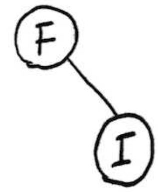
Berk Çakar - 22003021 - CS202-3

Homework 3 - Heaps, priority Queues and AVL Trees

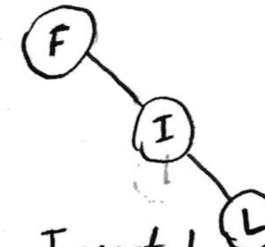
Question 1

a)

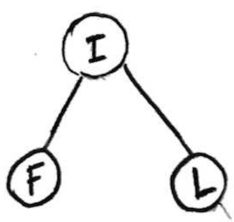
Insert F



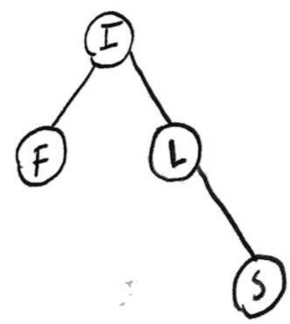
Insert I



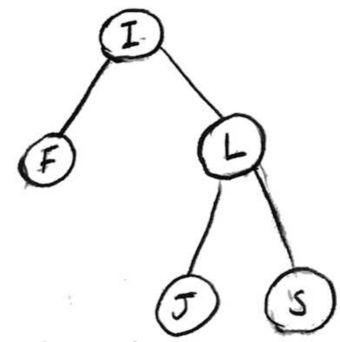
Insert L



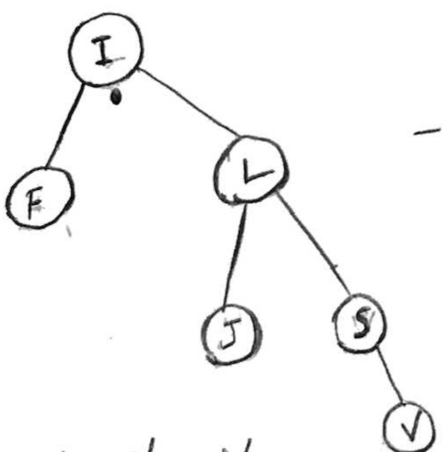
Left Rotation Around F



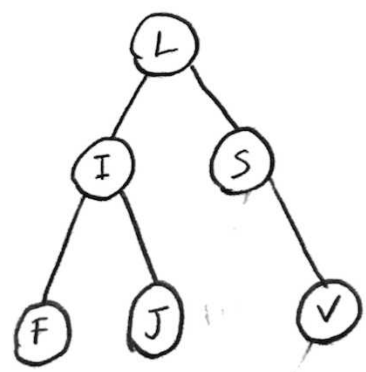
Insert S



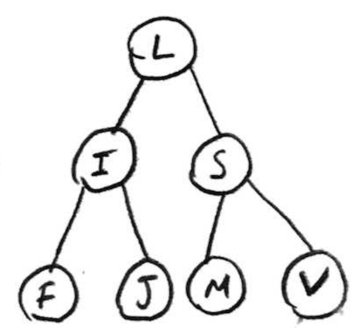
Insert J



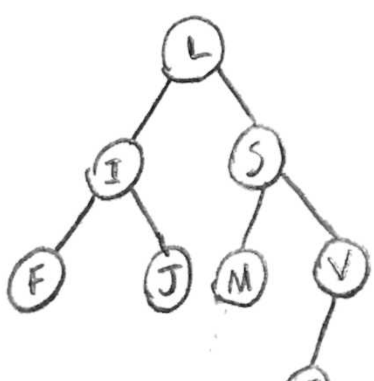
Insert V



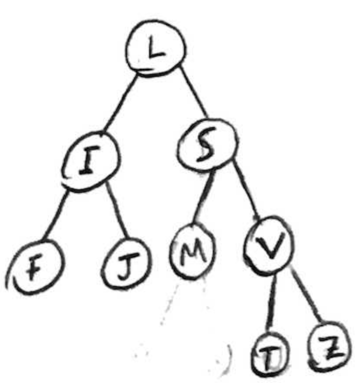
Left Rotation Around I



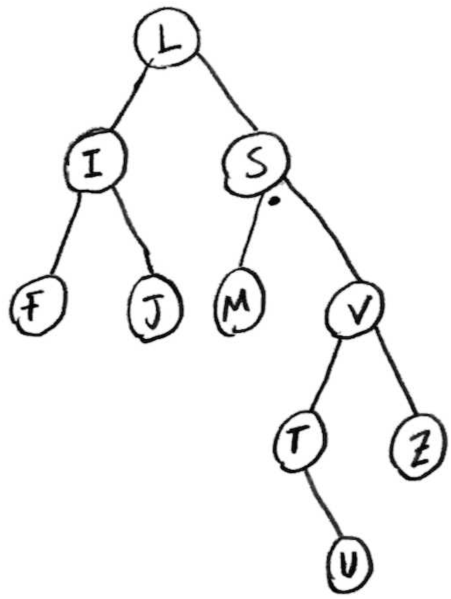
Insert M



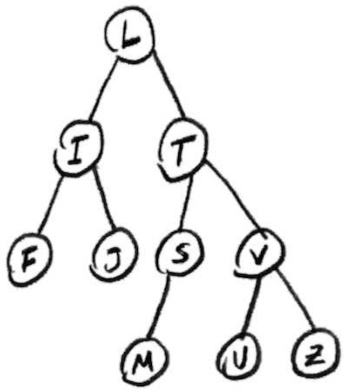
Insert T



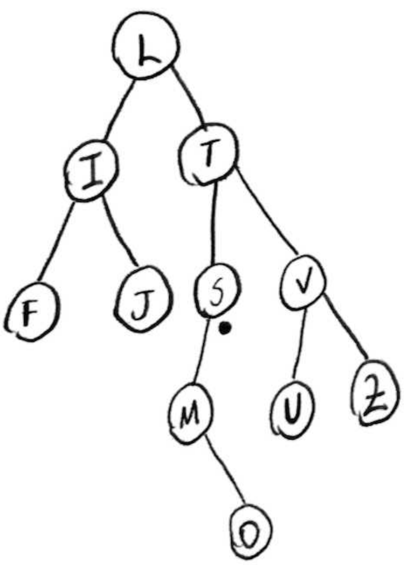
Insert Z



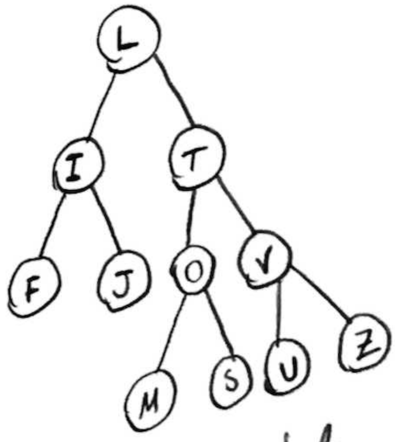
Insert U



Double Right-Left Rotation around the subtree rooted with S



Insert O



Double Left-Right Rotation around the subtree rooted with S

b)
```
struct Node {
    int size;   // I added size attribute to get the size
                // of AVL Tree in O(1) and for the select function
    int data;
    Node* left;
    Node* right;
}

double avlMedian (Node* root) {
    if (root == NULL || (root != NULL && root->size == 0))
        return 0;

    if (root->size is ODD) {
        return select(root, (root->size)/2);
    }
    else { // root->size is EVEN
        return (select(root, ((root->size)-1)/2)
            + select(root, (root->size)/2)) / 2;

}

int select (Node* root, int index) {
    if (root == NULL) { return 0; }
    if (root->left != NULL && root->left->size == index+1) {
        return root->data;
    }
    else if (root->left != NULL && index+1 > root->left->size) {
        return select(root->right, index + 1 - (root->left->size));
    }
    else {
        return select(root->left, index);
    }
}
```

First of all, whole logic entirely depends on the size property of the tree. We know that median elements are located middle of a sorted array. Using a similar logic, in "avlMedian" function, we determine the location of the median nodes and send those nodes to a select function. After select function does its job we return the median value to caller. The "select" function is an improved version of the regular "select k-th minimum item in a BST" function. By using the size property, our function tries to locate the corresponding indices in left or right subtrees. If the passed index equals to left subtree's size that means the correct location is found. On the other hand, if the index is bigger than left subtrees size we continue our search in right subtree. Conversely, if the index is smaller than the left subtree's size, it indicates that the k-th smallest element is somewhere in the left subtree. As can be seen, one half of the given tree is being searched at a time. Since we are searching on AVL Trees, this search is tightly bounded by $\Theta(\log N)$, because AVL Trees are balanced everytime. Therefore, using the given implementation, since we are searching only one half of the tree at a time and the tree is an AVL tree, computing the median of a given AVL Tree has a time complexity of $O(\log N)$, where N is the number of nodes.

c)

```
bool checkAVL (Node* root) {
    if (root == NULL) {
        return true;
    }

    return abs (getHeight (root->left) - getHeight (root->right))
            <= 1 && checkAVL (root->left) &&
        checkAVL (root->right);
}

int getHeight (Node* root) {
    if (root == NULL) {
        return 0;
    }

    return max (getHeight (root->left), getHeight (root->right)) + 1;
}
```

checkAVL uses a similar algorithm which is used for detecting if a given binary search tree is balanced. In that function, for every node in the AVL tree, we are checking if the node is balanced, which means that the height difference between its left subtree and right subtree can only be -1, 0 or 1. Since checkAVL visits every node in the tree, just visiting nodes takes $O(n)$ time. But the function also recurs into getHeight function for every node. And because of getHeight function also has a time complexity of $O(N)$, checkAVL function becomes bounded by $O(N^2)$ in worst case. We can eliminate the time required for calculating the height of every subtree with adding size property to nodes. In that case, checkAVL would be proportional to $O(N)$ at worst case.

$N \longrightarrow$ Number of nodes in the AVL Tree

Question 3

For very large request numbers, time complexity becomes quite high if timing values are compared for all computers from 1 to N in order to find the optimum number of computers, thus it would not be a good idea to follow such a way. Namely, the worst-case time complexity of the program would be $N * O(s) = O(N*s)$ since the simulation would run N times in the worst case, where s is the time complexity order of the simulation (for example, s can be linear, but this depends on the implementation of the simulation) As a better strategy, we can change how many computers we test. For this, a method similar to binary search can be used as the numbers from 1 to N increase. Starting from N/2, if the average time given by this amount of computers in the simulation is greater than K, another middle element (i.e 3N/4) can be selected from the right side of N/2. On the contrary, if the time obtained as a result of the simulation is less than K, another middle element (i.e. N/4) can be selected from the left side of N/2. Understandably, these steps are repeated recursively, and as soon as the next move to the left is known to be insufficient, the search can be stopped and that N value can be used as the optimum number of computers. As a result of this procedure, we would have run the simulation at most logN times, and the overall worst-case time complexity would have been $\log N * O(s) = O(\log N*s)$, which is better than having $O(N*s)$.