1. Problem 1 (7 pts): Weiss 4.6 - We're looking for a full induction proof

A full node is a node with two children. Prove that the number of full nodes plus one is equal to the number of leaves in a nonempty binary tree.

Let N be the number of full nodes and L be the number of leaves.
Base Case: A tree with 1 full node has 2 leaves. Therefore, N = 1 and L =2

```
   O        O        O
  / \      / \      / \
 O   O  O    O  O    O
       /         /    /
      O         O    O
```

These are the only three possible cases for $N = 1$.

Assumption: Assume that a tree with N = n full nodes has L = n + 1 leaves.

Inductive Step: Prove for N = n + 1 L = (n + 1) + 1. Prove for N = n + 1 L = (n + 1) + 1.

$$\underbrace{(n+1)+1}_{\text{Lefthand Side}} = \underbrace{n+1+Y}_{\text{Righthand Side}}$$

- Lefthand Side: Number of leaves a tree with n+1 full nodes has.
- Righthand Side: Number of leaves a tree with n nodes has plus Y.
- Y: Number of leaves needed to be added to a tree with n full nodes to make it have $n+1$ full nodes.

In order to add an additional full node to the tree (i.e. convert a leaf to a full node), we can (1) add two leaves to a node that was once a leaf.

```
   O
  / \
 O   X
    / \
   A   A
```

In the diagram above, `X` and `O`s are pre-existing node and `A`s are newly added. In order to make the leaf `X` a full node, we have to add two children (`A`s) to `X`. This causes the number of leaves to reduce by one (since `X` is no longer a leaf) and then increase by 2 (since we added 2 leaves). Hence, $Y = 2 - 1$

We can also (2) add a leaf to a node that has one child.

```
  O
 / \
O   Z
   / \
  X   A
```

Again, `X`s, `Z`s and `O`s are pre-existing nodes. We add `A` as a leaf. This also increases the number of leaves by 1 in total. Note that `Z` was not a leaf node. Hence, $Y = 1$

These two cases cover all the possible cases of adding a full node to a tree.

$$\begin{aligned}(n+1)+1 &= n+1+Y\\(n+1)+1 &= n+1+(1)\\n+2 &= n+2\end{aligned}$$

We have thus proven the inductive step. Since the base case is proven and the inductive step is proven, we have an inductive proof.

Problem 2 (7 pts): Weiss 4.9 - In part a show the tree after each insert - a total of 8 trees. In part b, we're doing full deletion, not lazy deletion. Show the tree after each step of the removal.

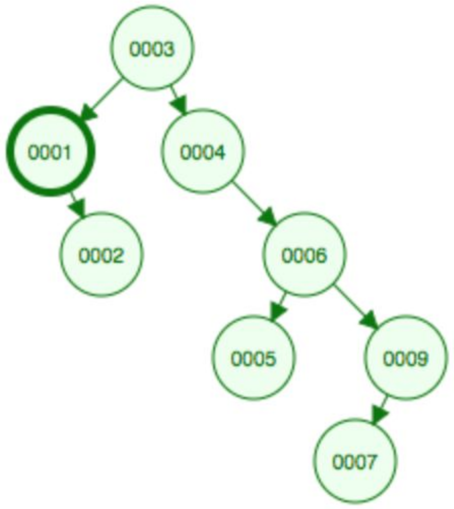
(a) Inserting 3, 1, 4, 6, 9, 2, 5, 7 into a BST

```
   3

   3
  /
1

   3
  / \
1    4

   3
  / \
1    4
      \
       6
```

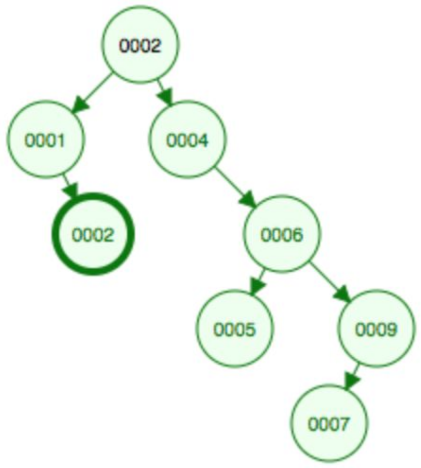
```
    3
   / \
  1   4
       \
        6
         \
          9
    3
   / \
  1   4
   \   \
    2   6
         \
          9
    3
   / \
  1   4
   \   \
    2   6
       / \
      5   9
    3
   / \
  1   4
   \   \
    2   6
       / \
      5   9
         /
        7
```

Show the result of deleting the root.

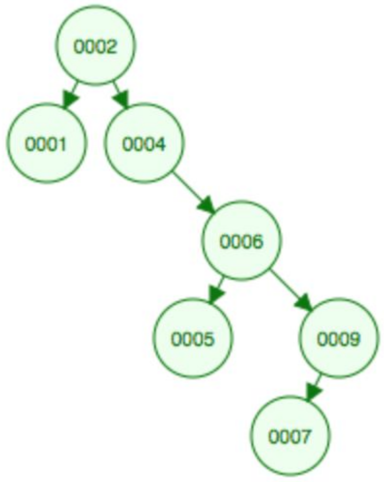
Node to delete has two childern.
Find largest node in left subtree.



Copy largest value of left subtree into node to delete.



Remove node whose value we copied.

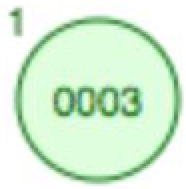


Problem 3 (7 pts): Weiss 4.9 - In part a show the tree after each insert - a total of 8 trees. In part b, we're doing full deletion, not lazy deletion. Show the tree after each step of the removal.

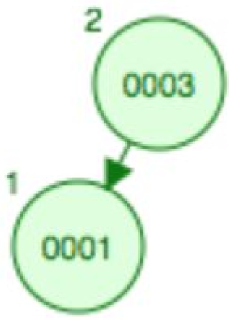
Inserting 3, 1, 4, 6, 9, 2, 5, 7 into a BST

9 - single rotation

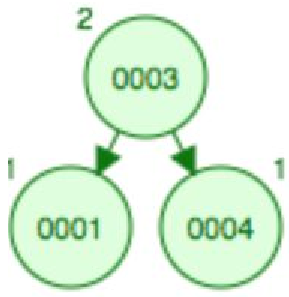
insert (3)



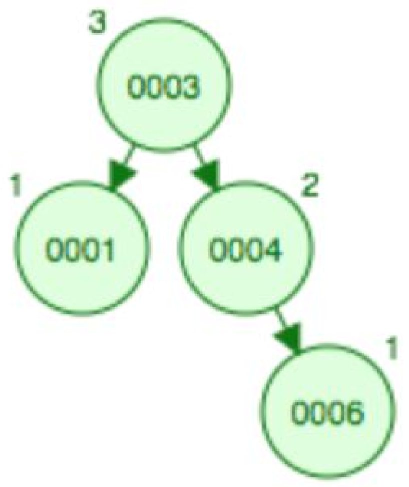
insert(1)



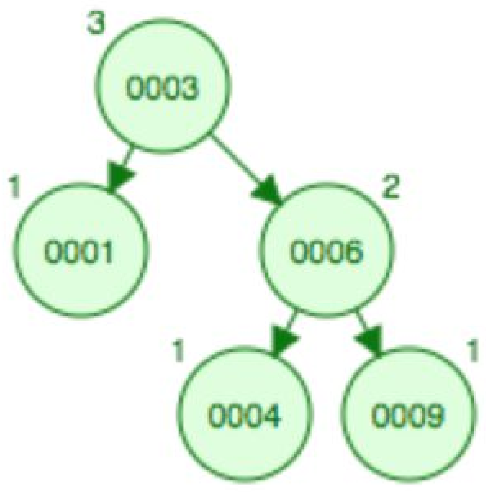
insert(4)



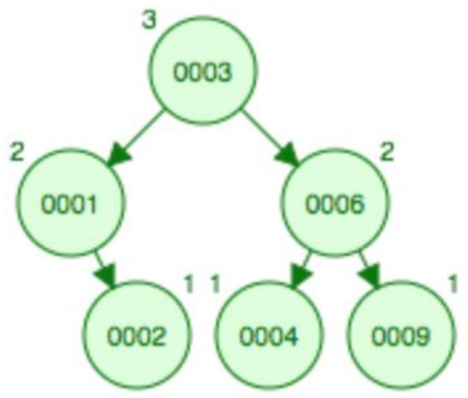
insert(6)



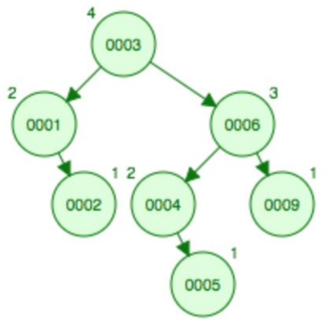
insert(9) and single rotation on 4



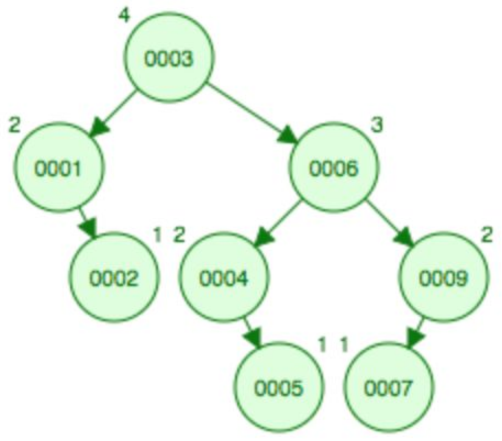
insert(2)



insert(5)

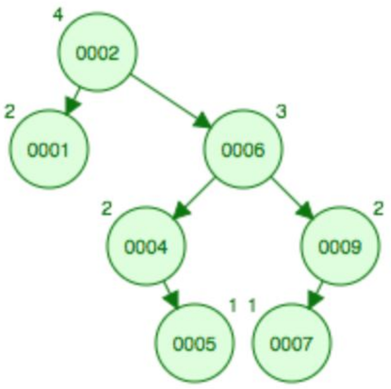


insert(7)

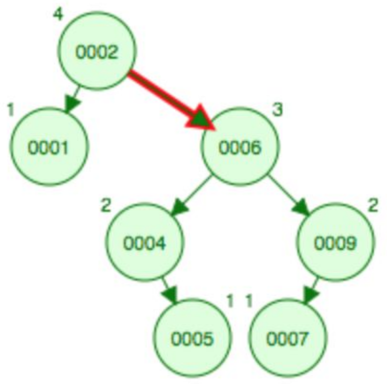


Show the result of deleting the root.
Find the largest node in the left subtree

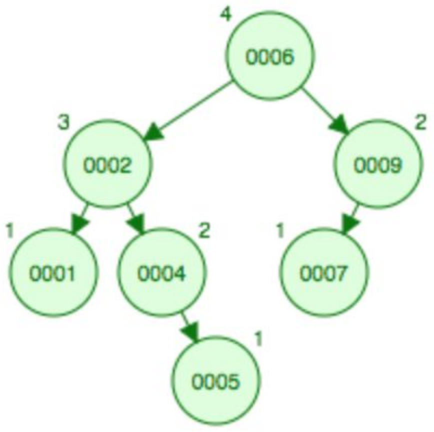
Remove node whose value we copied.



Single Rotate Left



Single Rotate Left



Problem 4 (7 pts): Assume that you are given both the preorder and postorder traversal of some binary tree *t*. Prove that this information, taken together, is not necessarily sufficient to reconstruct *t* uniquely, even if each value in *t* occurs only once.

Consider the following two trees:

```
 1              1
   \           /
    2        2
```

They both have the following preorder and postorder traversals:

- Preorder: [1, 2]
- Postorder: [2, 1]

Hence, having only the preorder and postorder traversal output does not reconstruct trees.

However, if we add an additional constraint: that no node can have a right child without having a left child (or equivalently, that the children of nodes are symmetric), then preorder and postorder do uniquely determine a tree.

For example, say we have the following traversals:

- Preorder: [4, 2, 1, 3, 5, 6]
- Postorder: [1, 3, 2, 6, 5, 4]

Since preorder goes Self, Left, Right and postorder goes Left, Right, Self, we know that 4 is the root of the tree. This is because 4 is first in the preorder traversal, and since we start from the root of the tree, the root (self in that step of the recursion) will be printed first by the preorder.

We also know that 2 is the root of the left subtree. This is from the preorder traversal and from our condition: since the preorder traversal prints left after self (self was 4), and that 4 cannot have a right child without having a left child first, and that 4 must have at least 1 child (if not the tree won't be able to continue), then 2 must be the left child of 4. Hence, 2 is the root of the left subtree.

We further know that 5 is the root of the right subtree. This is from the postorder traversal. Since the postorder traversal goes Left, Right, Self, the node immediately before self would be right. How do we know that 5 is not on the left? Because we know for sure that 2 is the root of the left subtree. Hence, if 4 only had a left subtree, the sequence would end with [... 2, 4]. However, since it ends with [... 2 ... 5, 4], the portion ... 5 would be part of the postorder traversal of the right subtree, which eventually comes back to 5, then the root 4.

Hence, we have the structure of two subtrees:

- Root: 4
- Left subtree:
– Preorder: [2, 1, 3]

– Postorder: [1, 3, 2] • Right subtree:
– Preorder: [5, 6] – Postorder: [6, 5]

Each of the subtrees can be solved recursively using the same analysis.
Hence, a tree can be uniquely determined using preorder and postorder traversals **if we impose the restriction on a node's children**.

Redo the binary search tree class to implement lazy deletion. Note carefully that this affects all of the routines. Especially challenging are findMin and findMax, which must now be done recursively.

[see BSTLazy.java for a working solution]

Note on Problem 5:
The idea is that when we are at a deleted node, we would first look for a minimum in the node's left subtree. If there is such a minimum, we return it. Otherwise, we need to look in the deleted node's right subtree for another minimum candidate.

If the node isn't deleted, we proceed as normal – keep descending via the left child.

There's a public findMin() and a private recursive findMin that takes in a NodeLazy<T> as parameter. If the node is null, we return null. Otherwise, we recursively go one level deeper by trying to find the minimum in the left subtree of the current node in the line NodeLazy<T> leftMin = findMin(node.left);. If the returned minimum of the left subtree is not null, we have successfully found a minimum. We return it. Otherwise (ie. leftMin == null is true), we know that the left subtree does not contain a minimum. We check if the current node is deleted. If it is not deleted, then the current node will be the minimum. Otherwise, if it is deleted, it is ineligible for being the minimum. Hence, we are left with the option of searching the right subtree which we do in the line findMin(node.right);