```java
package com.company;

/*
 *@author Nico McFarlane
 *@Student ID 7001811
 *@version 1.0 (10/18/2021)
*/

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;
public class Main {

    File file = new File("src/com/company/volunteer_data.txt");//text file
    Scanner scan = new Scanner(file);
    FileWriter write = new FileWriter("src/com/company/results.txt");
    Node root;


    public Main() throws IOException {
        createBST();
    }

    /**
     * This method will read data from the input text file and add the data
to the output file.
     * The data read should contain all information of a volunteer (ID,
name, address, contact).
     * @throws IOException
     */
    private void createBST() throws IOException {
root = null;
scan.nextLine();
        while(scan.hasNextLine()&& scan.hasNextInt()){
            int ID = scan.nextInt();
            String name = scan.next();
            String address = scan.next() + " " + scan.next() + " " +
scan.next();
            String contact = scan.nextLine();

            insert(ID,name,address,contact);
           // DeleteNode(ID,name,address,contact);
            DeleteNode(24, name,address,contact);
            DeleteNode(71, name,address,contact);
            DeleteNode(60, name,address,contact);
        }
        write.write("preOrder");
        write.write("\n");
```

```
        preOrderTraverse(root);
            write.write("\n");
            System.out.println();
            write.write("inOrder");
            write.write("\n");
            inOrderTraverse(root);
            write.write("\n");
            System.out.println();
            write.write("postOrder");
            write.write("\n");
            postOrderTraverse(root);
            write.write("\n");
            System.out.println();
            System.out.println(height(root));
            write.write("Height is " + height(root));
            write.write("\n");
            System.out.println(depth(root));
            write.write("Depth is " + depth(root));

            write.close();
        }

        /**
         * This method will delete a node from the BST however, it will delete
the correct nodes by the volunteer ID's.
         * For example, 24, 60 and 71 will delete all (ID, name, address and
contact) data
         * @param ID
         * @param name
         * @param address
         * @param contact
         */
    private void DeleteNode(int ID, String name, String address, String
contact){
            root = delete(root,ID, name,address,contact);
    }
        private void insert (int ID, String name, String address, String
contact){
            root = insert1(root, ID, name, address, contact);
        }

        /**
         * This method uses recursion to insert(add) the data into the BST (ID,
name, address, contact) depending on
         * the conditions.
         * @param root
         * @param ID
         * @param name
         * @param address
```

```
         * @param contact
         * @return
         */
        Node insert1(Node root, int ID, String name, String address, String
contact){
            if(root == null){
                root = new Node(ID, name, address, contact);
                return root;
            }
            if (ID < root.ID)
                root.left = insert1(root.left,ID,name,address,contact);
            else if ( ID > root.ID)
                root.right = insert1(root.right,ID,name,address,contact);
            return root;
        }

        /**
         * This method traverses through the BST data, visiting the root, then
traverses to the left then right.
         * It will print the results in that order.
         * preOrder does not change.
         * @param node
         * @throws IOException
         */
        public void preOrderTraverse(Node node) throws IOException {
            if(node == null) // base case
                return;
            else // recursive case - 2 recursive calls
            {
                System.out.println(node.ID + " " + node.name + " " +
node.address + node.contact);
                write.write(node.ID + " " + node.name + " " + node.address +
node.contact);
                write.write("\n");
                preOrderTraverse(node.left);
                preOrderTraverse(node.right);

            }

        }

        /**
         *  This method traverses the left side, then right then visits the
root and puts them in that order.
         *  It will print the results in that order.
         * @param node
         * @throws IOException
         */
        public void postOrderTraverse(Node node) throws IOException {
```

```
            if(node == null)
                return;
            else
            {
                postOrderTraverse(node.left);
                postOrderTraverse(node.right);
                System.out.println(node.ID + " " + node.name + " " +
node.address + node.contact);
                write.write(node.ID + " " + node.name + " " + node.address +
node.contact);
                write.write("\n");
            }
        }

        /**
         * This method traverses the left side, then visits root then traverses
right as well as print the results in
         * that order.
         * @param node
         * @throws IOException
         */
        public void inOrderTraverse(Node node) throws IOException {
            if(node == null)
                return;
            else
            {
                inOrderTraverse(node.left);
                System.out.println(node.ID + " " + node.name + " " +
node.address + node.contact);
                write.write(node.ID + " " + node.name + " " + node.address +
node.contact);
                write.write("\n");
                inOrderTraverse(node.right);

            }
        }

        /**
         * This method uses recursion to check the left and right side of the
BST and returns the height of the BST as
         * an int.
         * @param node
         * @return
         */
         int height(Node node)
        {
            if(node == null){
                return -1; }
            else
```

```
                return 1 +
                        Math.max(height(node.left),
                                height(node.right));
        }

        /**
         * This uses recursion to check left side of the BST and checks the
right side to get the deepest
         * node in the BST.
         * It returns a int value.
         *
         * @param node
         * @return
         */
    int depth(Node node) {
        if (node == null) {
            return (-1); // an empty tree  has height -1
        } else {
            // compute the depth of each subtree
            int leftDepth = depth(node.left);
            int rightDepth = depth(node.right);
            // use the larger one
            if (leftDepth > rightDepth)
                return (leftDepth + 1);
            else
                return (rightDepth + 1);
        }
    }

        /**
         * This method deletes the nodes from the BST depending on the
volunteerID (ex. 24)
         * Deletes and returns the root of the new tree.
         * It will remove (ID, name, address and contact)
         * @param node
         * @param ID
         * @param name
         * @param address
         * @param contact
         * @return
         */
        Node delete(Node node, int ID, String name, String address, String
contact){
    // delete and return the root of the new tree
            if (node == null) // null tree, or not found
                return node;
            if (ID < node.ID)
                node.left = delete(node.left,ID,  name,  address,  contact);
//del from left, update
```

```
            else if (ID > node.ID)
                node.right = delete(node.right, ID,  name,  address,
contact);//del from right,update
            else
                if (node.left !=null && node.right != null )
                {
                    node.ID = findMin(node.right).ID;
                    node.right = delete(node.right,ID,  name,  address,
contact);
                }
                else
                    node = (node.left!=null) ? node.left : node.right;
            return node;
        }

        /**
         * This method will fine the minimum value in a subtree.
         * @param node
         * @return
         */
        private Node findMin(Node node){
            if (node == null)
                return null;
            else if(node.left == null)
                return node;
            return findMin(root.left);
        }

//         /**
//          * This method SHOULD obtain the depth of the node from a specific
ID given.
//          * I've been having issues with locating the depth of a specific
node. I check the left side and right side of
//          * the search tree and increment the depth until its found
//          * @param node
//          * @param ID
//          * @return
//          */
//         public int depth2(Node node, int ID) {
//             int depth2 = 0;
//             while (root != node.ID)
//                 root = root.left
//             root = root.right
//             depth2++;
//             if (root == null) {
//                 return 0;
//             }
//             return depth2;
//         }
```

```
        public static void main(String[] args) throws IOException {
            Main Tree = new Main();
        }
    }
```

NODE CLASS

```
package com.company;
/*
*@author Nico McFarlane
*@Student ID 7001811
*@version 1.0 (10/18/2021)
*/
public class Node {
    int ID;
    String name;
    String address;
    String contact;

    Node left;
    Node right;

    Node(int ID, String name, String address, String contact){
        this.ID = ID;
        this.name = name;
        this.address = address;
        this.contact = contact;
    }
}
```