Data Structure

Lecturer: Dr. Salwa Osama

AVL

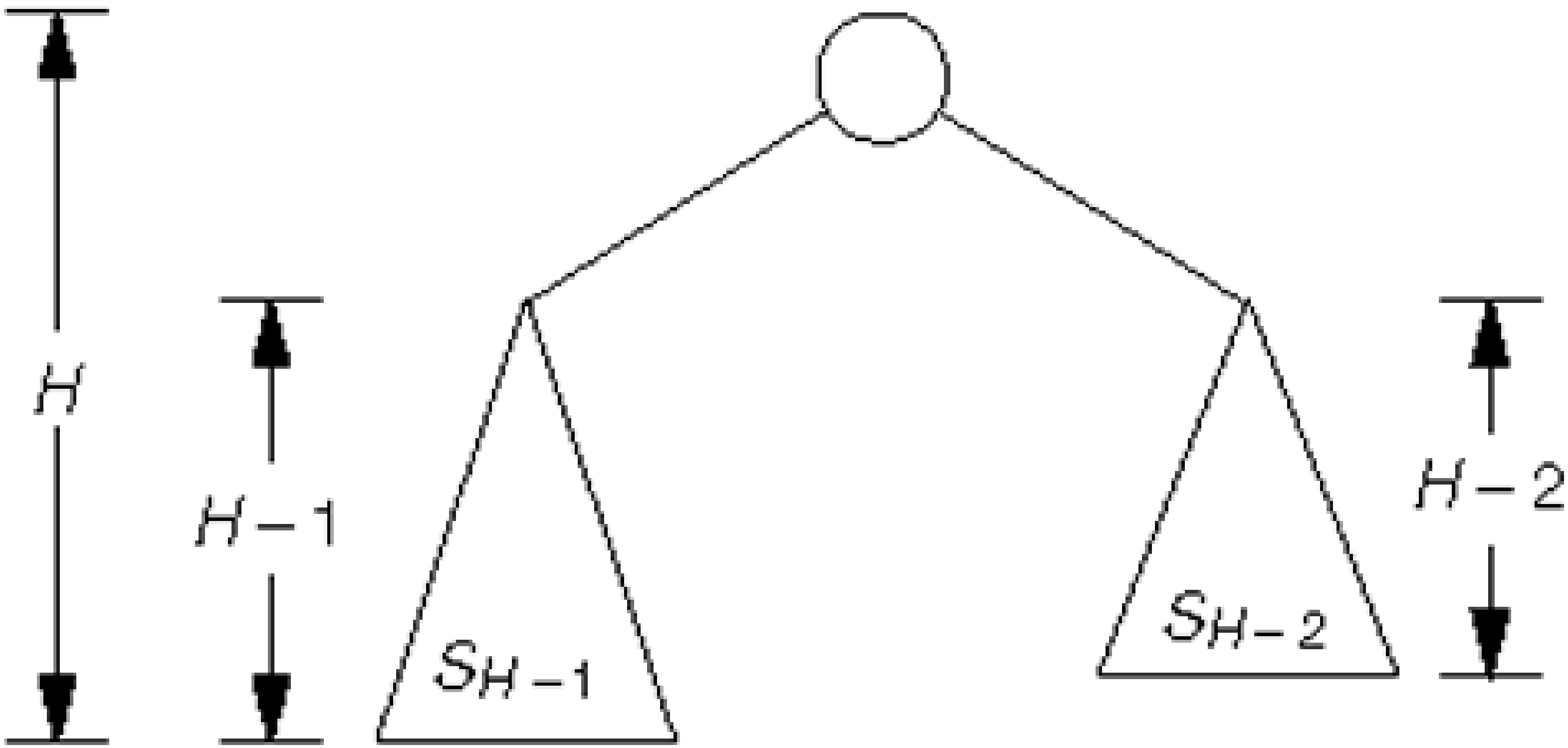
Balanced BST

- The disadvantage of a binary search tree is that its height can be as large as N-1 where N is the number of nodes in the tree.
- Thus, our goal is to keep the height of a binary search tree to be as small as we can.
- Such trees are called **balanced** binary search trees. Examples are ***<u>AVL tree</u>*** and red-black tree.

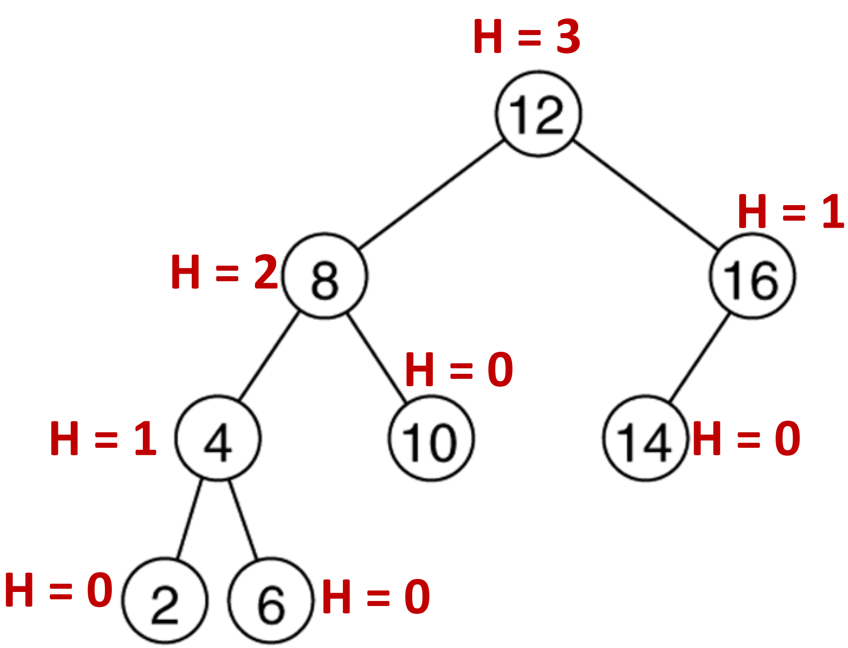
AVL Trees

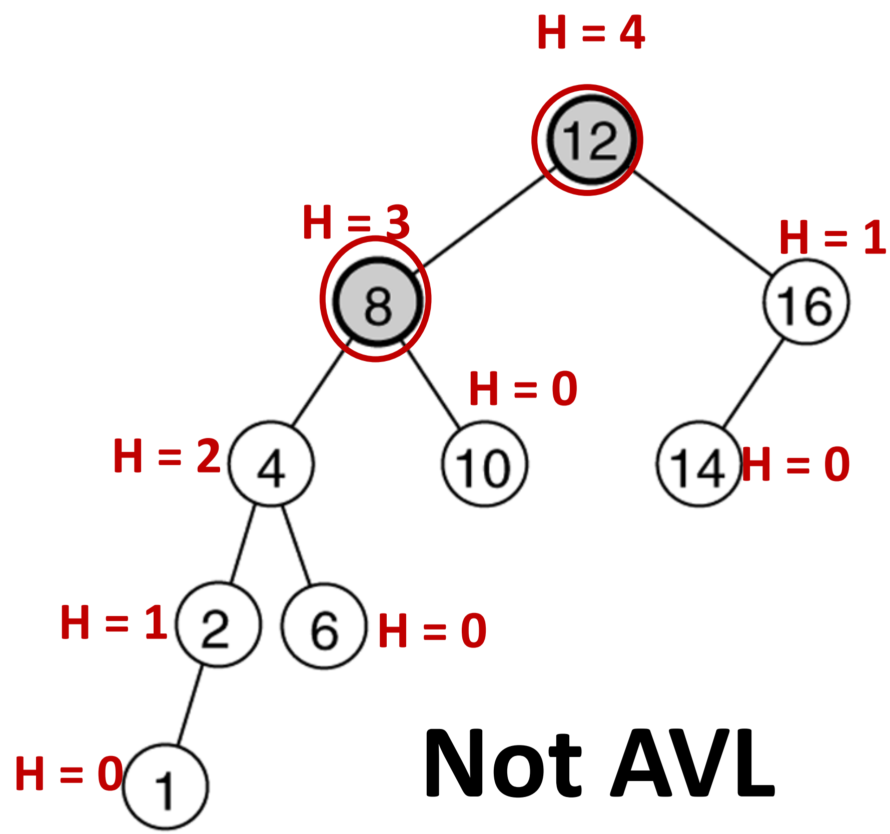
- An AVL tree is a binary search tree with a *balance* condition, which *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a ***height close to the minimum.***
- An AVL tree is a binary search tree such that, for any node in the tree, the height of the left and right subtrees can ***differ by at most 1***.
- An AVL tree could has ***balance factor*** calculated at every node, which is the difference between left subtree height and right subtree height

AVL Trees



AVL Trees





Not AVL

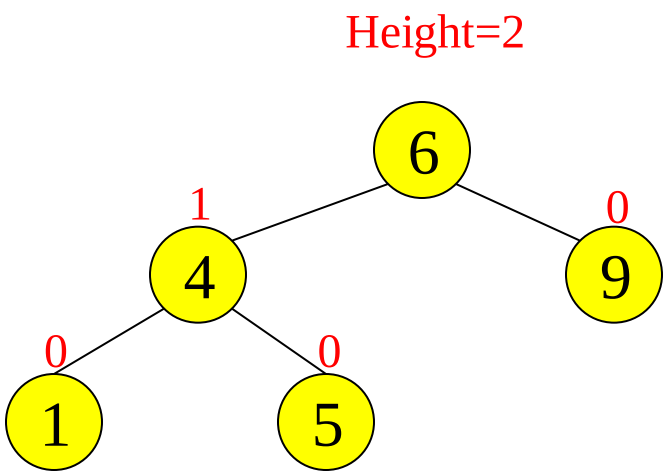
The height of a leaf is 0. The height of a NULL pointer is -1. The height of an internal node is the maximum height of its children plus 1

AVL Tree Implementation

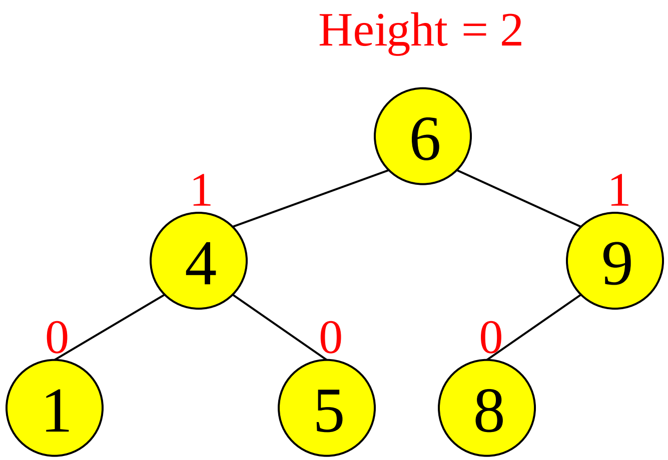
```
typedef struct {
    EntryType       info;
    NodeType        *right;
    NodeType        *left;
    int             height;
} AVLNodeType;

typedef  NodeType * TreeType
```

Node Heights

Tree A (AVL)

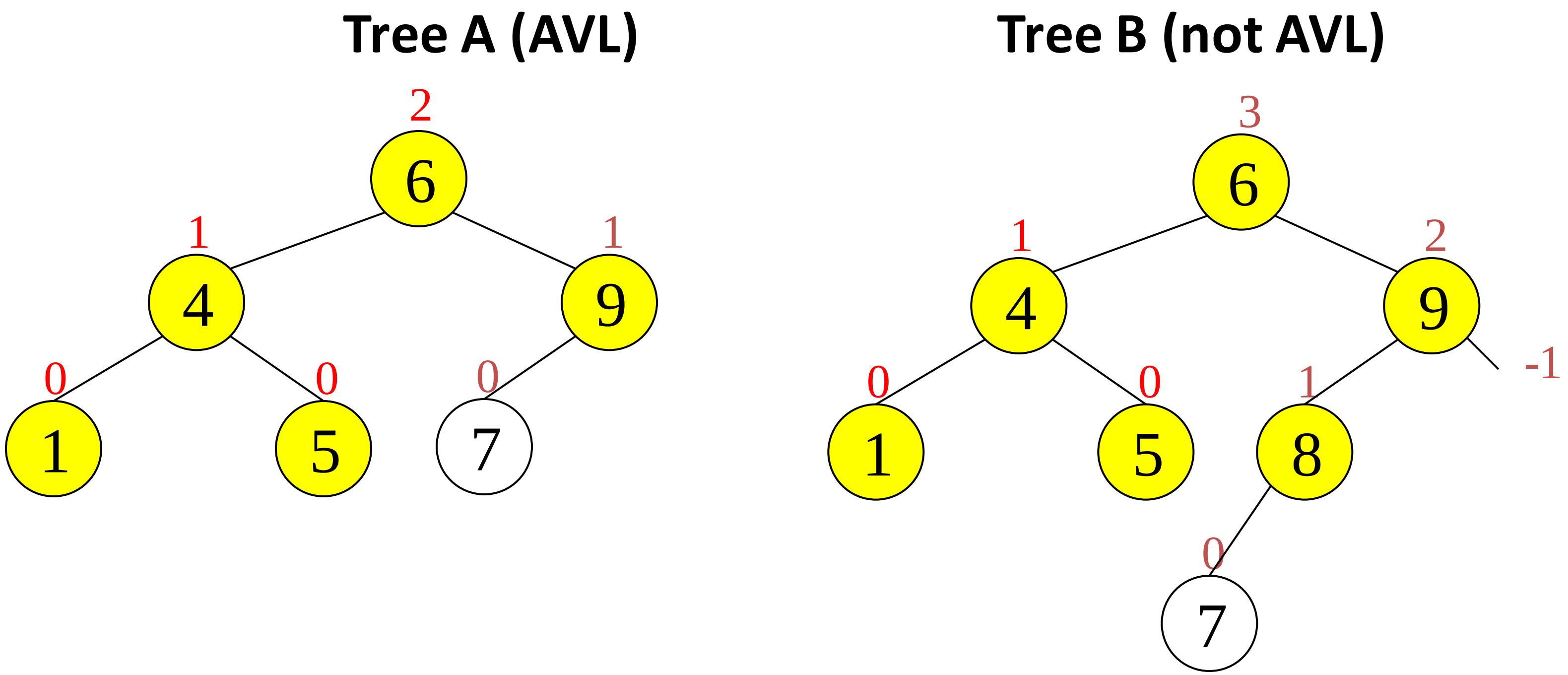


Tree B (AVL)



Now: Insert 7

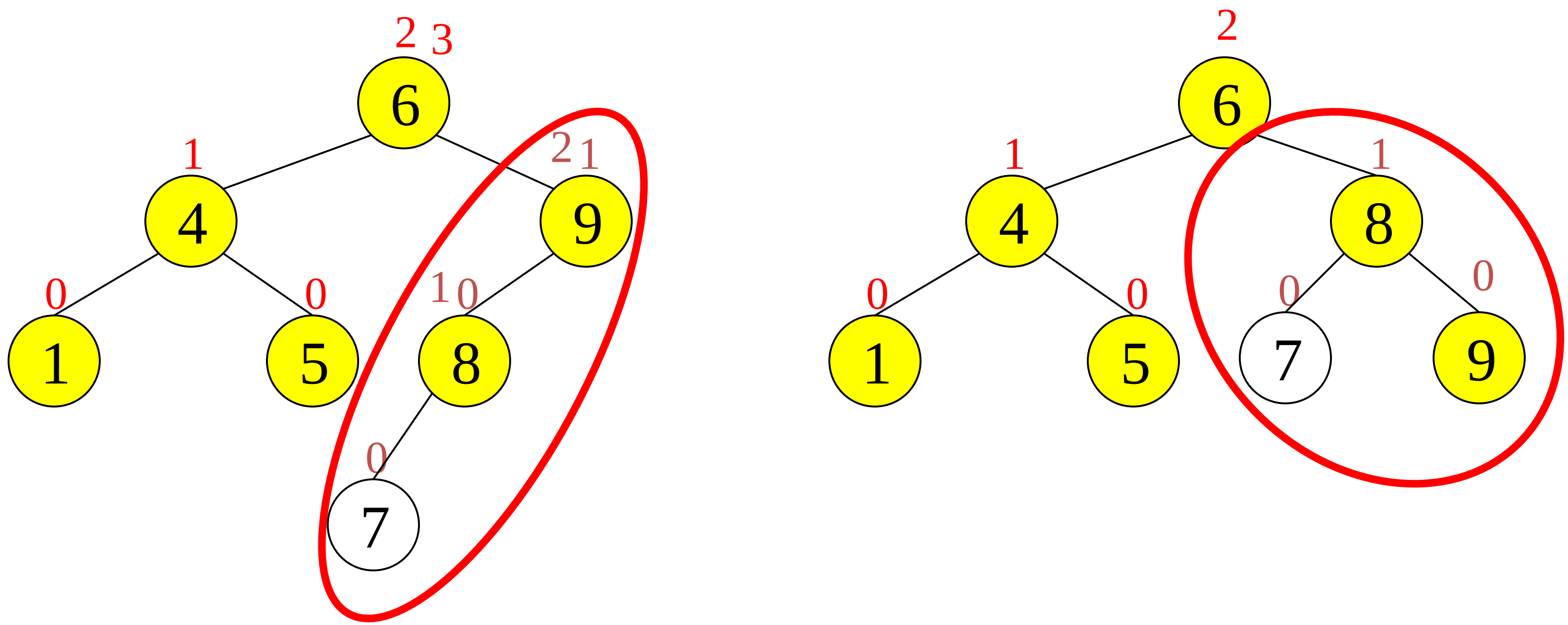
Node Heights after Insert 7



Insert and Deletion in AVL Trees

- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) and right(x) differ by exactly 2.
- If a balance factor (the difference h_{left}-h_{right}) become 2 or –2, adjust tree by *rotation* around the node

Insert in AVL Trees



Insert in AVL Tree

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, update its height.
- check if heights of left(x) and right(x) differ by at most 1. If yes, proceed to parent(x). If not, restructure by doing either a single rotation or a double rotation.
- For insertion, once we perform a rotation at a node x, we won't need to perform any rotation at any ancestor of x.

Insertions in AVL Trees

Let the node that needs rotation be x.

There are 4 cases:

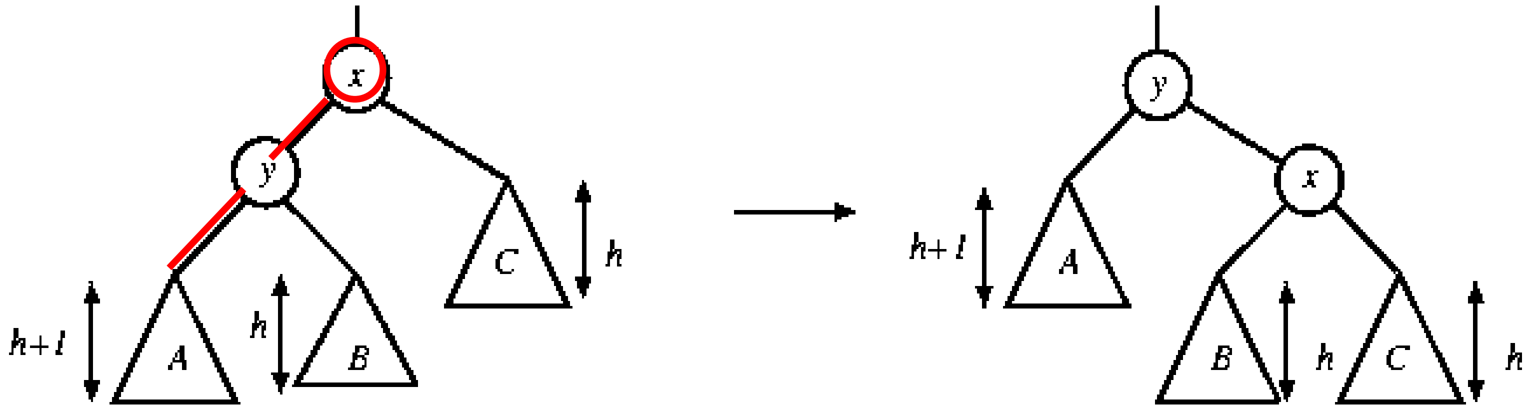
Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of X.
2. Insertion into right subtree of right child of X.

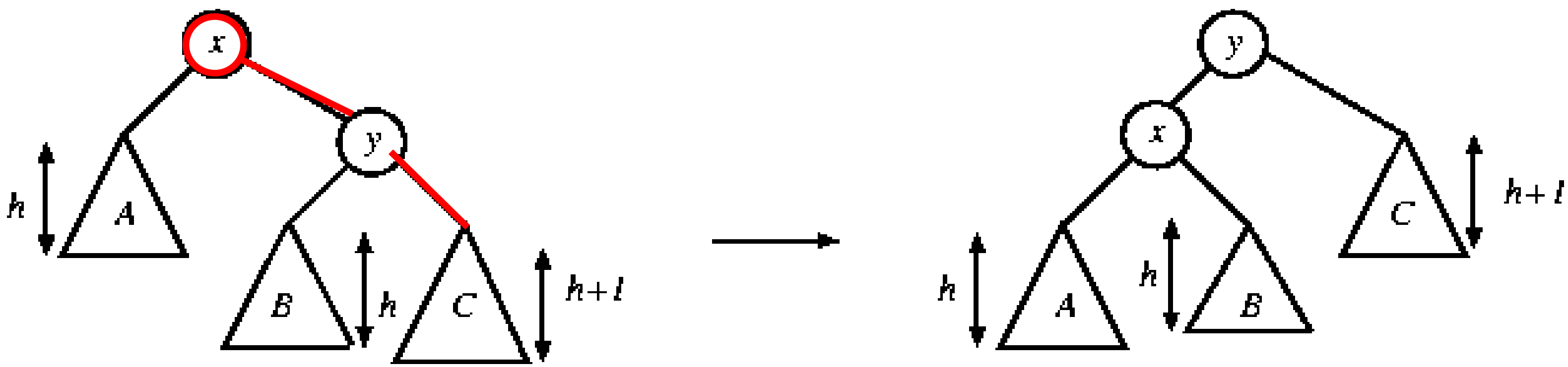
Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of X.
4. Insertion into left subtree of right child of X.

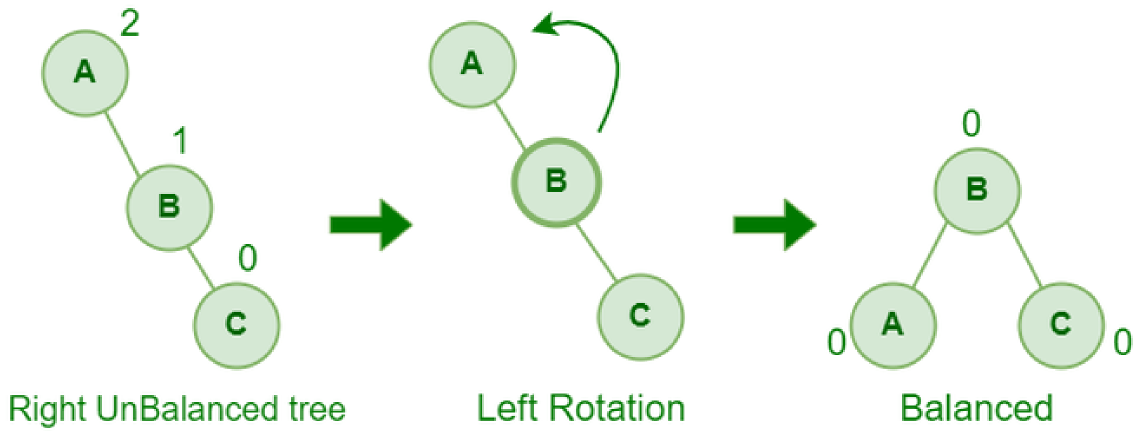
Single rotation



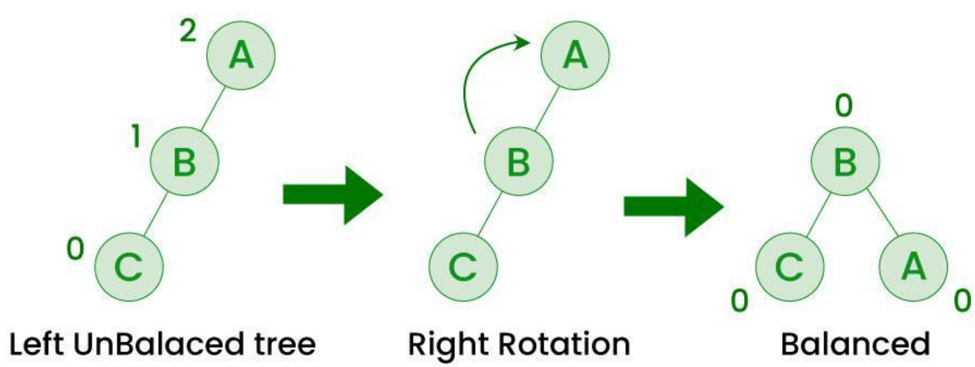
Rotate with left child



Rotate with right child

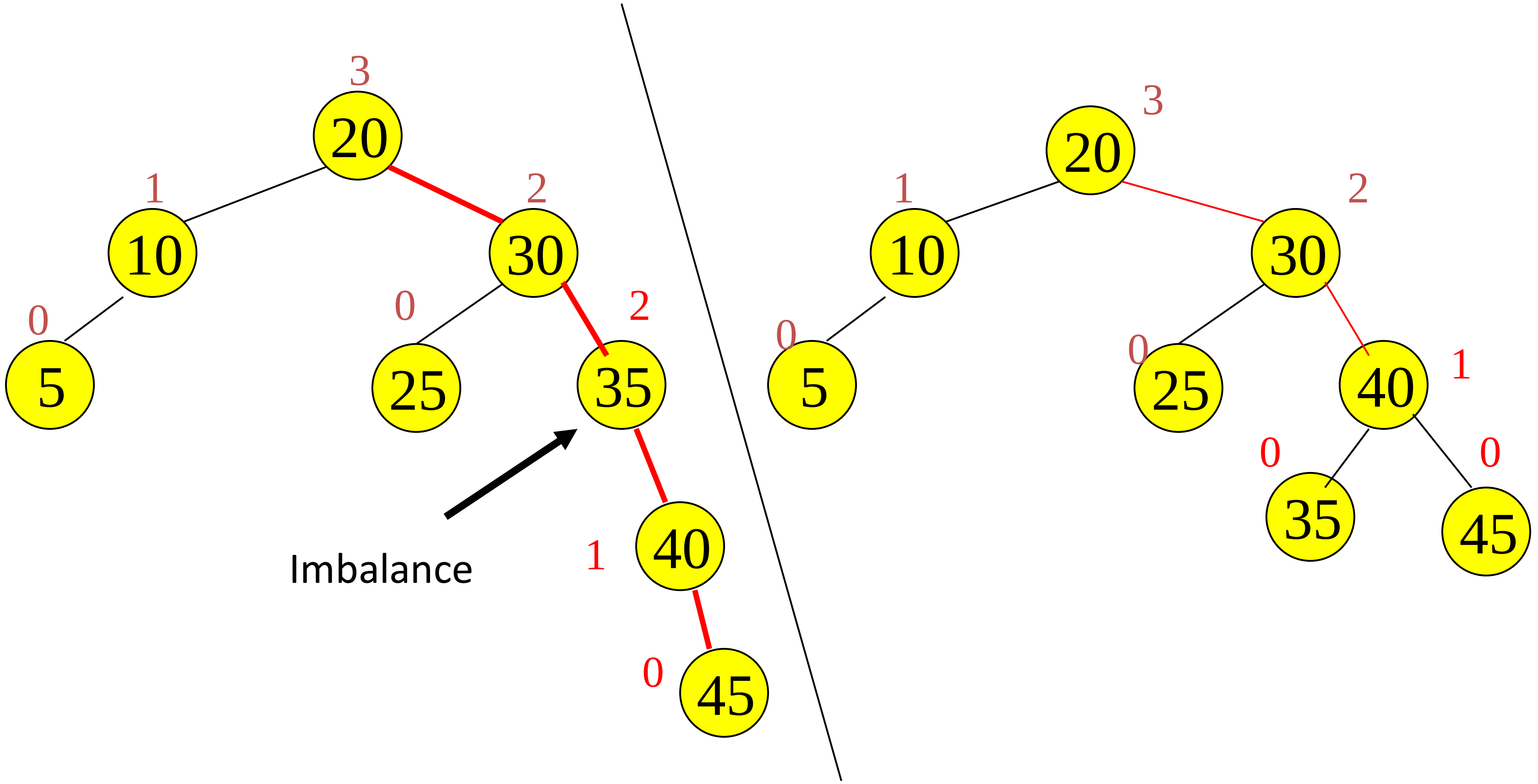


Left Rotation

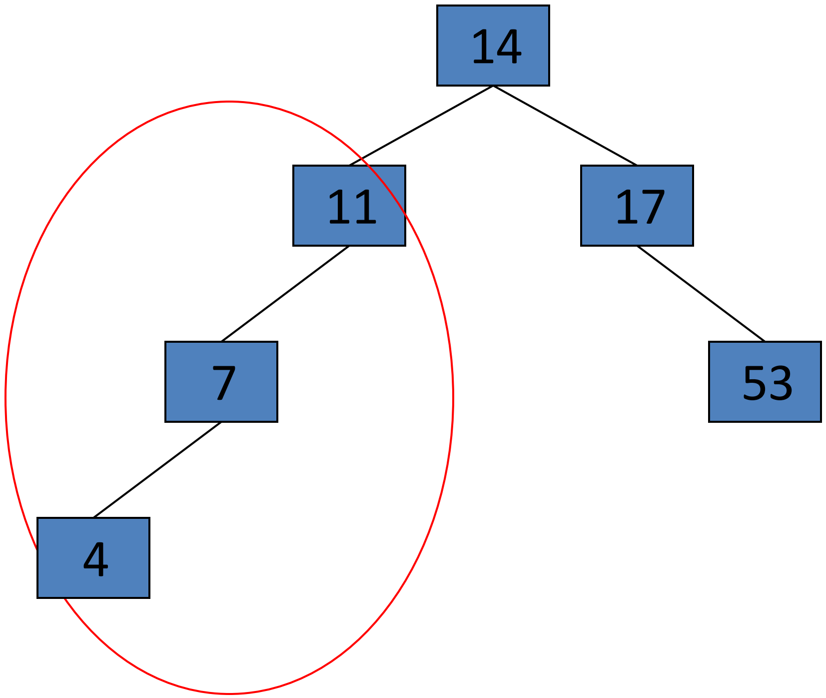


Right Rotation

Single rotation

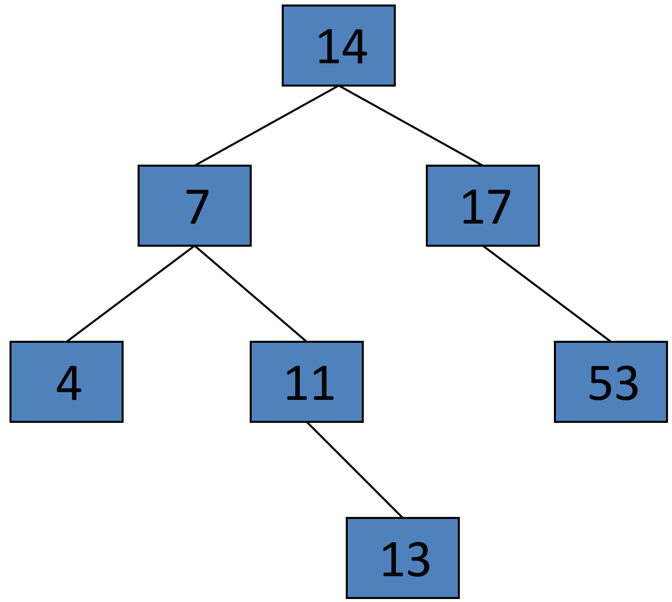


Insert in AVL Trees



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

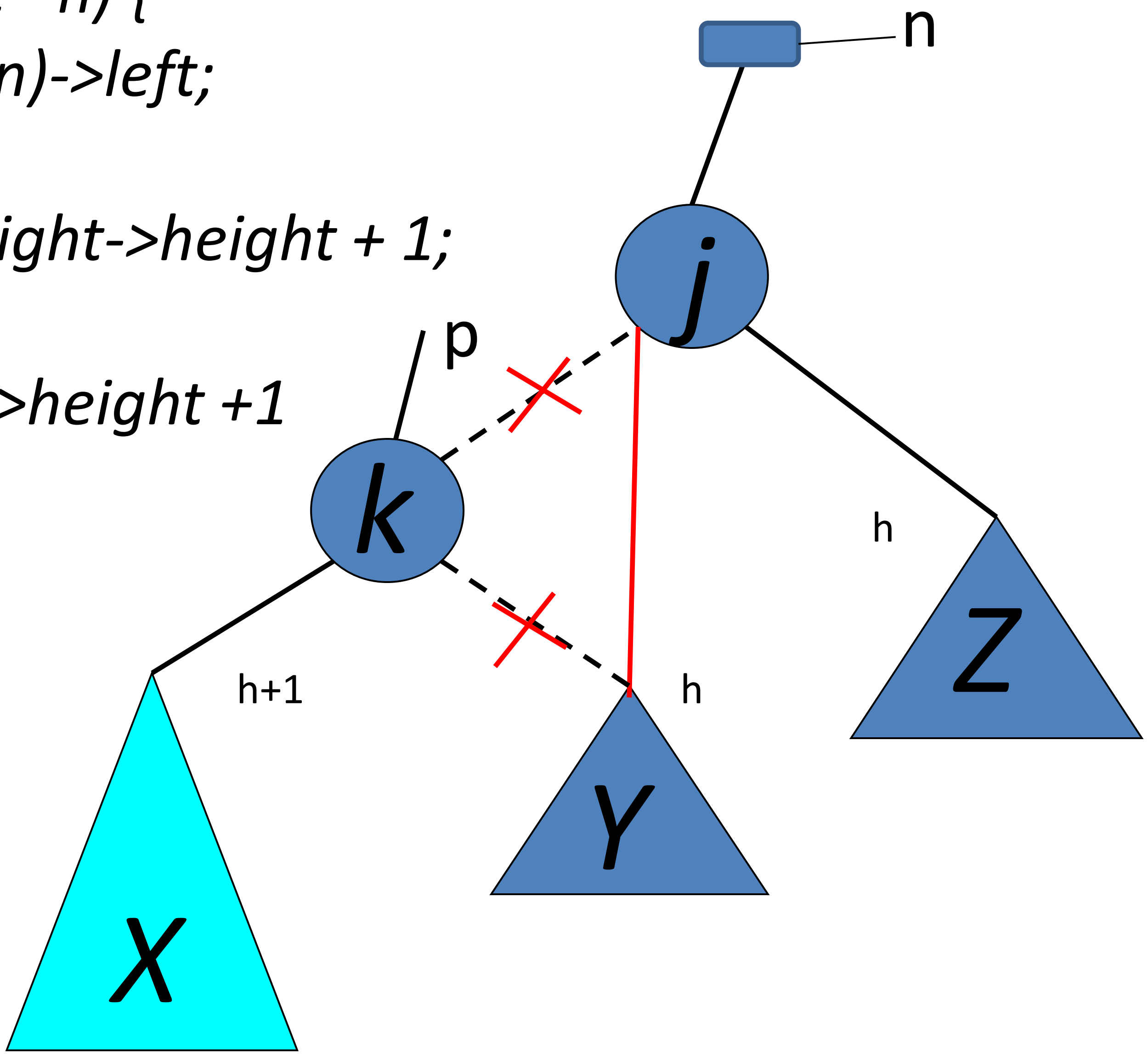
Insert in AVL Trees



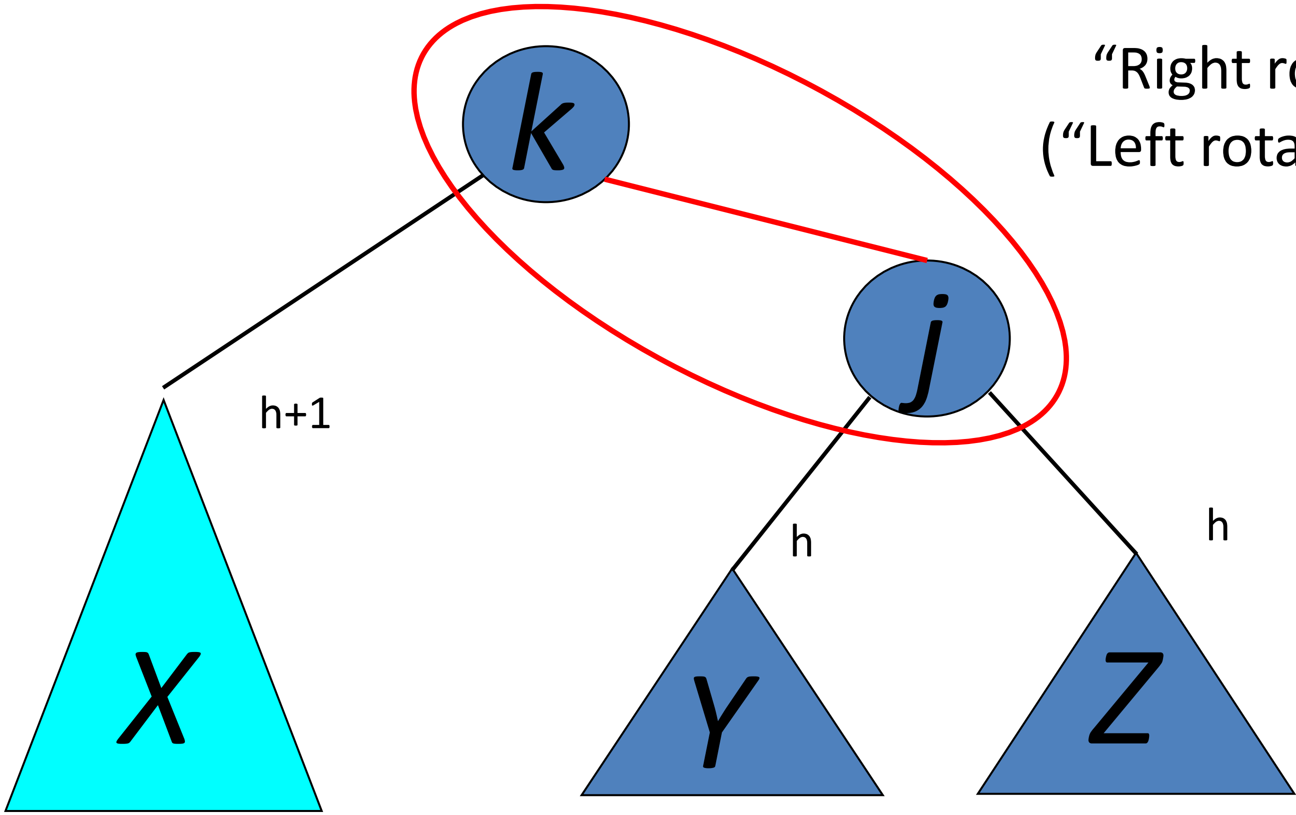
Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree

Single rotation

```
RotateRight(AVLTree *n) {
AVLTreeNode   p= (*n)->left;
(*n)->left= p->right;
(*n)-> height =  p->right->height + 1;
P->right= *n;
p->height = p->left ->height +1
*n = p
}
```

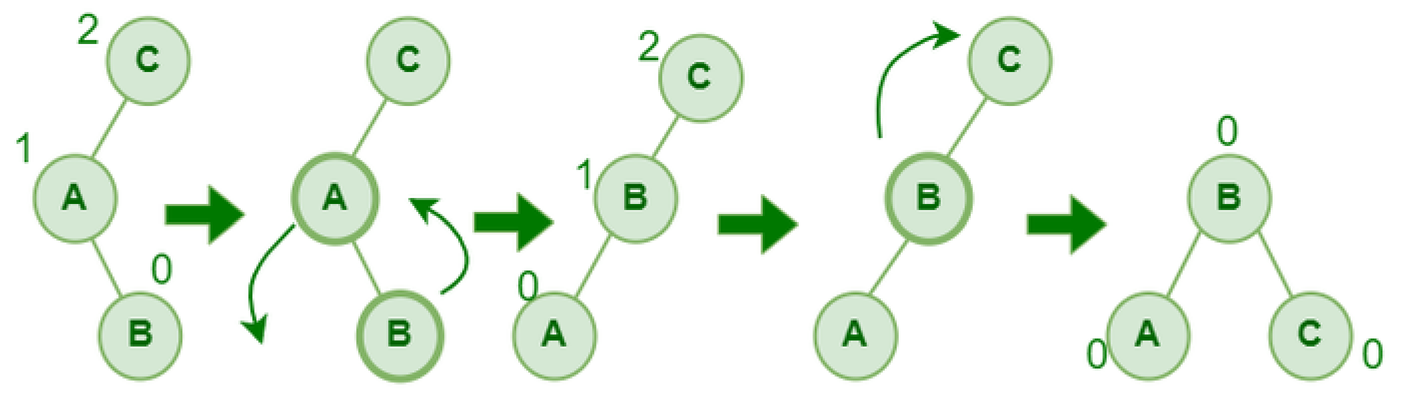


Single rotation

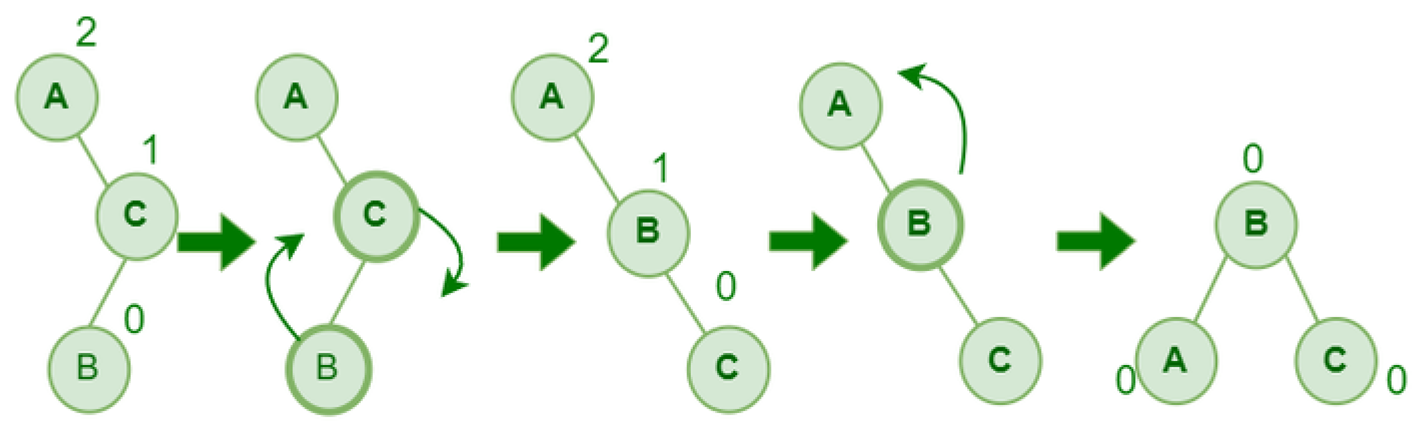


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored

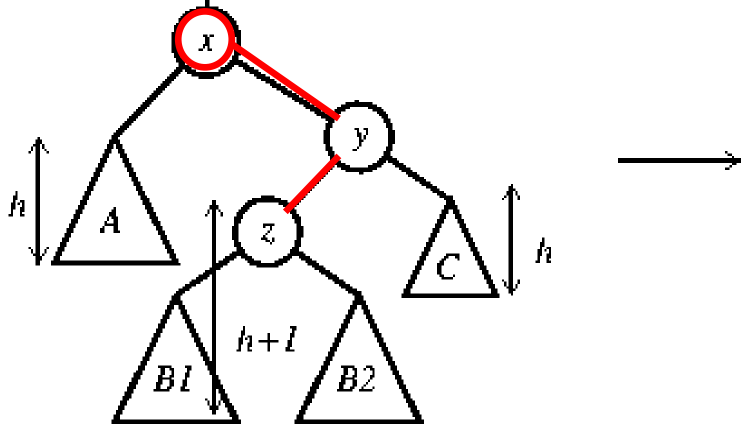


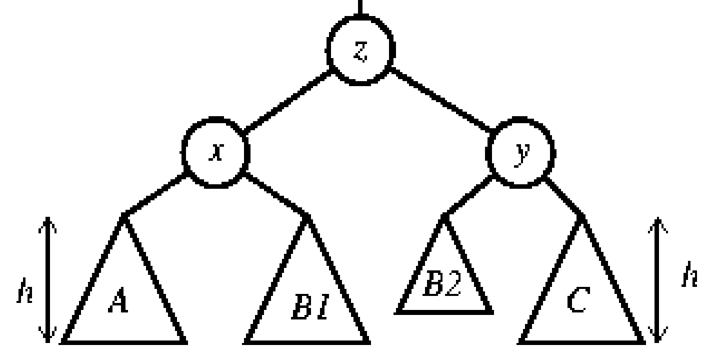
Left-Right Rotation



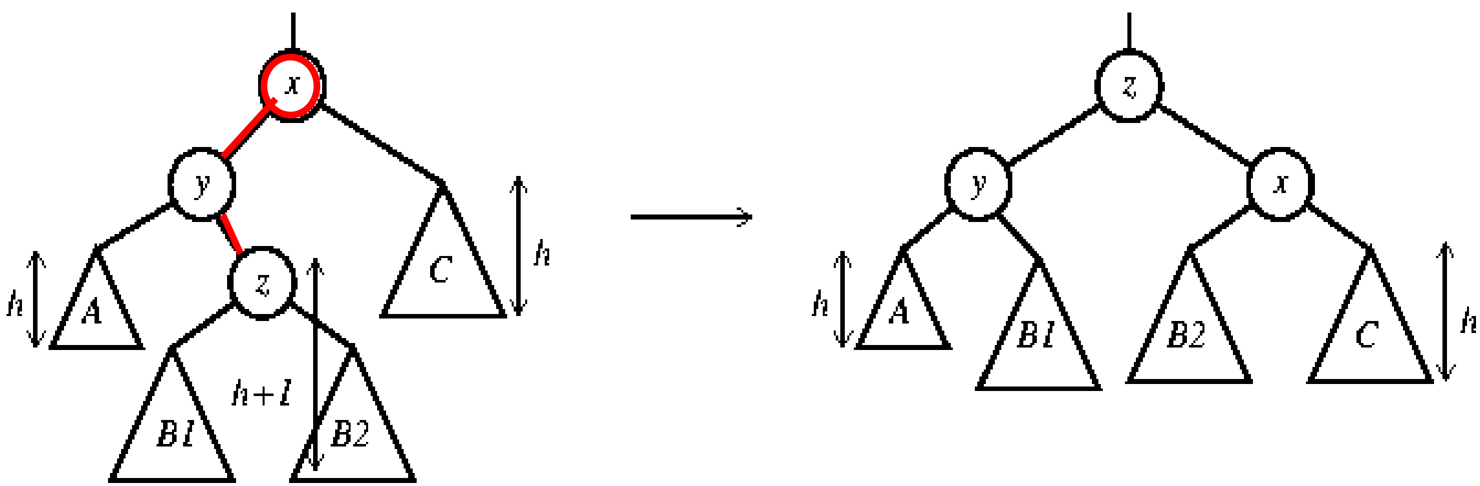
Right-Left Rotation

Double rotation





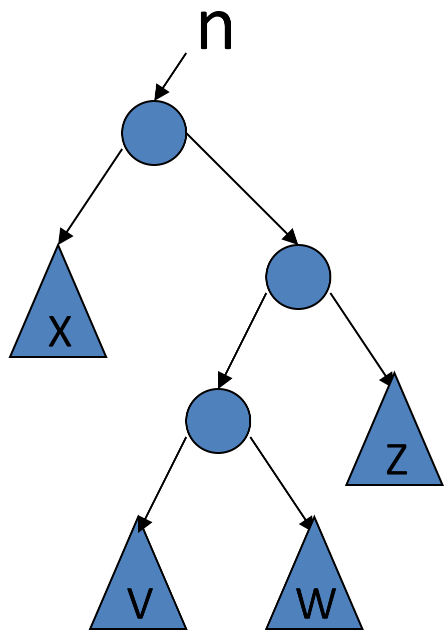
Double rotate with right child



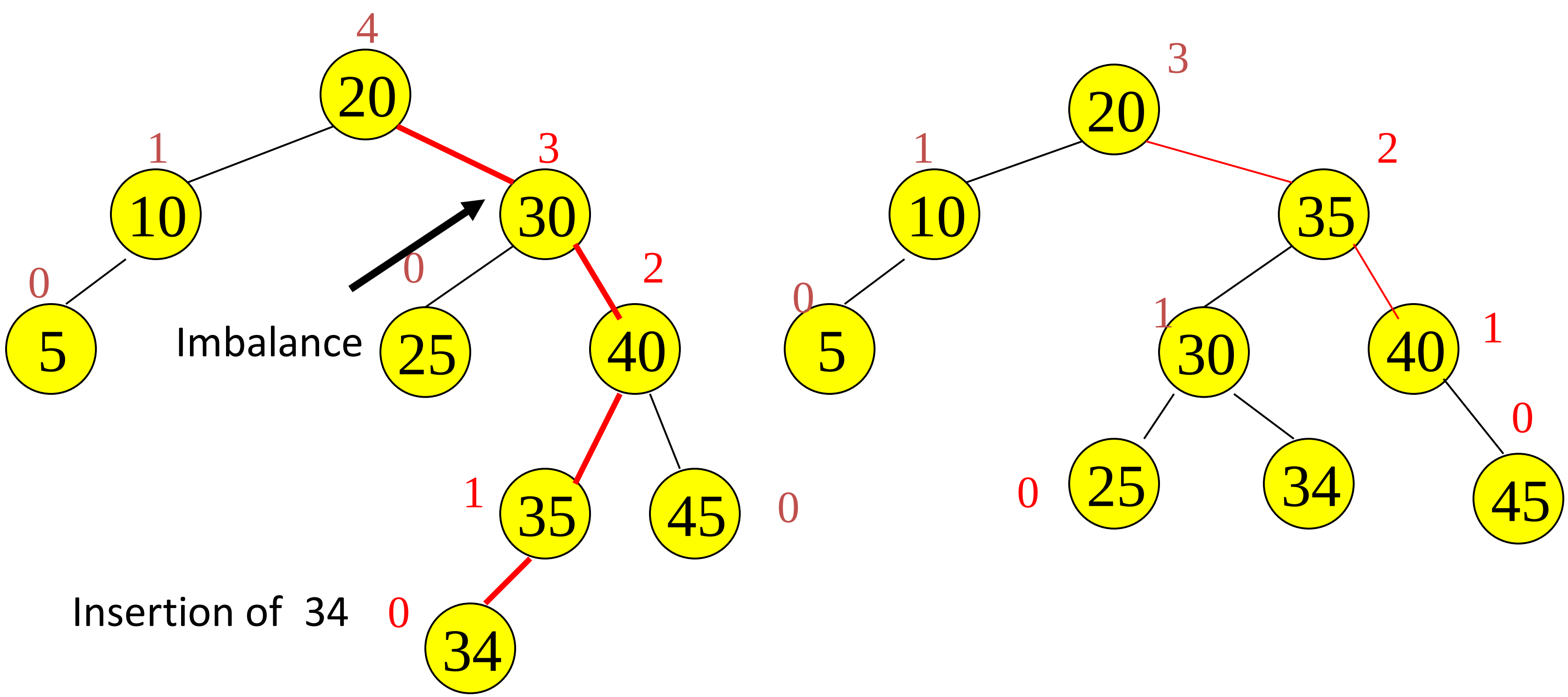
Double rotate with left child

Double Rotation

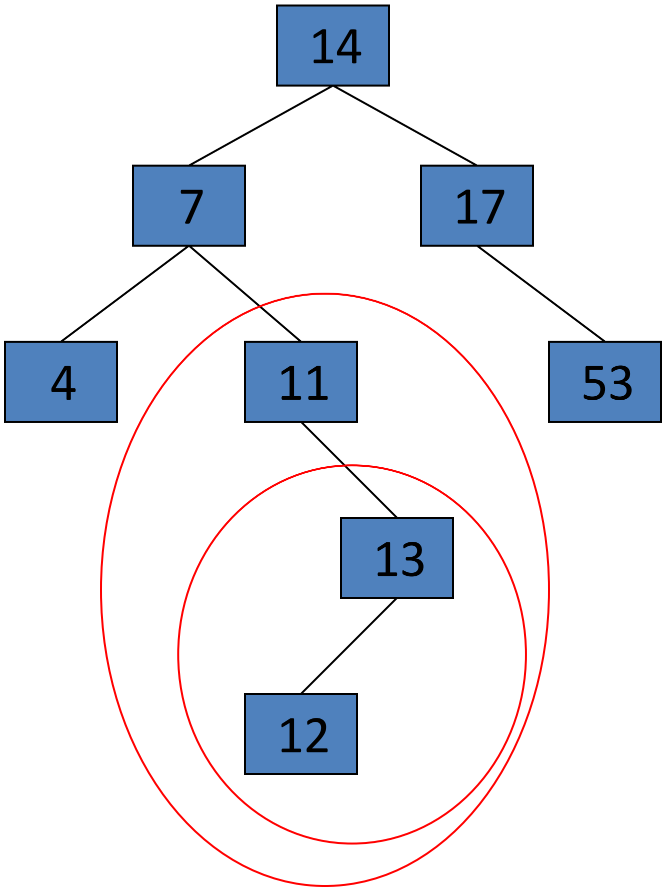
```
DoubleRotateFromRight(AVLTree *n) {
RotateRight(&(*n)->right);
RotateLeft(n);
}
```



Double rotation

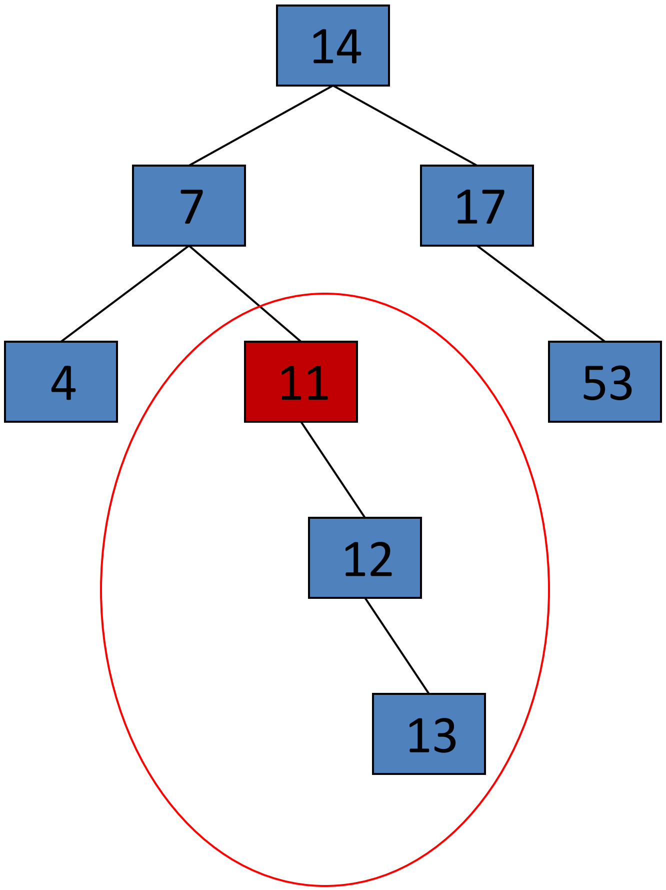


Insert in AVL Trees



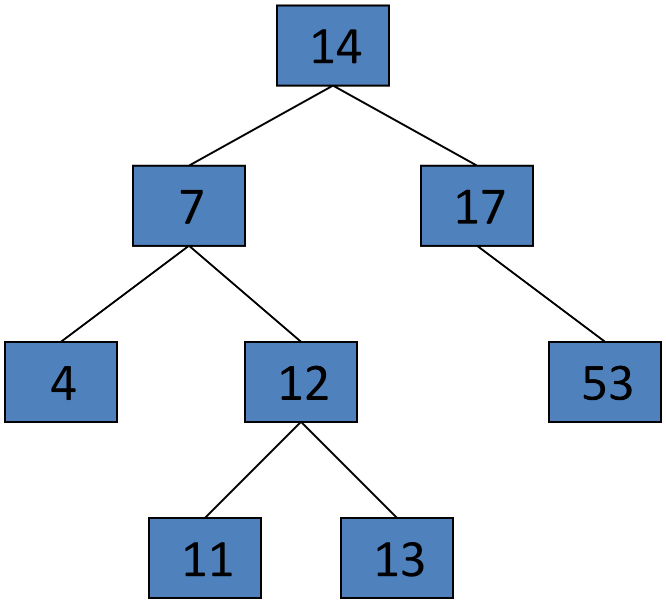
Now insert 12

Insert in AVL Trees

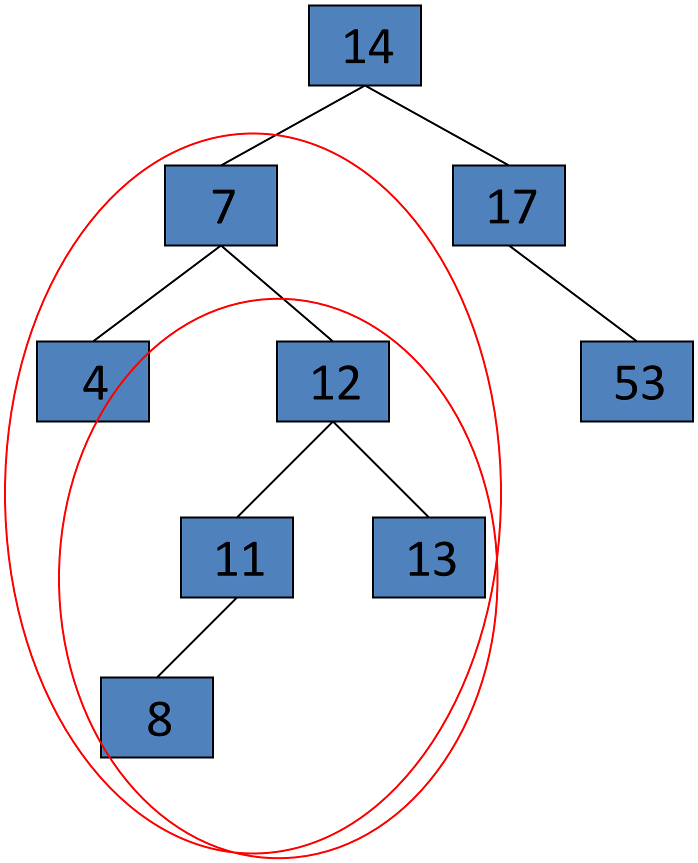


Now insert 12

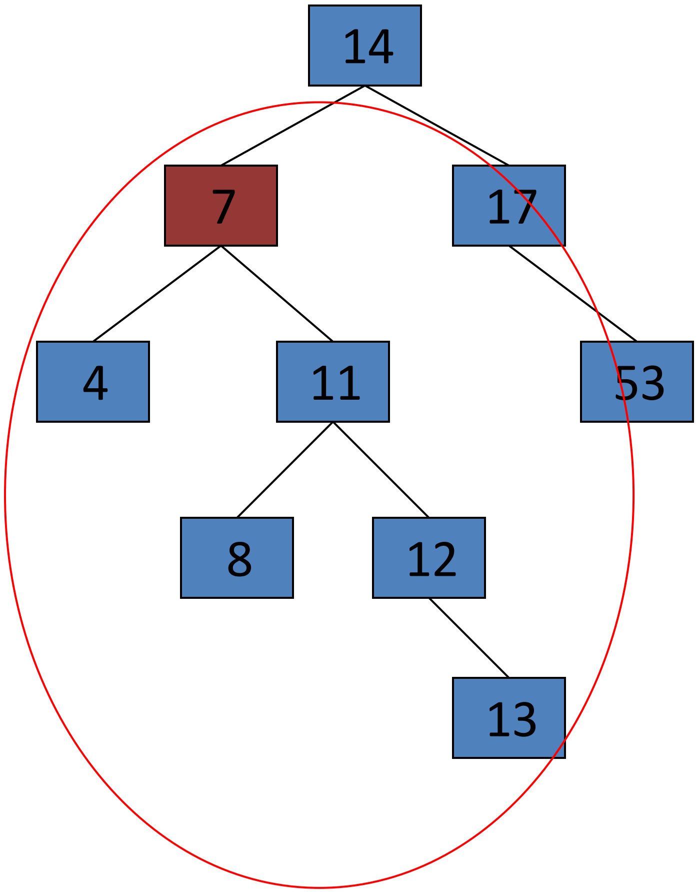
Insert in AVL Trees



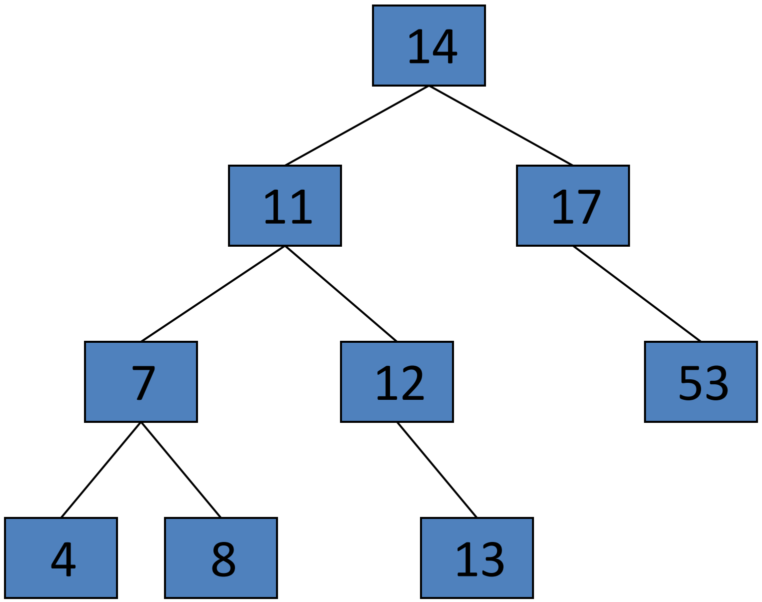
Insert in AVL Trees



Now insert 8



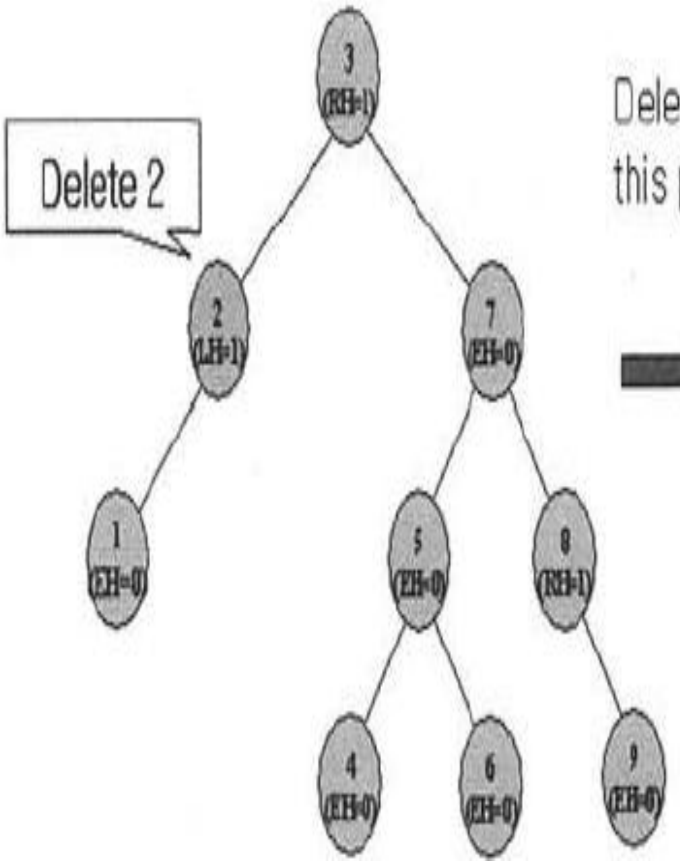
Now insert 8


14
11
17
7
12
53
4
8
13

AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

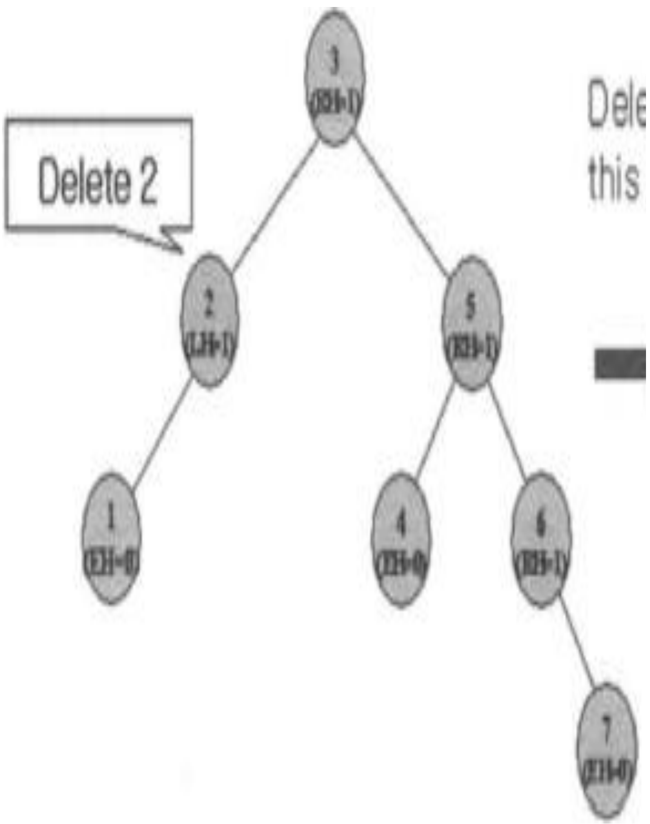
AVL Tree Deletion



LH =RH

Single left rotation is required

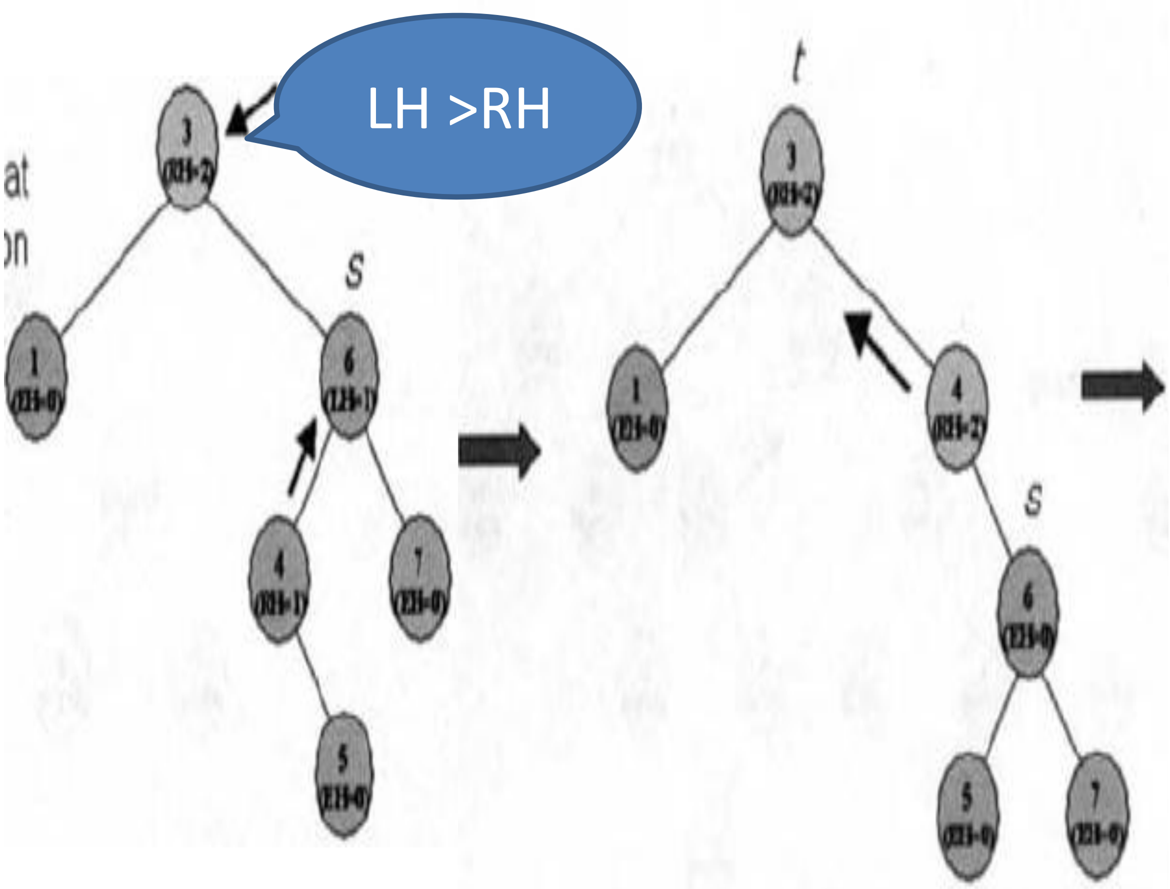
AVL Tree Deletion



LH < RH

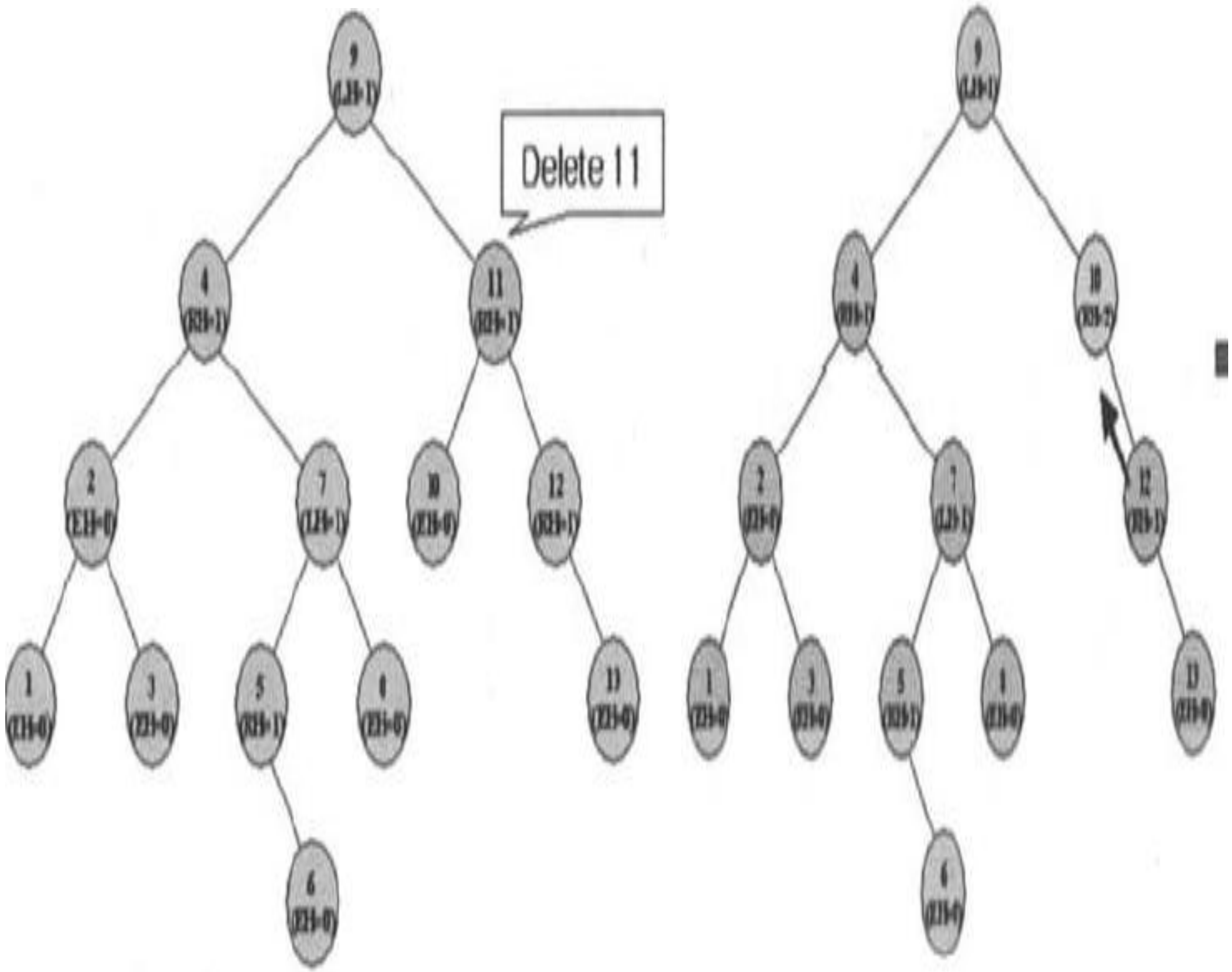
Single left rotation is required

AVL Tree Deletion

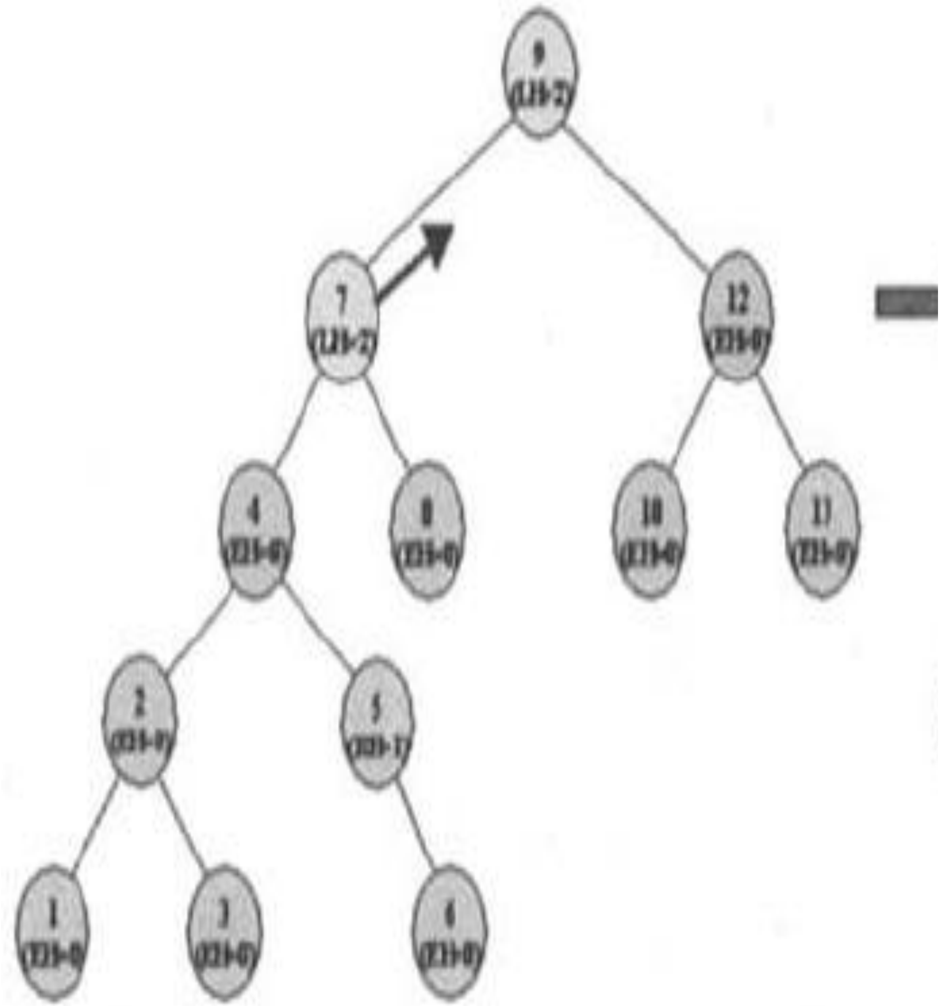


Double rotation is required

AVL Tree Deletion



AVL Tree Deletion



Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

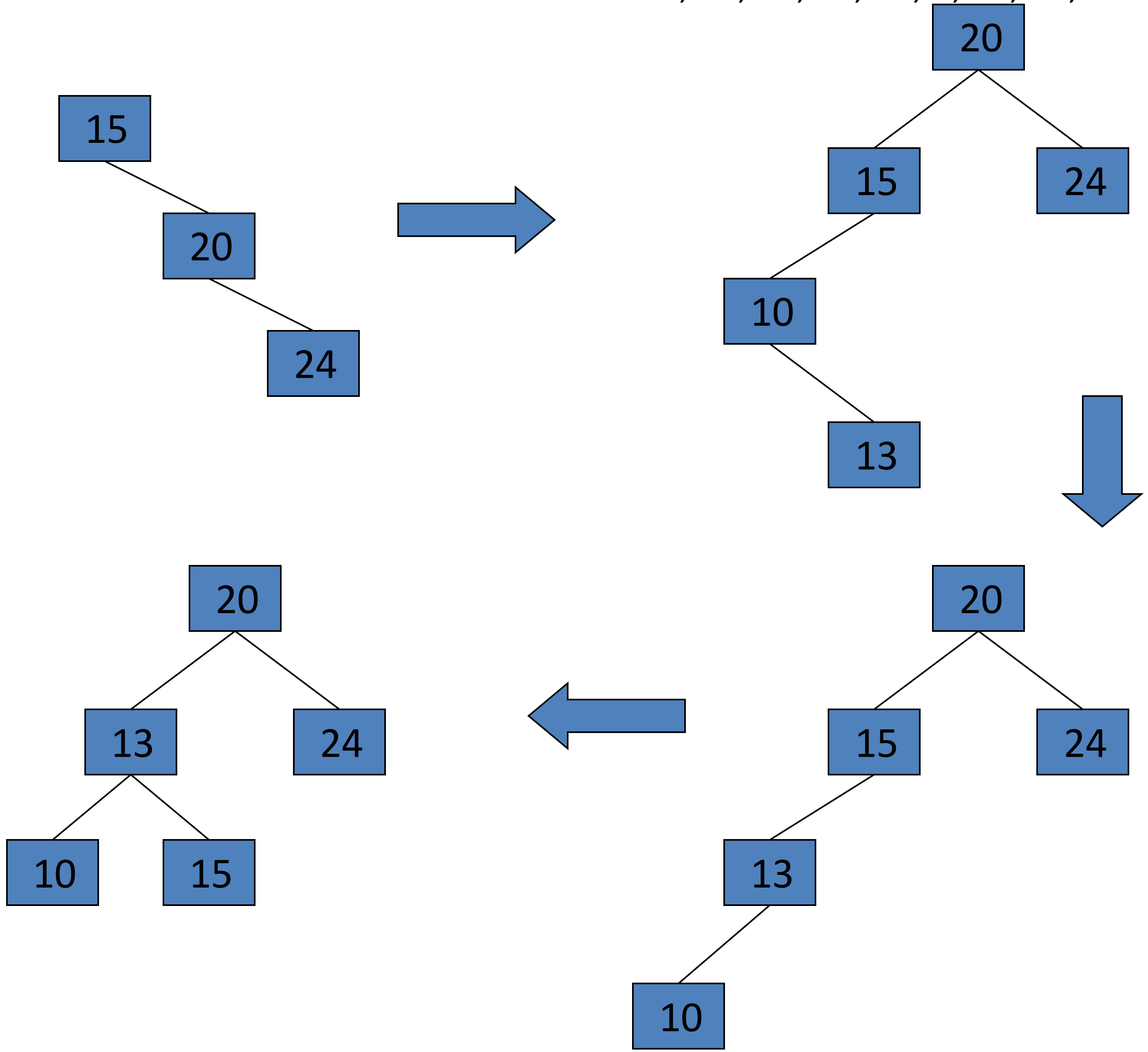
1. Difficult to program & debug; more space for balance factor.
2. Rebalancing costs time.

Exercises

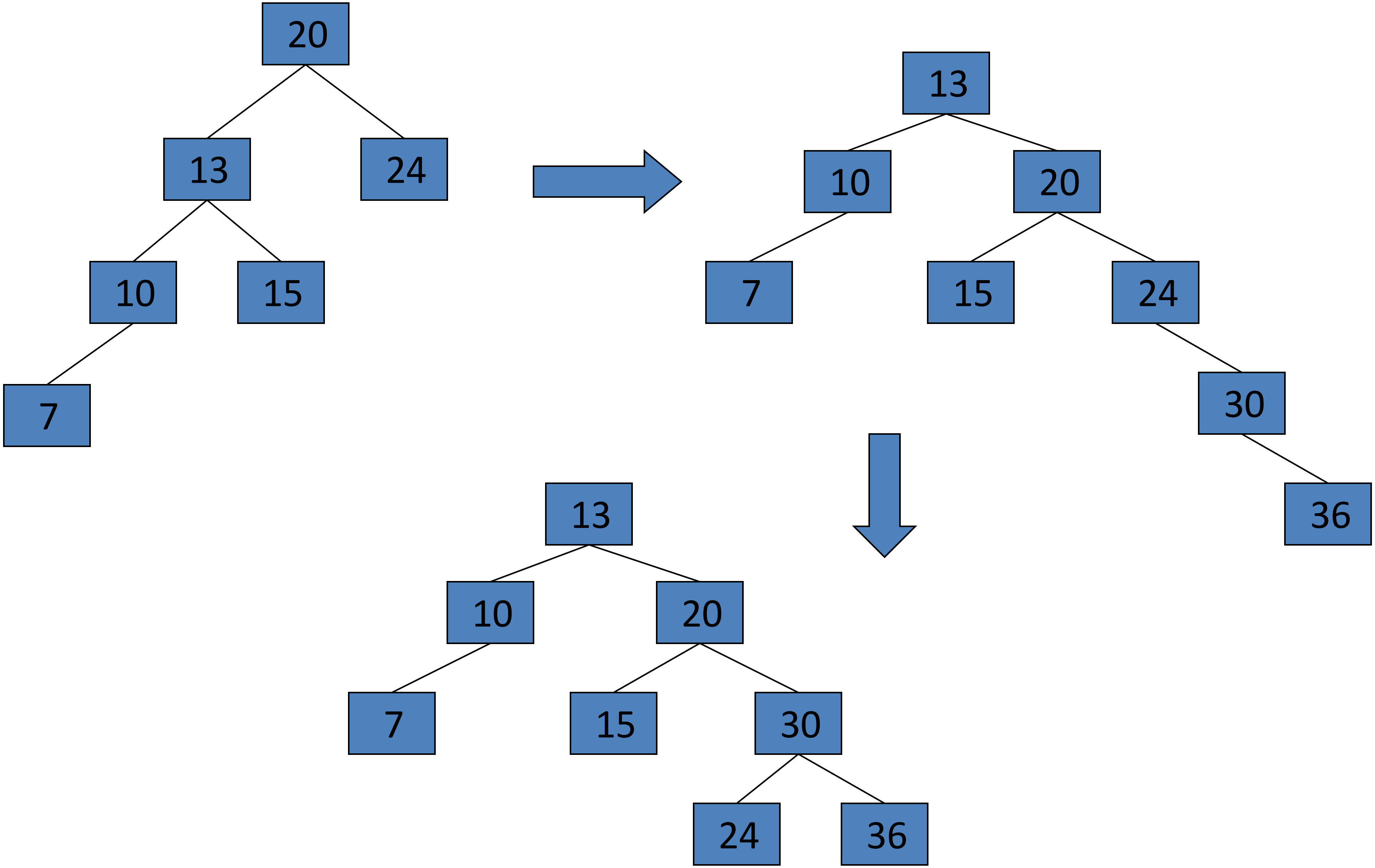
- Build an AVL tree with the following values:

  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