Assignment Four

Jake Vissicchio
Jake.Vissicchio1@Marist.edu

November 19, 2021

1 Analysis

1.1 Depth-First Traversal and Breadth-First Traversal:

Has a complexity of $O(|V| + |E|)$. With V = total vertices and E = total edges. This is because while traversing we are visiting each vertex once which gives us $O(|V|)$ and worst case scenario we must cross every edge once which gives us $O(|E|)$, then we put them together to get $O(|V| + |E|)$.

1.2 Lookups in the Binary Search Tree:

Has a complexity of $O(log(n))$ if the Binary Search Tree is balanced.
A lack of balance will increase the complexity to $O(n)$.
This is because the height of the tree actually represents the complexity.
The height of the tree should be around log(n) if the tree is balanced but if it is unbalanced it could have a height as big as n worst case scenario.
The average number of comparisons recorded from my Lab reflected my analysis as it was 10.64 which about equals log(666).

2 Code Listings

2.1

BST Insert with Comments:

```
public void BSTInsert(BinarySearchTreeVissicchio tree,
     NodeTreeVissicchio newNode)
    {
        //trailing is a pointer that is always one step behind
        NodeTreeVissicchio trailing = null;
        //current is a pointer to where we currently are at in the tree
        NodeTreeVissicchio current = tree.root;

        //go through BST until we find a null node
        while (current != null)
        {
            trailing = current;
            int result = newNode.myItem.compareToIgnoreCase(current.myItem);
            if (result < 0)
            {
                //go left
                current = current.left;
                System.out.print("L");

            }//if
            else //must be >=
            {
                //go right
                current = current.right;
                System.out.print("R");

            }//else
        }//while

        //the parent node will be the node one step behind
        newNode.parent = trailing;
        //set the first node of the BST
        if (trailing == null)
        {
            tree.root = newNode;
            System.out.print("Root: " + tree.root.myItem);
            //System.out.print(root.myItem);
        }//if
        //start adding child nodes
        else
        {
            int result = newNode.myItem.compareToIgnoreCase(trailing.myItem);
```

```
            if(result < 0)
            {
                //go left since less than (sooner in alphabet)
                trailing.left = newNode;
                System.out.print(": " + trailing.left.myItem);

            }//if
            else
            {
                //go right since greater than (further in alphabet)
                trailing.right = newNode;
                System.out.print(": " + trailing.right.myItem);

            }//else
        }//else
        System.out.println("");
    }//BSTInsert
```

2.2

In-Order Traversal with Comments:

```
//Works much like a Sort for a Binary Search Tree

public void inOrderTraversal(NodeTreeVissicchio myNode)
{
    //Start at the left most node and make your way to the right most node
    //Should print out the items in alphabetical order
    if (myNode != null)
    {
        inOrderTraversal(myNode.left);
        System.out.println(myNode.myItem);
        inOrderTraversal(myNode.right);
    }
}//InOrderTraversal
```

2.3

BST Lookup with comments:

```
//Works to find the specified target starting from the root
//using recursion.

public void BSTSearch(NodeTreeVissicchio root, String target)
```

```
{
    int result = root.myItem.compareToIgnoreCase(target);
    //target has been found
    if (result == 0)
    {
        compCount++;
        System.out.print(": The item, " + target + ", was found");
        found = true;
    }//if
    //continue searching
    else
    {
        if (result > 0 && root.left != null)
        {
            //go left to continue searching for the target
            compCount++;
            System.out.print("L");
            BSTSearch(root.left, target);
        }
        if (result < 0 && root.right != null)
        {
            //go right to continue searching for the target
            compCount++;
            System.out.print("R");
            BSTSearch(root.right, target);
        }
    }//else
    if (found == false)
    {
        //item was never found while going through the BST
        System.out.print(": The item, " + target + ", was not found");
        //set found back to true so it only prints once
        found = true;
    }
}//BSTSearch
```

2.4

Adding Edges for a Graph represented as Linked Objects with comments:

```
//add the edge by linking two verticies together making them neighbors

public void addEdgeLinkedObjects(VertexVissicchio current,
    VertexVissicchio newNeighbor)
```

```
{
    //add the newNeighbor vertex to the current vertex's neighbor arraylist
    current.neighbors.add(newNeighbor);

    //add the current vertex to newNeighbor vertex's neighbor arraylist
    newNeighbor.neighbors.add(current);
}//addEdgeLinkedObjects
```

2.5

Depth-First Traversal with comments:

```
//makes use of recursion
//will print out the source's first neighbor, then the
//first neighbor's first neighbor (unprocessed), and so on

public void depthFirstTraversal(VertexVissicchio v)
{
    if (v.isProcessed == false)
    {
        System.out.println(v.id);
        v.isProcessed = true;
    }
    for (int i = 0; i < v.neighbors.size(); i++)
    {
        VertexVissicchio neighbor = v.neighbors.get(i);
        if (neighbor.isProcessed == false)
        {
            depthFirstTraversal(neighbor);
        }
    }
}//depthFirstTraversal
```

2.6

Breadth-First Traversal with comments:

```
//makes use of queues since it is first-in, first-out.
//will print in order of the neighbors visited starting
//from the source vertex. All (non processed) neighbors will be printed
//before going to see that neighbor's neighbors and so on.

public void breadthFirstTraversal(VertexVissicchio v)
{
```

```
    QueueVissicchio queue = new QueueVissicchio();
    queue.enqueue(v);
    v.isProcessed = true;
    System.out.println("Breadth First Traversal: ");
    System.out.println("------------------------------------------");
    while (!queue.isEmpty())
    {
        VertexVissicchio current = queue.dequeue();
        System.out.println(current.id);
        //int n = current.neighbors.size();
        for (int i = 0; i < current.neighbors.size(); i++)
        {
            VertexVissicchio neighbor = current.neighbors.get(i);
            if (neighbor.isProcessed == false)
            {
                queue.enqueue(neighbor);
                neighbor.isProcessed = true;
            }//if
        }//for
    }//while
}//breadthFirstTraversal
```

2.7

Adding Edges for a Graph represented as a Matrix with comments:

```
//I used a 2D array to implement the Matrix
//NOTE: Because it uses an array the number of verticies must be known
//      so while reading the file I made sure to count the total number
//      of verticies for each graph to prevent problems.

//add the edge by linking two verticies together making them neighbors
public void addEdgeMatrix(int current, int neighbor)
{
    //the first index represents the current vertex
    //and the second index represents the neighboring vertex
    matrix[current][neighbor] = 1;
    matrix[neighbor][current] = 1;
}//addEdgeMatrix
```

2.8

Adding Edges for a Graph represented as a Adjacency List with comments:

```
//I used my hash table from the prior assignment which was an array of
//queues.
//NOTE: Because it uses an array the number of verticies must be known
//      so while reading the file I made sure to count the total number
//      of verticies for each graph to prevent problems.

//add the edge by linking two verticies together making them neighbors
public void addEdgeAdjList(int current, int neighbor)
{
    hashTable[current].enqueue(neighbor);
    hashTable[neighbor].enqueue(current);
}
```