## Question 2.

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?

To obtain a tree with minimal height, each node has to be inserted as full as possible. In other words, the keys should be inserted from left to right so that each parent has two children except for the last parent that is added children to, because it depends on the exact value of n. For instance, assume n=$2^h - 1,$ from the first layer to the last layer, there are 1,2,4,8,…,$2^{h-2}, 2^{h-1}$ nodes respectively.

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.

The worst-case would be finding the leaves, the elements that without children. To find, insert or remove an element at the position of leaves, it needs to go from the root to the leaves, repeating the algorithm h times, where h is the height.

$$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = n$$

$$\frac{1-2^h}{1-2} = n \qquad 2^h = n + 1$$

$$h = \log(n+1) < logn^2 = 2logn \quad for\, n > 1 \qquad\qquad h = O(\log(n))$$

Therefore, the running time of the worst-case in (a) is O(log(n)).

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

In the order that every node is inserted at the left child of the previous node. It will be like a single line from the root to the single leaf.

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.

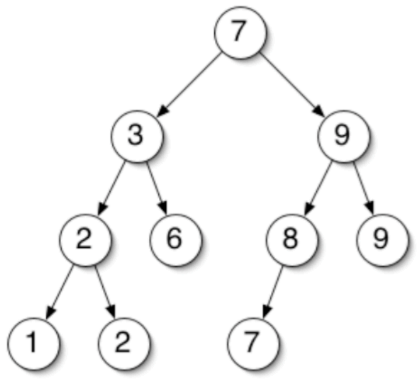
The worst case would be to find, insert or remove an element at the leaf. This needs to go through every element in the tree, repeating the algorithm n times. Therefore, the running time is O(n).

## Question 3.

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.getRightChild() )
    weirdPostOrder( n.getLeftChild() )
```

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



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

It prints out: 7 9 9 7 8 6 2 1 2 3

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

It prints out: 9 9 7 8 6 2 1 2 3 7

c) 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.

It prints out: 7 3 9 2 6 8 9 1 2 7

d) Consider the binary tree traversal algorithm below.

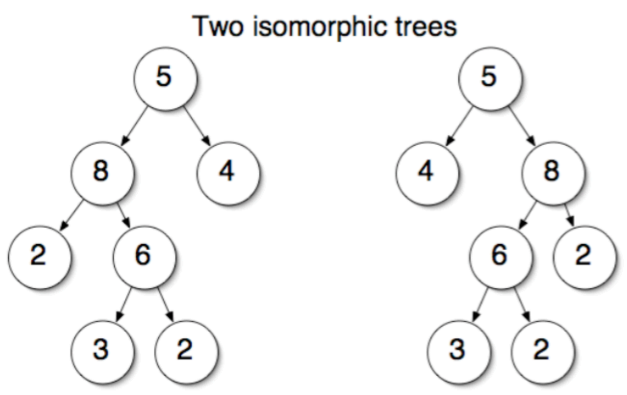
```
Algorithm stackTraversal(treeNode n)
Input: a treeNode n
Output: Prints the value of each node in the binary tree rooted at n
Stack s ← new Stack();
s.push(n);
while( !s.empty() )do
    x ← s.pop();
    print x.getValue();
    if (x.getRightChild() != null) then s.push(x.getRightChild());
    if (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?

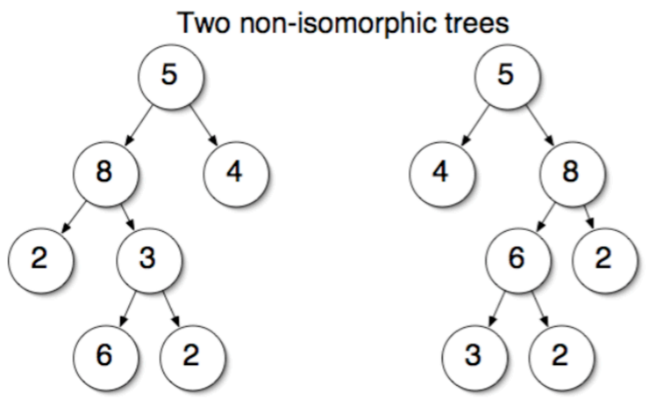
It prints out: 7 3 2 1 2 6 9 8 7 9. It is equivalent to Pre-order traversal.

## **Question 4.** 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, and leaving the children of node 6 unchanged, 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 the pseudocode of 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 testIsomorphic(treeNode m, treeNode n)
Input: a treeNode n, treeNode m
Output: return a boolean indicating that whether m and n are isomorphic or not.
if m = null and n = null then return true;
if ( m = null and n != null ) or ( m != null and n = null ) then return false;
  if m.getValue() != n.getValue() then return false;
  boolean leftLeft = testIsomorphic(m.leftChild, n.leftChild);
  boolean leftRight = testIsomorphic(m.leftChild, n.rightChild);
  boolean rightLeft = testIsomorphic(m.rightChild, n.leftChild);
  boolean rightRight = testIsomorphic(m.rightChild, n.rightChild);
  return (leftLeft and rightRight) or (leftRight and rightLeft);
```