/\*\*

\* Code for Exercise 2E

\* This is the Node class. Nodes make up the binary

\* tree.

\*/

public class Node{

private Node parent;

private Node leftNode;

private Node rightNode;

private Integer value;

public Node(Integer value){

this.value = value;

}

public void setParent(Node parent){

this.parent = parent;

}

public void setLeftNode(Node leftNode){

this.leftNode = leftNode;

}

public void setRightNode(Node rightNode){

this.rightNode = rightNode;

}

public void setValue(Integer value){

this.value = value;

}

public Node getParent(){

return this.parent;

}

public Node getRightNode(){

return this.rightNode;

}

public Node getLeftNode(){

return this.leftNode;

}

public Integer getValue(){

return this.value;

}

public Boolean hasRightNode(){

if(this.getRightNode() == null){

return false;

}

return true;

}

public Boolean hasLeftNode(){

if(this.getLeftNode() == null){

return false;

}

return true;

}

@Override

public String toString(){

return Integer.toString(this.value);

}

}

/\*\*

\* This is the BinaryTree class. The only way to traverse

\* the tree is by getting the root node, and then calling

\* the getLeft() and getRight() functions on that node and

\* its children.

\*/

public class BinaryTree{

private Node root;

public BinaryTree(Node root){

this.root = root;

}

// This method adds a node to the tree. If the node is added to a location

// in the tree where a node already exists, the existing node is replaced.

public void addNode(Node nodeToAdd, Node nodeToAddTo, Boolean addToRight){

nodeToAdd.setParent(nodeToAddTo);

if(addToRight){

if(nodeToAddTo.getRightNode() != null){

nodeToAdd.setRightNode(nodeToAddTo.getRightNode().getRightNode());

nodeToAdd.setLeftNode(nodeToAddTo.getRightNode().getLeftNode());

}

nodeToAddTo.setRightNode(nodeToAdd);

}else{

if(nodeToAddTo.getLeftNode() != null){

nodeToAdd.setRightNode(nodeToAddTo.getLeftNode().getRightNode());

nodeToAdd.setLeftNode(nodeToAddTo.getLeftNode().getLeftNode());

}

nodeToAddTo.setLeftNode(nodeToAdd);

}

}

// This method removes a given node from its parent node. If the

// the node that was removed had children, the leaf that is on the

// left side of the tree is used to replace it.

public Node removeNode(Node nodeToRemove, Node nodeToRemoveFrom){

Node replacement = this.findLeaf(nodeToRemove);

if(nodeToRemoveFrom.getLeftNode() == nodeToRemove){

if(nodeToRemove.hasLeftNode() || nodeToRemove.hasRightNode()){

nodeToRemoveFrom.setLeftNode(replacement);

}else{

nodeToRemoveFrom.setLeftNode(null);

}

swapLeaf(replacement, nodeToRemove);

return nodeToRemove;

}

if(nodeToRemoveFrom.getRightNode() == nodeToRemove){

if(nodeToRemove.hasLeftNode() || nodeToRemove.hasRightNode()){

nodeToRemove.setRightNode(replacement);

}else{

nodeToRemoveFrom.setRightNode(null);

}

swapLeaf(replacement, nodeToRemove);

return nodeToRemove;

}

return null;

}

// This method places a leaf where there is a node in the tree.

// The leaf is then deleted, but reference to the node

// in the tree by the node's parent is not deleted.

private Node swapLeaf(Node leaf, Node nodeToReplace){

if(leaf.getParent().getLeftNode() == leaf){

leaf.getParent().setLeftNode(null);

}

if(leaf.getParent().getRightNode() == leaf){

leaf.getParent().setRightNode(null);

}

leaf.setRightNode(nodeToReplace.getRightNode());

leaf.setLeftNode(nodeToReplace.getLeftNode());

leaf.setParent(nodeToReplace.getParent());

return nodeToReplace;

}

// This method uses recursion to find the leaf on

// the far left side of the tree.

private Node findLeaf(Node node){

if(node.hasLeftNode() == false && node.hasRightNode() == false){

return node;

}

if(node.hasLeftNode()){

return findLeaf(node.getLeftNode());

}

return findLeaf(node.getRightNode());

}

public Node getRoot(){

return this.root;

}

}