Binary Tree Deletion

Insert in BST

```
Insert(T : reference tree pointer, x : element) : integer {
if T = null then
  T := new tree; T.data := x; return 1;//the links to
                                       //children are null
case
  T.data = x : return 0; //Duplicate do nothing
  T.data > x : return Insert(T.left, x);
  T.data < x : return Insert(T.right, x);
endcase
}
```

Find the node x that contains the key k:

1) If x has no children, delete x.

2) If x has one child, delete x and link x's parent to x's child

3) If x has two children,

   -find x's successor z [the leftmost node in the rightsubtree of x]

   -replace x's contents with z's contents, and

   -delete z.

   (Note: z does not have a left child, but may have a right child)

   [since z has at most one child, so we use case (1) or (2) to delete z]

AVL Tree

Binary Search Tree - Best Time

All BST operations are O(d), where d is tree depth
minimum d is for a binary tree with N nodes
What is the best case tree?
What is the worst case tree?
So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

Worst case running time is O(N)

What happens when you Insert elements in ascending order?

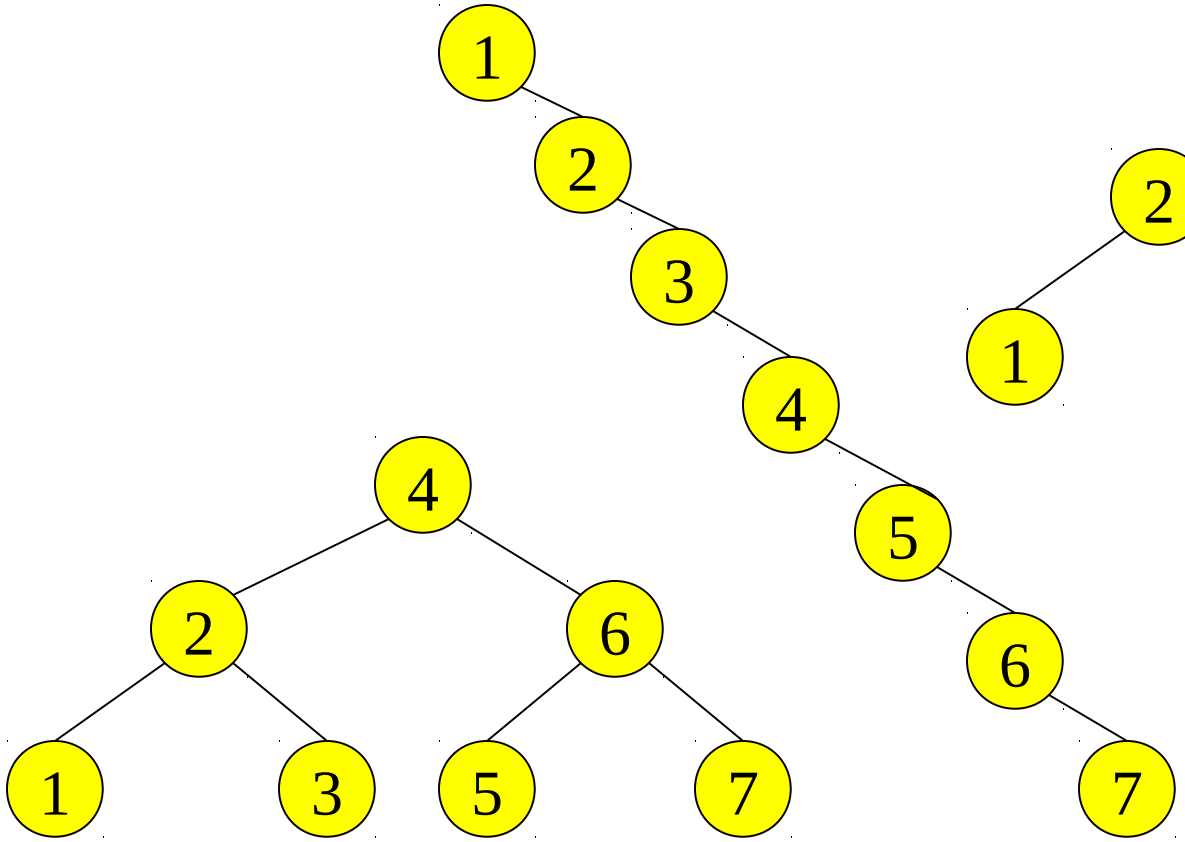
- Insert: 2, 4, 6, 8, 10, 12 into an empty BST

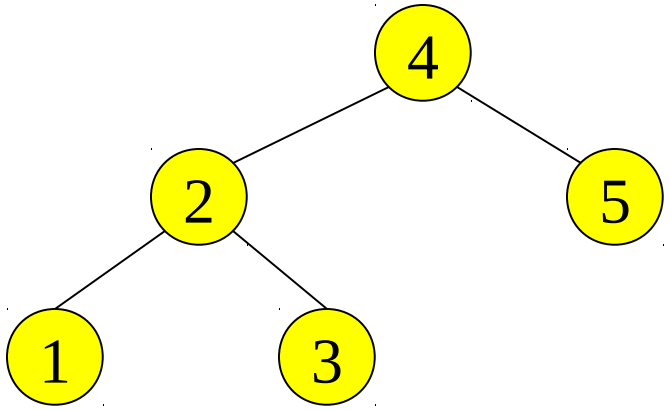
Problem: Lack of “balance”:

- compare depths of left and right subtree

Unbalanced degenerate tree

Balanced and unbalanced BST





Is this “balanced”?

Balancing Binary Search Trees

Many algorithms exist for keeping binary search trees balanced

Adelson-Velskii and Landis (AVL) trees (height-balanced trees)

Splay trees and other self-adjusting trees

B-trees and other multiway search trees

AVL - Good but not Perfect Balance

AVL trees are height-balanced binary search trees

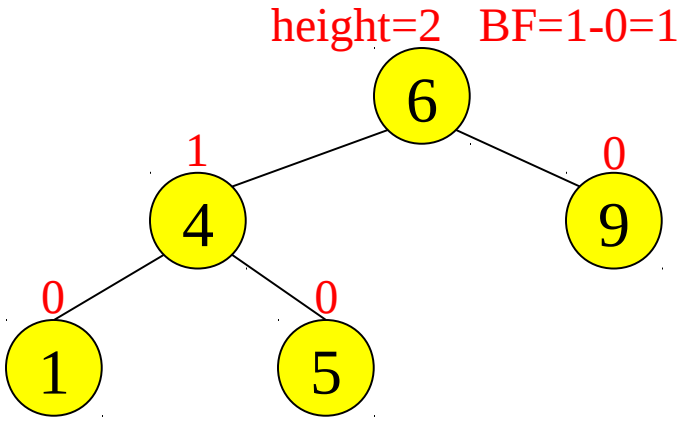
Balance factor of a node =height(left subtree) - height(right subtree)

An AVL tree has balance factor calculated at every node

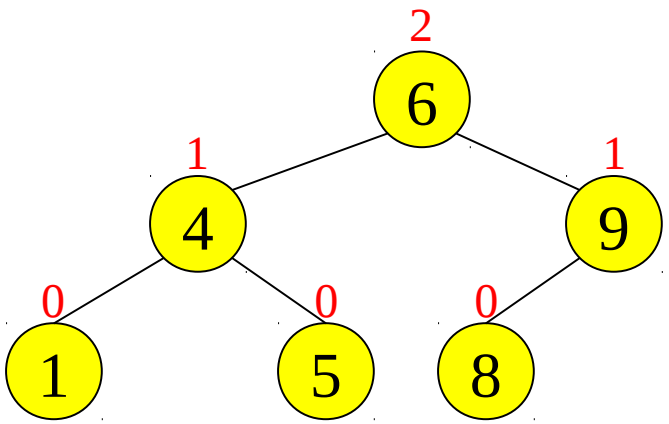
For every node, heights of left and right subtree can differ by no more than 1
Store current heights in each node

Node Heights

Tree A (AVL)



Tree B (AVL)



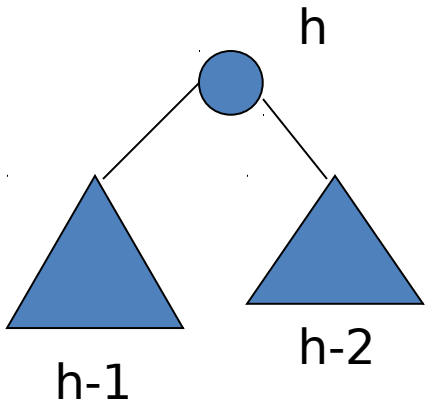
height of node = h
balance factor = hleft-hright
height = abs(-1)

Height of an AVL Tree

N(h) = minimum number of nodes in an AVL tree of height h.
Basis

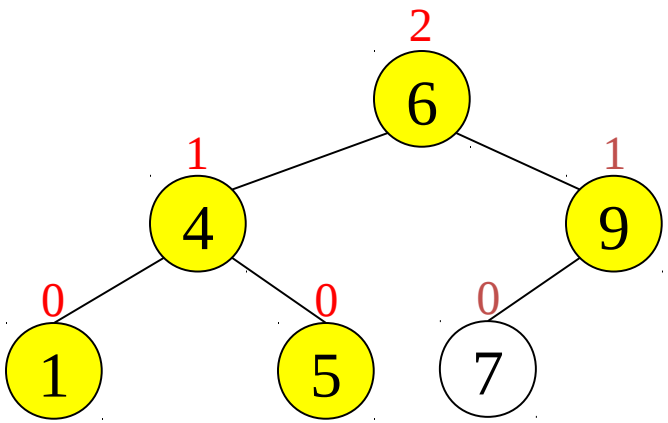
N(0) = 1, N(1) = 2

N(h) = N(h-1) + N(h-2) + 1

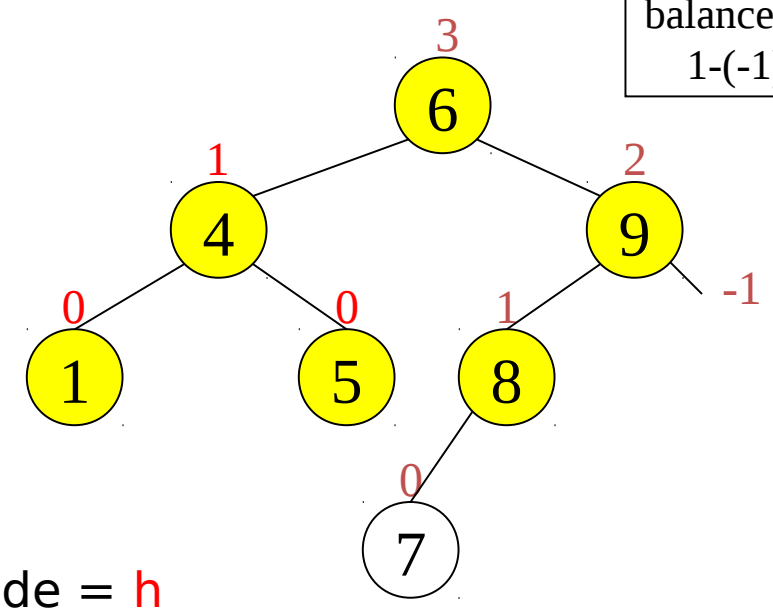


Node Heights after Insert 7

Tree A (AVL)



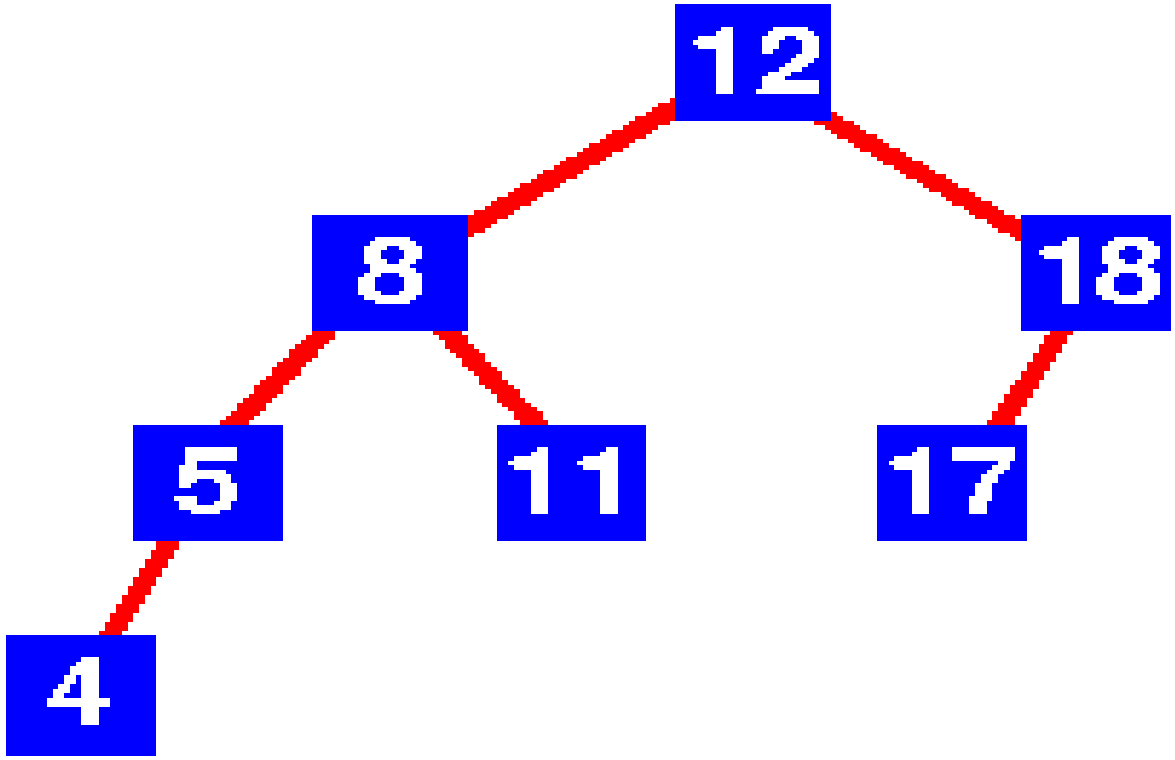
Tree B (not AVL)



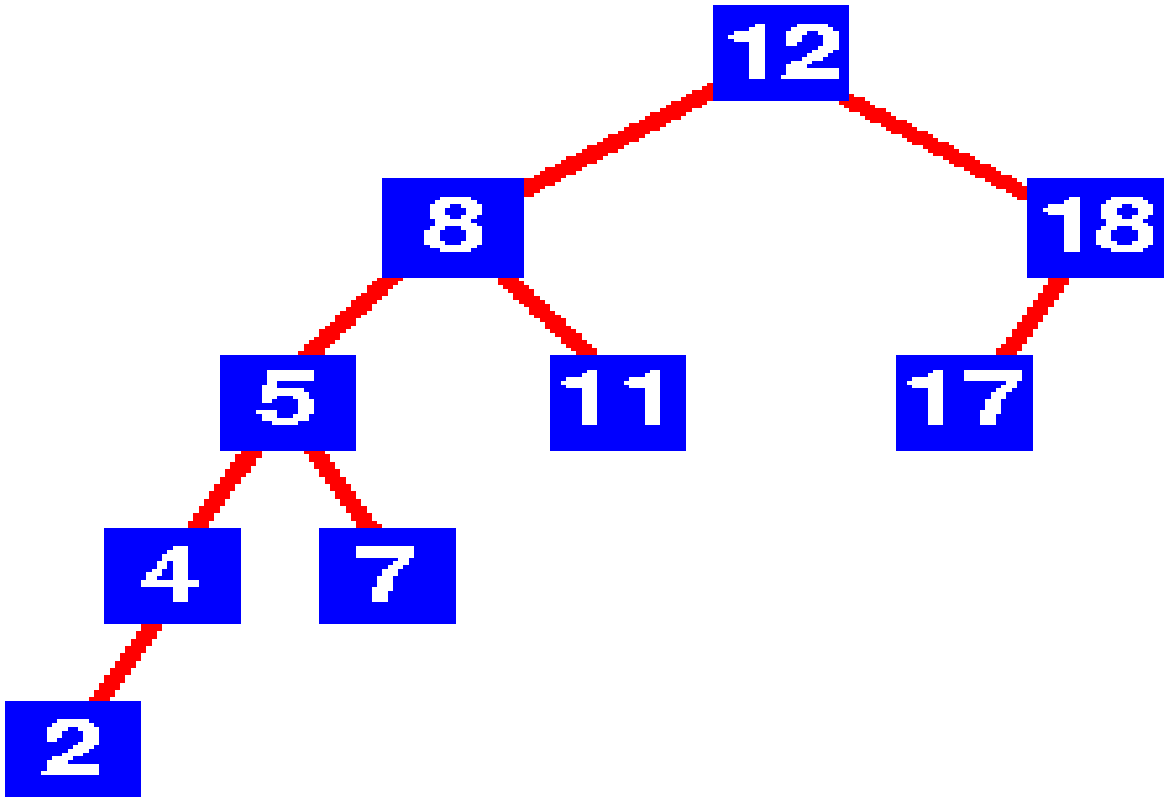
balance factor
1-(-1) = 2

height of node = h
balance factor = hleft-hright
empty height = -1

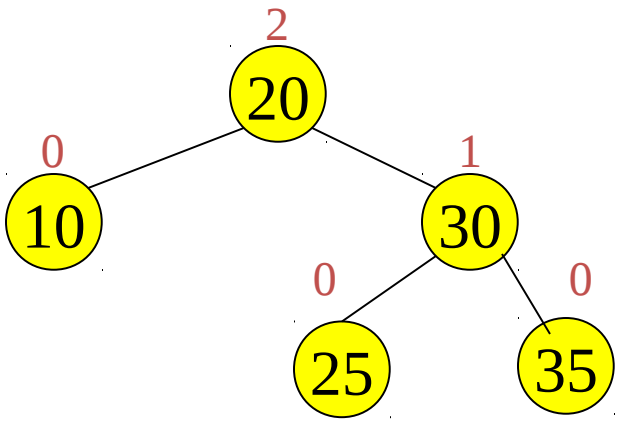
Is it AVL?



Is it AVL?

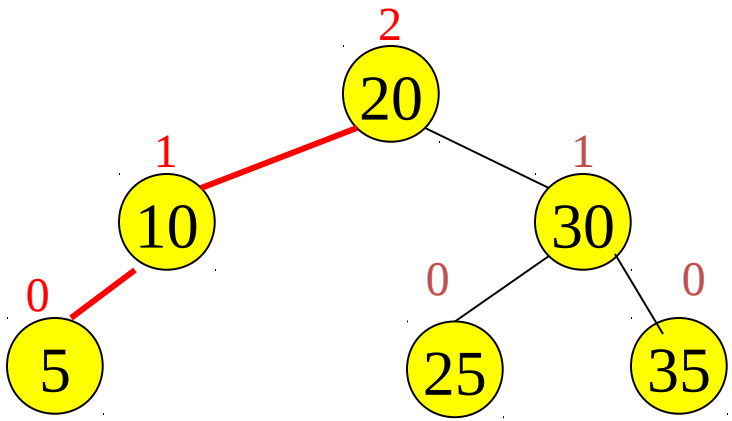


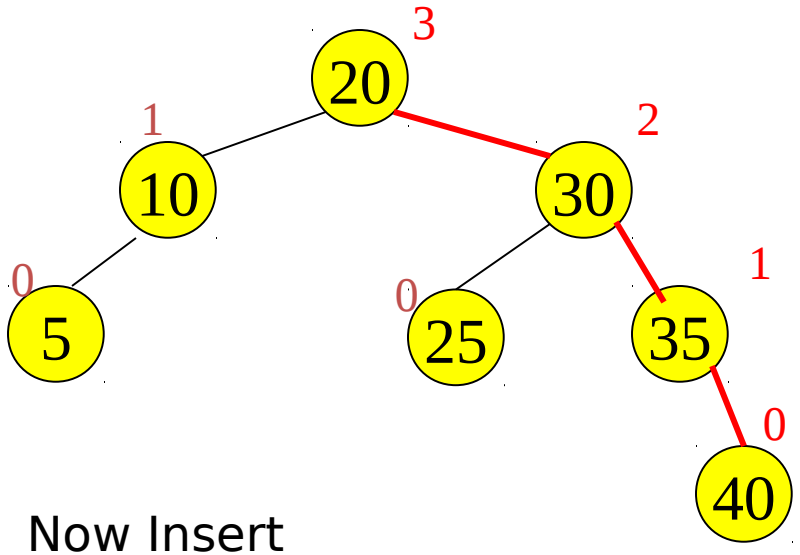
Example of Insertions in an AVL Tree



Insert 5, 40

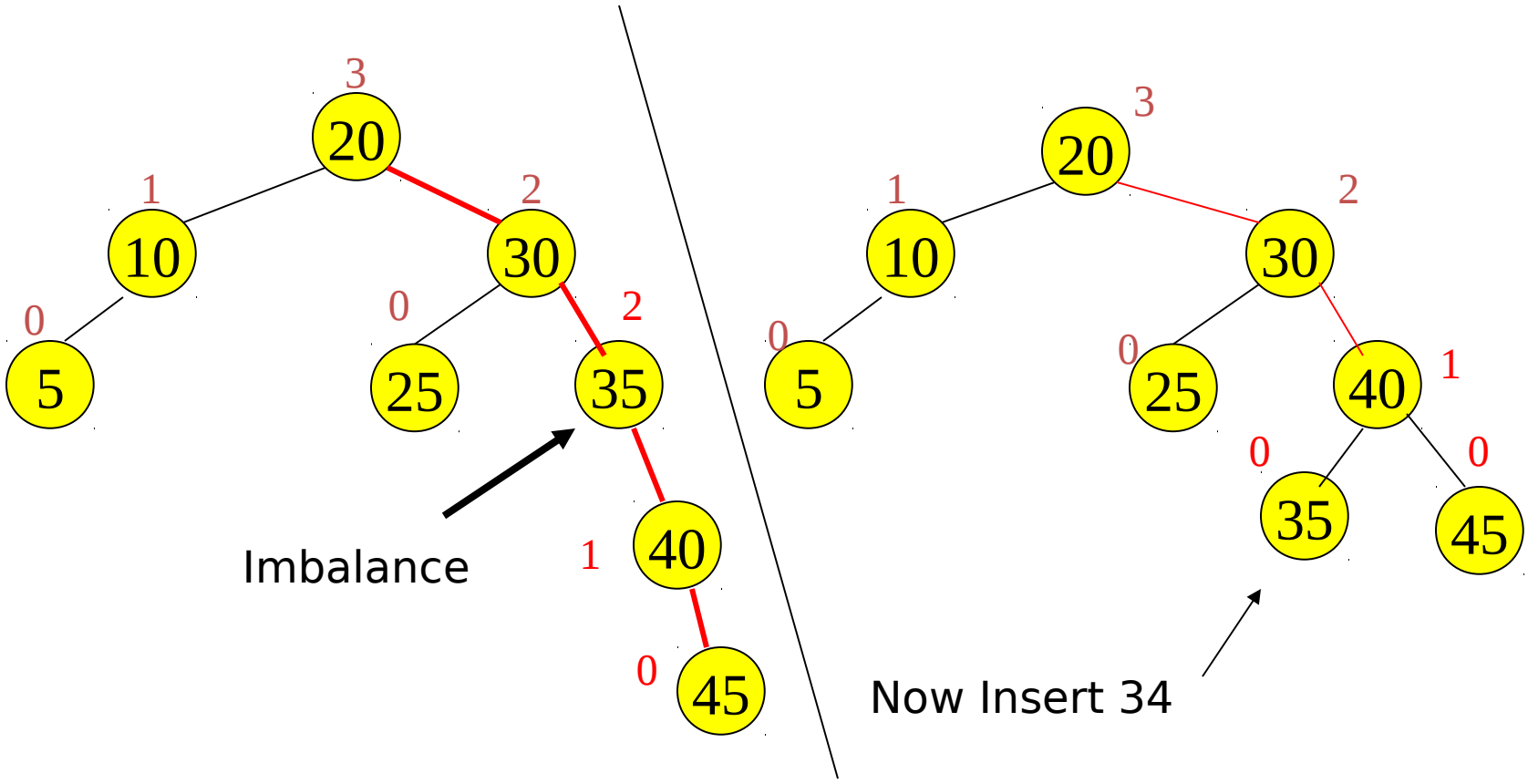
Example of Insertions in an AVL Tree





Now Insert 45

Single rotation (outside case)



Insertions in AVL Trees

Let the node that needs rebalancing be .

There are 4 cases:

Outside Cases (require single rotation) :

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

Inside Cases (require double rotation) :

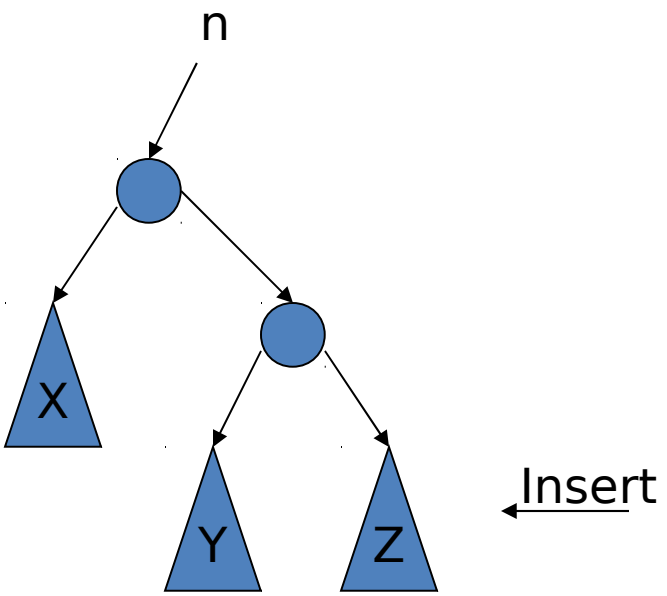
3. Insertion into right subtree of left child of .
4. Insertion into left subtree of right child of .

The rebalancing is performed through four separate rotation algorithms.

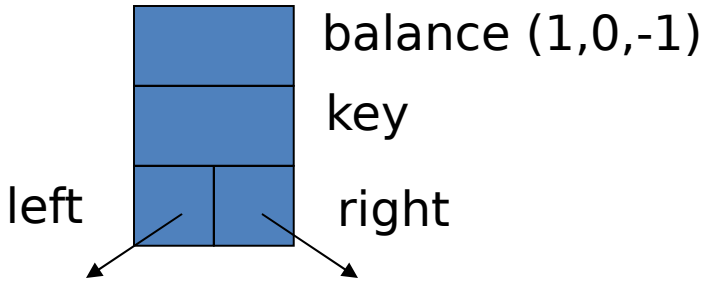
Single Rotation

```
RotateFromRight(n : reference node pointer) {
p : node pointer;
p := n.right;
n.right := p.left;
p.left := n;
n := p
}
```

You also need to modify the heights or balance factors of n and p



Implementation



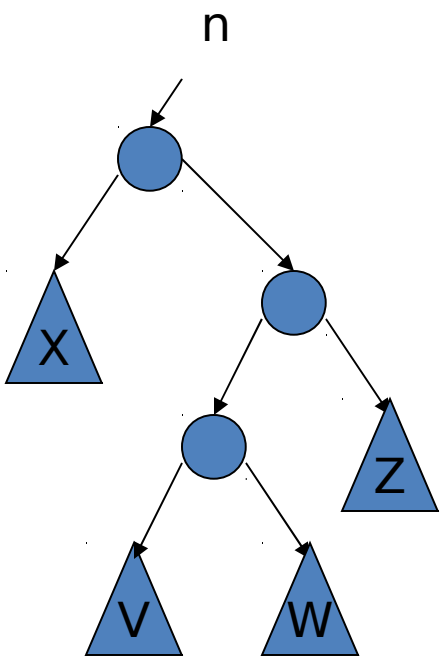
No need to keep the height; just the difference in height,
i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations
Once you have performed a rotation (single or double) you won't need to go back up the tree

Insert in AVL trees

```
Insert(T : reference tree pointer, x : element) : {
if T = null then
  {T := new tree; T.data := x; height := 0; return;}
case
  T.data = x : return ; //Duplicate do nothing
  T.data > x : Insert(T.left, x);
               if ((height(T.left)- height(T.right)) = 2){
                  if (T.left.data > x ) then //outside case
                         T = RotatefromLeft (T);
                  else                        //inside case
                         T = DoubleRotatefromLeft (T);}
  T.data < x :  Insert(T.right, x);
                code similar to the left case
Endcase
  T.height := max(height(T.left),height(T.right)) +1;
  return;
}
```

Double Rotation Solution

```
DoubleRotateFromRight(n : reference node pointer) {
RotateFromLeft(n.right);
RotateFromRight(n);
}
```



Thank You