DATA STRUCTURES

AVL Trees

Dr G.Kalyani

Department of Information Technology

VR Siddhartha Engineering College

Why AVL trees

- The time taken for all operations in a binary search tree depends on height of the tree. If height is h then time complexity is **O(h)**.
- However, it can be **O(n)** if the BST becomes skewed or degenerated (i.e. worst case).
- **AVL tree controls the height of the binary search tree** by not letting it to be skewed or degenerate.
- By **limiting the height to log n**, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Tree (Height Balanced BST)

- AVL Tree is invented by GM **A**delson–**V**elsky-EM **L**andis in 1962. The tree is named AVL in honor of its inventors.

- AVL Tree can be defined as **height balanced binary search tree in which each node is associated with a balance factor.**

- Tree is said to be **balanced if balance factor of each node is in between -1 to 1,** otherwise, the tree will be unbalanced and need to be balanced.

AVL Tree (Height Balanced BST)

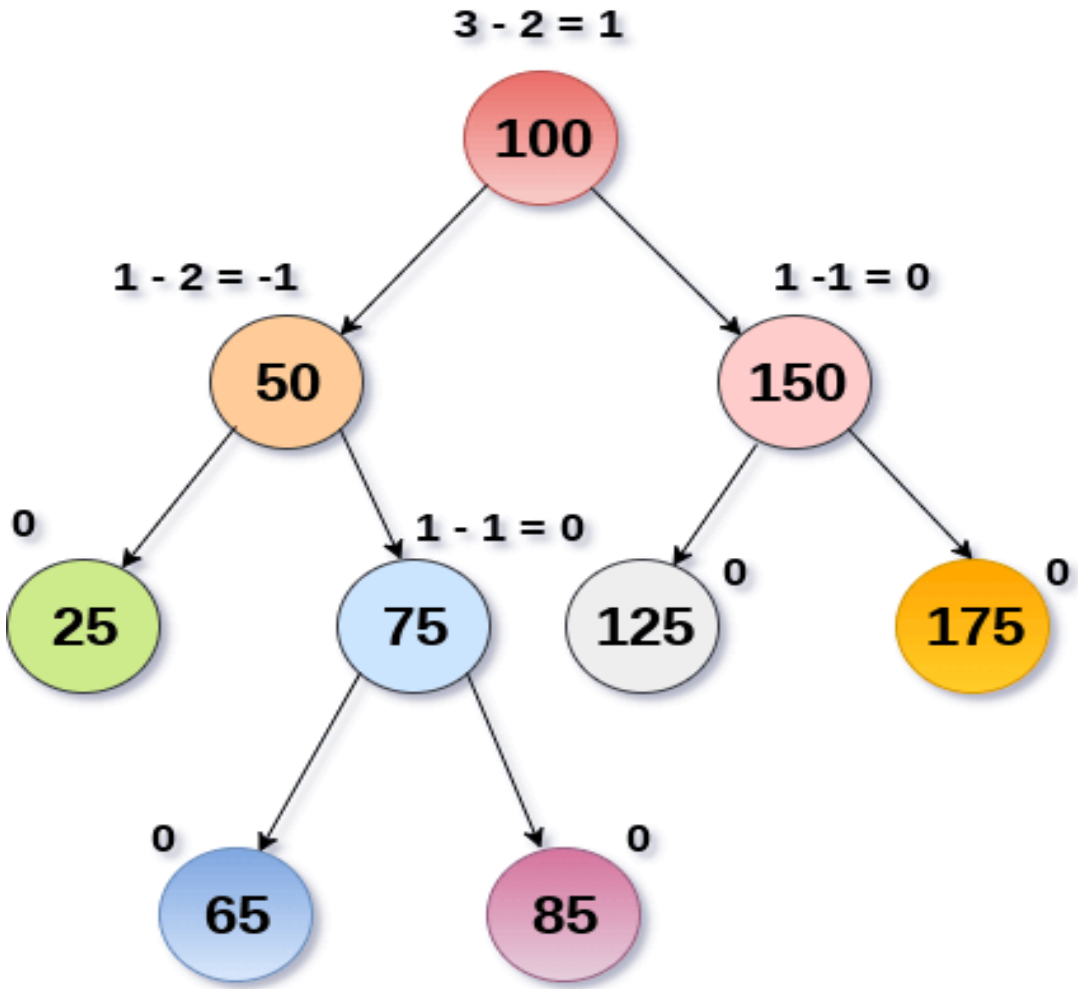
- **Balance Factor(k)= height (leftST(k)) - height (rightST(k))**

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

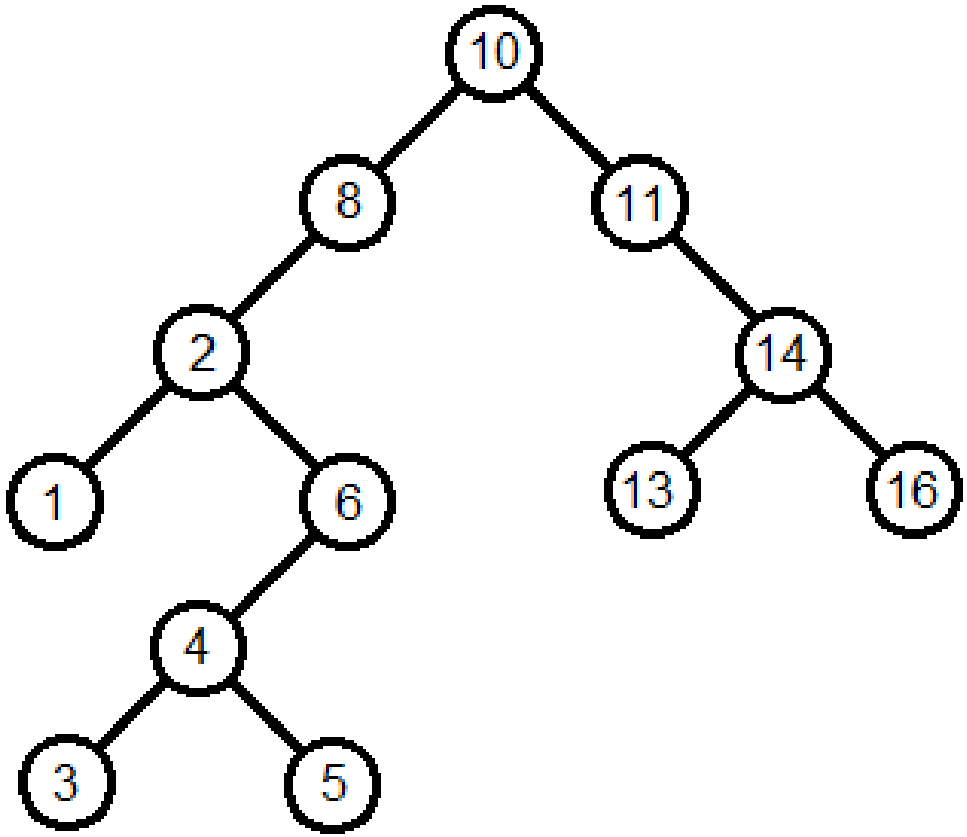
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

AVL Tree (Height Balanced BST)



AVL Tree

Calculate the Balance Factor of Each Node



Operations on AVL tree

- Due to the fact that, AVL tree is also a binary search tree therefore, **all the operations are performed in the same way as they are performed in a binary search tree.**

- **Searching and traversing do not lead to the violation in property of AVL tree.**

- However, **insertion and deletion are the operations which can violate this property** and therefore, they need to be revisited.

Operations on AVL Tree

Operation	Description
Searching	Same as BST
Traversing	Same as BST
Insertion	Adding a new element to the AVL tree at the appropriate location. so that the property of AVL tree may or may not violate. If violated then it is said to be unbalanced. So balance it
Deletion	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have. After deletion if the tree is unbalanced then balance it.

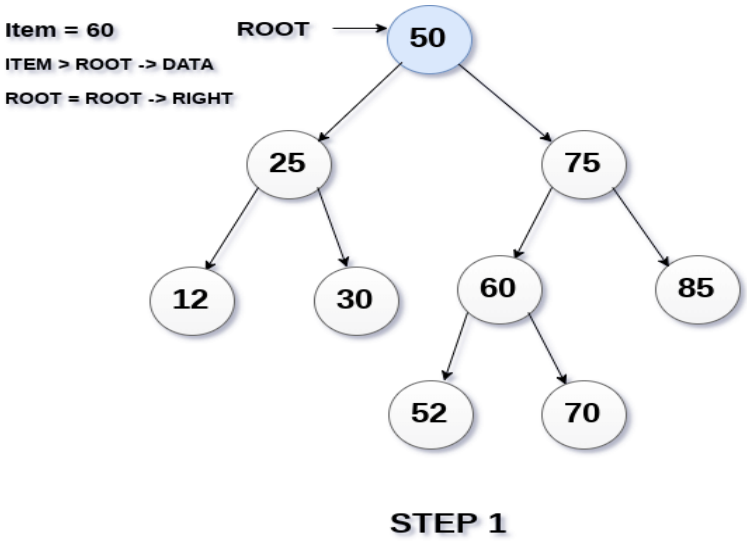
Searching in AVL TREE

- Searching means finding or locating some specific element or node within a data structure.

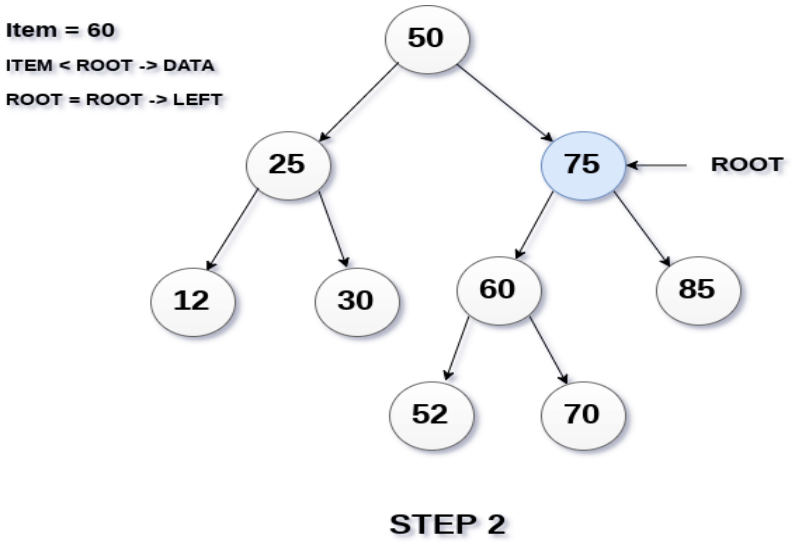
- However, searching for some specific node in AVL tree is pretty easy due to the fact that, element in AVL tree are stored in a particular order.

- Process:
  - Compare the element with the root of the tree.
  - If the item is matched then return the location of the node.
  - Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
  - If not, then move to the right sub-tree.
  - Repeat this procedure recursively until match found.
  - If element is not found then return NULL.

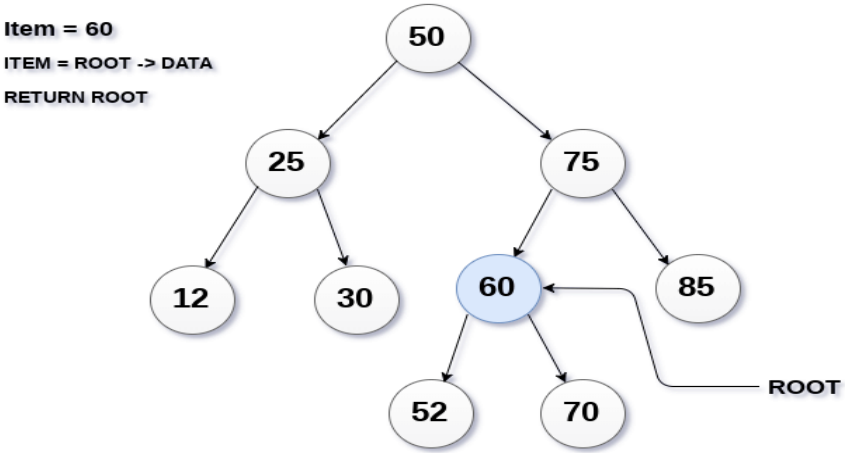
Example search in AVL Tree



STEP 1



STEP 2



STEP 3

Algorithm Searching in AVL Tree

```
Algorithm Search (ROOT, ITEM)
{
    IF ROOT = NULL
        Write not found; Return;
    ELSE IF ROOT -> DATA = ITEM
        Return ROOT
    ELSE   IF item < ROOT -> DATA
            search(ROOT -> LEFT, ITEM)
        ELSE
            search(ROOT -> RIGHT,ITEM)
}
```

Insertion in AVL Tree

- Insert function is used to **add a new element in a AVL tree at appropriate location which is same as insertion into binary search tree.**
- After insertion check the balance factors at every node of the tree.
- If all the balance factors are -1, 0 and 1 then tree is said to be balanced.
- Otherwise the insert operation makes the tree un balanced.
- Apply Rotation operations to make the tree balanced.

AVL Rotations

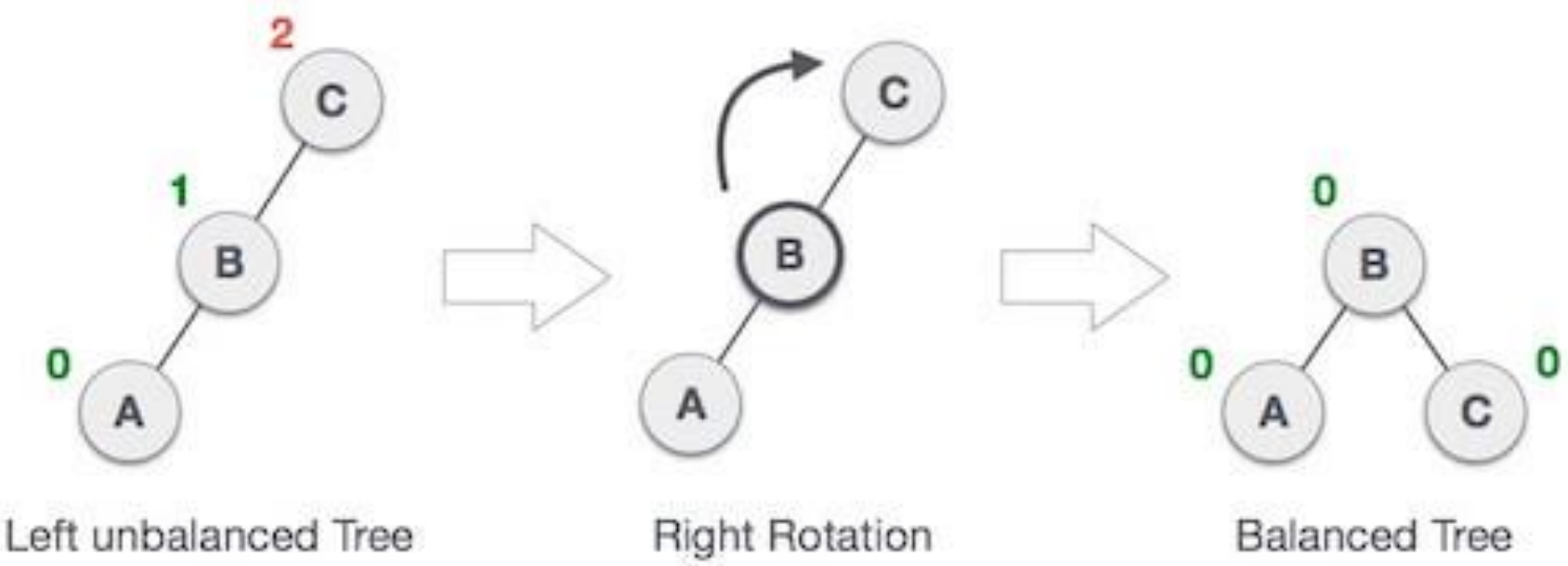
- We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

- Assume node A is the node whose balance Factor is other than -1, 0, 1.
- **L L rotation:** Inserted node is in the left subtree of left subtree of A
- **R R rotation :** Inserted node is in the right subtree of right subtree of A
- **L R rotation :** Inserted node is in the right subtree of left subtree of A
- **R L rotation :** Inserted node is in the left subtree of right subtree of A

- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations.

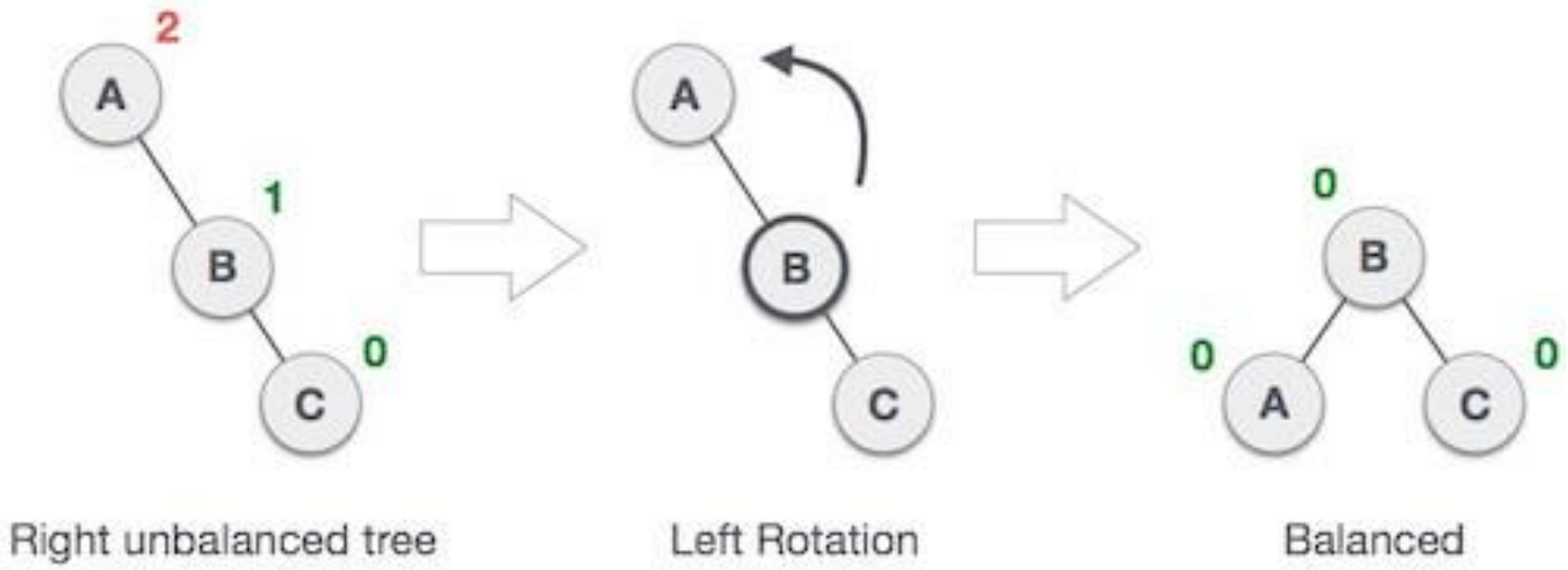
L L Rotation

- When BST becomes unbalanced, due to a **node is inserted into the left subtree of the left subtree of C,** then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

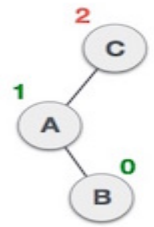
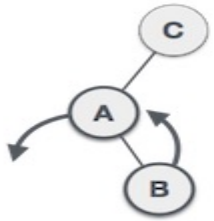
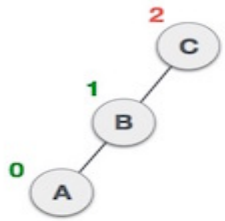
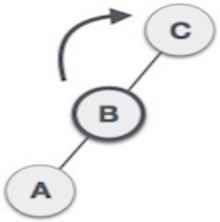
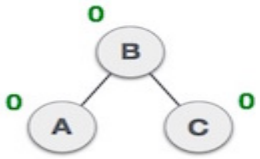


R R Rotation

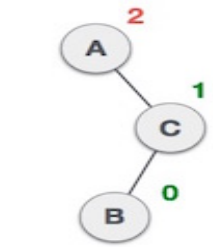
- When BST becomes unbalanced, due to a **node is inserted into the right subtree of the right subtree of A,** then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



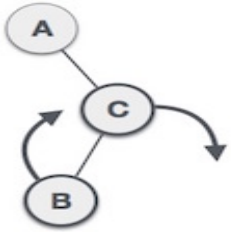
L R Rotation

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

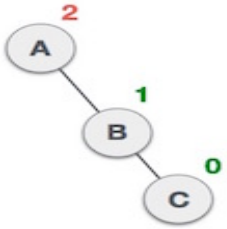
R L Rotation



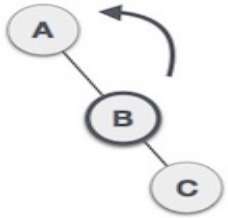
A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.



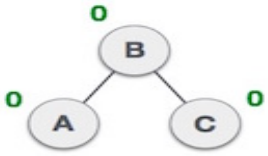
First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.

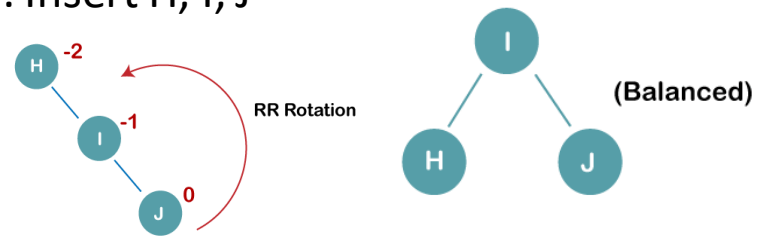


The tree is now balanced.

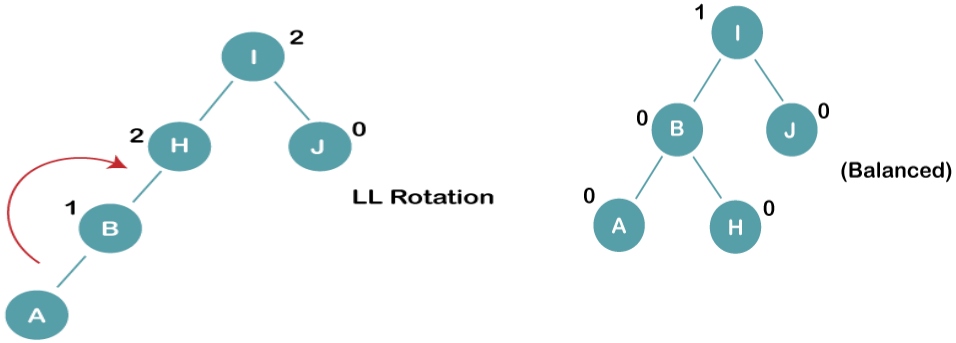
In Class Exercise1

- Q: Construct an AVL tree having the following elements
- **H, I, J, B, A, E, C, F, D, G, K, L**

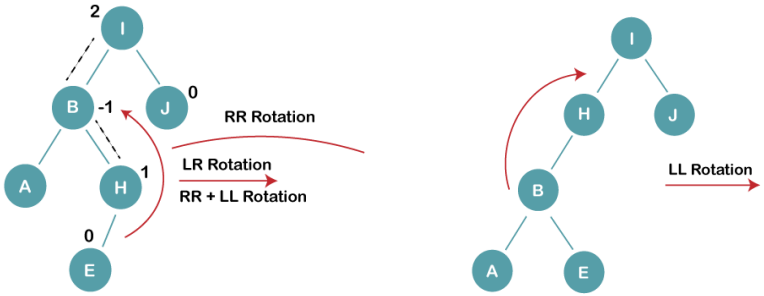
1. Insert H, I, J

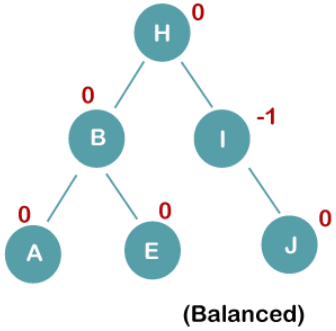


2. Insert B, A

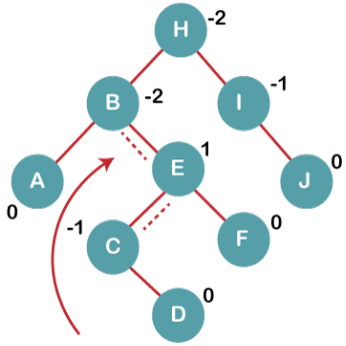


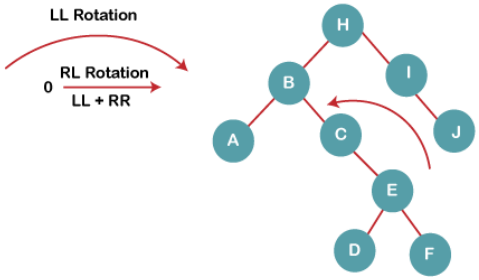
3. Insert E

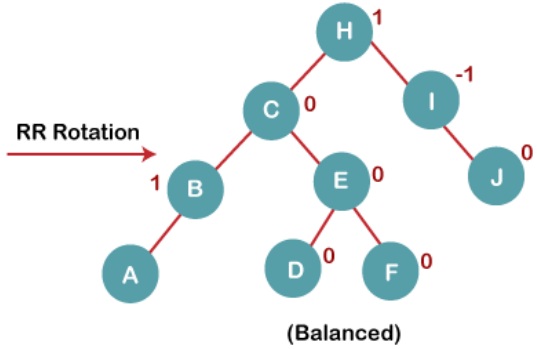




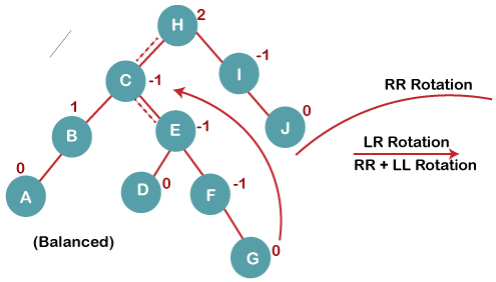
4. Insert C, F, D

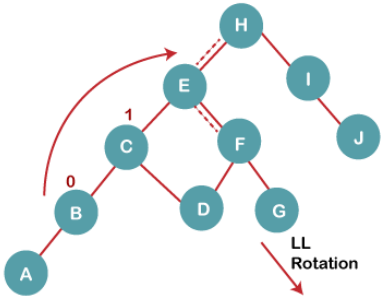


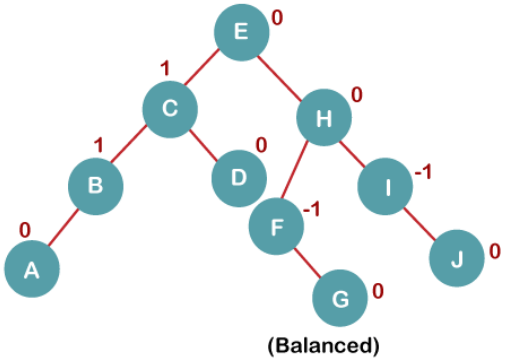




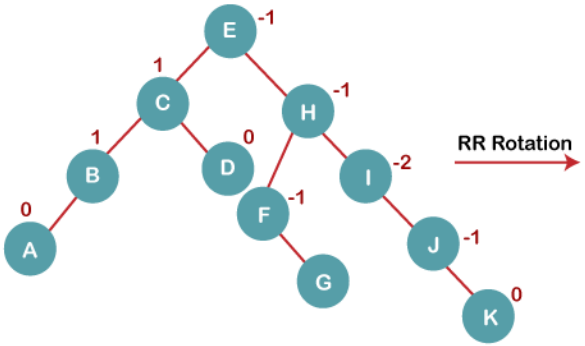
5. Insert G

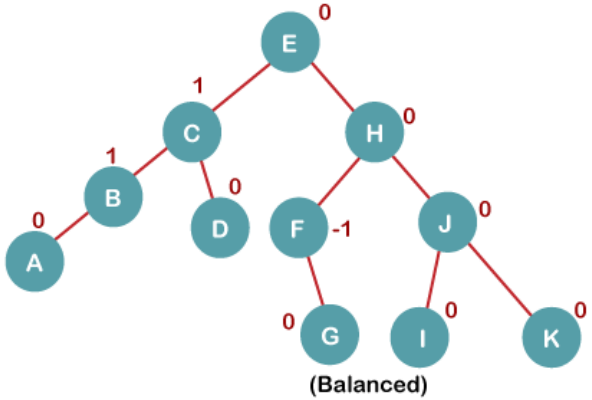




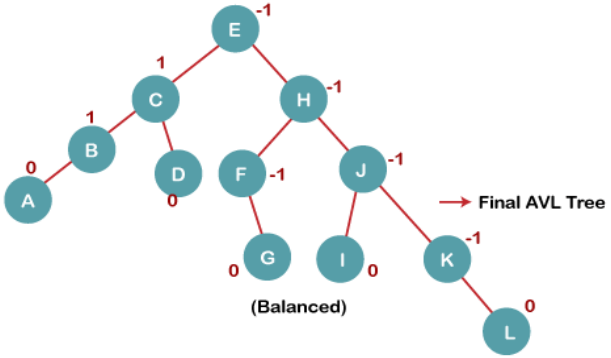


6. Insert K





7. Insert L



AVL Tree Deletion

- Let w be the node to be deleted

  1) Perform standard BST delete for w.

  2) Starting from leaf, travel up and find the first unbalanced node.

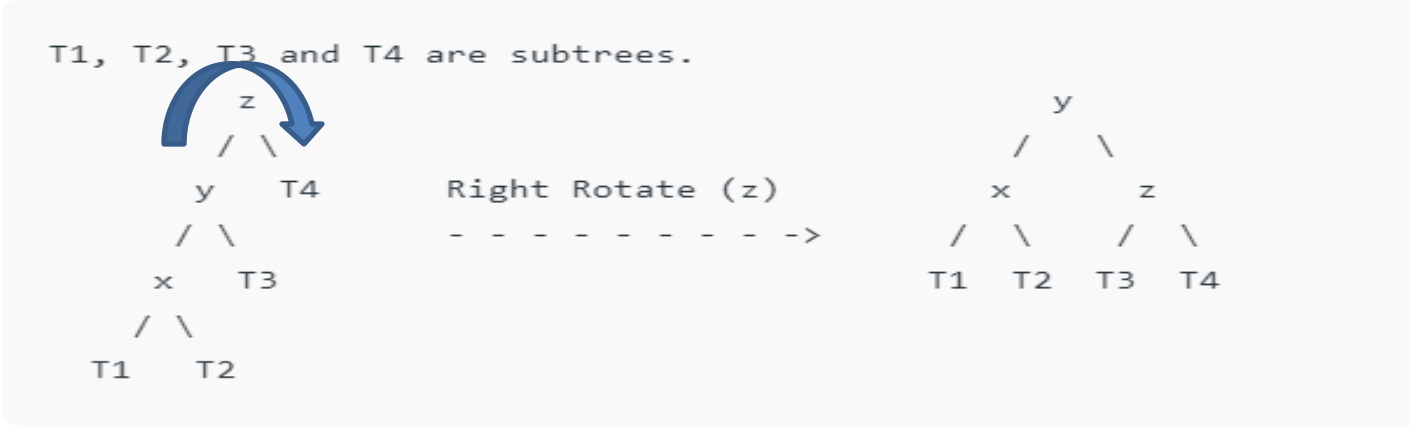
  3) Let ‘z‘be the first unbalanced node, ‘y’ be the larger height child of z, and ‘x’ be the larger height child of y.

  4) Re-balance the tree by performing appropriate rotations on the subtree rooted with z.

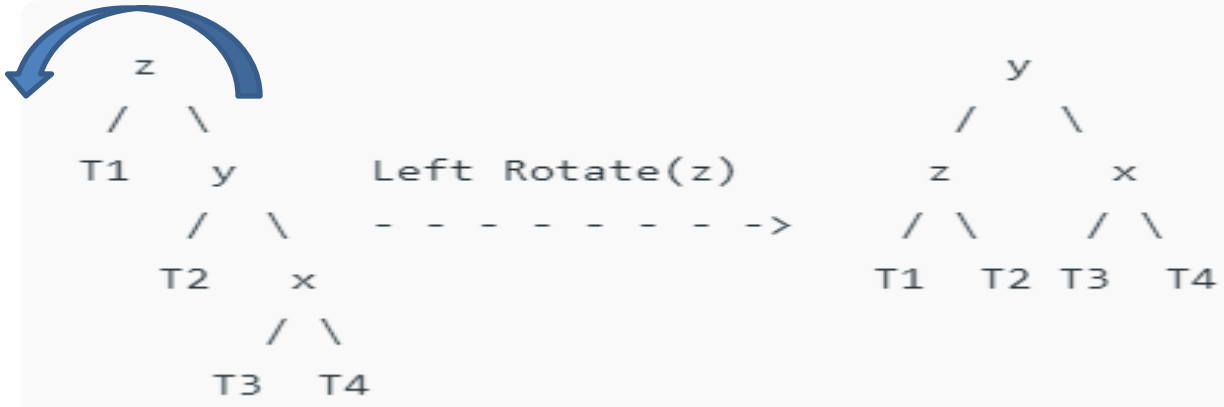
- There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways.
  a) **Left Left Case** (y is left child of z and x is left child of y)
  b) **Left Right Case** (y is left child of z and x is right child of y)
  c) **Right Right Case** (y is right child of z and x is right child of y)
  d) **Right Left Case** (y is right child of z and x is left child of y)

AVL Tree Deletion

a) Left Left Case

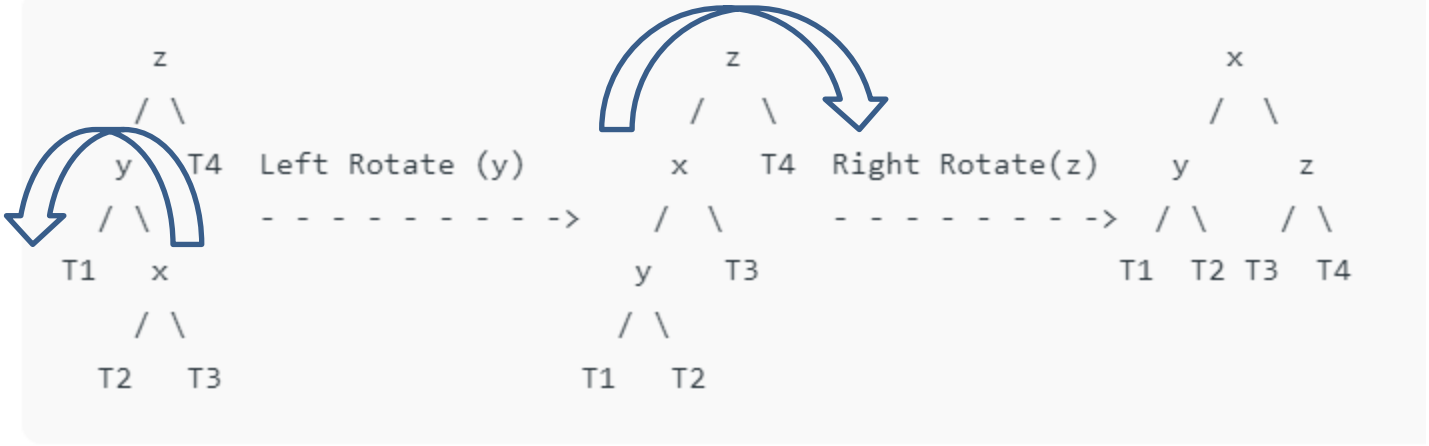


c) Right Right Case

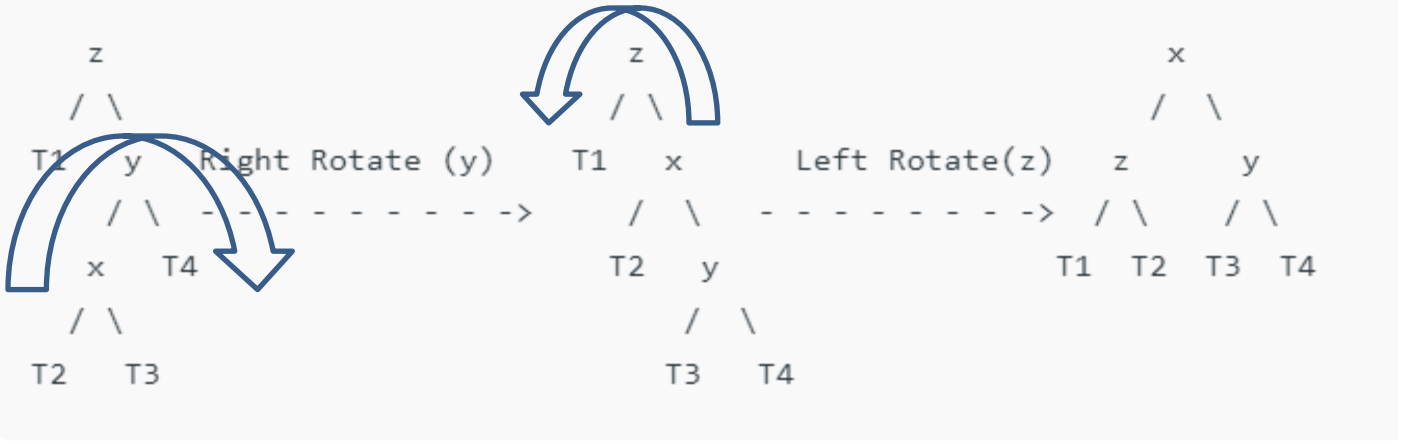


AVL Tree Deletion

b) Left Right Case



d) Right Left Case

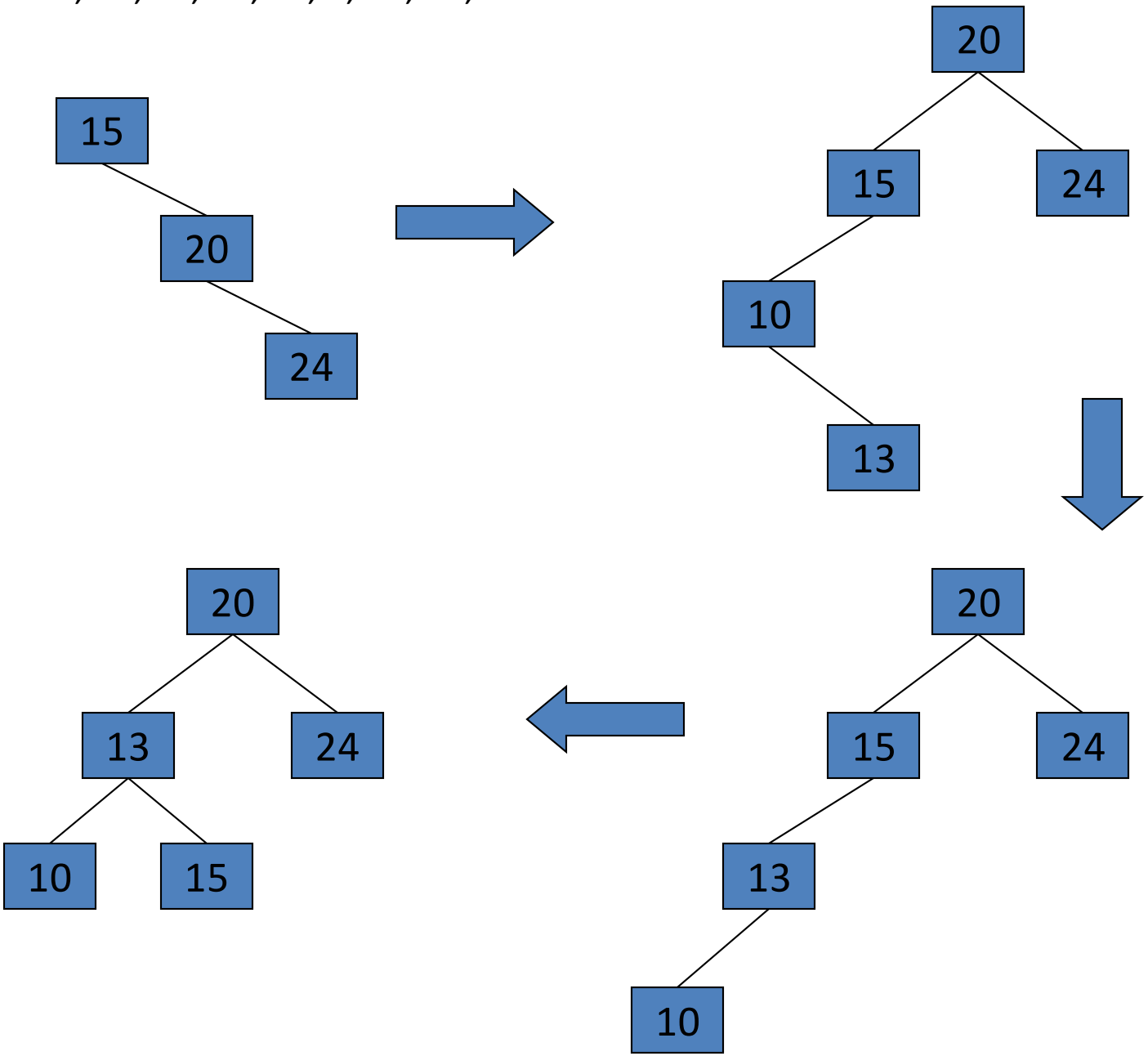


In Class Exercise2

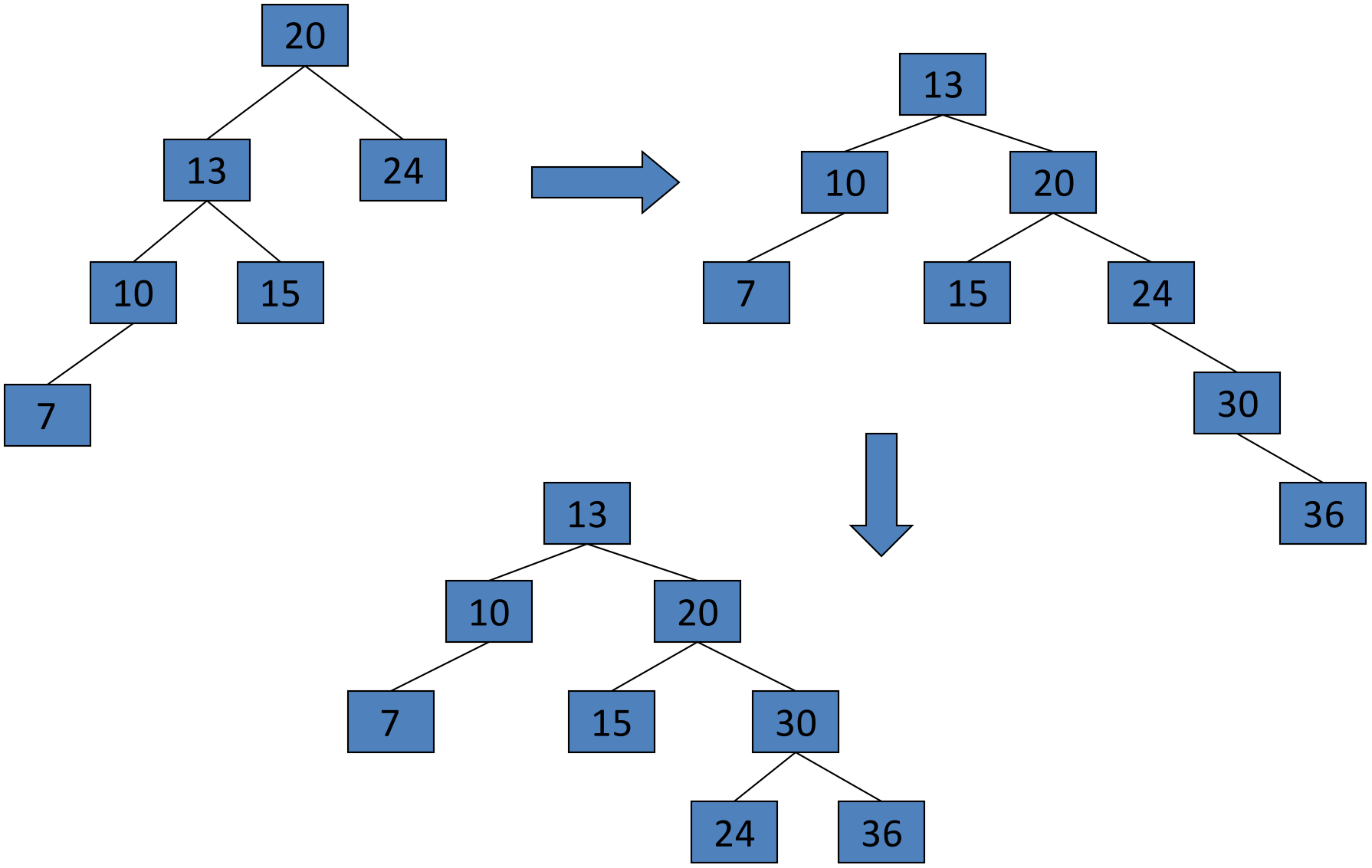
- Build an AVL tree with the following values:

  15, 20, 24, 10, 13, 7, 30, 36, 25

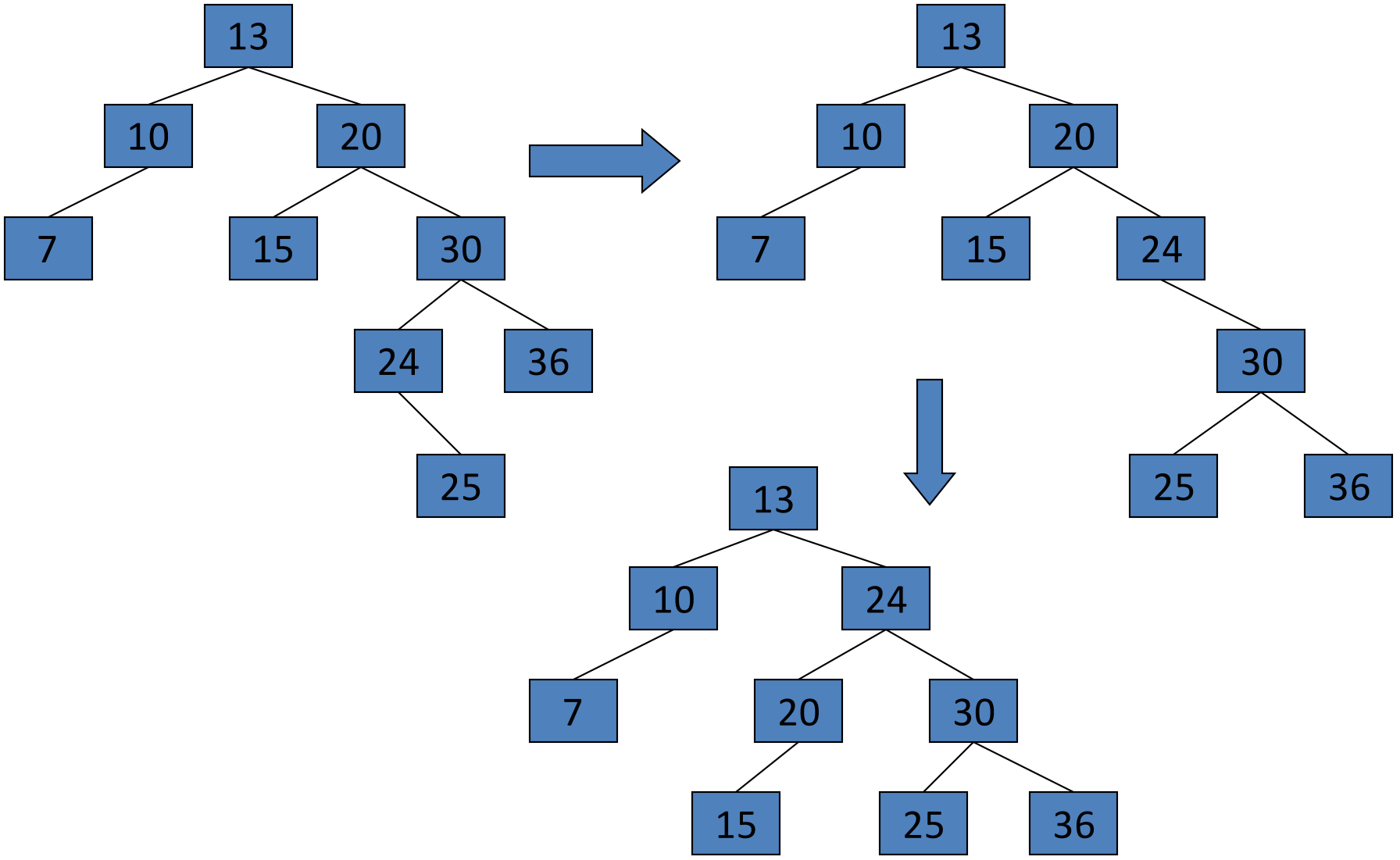
15, 20, 24, 10, 13, 7, 30, 36, 25



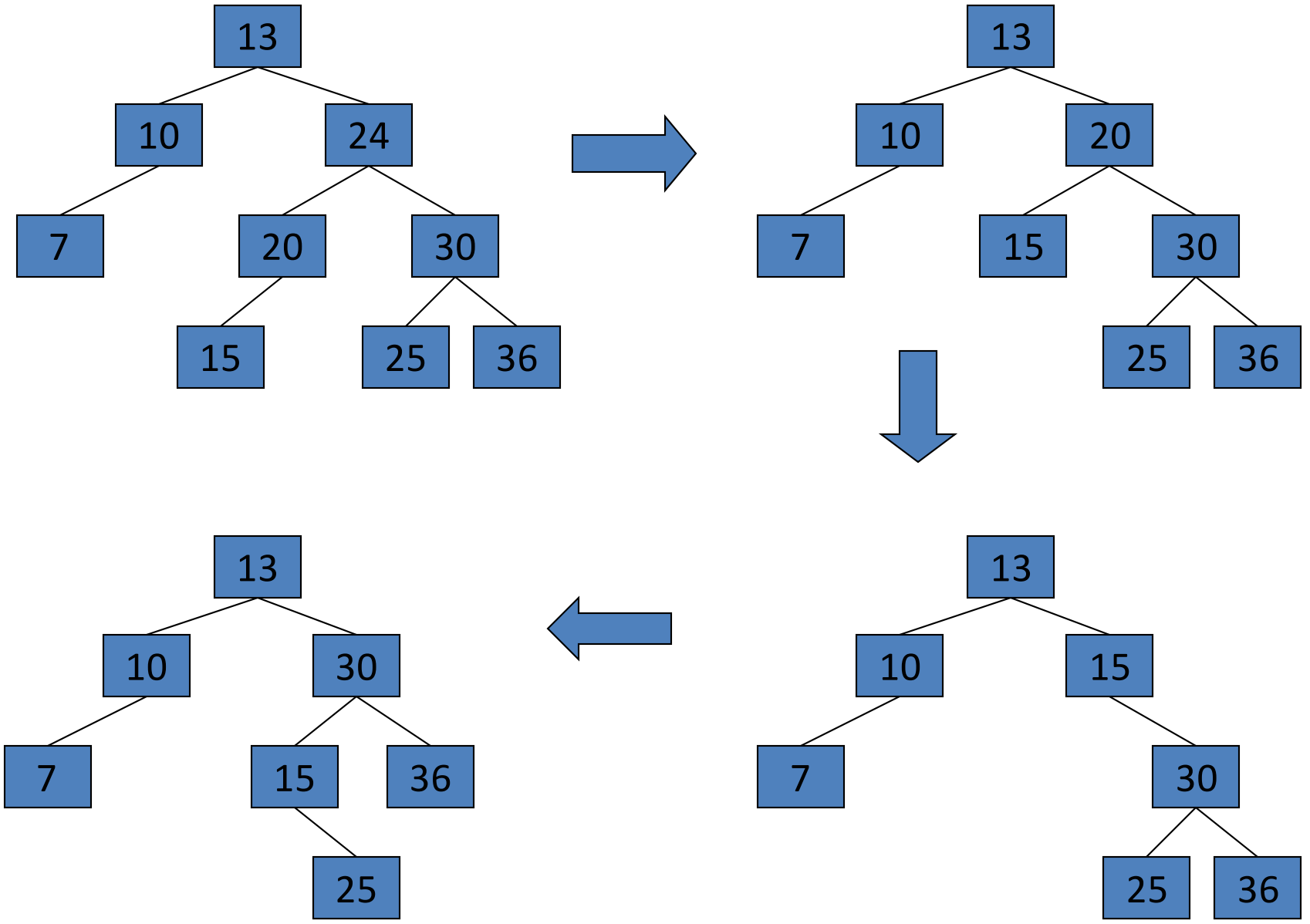
15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25

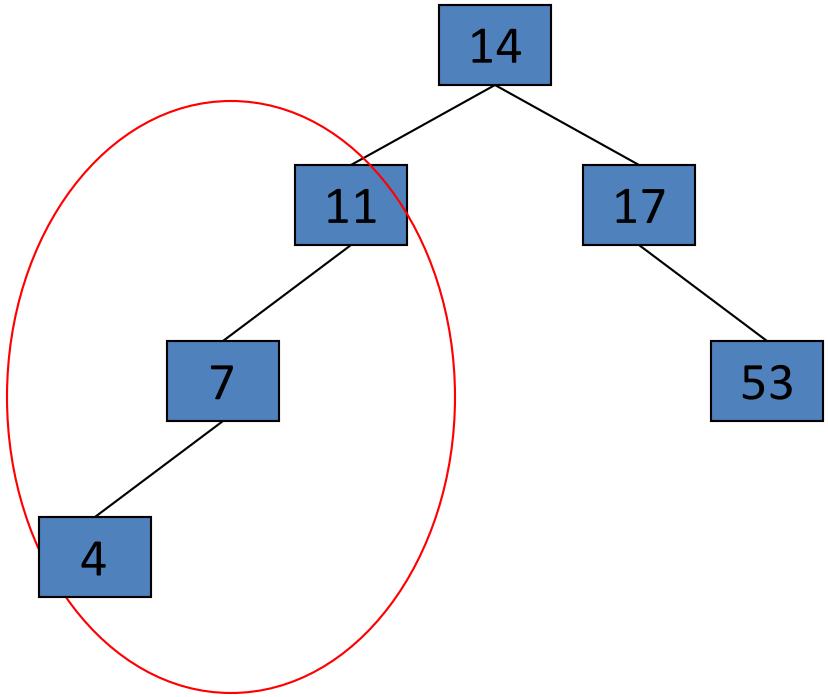


Remove 24 and 20 from the AVL tree.



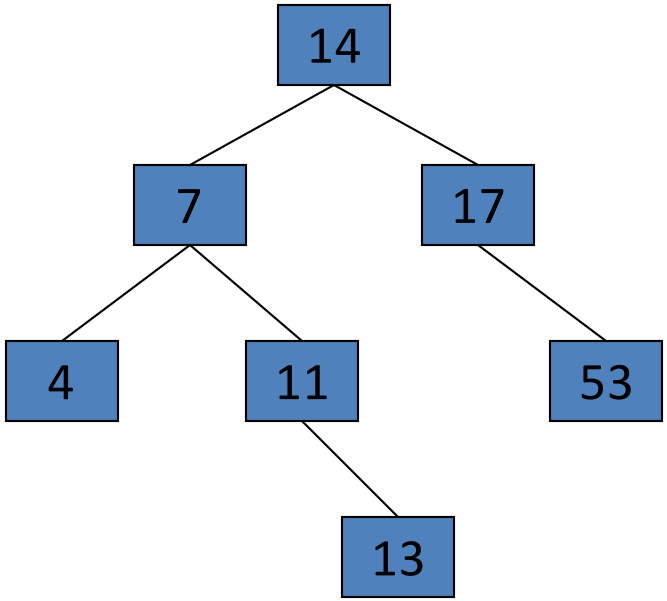
Another Example for AVL Tree

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



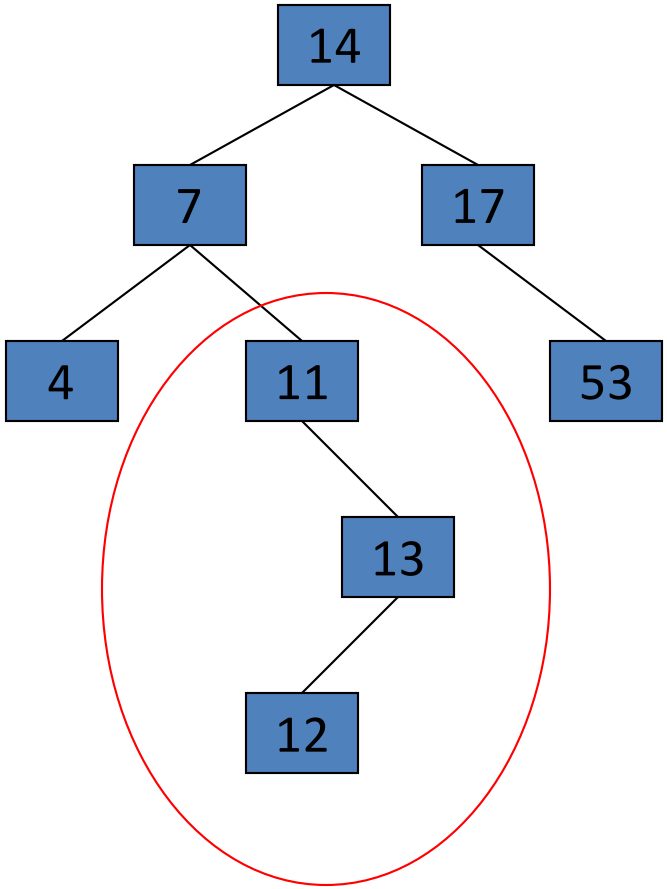
AVL Tree Example:

- **Insert 13 into an AVL tree**



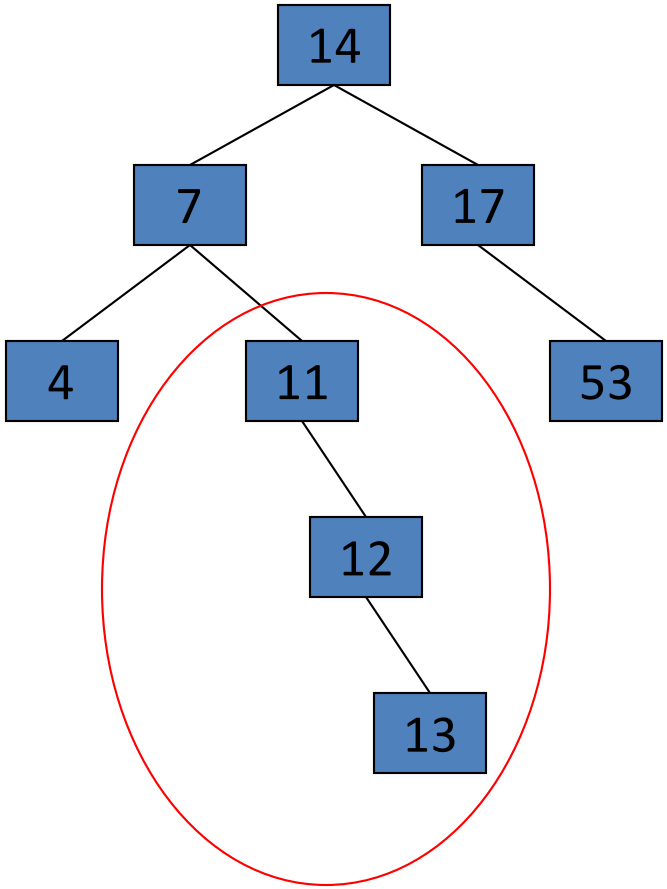
AVL Tree Example:

- **Now insert 12**



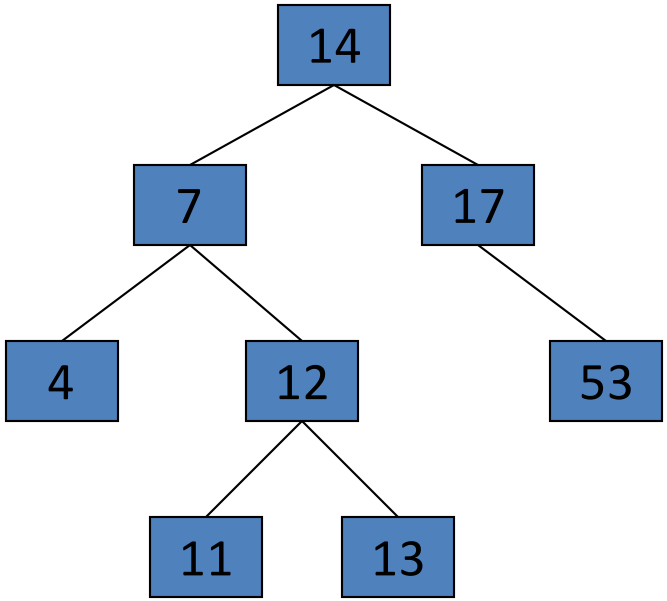
AVL Tree Example:

- **Now insert 12**



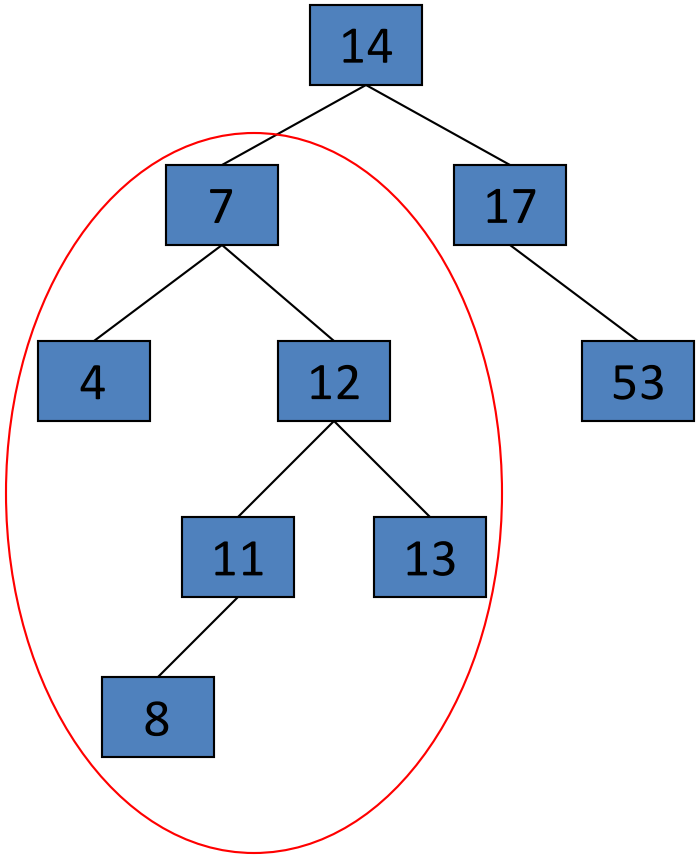
AVL Tree Example:

- **Now the AVL tree is balanced.**



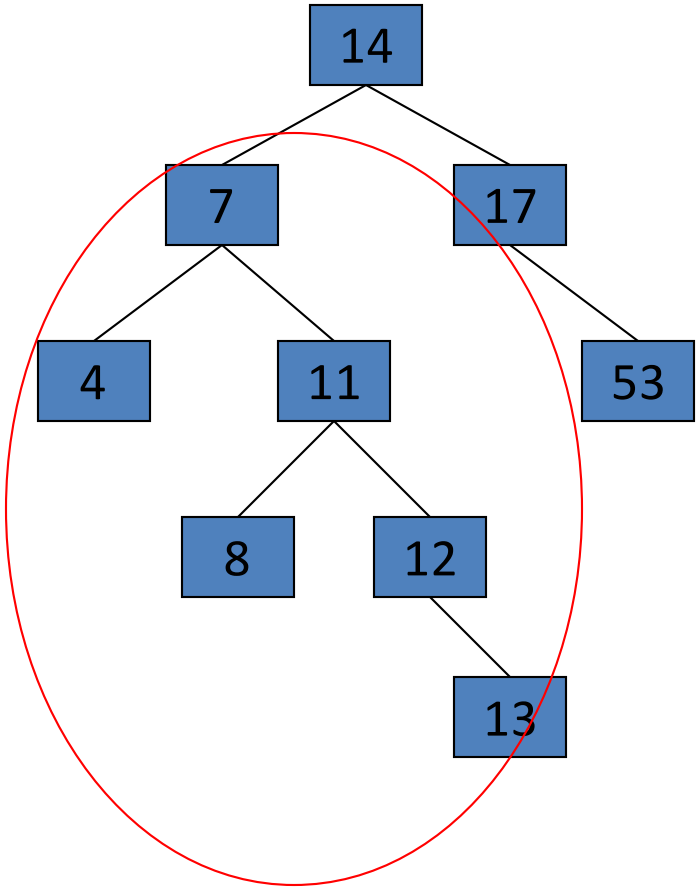
AVL Tree Example:

- **Now insert 8**



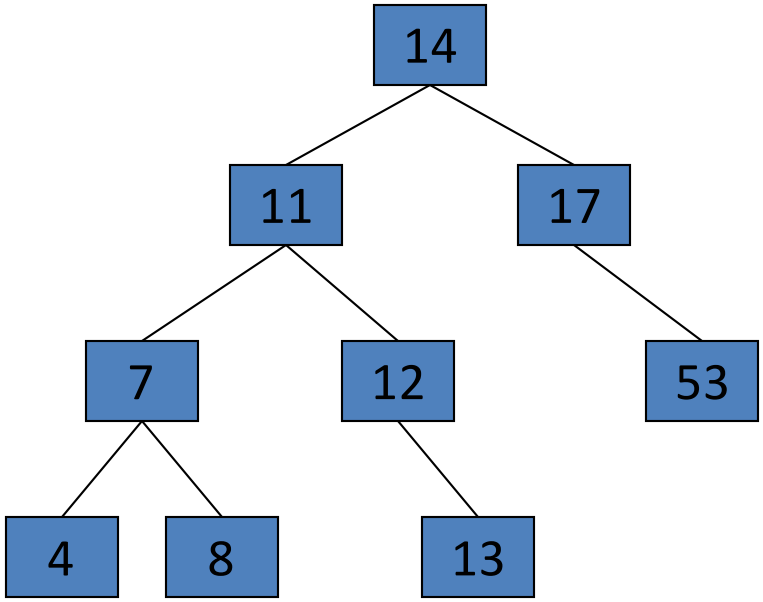
AVL Tree Example:

- **Now insert 8**



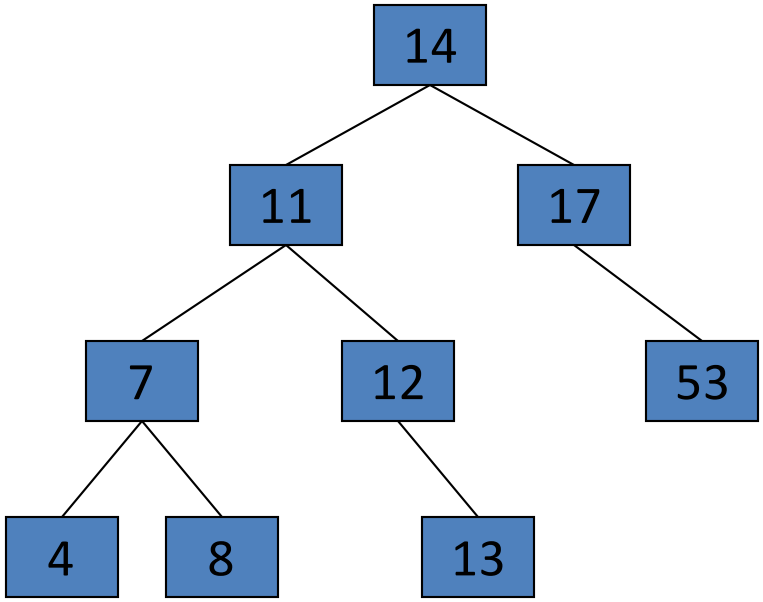
AVL Tree Example:

- **Now the AVL tree is balanced.**



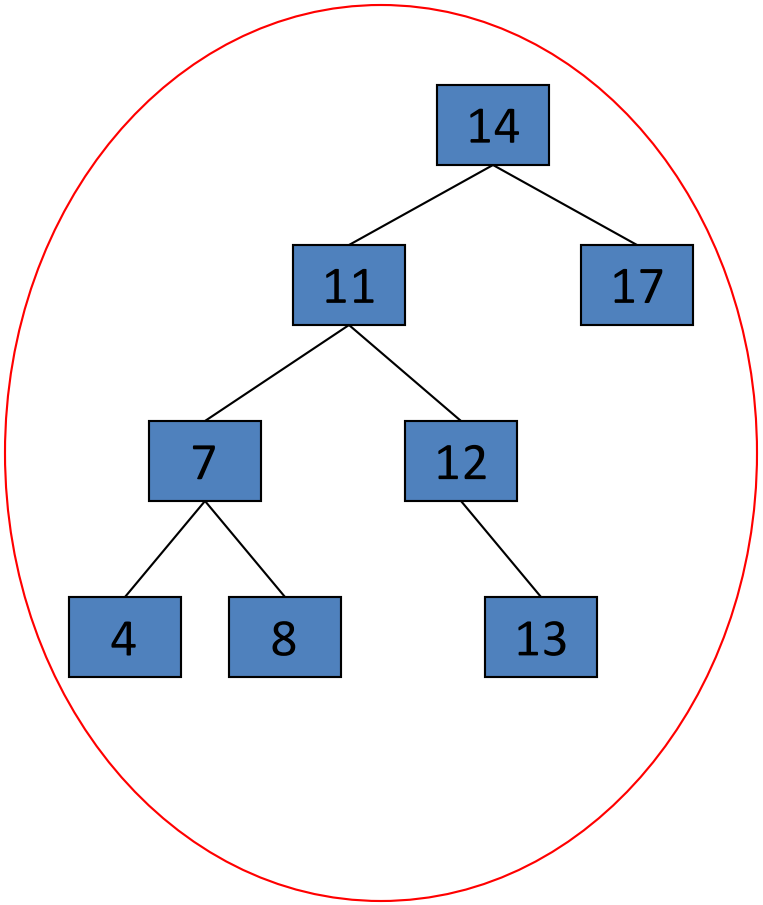
AVL Tree Example:

- **Now remove 53**



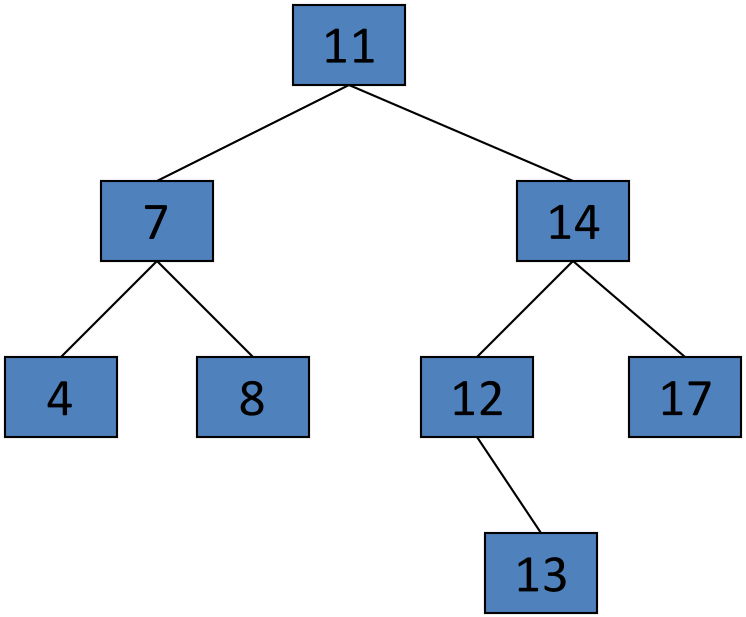
AVL Tree Example:

- **Now remove 53, unbalanced**



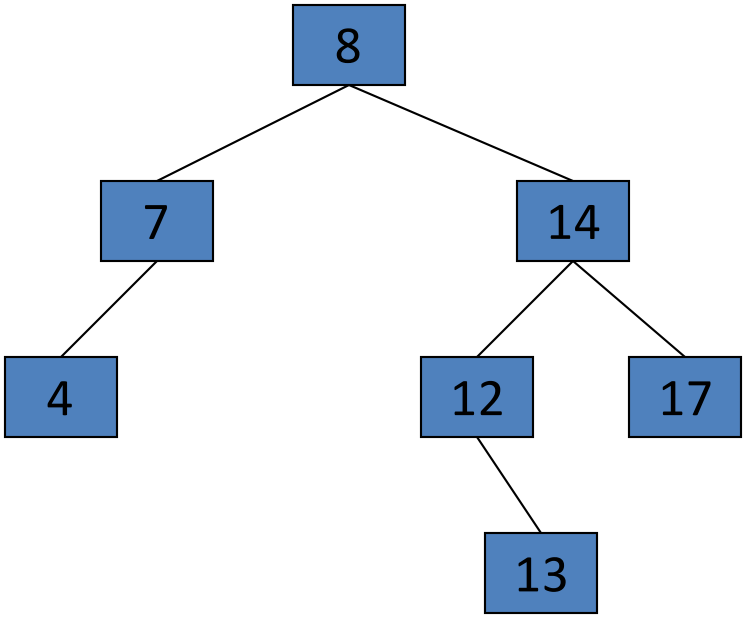
AVL Tree Example:

- **Balanced! Remove 11**



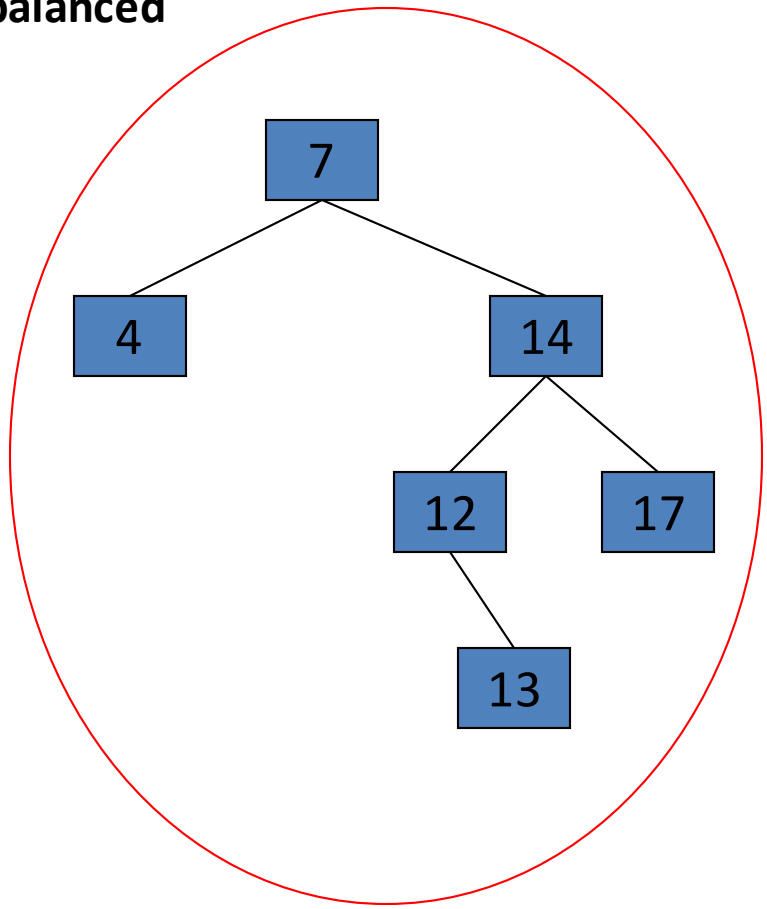
AVL Tree Example:

- **Remove 11, replace it with the largest in its left branch**



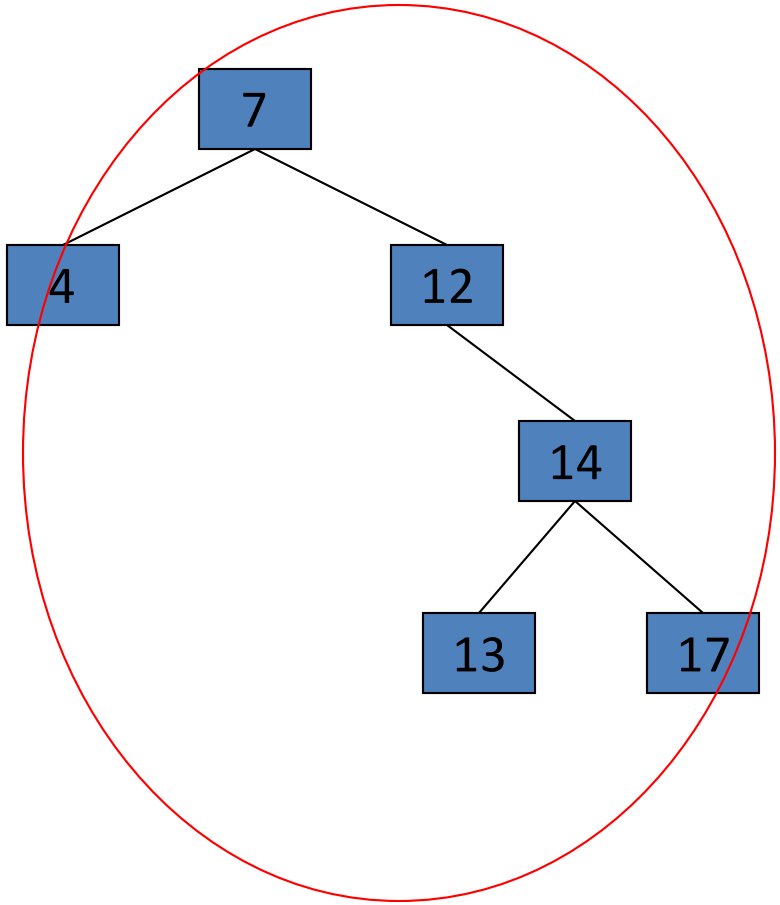
AVL Tree Example:

- **Remove 8, unbalanced**



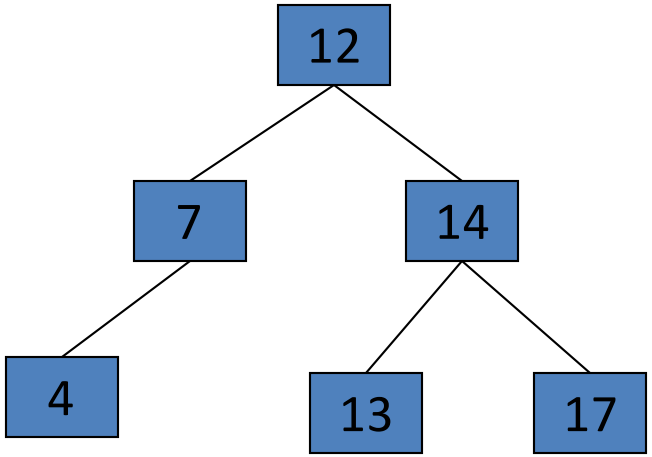
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

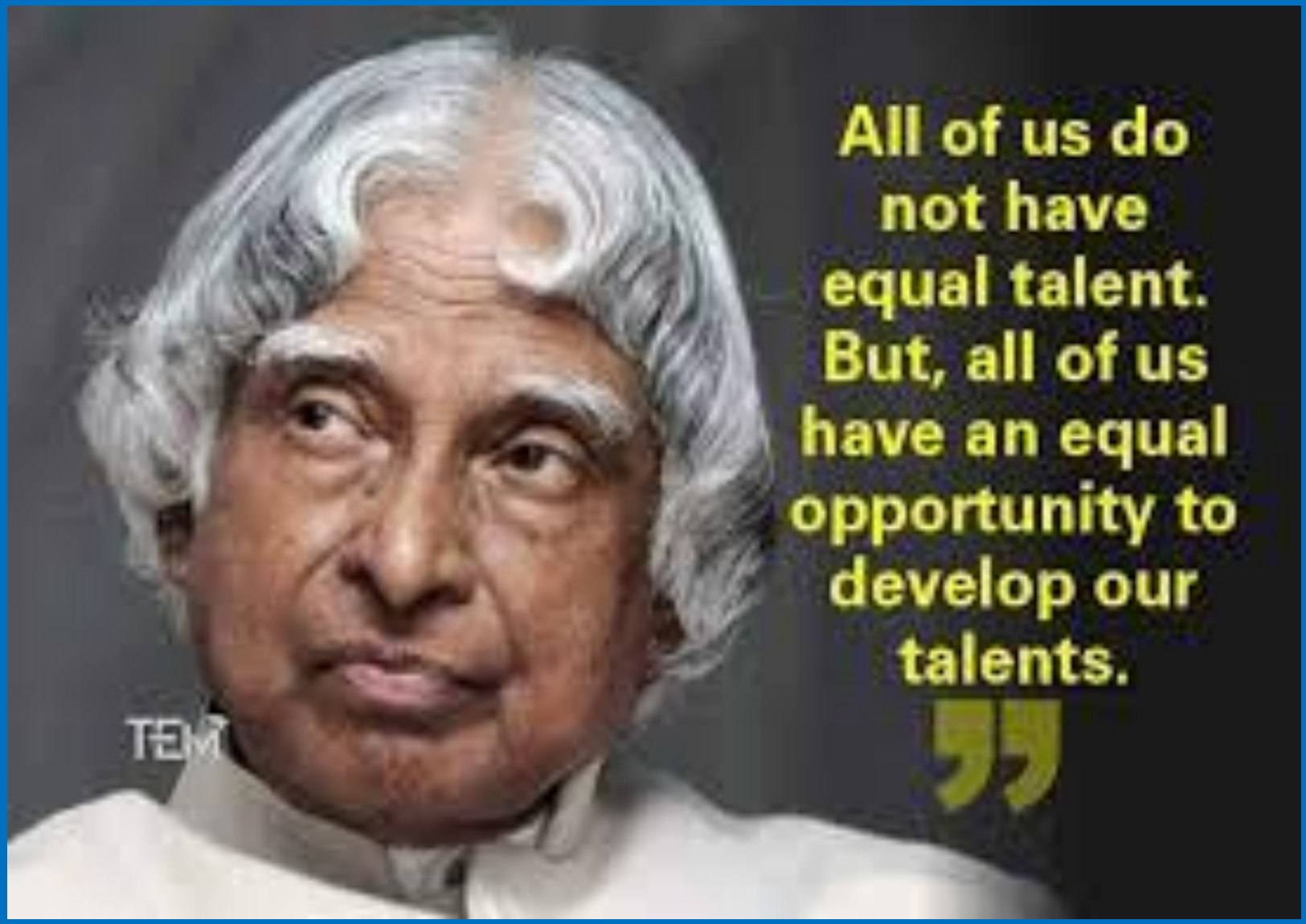
- **Balanced!!**



Exercise on AVL Trees

Construct an AVL tree for the list and after the construction make the tree NULL by deleting the elements in the same sequence.

1, 7, 2, 9, 4, 6, 3, 10, 5, 11, 13, 17, 12, 20,16,19,18,15.


All of us do not have equal talent. But, all of us have an equal opportunity to develop our talents.
”