Home Work 3
CMSC 341
Joud Omar

1 - Write a recursive C++ function which runs in O(n) for printing the depth of all the nodes of a BST, where n is the number of nodes of the tree. The running time of O(n) means we visit every node only once. Assume the node class name is “Node,” and every node has two member variables “m_left” and “m_right” which point to the left and right children of the node. You can use any name and any signature for the function. You can use the example BST class that is posted in Blackboard to run your function. Your function would be a member function of BST class. Only submit the function. The depth of a node is the number of edges from root to the node. For example, the depth of the root is zero. (25 points)

```
void BST::helperFunction(){
    preOrder(m_root,0);
}
int BST::preOrder(Node* root, int depthCounter ) {
    if (root != nullptr) {
        cout<<root->m_value<<" ";
        cout <<"Depth: " <<depthCounter <<endl;
        depthCounter++;
        preOrder(root->m_left,depthCounter);
        preOrder(root->m_right,depthCounter);
    }
}
```

changed type to int

```
input: int inputSrt[] = { 11, 5, 7, 4 ,19,9,20,30,35,39 ,40};
```

ouput:

11 Depth: 0
5 Depth: 1
4 Depth: 2
7 Depth: 2
9 Depth: 3
19 Depth: 1
20 Depth: 2
30 Depth: 3
35 Depth: 4
39 Depth: 5
40 Depth: 6

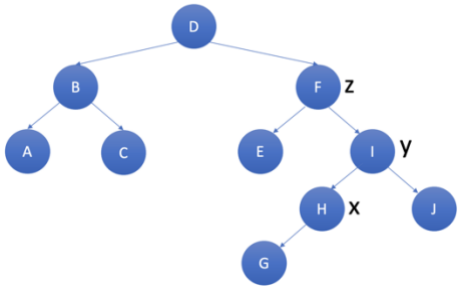
2 - Write a recursive test function that can verify whether an AVL tree is balanced or not. Assume the node class name is “Node,” and every node has two member variables “m_left” and “m_right” which point to the left and right children of the node. The height property is named m_height. The signature of the function is `bool testBalanceOfTree(Node* aNode)`. You can use the example AVL class (LinkedTree) that is posted in Blackboard to run your function. Only submit the test function. (25 points)

Hint: we need to visit all nodes and check the height property. Your test function returns true if the balance property is true at every node, otherwise the test function returns false.

```
bool TreeLinked::testBalanceOfTree(Node* aNode){
    //take height of both children, check if balanced
    //if not balanced return false
    //m_height
    int balance = -1;
    if (aNode != nullptr){
        int leftHeight = -1;
        int rightHeight = -1;
        if (aNode->m_left != nullptr)
            leftHeight= aNode->m_left->m_height ;
        if (aNode->m_right != nullptr)
            rightHeight = aNode->m_right->m_height ;
        balance = (leftHeight - rightHeight);
        if(balance < -1 || balance > 1){
            return false;
        }
        testBalanceOfTree(aNode->m_left);//first visit the left child
        testBalanceOfTree(aNode->m_right);//second visit the right child
    }

}
```

3 - Rebalance the following AVL tree after inserting G. You need to show the middle step if it happens. Briefly explain the operations. (25 points)




```
Node* y = aNode;
Node* x = y->m_left;
y->m_left = x->m_right;
x->m_right = y;
```

Case	Balancing Operations
Node's right child has greater height and left child of right child has a greater height	Right rotation about node's right child, left rotation about node

Node Z right child has greater height (Y) and Ys left child has greater height so a right rotation about nodes right child and left rotation about node must occur.

Right rotation around Y

Left rotation around Z

X will rotate to become Zs right child so the sub tree looks like

Then left rotate Z

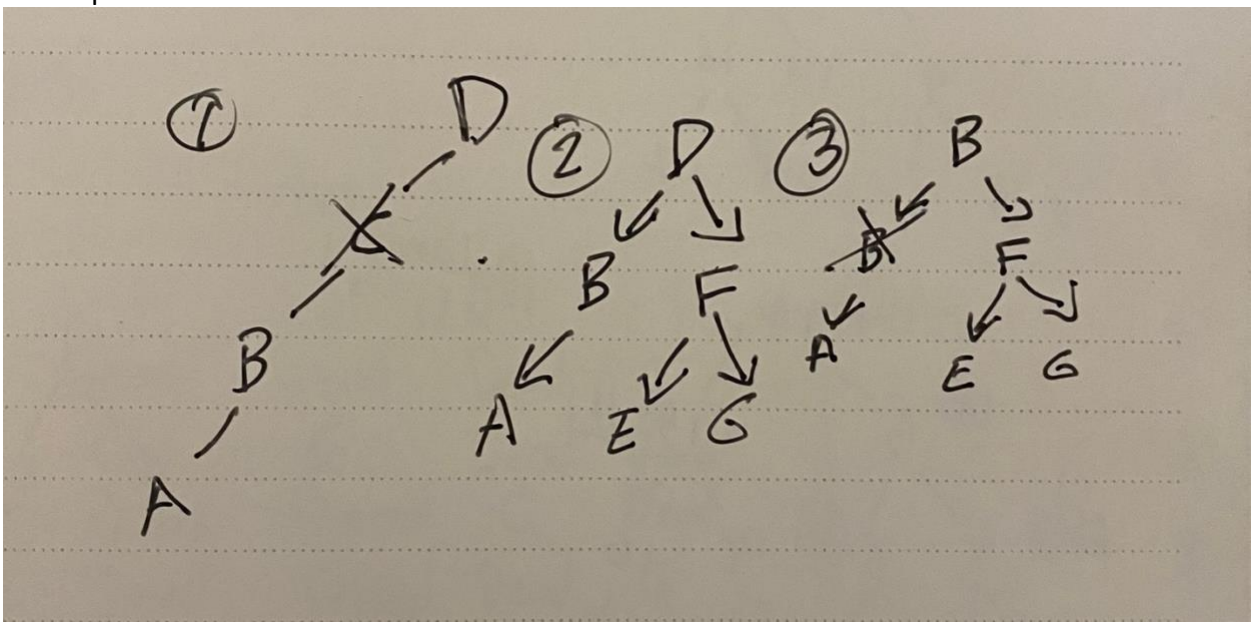
The operation it self is basically changing the pointers of the head. So repoint Y left child to x right child, then make X right child Y. Then update the right child of Z. Then preform a left rotation on Z.

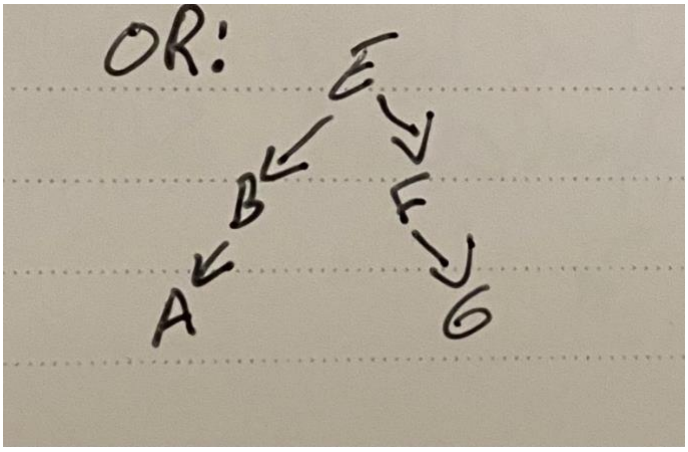
4 - We are constructing a BST from scratch. Draw the resulting tree after every remove operation. (25 points)
Note: if a removal operation can be performed in two different ways resulting in two different trees after removal, please draw both possible solutions.

Reminder: when removing a node with two children, we can replace it with either the min node in its right subtree or max node in its left subtree.

insert(D), insert(C), insert(B), insert(A) remove(C)
insert(F), insert(E), insert(G) remove(D)

- Remove C, make B the left child of D. Since C has no right child, the removal process is simple.
- Remove G, Node to delete has two children. Find largest node in left subtree. Copy largest value of left subtree into node to be deleted. Remove node in left subtree which was copied.





Or, the copy value of E into D then delete D which can keep the binary tree in the correct order.