices.isEmpty() ==  false) { // Traversals
                    System.out.println("-------------------------------------Below is the depth first traversal-------------------------------------");
                    graph.DFS(vertices.get(0));
                    graph.resetTraversal(); // Resetting traversal
                    System.out.println("-------------------------------------Below is the breadth first traversal-------------------------------------");
                    graph.BFS(vertices.get(0));
                }
                vertices.clear(); // So that the graphs aren't added on top of each other
            }
        }
        // For the last graph
        Graph graph = new Graph();
        graph.setGraph(vertices);
        graph.matrix();
        graph.adjList();
        graph.printResults();
        if (vertices.isEmpty() ==  false) {
            System.out.println("-------------------------------------Below is the depth first traversal-------------------------------------");
            graph.DFS(vertices.get(0));
```

```
                graph.resetTraversal();
                System.out.println("-------------------------------------Below is the breadth first traversal-------------------------------------");
                graph.BFS(vertices.get(0));
            }
            vertices.clear();
        }
        catch (FileNotFoundException e) { // If we cant find the file
            e.printStackTrace();
        }
    }
}
```