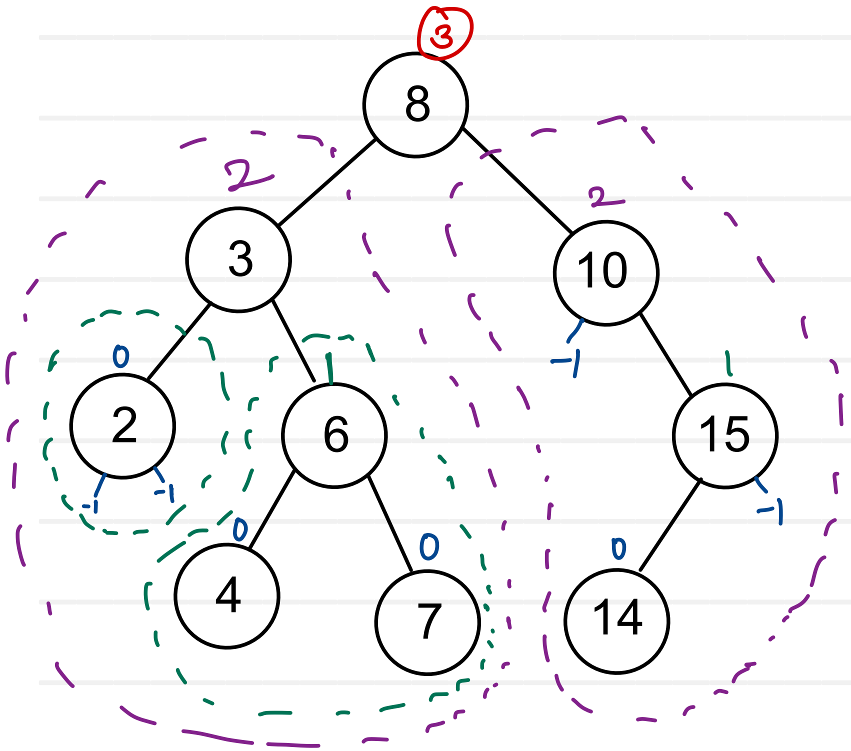
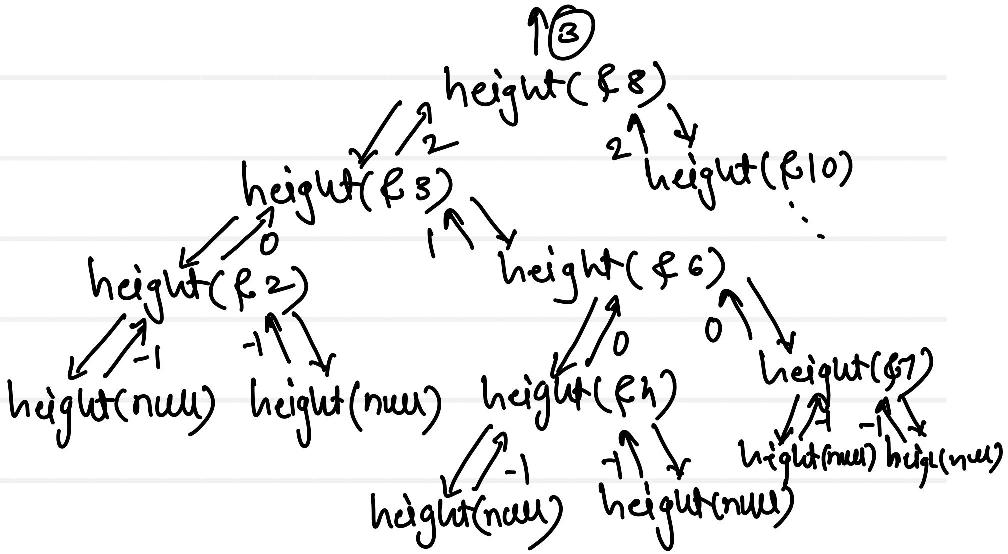
# Binary Search Tree - Height

Height of root = MAX ( height (left sub tree), height (right sub tree)) + 1

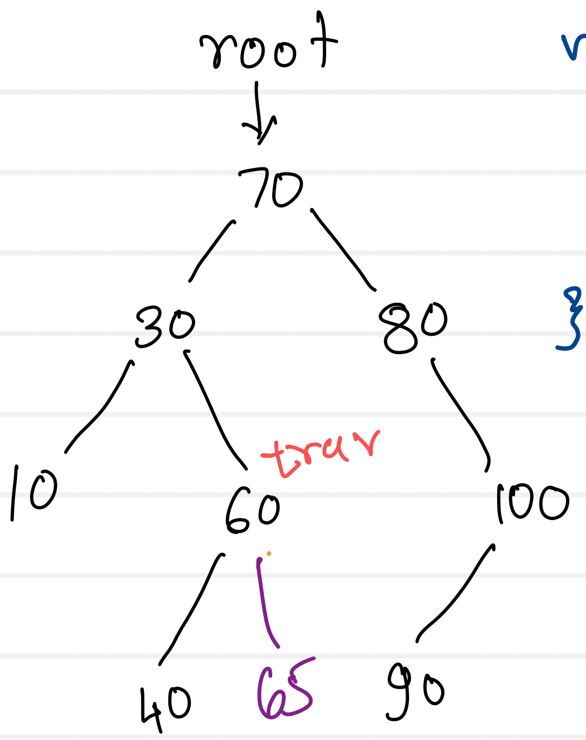



1. If left or right sub tree is absent then return -1
2. Find height of left sub tree
3. Find height of right sub tree
4. Find max height
5. Add one to max height and return

```
int height(Node trav) {
    if(trav == null)
        return -1;
    int hl = height(trav.left);
    int hr = height(trav.right);
    int max = hl > hr ? hl : hr;
    return max+1;
}
```



# Add Node (Recursion)

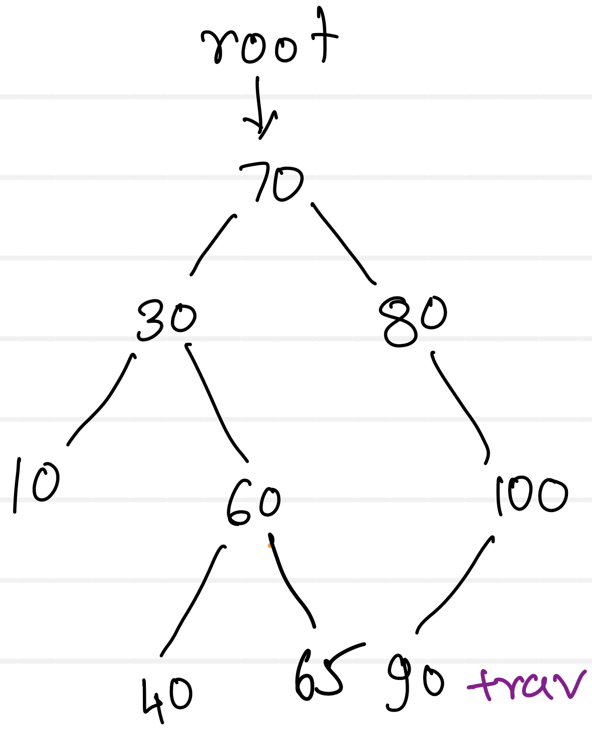



```
void add(int value) {
    if (root == null)
        root = new Node(value);
    else
        add(root, value);
}
```

```
add(65)
add(&70, 65)
add(&30, 65)
add(&60, 65)
```

```
void add(Node trav, int value) {
    if(value < trav.data) {
        if(trav.left == null)
            trav.left = new Node(value);
        else
            add(trav.left, value);
    }
    else {
        if(trav.right == null)
            trav.right = new Node(value);
        else
            add(trav.right, value);
    }
}
```

# Binary Search (Recursive)




```
Node binarySearch( Node trav , int key) {
if (trav == null)
        return null;
if( key == trav.data)
        return trav;
else if( Key < trav.data)
        return binarySearch(trav.left , key);
else
        return binarySearch(trav.right, key);

}
```

binarySearch (&70 , 90)
binarySearch( &80, 90)
binarySearch(&100, 90)
binary search (&90, 90)

# BST - Time complexity of operations

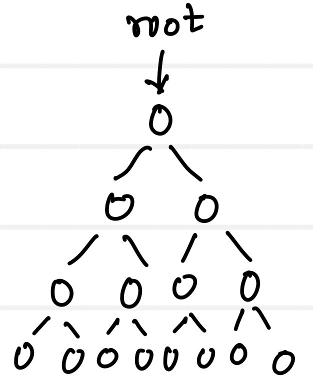
h - height of BST
n - no. of elements (Nodes) in BST

$$\boxed{n = 2^{h+1} - 1}$$

| | | | |
|---|---|---|---|
| Add | : | $O(h)$ | $O(\log n)$ |
| search | : | $O(h)$ | $O(\log n)$ |
| delete | : | $O(h)$ | $O(\log n)$ |
| traverse | : | | $O(n)$ |

Capacity : max number of nodes for given height

| height | No. of Node |
|---|---|
| -1 | 0 |
| 0 | 1 |
| 1 | 3 |
| 2 | 7 |
| 3 | 15 |




$$n = 2^{h+1} - 1$$

$$2^h \approx n$$

$$\log 2^h = \log n$$

$$h = \frac{\log n}{\log 2}$$

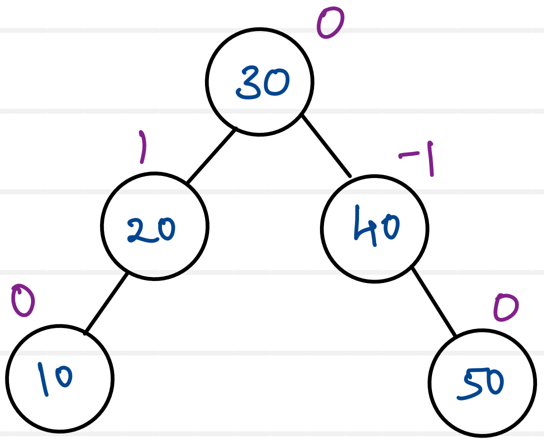
Time $\propto h$

Time $\propto \frac{1}{\log 2} \log n$

$$\boxed{T(n) = O(\log n)}$$

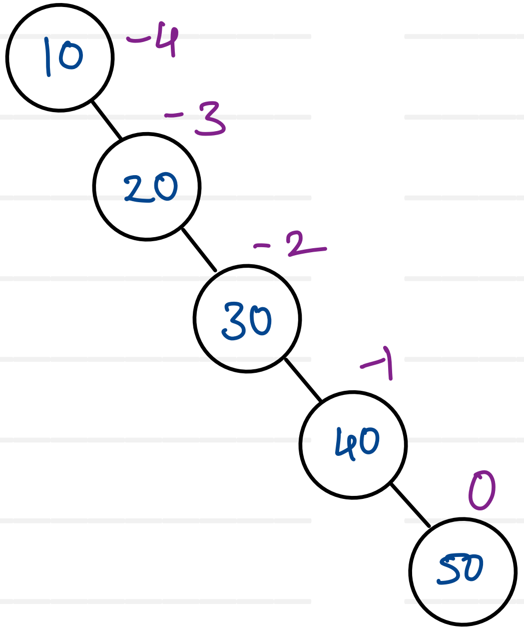
# Skewed Binary Search Tree

Keys : 30, 40, 20, 50, 10



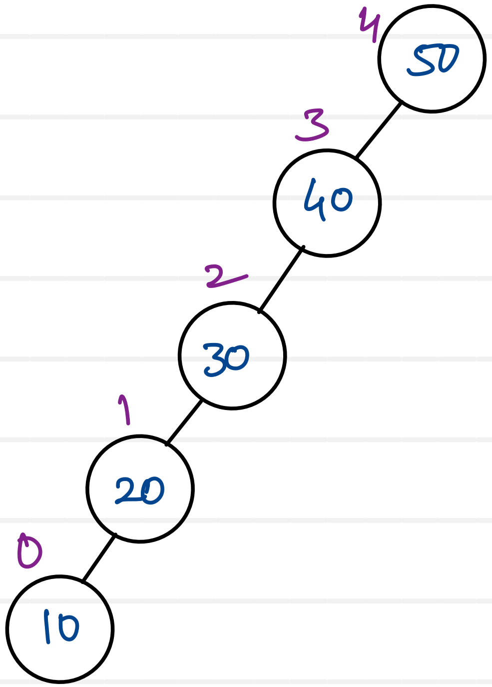

height = log n
$T(n) = O(\log n)$

Keys : 10, 20, 30, 40, 50




height = n
$T(n) = O(n)$

Keys : 50, 40, 30, 20, 10




- In binary tree if only left or right links are used, tree grows only in one direction such tree is called as skewed binary tree
  - Left skewed binary tree
  - Right skewed binary tree
- Time complexity of any BST is O(h)
- Skewed BST have maximum height ie same as number of elements.
- Time complexity of searching is skewed BST is O(n)

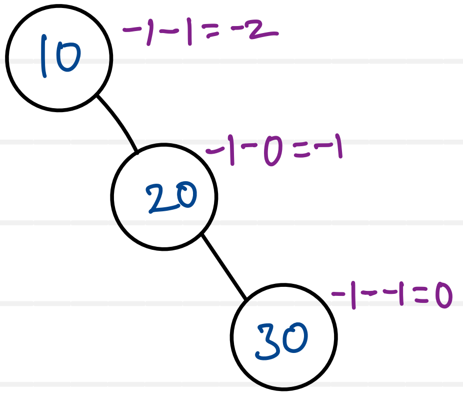
# Balanced BST

- To speed up searching, height of BST should be minimum as possible
- If nodes in BST are arranged, so that its height is kept as less as possible, is called as Balanced BST

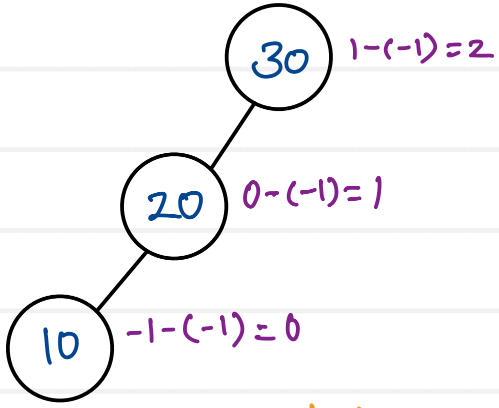
Balance factor = Height (left sub tree) - Height (right sub tree)

- tree is balanced if balance factors of all the nodes is either -1, 0 or +1
- balance factors = {-1, 0, +1}
- A tree can be balanced by applying series of left or right rotations on imbalance nodes (node having balance factor other than -1, 0 or +1)

Keys : 10, 20, 30

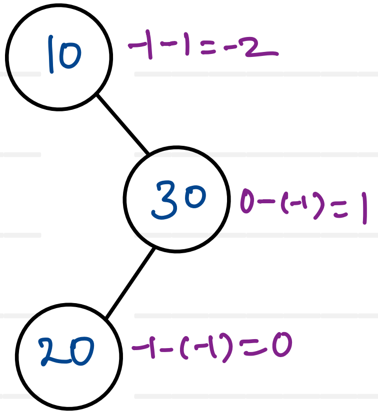



Keys : 30, 20, 10

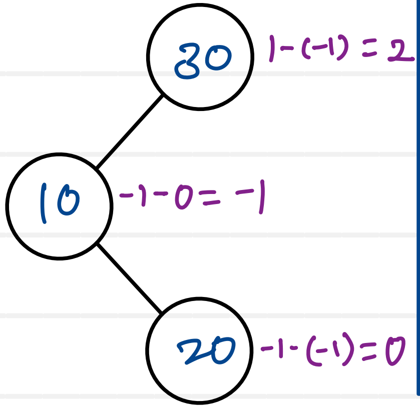



height= 2

Keys : 10, 30, 20

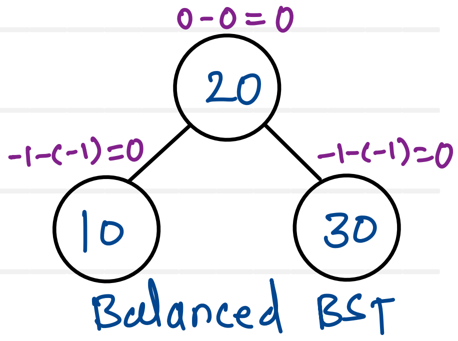



Keys : 30, 10, 20



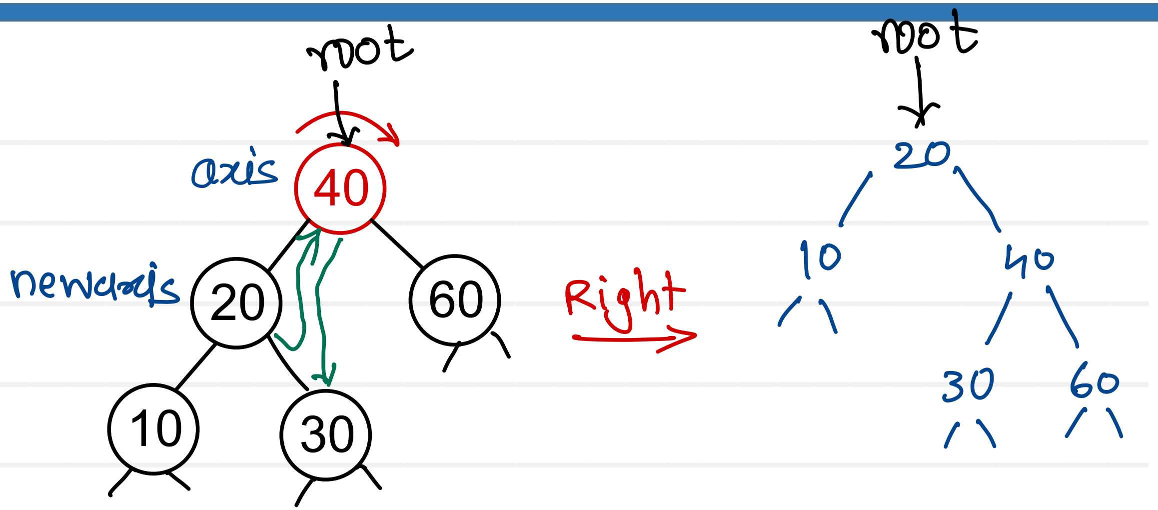

Keys : 20, 10, 30

Keys : 20, 30, 10

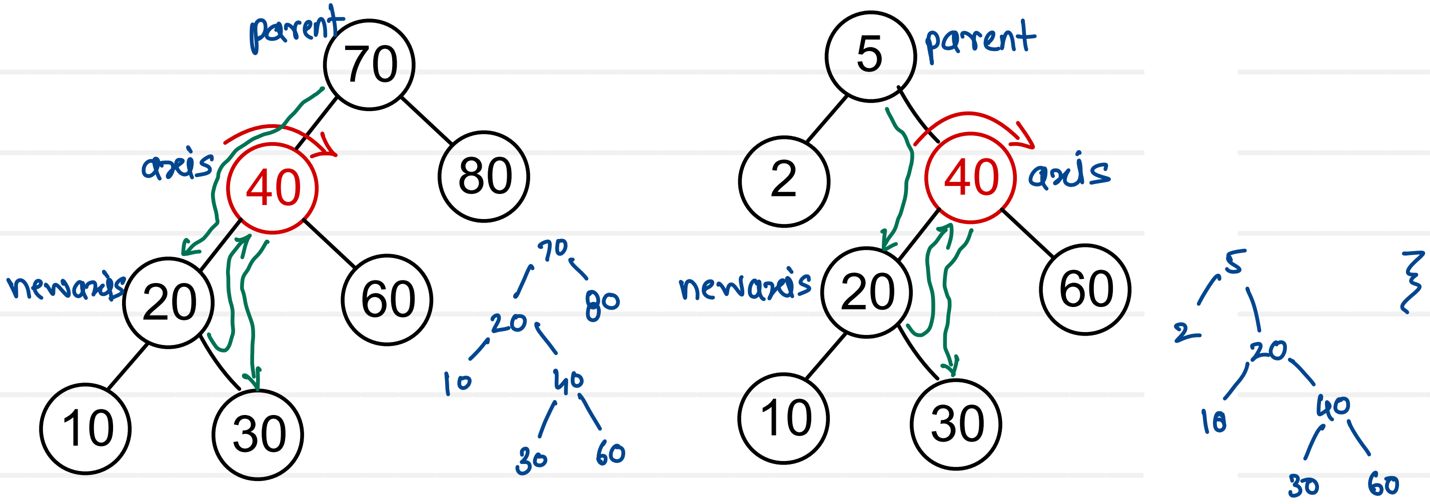



height = 1

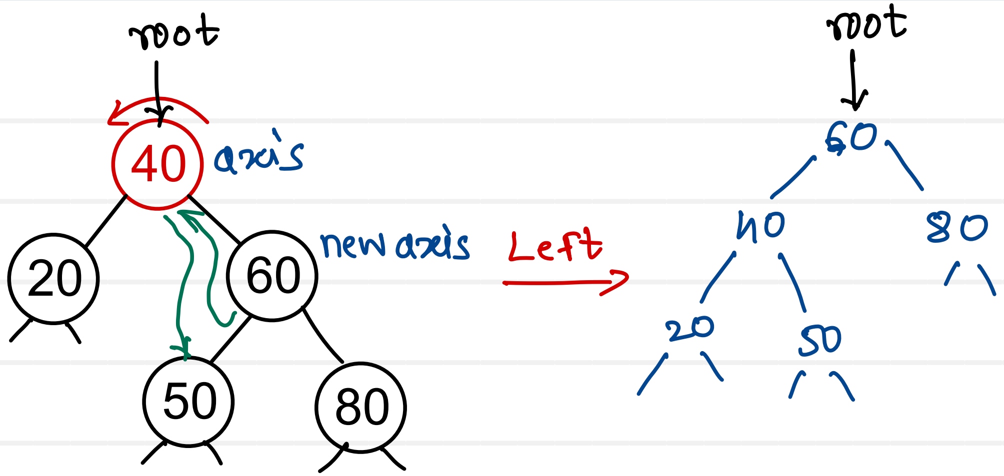
# Right Rotation



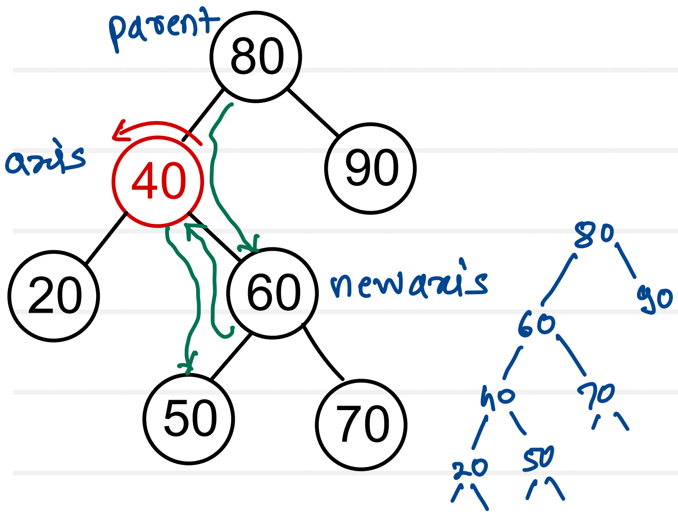

```
void rightRotation(axis, parent) {
    Node newaxis = axis.left;
    axis.left = newaxis.right;
    newaxis.right = axis;
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```

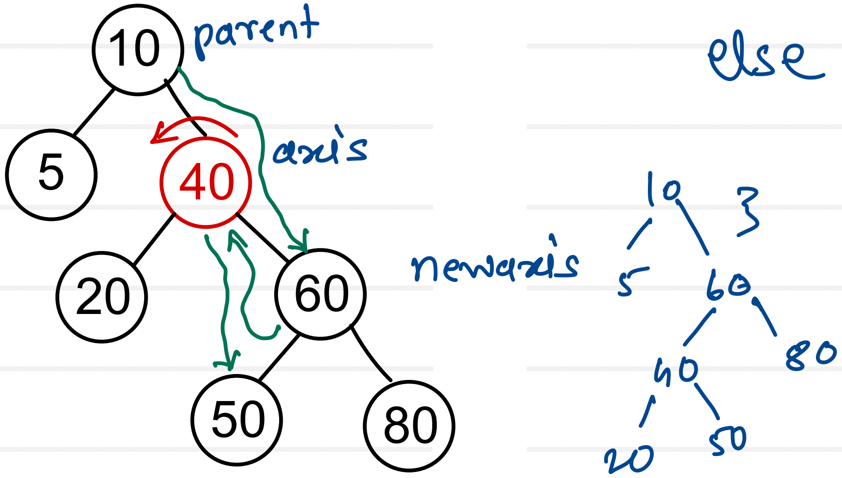


# Left Rotation




```
void leftRotation( parent, axis ){
    Node newaxis = axis.right;
    axis.right = newaxis.left;
    newaxis.left = axis;
    if( axis == root)
        root = newaxis;
    else if (axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```

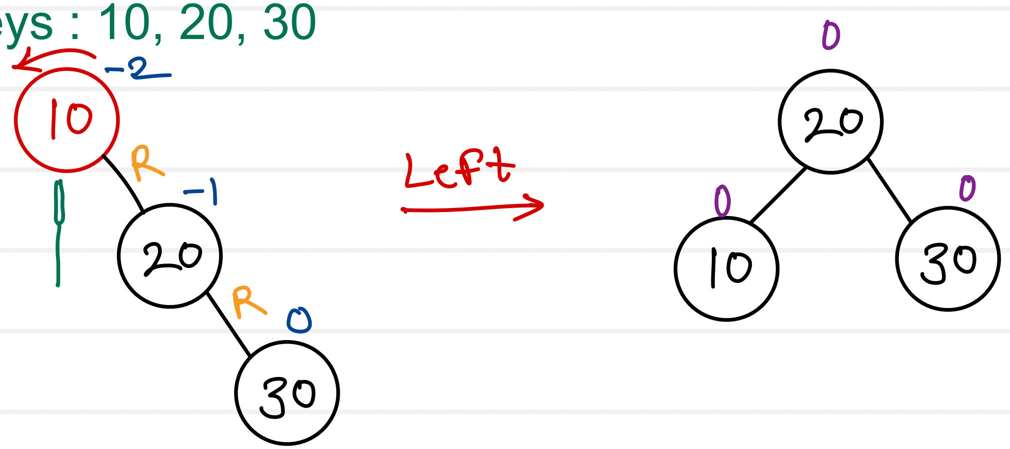


# Rotation cases (single Rotation)

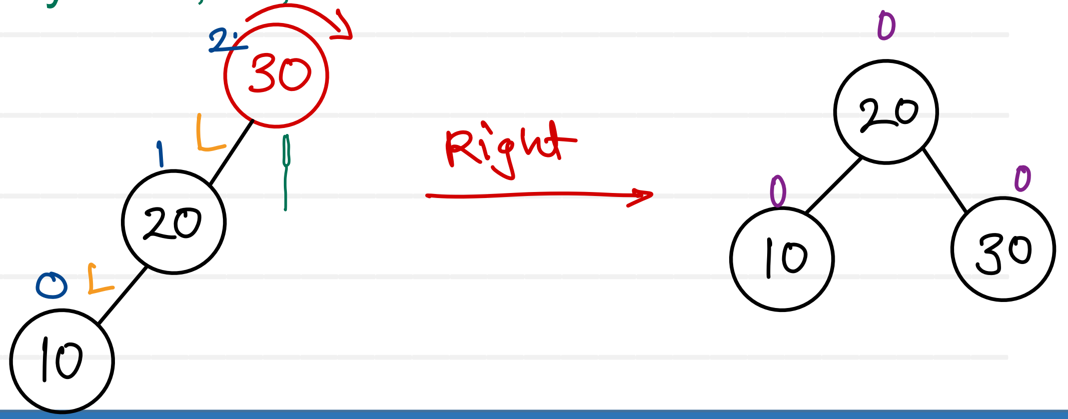
RR Imbalance

Keys : 10, 20, 30




LL Imbalance

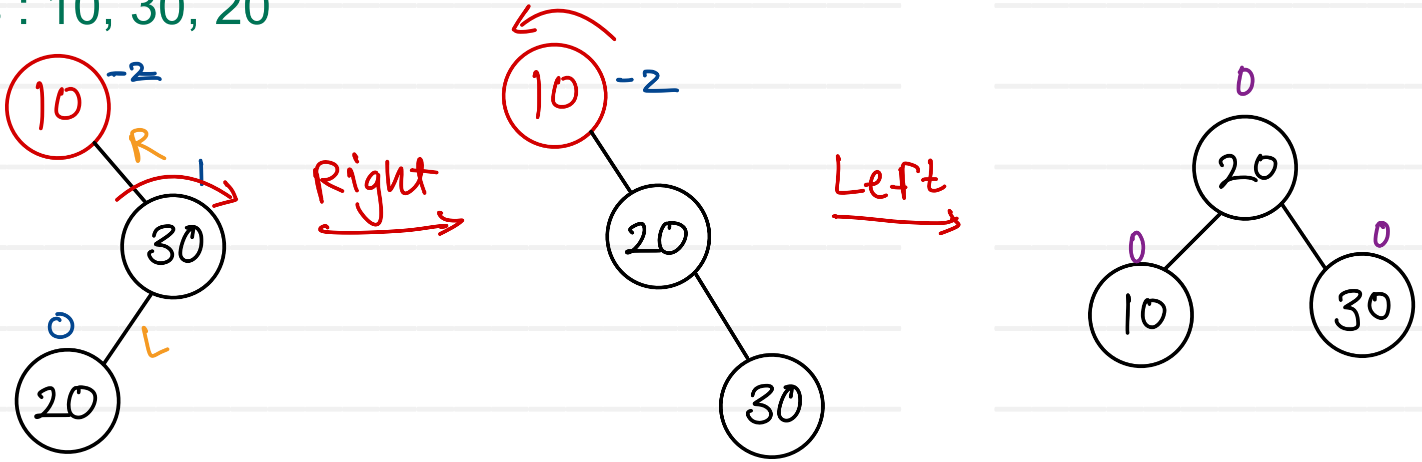
Keys : 30, 20, 10



# Rotation cases ( Double Rotation)

## RL Imbalance

Keys : 10, 30, 20



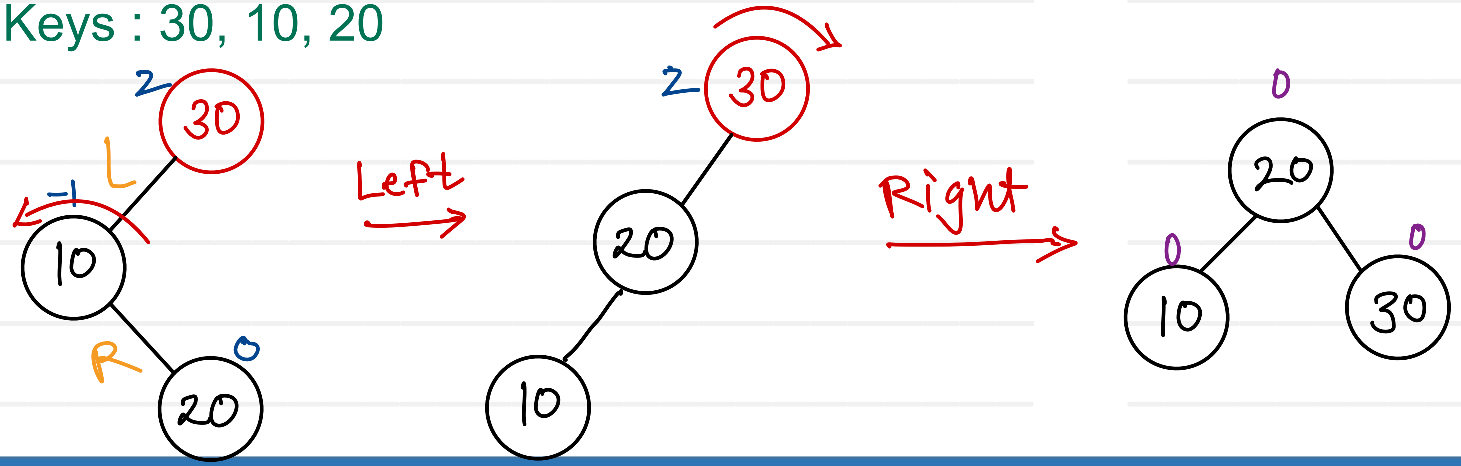

↓
RL

Right → right of imbalance node

Left → imbalance node

## LR Imbalance

Keys : 30, 10, 20




↓
LR

Left → left of imbalance node

right → imbalance node

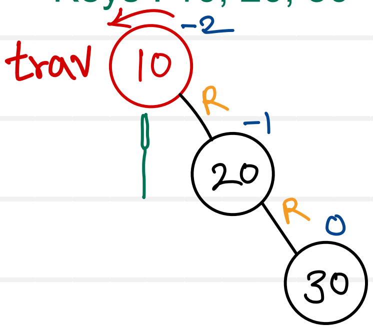
BF
↓
{-1, 0, +1}

BF < -1

BF > 1

RR Imbalance

Keys : 10, 20, 30

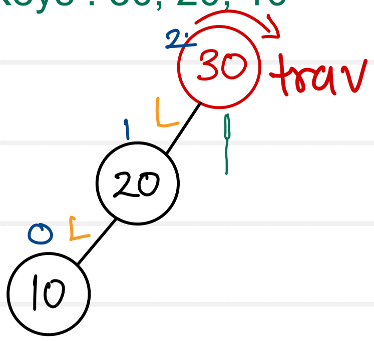



value > trav.right.data

(30 > 20)

LL Imbalance

Keys : 30, 20, 10

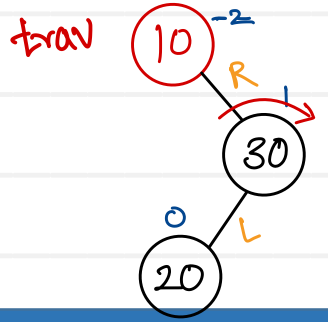



value < trav.left.data

(10 < 20)

RL Imbalance

Keys : 10, 30, 20

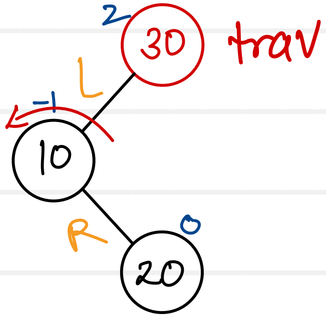



value < trav.right.data

(20 < 30)

LR Imbalance

Keys : 30, 10, 20




value > trav.left.data

(20 > 10)

# Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com