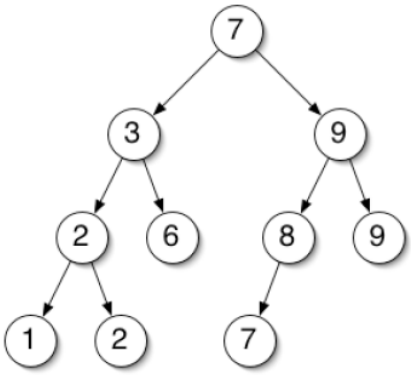
Consider the following pair of recursive algorithms calling each other to traverse a binary tree.

```
Algorithm weirdPreOrder(treeNode n)
   if (n != null) then
        print n.getValue()
        weirdPostOrder( n.getLeftChild() )
        weirdPostOrder( n.getRightChild() )
```

```
Algorithm weirdPostOrder(treeNode n)
   if (n != null) then
        weirdPreOrder( n.getRightChild() )
        weirdPreOrder( n.getLeftChild() )
        print n.getValue()
```

a) **(4 points)** Write the output being printed when weirdPreOrder(root) is executed on the following binary tree:



Method executed	Result printed
Pre(7)	7
Post(3)	
Pre(6)	6
Post(null)	
Post(null)	
Pre(2)	2
Post(1)	
Pre(null)	
Pre(null)	
Print	1
Post(2)	
Pre(null)	
Pre(null)	
Print	2
Print	3
Post(9)	
Pre(9)	9
Post(null)	
Post(null)	
Pre(8)	8

```
                Post(7)
                        Pre(null)
                        Pre(null)
                        Print                   7
        Print                                   9
```

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

b) **(4 points)** Write the output being printed when weirdPostOrder(root) is executed.

```
Method executed                                 Result printed
    Post(7)
        Pre(9)
                Print(9)                        9
                Post(8)
                        Pre(null)
                        Pre(7)
                                Print(7)        7
                                Post(null)
                                Post(null)
                        Print(8)                8
                Post(9)
                        Pre(null)
                        Pre(null)
                        Print(9)                9
        Pre(3)
                Print(3)                        3
                Post(2)
                        Pre(2)
                                Print(2)        2
                                Post(null)
                                Post(null)
                        Pre(1)
                                Print(1)        1
                                Post(null)
                                Post(null)
                        Print(2)                2
                Post(6)
                        Print(null)
                        Print(null)
                        Print(6)                6
        Print(7)                                7
```

Conclusion: 9 7 8 9 3 2 1 2 6 7

c) **(4 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) **(4 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.getLeftChild() != null) then s.push(x.getLeftChild());
    if (x.getRightChild() != null) then s.push(x.getRightChild());
```

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, but one that always starts with the right child instead of the left child:
7 9 9 8 7 3 6 2 2 1