About Lab 6

In Lab 6 we give you a binary tree structure for a TreeMap and a partial implementation of Binary Search trees. You need to implement recursive search( ) and insert( ) methods to complete the Binary Search tree implementation, then add an adjust( ) method to rebalance the tree after every insert. This converts your Binary Search Trees into AVL Trees.

The trees in Lab 6 are based on the following Node structure:

```
class Node {
        K key;
        V value;
        int height;
        Node leftChild, rightChild;
}
```

Unlike Lab 5, there is no EmptyTree class. We represent empty trees as nodes where the key, value, and both children are null and the height is -1.

A leaf node is one where both children are empty trees. Leaf nodes have keys and values; the height of a leaf is 0.

The Node class has an isEmpty( ) method and an isLeaf( ) method.

search( ):   The first method you should write is the recursive
private V search(K key, Node t)

This looks through the portion of the tree at node t and below for a node that has *key* as its key and if it finds such a node it retuirnsthe node's value. If node t is empty there is no such node and search returns null. If t is not empty you can compare the key you are seeking with t.key. If they are the same return t.value. If *key* is less than t.key recurse on t's leftChild; if *key* is larger recurse on t's right child.

Note that there is no effective way to test your search( ) method until you have also written the insert( ) method.

insert( ): Now you should implement the recursive method

```
private Node insert(K key, V  value, Node t)
```

This should return the tree that results from inserting the (key, value) pair into the tree with root t.

The insert( ) method uses the same recursive structure as search( ):

- If t is an empty node return a new leaf with the given key-value pair. The MyTreeMap<K,V> class has a factory method newLeafNode(key, value) that builds a leaf node for you.
- If the *key* you are seeking is t.key just change t.value to *value* and return t.
- If *key* is less than t.key replace t.leftChild with the result of recusing on t.leftChild, recompute the height of node t, and return t.
- Do something similar if *key* is larger than t.key

You should now be ready to test your insert( ) and search( ) methods. The starter code includes a program AVLTester.java. This does:

a) Starting with an empty tree it performs 10 inserts where the keys are fruit names and the associated values are the integers 0, 1, 2, … After each insertion it prints the height of the tree. You should be able to draw the Binary Search Trees trees this produces and check that you are getting trees of the correct heights.
b) It does a search for each of the keys and prints the resulting value. You should be able to match those against the key-value pairs that were inserted.

You can modify program AVLTester if you want. Feel free to change the values that are inserted into the tree (names and ages? Words and their lengths? There are many possibilities.

Only when you are confident that search( ) and insert( ) are correct should you go on to part 3.

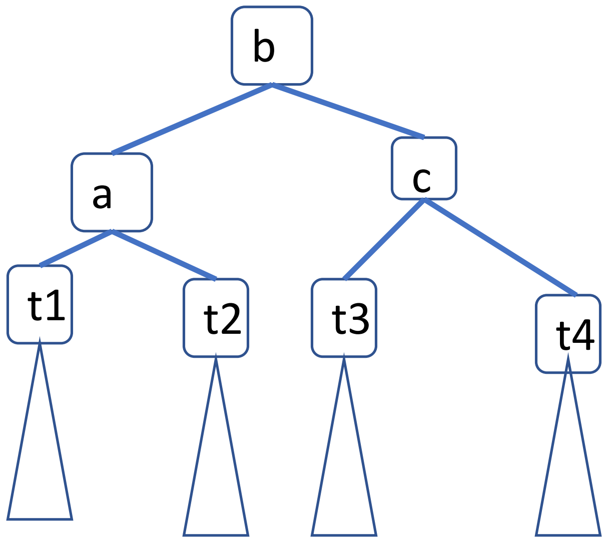
adjust( )   The last method you need to implement rebalances the tree after an insert. Remember the algorithm for this:

Node Z is the lowest node in the tree that fails the AVL test (heights of the two children differ by no more than 1). Node Y is Z's tallest child; node X is Y's tallest child.

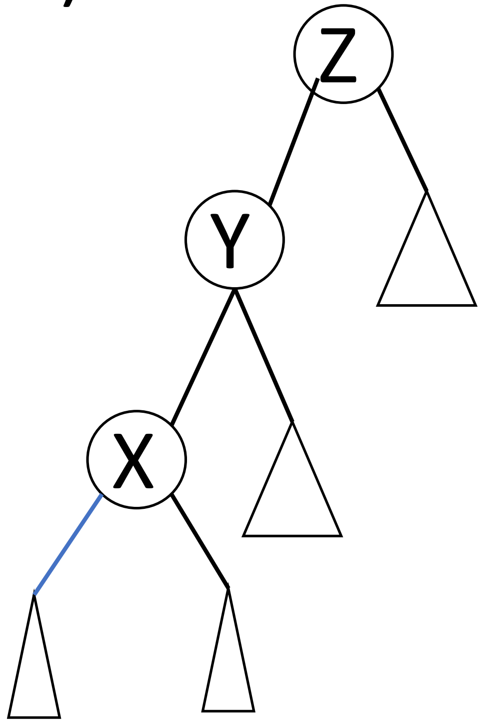
Nodes a, b, and c are new names for X, Y, and Z where a has the smallest key, b has the middle key and c the largest key.

Subtrees t1, t2, t3, and t4 are the four children of X, Y and Z that are not themselves one of X, Y and Z.

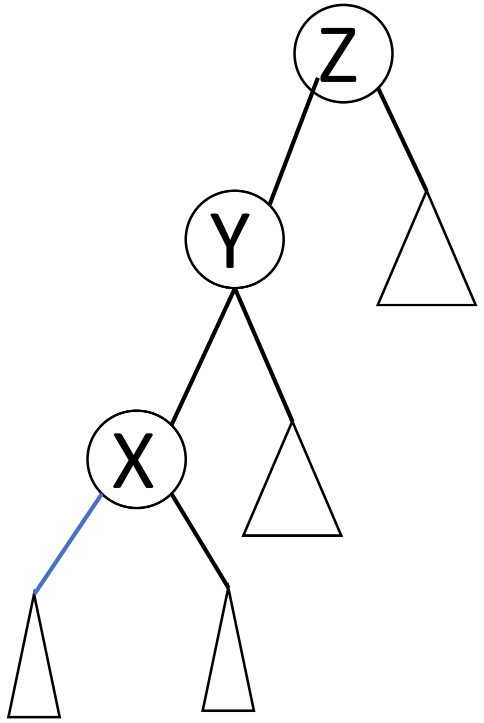
Once we identify Z, Y, X, a, b, c, t1, t2, t3, and t4, adjust( ) returns the f0llowing tree:



You can find everything from the way Z, Y, and X are related, Suppose Y (Z’s tallest child) is Z’s left child and X (Y’s tallestl child) is Y’s left child:



Remember that this is a Binary Search Tree: every key in a node’s left subtree has a value less than the node’s key; everything in the right subtree has a value larger than the node’s key.



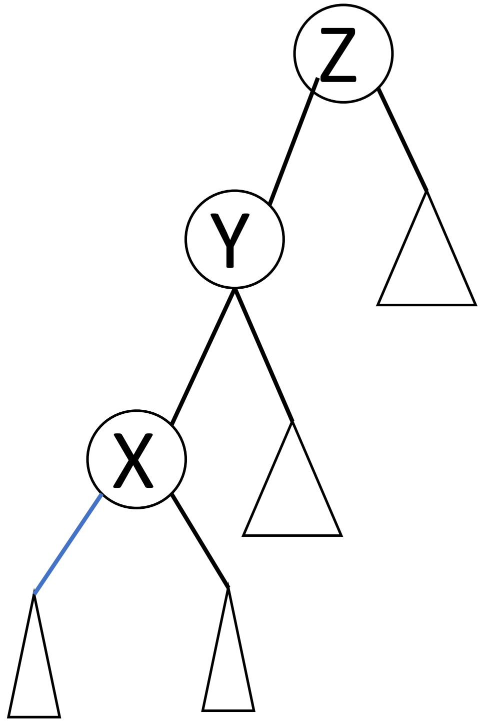
Question: In this case (Y is Z’s left child, X is Y’s left child) which of X, Y and Z is a? (i.e. which has the smallest value?)

A. X

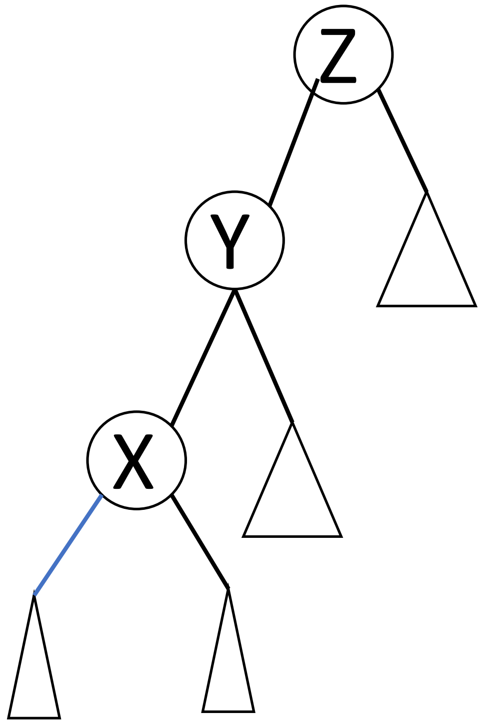
B. Y

C. Z

D. You need more information to decide



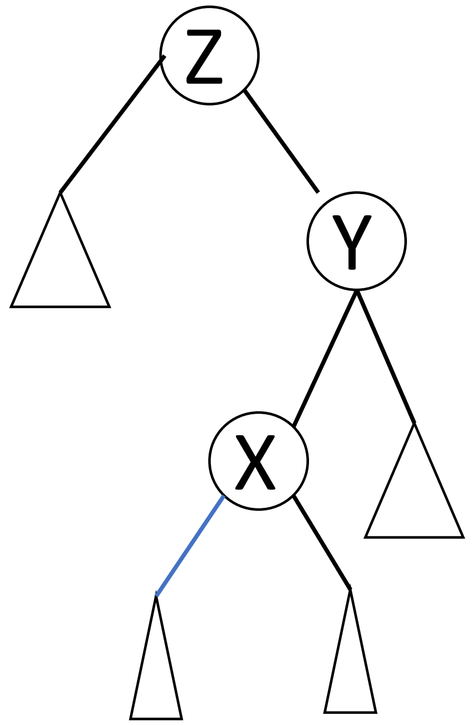
Answer: X has the smallest value. In fact, a=X; b=Y; c=Z;



Question: What are t1 through t4 in this case? They need to be in increasing order.

A. t1=Z.rightChild; t2=Y.rightChild; t3=X.rightChild; t4= X.leftChild;
B. t1=X.leftChild; t2=X.rightChild; t3=Y.rightChild; t4=Z.rightCHild;
C. t1=X.leftChild; t2=y.leftChild; t3=Z.leftChild; t4=Z.rightChild;
D. You need more information to decide.

Suppose Y (Z’s tallest child) is Z’s right child and X is Y’s left child:



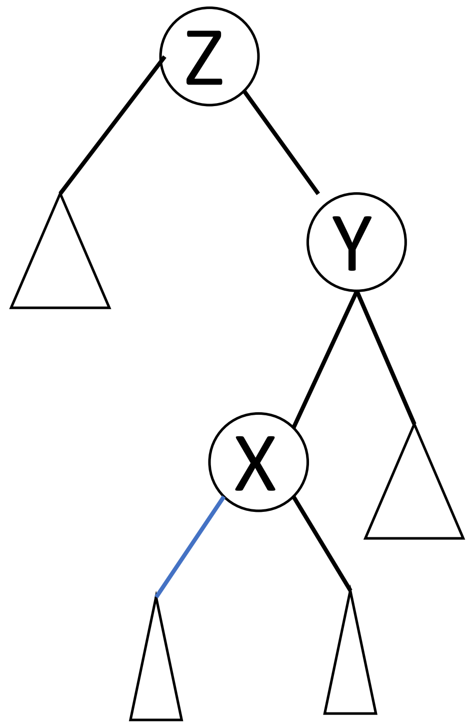
Which is the smallest of X, Y, and Z; which is the largest?

A. X is the smallest; Z is the largest.

B. X is the smallest; Y is the largest.

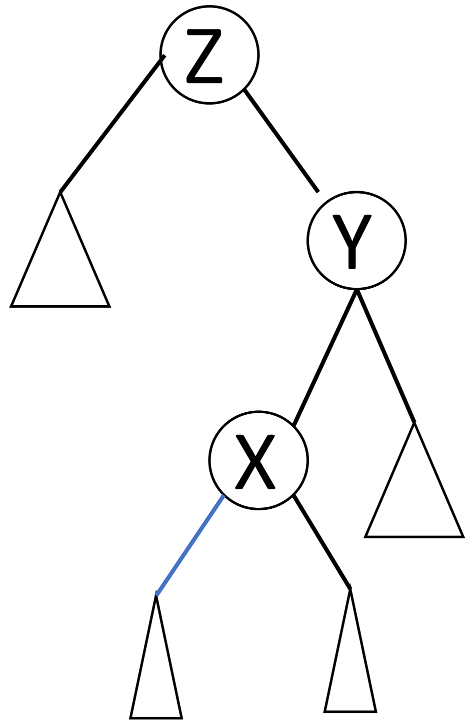
C. Z is the smallest; X is the largest

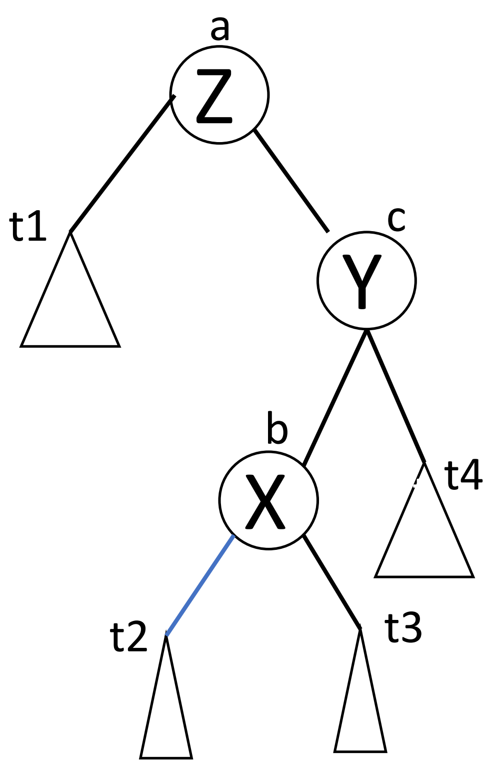
D. Z is the smallest; Y is the largest



Answer D: Z is the smallest; Y is the largest

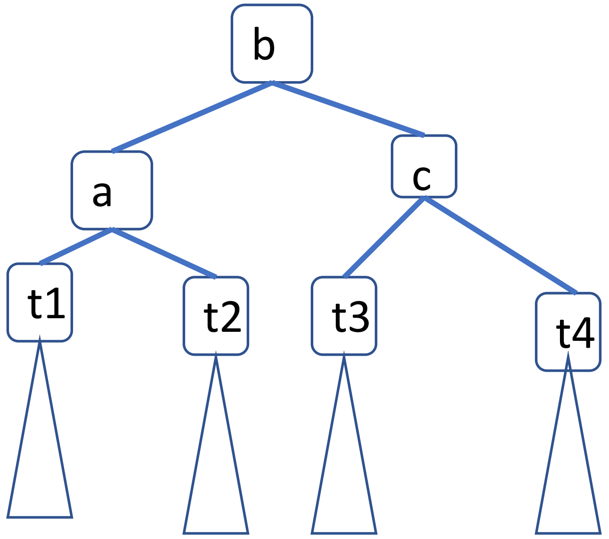
Label a, c, b, t1, …, t4 for this case (Y is Z’s right child, X is Y’s left child:





In other words, a=Z; b=X; c=Y; t1=Z.leftChild; t2=X.leftChild; t3=X.rightChild; t4=Y.rightChild;

You can do all four cases this way. Once you have assigned all of the variables build this tree:



i.e, a.leftChild=t1; a.rightChild=t2. Then recompute a’s height.
Do the same for c, then b, and return b.

All that remains is hooking this up with your insert() method. The only cases of insert that possibly imbalance the tree are when you recurse: t.leftChild = insert(key, value, t,leftChild) or a similar expression with the rightChild. After either case test whether t still satisfies the AVL property; if it does not, call adjust(t). If adjust( ) sets the height of node b you can just return adjust(t). If adjust doesn’t set the height of b then in insert say t=adjust(t); set the height of node t, then return t.

To test this run the AVLTester program. This time the heights of the trees should be appropriate for AVL trees. Your searches should still allow you to look up every key in the tree.