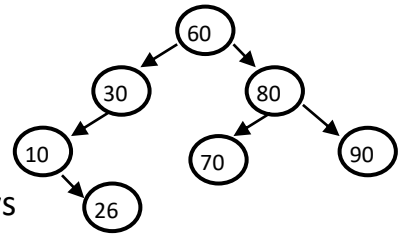
Binary Search Tree

A binary search tree is a search tree that makes search operation efficient. It is efficient in finding smallest and largest node in the tree. A binary search tree is a binary tree with following properties.

1. The keys in the left sub tree are less than to the key of the root.
2. The keys in the right sub tree are greater than or equal to the key of root.
3. The left and right sub trees are binary search tree.



Operations on binary sub-tree

1. Searching a key
2. Finding the node with minimum and maximum keys
3. Finding successor and predecessor of a node
4. Insertion and deletion of node

Searching of a node in a binary search tree:

```
BSTree_Search(X,k)
//This procedure search a key k in a binary search tree
//whose root is pointed by X.
1. if X = NIL  OR k = key[X]
2.       return X
3. if k< key[X]
4.       return BSTree_Search (Left[X], k)
5. else, return BSTree_Search (Right[X],k)
```

Analysis: At each recursive call, X is compared to key[X]. If it is equal, then search terminates. If it is not equal to the key, either search continues with left sub-tree or in right sub-tree until a match is found or set X = NIL. As only one path from the root to a leaf node is compared. So the running time is O(h). Where, h is the height of tree.

Find a node with minimum key

In a binary search tree, the minimum element always exists at the extreme left node of the tree. It can be obtained by tracing left child pointer from root until a NIL is encountered.

```
BSTree_Minimum(X)
1. While left[X] ≠ NIL
2.   X = left[X]
3. Return X
```

Insertion of a node in BST

To insert a node Z in a BST, compare the key[Z] with root . If Key[Z] < key[root], then move to left subtree. Else, move to right subtree. Continue the process until Null sub-tree is encountered. When there is no left sub-tree or right sub-tree remains, associate Z as a child of last processed node.

```
BSTreee_Insert(X,Z)
1. q = NIL
2. p = X
3. while p ≠ NIL
4.       q = p
5.       if key[z] < Key[p]
6.                ptr = left[p]
7.       else,    ptr = right[p]
8. if q == NIL
9.       X = Z // tree X was empty
10.Else if key[Z] < key[q]
11.      left[q] = Z
12.Else, right[q] = Z
```

Analysis: To insert a new node, one path of the tree starting from the root to a leaf node is traversed. The running time of algorithm is O(h).

Deletion of a node

To delete a node 'Z' from a BST 'T', consider following cases.

1. The node Z has no children,

   Modify the parent of Z to replace child Z with NULL.
2. Node Z has a single child(either left or right)

   Splice out the node Z to make a link between z's child and z's parent,.
3. Node Z has both children, (Perform either of following)
   - Find the node 'Y' with minimum key in right subtree of Z and replace Z's key with Y's key.

     [OR]
   - Find the node 'Y' with maximum key in left subtree of Z and replace Z's key with Y's key.

Height Balanced Trees

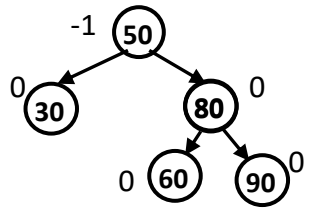
Height balanced tree is type of binary search tree with reduced height. As the running time of search operation directly dependent on height of the tree, balancing the height makes this tree makes search operation more efficient than that of simple binary search tree. The advantage of height balanced tree is that the worst case searching time is O(logn), whereas, it is O(n) in BS Tree. Some of the height balanced Trees are : AVL Tree, Red-Black Tree, B tree and etc.

AVL Tree

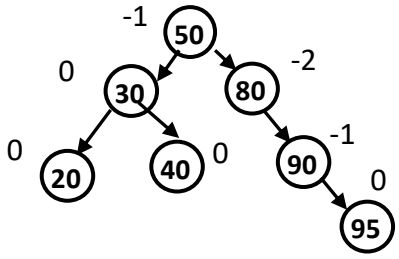
AVL tree is named after 2 mathematicians G.M Adelson-Velskii and E Landis.

A Binary Search tree is called AVL if the balance factor of each node does not exceed 1. i ,e balance factor of a node should be either 0 or 1 or -1

Balance factor of a node = Height of its Left sub tree - Height of its right sub tree.



AVL tree



Not an AVL tree

Balancing the Binary search Tree:

The AVL Tree becomes unbalanced due to insertion or deletion of nodes. So it must be re-balanced. There are four cases; a sub-tree /tree rooted at a node(X) become unbalanced.

Case-1: An insertion into left sub-tree of left child of node X

Case-2: An insertion into right sub-tree of right child of node X

Case-3: An insertion into left sub-tree of right child of node X

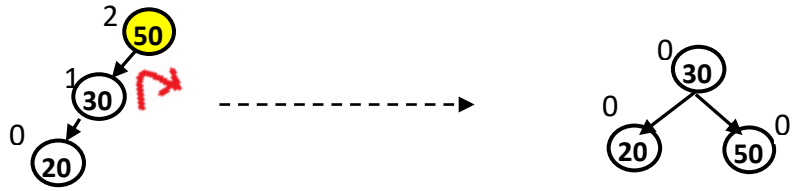
Case-4: An insertion into right sub-tree of left child of node X

Case-1: Insertion of a node into left sub-tree of left child of node X

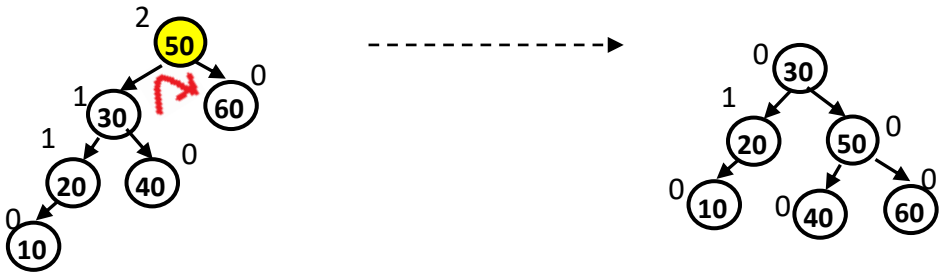
To balance it, A **single rotation** at unbalanced node is required as follows.

- Rotate right, the unbalanced node (X) towards right. If left child of unbalanced node has a right sub-tree, then root of the right sub-tree become the left sub-tree of unbalanced node.

Example 1a:



Example 1b

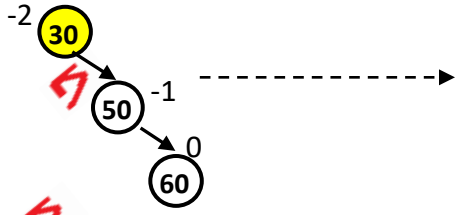


Case-2: Insertion of into right sub-tree of right child of node X

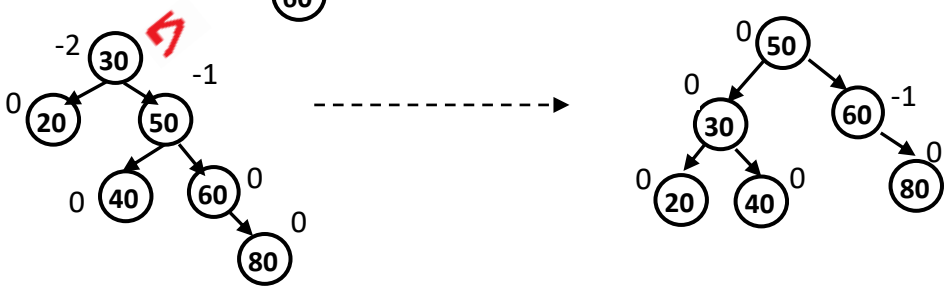
To balance it, A **single rotation** at unbalanced node is required as follows.

- Rotate left, the unbalanced node (X) towards left. If right child of unbalanced node has a left sub-tree, then root of the left sub-tree become the right sub-tree of unbalanced node.

Example 2a:



Example2 b

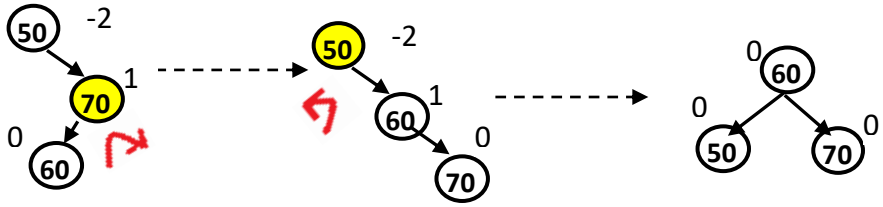


Case-3: Iinsertion of a node into left sub-tree of right child of X

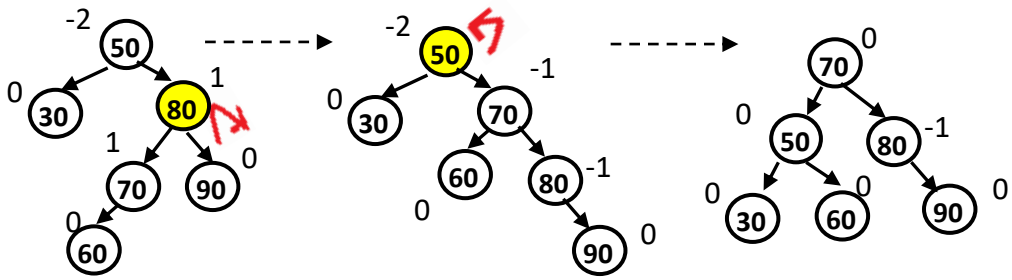
To balance it, **two rotations** are required as follows.

- Rotate right, The right child of node X with left child
- Rotate left the unbalance node (X) with right child of X

Example 3 a:



Example 3b

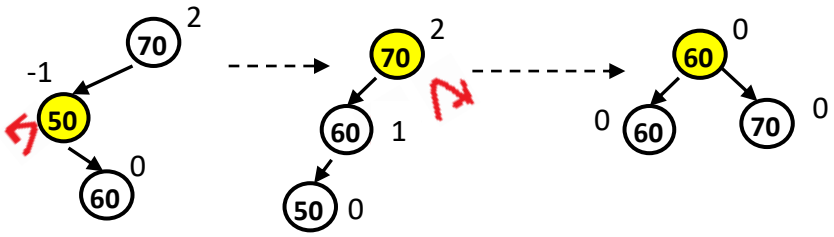


Case-4: **I**nsertion of a node into right sub-tree of left child of X

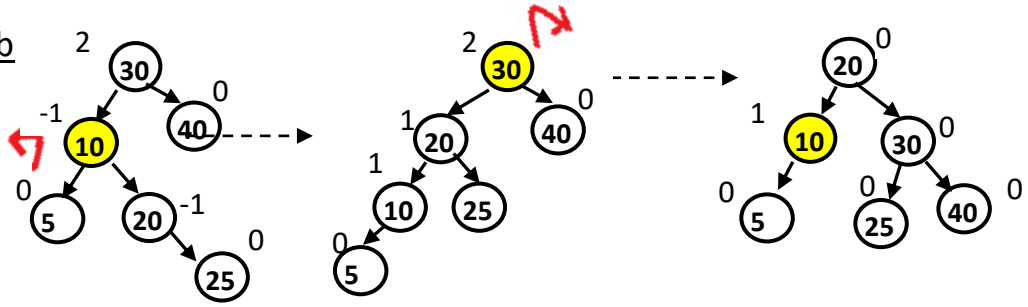
To balance it, **two rotations** are required as follows.

- Rotate left, The left child of node X with left child
- Rotate right the unbalance node (X) with right left of X

Example 4 a:



Example 4b



Try Yourself: Insert following nodes into AVL tree.

1. 10, 20, 30, 40, 50, 60, 70, 80, 90

2. JAN,FEB,,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC

3. 80,70,60,50,40,30,20,10

Algorithms:

Returning height of a node

```
AVLHeight(T)
1. if T = NIL
2.    return -1
3. else, return height[T]
```

Case-1 : Single rotation of a node with left child (Right Rotation)

```
RotateRightWithLeftChild(Z)
1. X = left[Z]
2. left[Z]= right[X]
3. right[X] = Z
4. height[Z] = Max(AVLHeight(left[Z) , AVLHeight(right[k])) + 1
5. height[X] = Max(AVLHeight(left[X) , AVLHeight(right[X])) + 1
6. Z = X
```

Case-2: Single rotation of a node with right child (Left Rotation)

```
RotateLeftWithRightChild(Z)
1. X = right[Z]
2. right[Z]= right[X]
3. left[X] = Z
4. height[Z] = Max(AVLHeight(left[Z) , AVLHeight(right[k])) + 1
5. height[X] = Max(AVLHeight(left[X) , AVLHeight(right[X])) + 1
6. Z = X
```

- Following lines may be inserted at the end, if Z is not the root

```
If P[Z] ≠NIL
        If Z = Left[P[Z]]
                Left[P[Z]] = X
        else,   right[P[Z]]= X
```

Case-3 : Double rotation

It requires two single rotations. First makes a right rotation at right child of unbalanced node. Then make a left rotation at unbalanced node.

```
DoubleRotateRightLeft(Z)
1. RotateRightWithLeftChild(right[Z])        //case-1
2. RotateLeftWithRightChild(Z)               //case-2
```

Case-4: Double rotation

It requires two single rotations. First makes a left rotation at left child of unbalanced node. Then make a right rotation at unbalanced node.

```
DoubleRotateLeftRight(Z)
1. RotateLeftWithRightChild(Left[Z])  //case-2
2. RotateRightWithLeftChild(Z)
```

Insertion of a node in an AVL Tree

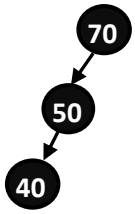
```
AVLInsert(T,x)
1. if T = NIL
2.      T = NewNode(X)
3.      return T  // Create a new node
3. elseif x< key[T]
4.      left[T] = AVLInsert(left[T],x)
6.      if ( Height(left[T]) – Height(right[T]) =2
9.           if ( x < key[left[T]])
10.                RotateRightWithLeftChild(T) // case-1
11.          else,   DoubleRotateRightLeft(T)      // case-4
12. else if x ≥ key[T]
13.     right[T] = AVLInsert(right[T],x)
15.     if ( Height(left[T]) – Height(right[T]) =-2
18.          if (x > key[left[T]])
19.                RotateLeftWithRightChild(T)   //case-2
20.          else,   DoubleRotateLeftAndRight(T)   //case-3
21. height[T] = Max(AVLHeight(left[T) , AVLHeight(right[T])) + 1
```

Red-Black Tree

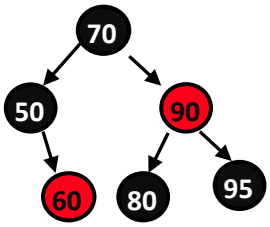
A red-black tree is a self-balancing search tree that ensures that no such path is more than twice as long as other is. It has following properties:

1. Every node is colored either **red** or **black**.
2. The root is **black**.
3. Every NIL leaf is **black**
4. If a node is **red**, its children are **black**. (No red-red combination)
5. Every path from a node to the descendent leaf(NIL leaf) contains same number of black nodes.

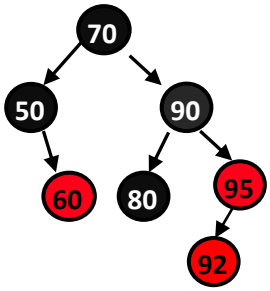
Example:



Not a Red-Black Tree.
Property-5 violation



A Red-Black Tree.



Not a Red-Black Tree.
Property-4 violation

Comparison to AVL Tree

1. Balancing condition is less restricted. So it is more preferable.
2. Useful for frequent insertion/deletion of nodes in tree.

Insertion into red-black tree

A node is inserted as red. Red-black property is violated if the parent of new node is **Red.** To make the tree red-black, follow the rules,

Situation -1 : if parent of new node (z) is a left child

Case-1 : If z's uncle is **red**, then apply rule-1.

Case-2 : If z's uncle is **black**, and z is a right child of its parent, then apply Rule-2 and Rule-3.

Case-3 : If there is no uncle or (uncle is **black**, and if z is not a right child of its parent,) then apply only Rule-3 .

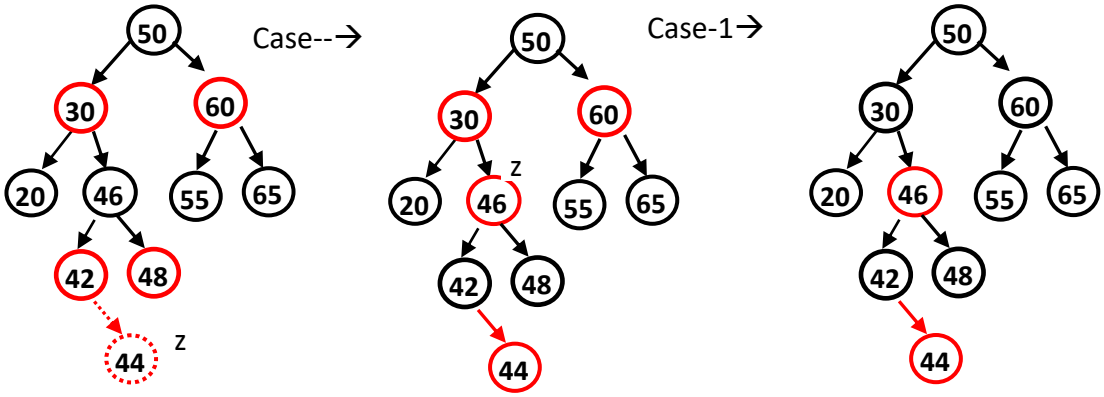
Situation- 2 : if parent of new node (z) is a right child, same cases and rules are applied with left<-> right exchange .

Rule-1*: Make both parent and sibling of parent as black and make grandparent as red. Apply this recursively by taking z =grandparent.*

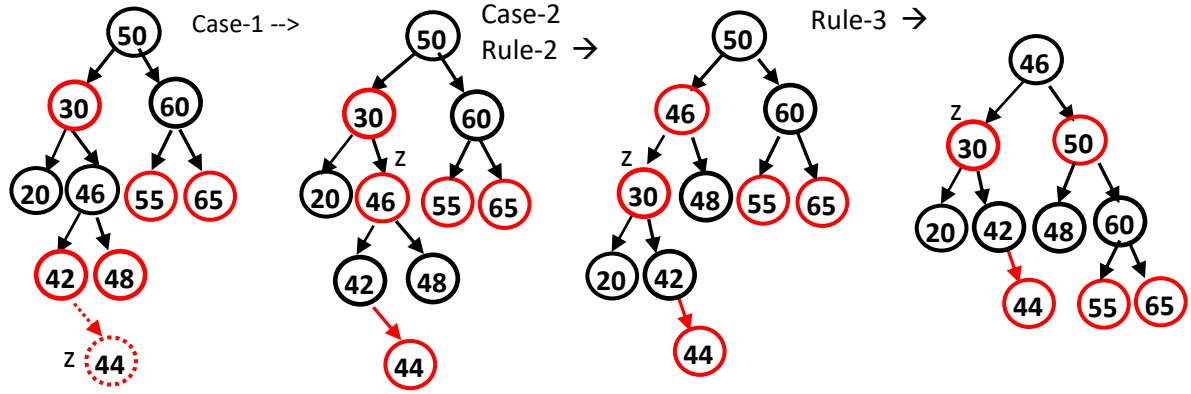
Rule-2: *Make z points to its parent (z=par[z]) . then make a left rotation at z.*

Rule-3: *Set color of parent of z to Black. Set color of grandparent of z to Red. then rotate right the tree at grandparent of z*

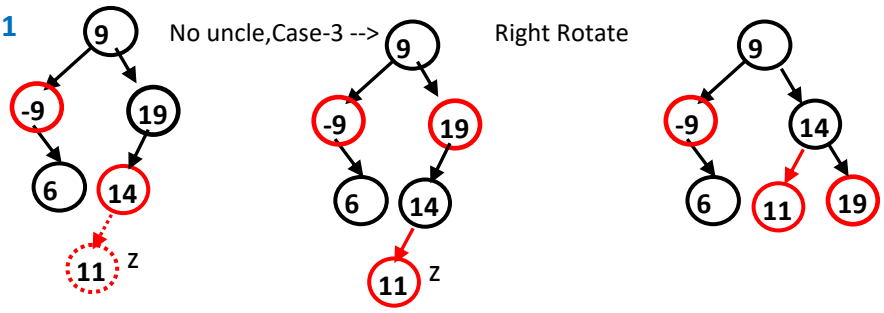
Example-1 **Insert 44**



Example-2 **Insert 44**



Example-3 **Insert 11**



Algorithm : Red Black Tree insertion

RBtreeInsert(T,z)

//Inserts a new node z into a red-black tree T.

```
1. BSTreeInsert(T,z)
2. color[z]= RED
3. while color[p[z]] = RED
4.        if p[z] = left[p[p[z]]]
5.                  u = right[p[p[z]]]
6.                  if color[u] = RED
7.                            color[p[z]]= BLACK
8.                            color[u] = BLACK
9.                            color[p[p[z]]]= RED
10.                           z = p[p[z]]
11.                 else
12.                           if z = right[p[z]]
13.                                     z = p[z]
14.                                     RotateLeft(T,z)
15.                           color[p[z]] = BLACK
16.                           color[p[p[z]]]= RED
17.                           RotateRight(T,p[p[z]])
18.       Else if  p[z] = Right[p[p[z]]]
19.                 u = left[p[p[z]]]
20.                 if color[u] = RED
21.                           color[p[z]]= BLACK
22.                           color[u] = BLACK
23.                           color[p[p[z]]]= RED
24.                           z = p[p[z]]
25.                 else
26.                           if z = left[p[z]]
27.                                     z = p[z]
28.                                     RotateLeft(T,z)
29.                           color[p[z]] = BLACK
30.                           color[p[p[z]]]= RED
31.                           RotateLeft(T,p[p[z]])
//end of loop
32.  color[Root[T]] = Black
```

Insert following nodes into Red Black Tree

10 20 -10 15 17 -4 50 60