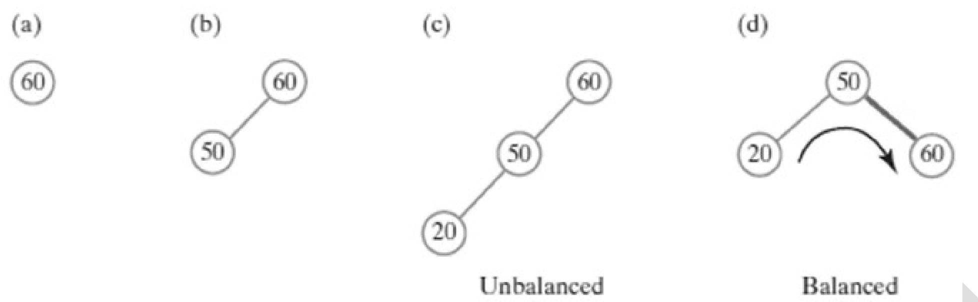
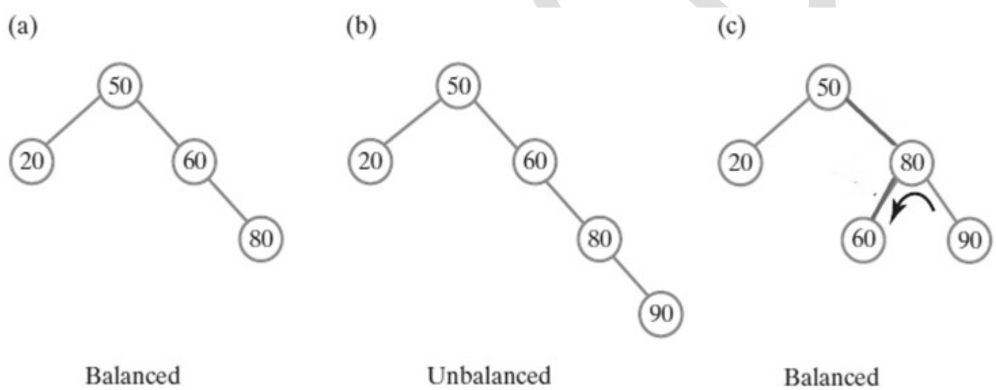
AVL Trees

- An **AVL tree (**Georgy **A**delson-**V**elsky and Evgenii **L**andis' tree**)** is a **BST** with the additional **balance** property that, for any node in the tree, the height of the **left** and **right** subtrees can differ by at most **1**.
- **Complete** binary trees are **balanced.**

Single Rotation

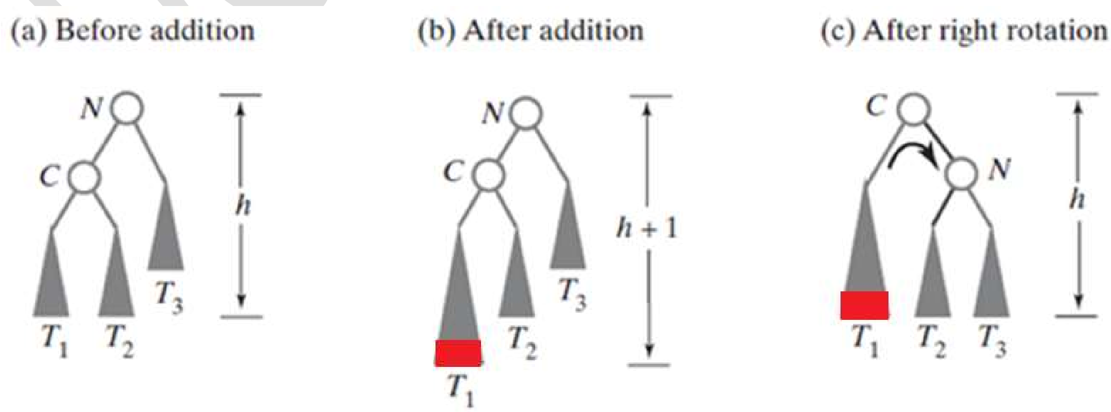


Example: After inserting (a) 60; (b) 50; and (c) 20 into an initially empty **BST**, the tree is **not balanced**;
(d) a corresponding **AVL** tree rotates its nodes to restore balance



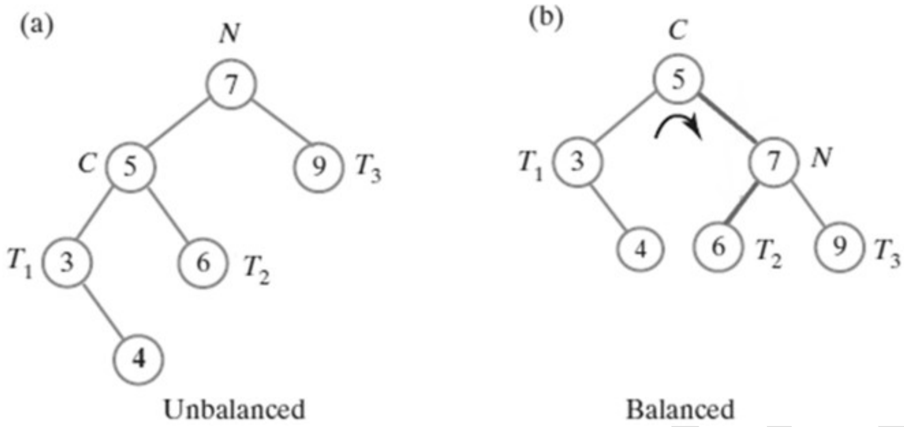
Example: (a) Adding 80 to the tree does not change the balance of the tree;
(b) a subsequent addition of 90 makes the tree **unbalanced** ;
(c) a left rotation restores its balance

Case 1: Single Right Rotation (left-left addition)



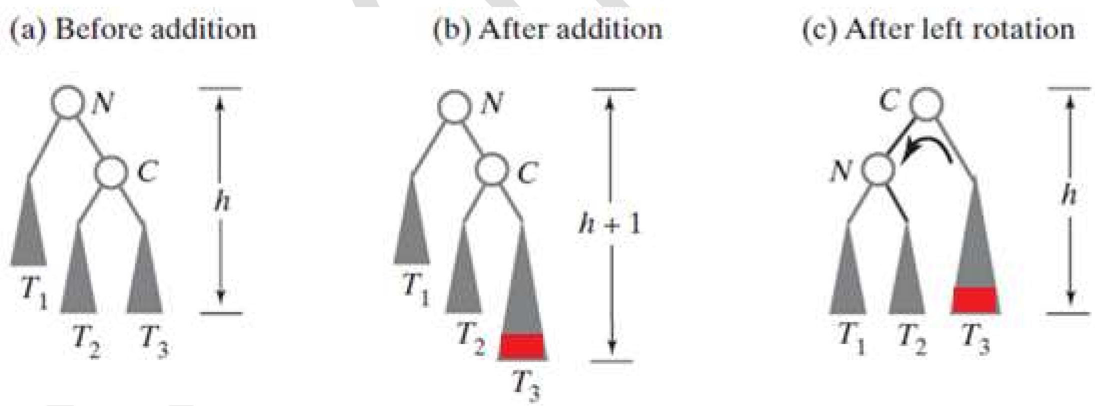
Before and after an addition to an **AVL** subtree that requires a **right rotation** to maintain its balance.

Example: a) before and b) after a **right rotation** restores balance to an **AVL** tree




```
Algorithm rotateRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to nodeC's right child
Set nodeC's right child to nodeN
return nodeC
```

Case 2: Single Left Rotation (right-right addition)



Before and after an addition to an **AVL** subtree that requires a **left rotation** to maintain its balance

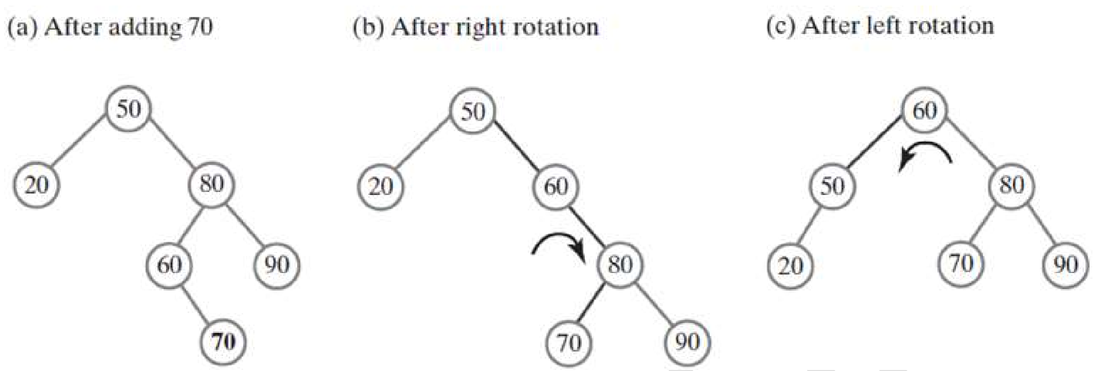
```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to nodeC's left child
Set nodeC's left child to nodeN
return nodeC
```

Double Rotations

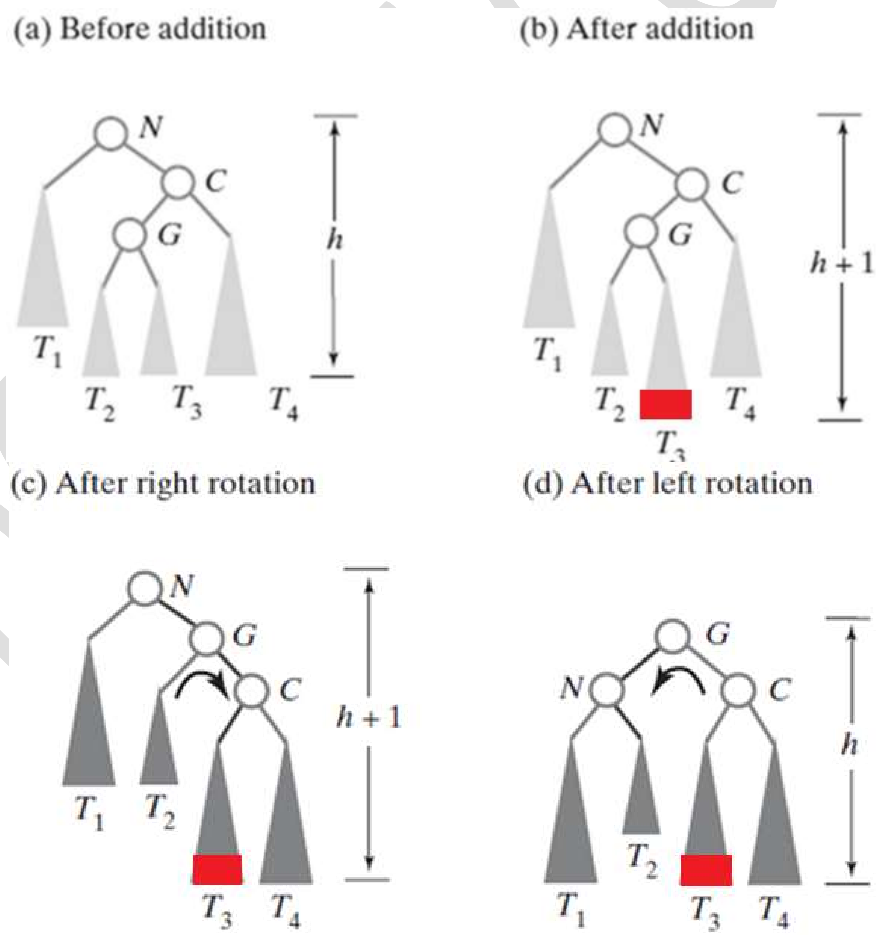
A **double rotation** is accomplished by performing two single rotations:

1. A rotation about node ***N*'s grandchild *G*** (its child's child)
2. A rotation about node ***N*'s new child**

Case 3: Right-Left Double Rotations (right-left addition)



Example: (a) Adding 70 destroys tree's balance; to restore the balance, perform both (b) a **right rotation** and (c) a **left rotation**

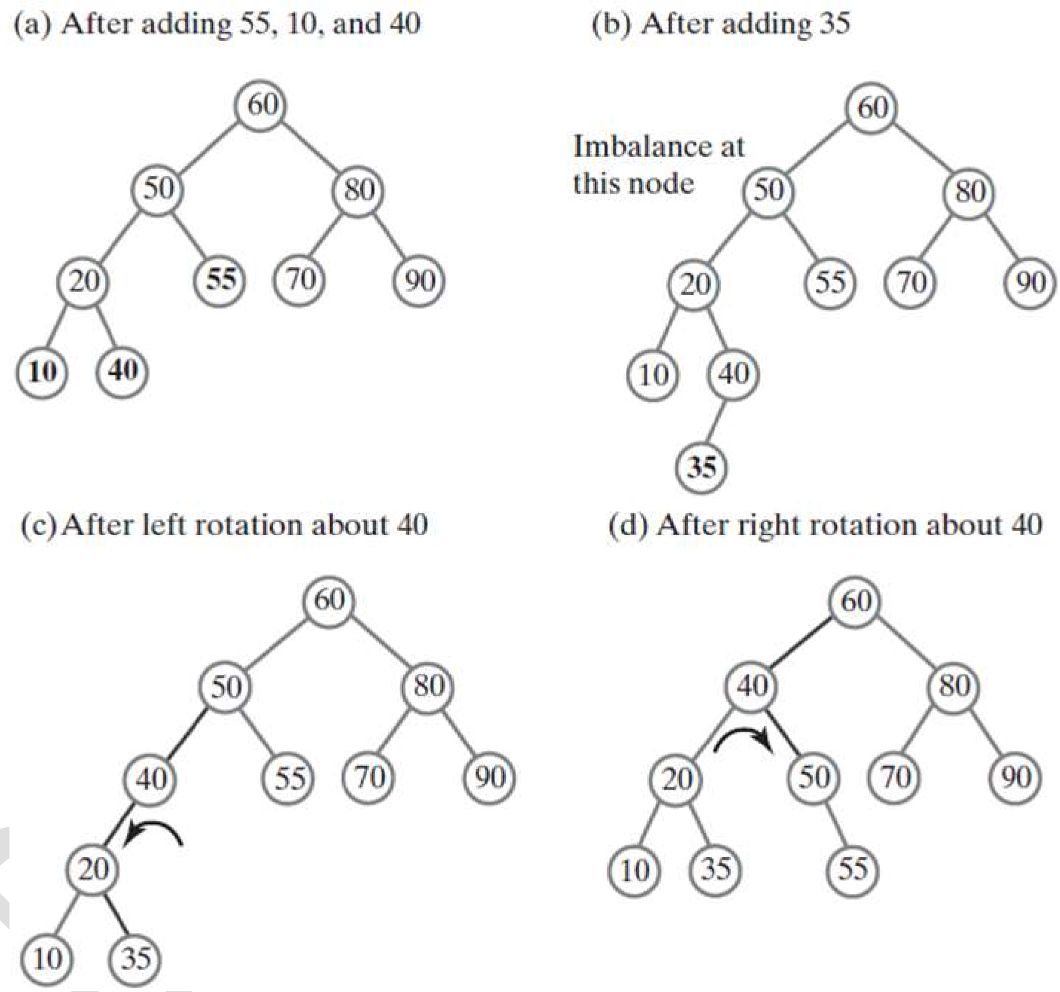


Before and after an addition to an **AVL** subtree that requires both a **right rotation** and a **left rotation** to maintain its balance

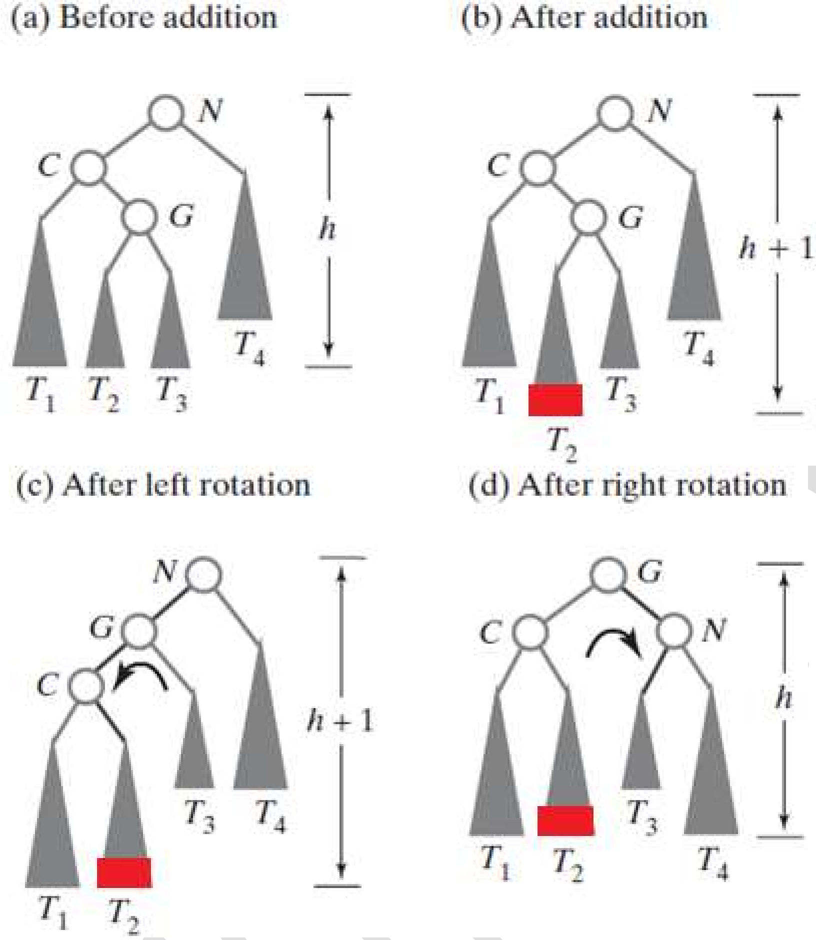
```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to the node returned by rotateRight(nodeC)
return rotateLeft(nodeN)
```

Case 4: Left-Right Double Rotations (left-right addition)

Example:



(a) The **AVL** tree after additions that maintain its balance;
(b) after an addition that destroys the balance;
(c) after a **left rotation**;
(d) after a **right rotation**



Before and after an **addition** to an **AVL** subtree that requires both a **left rotation** and a **right rotation** to maintain its balance

```
Algorithm rotateLeftRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to the node returned by rotateLeft(nodeC)
return rotateRight(nodeN)
```

- Four rotations cover the only four possibilities for the cause of the imbalance at node ***N***
- The addition occurred at:
  - The left subtree of **N**'s left child (case 1: right rotation)
  - The right subtree of **N**'s left child (case 4: left-right rotation)
  - The left subtree of **N**'s right child (case 3: right-left rotation)
  - The right subtree of **N**'s right child (case 2: left rotation)

Rebalance Code Implementation

- Pseudo-code to rebalance the tree:

```
Algorithm rebalance(nodeN)
if (nodeN's left subtree is taller than its right subtree by more than 1)
{   // Addition was in nodeN's left subtree
    if (the left child of nodeN has a left subtree that is taller than its right subtree)
        rotateRight(nodeN)       // Addition was in left subtree of left child
    else
        rotateLeftRight(nodeN)   // Addition was in right subtree of left child
}
else if (nodeN's right subtree is taller than its left subtree by more than 1)
{   // Addition was in nodeN's right subtree
    if (the right child of nodeN has a right subtree that is taller than its left subtree)
        rotateLeft(nodeN)        // Addition was in right subtree of right child
    else
        rotateRightLeft(nodeN)   // Addition was in left subtree of right child
}
```

```
private TNode rebalance(TNode nodeN){
  int diff = getHeightDifference(nodeN);
  if ( diff > 1) { // addition was in node's left subtree
    if(getHeightDifference(nodeN.left)>0)
      nodeN = rotateRight(nodeN);
    else
      nodeN = rotateLeftRight(nodeN);
  }
  else if ( diff < -1){  // addition was in node's right subtree
    if(getHeightDifference(nodeN.right)<0)
      nodeN = rotateLeft(nodeN);
    else
      nodeN = rotateRightLeft(nodeN);
  }
  return nodeN;
}
```

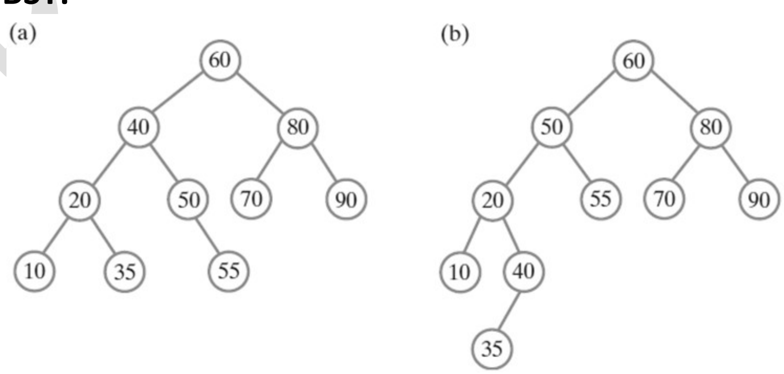
Insert Code Implementation:

```
public void insert(T data) {
   if(isEmpty())     root = new TNode<>(data);
   else {
      TNode rootNode = root;
      addEntry(data, rootNode);
      root = rebalance(rootNode);
   }
}

public void addEntry(T data, TNode rootNode){
   assert rootNode != null;
   if(data.compareTo((T)rootNode.data) < 0){ // right into left subtree
      if(rootNode.hasLeft()){
         TNode leftChild = rootNode.left;
         addEntry(data, leftChild);
         rootNode.left=rebalance(leftChild);
      }
      else         rootNode.left = new TNode(data);
   }
   else {  // right into right subtree
      if(rootNode.hasRight()){
         TNode rightChild = rootNode.right;
         addEntry(data, rightChild);
         rootNode.right=rebalance(rightChild);
      }
      else         rootNode.right = new TNode(data);
   }
}
```

Delete Code Implementation:

```
public TNode delete(T data) {
   TNode temp = super.delete(data);
   if(temp!= null){
      TNode rootNode = root;
      root = rebalance(rootNode);
   }
   return temp;
}
```

An **AVL** Tree versus a **BST:**



Example: The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty (a) **AVL** tree; (b) **BST**