CS 202 Homework #3

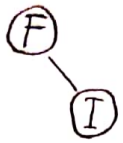
Boris Tan Ünal
22003617
CS202-03

Question #1:

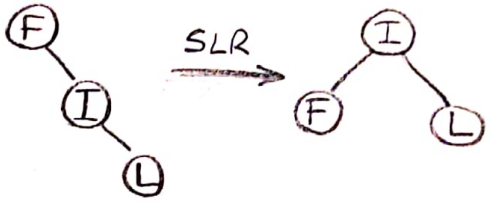
(a)

#1: add (F)

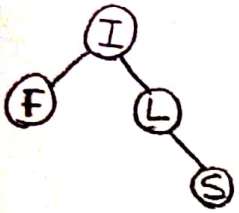
#2: add (I)



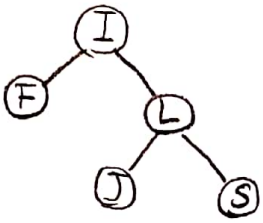
#3: add (L)



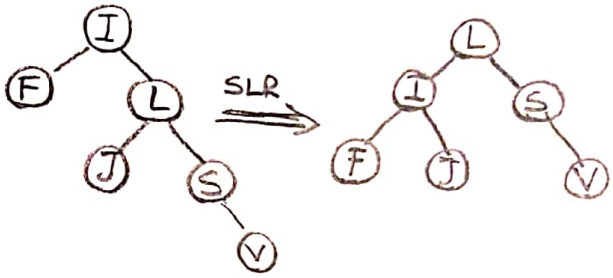
#4: add (S)



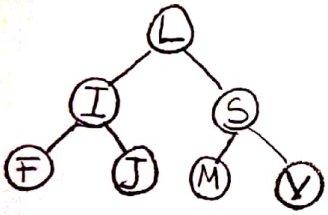
#5: add (J)



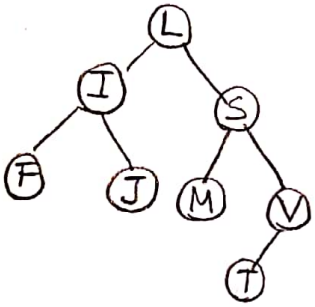
#6: add (V)



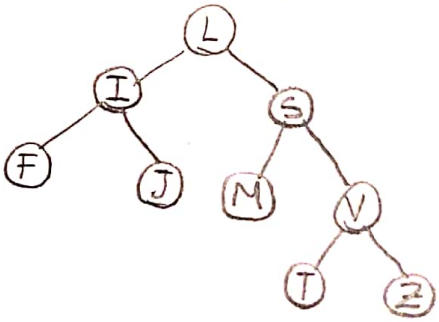
#7: add (M)



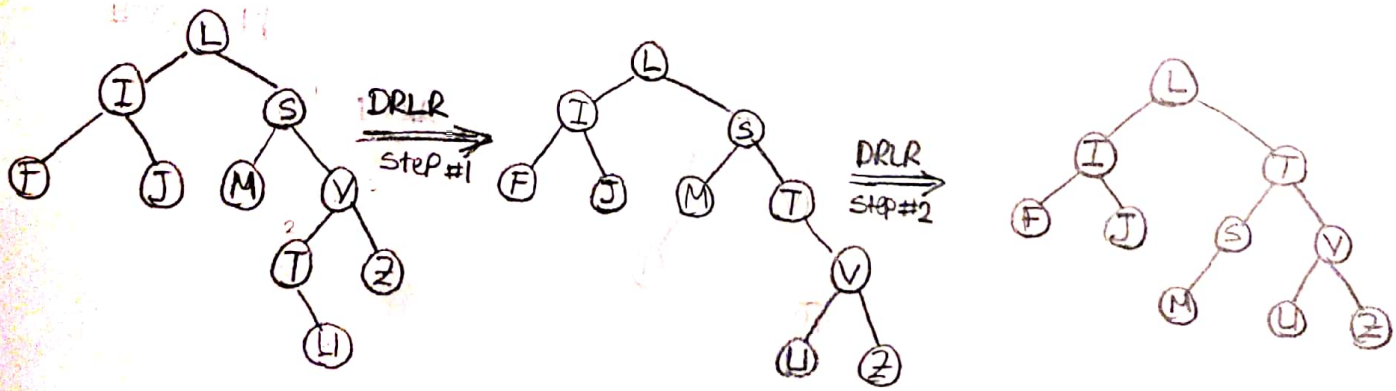
#8: add (T)



#9: add (Z)



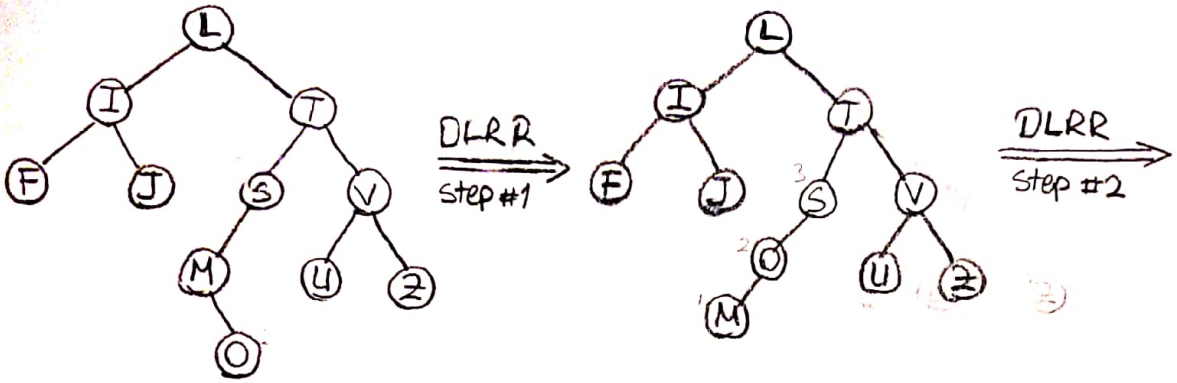
#10: add (U)

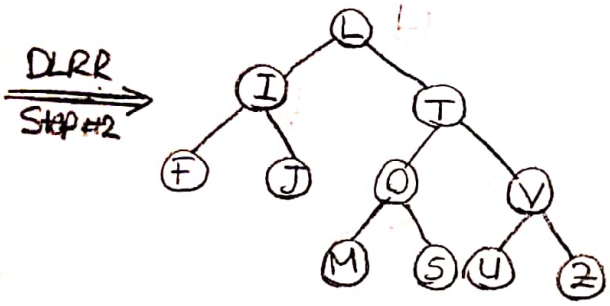


Question #1 (cont')

Barış Tan Ünal
2209367
CS202-03

#11 : add (O)





Notes :

- SLR : Single Left Rotation
- SRR : Single Right Rotation
- DLRR : Double Left-Right Rotation
- DRLR : Double Right-Left Rotation

Question #1b: Median of a given AVL tree.

Boris Tan Chad
22003617
CS-202-03

```
FUNCTION computeMedian ( TreeNode * root )
    CALL inorderTraverse to create an inorder traversed array
    IF the size of the array is odd
        RETURN arr [ size/2]
    ELSE
        RETURN the average of arr [size/2] & arr [size/2 - 1]
    END IF-ELSE
ENDFUNCTION


FUNCTION inorderTraverse ( TreeNode * root, int * arr)
    IF root is null
        RETURN
    ELSE
        CALL inorderTraverse for left subtree
        SET the next element of arr to root
        CALL inorderTraverse for right subtree
    END IF-ELSE
ENDFUNCTION
```

Note: This implementation has $O(n)$ complexity and it is not the most efficient algorithm in the terms of time complexity. Since it was not asked us to implement the most efficient algorithm, I implemented the easiest algorithm to understand & implement. However $O(\log n)$ is possible too.

Explanation: This algorithm traverses the tree in an inorder fashion to create an array. This contributes $O(n)$ time complexity since we visit all elements of the array. Then the algorithm computes the median for two possible cases. If the array size is odd, then the median is the middle, if it is even then it is the average of two middle elements.

Question #1.c : Check if AVL.

Barış Tan Ünal
22003617
CS292-03

```
FUNCTION checkAVL (TreeNode * root)
    IF root is null
        RETURN true
    ELSE IF height of the left and right subtrees differ more than one    //CALL getHeight
        RETURN false
    ELSE
        RETURN (CALL checkAVL for right subtree &&
                CALL checkAVL for left subtree )
    END IF-ELSE
ENDFUNCTION

FUNCTION getHeight (TreeNode * root)
    IF root is null
        RETURN 0;
    ELSE
        RETURN the height of the taller subtree plus one          //CALL getHeight
    END IF-ELSE
ENDFUNCTION
```

Explanation: This recursive function checks the height of a given root's left and right subtrees with the help of a helper function getHeight. The recursive call continues until either the house reached the leaf of the given tree, or the height of left and right subtrees differ more than one, which returns false.

Time Complexity: • Worst case: $O(n^2)$ because we might have to visit each node (which contributes to n iterations) and check their height (which again contributes to n iterations): $n \times n = n^2$

Barış Tan Ünal
22003617
CS 202-03

Question #3: Scability.

Instead of trying simulation with linearly increasing numbers such as 1, 2, 3, 4 ..., we can start testing from N/2 computers and try to find the suitable number of computers by performing a binary-search-like fashion. It would decrease the time complexity to $O(\log n)$ instead of $O(n)$.