CMPT-435-Assignment-4

Ahmed Handulle

December 4, 2022

In this document, I will be explaining my code from Assignment 4 in detail.

This first part of the project will show the implementation details of the undirected graphs using matrix and adjacency list. Then following is the implementation of the binary search tree.

1 External Java Packages

Below is a list of external Java packages that I have used to create the first part of the program which is the implementation of the undirected graphs.

```java
import java.io.File; // importing file utility package to manage our file
import java.util.Scanner; // Importing the Scanner class to read text files
import java.util.ArrayList; // importing the ArrayList class to store elements
```

Below is the Main Class of the first part of the project.

2 Main Class

```java

// This class uses external modules or classes such as BST, Matrix, AdjacencyList, and LinkedList
public class Main {

    public static void main(String[] args) {

        try {
            File myObj = new File("graphs1.txt");

            Scanner myReader = new Scanner(myObj);

            // Start reading the file line by line
            while (myReader.hasNextLine()) {
                //store the data of the line
                String data = myReader.nextLine();

                // Display information about the graph
```

```java
            displayGraphInfo(data);

            // Check if the line starts with "new graph" to create a new matrix and adjacencylist for that specific graph
            if (data.startsWith("new graph")) {

                // Create an Adjancecy List
                AdjacencyList adjacencyList = new AdjacencyList();

                // Creating an ArrayList to store the verticies of the linked objects graph representation
                ArrayList<LinkedObjects> linkObjGraph = new ArrayList<>();

                // Iterate over the nex few lines to count the vertices
                String tempString = myReader.nextLine();

                //Check if the graph starts at vertax zero which means it is at the ground level
                Boolean vertexStartsZero = isGroundLevel(tempString);

                // Start counting the vertices to to create the Matrix later
                int countVertices = 0;

                //this wile loop will jump from one line to another to count the verticies and add verticies to adjacency list and the linked Objects
                while (myReader.hasNextLine() & tempString.startsWith("add vertex")) {
                    adjacencyList.addVertax(tempString);
                    linkObjGraph.add(new LinkedObjects(getVertex(tempString)));
                    countVertices++;
                    tempString = myReader.nextLine();
                }

                // Create the matrix with the number of vertices (countVertices X countVertices)
                if (!vertexStartsZero) {
                    //New Matrix Object
                    Matrix matrixGraph = new Matrix(countVertices);
                    //Create Matrix
                    matrixGraph.createMatrix();

                    // create edges for both graphs
                    while (myReader.hasNextLine() & tempString.startsWith("add edge")) {
                        // iterate over the edges and add edges to the matrix, the adjacencyList and the linked Objects
                        // Example => edge (1-2)
                          int[] edgeVerticies = toFilterString(tempString);
```

```
                matrixGraph.addEdge(edgeVerticies);
                adjacencyList.addEdge(edgeVerticies, vertexStartsZero);
                //Add vertex object 2 to vertex object 1 (edgeVerticies[0] - 1 => because Arraylist start storing at index 0 when graph starts at vertex 1)
                linkObjGraph.get(edgeVerticies[0] - 1).neighbors
                        .add(linkObjGraph.get(edgeVerticies[1] - 1));
                //Add vertex object 1 to vertex object 2
                linkObjGraph.get(edgeVerticies[1] - 1).neighbors
                        .add(linkObjGraph.get(edgeVerticies[0] - 1));
                tempString = myReader.nextLine();

            }

            //Printing the graph in both forms (Matrix & AdjacencyList)
            matrixGraph.display();
            adjacencyList.display();
            System.out.println("Linked Objects(DFS): ");
            DFS(linkObjGraph.get(0), linkObjGraph);
            System.out.println();
            System.out.println();

            System.out.println("Linked Objects(BFS): ");
            BFS(linkObjGraph.get(0), linkObjGraph);
            System.out.println();

        } else {

            // Initialize matrix object
            Matrix matrixGraph = new Matrix(countVertices);
            // Create actual matrix that starts at zero
            matrixGraph.createGroundLevelMatrix();

            //Create edges for both graphs
            while (myReader.hasNextLine() & tempString.startsWith("add edge")) {
                // iterate over the edges and add edges to the matrix and the adjacencyList and then display
                // Example => edge (0-1)
                int[] edgeVerticies = toFilterString(tempString);
                matrixGraph.addEdgeGroundLevel(edgeVerticies);
                adjacencyList.addEdge(edgeVerticies, vertexStartsZero);
                //Add vertex object 1 to vertex object 0
```

```java
                            linkObjGraph.get(edgeVerticies[0]).neighbors
                                    .add(linkObjGraph.get(edgeVerticies[1]));
                            //Add vertex object 0 to vertex object 1
                            linkObjGraph.get(edgeVerticies[1]).neighbors
                                    .add(linkObjGraph.get(edgeVerticies[0]));
                            tempString = myReader.nextLine();
                        }

                        // Making sure the scanner reads the last line of the file
                        if (!myReader.hasNextLine() & tempString.startsWith("add edge")) {
                            matrixGraph.addEdgeGroundLevel(toFilterString(tempString));
                            adjacencyList.addEdge(toFilterString(tempString), vertexStartsZero);
                            //Add vertex object 1 to vertex object 0
                            linkObjGraph.get(toFilterString(tempString)[0]).neighbors
                                    .add(linkObjGraph.get(toFilterString(tempString)[1]));
                            //Add vertex object 0 to vertex object 1
                            linkObjGraph.get(toFilterString(tempString)[1]).neighbors
                                    .add(linkObjGraph.get(toFilterString(tempString)[0]));
                        }


                        //Printing the graph in both forms (Matrix & AdjacencyList)
                        matrixGraph.display();
                        adjacencyList.display();
                        System.out.println("Linked Objects(DFS): ");

                        DFS(linkObjGraph.get(0), linkObjGraph);
                        System.out.println();
                        System.out.println();

                        System.out.println("Linked Objects(BFS): ");
                        BFS(linkObjGraph.get(0), linkObjGraph);
                        System.out.println();

                    }
                }
            }
            myReader.close();
        } catch (Exception e) {
```

```java
            System.out.println("An error occurred.");
            e.printStackTrace();
        }

        // Creating an object of BST class from module BST.java in
    another file
        BST binarySearchTree = new BST();
        // Creating an ArrayList of String object to store lines of
     strings
        ArrayList<String> magicItems = new ArrayList<String>();

        // Creating another ArrayList of String Object to store the
     magic items we are searcing
        ArrayList<String> magicItemsFind = new ArrayList<String>();

        // creating a new file object
        File f = new File("magicitems.txt");

        // Creating another file for accessing the second magic
    items file
        File f2 = new File("magicitems-find-in-bst.txt");
        // Call the filtering function to filter the magic items
    and append it to the arrayList
        filterFile(magicItems, f);
        // Call the filtering function to filter the "magicitems-
    find-in-bst.txt" and append it to the arrayList
        filterFile(magicItemsFind, f2);

        // Converting the String ArrayList of magicitems to String
    Array
        String[] magicItemsArray = toArrayOfString(magicItems);
        // Converting the magicItemsFind ArrayList to String Array
        String[] magicItemsFindArray = toArrayOfString(
    magicItemsFind);
        //Inserting magicitems in the BST
        populateTree(magicItemsArray, binarySearchTree);
        System.out.println("
    ---------------------------------------------");
        System.out.println("Printing the elements in the tree in In
    -Order-Traversal");
        // Printing magic items in In-Order-Traversals from the BST
        binarySearchTree.inorder(binarySearchTree.root);
        // Searching the selected magic items from the BST and
    printing their path
        searchBST(magicItemsFindArray, binarySearchTree);
        System.out.println("
    ---------------------------------------------");
        // Calculating the Average comparison count for the
    searched elements
        AvgComparisonCount(binarySearchTree);
    }

    //This method will convert ArrayList String to Array String
    public static String[] toArrayOfString(ArrayList<String>
    arrayListofStrings) {
        String[] magicItemsArray = new String[arrayListofStrings.
    size()];
```

```java
        for (int i = 0; i < arrayListofStrings.size(); i++) {
            magicItemsArray[i] = arrayListofStrings.get(i);
        }
        return magicItemsArray;
    }
    // This function gets called from the main method and it inserts the data in the BST
    public static void populateTree(String[] magicItems, BST binarySearchTree) {
        System.out.println("-------------------------------------------");
        System.out.println("Populating the BST with elements and printing their path");
        // Populating the BST with the magic items
        for (String eachstring : magicItems) {
            binarySearchTree.insert(eachstring);
        }
    }
    // This function gets called from the main method and it searches gives strings from the BST
    public static void searchBST(String[] selectedMagicItems, BST binarySearchTree) {
        System.out.println("-------------------------------------------");
        System.out.println("Printing the path of each of the searched element in the BST");
        // Searching all the strings in the magicItemsFindArray one by one
        for (String eachString : selectedMagicItems) {
            binarySearchTree.search(eachString);
        }
    }
    // This function prints out the average comparison count from the BST
    public static void AvgComparisonCount(BST binarySearchTree) {
        System.out.println("Average Comparison Count: "
                + binarySearchTree.avgSearchComparison(binarySearchTree.totalComparisonCount));
    }
    // This function takes a file and an ArrayList, it filters out the file and append it to the ArrayList line by line
    public static void filterFile(ArrayList<String> magicItems, File file) {
        try {
            //Create scanner object to read the file
            Scanner myreader = new Scanner(file);
            //filter out each line using the regression expression
            while (myreader.hasNextLine()) {
                String linee = myreader.nextLine();
                magicItems.add(linee.replaceAll("[^A-Za-z]", "").toLowerCase());
            }
            myreader.close();
            //Catch if there are any errors while processing the file
        } catch (Exception e) {
            e.getStackTrace();
```

```java
        }
    }

    // This function will check if a graph has a ground level zero vertex
    public static Boolean isGroundLevel(String tempString) {
        //split string at spaces
        String[] stringParts = tempString.split(" ");

        //Converting Char to Interger
        int charConvertion = Integer.parseInt(stringParts[stringParts.length - 1]);
        if (charConvertion == 0) {
            return true;
        } else {
            return false;
        }
    }

    // This function will filter a string and return the two vertices where an edge will be created
    public static int[] toFilterString(String string) {
        //split the string at spaces and get the vertices from the string
        int[] edge = new int[2];
        String[] splitLine = string.split(" ");
        edge[0] = Integer.parseInt(splitLine[2]);
        edge[1] = Integer.parseInt(splitLine[4]);
        return edge;

    }

    // This funtion dispalys information about the graph
    public static void displayGraphInfo(String data) {
        if (data.startsWith("--")) {
            System.out.println(data);
            System.out.println();
        }
    }

    // This funtion return the vertex number as an integer
    public static int getVertex(String tempString) {
        //Filter String
        String[] stringParts = tempString.split(" ");
        int vertex = Integer.parseInt(stringParts[stringParts.length - 1]);

        return vertex;
    }

    // Depth First Search Algorithm

    // This following functions takes an object of class linkedObjects and performs DFS
    public static void DFS(LinkedObjects v, ArrayList<LinkedObjects> linkObjGraph) {

```

```
        // V is the current vertext of the graph
        if (!v.processed) {
            System.out.print(v.id + " ");
            v.processed = true;
        }

        // Loop through the neighbors of the current vertex
        for (LinkedObjects s : v.neighbors) {
            if (!s.processed) {
                DFS(s, linkObjGraph);
            }
        }

        // Edge Case => iteratng through the entire linkedObjects when the graphs has disconnected parts to make sure every vertex is visited
        for (LinkedObjects s : linkObjGraph) {
            if (!s.processed) {
                DFS(s, linkObjGraph);
            }
        }

    }

    // breadth First Search Algorithm

    // This following functions takes an object of class linkedObjects and performs BFS
    public static void BFS(LinkedObjects v, ArrayList<LinkedObjects> linkObjGraph) {
        // Setting the status (processed) of the verticies back to false after it has been modified by the DFS function
        for (LinkedObjects x : linkObjGraph) {
            if (x.processed)
                x.processed = false;
        }
        // initialie a queue object
        Queue q = new Queue();
        // push unvisited vertex to the queue
        q.enqueue(v);
        // mark the vertex as visited
        v.processed = true;
        // iterate through the neighboring verticies
        while (!q.empty()) {
            // Retriefe object from the queue
            LinkedObjects currentVertex = q.dequeue();
            // Print the id of the retrieved object
            System.out.print(currentVertex.id + " ");
            // Iterate through the neighbors of the retrieved object and push them to the queue
            for (LinkedObjects s : currentVertex.neighbors) {
                if (!s.processed) {
                    q.enqueue(s);
                    s.processed = true;
                }
            }
        }
```

```
        // Edge case => if the graph has some disconnected parts
        for(LinkedObjects s : linkObjGraph){
            if(!s.processed){
                // push unvisited vertex to the queue
                q.enqueue(s);
                // mark the vertex as visited
                s.processed = true;
                // iterate through the neighboring verticies
                while (!q.empty()) {
                    // Retriefe object from the queue
                    LinkedObjects currentVertex = q.dequeue();
                    // Print the id of the retrieved object
                    System.out.print(currentVertex.id + " ");
                    // Iterate through the neighbors of the retrieved object and push them to the queue
                    for (LinkedObjects x : currentVertex.neighbors) {
                        if (!x.processed) {
                            q.enqueue(x);
                            x.processed = true;
                        }
                    }
                }
            }
        }

        System.out.println();
    }
}

----------------------End of the Main Class------------------------
```

3 Binary Search Tree Class

Below is the Binary Search Tree Class implementation from the BST module

```
import java.util.ArrayList; // importing the ArrayList class to store elements

public class BST {

    // This Node class will be used to store elements
    public class Node {
        String data;
        Node left;
        Node right;

        Node(String element) {
            this.data = element;
            this.left = null;
            this.right = null;
        }
    }

    // Initializing the root node of the BST
    Node root;
```

```java

    // Constructor for the BST where the tree is empty
    BST() {
        root = null;
    }

    public void insert(String string) {

        // Creating an CharacterArrayList to store the path of each node
        ArrayList<Character> pathArray = new ArrayList<Character>();

        Node newNode = new Node(string);

        if (this.root == null) {
            this.root = newNode;
        } else {

            // Temporary node for storing the root of the tree
            Node temp = this.root;
            // Keeping truck of the parent node of the position where the new node will be inserted
            Node refPositionOfNewParentNode = null;

            // Find the position of the new Node
            while (temp != null) {
                refPositionOfNewParentNode = temp;
                if (newNode.data.compareTo(temp.data) < 0) {
                    temp = temp.left;
                    pathArray.add('L');

                } else if (newNode.data.compareTo(temp.data) > 0) {
                    temp = temp.right;
                    pathArray.add('R');
                } else {
                    return;
                }
            }
            // Printing out the path of the Node in a CharacterArrayList
            System.out.println(pathArray);

            // Check if the new node is greater or less than it's parent and insert newNode in it's correct positioin
            if (newNode.data.compareTo(refPositionOfNewParentNode.data) < 0) {
                refPositionOfNewParentNode.left = newNode;
            } else {
                refPositionOfNewParentNode.right = newNode;
            }
        }
    }

    // Printing elements in In-order-traversal (left, root, right) using Recursiosn
    public void inorder(Node root) {
```

```
        if (root != null) {
            inorder(root.left);
            System.out.println(root.data);
            inorder(root.right);
        }
    }

    // Storing the number of comparison of each look up in this arrayList
    ArrayList<Integer> totalComparisonCount = new ArrayList<Integer>();

    // This function returns the average comparisons of each look up
    public Double avgSearchComparison(ArrayList<Integer> comparisonCounting) {

        int sum = 0;
        for (int i = 0; i < comparisonCounting.size(); i++) {
            sum += comparisonCounting.get(i);
        }
        double avg = sum / comparisonCounting.size();

        return avg;
    }

    // Searching elements in the BST and retrun their path in a characterArrayListx
    public void search(String element) {

        int comparisonsCount = 0;

        // Creating an CharacterArrayList to store the path of each node
        ArrayList<Character> pathArray2 = new ArrayList<Character>();

        Node treeRoot = this.root;

        while (treeRoot != null) {
            if (element.compareTo(treeRoot.data) < 0) {
                treeRoot = treeRoot.left;
                pathArray2.add('L');
                comparisonsCount++;
            } else if (element.compareTo(treeRoot.data) > 0) {
                treeRoot = treeRoot.right;
                pathArray2.add('R');
                comparisonsCount++;
            } else {
                //Printing the look up path
                System.out.println("Look-up-Path: " + pathArray2);
                // Printing the number of comparison for each look up
                System.out.println("Number of comparisons: " + comparisonsCount);
                break;
            }
```

```
        }
        // Check if the element is not in the tree
        if (treeRoot == null) {
            System.out.println(element + " : is not in the tree");
        }

        //Adding the comparison count to the totalComparisonCount
    arrayList to calculate the average
        totalComparisonCount.add(comparisonsCount);
    }
}
```

4 LinkedList Class

Below is the LinkedList Class implementation from the Linkedlist module

```

public class LinkedList {
    // This Node class will be used to store elements in the linked
     list
    public class Node {
        int data;
        Node next;

        Node(int element) {
            this.data = element;
            this.next = null;
        }
    }
    // Initialize the root of the linkedlist
    Node head;
    Node tail;
    int size = 1;

    // Constructor for the Linked List

    LinkedList(int vertax) {
        this.head = new Node(vertax);
        this.tail = null;
    }

    /**
     * this Function returns the length of the linked List
     */
    public int len(){
        return this.size;
    }

    /**
     * This function checks if the linked list is empty
     */
    public boolean isEmpty() {
        if (len() > 1) {
            return false;
        } else {
            return true;
```

```java
        }
    }
    /**
     * Now we are creating a Funtion to put elements into the hashtable using linkedlist object
     */
    public void inputEdge(int vertax) {
        Node newNode = new Node(vertax);

        if (isEmpty()) {
            this.head.next = newNode;
        } else {
            this.tail.next = newNode;
        }
        this.tail = newNode;
        this.size++;
    }

// Printing elements in the adjacencyList
    public void print(){
        Node currentNode = this.head;
        while (currentNode != null) {
            System.out.print(currentNode.data + " --> ");
            currentNode = currentNode.next;
        }
        System.out.println();
    }
}
```

5 Adjacency List Class

Below is the Adjacency List Class implementation from the AdjacencyList module

```java
import java.util.ArrayList;

public class AdjacencyList {
    // This class will have an ArrayList of LinkedLists of Nodes
    ArrayList<LinkedList> arrayList;
    AdjacencyList() {
        // Create an ArrayList of LinkedList of Nodes
        arrayList = new ArrayList<>();
    }

    // This function will add a vertex to the adjacencylist graph represeantion
    public void addVertax(String tempString) {
        //Add a vertex to the AdjacencyList
        String[] splitString = tempString.split(" ");
        int vertex = Integer.parseInt(splitString[splitString.length - 1]);
        arrayList.add(new LinkedList(vertex));
    }

    public void addEdge(int[] vertexArray, Boolean vertexStartsZero) {
```

```
        //Add edge to the adjacencyList
        if (!vertexStartsZero) {
            arrayList.get(vertexArray[0] - 1).inputEdge(vertexArray[1]);
        } else {
            arrayList.get(vertexArray[0]).inputEdge(vertexArray[1]);
        }
    }

    // This method will print everything in the Adjacency List
    public void display() {
        for (int i = 0; i < arrayList.size(); i++) {
            arrayList.get(i).print();
        }
        System.out.println();
    }
}
```

6 Matrix Class

Below is the Matrix Class implementation from the Matrix module

```

public class Matrix {
    int vertices;
    int matrix[][];

    Matrix(int numberOfVertices) {
        this.vertices = numberOfVertices;
        // add number 1 to the numberOfVertices to make room for the matrix representation nicely
        this.matrix = new int[numberOfVertices + 1][numberOfVertices + 1];
    }

    /*
    * [0,1,2,3,4]
    * [1,0,0,0,0]
    * [2,0,0,0,0]
    * [3,0,0,0,0]
    * [4,0,0,0,0]
    */

    // This function create a matrix for a graph that starts at vertex 1
    public void createMatrix() {
        // Length of the Matrix
        int n = matrix.length;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                // Check if the first row is zero to print the values of the vertices
                if (i == 0) {
                    matrix[i][j] = j;
```

```
                    // check if the first column is zero to print the values of the vertices
                } else if (j == 0) {
                    matrix[i][j] = i;
                    // else make the relationship (edge) place holder a zero
                } else {
                    matrix[i][j] = 0;
                }
            }
        }
    }

    // This function create a Matrix for a graph that starts at vertex zero
    public void createGroundLevelMatrix() {
        int n = matrix.length;
        /*
        * [0,0,1,2,3]
        * [0,0,0,0,0]
        * [1,0,0,0,0]
        * [2,0,0,0,0]
        * [3,0,0,0,0]
    */

        for (int i = 0; i < n; i++) {
            int k = 2;
            for (int j = 0; j < n; j++) {
                if (i < k & j < k) {
                    matrix[i][j] = 0;
                } else if (i == 0 & j >= k) {
                    matrix[i][j] = j - 1;
                } else if (j == 0 & i >= k) {
                    matrix[i][j] = i - 1;
                } else {
                    matrix[i][j] = 0;
                }
            }
        }
    }

    // this function takes an array of two integer verticies and then creates an edge between them
    public void addEdge(int[] vertexArray) {
        // Find the location of both vertices and set their relationship to 1
        matrix[vertexArray[0]][vertexArray[1]] = 1;
        matrix[vertexArray[1]][vertexArray[0]] = 1;

    }

    // this function takes an array of two integer verticies and then creates an edge between them
    public void addEdgeGroundLevel(int[] vertexArray) {
        // Find the location of both vertices and set their relationship to 1
        matrix[vertexArray[0] + 1][vertexArray[1] + 1] = 1;
```

```
        matrix[vertexArray[1] + 1][vertexArray[0] + 1] = 1;
    }

    //Display the matrix
    public void display() {
        // Length of the Matrix
        int n = matrix.length;
        for (int i = 0; i < n; i++) {
            System.out.print("[ ");
            for (int j = 0; j < n; j++) {
                System.out.print(+this.matrix[i][j] + " ");
            }
            System.out.print("]");
            System.out.println("");
        }
        System.out.println();
    }
}
```

7 LinkedObjects Class

Below is the LinkedObjects Class implementation from the LinkedObjects module

```
import java.util.ArrayList;

public class LinkedObjects {
    // Object Attributes => Each Vertex is represented as an object
     with attributes as class variables
    int id;
    Boolean processed;
    ArrayList<LinkedObjects> neighbors;

    // Initialize the class/Vertex object
    LinkedObjects(int id) {
        this.id = id;
        this.processed = false;
        this.neighbors = new ArrayList<>();
    }

}
```

8 Queue Class

Below is the Queue Class implementation from the Queue module

```

import java.util.ArrayList;

public class Queue {

    // This queue class will keep truck of the visited verticie

```

```
    ArrayList<LinkedObjects> queueData;

    Queue() {
        queueData = new ArrayList<>();
    }

     // Checking if the Queue is empty
     public boolean empty() {
         if (queueData.size() == 0) {
             return true;
         } else {
             return false;
         }
     }
     // Add an element to the queue
     public void enqueue(LinkedObjects v) {
         this.queueData.add(v);
     }

     // Remove an element from the queue
     public LinkedObjects dequeue() {
         LinkedObjects v = this.queueData.get(0);
         this.queueData.remove(0);
         return v;
     }
}


//-------------------Go to the Next Page for Results---------------
```

9 Results

The Average number of Comparison Counts or each of the look-ups was: 9.0

The worst time complexity of the insertion function in the binary search tree is O(h) where "h" is the height of the tree since we need to go down h number of levels until we find the correct position for the element.

The worst time complexity of the search function of the binary search tree is O(logn) since we are cutting the tree into half each time until we find the element.

The inputEdge function of the LinkedList class's worst case is (1) since we are inserting the edge at the end of LinkedList of the vertex which also takes O(1)

The run time of both the BFS and SFS is O(V+E) where V and E are the numbers of vertices and edges of the graph respectively.