```java
package com.Assign_2;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.LinkedList;
import java.util.Scanner;

//Sebastian Wood 6664189
//Intellij

//Allows the creation and testing of a tree structure
public class VirusTree {
    //Root of the tree
    Node root;

    //Creates a tree and then tests various traversals of that tree
    public VirusTree() throws FileNotFoundException {
        //Scanner which allows input from user
        Scanner scan = new Scanner(System.in);
        //Request for input from user
        System.out.println("Input file name");
        //tree_of_virus_input.txt
        //Reads user input
        String fileName = scan.nextLine();
        //Creates file from user input
        File file = new File(fileName);
        //Scanner which allows input from file
        scan = new Scanner(file);

        //Stores input from file
        LinkedList<String> words;

        //Current node
        Node current;
        //Reads from file to create tree, Stops when file is empty
        while(scan.hasNext()) {
            //The next line of the file
            String line = scan.nextLine();
            //Turns the line from file into separate values
            words = stringMaker(line);

```

```
42              //If this is the first time through the
   wile loop
43              if(root == null) {
44                  //Create and display root with first
   value from file
45                  root = new Node(null, words.
   removeFirst(), null);
46                  root.displayNode(true);
47                  //Sets the roots first child as
   second value from file
48                  root.firstChild = new Node(null,
   words.removeFirst(), null);
49                  //Sets current for next part
50                  current = root.firstChild;
51                  current.displayNode(false);
52              }
53              //If this is not the first time running
   the while loop
54              else {
55                  //Sets current to parent
56                  current = searchTree(root, words.
   removeFirst());
57                  //Checks if there if the parent
   already has a child
58                  if(current.firstChild != null) {
59                      current.displayNode(true);
60                      //Sets current to first child
61                      current = current.firstChild;
62                      current.displayNode(false);
63                      //Sets current to farthest
   sibling for next part
64                      while(current.nextSibling != null
   ) {
65                          current = current.nextSibling
   ;
66                          current.displayNode(false);
67                      }
68                  }
69                  //If the parent has no children
70                  else {
71                      //Create first child and set
   current to it for next part
72                      current.firstChild = new Node(
   null, words.removeFirst(), null);
```

```
 73                      current = current.firstChild;
 74                      current.displayNode(false);
 75                  }
 76              }
 77              //Sets the siblings of current. Keeps
    going until there are no values left from the line
 78              while(!words.isEmpty()) {
 79                  current.nextSibling = new Node(null
    , words.removeFirst(), null);
 80                  current = current.nextSibling;
 81                  current.displayNode(false);
 82              }
 83              System.out.println();
 84          }
 85          //Testing of the traversal methods
 86          System.out.println("Breadth First:");
 87          breadthFirst(root);
 88          System.out.println();
 89
 90          System.out.println("Pre-Order");
 91          preOrder(root);
 92          System.out.println();
 93
 94          System.out.println("Post-Order");
 95          postOrder(root);
 96          System.out.println();
 97
 98          //Testing of the height method
 99          System.out.println("Height:");
100          System.out.println(getHeight(root));
101          System.out.println();
102
103          //Testing of the length between method
104          //Something to note, The last two test
    should result in 2 and 3 respectively. I don't know
    why the pdf says 1 and 2
105          //By the logic put out by the first two
    tests that should be wrong
106          lengthBetween(root, "Ebola virus", "Bombali
    virus");
107          System.out.println();
108
109          lengthBetween(root, "Ebola virus", "Marburg
    virus");
```

```
110         System.out.println();
111
112         lengthBetween(root, "HCoV-OC43", "Hcov-229E"
    );
113         System.out.println();
114
115         lengthBetween(root, "SARS-CoV", "Zika virus"
    );
116         System.out.println();
117     }
118
119     //Searches the tree iteratively for a Node with
    a specific key, Searches Breadth First
120     private Node searchTree(Node T, String key) {
121         //Queue that allows searching nodes in
    specific order
122         LinkedList<Node> queue = new LinkedList<>();
123
124         //Adds root to queue
125         queue.add(T);
126         //Loops while there are still nodes left
127         while(!queue.isEmpty()) {
128             //Removes current node
129             Node temp = queue.removeFirst();
130             //Checks if node is null
131             if(temp != null) {
132                 //Algorithm for breadth first
    searching of a tree
133                 if (temp.item.equals(key)) return
    temp;
134                 if (temp.firstChild != null)
135                     queue.addLast(temp.firstChild);
136                 if (temp.nextSibling != null)
137                     queue.add(temp.nextSibling);
138             }
139         }
140         //Means Node with key was not found
141         return null;
142     }
143
144     //Takes the raw input from input file and turns
    it into separate values
145     private LinkedList<String> stringMaker(String
    line) {
```

```
146         //Creates list for storage
147         LinkedList<String> words = new LinkedList
    <>();
148         //Turns line into char array
149         char[] array = line.toCharArray();
150         //Makes a builder for the string
151         StringBuilder builder = new StringBuilder();
152 
153         //Creates word from char array. Separates
    each word based on commas
154         for (char c : array) {
155             if (c == ',') {
156                 words.addLast(builder.toString());
157                 builder = new StringBuilder();
158             } else {
159                 builder.append(c);
160             }
161         }
162 
163         //Adds last word to storage
164         words.addLast(builder.toString());
165         builder = new StringBuilder();
166 
167         //Returns result
168         return words;
169     }
170 
171     //Gets the height of a node
172     private int getHeight(Node T) {
173         //Checks if the end of a branch has been
    reached and adjusts final result
174         if(T == null) {
175             return -1;
176         }
177         //Checks if T has a sibling
178         else if(T.nextSibling != null) {
179             //Gets the max value between the child
    subtree and sibling subtree I.E. left and right
    subtree
180             // Adds 1 to left subtree because it is
    traversing down
181             return Math.max(1 + getHeight(T.
    firstChild), getHeight(T.nextSibling));
182         }
```

```
183         //If T is the last sibling
184         else {
185             return 1 + getHeight(T.firstChild);
186         }
187     }
188     //The time complexity of getHeight is O(n) this
    is because each node of the tree is visited and
    during
189     //each visit only one operation is performed
190 
191     //Finds the length between two nodes from its
    ancestor. This assumes that the two nodes
192     //are the leaves of the tree and the tree is
    complete
193     private Node lengthBetween(Node T, String key1,
    String key2) {
194         //Check for the end of a branch
195         if(T == null) return null;
196 
197         //Checks if the two key nodes are both
    direct siblings of current node and outputs result
198         if(isSiblings(T.firstChild, key1, key2)) {
199             System.out.println("The distance between
     " + key1 + " and " +
200                     key2 + " is " + getHeight(T) +
    ". They have common ancestor " + T.item);
201             //Returns the current Node
202             return T;
203         }
204         //Checks to see if this node is a key node
    and returns it if it is
205         else {
206             if (T.item.equals(key1)) {
207                 return T;
208             }
209             if (T.item.equals(key2)) {
210                 return T;
211             }
212         }
213 
214         //The recursive call. The results are stored
     in variables
215         Node node1 = lengthBetween(T.firstChild,
    key1, key2);
```

```
216         Node node2 = lengthBetween(T.nextSibling,
    key1, key2);
217 
218         //Because of the nature of a general tree
    the only node that matters when checking if this
    node is the lowest common ancestor is the
219         //result from the child node. This is
    because if the key node comes from a sibling then
    the current node cannot be the common ancestor
220         //and is in fact only a part of the subtree
    whose root is the lowest common ancestor.
221         //Check if node 1 is null, which means
    nothing was found from the child subtree
222         if(node1 != null) {
223             //Checks if the two key nodes are
    contained within the current nodes subtree. Note
    that each key is checked twice but on
224             //different subtrees each time
225             if (searchTree(node1.firstChild, key1
    ) != null && searchTree(node1.nextSibling, key2) !=
    null ||
226                     searchTree(node1.firstChild,
    key2) != null && searchTree(node1.nextSibling, key1
    ) != null) {
227                 //Output result
228                 System.out.println("The distance
    between " + key1 + " and " +
229                         key2 + " is " + getHeight(T
    ) + ". They have common ancestor " + T.item);
230                 //Returns current node up to stop
    another node from satisfying previous if condition
231                 return T;
232             }
233         }
234 
235         //Check if both key nodes were found and
    returns current
236         if(node1 != null & node2 != null) return T;
237         //Otherwise return non-null node
238         if(node1 != null) return node1;
239         if(node2 != null) return node2;
240         //Key node not found
241         return null;
242     }
```

```
243
244     //Checks if key nodes are siblings
245     private boolean isSiblings(Node T, String key1,
    String key2) {
246         //Storage of results
247         boolean result1 = false;
248         boolean result2 = false;
249
250         //Sets pointer
251         Node curr = T;
252         //Checks each sibling to see if they are a
    key node. then continues down the list
253         while(curr != null) {
254             if(curr.item.equals(key1)) {
255                 result1 = true;
256             }
257             if(curr.item.equals(key2)) {
258                 result2 = true;
259             }
260             curr = curr.nextSibling;
261         }
262
263         //If both results are true then both key
    nodes are siblings
264         if(result1 && result2) return true;
265         return false;
266     }
267
268     //Iteratively traverses tree in a breath first
    traversal
269     private void breadthFirst(Node T) {
270         //Queue that allows traversal of nodes in
    specific order
271         LinkedList<Node> queue = new LinkedList<>();
272
273         //Adds root to queue
274         queue.add(T);
275         //Loops while there are still nodes left
276         while(!queue.isEmpty()) {
277             Node temp = queue.removeFirst();
278             //Visits node
279             temp.displayNode();
280             System.out.println();
281             //Adds subtrees if they exist
```

```
282             if(temp.firstChild != null)
283                 queue.addLast(temp.firstChild);
284             if(temp.nextSibling != null)
285                 queue.addFirst(temp.nextSibling);
286         }
287     }
288 
289     //Recursively traverses tree in a pre-order
    traversal
290     private void preOrder(Node T) {
291         //If end of branch was reached
292         if(T == null) return;
293         else {
294             //Visit node
295             T.displayNode();
296             System.out.println();
297             //recursive call to subtrees
298             preOrder(T.firstChild);
299             preOrder(T.nextSibling);
300         }
301     }
302 
303     //Recursively traverses tree in a post-order
    traversal
304     private void postOrder(Node T) {
305         //If end of branch was reached
306         if(T == null) return;
307         else {
308             //recursive call on left subtree
309             postOrder(T.firstChild);
310             //If T has  sibling then visit node and
    then recursively call teh right subtree
311             if(T.nextSibling != null) {
312                 T.displayNode();
313                 System.out.println();
314                 postOrder(T.nextSibling);
315             }
316             //If not then just visit node
317             else {
318                 T.displayNode();
319                 System.out.println();
320             }
321 
322         }
```

```
323     }
324 
325     //Starts the program by making a new VirusTree.
326     public static void main ( String[] args ) throws
     java.io.FileNotFoundException { VirusTree d = new
    VirusTree(); }
327 }
328 
```