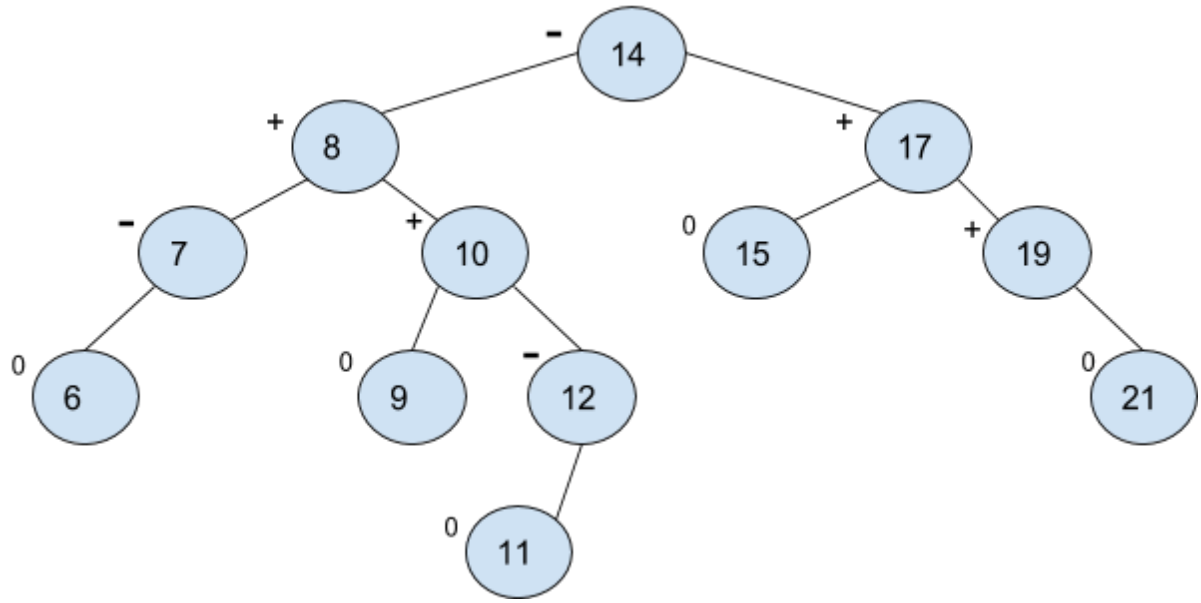
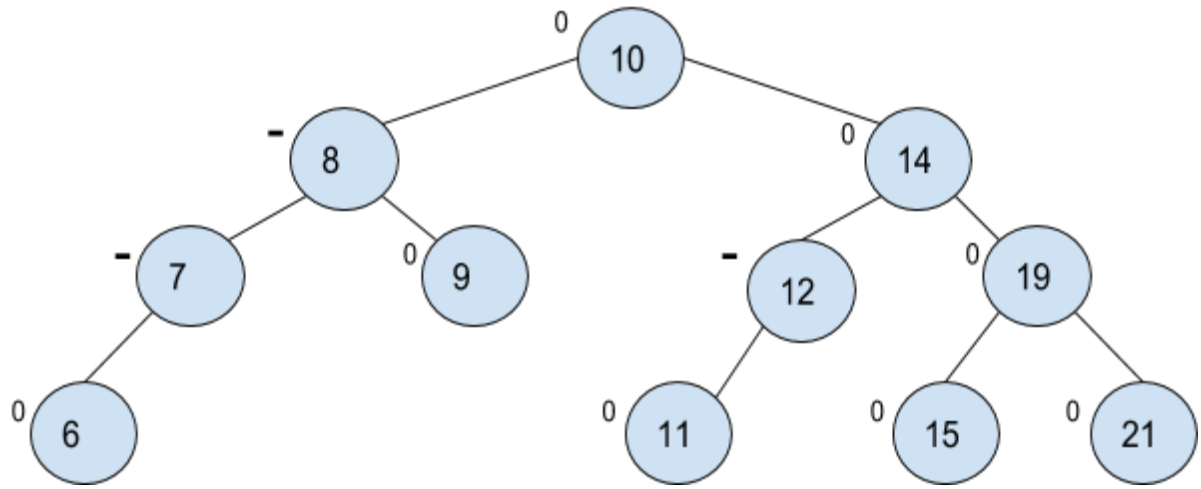
Question 1(Zhenjie Xiong):

//Verified by Xiaoxue Xing

a)



b)

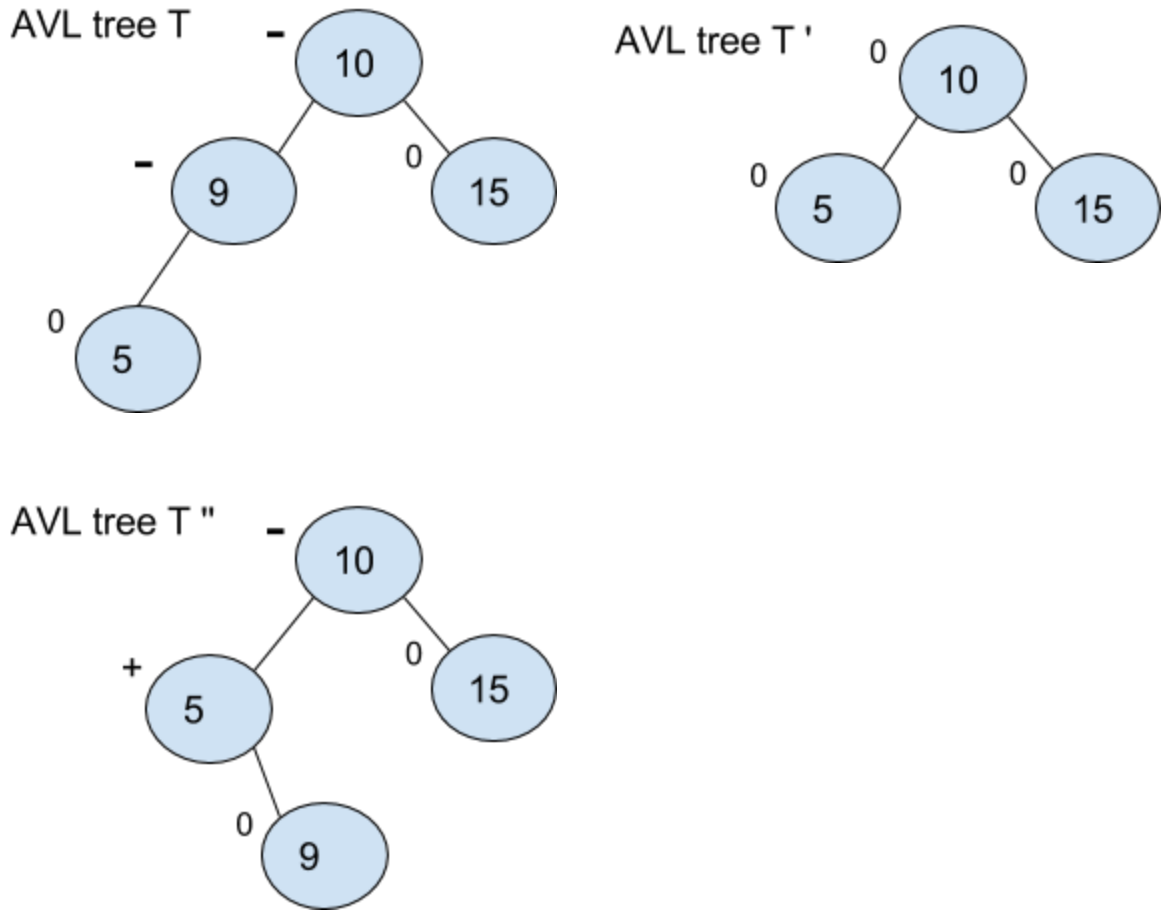


Question 2(Zhenjie Xiong):

//Verified by Xiaoxue Xing

a)

I disprove this statement.
One AVL tree example I found, that doesn't support the statement is drawn below. I choose the key x in T to be 9.

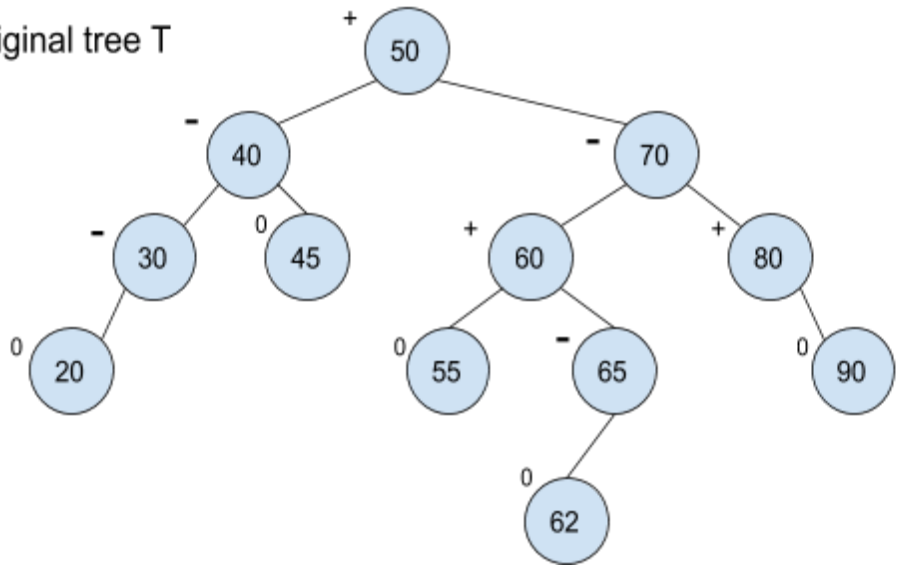


So I let T ' = DELETE(T, 9), and T " = INSERT(T ' , 9). As you can see T " is not the same as T. So the statement is incorrect.

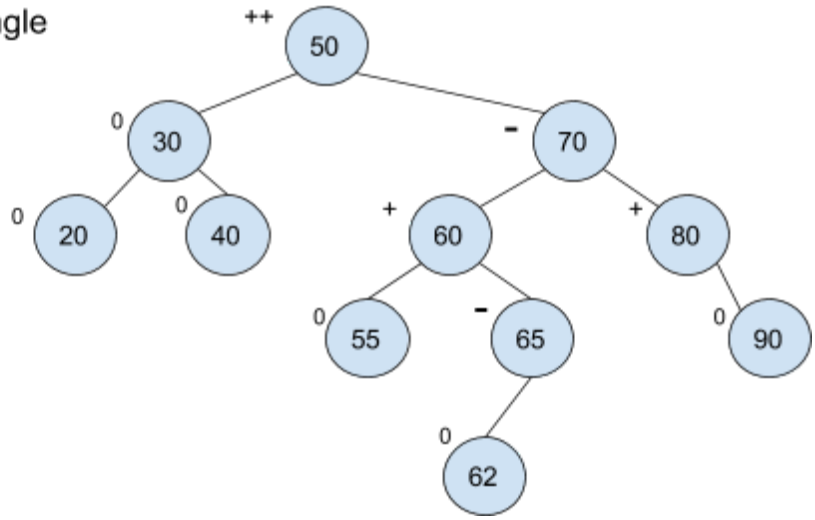
b)

The key x that's about to be deleted from the AVL tree T is: 45.

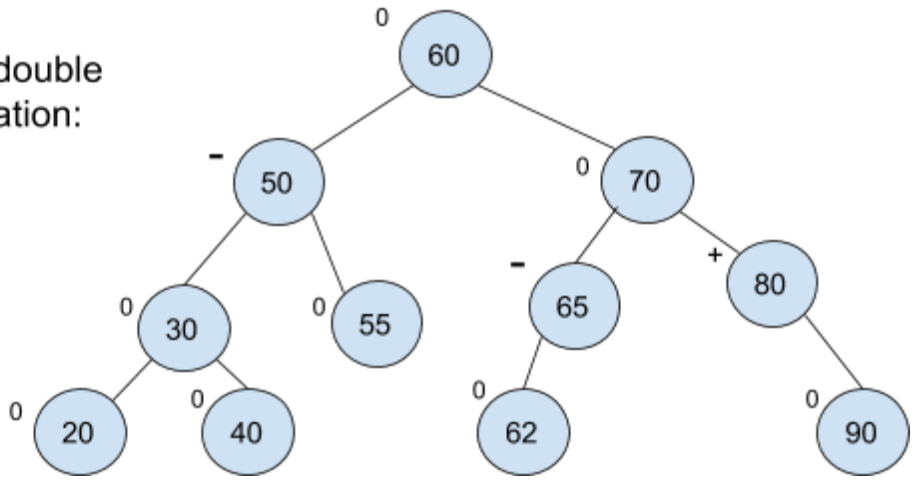
Original tree T



Result of a single right rotation:



Result of a double right-left rotation:



Question 3(Xiaoxue Xing):

//Verified by Zhenjie Xiong

```
IsBalanced(u) {
        return CheckBalance(root) != -1;      //return TRUE only if the value is NOT -1
}

CheckBalance(u){                              //a helper function
        if(u == null)
                return 0;                     //an empty leaf gives a height 0

        int left = CheckBalance(u.lchild);
        if(left == -1)                        //If left child is unbalanced
                return -1;

        int right = CheckBalance(u.rchild);
        if(right == -1)                       //If right child is unbalanced
                return -1;

        If (Math.abs(left - right) > 1) {     //height difference of children too large
                return -1;
        } else {
                return 1 + Math.max(left, right);          //return the height
        }
}
```

In this algorithm, we explore n nodes of the tree and visit each link exactly twice. Firstly, visit downward to enter the subtree rooted by that node. Then, visit upward to leave that node's subtree after exploring it.

Firstly, it is O(n).
Since there are n-1 linkages and n nodes in any AVL trees, so the worst case running time is 2*(n-1) + n that is 3n-2. Because for c > 4, c*n is always greater than 3n-2. In other words, because each node will be visited at least once, and n represents the number of node, the worst case running time should be O(n).

It is also Ω(n).
There exist an input where the root is 50, its left child is 25 and its right child is 75.
So the root is not empty, and its left and right child get recursed into the CheckBalance function. For each of them, they are not empty either. But because they have no children, the recursive function returns the height of 1+Math.max(0, 0), which is 1.

Then for the root, its left value and right value are both 1. Their difference is not greater than 1 so the height is returned, which is 1+Math.max(left, right) = 2. The IsBalanced function checks that 2 is NOT -1, so return true; it is indeed an AVL tree.
In this case, every node is visited at least once, so I've shown that there exist an input such that the worst case running time is Ω(n).

Question 4(Xiaoxue Xing):

//Verified by Zhenjie Xiong

1) We use AVL trees. The AVL tree contains m smallest keys among all the keys that already input to the algorithm.

2) Process each input key :
Every time we input a key, we first find the minimum number in the AVL tree. The minimum number of the tree is the left most element in the tree.
Then we compare the value between the minimum number in the tree and the new input key, if the input key is larger than the minimum value in the tree, we will delete the minimum number in the tree and insert the new key to the tree. If the minimum value is larger than the new key, we will change nothing to the tree.

Process each input query:
We are looking for the m smallest elements in an AVL tree. We do this by doing an in-order traversal of the tree recursively. We keep the current position of the array. We then store the node's value at the next current position of the output array.

3) Code for processing:

```
Build an empty AVL tree;
If key < minimum node of AVL tree
        Insert key into AVL tree
If key larger or equal minimum node of AVL tree
        Do nothing.
```

Code for query:

```
filling_array(root, array, position) {
        If the left rood != null:
                pos = filling_array(root.left, array, position);
        Array[next position] = root.element;
        If (root. Right != null)
                pos = filling_array(root.right, array, position);
        return position;
}
```

4) Process each input key :
To find the minmium number, the edges we need go through is h-1, where h is the height of the tree. Since the height is C*logn, so it takes O(log n) to go to the left most element.
The delete method in AVL tree is O(logn) and the insert method is also O(logn).
Therefore, for each input key, we need O(logn) +2*O(logn) cost, that is still O(logn).

Process each input query:
Because we traverse the left subtree, node, and right subtree to the extent of the number of m. The worst case running time should be O(m)