CMPT-435-Assignment-4

Ahmed Handulle

November 30, 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.

```
1 import java.io.File; // importing file utility package to manage
      our file
2 import java.util.Scanner; // Importing the Scanner class to read
      text files
3 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

```
1  public class Main {
2  // This class will contain all the logic for the program. The first
       part will show the representation of the graphs using Matrix
      and Adjacency List. The second part will be the logic for the
      Binary search tree.
3
4
5      //This method will convert ArrayList String to Array String
6      public static String[] toArrayOfString(ArrayList<String>
      arrayListofStrings) {
7          String[] magicItemsArray = new String[arrayListofStrings.
      size()];
8          for (int i = 0; i < arrayListofStrings.size(); i++) {
9              magicItemsArray[i] = arrayListofStrings.get(i);
10         }
11         return magicItemsArray;
12     }
13     // This function gets called from the main method and it
      inserts the data in the BST
```

```
14      public static void populateTree(String[] magicItems, BST
        binarySearchTree) {
15          System.out.println("
        -----------------------------------------");
16          System.out.println("Populating the BST with elements and
        printing their path");
17          // Populating the BST with the magic items
18          for (String eachstring : magicItems) {
19              binarySearchTree.insert(eachstring);
20          }
21      }
22      // This function gets called from the main method and it
        searches gives strings from the BST
23      public static void searchBST(String[] selectedMagicItems, BST
        binarySearchTree) {
24          System.out.println("
        -----------------------------------------");
25          System.out.println("Printing the path of each of the
        searched element in the BST");
26          // Searching all the strings in the magicItemsFindArray one
         by one
27          for (String eachString : selectedMagicItems) {
28              binarySearchTree.search(eachString);
29          }
30      }
31      // This function prints out the average comparison count from
        the BST
32      public static void AvgComparisonCount(BST binarySearchTree) {
33          System.out.println("Average Comparison Count: "
34                  + binarySearchTree.avgSearchComparison(
        binarySearchTree.totalComparisonCount));
35      }
36      // This function takes a file and an ArrayList, it filters out
        the file and append it to the ArrayList line by line
37      public static void filterFile(ArrayList<String> magicItems,
        File file) {
38          try {
39              //Create scanner object to read the file
40              Scanner myreader = new Scanner(file);
41              //filter out each line using the regression expression
42              while (myreader.hasNextLine()) {
43                  String linee = myreader.nextLine();
44                  magicItems.add(linee.replaceAll("[^A-Za-z]", "").
        toLowerCase());
45              }
46              myreader.close();
47              //Catch if there are any errors while processing the
        file
48          } catch (Exception e) {
49              e.getStackTrace();
50          }
51      }
52
53      // This function will check if a graph has a ground level zero
        vertex
54      public static Boolean isGroundLevel(String groundLevelVertex) {
55          //Converting Char to Interger
```

```
56          int charConvertion = Integer.parseInt(groundLevelVertex);
57          if (charConvertion == 0) {
58              return true;
59          } else {
60              return false;
61          }
62
63      }
64
65      // This function will filter a string and return the two
        vertices where an edge will be created
66      public static int[] toFilterString(String string) {
67          //split the string at spaces and get the vertices from the
        string
68          int[] edge = new int[2];
69          String[] splitLine = string.split(" ");
70          edge[0] = Integer.parseInt(splitLine[2]);
71          edge[1] = Integer.parseInt(splitLine[4]);
72          return edge;
73
74      }
75
76      public static void main(String[] args) {
77
78          try {
79              File myObj = new File("graphs1.txt");
80
81              Scanner myReader = new Scanner(myObj);
82
83              // Start reading the file line by line
84              while (myReader.hasNextLine()) {
85                  //Safing the data of the line
86                  String data = myReader.nextLine();
87
88                  // Display information about the graph
89                  if (data.startsWith("--")) {
90                      System.out.println(data);
91                      System.out.println();
92                  }
93
94                  // Check if the line starts with "new graph" to
        create a new matrix for that specific graph
95                  if (data.startsWith("new graph")) {
96
97                      // Create an Adjancecy List
98                      AdjacencyList adjacencyList = new AdjacencyList
        ();
99
100                     // Iterate over the nex few lines to count the
        vertices
101                     String tempString = myReader.nextLine();
102                     //Check if the graph starts at vertax zero
        which means it is at the ground level
103                     String[] stringParts = tempString.split(" ");
104                     Boolean vertexStartsZero = isGroundLevel(
        stringParts[stringParts.length - 1]);
105                     // Start counting the vertices to to create the
```

```
        Matrix later
106                 int countVertices = 0;
107
108                 while (myReader.hasNextLine() & tempString.
    startsWith("add vertex")) {
109                     //Add a vertex to the AdjacencyList
110                     String line = tempString;
111                     String[] splitString = line.split(" ");
112                     int graphVertex = Integer.parseInt(
    splitString[stringParts.length - 1]);
113                     adjacencyList.arrayList.add(new LinkedList(
    graphVertex));
114
115                     //Increate the vertex count and move the
    scanner to the next line
116                     countVertices++;
117                     tempString = myReader.nextLine();
118                 }
119
120                 // Create the matrix with the number of
    vertices (countVertices X countVertices)
121
122                 if (!vertexStartsZero) {
123                     //New Matrix Object
124                     Matrix matrixGraph = new Matrix(
    countVertices);
125                     //Create Matrix
126                     matrixGraph.createMatrix();
127
128                     // iterate over the edges and add edges to
    the matrix and the adjacencyList
129                     while (myReader.hasNextLine() & tempString.
    startsWith("add edge")) {
130                         //split the string at spaces and get
    the vertices from the string
131                         int[] edgeVertices = toFilterString(
    tempString);
132                         // Add edge to the matrix
133                         matrixGraph.addEdge(edgeVertices[0],
    edgeVertices[1]);
134                         //Add edge to the adjacencyList
135                         adjacencyList.arrayList.get(
    edgeVertices[0]-1).inputEdge(edgeVertices[1]);
136                         //Move the scanner to the next line
137                         tempString = myReader.nextLine();
138                     }
139                     //Printing the graph in both forms (Matrix
    & AdjacencyList)
140                     matrixGraph.displayMatrix();
141                     adjacencyList.displayAdjacencyList();
142
143                 } else {
144                     countVertices++;
145                     // Initialize matrix object
146                     Matrix matrixGraph = new Matrix(
    countVertices);
147                     // Create actual matrix that starts at zero
```

```
148                         matrixGraph.createGroundLevelMatrix();
149
150                         // iterate over the edges and add edges to
    the matrix and the adjacencyList
151                         while (myReader.hasNextLine() & tempString.
    startsWith("add edge")) {
152                             //split the string at spaces and get
    the vertices from the string
153                             int[] edgeVertices = toFilterString(
    tempString);
154                             // Add edge to the matrix
155                             matrixGraph.addEdge(edgeVertices[0],
    edgeVertices[1]);
156                             //Add edge to the adjacencyList
157                             adjacencyList.arrayList.get(
    edgeVertices[0]).inputEdge(edgeVertices[1]);
158                             //Move the scanner to the next line
159                             tempString = myReader.nextLine();
160                         }
161                         //Printing the graph in both forms (Matrix
    & AdjacencyList)
162                         matrixGraph.displayMatrix();
163                         adjacencyList.displayAdjacencyList();
164                     }
165                 }
166
167             }
168             myReader.close();
169         } catch (Exception e) {
170             System.out.println("An error occurred.");
171             e.printStackTrace();
172         }
173
174
175
176
177
178         // Creating an object of BST class from module BST.java in
    another file
179         BST binarySearchTree = new BST();
180         // Creating an ArrayList of String object to store lines of
     strings
181         ArrayList<String> magicItems = new ArrayList<String>();
182
183         // Creating another ArrayList of String Object to store the
     magic items we are searcing
184         ArrayList<String> magicItemsFind = new ArrayList<String>();
185
186         // creating a new file object
187         File f = new File("magicitems.txt");
188
189         // Creating another file for accessing the second magic
    items file
190         File f2 = new File("magicitems-find-in-bst.txt");
191         // Call the filtering function to filter the magic items
    and append it to the arrayList
192         filterFile(magicItems, f);
```

```
193         // Call the filtering function to filter the "magicitems-
    find-in-bst.txt" and append it to the arrayList
194         filterFile(magicItemsFind, f2);
195 
196         // Converting the String ArrayList of magicitems to String
    Array
197         String[] magicItemsArray = toArrayOfString(magicItems);
198         // Converting the magicItemsFind ArrayList to String Array
199         String[] magicItemsFindArray = toArrayOfString(
    magicItemsFind);
200         //Inserting magicitems in the BST
201         populateTree(magicItemsArray, binarySearchTree);
202         System.out.println("
    ----------------------------------------");
203         System.out.println("Printing the elements in the tree in In
    -Order-Traversal");
204         // Printing magic items in In-Order-Traversals from the BST
205         binarySearchTree.inorder(binarySearchTree.root);
206         // Searching the selected magic items from the BST and
    printing their path
207         searchBST(magicItemsFindArray, binarySearchTree);
208         System.out.println("
    ----------------------------------------");
209         // Calculating the Average comparison count for the
    searched elements
210         AvgComparisonCount(binarySearchTree);
211     }
212 
213 }
214 
215 ----------------------End of the Main Class-----------------------
```

3 Binary Search Tree Class

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

```
1  import java.util.ArrayList; // importing the ArrayList class to
       store elements
2  
3  public class BST {
4  
5      // This Node class will be used to store elements
6      public class Node {
7          String data;
8          Node left;
9          Node right;
10 
11         Node(String element) {
12             this.data = element;
13             this.left = null;
14             this.right = null;
15         }
16     }
17 
18     // Initializing the root node of the BST
19     Node root;
```

```
20
21      // Constructor for the BST where the tree is empty
22      BST() {
23          root = null;
24      }
25
26      public void insert(String string) {
27
28          // Creating an CharacterArrayList to store the path of each
         node
29          ArrayList<Character> pathArray = new ArrayList<Character>()
        ;
30
31          Node newNode = new Node(string);
32
33          if (this.root == null) {
34              this.root = newNode;
35          } else {
36
37              // Temporary node for storing the root of the tree
38              Node temp = this.root;
39              // Keeping truck of the parent node of the position
        where the new node will be inserted
40              Node refPositionOfNewParentNode = null;
41
42              // Find the position of the new Node
43              while (temp != null) {
44                  refPositionOfNewParentNode = temp;
45                  if (newNode.data.compareTo(temp.data) < 0) {
46                      temp = temp.left;
47                      pathArray.add('L');
48
49                  } else if (newNode.data.compareTo(temp.data) > 0) {
50                      temp = temp.right;
51                      pathArray.add('R');
52                  } else {
53                      return;
54                  }
55              }
56              // Printing out the path of the Node in a
        CharacterArrayList
57              System.out.println(pathArray);
58
59              // Check if the new node is greater or less than it's
        parent and insert newNode in it's correct positioin
60              if (newNode.data.compareTo(refPositionOfNewParentNode.
        data) < 0) {
61                  refPositionOfNewParentNode.left = newNode;
62              } else {
63                  refPositionOfNewParentNode.right = newNode;
64              }
65          }
66      }
67
68      // Printing elements in In-order-traversal (left, root, right)
        using Recursiosn
69      public void inorder(Node root) {
```

```
70          if (root != null) {
71              inorder(root.left);
72              System.out.println(root.data);
73              inorder(root.right);
74          }
75      }
76
77      // Storing the number of comparison of each look up in this
        arrayList
78      ArrayList<Integer> totalComparisonCount = new ArrayList<Integer
        >();
79
80      // This function returns the average comparisons of each look
        up
81      public Double avgSearchComparison(ArrayList<Integer>
        comparisonCounting) {
82
83          int sum = 0;
84          for (int i = 0; i < comparisonCounting.size(); i++) {
85              sum += comparisonCounting.get(i);
86          }
87          double avg = sum / comparisonCounting.size();
88
89          return avg;
90      }
91
92      // Searching elements in the BST and retrun their path in a
        characterArrayListx
93      public void search(String element) {
94
95          int comparisonsCount = 0;
96
97          // Creating an CharacterArrayList to store the path of each
         node
98          ArrayList<Character> pathArray2 = new ArrayList<Character
        >();
99
100         Node treeRoot = this.root;
101
102         while (treeRoot != null) {
103             if (element.compareTo(treeRoot.data) < 0) {
104                 treeRoot = treeRoot.left;
105                 pathArray2.add('L');
106                 comparisonsCount++;
107             } else if (element.compareTo(treeRoot.data) > 0) {
108                 treeRoot = treeRoot.right;
109                 pathArray2.add('R');
110                 comparisonsCount++;
111             } else {
112                 //Printing the look up path
113                 System.out.println("Look-up-Path: " + pathArray2);
114                 // Printing the number of comparison for each look
        up
115                 System.out.println("Number of comparisons: " +
        comparisonsCount);
116                 break;
117             }
```

```
118         }
119         // Check if the element is not in the tree
120         if (treeRoot == null) {
121             System.out.println(element + " : is not in the tree");
122         }
123 
124         //Adding the comparison count to the totalComparisonCount
    arrayList to calculate the average
125         totalComparisonCount.add(comparisonsCount);
126     }
127 }
```

4 LinkedList Class

Below is the LinkedList Class implementation from the Linkedlist module

```
1 
2 public class LinkedList {
3     // This Node class will be used to store elements in the linked
       list
4     public class Node {
5         int data;
6         Node next;
7 
8         Node(int element) {
9             this.data = element;
10            this.next = null;
11        }
12    }
13    // Initialize the root of the linkedlist
14    Node head;
15    Node tail;
16    int size = 1;
17 
18    // Constructor for the Linked List
19 
20    LinkedList(int vertax) {
21        this.head = new Node(vertax);
22        this.tail = null;
23    }
24 
25    /**
26     * this Function returns the length of the linked List
27     */
28    public int len(){
29        return this.size;
30    }
31 
32    /**
33     * This function checks if the linked list is empty
34     */
35    public boolean isEmpty() {
36        if (len() > 1) {
37            return false;
38        } else {
39            return true;
```

```
        }
    }
    /**
     * 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

```
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 method will print everything in the Adjacency List
    public void displayAdjacencyList() {
        for (int i = 0; i < arrayList.size(); i++) {
            arrayList.get(i).print();
        }
        System.out.println();
    }
}
//-------------------Go to the Next Page for Results---------------
```

6 Results

The worst time complexity of insertion function 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)