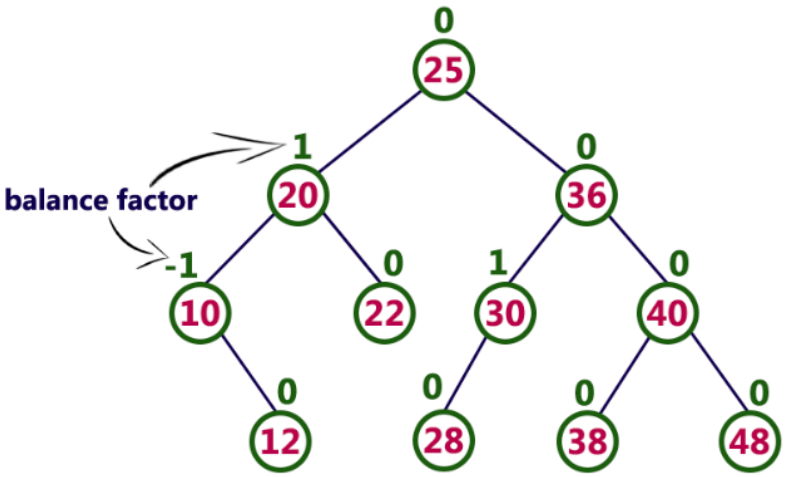
CSCI 3110 AVL (Adel'son-Vel'skii and Landis) tree

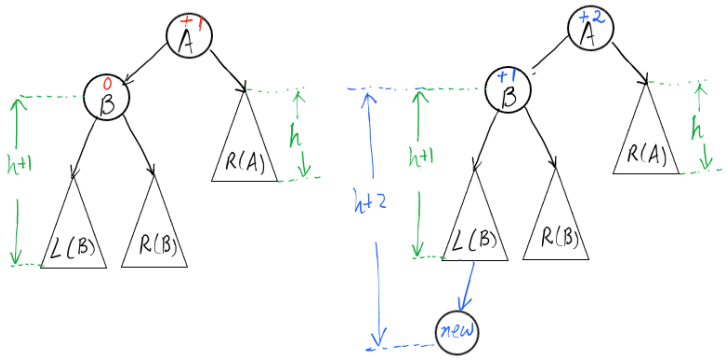
Main advantage of BST over linked list in organizing data is efficiency (search efficiently: visit as few nodes as possible before reaching the target record)

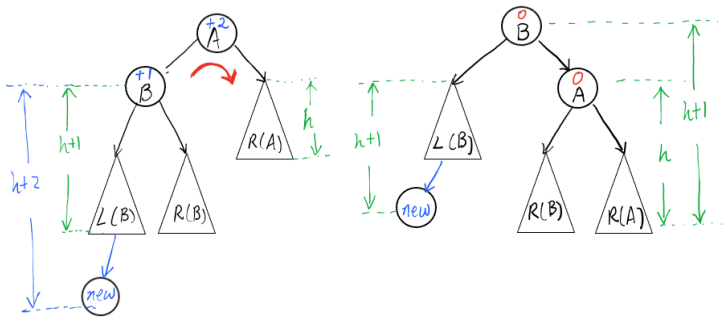
- If the tree is skewed, it does not improve efficiency, e.g., it is equivalent to linked list
- If the BST is complete/perfect BST, it has the smallest height, most efficient, require the minimum number of visits for any target record
- It is quite expensive to build/maintain the complete/perfect BST, but it is quite inexpensive (computational-wise) to build/maintain a BST that has a height that is close to minimum height. This type of BST is called the AVL tree.
- AVL tree is a balanced BST, (for any node in the tree, the height of its left subtree and right subtree differs no more than 1).
  AVL trees can be searched almost as efficiently as a minimum height BST



Use rotation operations to maintain an AVL tree after insertions

- It is possible to re-arrange any BST of N nodes to obtain a BST with minimum height: require the saving (in-order traversal) and restoring the tree.
- To *maintain* an AVL tree is much less expensive: constantly monitor the shape of the tree, after each insertion/deletion operation, use ***rotation*** operations
- Procedure to apply rotation operation:
  Examine the tree to find the **first** imbalanced node *along the path from the newly inserted node to the root of the tree.*
  - **Case 1: single right rotation**: the inserted item is in the left subtree of the left child of the nearest ancestor with balance factor +2
    Resetting pointers for this operation: assuming A is the nearest ancestor of the inserted item that has balance factor +2, and let B be its left child:
    - **Set the pointer from the parent of A to point to B**
    - **Set the left child pointer of A to the right child of B**
    - **Set the right child pointer of B to point to A**

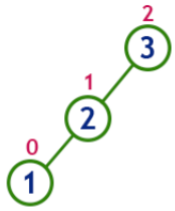




Examples:

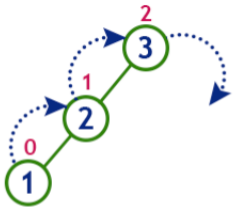
Example 1

insert 3, 2 and 1

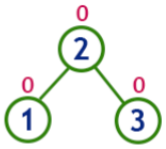


Tree is imbalanced

because node 3 has balance factor 2

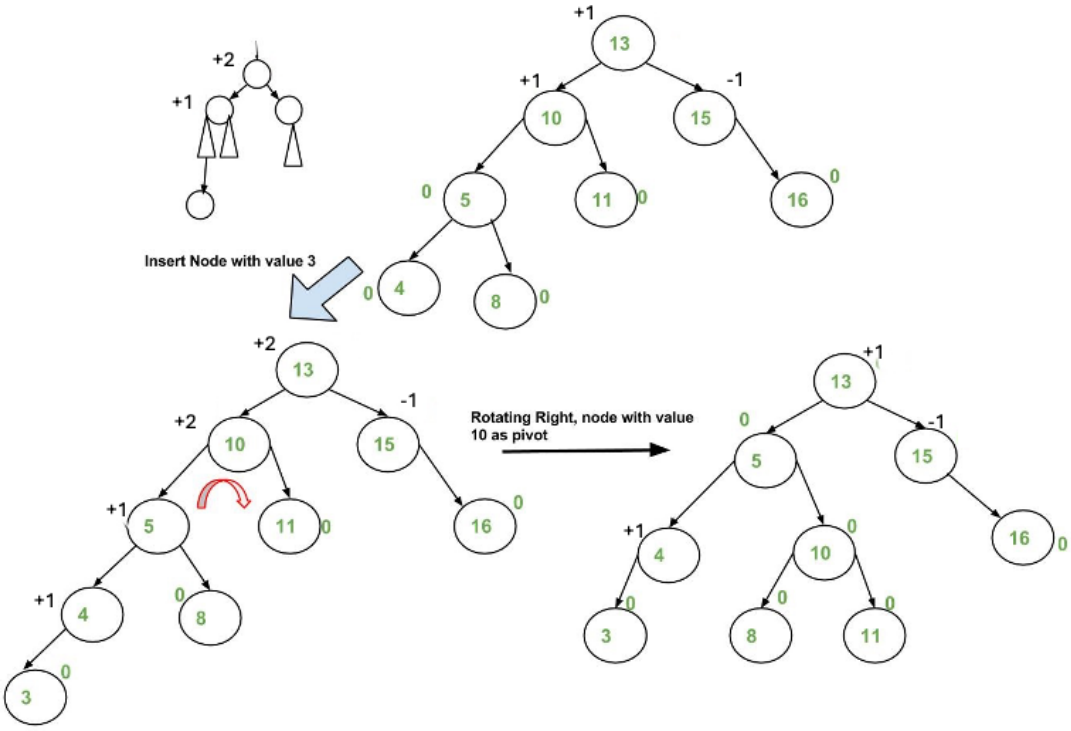


To make balanced we use RR Rotation which moves nodes one position to right

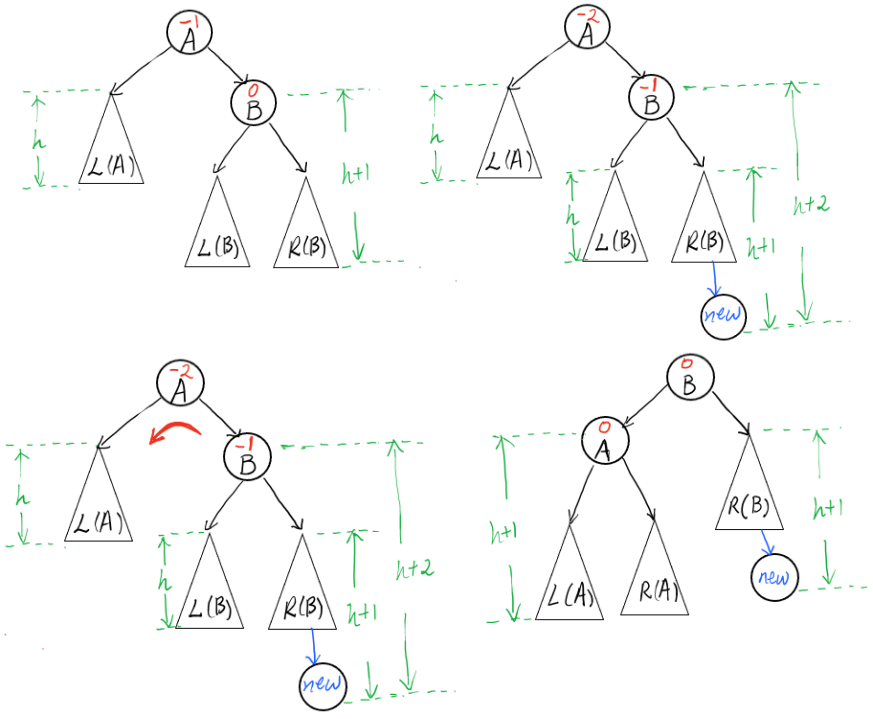


After RR Rotation Tree is Balanced

Example 2

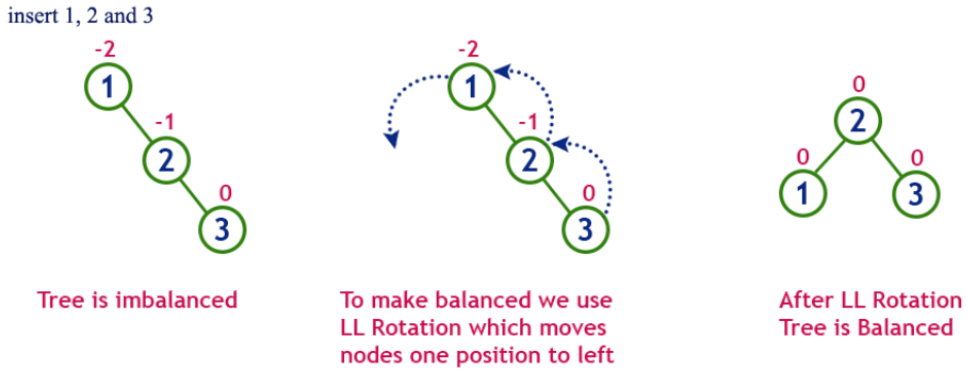


- **Case 2: single left rotation**: the inserted item is in the right subtree of the right child of the nearest ancestor with balance factor -2
  - **Set the pointer from the parent of A to point to B**
  - **Set the right child pointer of A to point to the left child of B**
  - **Set the left child pointer of B to point to A**

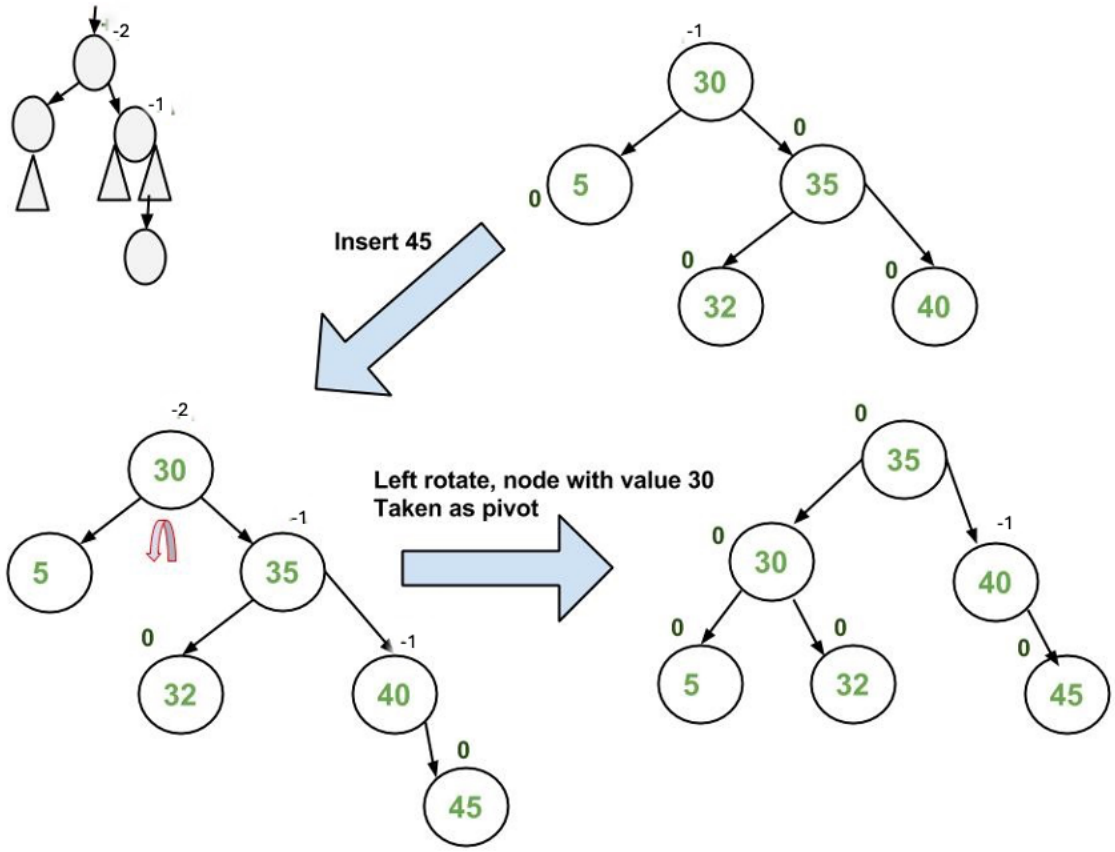


Examples:

Example 1:



Example 2:



Practice example on maintaining AVL trees using single rotations

1. insert 30, 20 → balanced
2. insert 10 → imbalanced → in-balanced node is 30, right rotate at 30 → balanced
3. insert 40 → balanced
4. insert 50 → imbalanced → in-balanced node is 30, left rotate at 30 →balanced
5. insert 60 → imbalanced at node 20, left rotate at 20
6. insert 8 → balanced
7. Practice Exercise
    - Insert 65, what to do?
    - Or Insert 5, what to do?

- **Case 3: double(left-right) rotation**: the inserted item is in the right subtree of the left child of the nearest ancestor with balance factor +2

  Resetting pointers for this operation: assuming A is the nearest ancestor of the inserted item that has balance factor +2, and let B be its left child

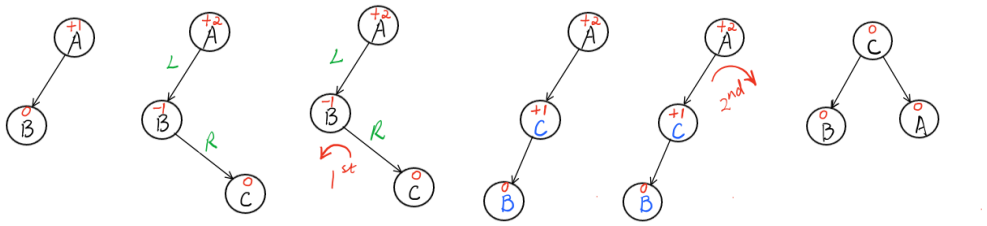
  Left rotation:

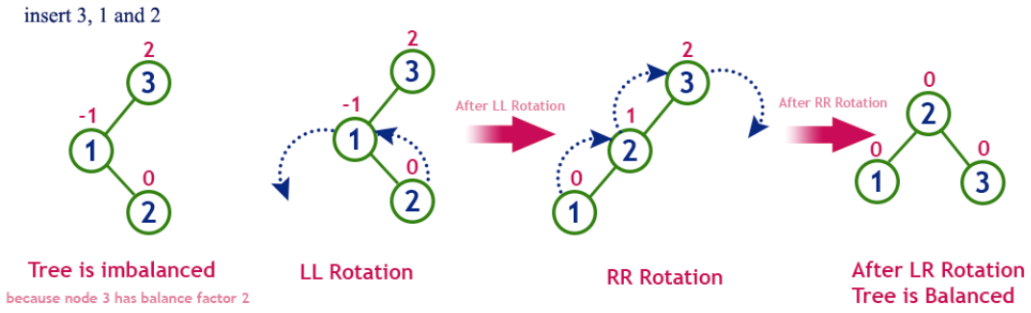
  - Let the left child pointer of node A point to the node C, which is root of the right subtree of node B
  - Let the right child pointer of node B point to the left child of node C
  - Let the left child pointer of C point to node B

  Right rotation:

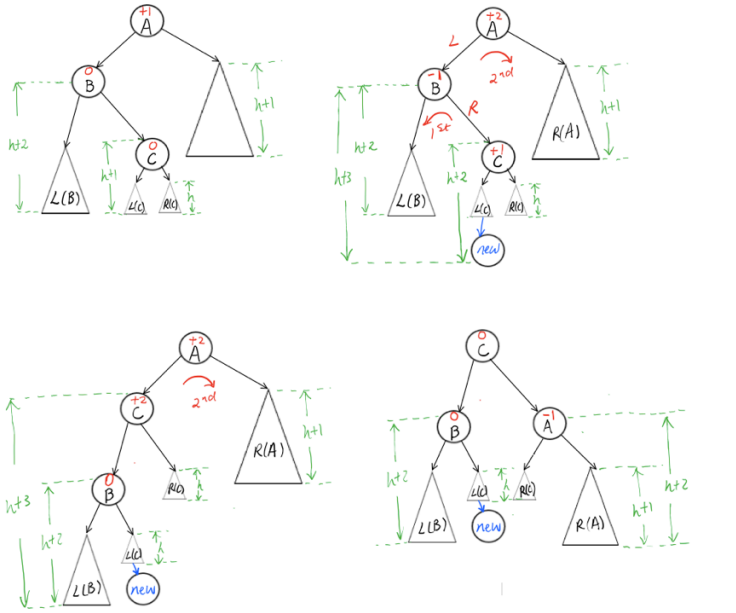
  - Let the pointer from the parent of node A point to node C
  - Let the left child pointer of A point to the right child of node C
  - Let the right child of node C point to node A

1st situation: node B has no right child before the new node is inserted, the new node becomes the right child C of node B

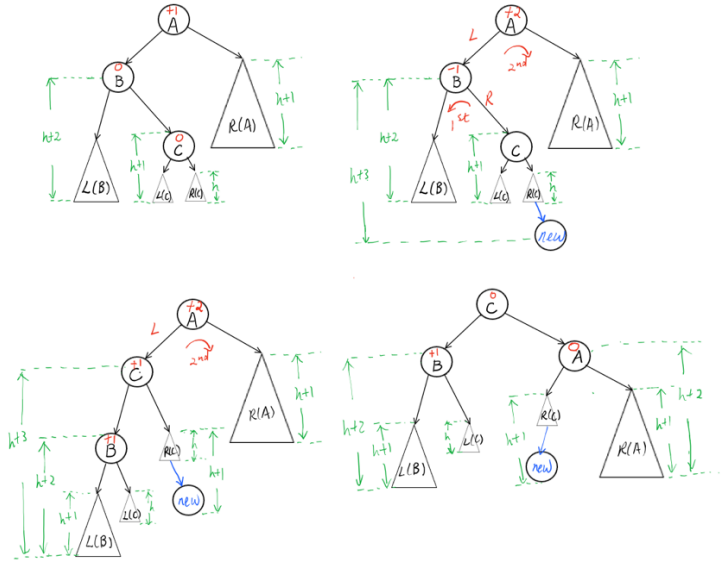




2nd situation: node B has a right child C, and the new node is inserted in the left subtree of C

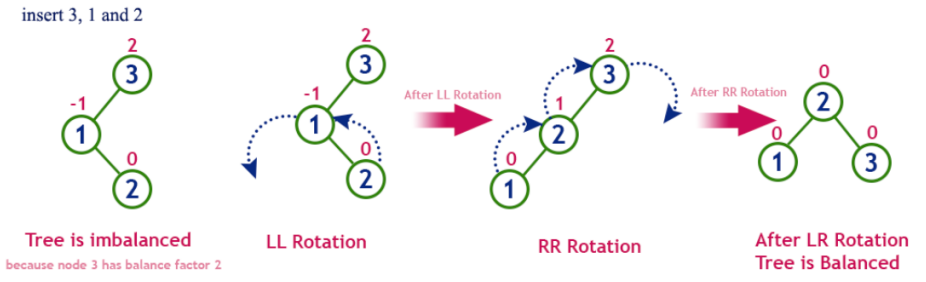


3rd situation: B has a right child C, and the new node is inserted in the right subtree of C

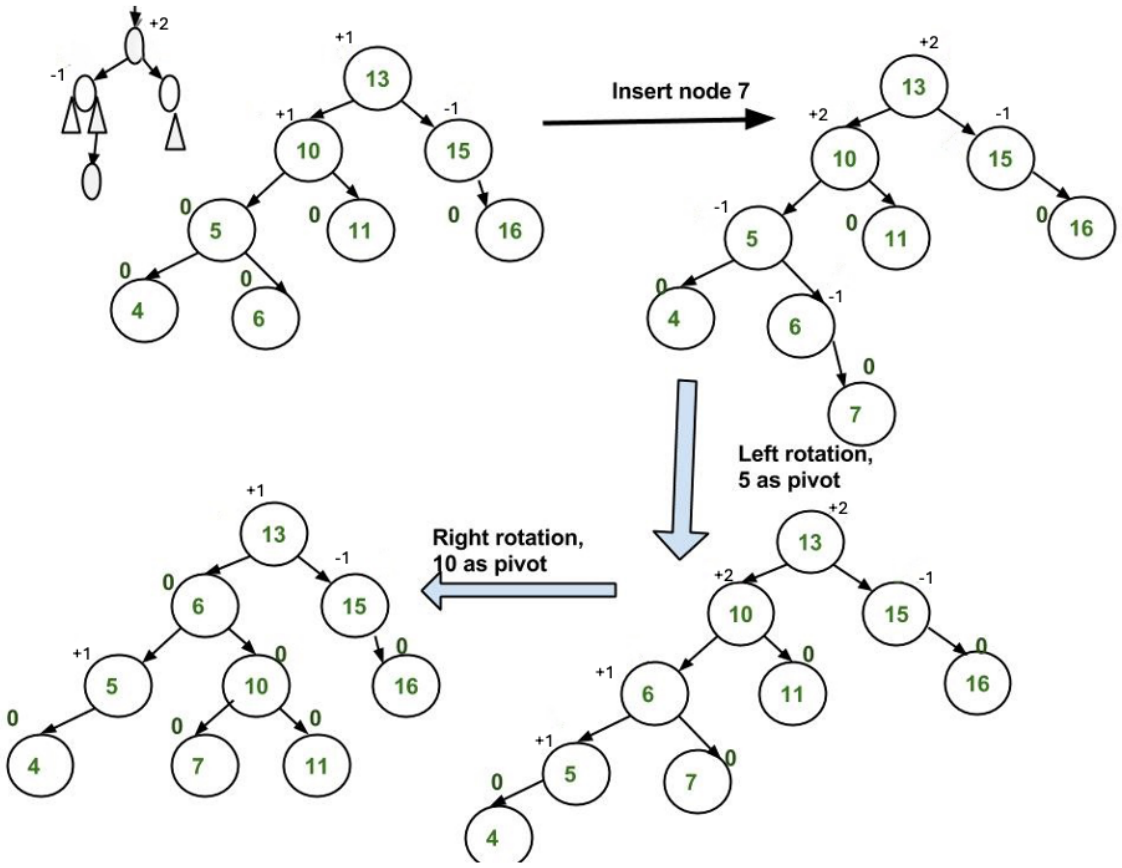


Examples:

Example1:

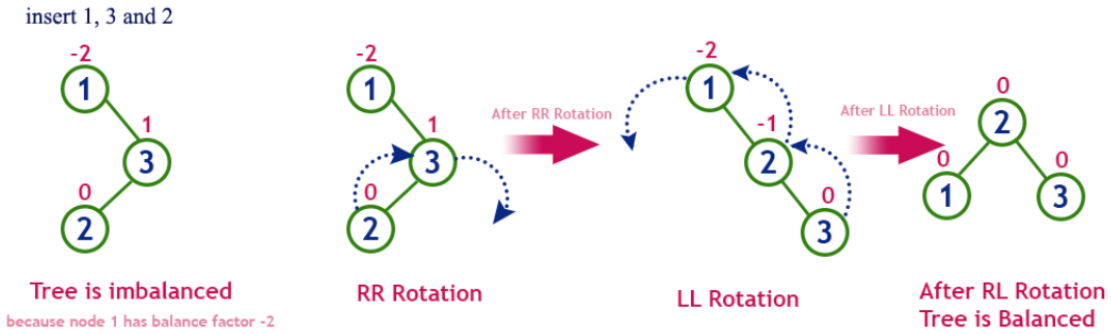


Example 2

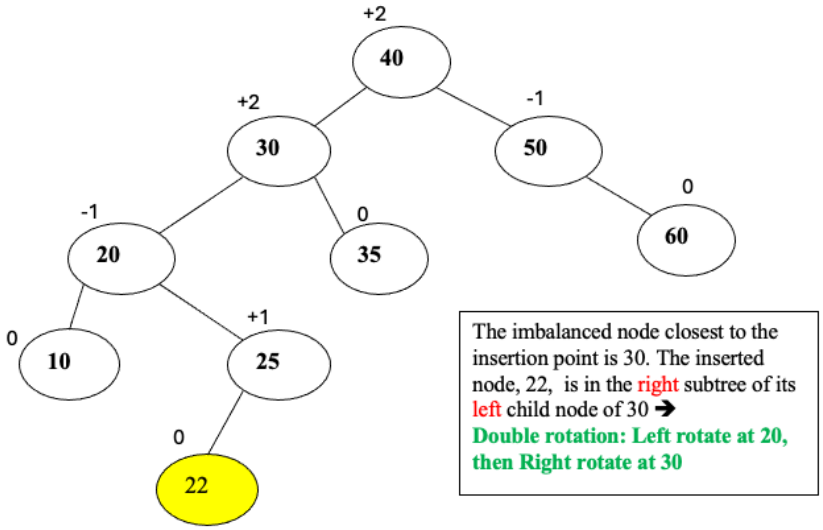


- **Case 4: double(right-left) rotation**: the inserted item is in the left subtree of the right child of the nearest ancestor with balance factor -2

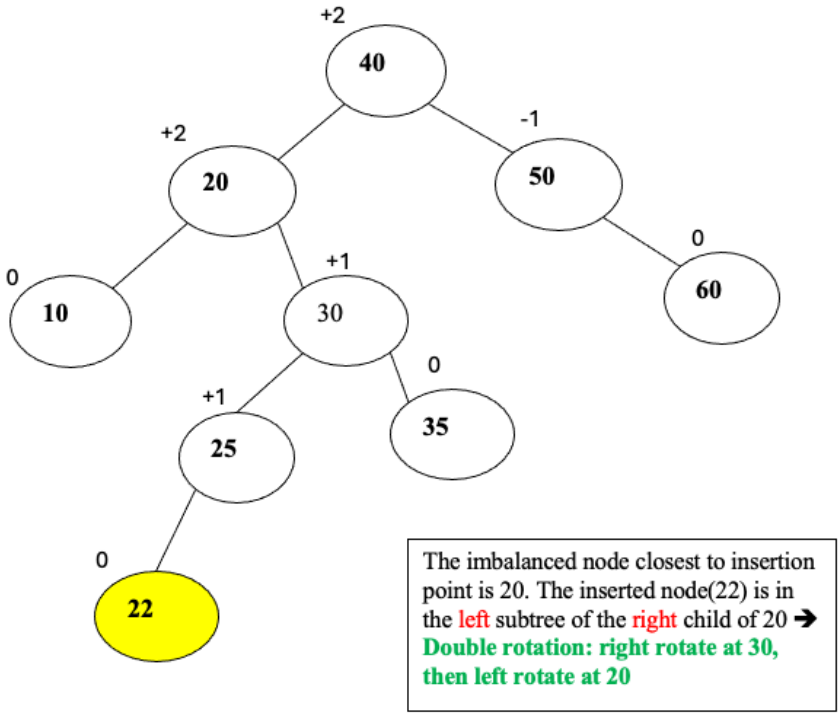
  Similar to Case 3, the insertion of the new node occurs in the left subtree of the right child of the imbalanced node



Examples of double rotations:
Example 1:



Example 2:

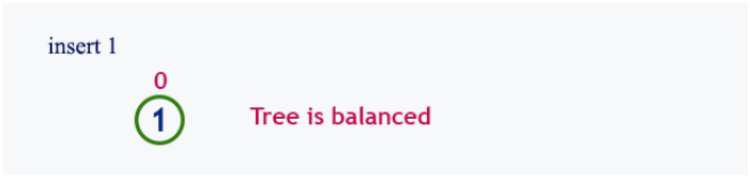


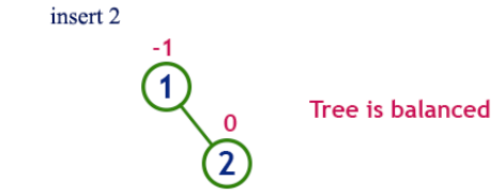
Practice Question:

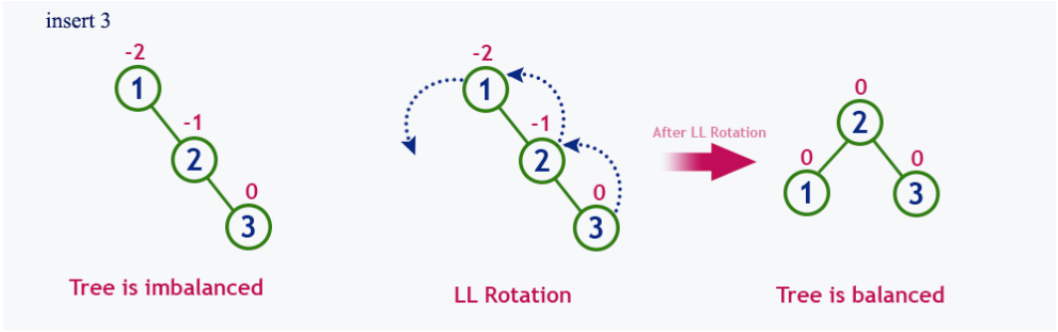
What will the AVL tree look like after nodes with these key values are inserted into an empty AVL tree?

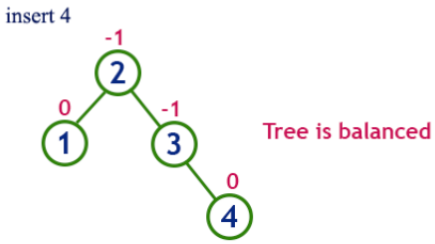
50, 20, 14, 60, 55, 52

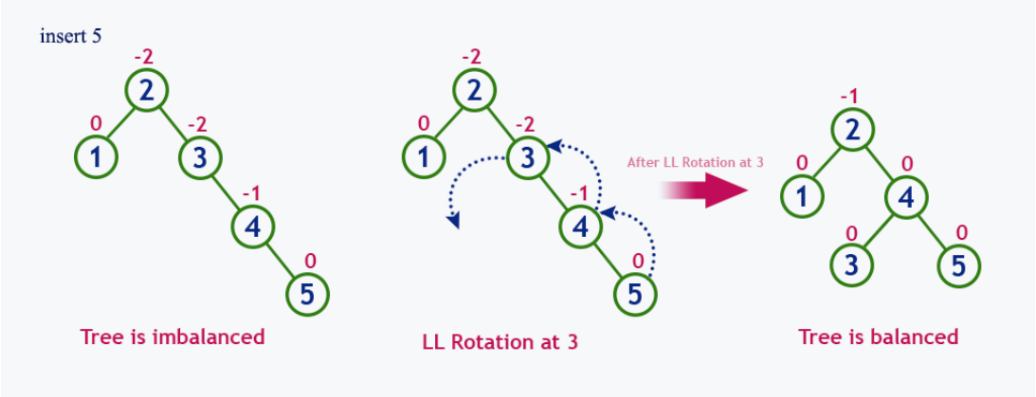
Another Full Example

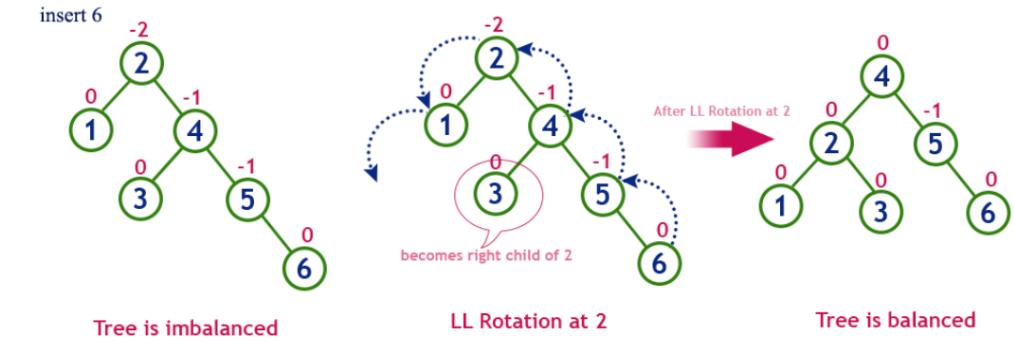


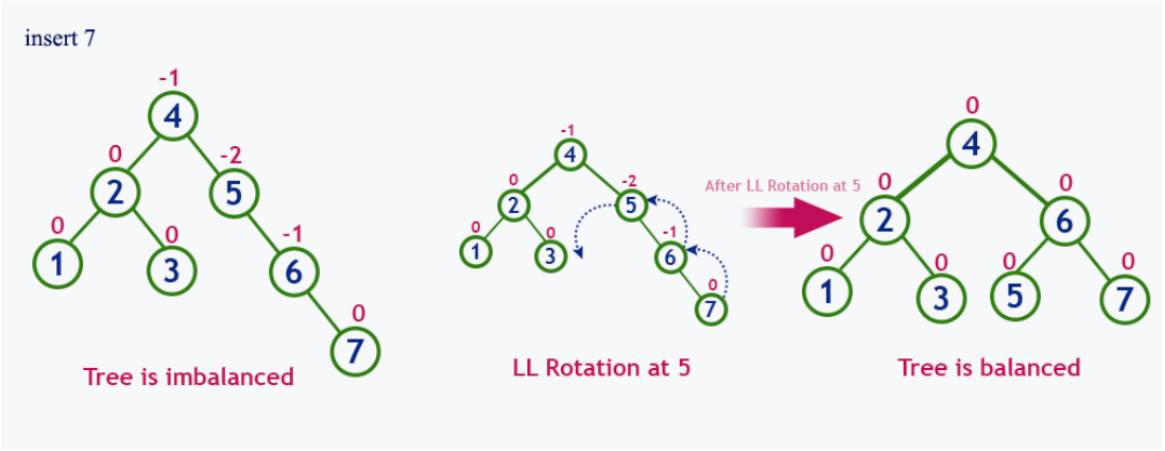


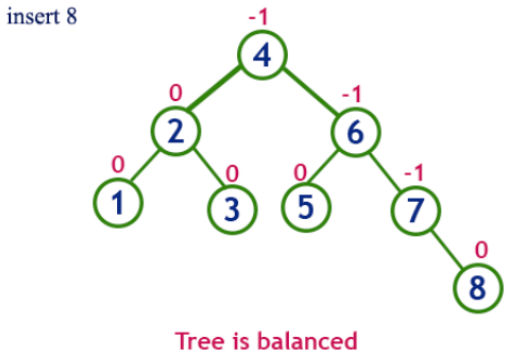





insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced


insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced


insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced


insert 8
Tree is balanced

How to decide whether a binary search tree is AVL tree? (post-order traversal)

```
bool   IsAVL(treePtr  nodePtr){
        bool  avl=true;
        if (nodePtr != NULL) {
            if (!IsAVL(nodePtr→LChildPtr) || !IsAVL(nodePtr→RChildPtr))
                avl = false;

            if (avl)  {
                leftHeight=FindHeight(nodePtr→LChildPtr);
                rightHeight=FindHeight(nodePtr→RChildPtr);
                if (abs(leftHeight – rightHeight) > 1)
                        avl = false;
             }
        }
        return avl;
}
```