Assignment Four

Shannon Brady

shannon.brady2@Marist.edu

November 19, 2022

1 Binary Search Tree

1. The Binary Search Tree class has the following attributes:
   a) root: the tree root
   b) numComparisons: number of comparisons necessary to complete a search
   c) totalComparisons: a provisional attribute that tracks the total number of comparisons completed over a certain number of searches.
2. Line 13-36: Insert a new node in the tree by traversing through the tree to find the location where it belongs, and then setting it's left and right pointers accordingly.
3. Line 39-66: Print the node path from the root to a given target
   a) Line 48-51: Break if the target is the root node.
   b) Line 53-58: Traverse to the left or right depending on if the target is to the left or right of the current node.
   c) Line 59-62: Break from the loop because target node has been found.
4. Line 75-81: Prints out the entire Binary Search tree by traversing the tree in order.
5. Running Time Analysis: Given that the tree is balanced, the expected running time complexity of a BST lookup is $O(log_2 n)$. This is due to the fact that with each comparison, the remaining search area is cut in half; therefore, running time is proportional to the height of the tree. In an unbalanced tree, the worst-case scenario is a time complexity off O(n), with each node being linked in direct succession of the previous node.

```
1  class BinarySearchTree {
2
3      private Node root = null;
4      private int numComparisons;
5      private double totalComparisons;
6
```

```
7       public BinarySearchTree() {
8           this.numComparisons = 0;
9           this.totalComparisons = 0;
10      }
11
12      // Inserts a new node into the tree
13      public void insert(String nodeName) {
14          Node node = new Node(nodeName);
15          Node curr = this.root;
16          Node trailing = null;
17
18          while (curr != null) {
19              trailing = curr;
20              if (node.getName().compareTo(curr.getName()) < 0) {
21                  curr = curr.getLeft();
22              } else {
23                  curr = curr.getRight();
24              }
25          }
26          node.setParent(trailing);
27          if (trailing == null) {
28              this.root = node;
29          } else {
30              if (node.getName().compareTo(trailing.getName()) < 0) {
31                  trailing.setLeft(node);
32              } else {
33                  trailing.setRight(node);
34              }
35          }
36      }
37
38      // Prints the path from the root to a target node
39      public void printNodePath(String target) {
40          Node curr = this.root;
41          String path = "";
42          Boolean isFound = false;
43          this.numComparisons = 0;
44
45          System.out.println("Node path for: " + target);
46          while (curr != null) {
47              this.numComparisons++;
48              if (target.compareTo(this.root.getName()) == 0) {
49                  path = "no path to root node";
50                  isFound = true;
51                  break;
52              } else {
53                  if (target.compareTo(curr.getName()) < 0) {
54                      curr = curr.getLeft();
55                      path += " L";
56                  } else if (target.compareTo(curr.getName()) > 0) {
57                      curr = curr.getRight();
58                      path += " R";
59                  } else {
60                      if (target.compareTo(curr.getName()) == 0 || target.compareTo(curr.getName()) == 0) {
```

```
                        isFound = true;
                        break;
                    }
                }
            }
        }
        this.totalComparisons += this.numComparisons;
        if (isFound) {
            System.out.println(path);
        } else {
            System.out.println("Cannot find " + target);
        }
    }

    public void inOrderTraversal(Node node) {
        if (node != null) {
            inOrderTraversal(node.getLeft());
            System.out.print(" => " + node.getName());
            inOrderTraversal(node.getRight());
        }
    }

    // Print the number of comparisons for each search
    public void printNumComparisons() {
        System.out.println("Number of Comparisons: " + this.numComparisons);
    }

    // Print the average number of comparisons after a given number of searches
    public void printAvgComparison(int total) {
        System.out.printf("\nAverage Number of Comparisons:  %.2f %n", this.totalComparisons/total);
    }

    /* Getters and Setters */
    public Node getRoot() {
        return this.root;
    } // getRoot

    public void setRoot(Node newRoot) {
        this.root = newRoot;
    } // setRoot
}
```

2 Node (used in the Binary Search class)

1. The Node class has the following attributes:
    a) name: node name
    b) left: pointer to the left node
    c) right: pointer to the right node
    d) parent: pointer to the parent node

```java
class Node {

    private String name = "";
    private Node left = null;
    private Node right = null;
    private Node parent = null;

    /* Node class constructor */
    public Node(String name) {
        this.name = name;
        this.left = null;
        this.right = null;
        this.parent = null;
    }

    /* Getters and Setters */
    public String getName() {
        return this.name;
    } // getName

    public Node getLeft() {
        return this.left;
    } // getLeft

    public Node getRight() {
        return this.right;
    } // getRight

    public Node getParent() {
        return this.parent;
    } // getParent

    public void setName(String newName) {
        this.name = newName;
    } // setName

    public void setLeft(Node newLeft) {
        this.left = newLeft;
    } // setLeft

    public void setRight(Node newRight) {
        this.right = newRight;
    } // setLeft

    public void setParent(Node newParent) {
        this.parent = newParent;
    } // setParent

    @Override
    public String toString() {
        return("Name: " + this.getName() + "\n");
    } // toString
}
```

3 Undirected Graph

1. The Graph class has the following attributed:
    a) vertices: a Linked List of vertex objects within the graph
    b) adjList: an AdjacencyList object containing graph adjacency list data
    c) matrix: a Matrix object containing graph matrix data
2. Line 13-15: Add a vertex to the vertices list
3. Line 18-26: Create a graph edge
    a)
    a) Line 19-22: Get the two vertex objects and add them to each other's neighbors list
    b) Line 24: Create an entry in the adjacency list
    c) Line 25: Create a matrix entry
4. Line 29-44: Complete a depth-first traversal of the graph
    a) Line 30-33: Print all unprocessed vertices and set to processed
    b) Line Line 34-43: Loop through each vertex's neighbors, and if a neighbor isn't already processed, recursively call DFS with that neighbor as the parameter.
5. Line 47-69: Complete a breadth-first traversal of the graph
    a) Line 48-51: Create a queue of vertices to be processed. Enqueue the given vertex and set to processed.
    b) Line 53-67: While the queue isn't empty, dequeue from the queue and loop through the current vertex's neighbors. If any of the neighbors aren't already processed, enqueue the neighbor and set to processed.
6. Line 71-75: De-process all the vertices
7. Running Time Analysis: A DFS traversal is a stack-based algorithm that explores as far as possible along each edge before backtracking and repeating the cycle again. A BFS traversal is a queue-based algorithms explores vertices in order of their distance from the original vertex, with closest vertices being traversed first. Both traversals are have an asymptotic running time of roughly O(V+E), with V being the set of vertices in the graph, and E being the set of edges.

```
1  class Graph {
2
3      private LinkedList vertices = null;
4      private AdjacencyList adjList = null;
5      private Matrix matrix = null;
6
7      Graph(String[] verticeArr) {
8          this.vertices = new LinkedList();
9          this.adjList = new AdjacencyList(verticeArr.length);
10         this.matrix = new Matrix(verticeArr);
11     }
12
13     public void addVertex(String vid) {
14         this.vertices.add(vid);
```

```
    }

    // Create an edge between two vertices
    public void createEdge(String vid1, String vid2) {
        Vertex vertex1 = this.vertices.getVertexByID(vid1);
        Vertex vertex2 = this.vertices.getVertexByID(vid2);
        vertex1.getNeighbors().add(vid2);
        vertex2.getNeighbors().add(vid1);

        this.adjList.createEntry(vid1, vid2);
        this.matrix.createEntry(vid1, vid2);
    }

    // Depth-first traveral
    public void DFS(Vertex vertex) {
        if (!vertex.isProcessed) {
            System.out.print(vertex.getID() + " ");
            vertex.isProcessed = true;
        }
        for (int i=0; i<vertex.getNeighbors().getSize(); i++) {
            Vertex neighbor = vertex.getNeighbors().getVertexAt(i);

            // Get the og vertex object with this id
            // This is the object with populated neighbors
            neighbor = this.vertices.getVertexByID(neighbor.getID());
            if (!neighbor.isProcessed) {
                this.DFS(neighbor);
            }
        }
    }

    // Breadth-first traversal
    public void BFS(Vertex vertex) {
        Queue queue = new Queue();
        Vertex temp = new Vertex(vertex.getID());
        queue.enqueue(temp);
        vertex.isProcessed = true;

        while (!queue.isEmpty()) {
            Vertex currVertex = queue.dequeue();
            currVertex = this.vertices.getVertexByID(currVertex.getID());

            System.out.print(currVertex.getID() + " ");

            for (int i=0; i<currVertex.getNeighbors().getSize(); i++) {
                Vertex neighbor = currVertex.getNeighbors().getVertexAt(i);
                neighbor = this.vertices.getVertexByID(neighbor.getID());
                if (!neighbor.isProcessed) {
                    temp = new Vertex(neighbor.getID());
                    queue.enqueue(temp);
                    neighbor.isProcessed = true;
                }
            }
        }
```

```
69      }
70
71      public void resetVerticeProcessStatuses() {
72          for (int i=0; i<this.vertices.getSize(); i++) {
73              this.vertices.getVertexAt(i).isProcessed = false;
74          }
75      }
76
77      /* Getters */
78      public LinkedList getVertices() {
79          return this.vertices;
80      } // getVertices
81
82      public AdjacencyList getAdjacencyList() {
83          return this.adjList;
84      } // getAdjacencyList
85
86      public Matrix getMatrix() {
87          return this.matrix;
88      } // getMatrix
89  }
```

4 Matrix

1. The Matrix class has the following attributes:
   a) matrixArr: a two-dimensional array used to represent a matrix
   b) verticeArr: an array of Vertex objects used to create the matrix
   c) Line 6-10: Initialize the matrix with dashes.
   d) Line 22-31: Create an entry at both intersection with the provided vertices. Mark entries with a "1".
   e) Line 34-43: Print the matrix

```
1   class Matrix {
2
3       private String[][] matrixArr = null;
4       private String[] verticeArr = null;
5
6       public Matrix(String[] vertices) {
7           this.matrixArr = new String[vertices.length][vertices.length];
8           this.verticeArr = vertices;
9           this.init();
10      }
11
12      // Initializes the matrix array with dashes
13      public void init() {
14          for (int i=0; i<this.matrixArr.length; i++) {
15              for (int j=0; j<this.matrixArr[i].length; j++) {
16                  this.matrixArr[i][j] = "-";
17              }
```

```
        }
    }

    // Creates an entry in the matrix array with a value of "1"
    public void createEntry(String rowEle, String columnEle) {
        for (int i=0; i<this.matrixArr.length; i++) {
            for (int j=0; j<this.matrixArr[i].length; j++) {
                if (this.verticeArr[i].compareTo(rowEle) == 0 && this.verticeArr[j].compareTo(columnEle) == 0) {
                    this.matrixArr[i][j] = "1";
                    this.matrixArr[j][i] = "1";
                }
            }
        }
    }

    // Prints the matrix array in matrix format
    public void print() {
        System.out.println();
        for (int i=0; i<this.matrixArr.length; i++) {
            System.out.print(this.verticeArr[i] + " ");
            for (int j=0; j<this.matrixArr[i].length; j++) {
                System.out.print(this.matrixArr[i][j] + " ");
            }
            System.out.println();
        }
    }

    public Matrix getMatrix() {
        return this;
    }
}
```

5 Adjacency List

1. The AdjacencyList class has the following attributes:
    a) adjTable: a HashTable used to represent an adjacency list of vertices and their neighbors
    b) Line 11-14: Add both vertices and each other's adjTable
    c) Line 17-19: Print the adjacency list

```
class AdjacencyList {

    private HashTable adjTable = null;

    AdjacencyList(int numVertices) {
        // a hash table of vertices
        this.adjTable = new HashTable(numVertices+1);
    }

    // Add each vertice to the other vertice's neighbors list
    public void createEntry(String vertex1, String vertex2) {
```

```
12          this.adjTable.put(Integer.parseInt(vertex1), vertex2);
13          this.adjTable.put(Integer.parseInt(vertex2), vertex1);
14      }
15
16      // Print the adjacency list
17      public void print() {
18          this.adjTable.print();;
19      }
20
21      public AdjacencyList getAdjacenyList() {
22          return this;
23      }
24  }
```

6 Main

1. Line 9-40: Create an array of magic items
2. Line 43-73: Create an array of target magic items
3. Line 76-94: Binary Search Tree
    a) Line 77-80: Create and initialize the BST
    b) Line 82-88: Print the node path from the root to each of the target magic items. Print the individual number of comparisons for each search and calculate the average.
    c) Line 92: Print and in-order tree traversal of the BST
4. Line 97-136: Undirected Graph
    a) Line 104-114: Create an array with a length of the number of graphs in the file
    b) 119-154: For each graph in the file, create an array of vertex id's and then use this array to initialize a new graph. Add each of the vertices to the graph. Then, create each of the edges.
5. Line 157: Print the vertices in the graph
6. Line 161: Complete a depth-first traversal
7. Line 169: Complete a breadth-first traversal
8. Line 173: Print the adjacency list
9. Line 176: Print the matrix

```
1   import java.io.File;
2   import java.io.FileNotFoundException;
3   import java.util.Scanner;
4
5   class Assignment_4 {
6
7       public static void main(String[] args) {
8
9           // Get magic items ---------------------------------------------------
10          File file = new File("magicitems.txt");
11          Scanner myReader = null;
```

```
12          String[] magicItems = null;
13          int i = 0;
14          int numLines = 0;
15
16          try {
17              myReader = new Scanner(file);
18              while (myReader.hasNextLine()) {
19                  // count the number of lines in the file
20                  numLines++;
21                  myReader.nextLine();
22              }
23              myReader.close();
24
25              // create magic items array
26              magicItems = new String[numLines];
27
28              myReader = new Scanner(file);
29              while (myReader.hasNextLine()) {
30                  // initialize magic items array with magic items
31                  String data = myReader.nextLine();
32                  magicItems[i] = data.toLowerCase();
33                  i++;
34              }
35              myReader.close();
36          } catch (FileNotFoundException e) {
37              System.out.println("An error occurred.");
38              e.printStackTrace();
39          }
40          // END get magic items ------------------------------------------------
41
42
43          // Get the target magic items -----------------------------------------
44          file = new File("magicitems-find-in-bst.txt");
45          String[] targetMagicItems = null;
46          i = 0;
47          numLines = 0;
48
49          try {
50              myReader = new Scanner(file);
51              while (myReader.hasNextLine()) {
52                  // count the number of lines in the file
53                  numLines++;
54                  myReader.nextLine();
55              }
56              myReader.close();
57
58              // create target magic items array
59              targetMagicItems = new String[numLines];
60
61              myReader = new Scanner(file);
62              while (myReader.hasNextLine()) {
63                  // initialize with target magic items
64                  String data = myReader.nextLine();
65                  targetMagicItems[i] = data.toLowerCase();
```

```
            i++;
        }
        myReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
    // END get the target magic items ----------------------------------------


    // Binary Search Tree ----------------------------------------------------
    BinarySearchTree myBST = new BinarySearchTree();
    for (i=0; i<magicItems.length; i++) {
        myBST.insert(magicItems[i]);
    }

    for (i=0; i<targetMagicItems.length; i++) {
        myBST.printNodePath(targetMagicItems[i]);
        myBST.printNumComparisons();
        System.out.println();
    }

    myBST.printAvgComparison(targetMagicItems.length);
    System.out.println();

    // Print an in-order traversal of the tree
    myBST.inOrderTraversal(myBST.getRoot());

    // END Binary Search Tree ------------------------------------------------


    // Undirected Graph ------------------------------------------------------

    // Get the graph file and process graph data
    file = new File("graphs1.txt");
    int numGraphs = 0;

    try {
        // First, count the number of graphs in the file
        myReader = new Scanner(file);
        while (myReader.hasNextLine()) {
            if (myReader.nextLine().contains("new")) {
                numGraphs++;
            }
        }
        // Initialize an array of graphs
        Graph[] graphs = new Graph[numGraphs];
        System.out.println("\n\n\nNumber of graphs in file: " + numGraphs);
        myReader.close();

        // Next, process the graph data for each graph
        myReader = new Scanner(file);
        i=0;
        while (i < numGraphs) {
```

```
                if (myReader.nextLine().contains("new")) {
                    String line = myReader.nextLine();
                    String verticesStr = "";
                    while (line.contains("vertex")) {
                        // Get the vertex id (vid)
                        String vid = line.substring(line.lastIndexOf(" ") + 1);
                        // Keep a string of vid data
                        verticesStr += vid + " ";
                        line = myReader.nextLine();
                    }

                    // Create the graph using an array of vid's
                    String[] verticeArr = verticesStr.split(" ");
                    graphs[i] = new Graph (verticeArr);
                    for (int j=0; j<verticeArr.length; j++) {
                        graphs[i].addVertex(verticeArr[j]);
                    }

                    // Create all the edges
                    while (line.contains("edge")) {
                        // The line substring with edge data
                        String edgeStr = line.substring(9);

                        // Get each vid connecting the edge and create the edge
                        String[] vertices = edgeStr.split(" - ");
                        String vertex1 = vertices[0];
                        String vertex2 = vertices[1];
                        graphs[i].createEdge(vertex1, vertex2);

                        if (myReader.hasNextLine()) {
                            line = myReader.nextLine();
                        } else {
                            break;
                        }
                    }
                    System.out.println("--------------------------------------------------------------------------------");
                    System.out.println("Graph " + i + ":");
                    graphs[i].getVertices().print();
                    System.out.println();

                    System.out.println("Depth-first Traversal:");
                    graphs[i].DFS(graphs[i].getVertices().getHead());
                    System.out.println();
                    System.out.println();

                    // De-process all vertices
                    graphs[i].resetVerticeProcessStatuses();

                    System.out.println("Breadth-first Traversal:");
                    graphs[i].BFS(graphs[i].getVertices().getHead());
                    System.out.println();
                    System.out.println();

                    graphs[i].getAdjacencyList().print();
```

```
174                     System.out.println();
175
176                     graphs[i].getMatrix().print();
177                     System.out.println("--------------------------------------------------------------------------------");
178                     i++;
179                 }
180             }
181             myReader.close();
182         } catch (FileNotFoundException e) {
183             System.out.println("An error occurred.");
184             e.printStackTrace();
185         }
186         // END Undirected Graph ----------------------------------------------------
187     }
188 }
```