COMP 250 Winter 2017 - Solution - Homework #4

1)

```
   // Evaluates a single operation
    static public double apply(String op, double x1, double x2) {
        if (op.equals("add")) return x1+x2;
        if (op.equals("mult")) return x1*x2;
        if (op.equals("minus")) return x1-x2;
        if (op.equals("sin")) return Math.sin(x1);
        if (op.equals("cos")) return Math.cos(x1);
        if (op.equals("exp")) return Math.exp(x1);
        else return 0; // should probably throw an exception instead
    }


    // Returns the value of the expression rooted at a given node
    // when x has a certain value
    double evaluate(double x) {
        if (getLeftChild()==null) {
            // this is a leaf
            if (getValue().equals("x")) return x;
            else return Double.parseDoublegetValue();
        }
        else {
            if (getRightChild()!=null) {
                return apply(getValue(),
                             getLeftChild().evaluate(x),
                             getRightChild().evaluate(x));
            }
            else {
                return apply(getValue(),
                             getLeftChild().evaluate(x),
                             0);
            }
        }
    }

    /* returns the root of a new expression tree representing the derivative of the
       original expression */
    treeNode differentiate() {
        if (getLeftChild()==null) {
            // this is a leaf
            if (getValue().equals("x")) return new treeNode("1");
            else return new treeNode("0");
        }

        if (getValue().equals("add") || getValue().equals("minus")) {
            return new treeNode(getValue(),
                                getLeftChild().differentiate(),
                                getRightChild().differentiate());
        }

        if (getValue().equals("mult")) {
            // build three new nodes: "add" with two children "mult"
            treeNode l,r;
            l=new treeNode("mult", getLeftChild().differentiate(),
getRightChild().deepCopy() );
            r=new treeNode("mult", getLeftChild().deepCopy(),
getRightChild().differentiate() );
            return new treeNode("add", l, r);
        }

        if (getValue().equals("sin")) {
            // build three new nodes: "mult" with two children "cos" and diff.
            treeNode l,r;
            l=new treeNode("cos", getLeftChild().deepCopy(),null);
            r=getLeftChild().differentiate();
```

```
            return new treeNode("mult",l,r);
        }

        if (getValue().equals("cos")) {
            // build three new nodes: "minus" with children 0 and ("mult" with two
children "sin" and diff).
            treeNode r;
            r=new treeNode("mult", new treeNode("sin",getLeftChild().deepCopy(), null),
                           getLeftChild().differentiate());

            return new treeNode("minus", new treeNode("0",null,null), r);
        }
        if (getValue().equals("exp")) {
            return new treeNode("mult", deepCopy(), getLeftChild().differentiate());
        }

        return null;
    }
```

2) (10 points) Binary search trees

Consider a binary search tree that contains *n* nodes with keys 1, 2, 3, ... , *n*.
The shape of the tree depends on the order in which the keys have been inserted in the tree.

a) In what order should the keys be inserted into the binary search tree to obtain a tree with minimal height?

There are several possible answers. One possible answer, assuming that $n=2^k-1$ is:
1/2*(n+1), 1/4*(n+1), 3/4*(n+1), 1/8*(n+1), 3/8(n+1), 5/8(n+1),7/8(n+1)...
This actually fills the binary search tree layer by layer, resulting in a balanced tree of minimal height $\lfloor \log(n) \rfloor$.

b) On the tree obtained in (a), what would be the worst-case running time of a find, insert, or remove operation? Use the big-Oh notation.

Since the worst-case running time for all these operations is O(h), where h is the height of tree, and $h = \lfloor \log(n) \rfloor$, we get a worst-case running time that is O(log(n)).
This means that having a binary search tree that is balanced is a good thing!

c) In what order should the keys be inserted into the binary search tree to obtain a tree with maximal height?

One possible answer is: 1, 2, 3, ..., n-1, n. This would result in a very high tree, where nodes only have a right child. The height of the tree would be n-1.

d) On the tree obtained in (c), what would be the worst-case running time of a find, insert, or remove operation? Use the big-Oh notation.

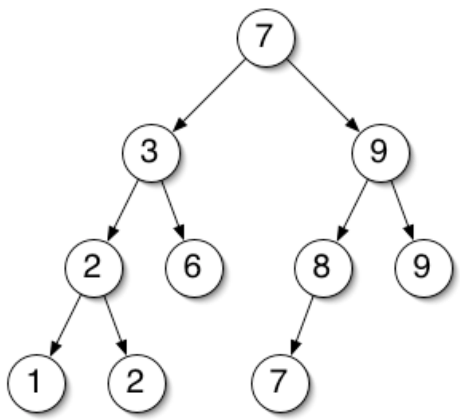
The running time would be O(n), because the height is O(n).

3) **(16 points)** Tree traversals

Consider the following pair of recursive algorithms calling each other to traverse a binary tree.

Algorithm weirdPreOrder(treeNode n)
  if (n != null) **then**
    print n.getValue()
    weirdPreOrder(n.get**Right**Child())
    weirdPostOrder(n.get**Left**Child())

Algorithm weirdPostOrder(treeNode n)
  if (n != null) **then**
    weirdPreOrder(n.get**Right**Child())
    weirdPostOrder(n.get**Left**Child())
    print n.getValue()



a) **(5 points)** Write the output being printed when weirdPreOrder(root) is executed on the following binary tree.

Method executed	Result printed
Pre(7)	
Print 7	7
Pre(9)	
Print 9	9
Pre(9)	
Print 9	9
Pre(null)	
Post(null)	
Post(8)	
Pre(null)	
Post(7)	
Pre(null)	
Post(null)	
Print 7	7
Print 8	8
Post(3)	
Pre(6)	
Print 6	6
Post(null)	

Method executed	Result printed
Pre(null)	
Post(2)	
Pre(2)	
Print 2	2
Pre(null)	
Post(null)	
Post(1)	
Post(null)	
Pre(null)	
Print 1	1
Print 2	2
Print 3	3

Conclusion: The order is : 7 9 9 7 8 6 2 1 2 3

b) **(5 points)** Write the output being printed when weirdPostOrder(root) is executed.

Method executed	Result printed
Post(7)	
Pre(9)	
Print 9	9
Pre(9)	
Print 9	9
Pre(null)	
Post(null)	
Post(8)	
Pre(null)	
Post(7)	
Pre(null)	
Post(null)	
Print 7	7
Print 8	8
Post(3)	
Pre(6)	
Print 6	6
Post(null)	
Pre(null)	
Post(2)	
Pre(2)	
Print 2	2

Pre(null)
Post(null)
Post(1)
Post(null)
Pre(null)
Print 1 1
Print 2 2
Print 3 3
Print 7 7

Conclusion: The order is : 9 9 7 8 6 2 1 2 3 7

c) **(5 points)** Consider the binary tree traversal algorithm below.
Algorithm queueTraversal(treeNode n)
Input: a treeNode n
Output: Prints the value of each node in the binary tree rooted at n
Queue q ← new Queue();
q.enqueue(n);
while (! q.empty()) **do**
    x ← q.dequeue();
    print x.getValue();
    if (x.getLeftChild() != null) **then** q.enqueue(x.getLeftChild());
    if (x.getRightChild() != null) **then** q.enqueue(x.getRightChild());

Question: Write the output being printed when queueTraversal(root) is executed.

This produces a breadth-first search of the tree:
7 3 9 2 6 8 9 1 2 7

d) **(5 points)** Consider the binary tree traversal algorithm below.

Algorithm stackTraversal(treeNode n)
Input: a treeNode n
Output: Prints the value of each node in the binary tree rooted at n
Stack s ← new Stack();
s.push(n);
while (! s.empty()) **do**
    x ← s.pop();

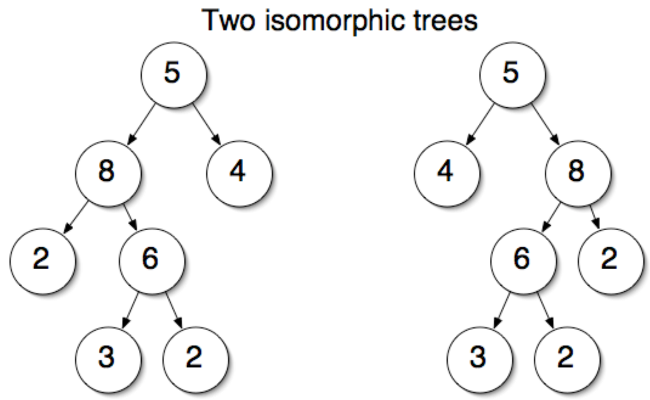
```
print x.getValue();
if (x.getRightChild() != null) then s.push(x.getRightChild());
if (x.getLeftChild() != null) then s.push(x.getLeftChild());
```

Question: Write the output being printed when stackTraversal(root) is executed. This is the equivalent of what traversal method seen previously in class?

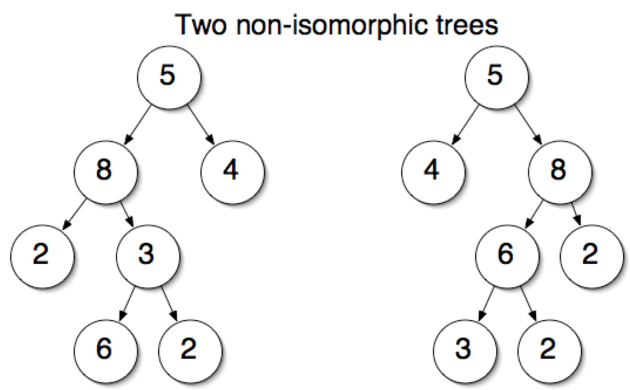
This produces a depth-first search of the tree: 7 3 2 1 2 6 9 8 7 9

1) **(14 points)** Tree isomorphism

Two unordered binary trees A and B are said to be *isomorphic* if, by swapping the left and right subtrees of certain nodes of A, one can obtain a tree identical to B. For example, the following two trees are isomorphic:



because starting from the first tree and exchanging the left and right subtrees of node 5 and of node 8, one obtains the second tree. On the other hand, the following two trees are not isomorphic, because it is impossible to rearrange one into the other:



Question: Write a recursive algorithm that tests if the trees rooted at two given treeNodes are isomorphic. Hint: if your algorithm takes more than 10 lines to write, you're probably not doing the right thing.

Algorithm isIsomorphic(treeNode A, treeNode B)
Input: Two treeNodes A and B
Output: Returns true if the trees rooted at A and B are isomorphic
/* Complete this pseudocode */
if (A = null and B = null) then return true;
if ((A = null and B !=null) or (A != null and B = null)) then return false;
if (A.getKey() != B.getKey()) return false;
boolean isoLL = isIsomorphic(A.getLeftChild(), B.getLeftChild());

```
boolean isoLR = isIsomorphic(A.getLeftChild(), B.getRightChild() );
boolean isoRL = isIsomorphic(A.getRightChild(), B.getLeftChild() );
boolean isoRR = isIsomorphic(A.getRightChild(), B.getRightChild() );
return ((isoLL and isoRR) or (isoLR and isoRL));
```