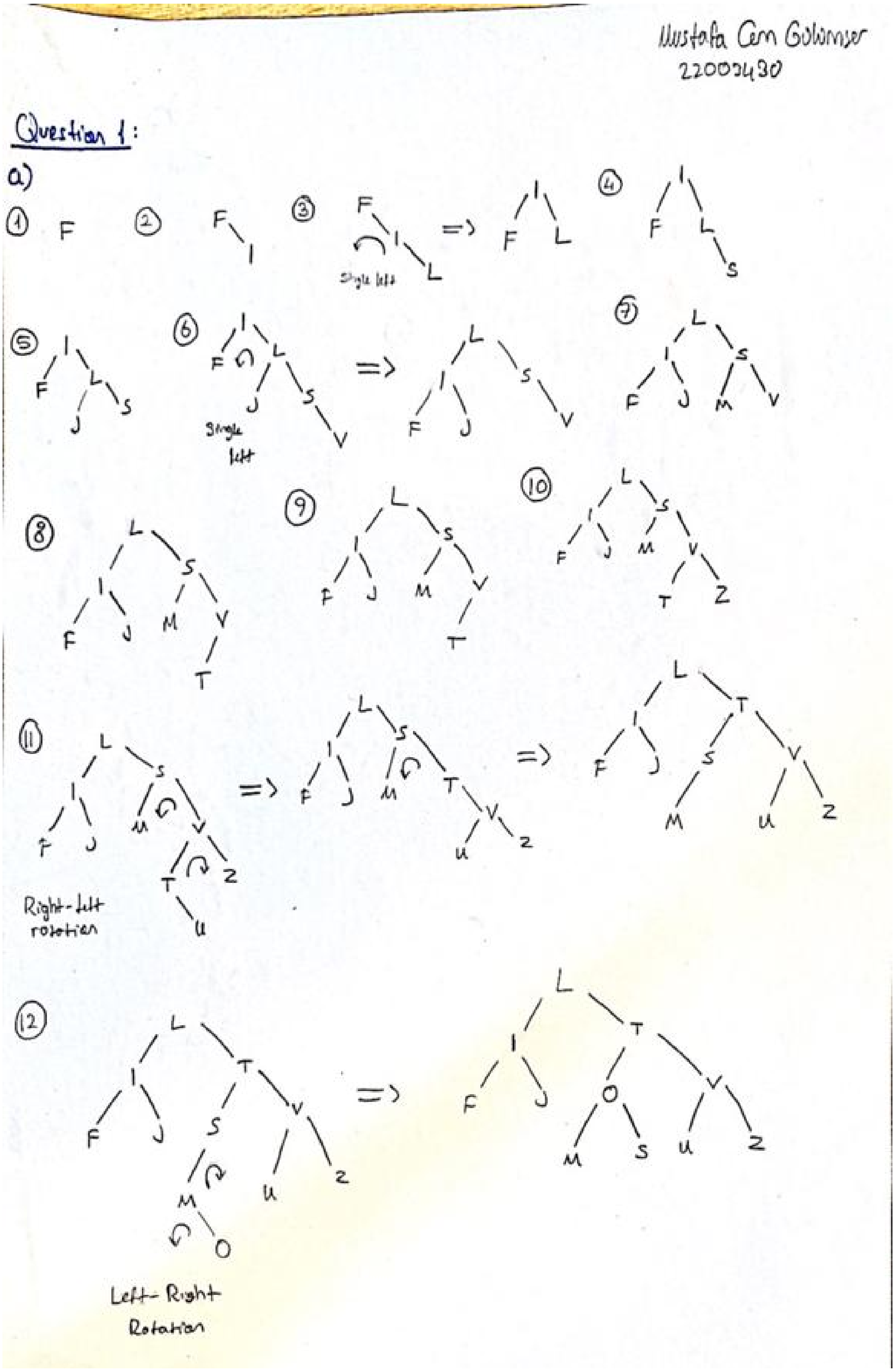
Question 1:



Mustafa Cem Gulumser
22002430

Question 1:

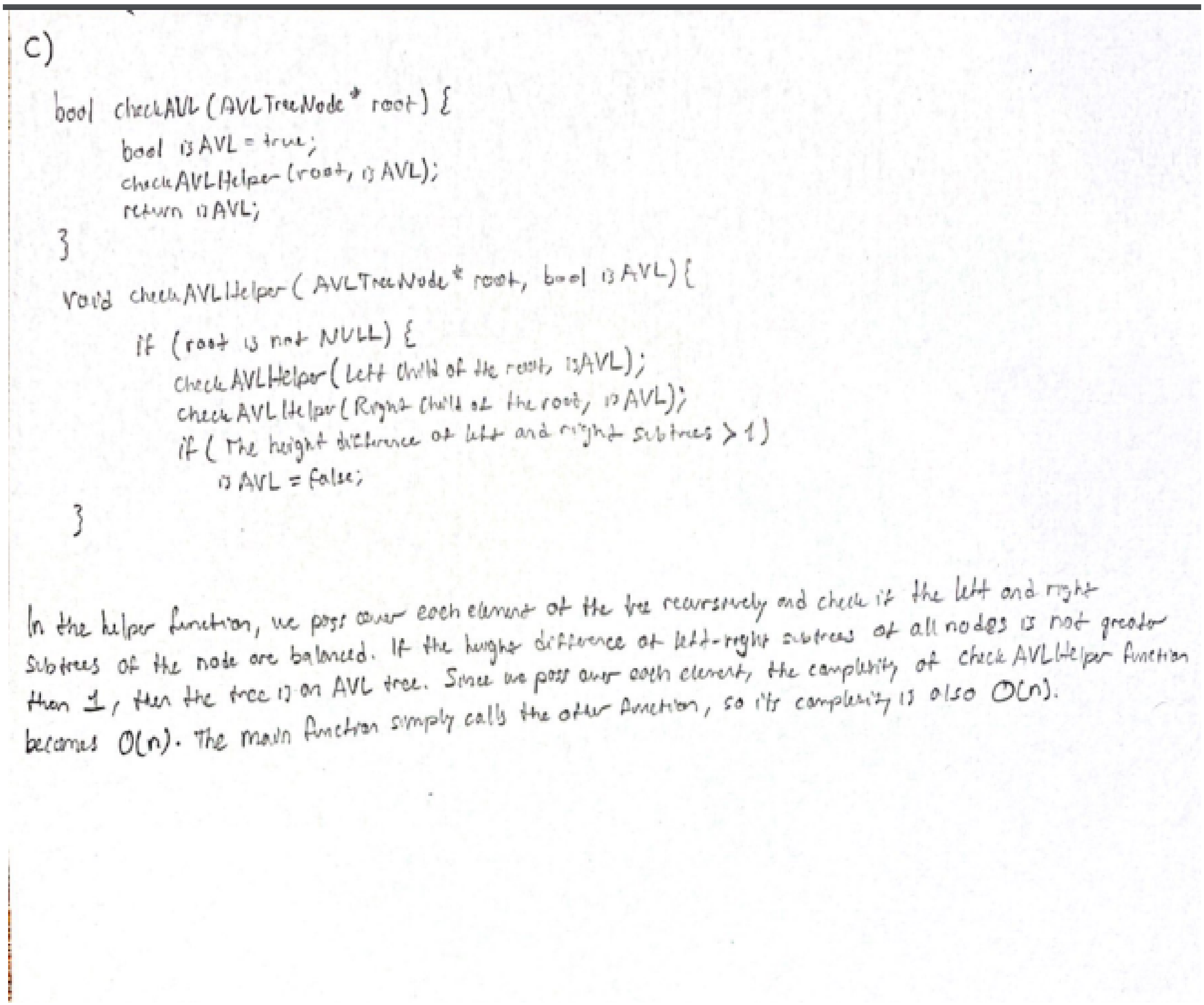
a)

b) For this algorithm to work, we assume that the node class of the AVL tree has an integer numberOfNodes variable that keeps the count of the number of nodes of the subtree with that node as the root. The node class also has an integer index variable, which is the index of the node when the elements of the tree are sorted (starting from 1).

```
int computeMedian( AVLTreeNode* root) {
    if ( Size of the root is odd) {
        medianIndex = (size of the root +1 )/2;
        return findMedianInorder( root, medianIndex);
    }
    else { medianIndex = (size of the root +1)/2;
        return ( findMedianInorder (root, medianIndex-1) + findMedianInorder (root, medianIndex ))/2;
    }
}
int findMedianInorder( AVLTreeNode* root, const int medianIndex) {

    if ( The left node of the root is not NULL)
     return findMedianInorder( left node of the root, medianIndex);
    if ( The index of the root is medianIndex)
         return the value of the node;
    if ( The right node of the root is not NULL)
     return findMedianInorder (right node of the root, medianIndex);   // The root is referred as the input node
}                                                                      // of the function
```

In the findMedianInorder method, we traverse the given AVL tree inorder, and the inorder traversal of an AVL tree yields the complexity O(n). The logic is that the median in an AVL tree is the n+1/2 th item in the sorted order. In the computeMedian method, we check the cases for an odd or even sized tree, because if the tree size is odd, the median is the value in the middle. Else, the median is the average of the values in the middle. So in the worst case, where the tree size is odd, we call findMedianInorder twice. The complexity therefore becomes $O(n) + O(n) = O(n)$.



c)

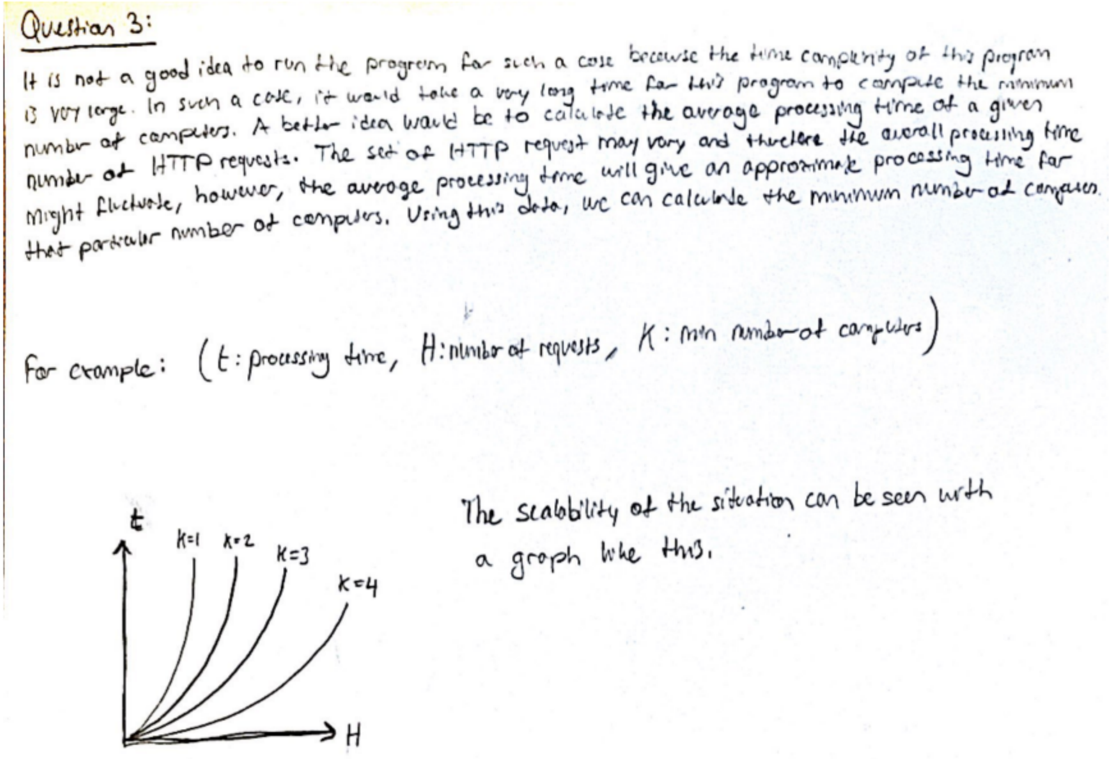
```
bool checkAVL (AVLTreeNode* root) {
        bool isAVL = true;
        checkAVLHelper (root, isAVL);
        return isAVL;
}
void checkAVLHelper ( AVLTreeNode* root, bool isAVL) {

        if (root is not NULL) {
            checkAVLHelper (Left child of the root, isAVL);
            checkAVLHelper (Right child of the root, isAVL);
            if ( The height difference of left and right subtrees > 1)
                isAVL = false;

}
```

In the helper function, we pass over each element of the tree recursively and check if the left and right subtrees of the node are balanced. If the height difference of left-right subtrees of all nodes is not greater than 1, then the tree is an AVL tree. Since we pass over each element, the complexity of checkAVLHelper function becomes O(n). The main function simply calls the other function, so it's complexity is also O(n).

Question 3:



Question 3:

It is not a good idea to run the program for such a case because the time complexity of this program is very large. In such a case, it would take a very long time for this program to compute the minimum number of computers. A better idea would be to calculate the average processing time of a given number of HTTP requests. The set of HTTP request may vary and therefore the overall processing time might fluctuate, however, the average processing time will give an approximate processing time for that particular number of computers. Using this data, we can calculate the minimum number of computers.

For example: (t : processing time, H : number of requests, K : min number of computers)

The scalability of the situation can be seen with a graph like this.