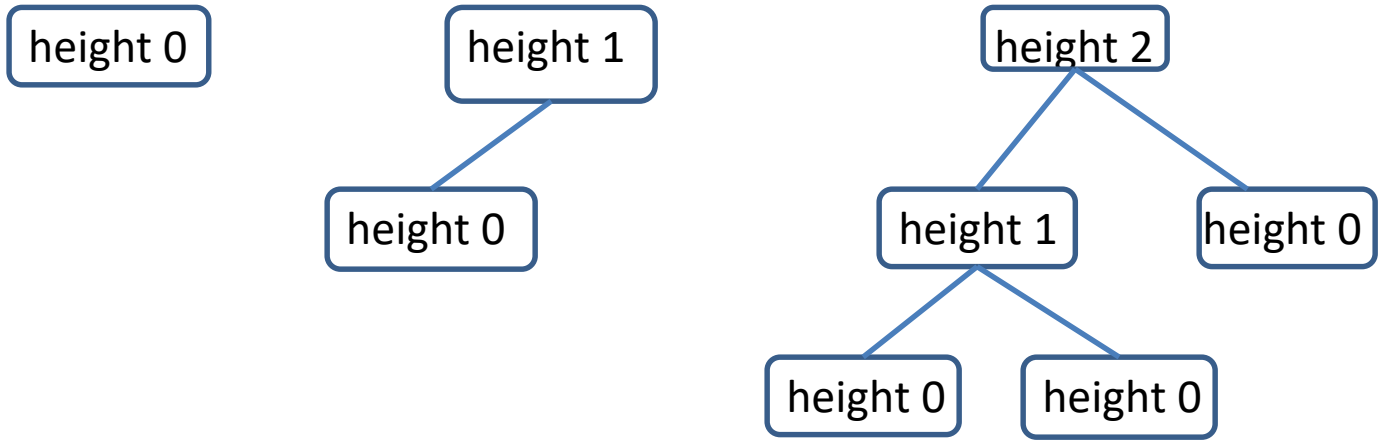
AVL Trees

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

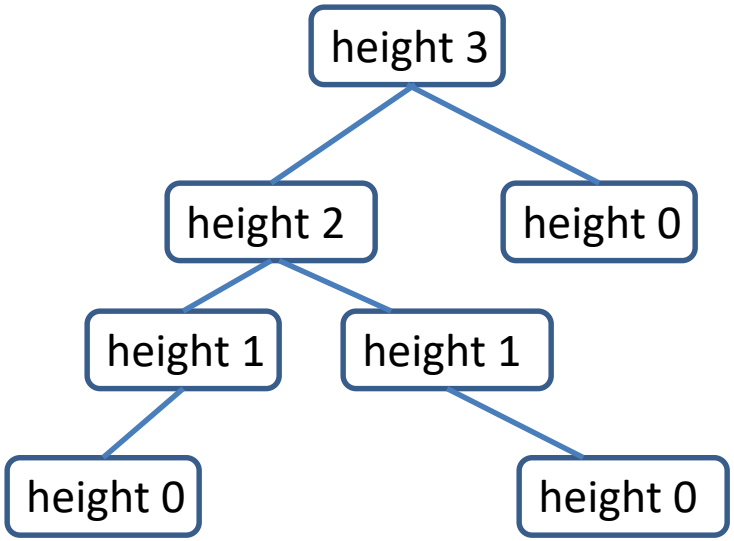
AVL trees are self-balancing Binary Search Trees. When you either insert or remove a node the tree adjusts its structure so that the height remains a logarithm of the number of nodes. No matter what order we insert the nodes, we can search an AVL in O(log(n)) time.

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

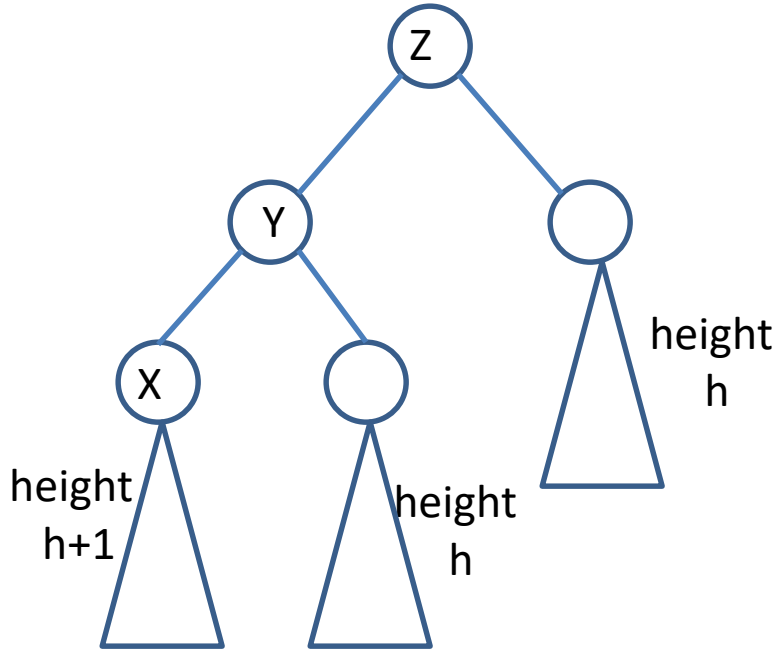
We also change the insert() and remove() methods to adjust the height at each node on the path between the root and the modified node. The recursive version of the insert method is easy to modify. After we come back from the recursive call we reevaluate the height:

```
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);
                t.height = 1+ max( height(t.left), height(t.right));
                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. There are 4 possible cases:

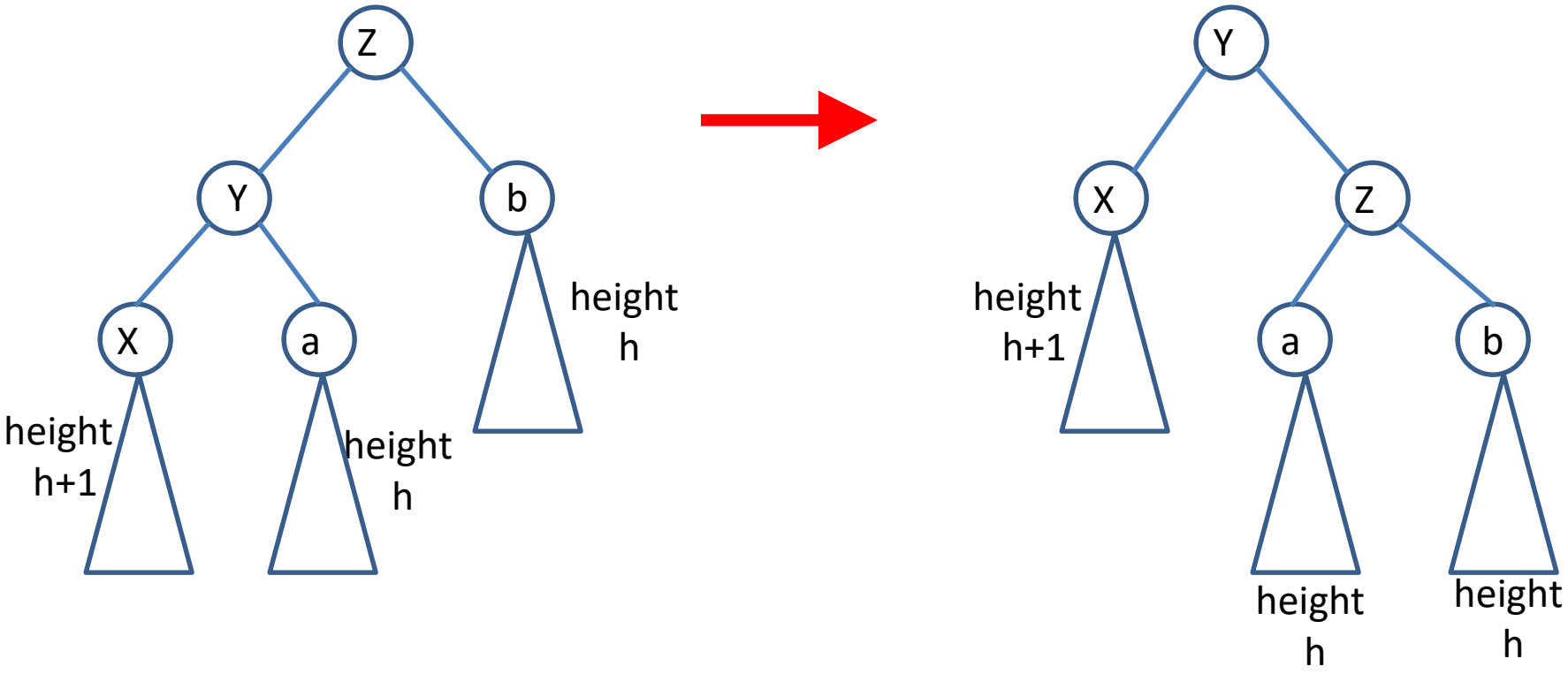
a) The insertion was in the left subtree of the left child of Z
b) The insertion was in the right subtree of the left child of Z
c) The insertion was in the left subtree of the right child of Z
d) The insertion was in the right subtree of the right child of Z

Consider case (a):



Suppose that after the insert the left child of Node Z has height h+2, while the right child has height h.

We modify this as follows:

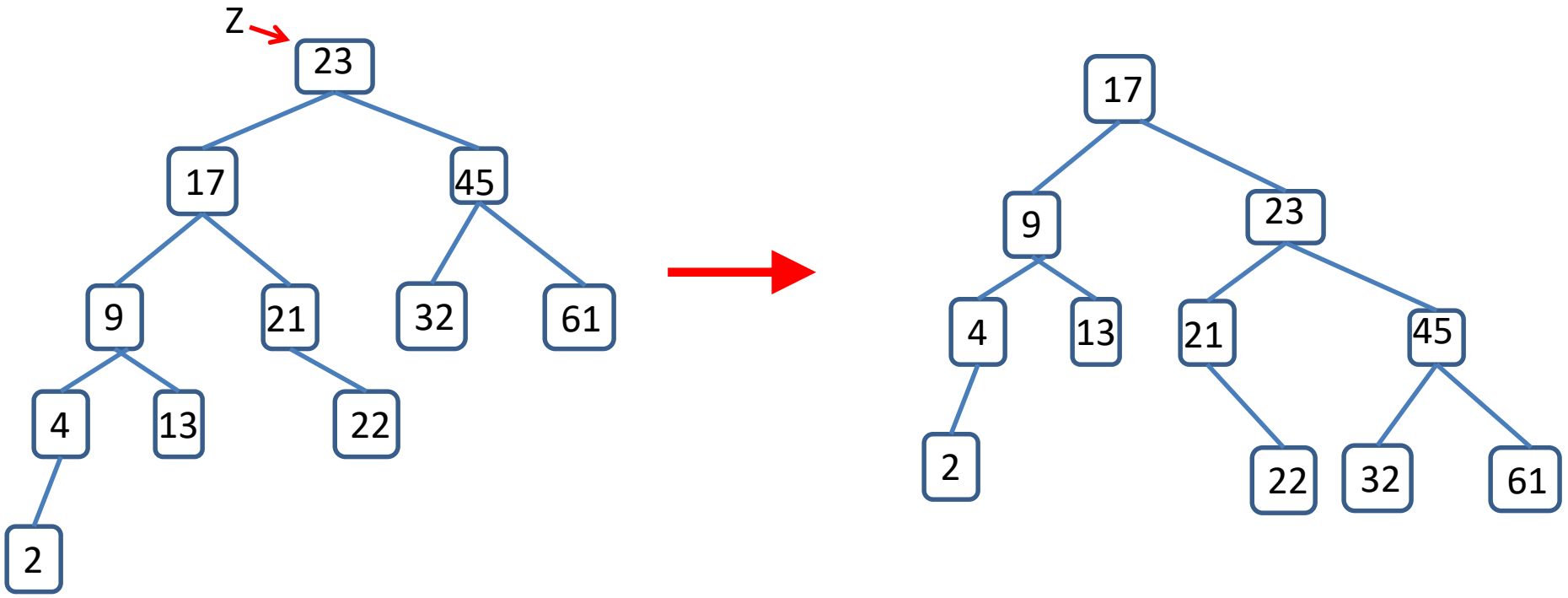


Note that we produce this new tree by just reassigning a few pointers, so it is quick to produce. It is easy to check that this is a BST tree, and it now satisfies the AVL property: the two children of the top node now have height h+1. Note that the top node has height h+2, which is what it had before the insertion, so no further changes are needed in the tree.

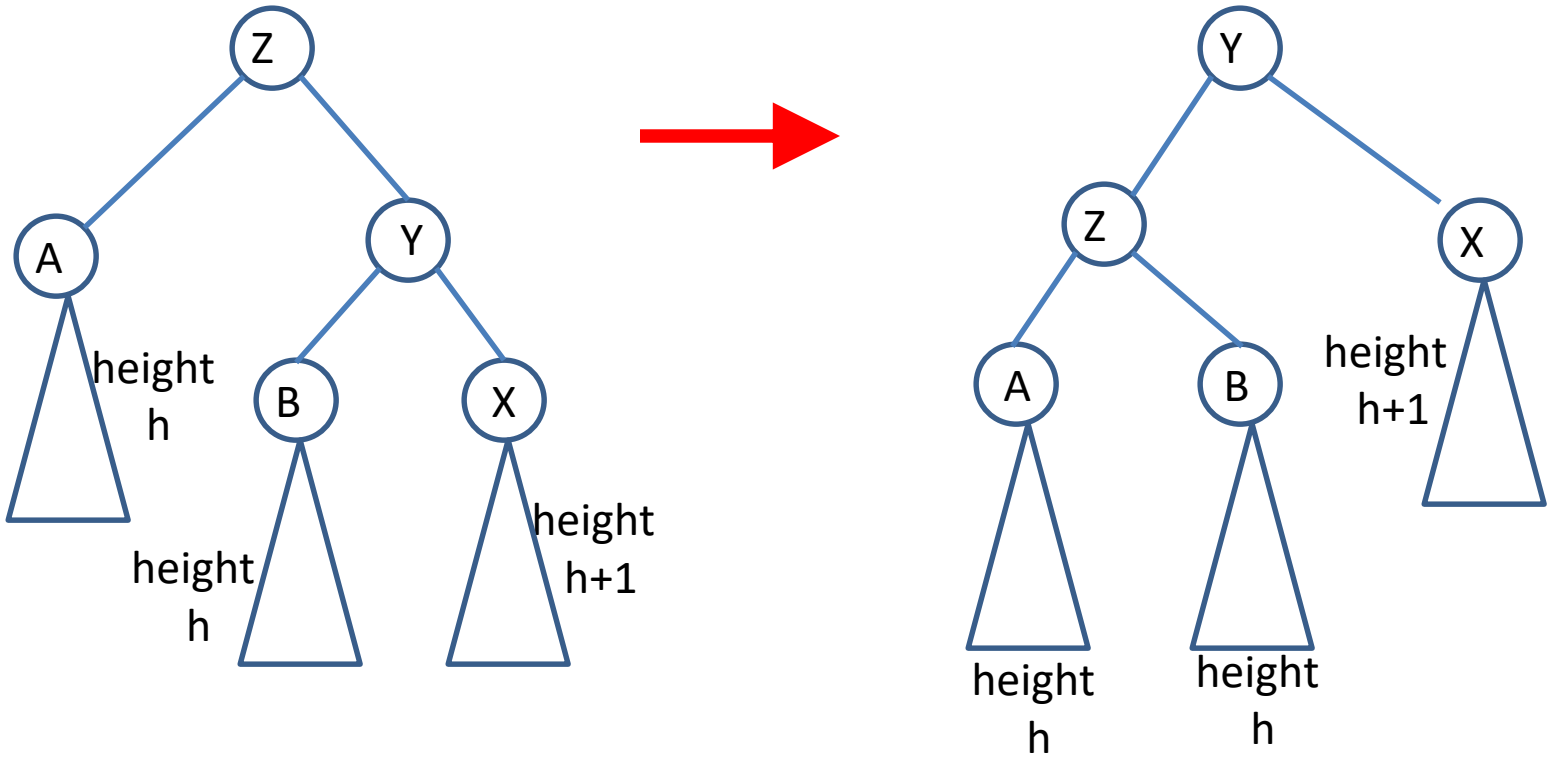
Here is code for this:

```
Node rotateWithLeftChild( Node Z) {
        Node Y = Z.left;
        Z.left = Y.right;
        Y.right = Z;
        return Y;
}
```

Example:



There is a symmetric case for inserting into the right subtree of the right child of Z:



The code for this is:

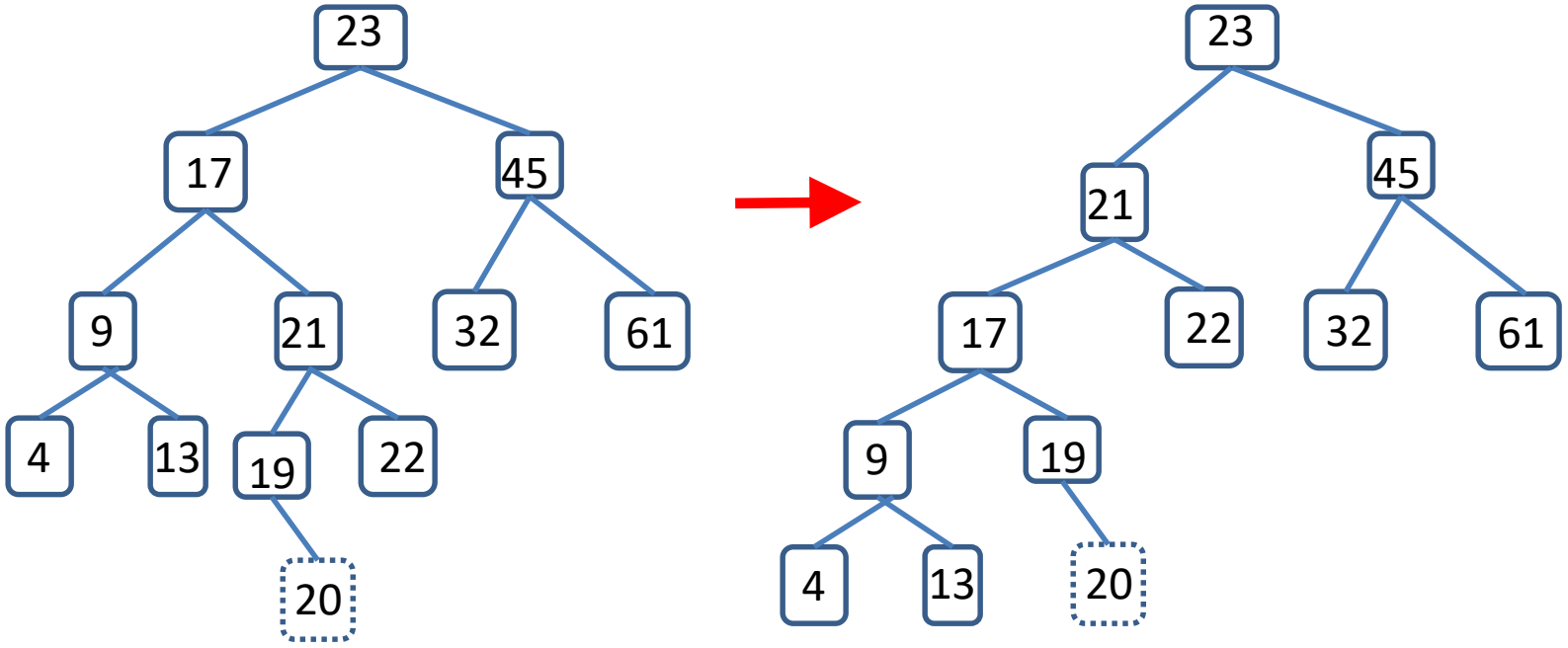
```
Node  rotateWithRightChild( Node Z) {
        Node Y = Z.right;
        Z.right = Y.left;
        Y.left = Z;
        return Y;
}
```

We started out with 4 cases:

a) Insertion in the left subtree of the left child of Z
b) Insertion in the right subtree of the left child of Z
c) Insertion in the left subtree of the right child of Z
d) Insertion in the right subtree of the right child of Z

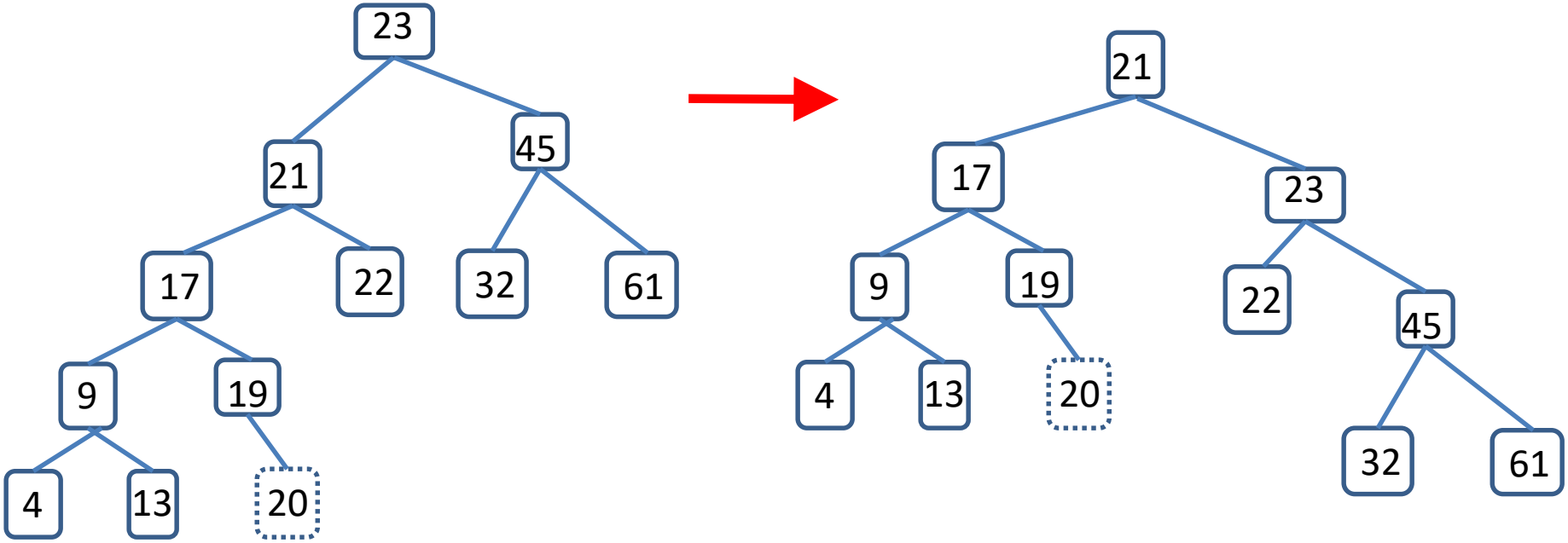
Our simple rotations have fixed cases (a) and (d). The other cases are a bit more complex and require a double rotation.

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

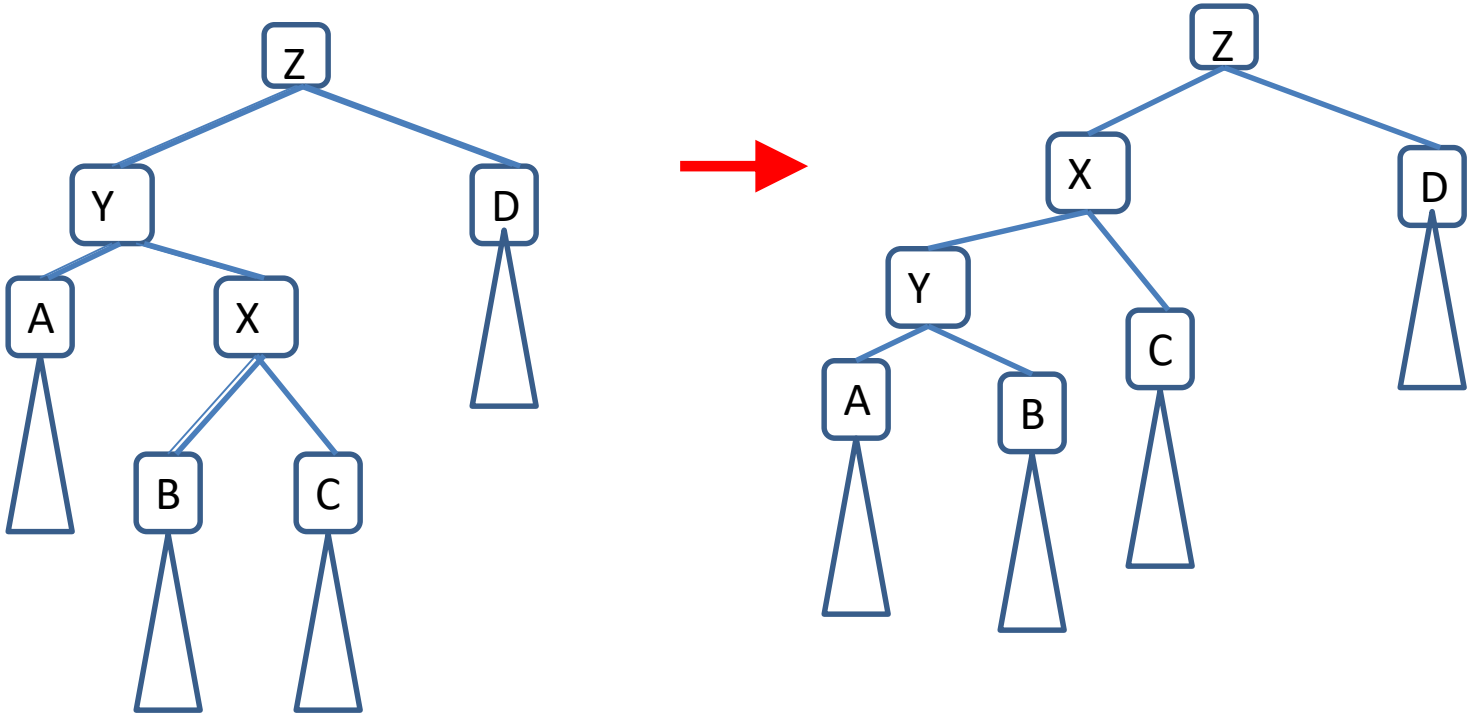


We first do a rotation from node 21 to node 17, but that doesn't fix the problem; the left child of 23 has height 3, the right child height 1.

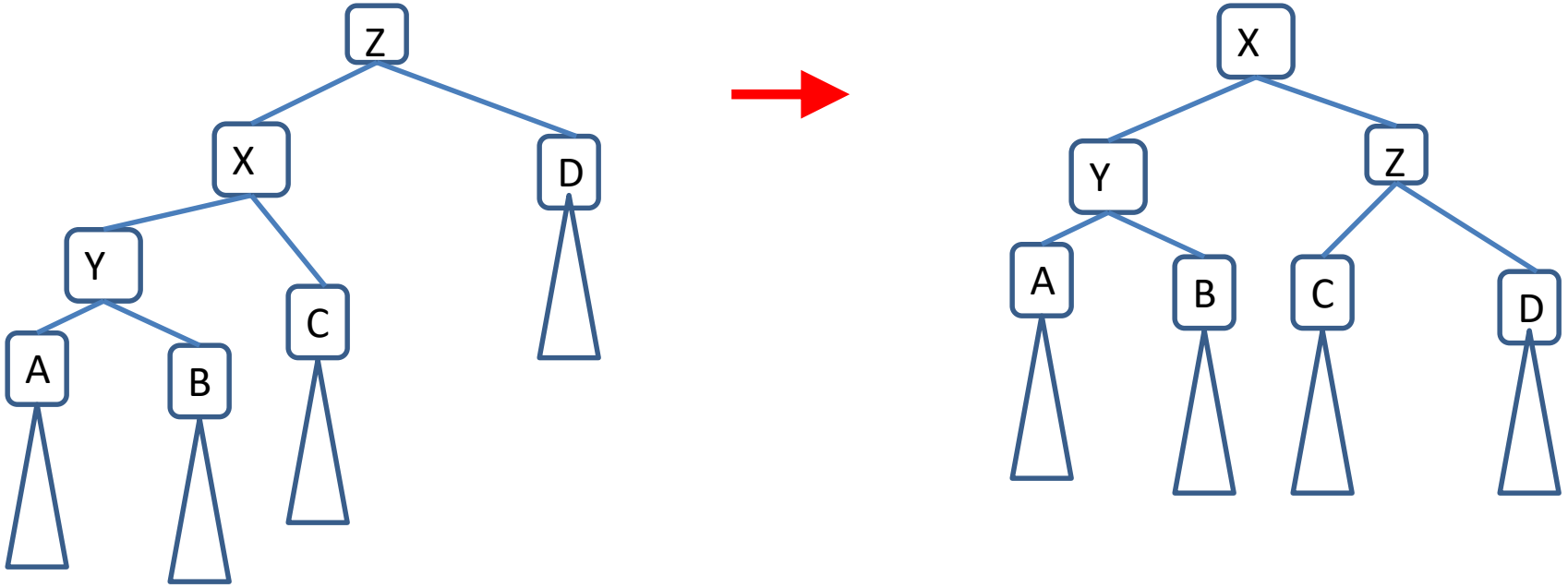
We then rotate node 21 to node 23 and this does fix the problem:



In general, if we have a situation like this, where we have inserted into the right child X of the left child Y of Z, we first rotate node X to Y:



We then rotate node X to Z:



Now both children of the root have height h+1.

The code for this is simple:

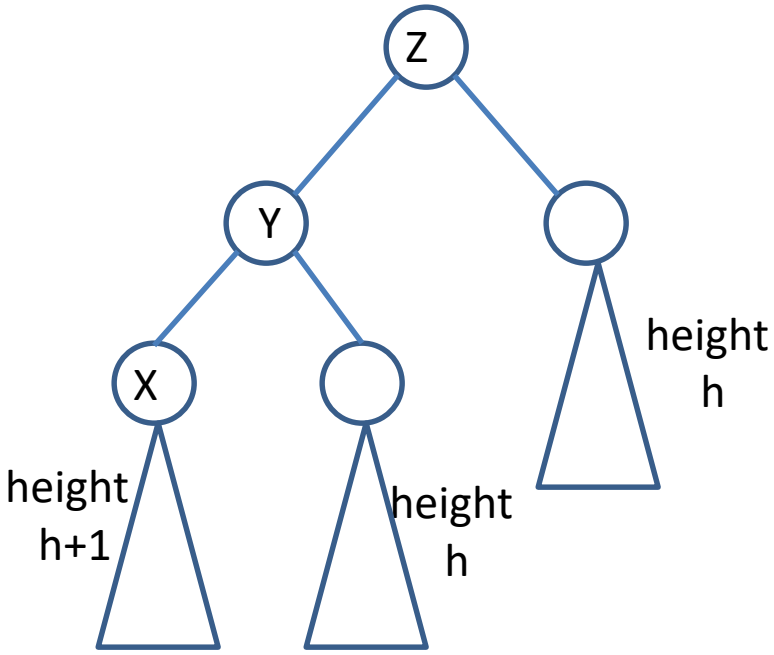
```
Node doubleRotationWithLeftChild( node Z) {
        Node Y = Z.left;
        Node X = Y.right;
        Z.left = rotateWithRightChild(Y);
        return rotateWithLeftChild(Z);
}
```

Of course there is a symmetric method:

```
Node doubleRotationWithRightChild( node Z) {
        Node Y = Z.right;
        Node X = Y.left;
        Z.right = rotateWithLeftChild(Y);
        return rotateWithRightChild(Z);
}
```

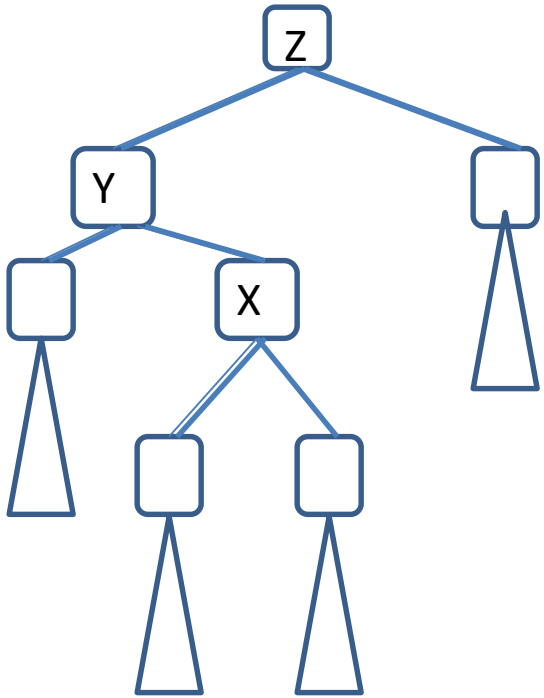
In Lab 6 we have slightly different notation for this. Let Z be the unbalanced node, let Y be Z's tallest child, and let X be Y's tallest child. Now introduce three new variables a, b, c where a is the one of X, Y, Z with the smallest value, b the one with the middle value, and c the one with the largest value.

For example, in this tree



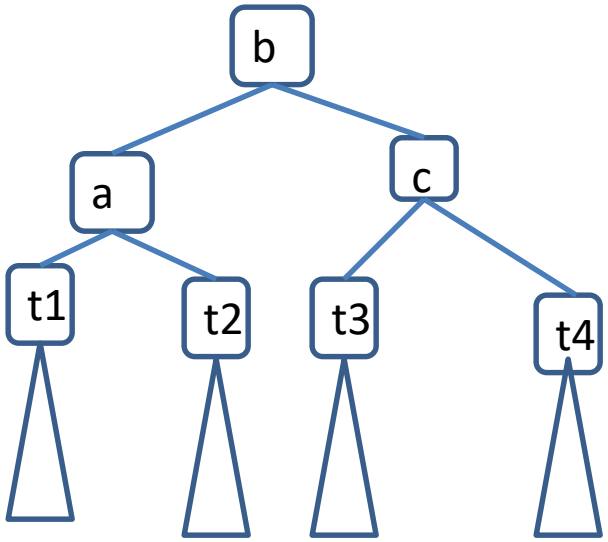
We have a is X, b is Y, and c is Z.

In this tree

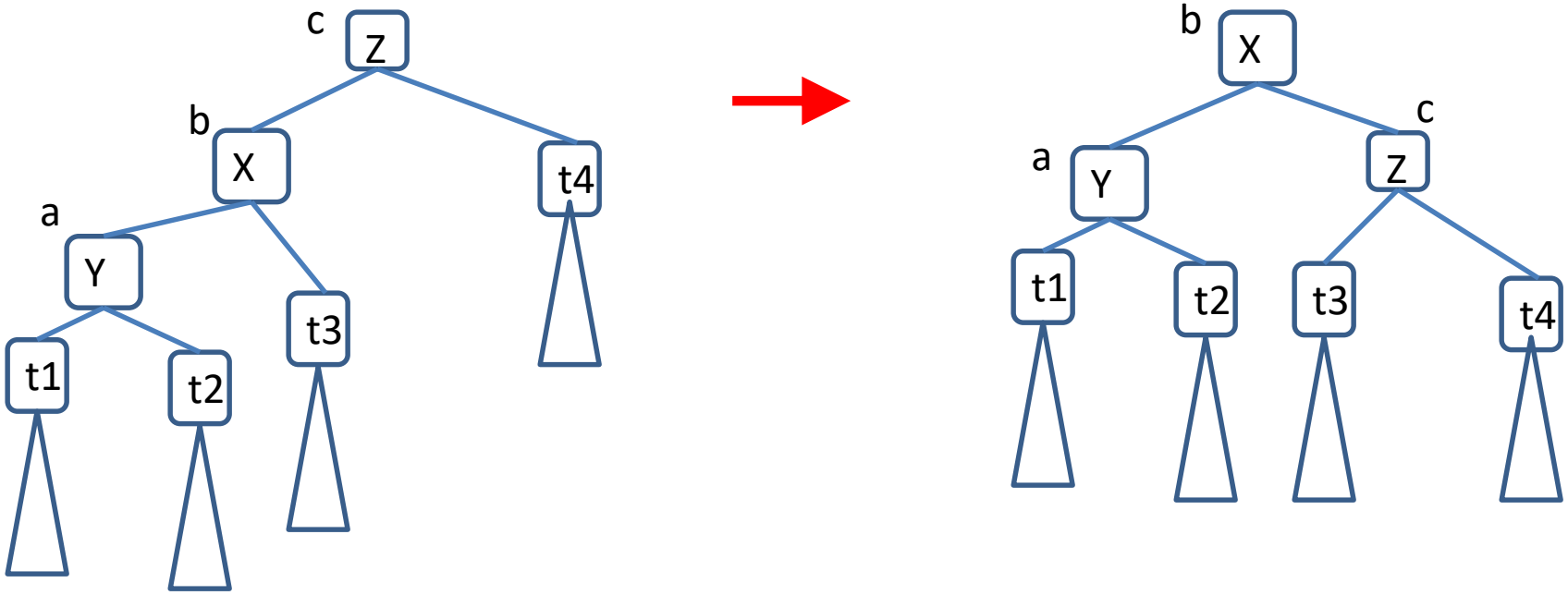


a is Y, b is X and c is Z.

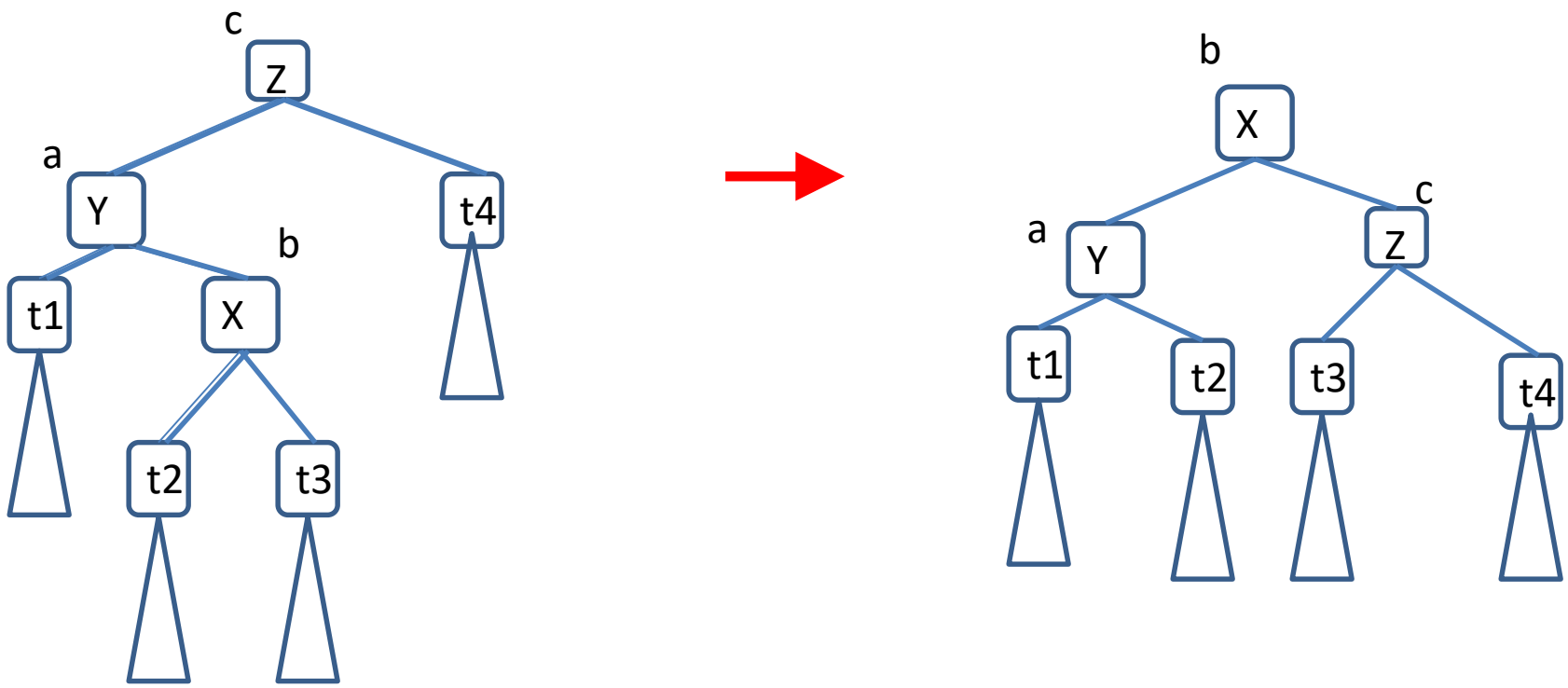
We then let t1, t2, t3 and t4 be the left-to-right children of nodes a, b and c. It turns out that all four cases of our rotation can be reduced to building the tree



For example,



Here is another example; the remaining two cases are symmetrical to these



In Lab 6, once you have identified nodes a, b, and c, and their children t1 through t4, code that is given to you completes the rotation by building the resulting tree in terms of these 7 variables.