AVL Tree

- Shital Dongre, VIT, Pune

Background

So far …

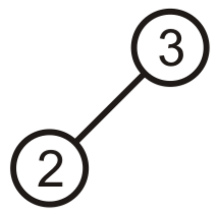
- Binary search trees store linearly ordered data
- Best case height: $\Theta(\ln(n))$
- Worst case height: $\mathbf{O}(n)$

Requirement:

- Define and maintain a *balance* to ensure $\Theta(\ln(n))$ height

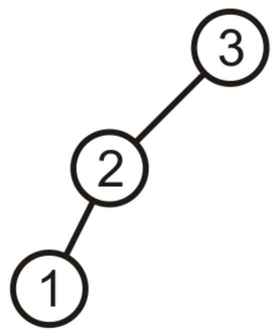
Prototypical Examples

These two examples demonstrate how we can correct for imbalances: starting with this tree, add 1:



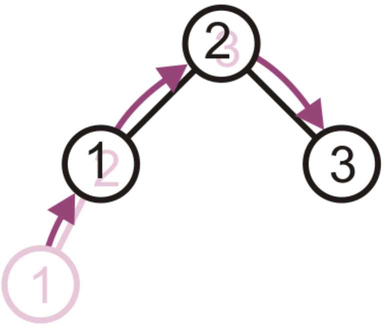
Prototypical Examples

This is more like a linked list; however, we can fix this…



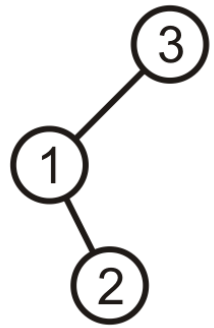
Prototypical Examples

Promote 2 to the root, demote 3 to be 2’s right child, and 1 remains the left child of 2



Prototypical Examples

Again, the product is a linked list; however, we can fix this, too



Why AVL Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.
- The cost of these operations may become O(n) for a skewed Binary tree.
- If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations.
- The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree

AVL Trees

We will focus on the first strategy: AVL trees

– Named after Adelson-Velskii and Landis

Balance is defined by comparing the height of the two sub-trees

Recall:

– An empty tree has height -1

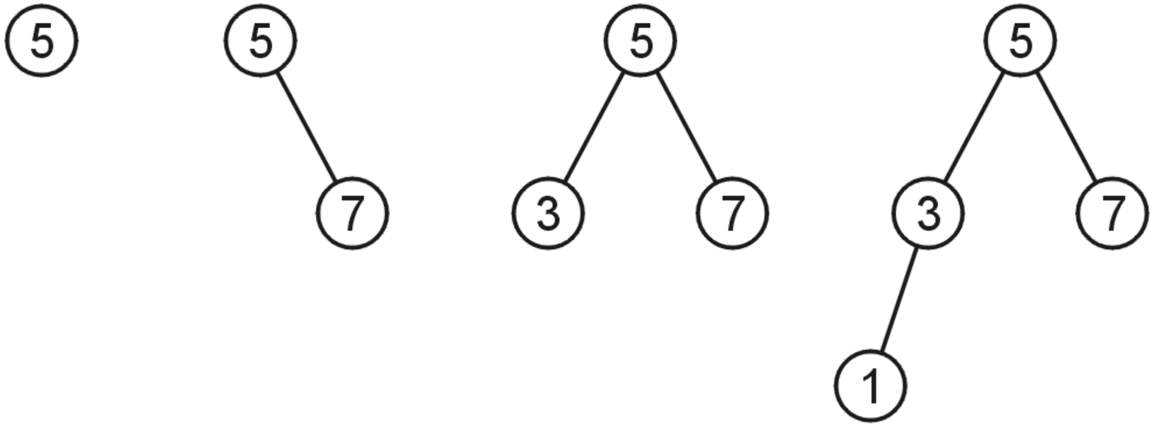
– A tree with a single node has height 0

AVL Trees

- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- A binary search tree is said to be AVL balanced if:
  - The difference in the heights between the left and right sub-trees is at most 1, and
  - Both sub-trees are themselves AVL trees

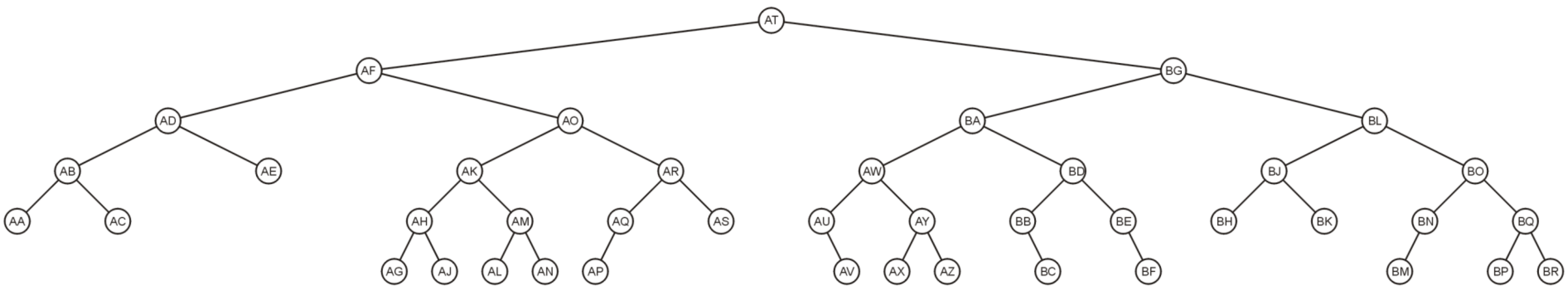
AVL Trees

AVL trees with 1, 2, 3, and 4 nodes:



AVL Trees

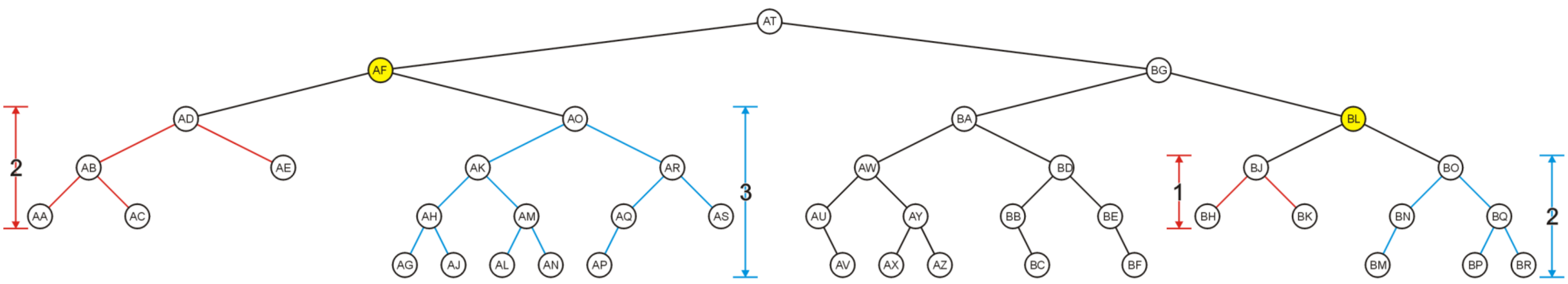
Here is a larger AVL tree (42 nodes):



AVL Trees

All other nodes are AVL balanced

– The sub-trees differ in height by at most one



Height of an AVL Tree

By the definition of complete trees, any complete binary search tree is an AVL tree

Thus an upper bound on the number of nodes in an AVL tree of height h a perfect binary tree with $2^{h+1} - 1$ nodes

Insertion

Steps to follow for insertion
Let the newly inserted node be w
1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3) 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. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (**Left Left** Case)
b) y is left child of z and x is right child of y (**Left Right** Case)
c) y is right child of z and x is right child of y (**Right Right** Case)
d) y is right child of z and x is left child of y (**Right Left** Case)

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

b) Left Right Case

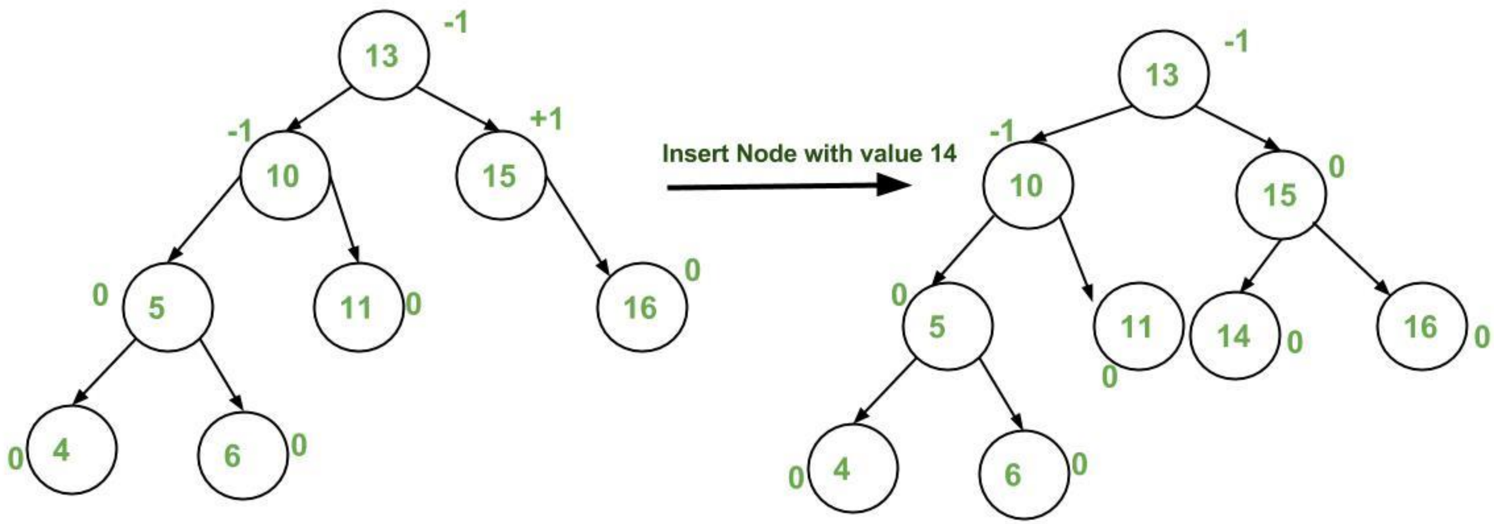
```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

c) Right Right Case

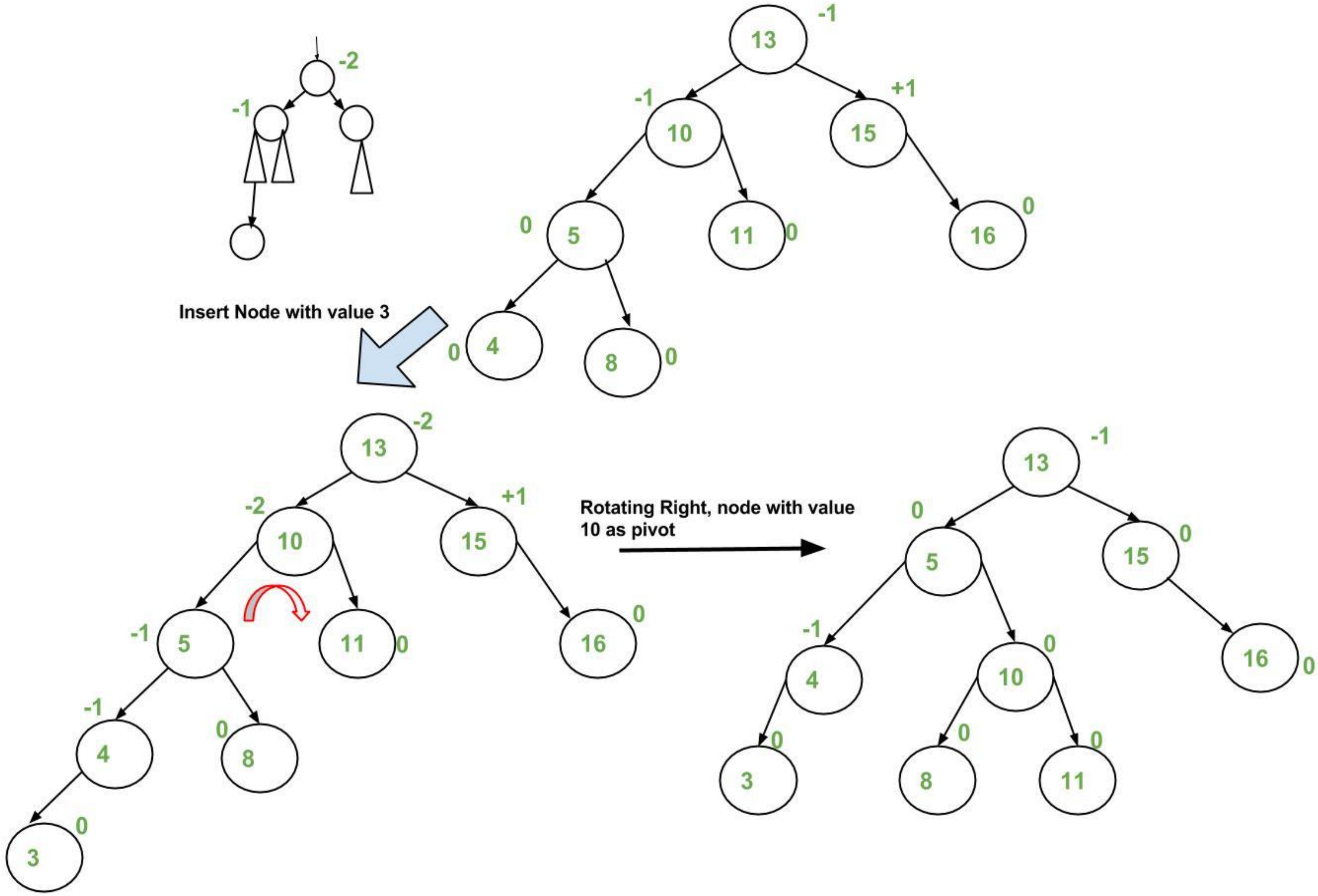
```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - - ->  / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

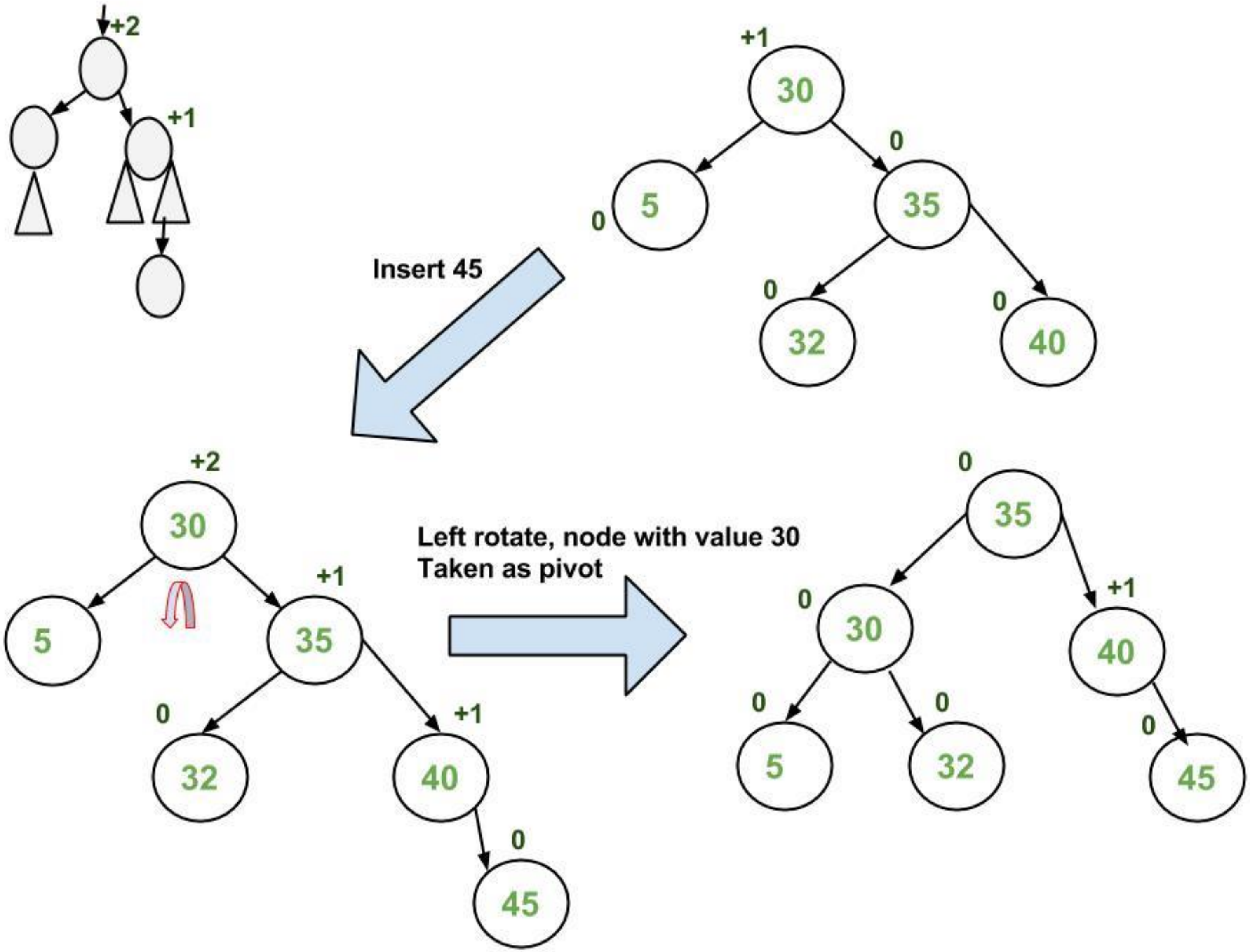
d) Right Left Case

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - -> / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                          T3   T4
```

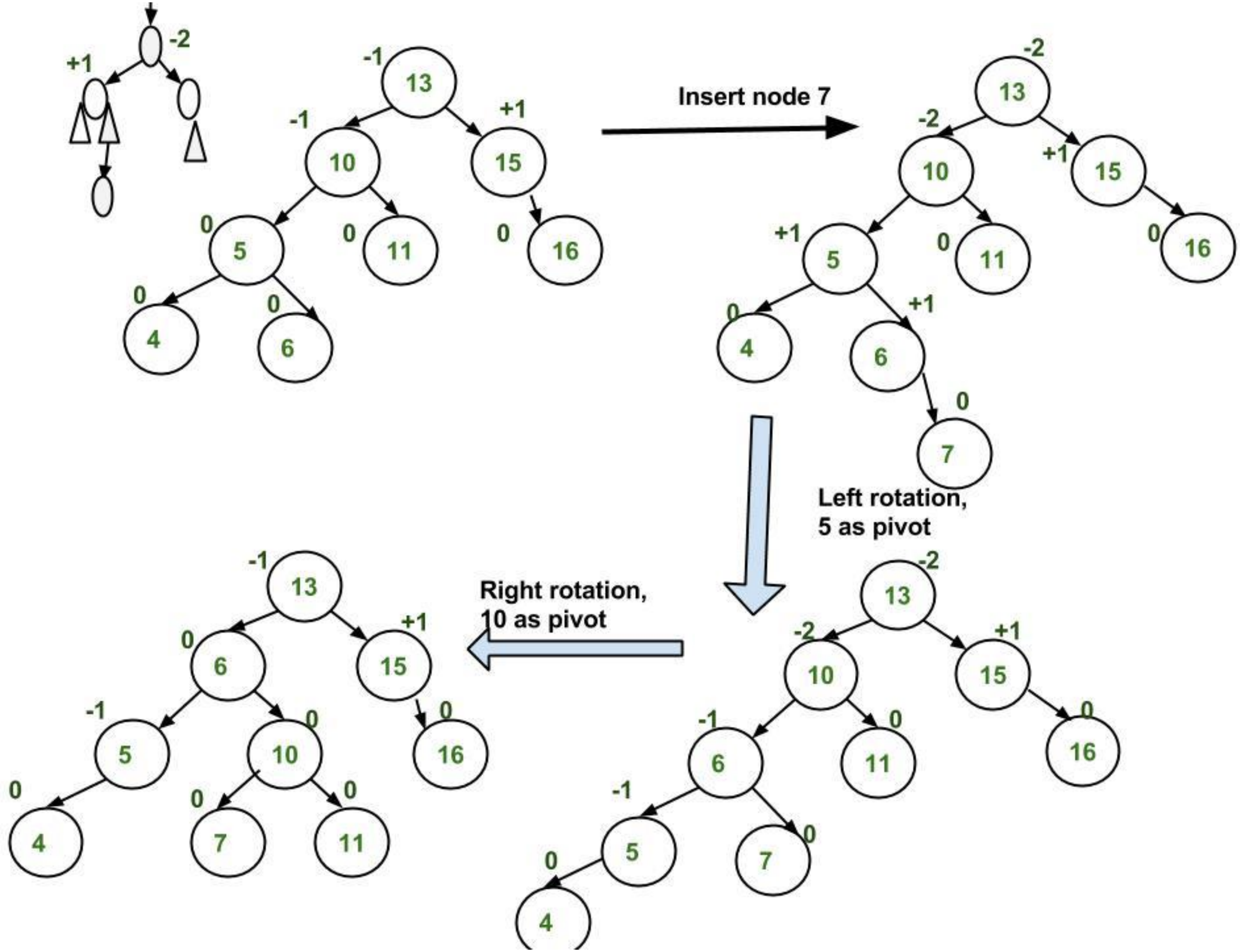


13
-1
10
-1
15
+1
5
0
11
0
16
0
4
0
6
0
Insert Node with value 14
13
-1
10
-1
15
0
5
0
11
0
14
0
16
0
4
0
6
0

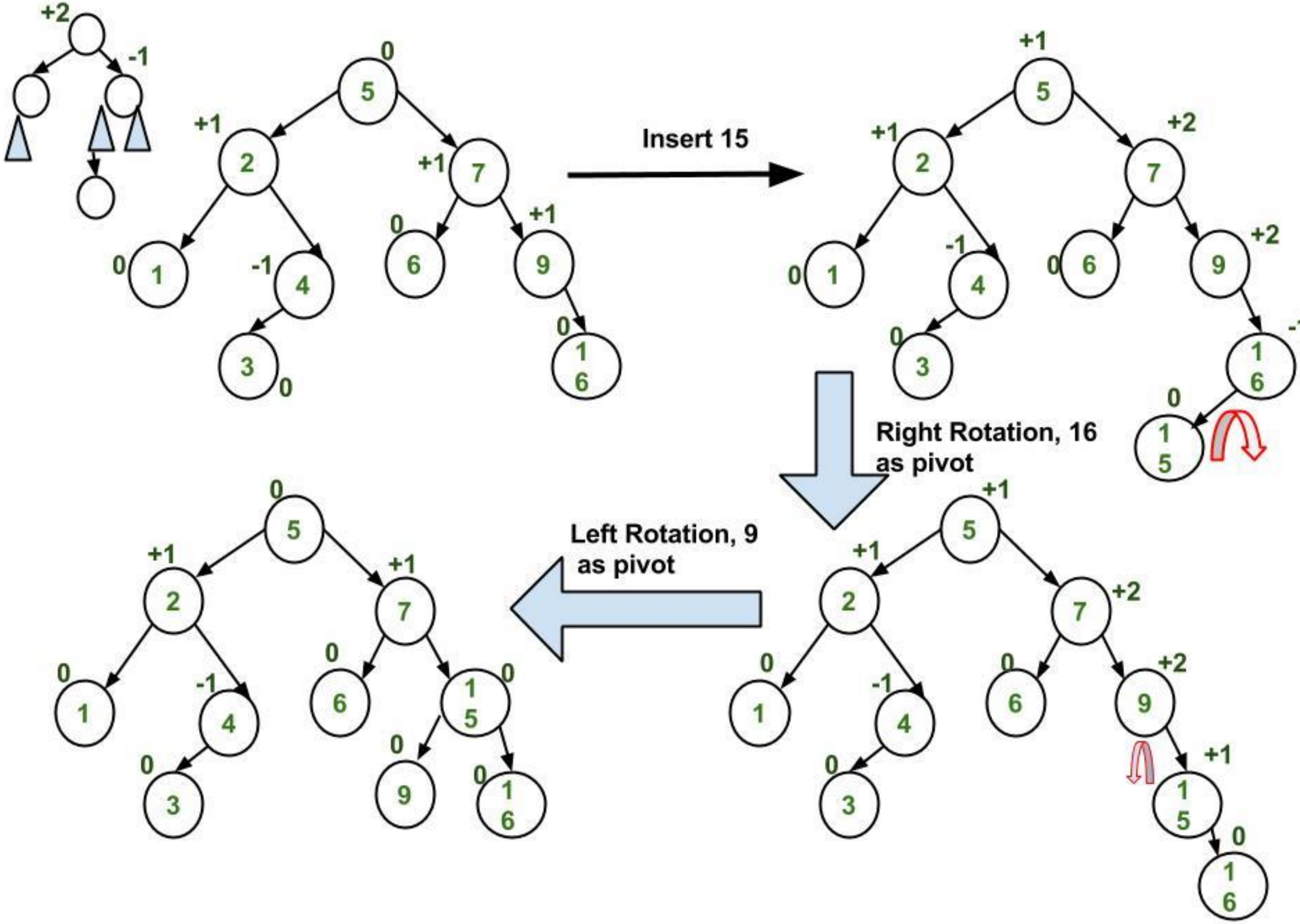

Insert Node with value 3
Rotating Right, node with value 10 as pivot



Insert 45
Left rotate, node with value 30
Taken as pivot



Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot

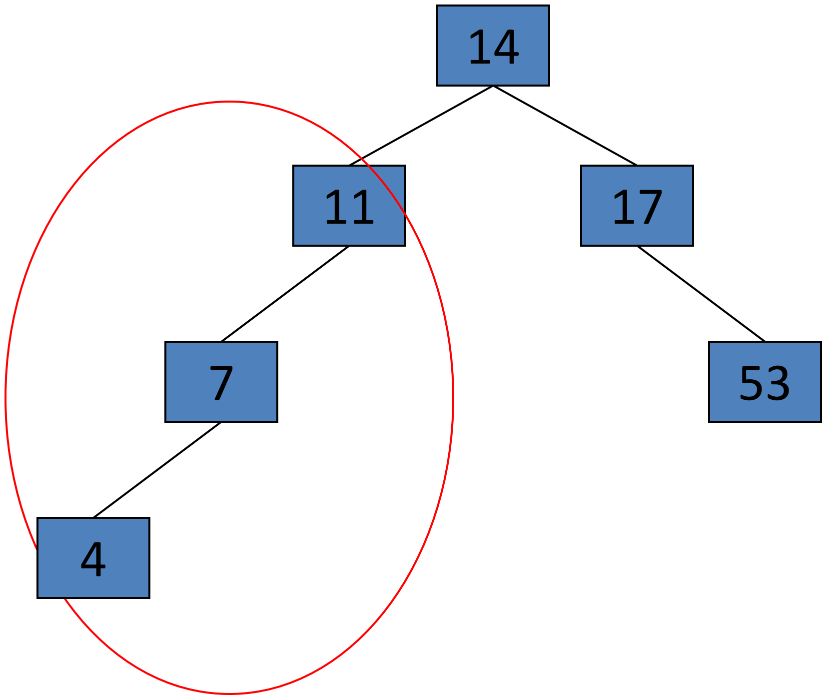

Insert 15
Right Rotation, 16
as pivot
Left Rotation, 9
as pivot

Examples

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree. Remove 53, 11, 8
- Build an AVL tree with the following values:
  - 15, 20, 24, 10, 13, 7, 30, 36, 25

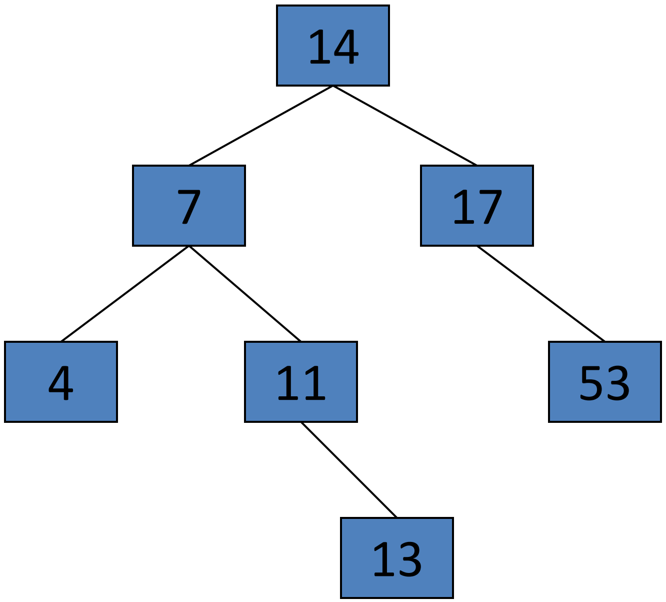
AVL Tree Example:

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



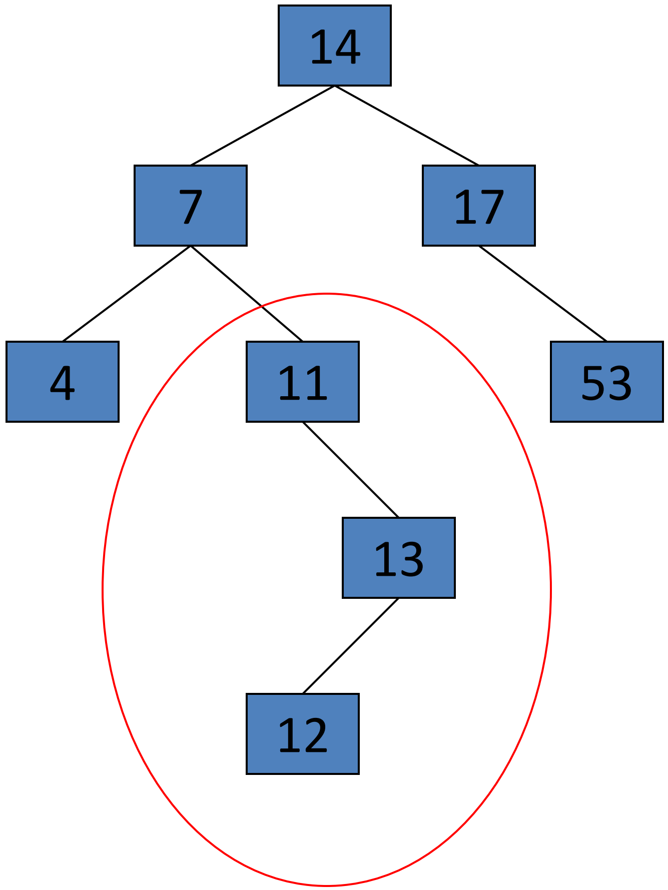
AVL Tree Example:

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



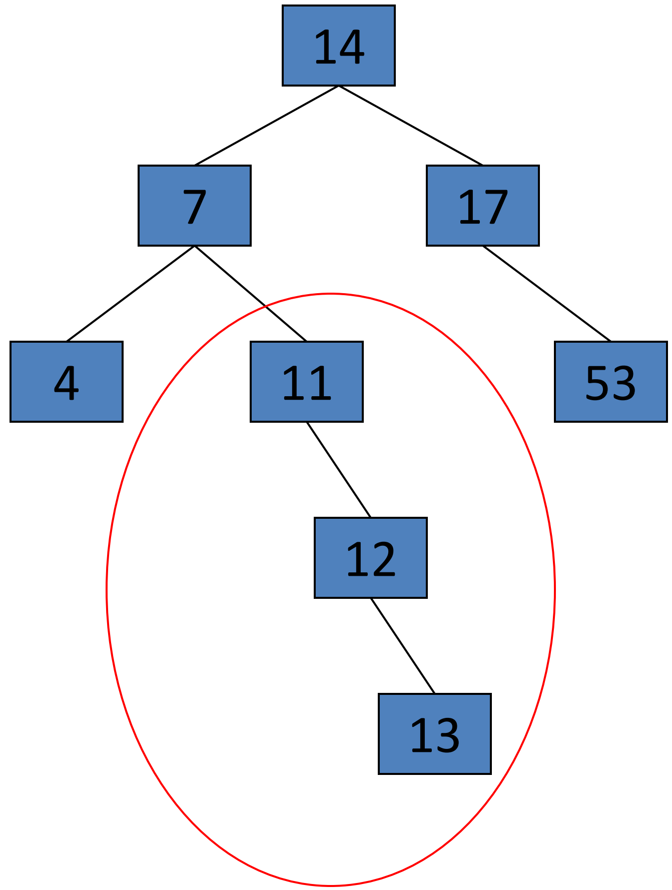
AVL Tree Example:

- **Now insert 12**



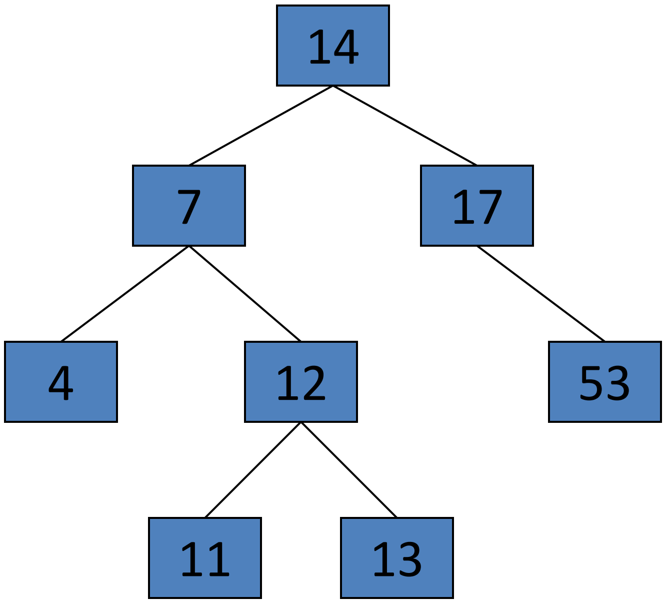
AVL Tree Example:

- **Now insert 12**



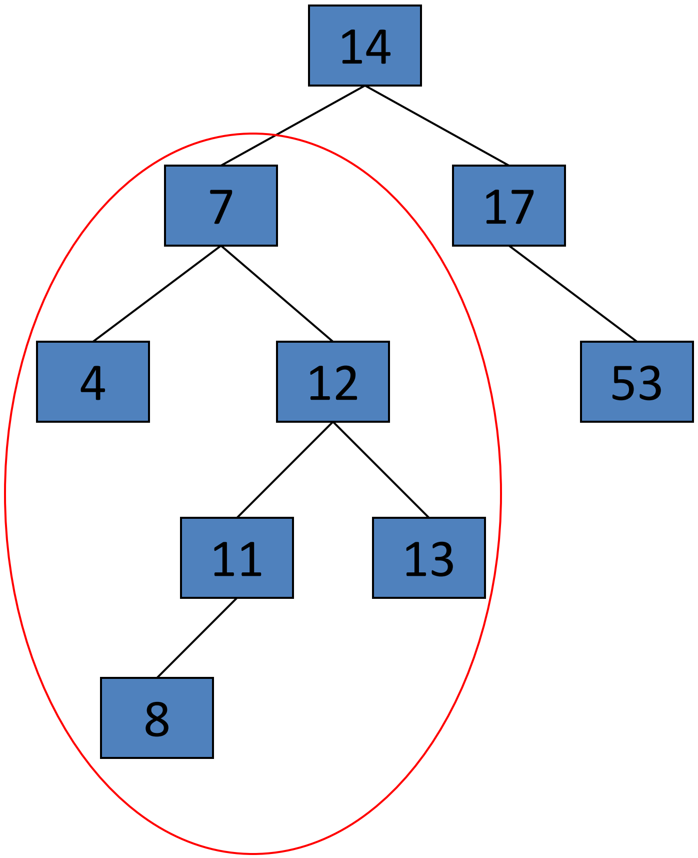
AVL Tree Example:

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



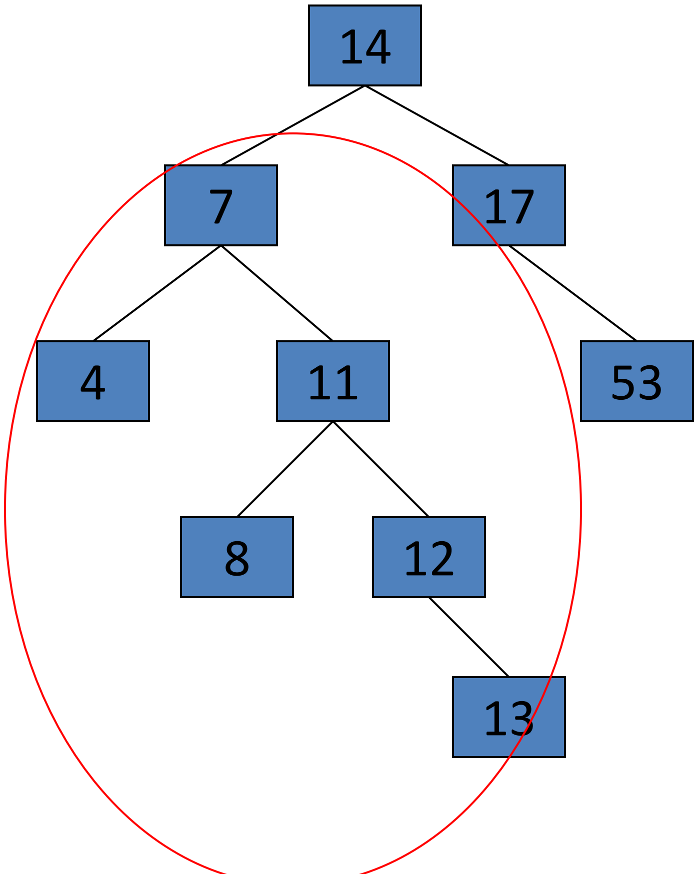
AVL Tree Example:

- **Now insert 8**



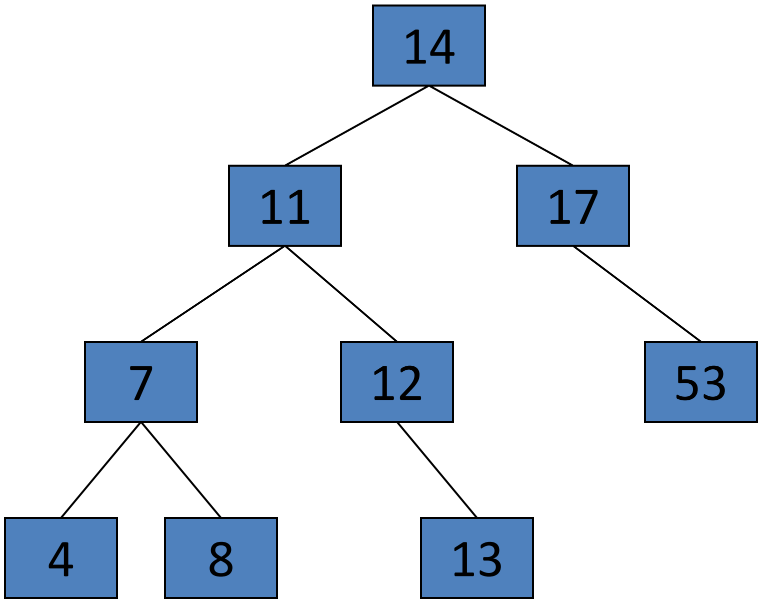
AVL Tree Example:

- **Now insert 8**



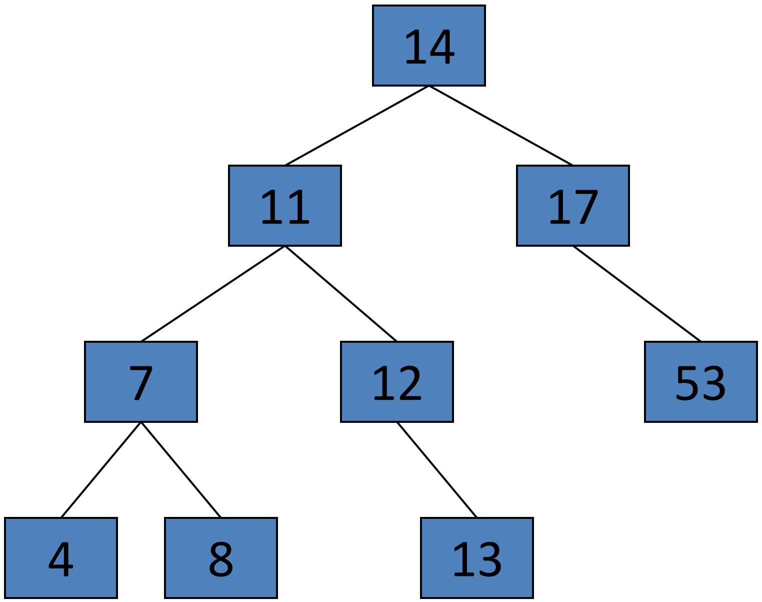
AVL Tree Example:

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



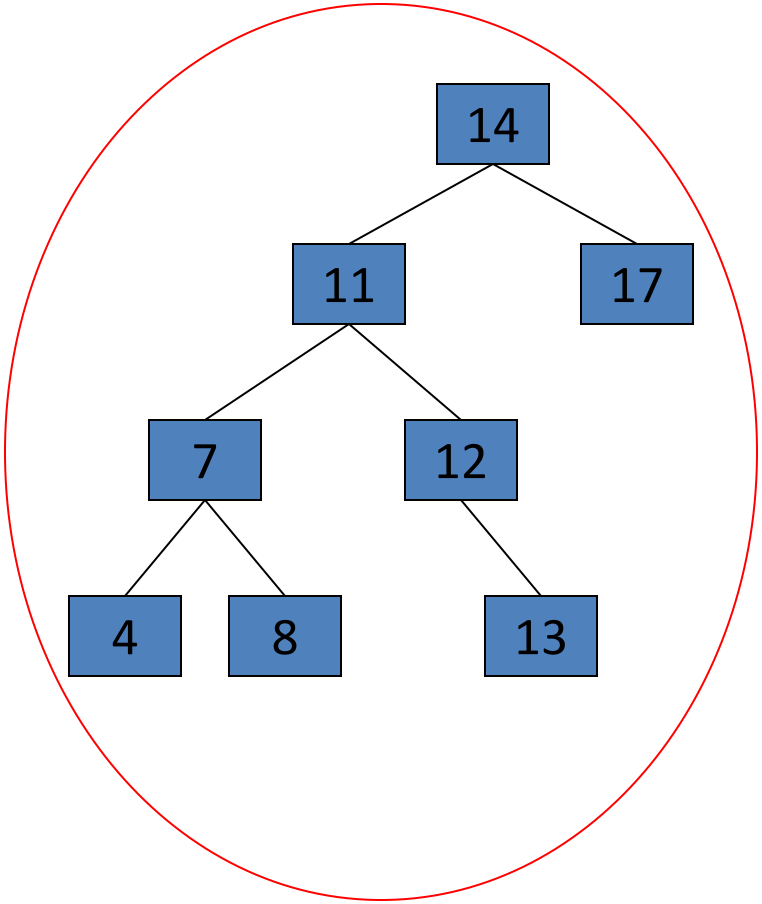
AVL Tree Example:

- **Now remove 53**



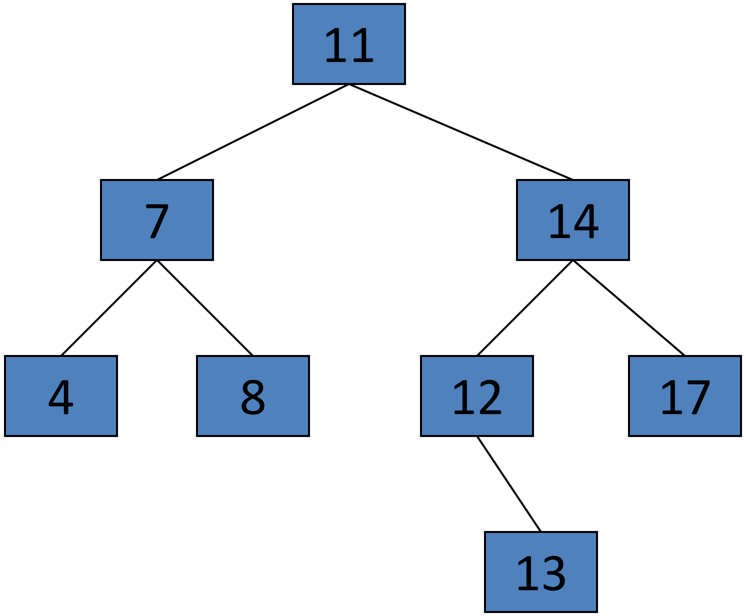
AVL Tree Example:

- **Now remove 53, unbalanced**



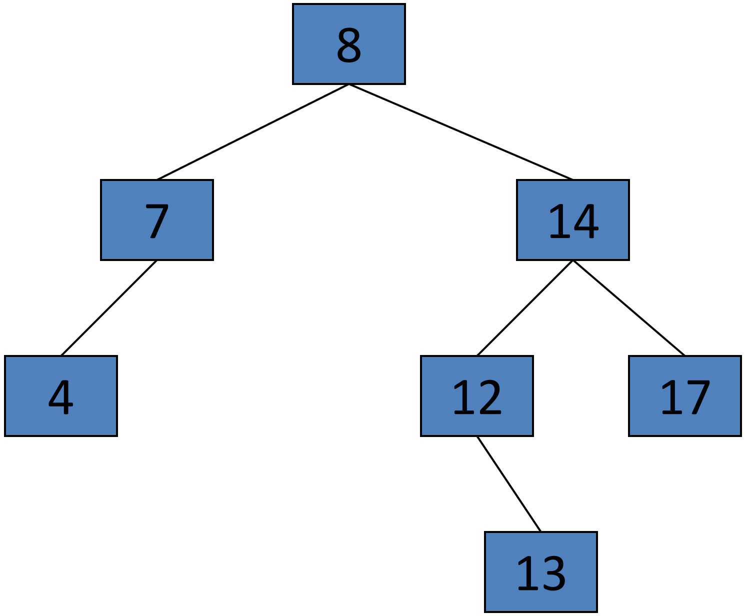
AVL Tree Example:

- **Balanced!**   **Remove 11**



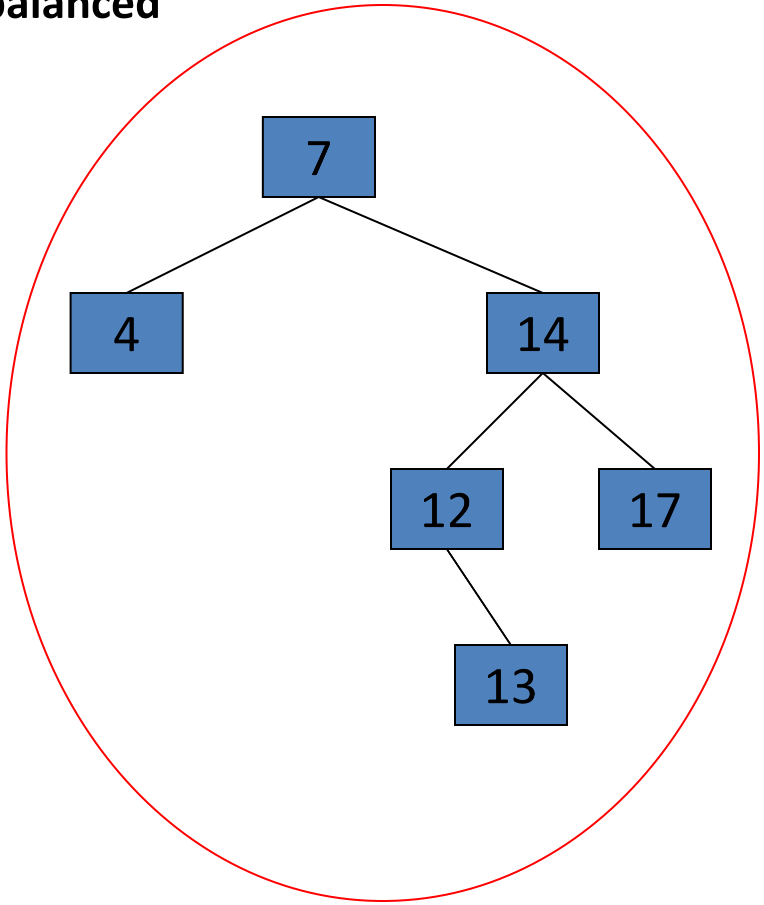
AVL Tree Example:

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



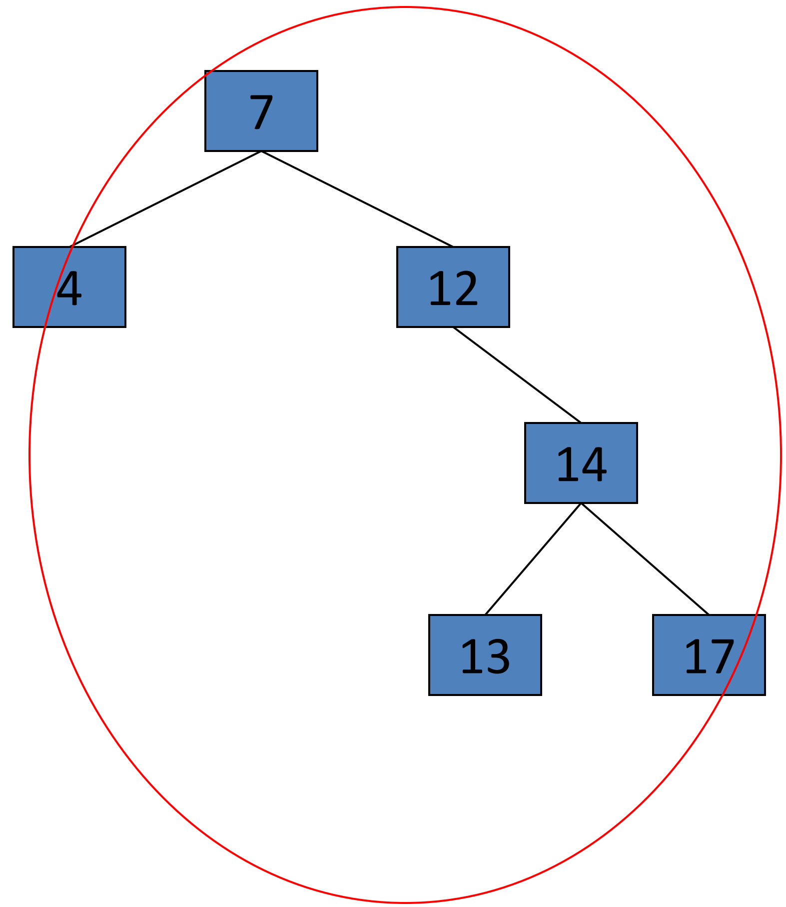
AVL Tree Example:

- **Remove 8, unbalanced**



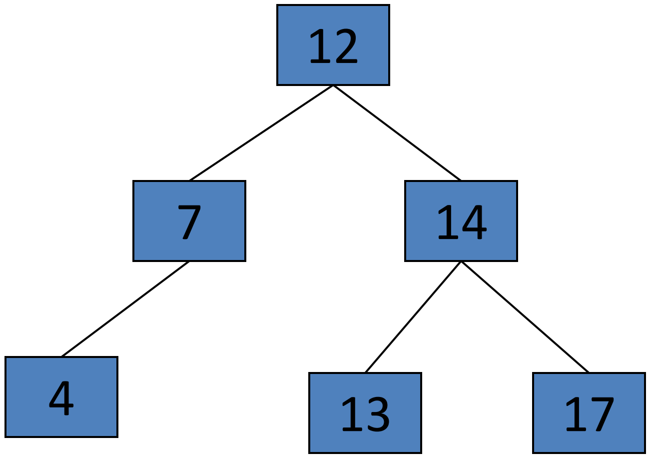
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

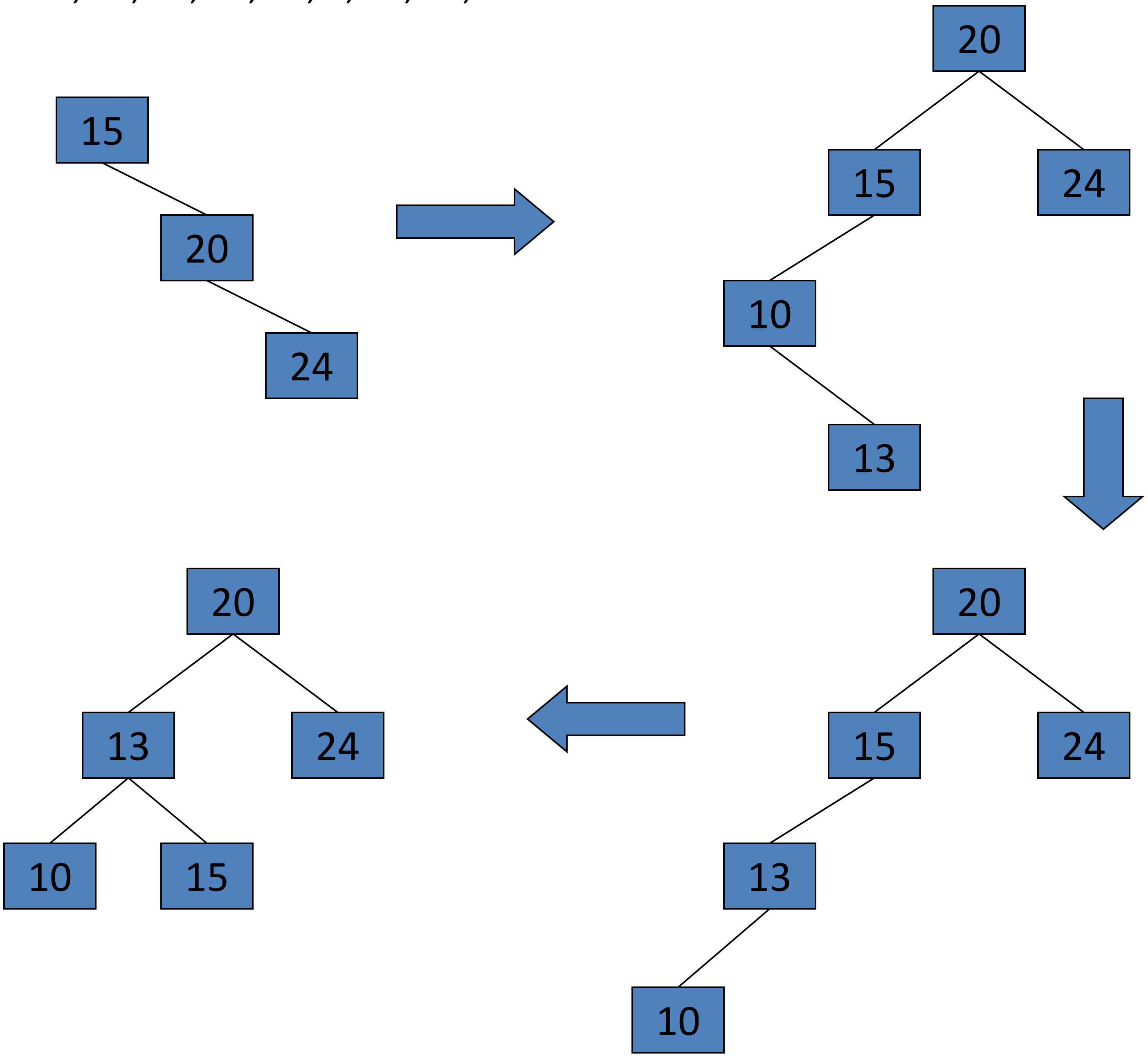
- **Balanced!!**



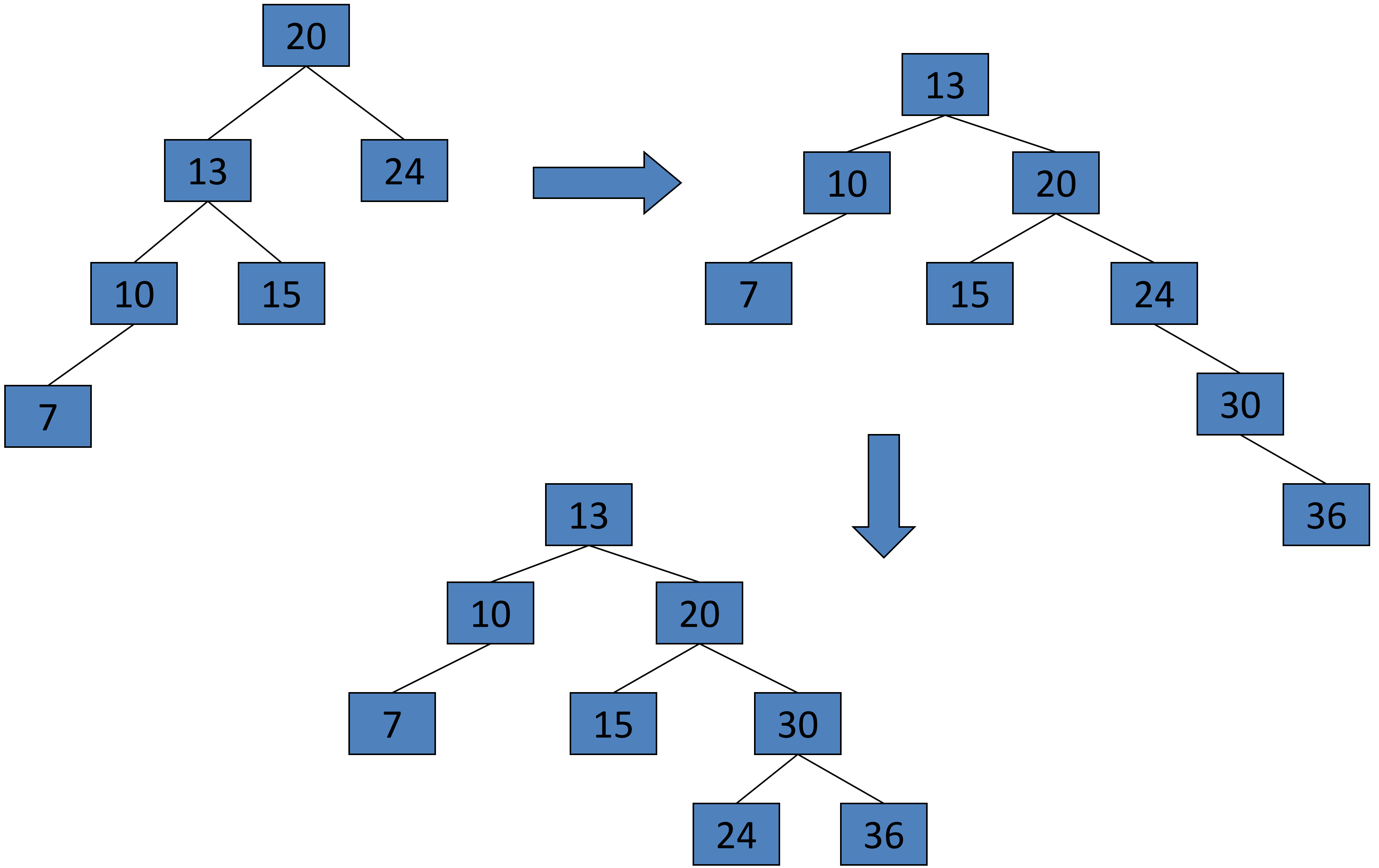
- Build an AVL tree with the following values:

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

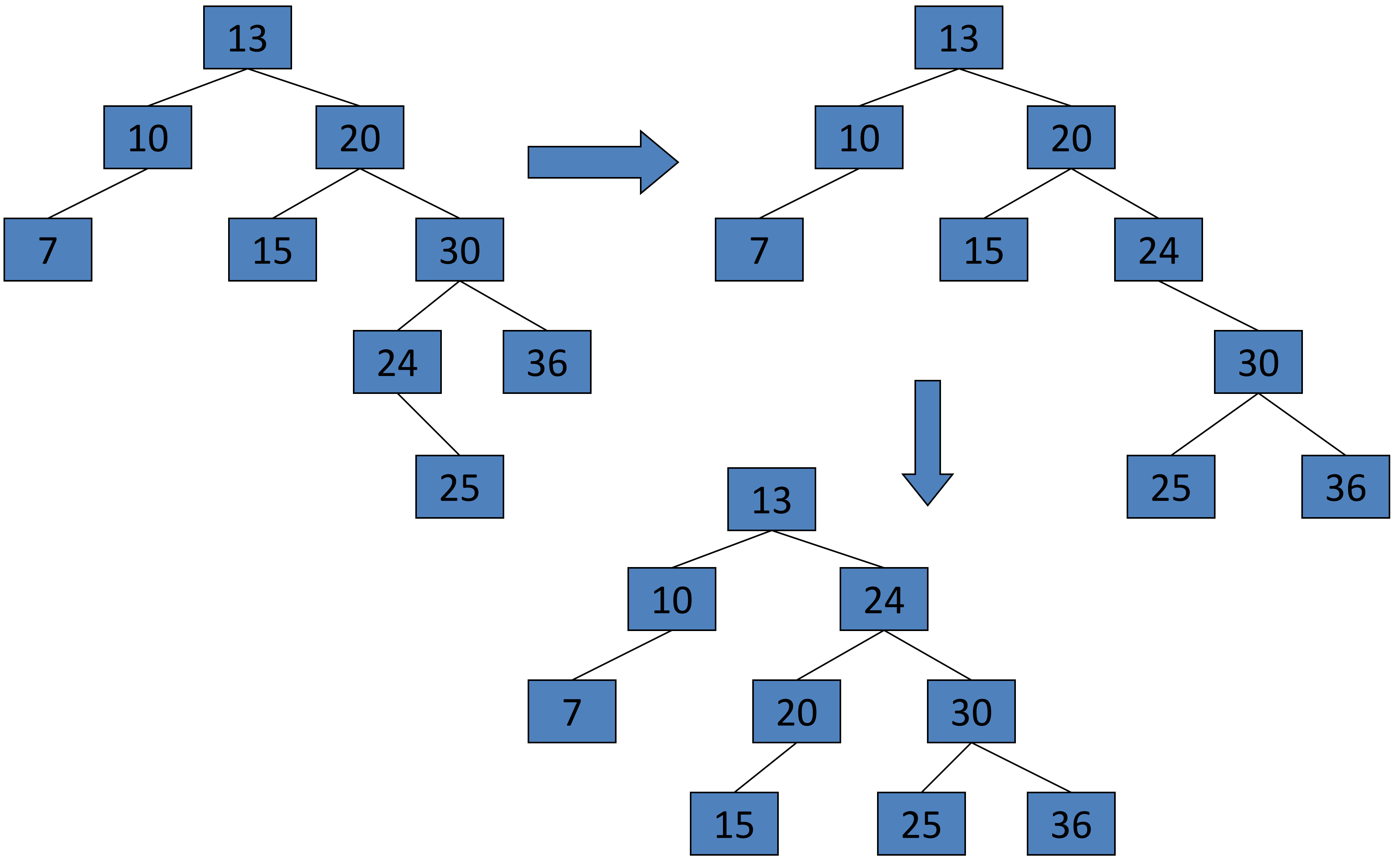
15, 20, 24, 10, 13, 7, 30, 36, 25



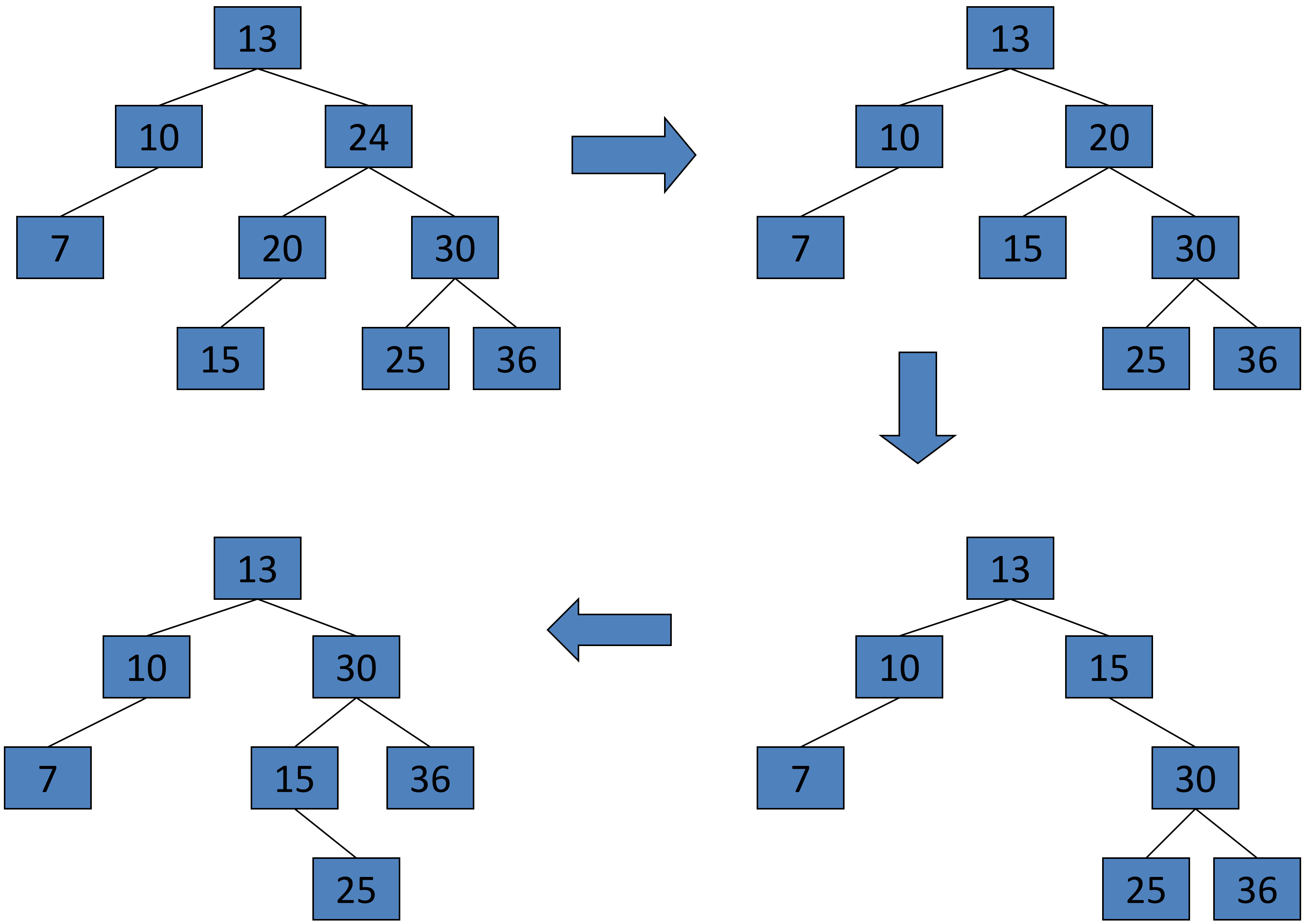
15, 20, 24, 10, 13, 7, 30, 36, 25



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



Remove 24 and 20 from the AVL tree.



```
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3
```

```c
struct Node *rightRotate(struct Node *y)
{
   struct Node *x = y->left;
   struct Node *T2 = x->right;

   // Perform rotation
   x->right = y;
   y->left = T2;

   // Update heights
   y->height = max(height(y->left), height(y->right))+1;
   x->height = max(height(x->left), height(x->right))+1;

   // Return new root
   return x;
}
```

```
struct Node *leftRotate(struct Node *x)
{
   struct Node *y = x->right;
   struct Node *T2 = y->left;

   // Perform rotation
   y->left = x;
   x->right = T2;

   //  Update heights
   x->height = max(height(x->left), height(x->right))+1;
   y->height = max(height(y->left), height(y->right))+1;

   // Return new root
   return y;
}
```

```
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3
```

```
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```

```
struct Node* insert(struct Node* node, int key)
{
   /* 1.  Perform the normal BST insertion */
   if (node == NULL)
      return(newNode(key));

   if (key < node->key)
      node->left  = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);
   else // Equal keys are not allowed in BST
      return node;

   /* 2. Update height of this ancestor node */
   node->height = 1 + max(height(node->left),
                          height(node->right));

   /* 3. Get the balance factor of this ancestor
         node to check whether this node became
         unbalanced */
   int balance = getBalance(node);

  // If this node becomes unbalanced, then
   // there are 4 cases

   // Left Left Case
   if (balance > 1 && key < node->left->key)
      return rightRotate(node);

   // Right Right Case
   if (balance < -1 && key > node->right->key)
      return leftRotate(node);

   // Left Right Case
   if (balance > 1 && key > node->left->key)
   {
      node->left =  leftRotate(node->left);
      return rightRotate(node);
   }

   // Right Left Case
   if (balance < -1 && key < node->right->key)
   {
      node->right = rightRotate(node->right);
      return leftRotate(node);
   }
   /* return the (unchanged) node pointer */
   return node;
}
```