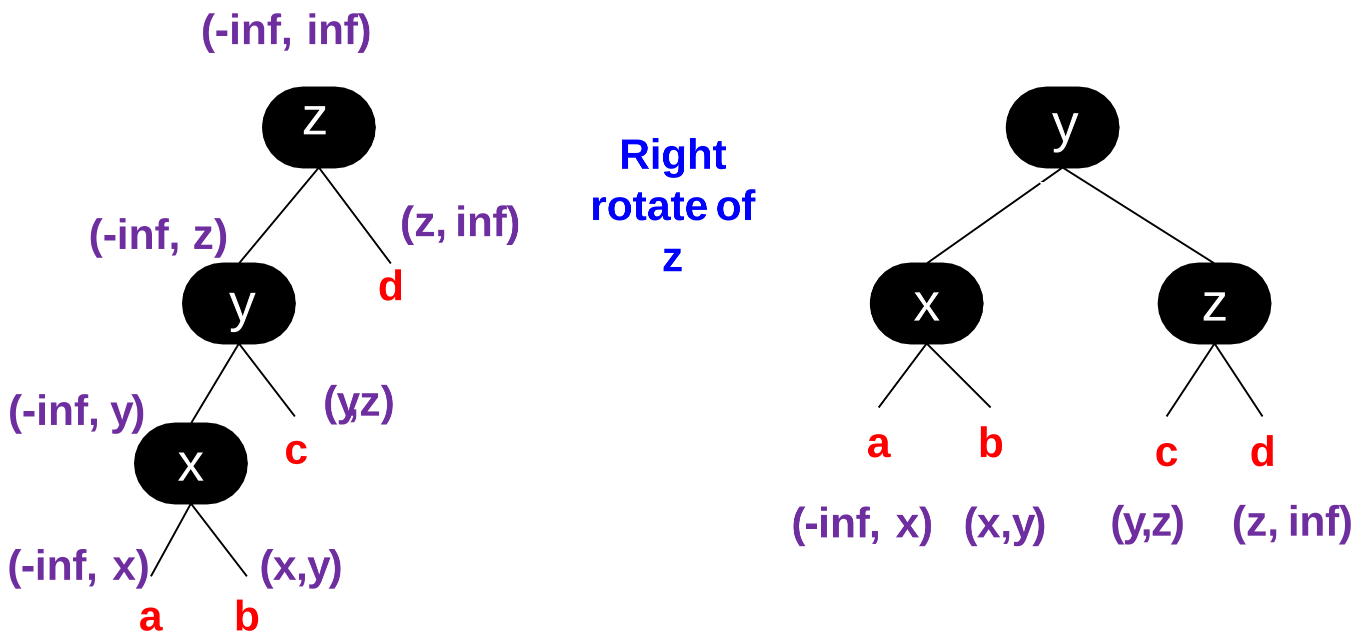
The key to balancing…

TREE ROTATIONS

Right Rotation

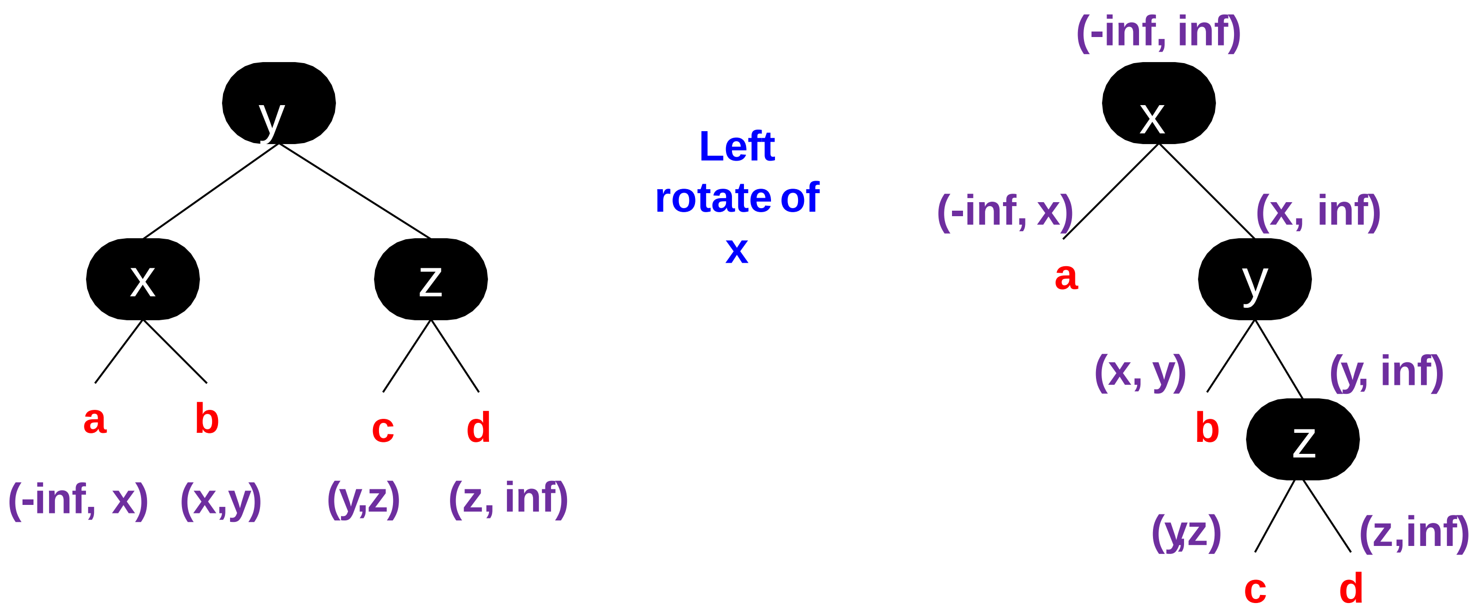
- Defining a right rotation at z:
  i) original left child of z, y, becomes parent of z
  ii) z becomes right child of y
  iii) original right child of y, c, becomes left child of z
- Call this **rotateRight(z, y)**



Left Rotation

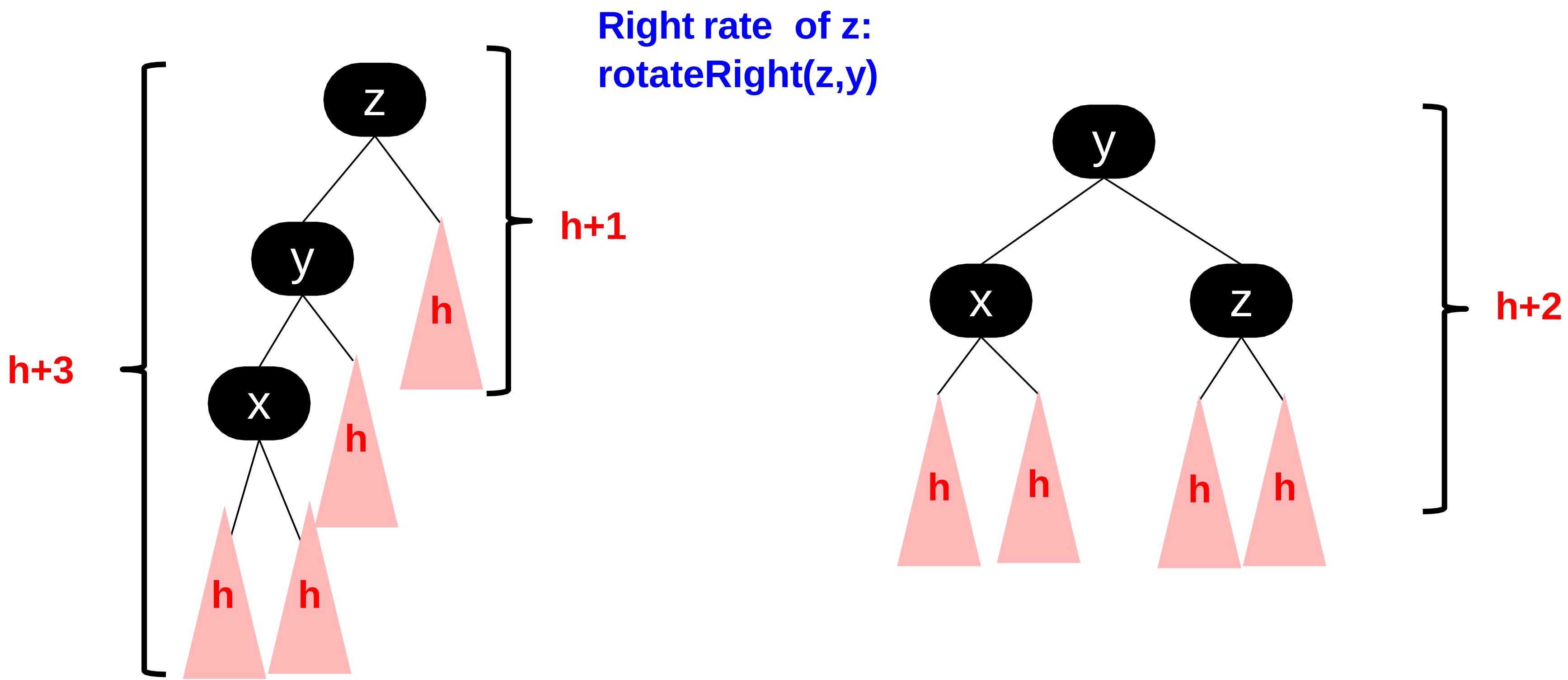
- Defining a left rotation at x:
  i) original right child of x, y, becomes parent of x
  ii) x becomes left child of y
  iii) original left child of y, b, becomes right child of x

Call this **rotateLeft(x, y)**



Rotation's Effect on Height

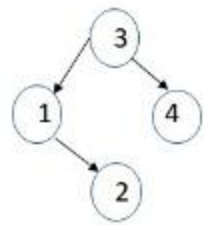
- When we rotate, it serves to re-balance the tree



Fundamental Theorem of Rotations

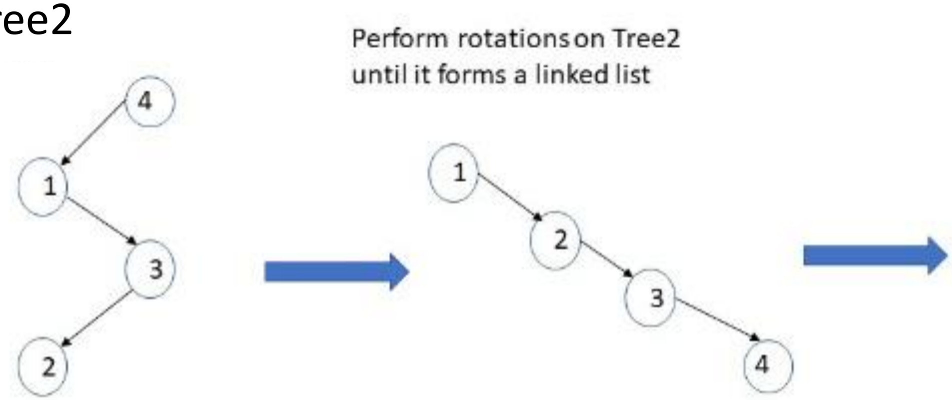
Any BST may be transformed into another BST on the same key values using only rotations.

Tree1



Perform rotations until
Tree2 is same as Tree 1

Tree2



Self-balancing tree proposed by Adelson-Velsky and Landis

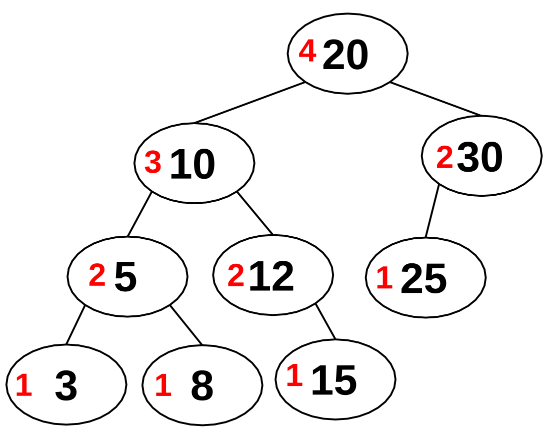
AVL TREES

AVL Trees

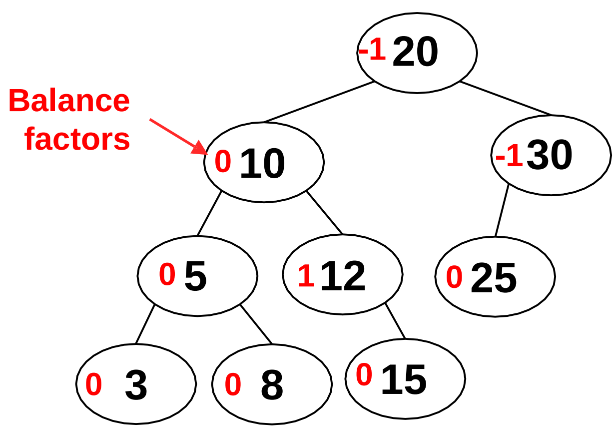
- AVL trees are binary trees such that

1) The Binary Search Tree (BST) Property holds: Left subtree keys are less than the root and right subtree keys are greater

2) **The height-balance property** holds: **height difference** between left and right subtrees of a node is **at most 1**



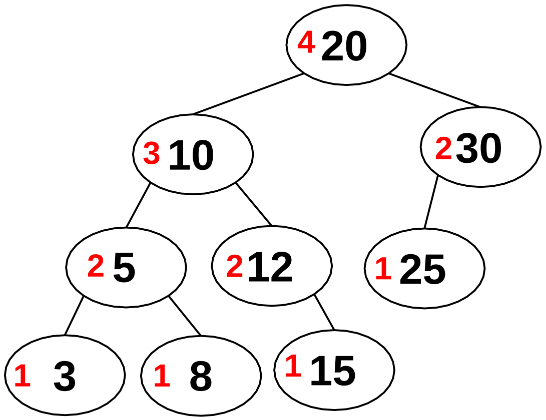
AVL Tree storing Heights



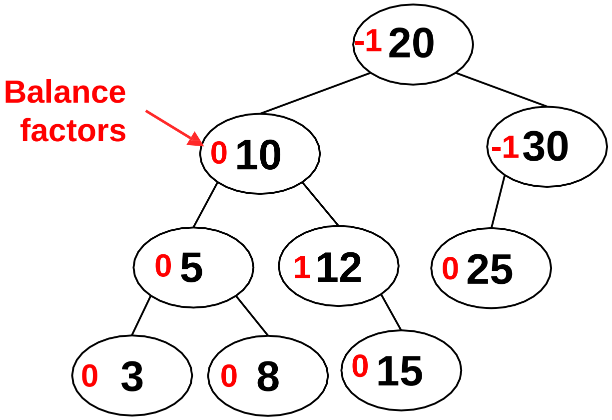
AVL Tree storing balances

AVL Trees

- Two implementations:
  - Height: Just store the height of the tree rooted at that node
  - **Balance: Define b(n) as the balance of a node = Height(right) –Height(left)**
    - Legal values are -1, 0, 1
    - Balances require at most 2-bits if we are trying to save memory.
    - **<u>We will use balance.</u>**



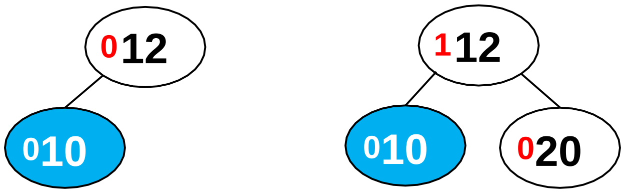
AVL Tree storing Heights



AVL Tree storing balances

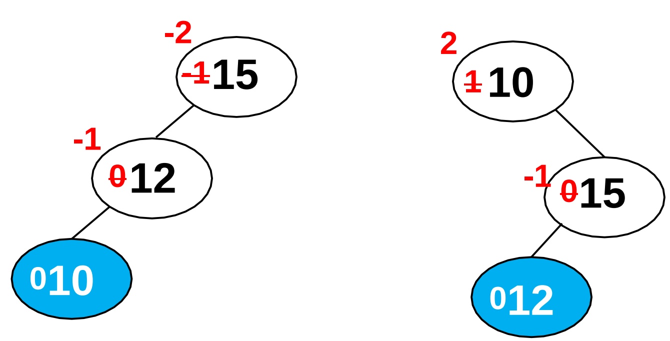
Adding a New Node

- A balanced parent node cannot be made out of balance by adding a child node
- What can happen now if we add a node at 10?

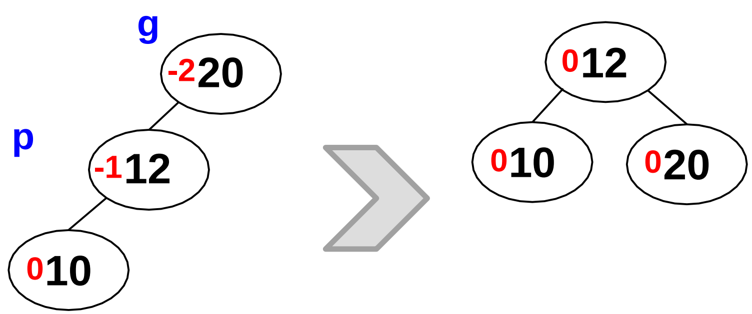


Losing Balance

- A grandparent node can lose balance.
- To fix, we will need rotations
- The rotations required to balance a tree are dependent on the grandparent, parent, child relationships



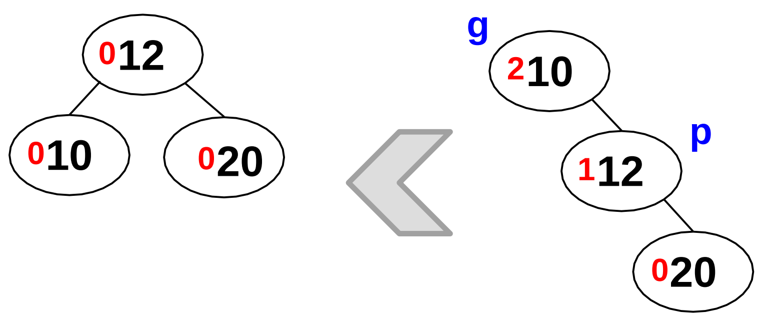
Single Rotation



- If the parent is left child of grandparent and child is left child of parent -> rotateRight(g, p)

- This pattern is called a zig-zig

Single Rotation



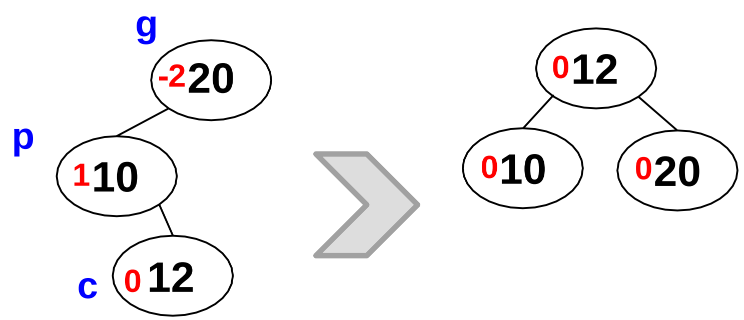
- If the parent is right child of grandparent and child is right child of parent -> rotateLeft(g,p)

This pattern is called a zig-zig

Double Rotations

- If the parent is left child of grandparent and child is right child of parent -> rotateLeft(p,c) followed by rotateRight(g,c)

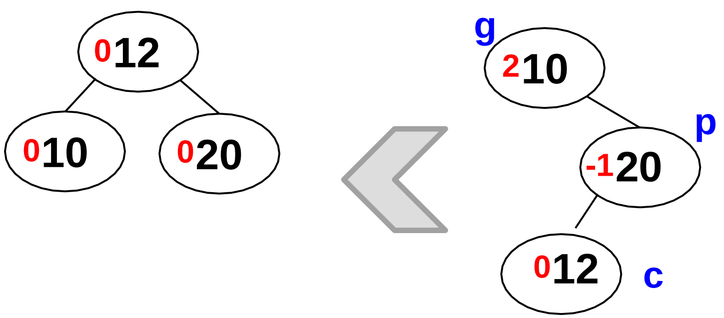
This pattern is called a zig-zag



Double Rotations

- If the parent is right child of grandparent and child is left child of parent -> rotateRight(p,c) followed by rotateLeft(g,c)

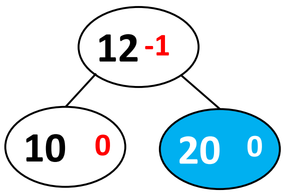
This pattern is called a zig-zag

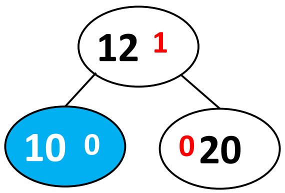


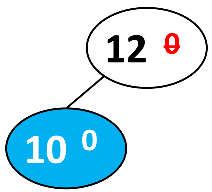
USC Viterbi School of Engineering

AVL Insert(n)

- If empty tree => set n as root, b(n) = 0, done!
- **BST insert(n**)
- Set balance of n: b(n) = 0
- Set balance of parent node p, b(p):
  If b(p) was -1, then b(p) = 0. Done!
  If b(p) was +1, then b(p) = 0. Done!
  If b(p) was 0, then update b(p) and call insert-fix(p, n)







AVL Insert-fix(p, n)

General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.

Precondition: p and n are balanced: {-1,0,1}

Postcondition: g, p, and n are balanced: {-1,0,1}

If p is null or `parent(p)` is null, return

Let `g = parent(p)`

Assume p is left child of g [For right child swap left/right, +/-]

```
b(g) += -1 // Update g's balance for taller left subtree
if b(g) == 0, return
if b(g) == -1, insertFix(g, p) // recurse
```

Insert-fix(p, n)

```
If b(g) == -2
    If zig-zig then rotateRight(g); b(p) = b(g) = 0
    If zig-zag then rotateLeft(p); rotateRight(g);
        Case 1: b(n) == -1 then b(p) = 0; b(g) = +1; b(n) = 0;
        Case 2: b(n) == 0  then b(p) = 0; b(g)= 0;   b(n) = 0;
        Case 3: b(n) == +1 then b(p)= -1; b(g) = 0;  b(n) = 0;
```

General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.

Note: Once a rotation is performed to balance a node, algorithm stops

Insert-fix(p, n)

```
If b(g) == -2
    If zig-zig then rotateRight(g); b(p) = b(g) = 0
    If zig-zag then rotateLeft(p); rotateRight(g);
        Case 1: b(n) == -1 then b(p) = 0; b(g) = +1; b(n) = 0;
        Case 2: b(n) == 0  then b(p) = 0; b(g)= 0;   b(n) = 0;
        Case 3: b(n) == +1 then b(p)= -1; b(g) = 0;  b(n) = 0;
```

General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.

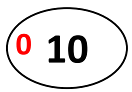
Note: Once a rotation is performed to balance a node, algorithm stops

Insertion

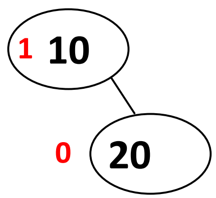
Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

Empty

Insert 10

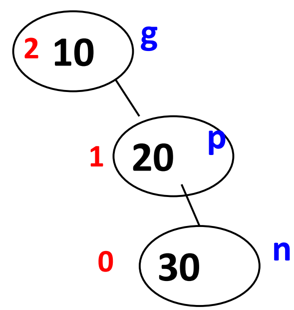


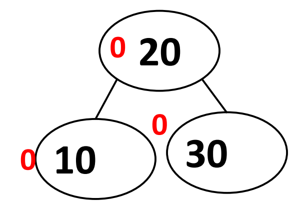
Insert 20



Insert 30

10 violates balance

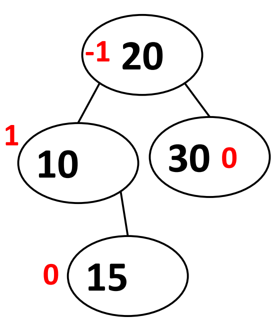




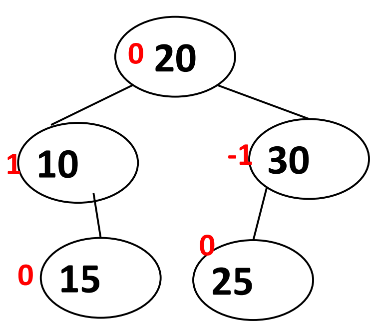
Insertion

Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

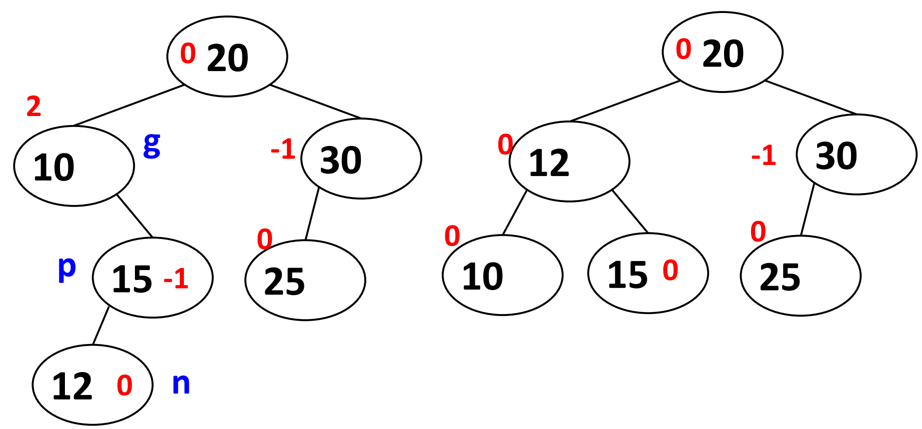
Insert 15



Insert 25



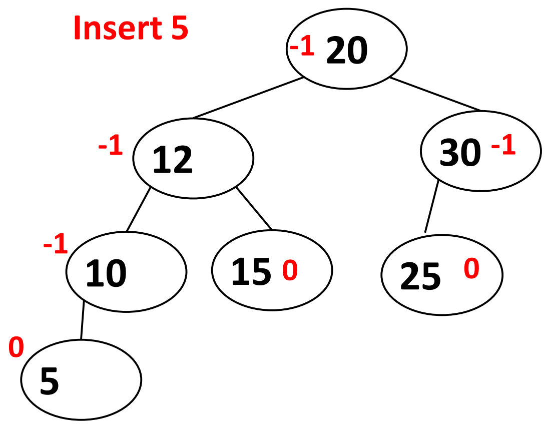
Insert 12



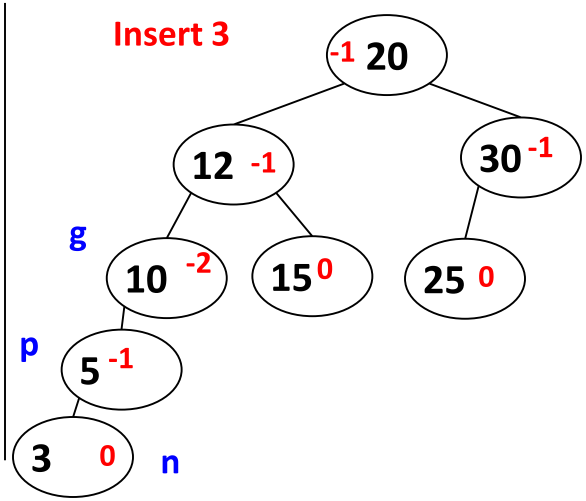
Insertion

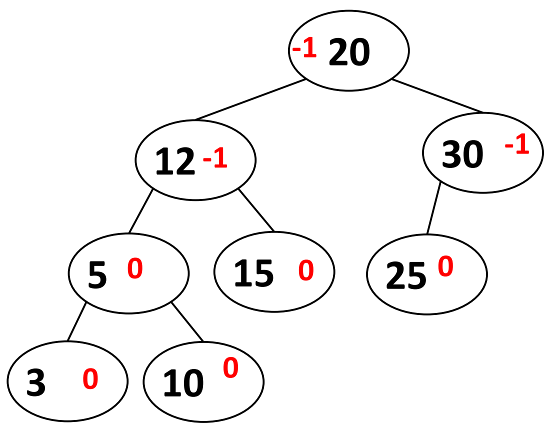
Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

Insert 5



Insert 3





Insertion

Insert 10, 20, 30, 15, 25, 12, 5, 3, 8