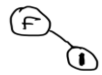
Question 1:

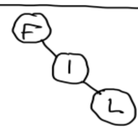
a) 1. Insertion (F)

2. Insertion (I)

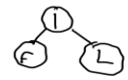


3. Insertion (L)

Before Rotation

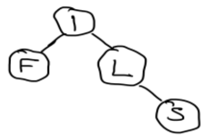


After Rotation

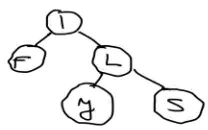


Rotation Type:
Single Rotate Left

4. Insertion (S)

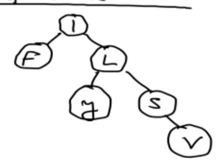


5. Insertion (J)

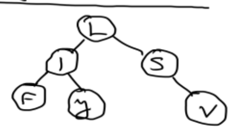


6. Insertion (V)

Before Rotation

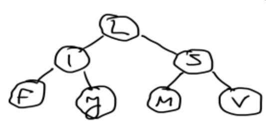


After Rotation

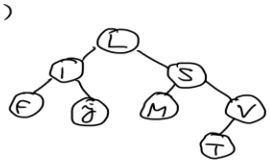


Rotation Type:
Single Rotate Left

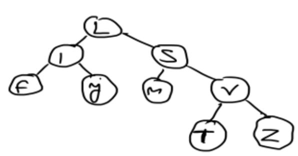
7. Insertion (M)



8. Insertion (T)

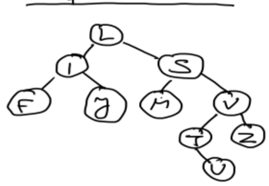


9. Insertion (Z)

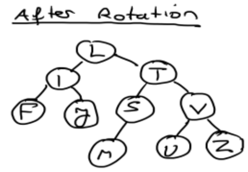


10. Insertion (U)

Before Rotation



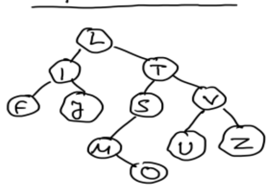
After Rotation



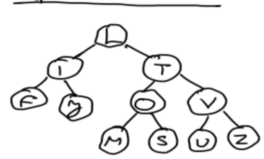
Rotation Type:
Double Rotate Left

11. Insertion (O)

Before Rotation



After Rotation



Rotation Type:
Double Rotate Right

Part B:

For the following code, I assume that treenodes has variables that stores the sizes of the subtrees of it. Then, we can find the median value in O(log n) time since the tree is balanced.

```
double computeMedian( TreeNode* root, int excessFromPrev){
      int totalsize = 1 + root.leftsize + root.rightsize + excessFromPrev;
      double median = totalsize /2;
      if ( median <= root.leftsize) {
        return computeMedian (root->left, total size - root->leftsize);
      } else if (median <root->leftsize+1) { // it is between left and root
           return (root->data + root->left->data) /2;
      } else if( median == root->leftsize+1) {
          return root->data;
      } else if (median < root->rightsize){ //between root and right
          return (root-> data + root->right -> data) / 2;
      } else {
          return computeMedian(root->right, root->rightsize - totalsize);
      }
}
```

ExcessFromPrev is required, think 11 elements 7 left 3 right. 5 is median, we traverse to the left. It should know that it needs to return 6 rather than its own median 4. Same is applied to the right with only change that we subtract previous value. Think 11 element 7 right 3 left. we traverse right. It should return second one, as we give parameter -4 it should select 2 from its element, 6 for the actual tree.

Part C:

```
int  checkAVL ( TreeNode* currentNode)
     if ( currentNode == NULL)
        return 0;   // height 0
     // check right subtree
     int rightSubtreeHeight = checkAVL(currentNode.right);
     if (rightSubtreeHeight == -1) return -1;
     //check left subtree
     int leftSubtreeHeight = checkAVL(currentNode.left);
     if (leftSubtreeHeight == -1) return -1;
     //check the difference between right and left for current node
     if (Math.abs(rightSubtreeHeight - leftSubtreeHeight) > 1)
        return -1;
     return (Math.max(rightSubtreeHeight, leftSubtreeHeight) + 1);
```

Above code will return -1, if the BST is not AVL tree; otherwise, it is AVL tree.
I use the fact that all AVL trees is a balanced BST.
The code check if right and left subtrees are AVL tree, if they are, then it checks absolute value of their balance factors. If balance factor greater than 1 or less than -1, tree is not balanced, not AVL trees. If everything well, the code returns the height of the current node.

worst-case time complexity ⇒ O(N)
Average case time complexity ⇒ O(log N)
Best case time complexity ⇒ O(log N)

In the worst case, BST is AVL tree, we need to look at all elements in BST.
In the best case, right subtrees are not AVL tree, we do not need to look at left subtrees. Therefore, our search decrease in logarithmic.
Average case is between O(log N) and O(n), we cannot say anything certain.

Question 3:

We can assume that when the computer number is increasing, the average waiting time is decreasing. Then we can use search algorithms like binary search to find the optimum number of computers. That way time complexity becomes $O(\log n)$ from $O(n)$. In binary search, if the current element has less average waiting time that we want, we go left and otherwise right. When we go left, if the medium value of the left is higher than what we want, then we go right to find optimum value.