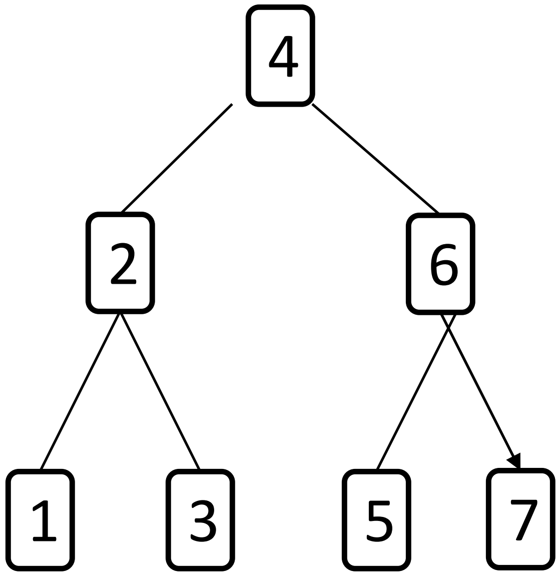
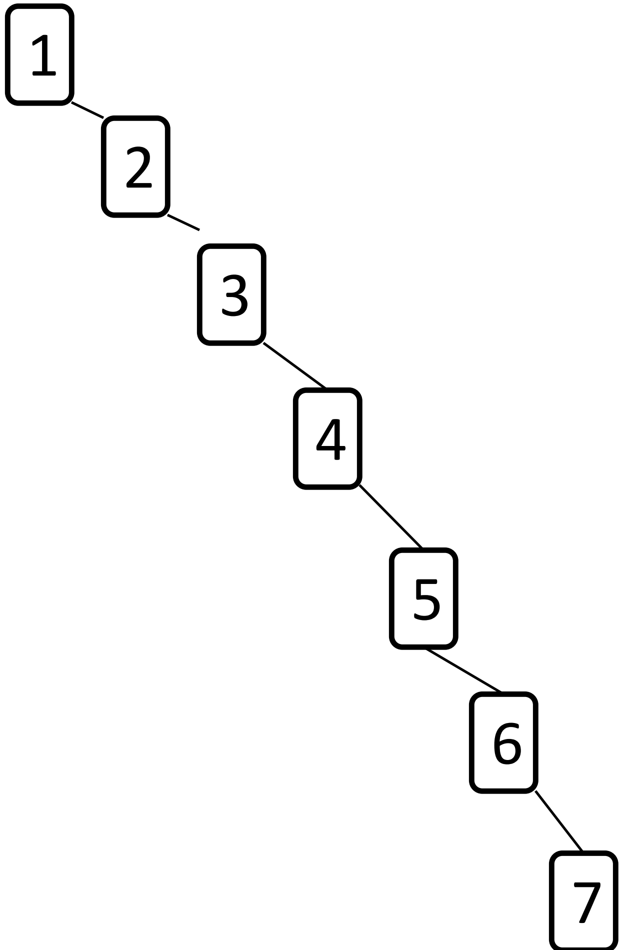
AVL Trees

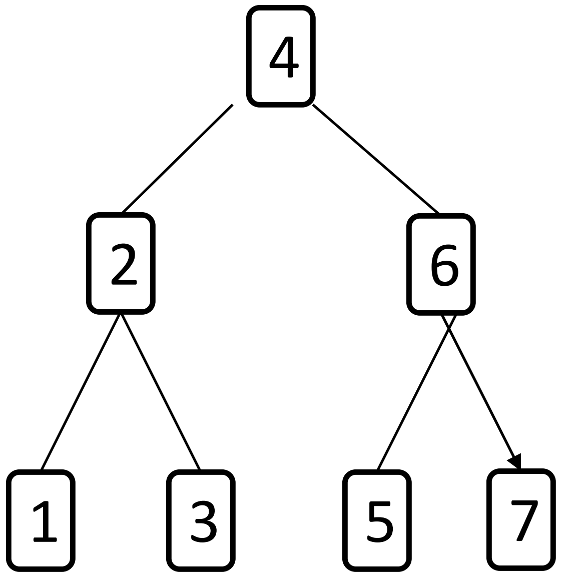
See Section 19.4of the text, p. 706-714.

The *idea* of a binary search tree is great, but in practice they might not be very good structures for storing data. Here are two correct binary search trees for the numbers 1-7:

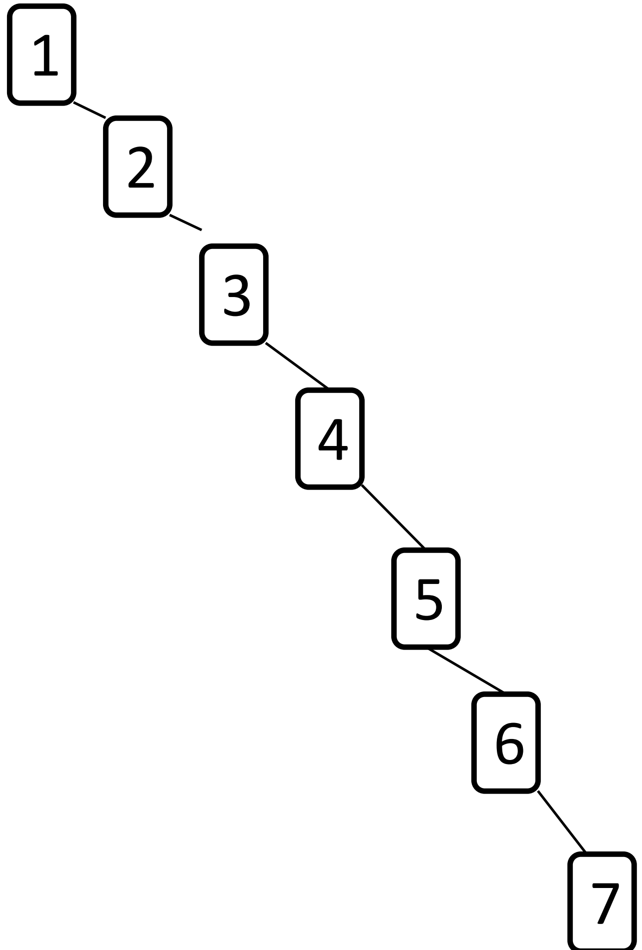




This tree is *balanced.* All of the nodes at a given depth have the same numbers of nodes in their left and right subtrees. The find( ), add( ) and remove( ) algorithms for balanced binary trees all have O(log(n)) *running time.*



This is a binary search tree that is about as unbalanced as a tree can get. If we build a binary search tree by adding nodes in increasing or decreasing order, it defaults to a linked list.

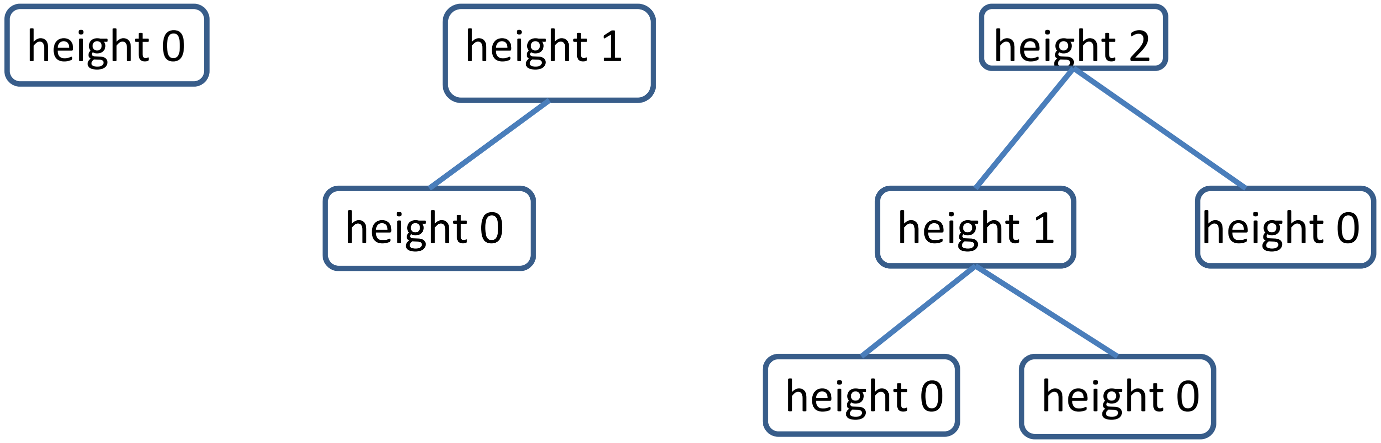


Note that the add( ), remove( ), and find( ) algorithms for binary search trees all start and the root and walk down a path to a leaf. The worst-case numbers of operations for these algorithms are all proportional to the height of the tree. If the tree is balanced those operations will all be O(log(n)); if the tree is not balanced they may be O(n).

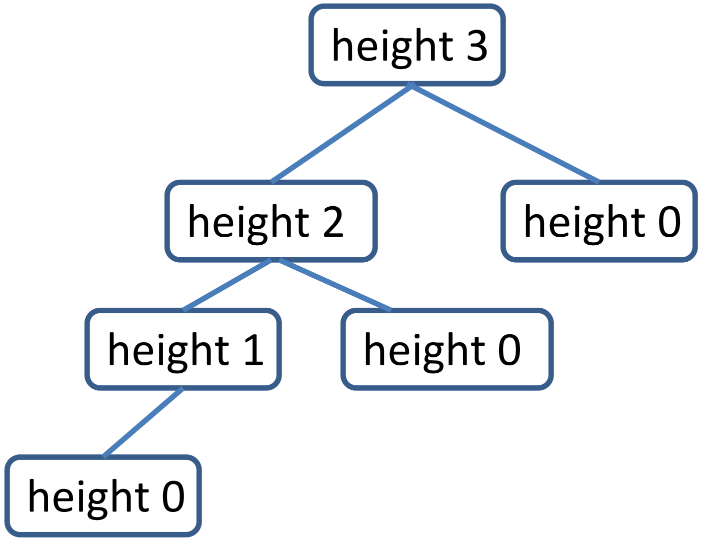
AVL trees are *self-balancing* Binary Search Trees. When you either insert or remove a node the tree adjusts its structure so that the height remains a logarithm of the number of nodes. No matter what order we insert the nodes, we can search an AVL in O(log(n)) time.

AVL trees were the first such self-balancing trees. They were invented by Georgy Adelson-Velski and Evgenii Landis in 1962.

Definition: An AVL tree is a Binary Search Tree with the additional property that for every node in the tree, the left and right subtrees have height that differ by at most 1. We say that the height of a null tree is -1 and the height of a single node is 0; the height of any other node is 1 more than the max of the heights of its children. Here are some AVL trees



Here is a tree that is not an AVL tree; the children of the root have heights that differ by 2:



One issue with implementing AVL trees is having access to the height of each node. We change the Node class so it has fields

```
E data;
int height;
Node left, right;
```

Or, for a map:

```
 K key;
V value;
int height;
Node left, right;
```

We also change the insert() and remove() methods to adjust the height at each node on the path between the root and the modified node. The recursive version of the insert method is easy to modify. After we come back from the recursive call we reevaluate the height:

```
private Node insert(E x, Node t) {
        if (t == null) {
                Node s = new Node(x);
                s.height = 0;
                return s;
        }
        else {
                int comparison = x.compareTo(t.data);
                if (comparison == 0)
                        t.data = x;
                else if (comparison < 0)
                        t.left = insert(x, t.left);
                else
                        t.right = insert(x, t.right);
                // if t fails the AVL test adjust t
                // set t.height
                return t;
        }
}
```

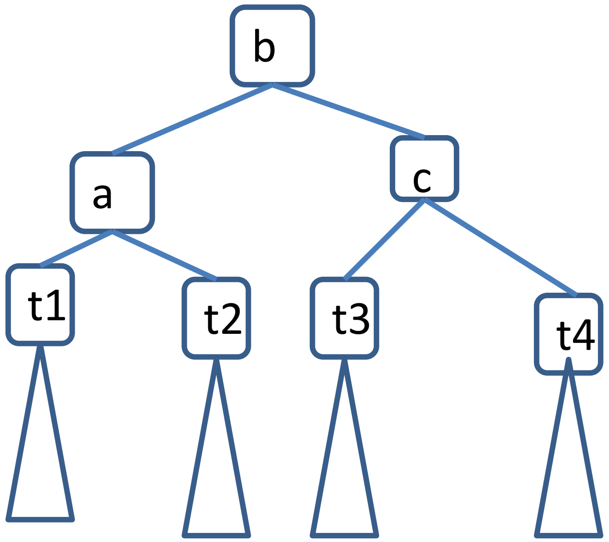
This much is easy; the interesting part of AVL trees comes lies in the adjustments we need to do when a tree becomes imbalanced. Consider inserts. We have a balanced tree and insert a node and the tree is then unbalanced. Since inserting can add at most one to the height of a subtree this must mean that we have a node Z on the path between the root and the inserted node where the height of one child is 2 more than the height of the other child. There are 4 possible cases:

a) The insertion was in the left subtree of the left child of Z
b) The insertion was in the right subtree of the left child of Z
c) The insertion was in the left subtree of the right child of Z
d) The insertion was in the right subtree of the right child of Z

Here is how we will deal with these cases. In every case Z is the node that fails the AVL condition after the insertion. We let Y be Z's tallest child, and X be Y's tallest child. The insertion must have happened in or below node X.

Now let nodes a, b, and c be X, Y and Z ordered from smallest to largest key values. As you will see when we draw pictures, nodes a, b, and c have a total of 4 children that aren't themselves one of a, b, or c. We let t1, t2, t3, and t4 be these children, in increasing order.

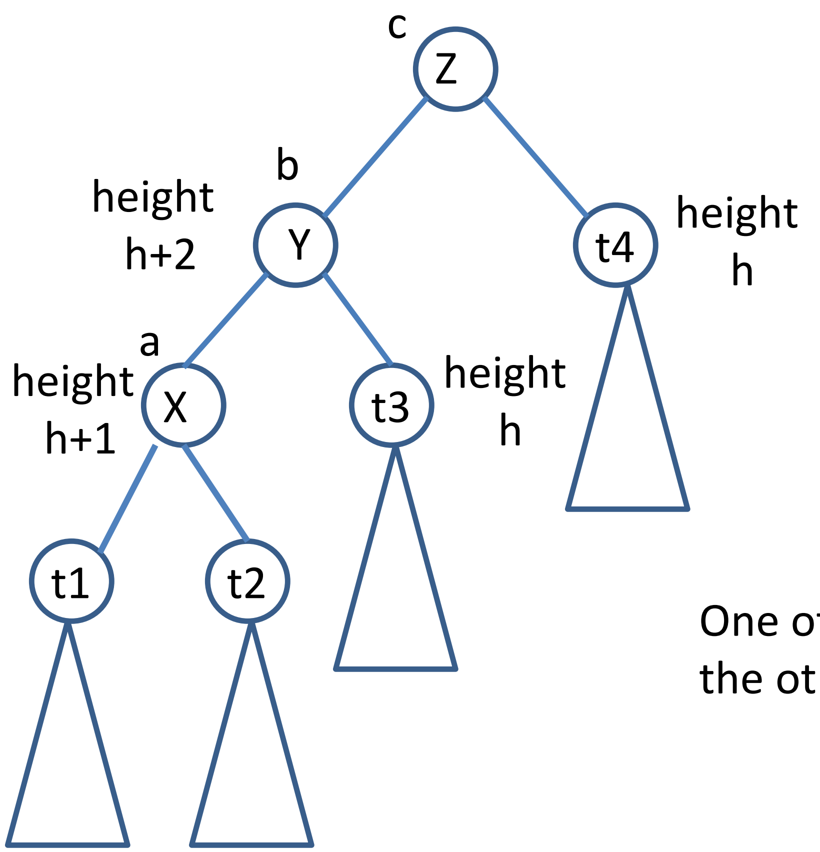
In all 4 cases we build the following tree:



It is relatively easy to verify that this is a binary search tree, that all of its nodes satisfy the AVL property, and that node b has the same height as node Z did before the insertion. This means this is the only adjustment we need to make to restore the tree to the AVL property.

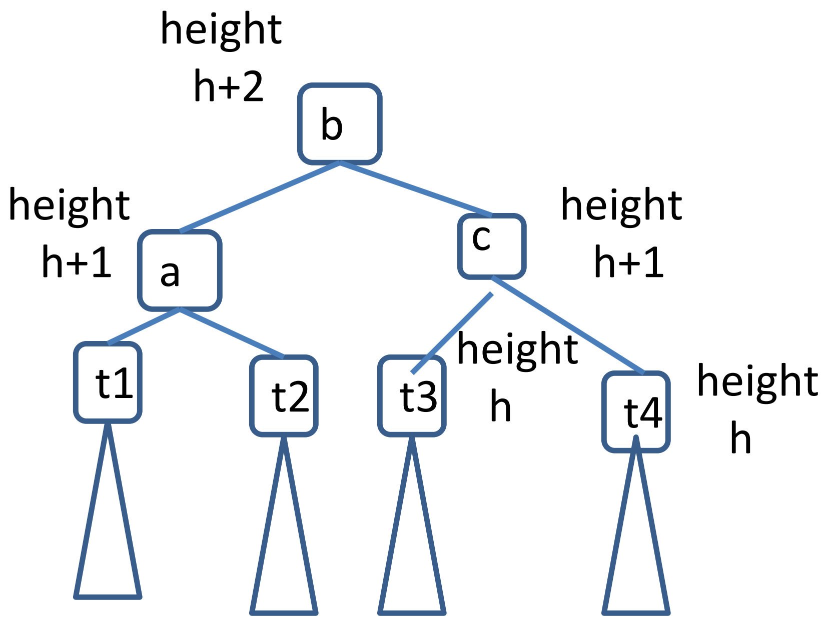
Remember that Z is the node that fails the AVL condition, Y is Z’s tallest child, X is Y’s tallest child

Case 1: Y is Z’s left child, X is Y’s left child

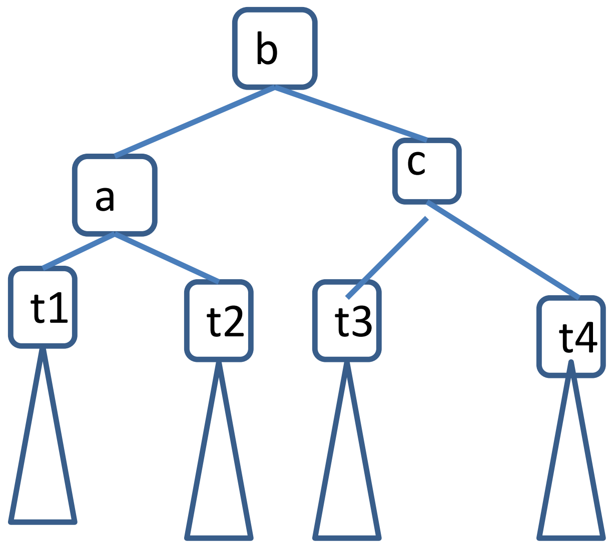


One of nodes t1 and t2 must have height h; the other has height h-1.

Putting these heights into the tree we build we see that every node satisfies the AVL property:



One of nodes t1 and t2 has height h; the other has height h-1.

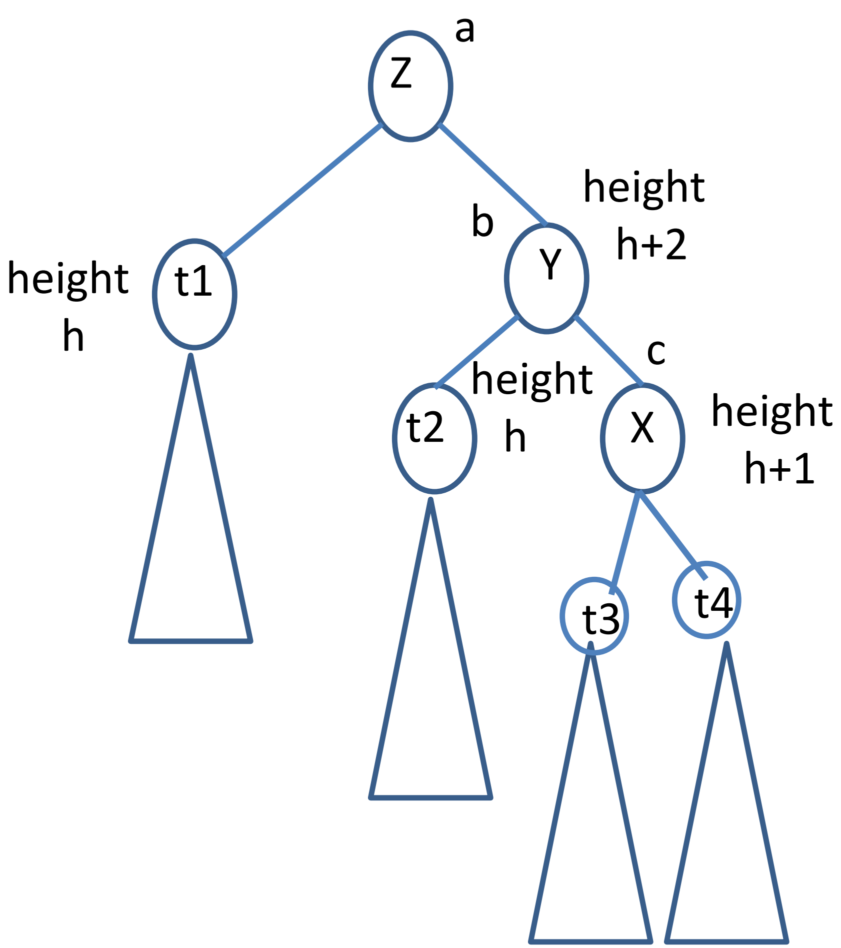


Furthermore, t1 and t2 have the same relation to a as before, all of the values in t3 are greater than the value of b and less than the value of c, and t4 remains the right child of c. This means that the new tree is a Binary Search tree that satisfies the AVL property.

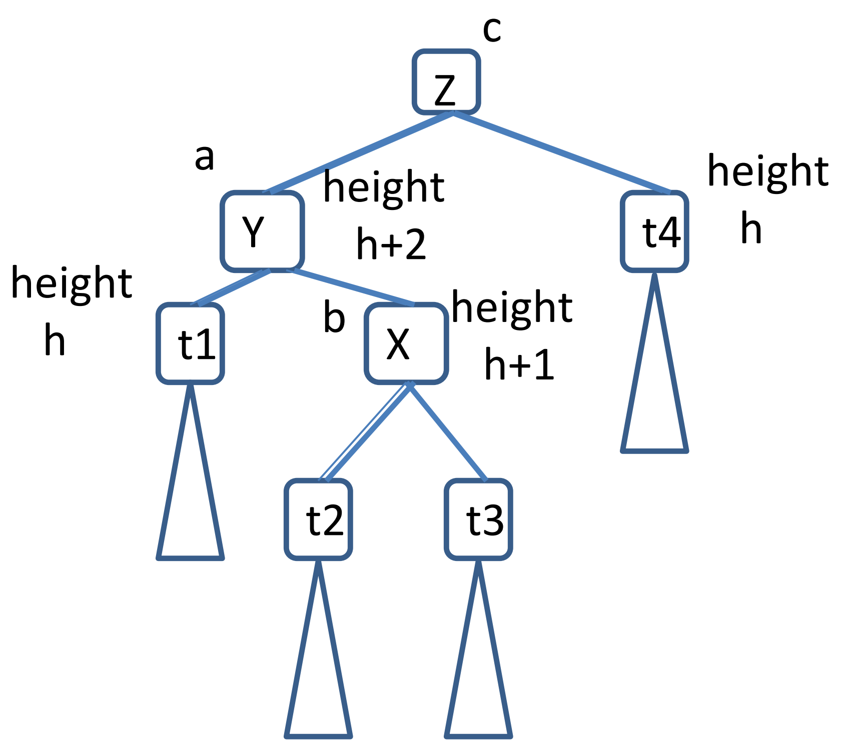
The other three cases are similar; we will go through them more quickly.

Case 2: Y is Z’s right child, X is Y’s right child.

This is symmetric to case 1

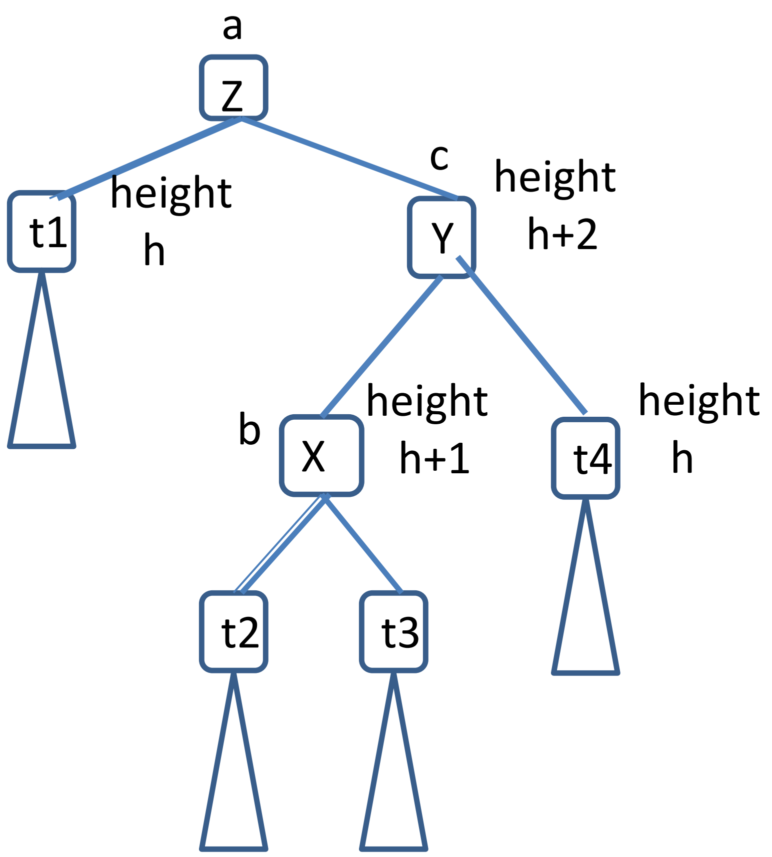


Case 3: Y is Z’s left child and X is Y’s right child:



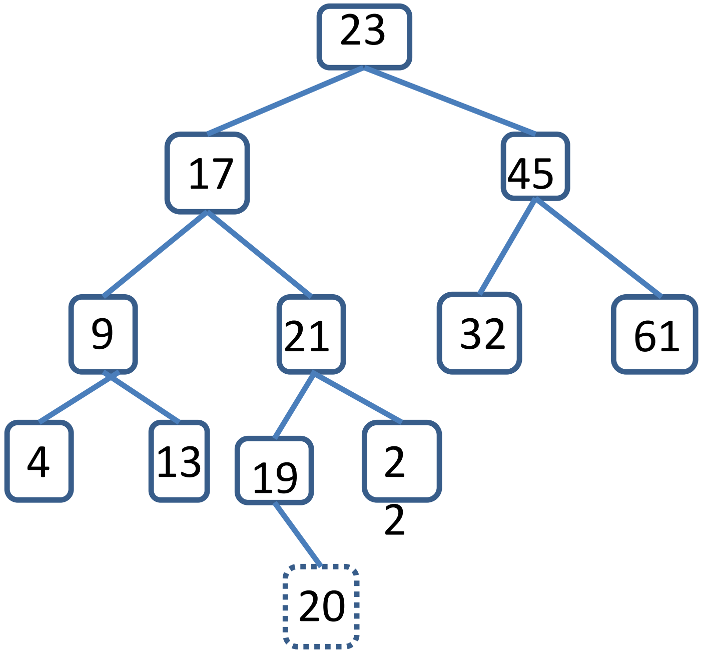
One of nodes t2 and t3 has height h; the other has height h-1.

Case 4: Y is Z's right child and X is Y's left child; this is the mirror image of Case 3:

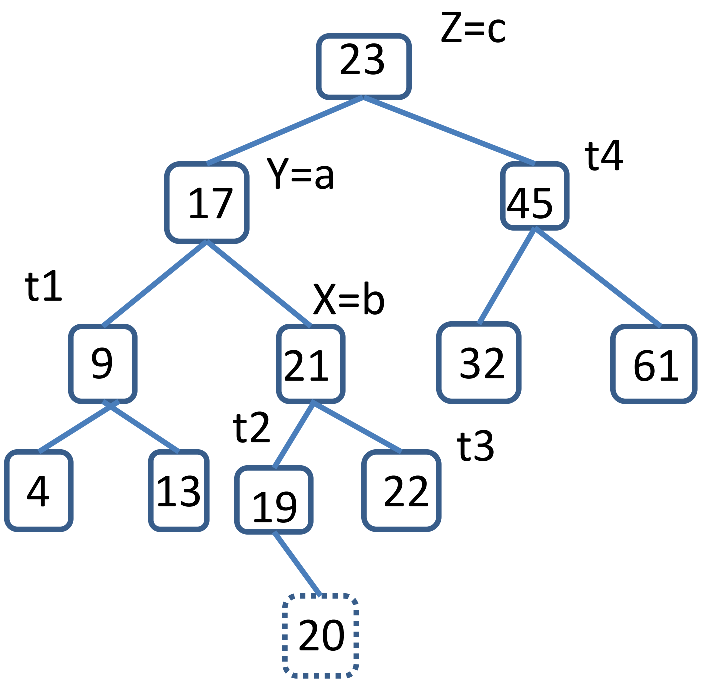


One of nodes t2 and t3 has height h; the other has height h-1.

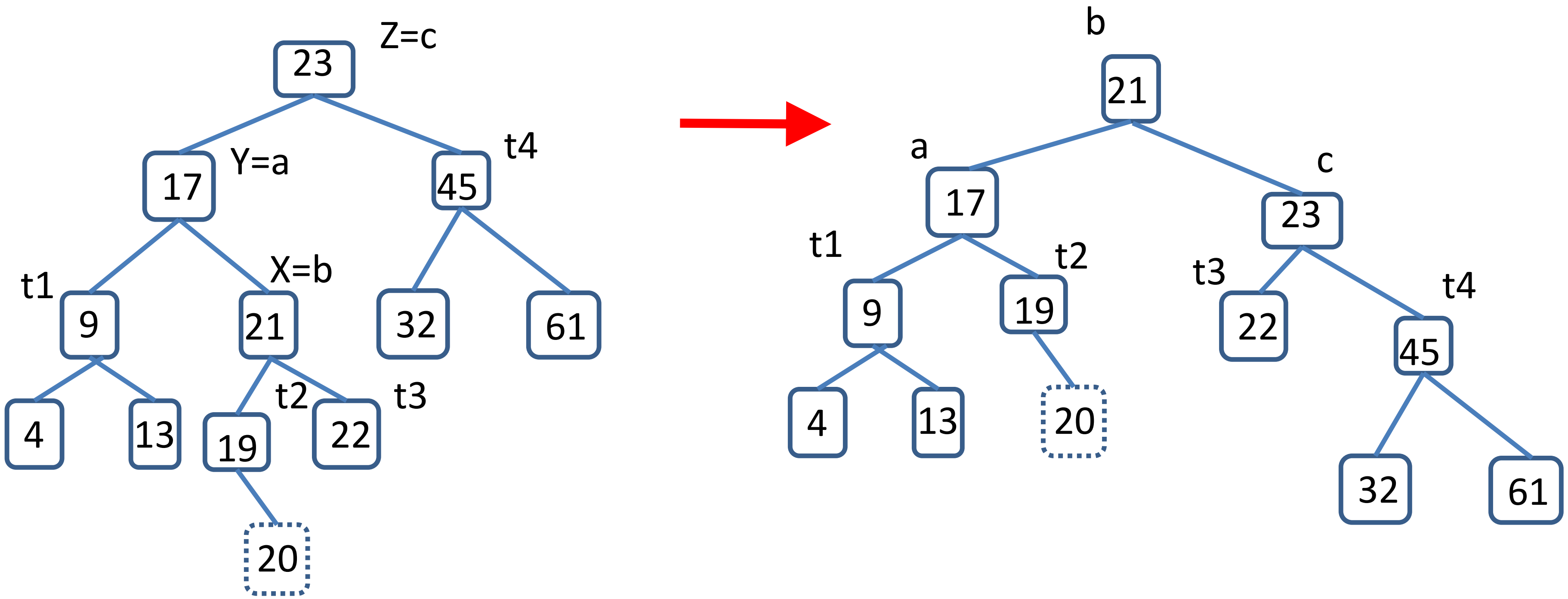
Consider this example, where we have just inserted 20. The left child of node 23 has height 3, the right child has height 1.



We assign our labels: Z is the node whose children fail the AVL property: that is 23; Y is Z’s tallest child: 17, and X is Y’s tallest child: 21.



Once we have Z, Y, and X identified we can label a, b and c as those same nodes in increasing order, and we identify t1 through t4 as the children of a, b, and c. We can now rebuild the tree:



It is easy to verify that this is a Binary Search tree and all of it nodes satisfy the AVL height property.