```java
package asg_2;

import BasicIO.ASCIIDataFile;

public class FamilyTree {
    /*
     * Author: Trevor Vanderee
     * ID: 5877022
     * Assignment 2
     * Family Tree
     */
    private ASCIIDataFile in;
    private int year, nChild, nAnChecks,cur1, cur2;
    private String name, aName1, aName2;
    private Node tree;
    private Stack qq1,qq2;
    private Node[] NL1,NL2;

    public FamilyTree( ){
        in = new ASCIIDataFile();
        name = in.readString();
        year = in.readInt();
        nChild = in.readInt();
        System.out.println();
        tree = new Node(name,year,null,null);
        createTree(tree,nChild,0);

        System.out.println("\n PreOrder");
        preorderPrint(tree);

        System.out.println("\n PostOrder");
        postorderPrint(tree);

        System.out.println("\n Breadth-First");
        breadthPrint(tree);


        System.out.println("\n Ancestor Check");
        nAnChecks = in.readInt();
        for(int i =0; i < nAnChecks; i++){
            aName1 = in.readString();
            aName2 = in.readString();
            qq1 = new Stack();
            qq2 = new Stack();
            findAncestors(aName1,aName2);
        }
        in.close();
        System.exit(2);
    }

    /**
     * This method recursively creates a tree by reading data from an ASCII
     *  Data file.
     * @param Node t: The root node for the tree to be created on
     * @param int children: The amount of children that the given parent has
     * @param int siblings: The amount of  siblings that a node has
     */
    private void createTree(Node t, int children,int siblings){
        if(children!=0){
            name = in.readString();
            year = in.readInt();
            nChild = in.readInt();
            t.child = new Node(name,year,null,null);
            createTree(t.child,nChild,children-1);
        }
        if(siblings!=0){
```

```
            name = in.readString();
            year = in.readInt();
            nChild = in.readInt();
            t.sibling = new Node(name,year,null,null);
            createTree(t.sibling,nChild,siblings-1);
            }
    }//createTree

    /**
     * This method is a recursive function that finds
     *  the pre order print of a tree
     * @param Node pre: The node to be visited
     */
    private void preorderPrint(Node pre){
        //Handles Empty Tree
        if(pre==null){
            System.out.println("Tree is Empty");
            return;
        }
        System.out.println(pre.name +", "+ pre.year);
        if(!(pre.child == null)){
            preorderPrint(pre.child);
        }
        if(!(pre.sibling == null)){
            preorderPrint(pre.sibling);
        }

    }//preorderPrint

    /**
     * This method is a recursive function that finds
     *  the post order print of a tree
     * @param Node post: The node to be visited
     */
    private void postorderPrint(Node post){
        //Handles Empty Tree
        if(post==null){
            System.out.println("Tree is Empty");
            return;
        }
        if(!(post.sibling==null)){
            postorderPrint(post.sibling);
        }
        if(!(post.child==null)){
            postorderPrint(post.child);
        }
        System.out.println(post.name + ", " + post.year);
    }//postorderPrint

    /**
     * This method prints all nodes from left to right, top to bottom.
     * @param Node brt: The Node to be visited
     */
    private void breadthPrint(Node brt){
        Node p = brt;
        //Handles Empty Tree
        if(brt==null){
            System.out.println("Tree is Empty");
            return;
        }
        Queue qu = new Queue();
        while(p!=null){
            System.out.println(p.name +","+ p.year);
            if(p.child!=null){
                qu.enqueue(p.child);
            }
```

```
            if(p.sibling !=null){
                p=p.sibling;
            }else if(!qu.empty()){
                p = qu.dequeue();
            }else{
                p = null;
            }
        }

    }//breadthPrint

    /**
     * This method finds common ancestors between two people in the family tree
     * @param String name1: The first Name given by the ASCIIData File
     * @param String name2: The second name given by the ASCIIData File
     */
    private void findAncestors(String name1, String name2){
        Stack st1 = new Stack();
        Stack st2 = new Stack();
        cur1 = 0;
        cur2 = 0;
        //Handles Empty Tree
        if(tree==null){
            System.out.println("Tree is Empty");
            return;
        }
        //Creates a stack of all Nodes with "name1"
        loadNames(name1,1,tree);
        //Array used as they are reusable
        if(qq1.depth()==0){
            System.out.println("There is noone by the name "+ name1);
            return;
        }
        NL1 = new Node[qq1.depth()];
        for(int i= qq1.depth()-1; i>=0;i--){
            NL1[i]=qq1.pop();
        }
        loadNames(name2,2,tree);
        if(qq2.depth()==0){
            System.out.println("There is noone by the name "+ name2);
            return;
        }
        NL2 = new Node[qq2.depth()];
        for(int i = qq2.depth()-1; i>=0;i--){
            NL2[i]=qq2.pop();
        }
        while(true){
            //used to ensure the method does not reach out of the array.
            if(cur1==NL1.length)
                break;
            st1= getStacks(name1,1,NL1[cur1].year);
            if(cur2==NL2.length)
                break;
            st2 = getStacks(name2,2,NL2[cur2].year);
            compareStacks(st1,st2);
        }
    }//findAncestors

    /**
     * This method finds all the ancestors of a given Node.
     * @param String names: the name on which the Ancestor Stack is built
     * @param int r: indicates whether the function is finding the first or second stack
     * @param int xYear: indicates the current year if there is multiple Nodes with the same name
     * @return Stack trace: the completed ancestor stack
     */
    private  Stack getStacks(String names, int r, int xYear){
```

```
        Node q;
        Stack next = new Stack();
        Stack trace = new Stack();
        q = tree;
        while(q!=null){
            trace.push(q);

            if(q.sibling!=null){
                next.push(q);
            }
            if(q.name.equals(names) && xYear==q.year){
                    if(r==2){
                        cur2++;
                    }
                    return trace;
            }else if(q.child != null){
                    q = q.child;
            }else{
                q= next.pop();
                while(q.year != trace.top().year){
                        trace.pop();
                }
                trace.pop();
                q= q.sibling;
            }

            //If traversal reaches end of tree this brings it to the next set of name1
            if(q.child== null && q.sibling == null && next.empty()){
                if(r==2){
                    cur1++;
                    cur2=0;
                }
            }

        }
        return null;
    }//getStacks

    /**
     * This method loads all the Nodes of  a given name into a stack
     * @param String n1: The set of Names to be Loaded into a stack
     * @param int r: Indicates either name1 or name2 for the stacks
     * @param Node in: the current Node being examined
     */
    private void loadNames(String n1, int r,Node in){
        if(n1.equals(in.name)){
            if(r==1){
                qq1.push(in);
            }else if(r==2){
                qq2.push(in);
            }
        }
        if(!(in.child == null)){
            loadNames(n1,r,in.child);
        }
        if(!(in.sibling == null)){
            loadNames(n1,r,in.sibling);
        }
    }//loadNames

    /**
     * This method takes two Stacks and finds the point at
     *  which they have  common ancestors.
     * @param Stack a: The first Stack given
     * @param Stack b: The second Stack given
     */
```

```
    private void compareStacks(Stack a, Stack b){
        String pName1, pName2, out1, out2;
        int pYear1, pYear2;
        int aDepth, bDepth;
        Node ax,bx;
        aDepth = a.depth();
        bDepth = b.depth();
        pName1 = a.top().name;
        pName2 = b.top().name;
        pYear1 = a.top().year;
        pYear2 = b.top().year;
        out1 = "("+ pName1 + ", " + pYear1 + ")";
        out2 = "("+ pName2 + ", " + pYear2 + ")";
        while(aDepth != bDepth){
            if(aDepth>bDepth){
                a.pop();
            }else if(aDepth<bDepth){
                b.pop();
            }
            aDepth = a.depth();
            bDepth = b.depth();

        }
        while(!a.empty()){
            ax=a.pop();
            bx=b.pop();
            if(ax.name.equals(bx.name)&& ax.year == bx.year){
                a.push(ax);
                printStack(a,out1,out2 );
                break;
            }
        }
    }//compareStacks

    /**
     * This method takes a Stack and prints out it's contents
     * @param Stack out: The Stack to be printed
     * @param String s1: The Name of the first person
     * @param String s2: The Name of the second person
     */
    private void printStack(Stack out, String s1, String s2){
        String output = "";
        System.out.println("\n Common ancestors of " + s1 + " and " + s2 + ":");
        while(!out.empty()){
            output += "(" + out.top().name + ", "+ out.pop().year +")  ";
        }
        System.out.println(output);
    }//printStack

    public static void main(String[] args){@SuppressWarnings("unused")  FamilyTree F = new
    FamilyTree( );}
}
```