UNIT-IV : EFFICIENT BINARY SEARCH TREES

Optimal Binary Search Trees, AVL Trees, Red-Black Trees, Definition- Representation of a Red- Black Tree- Searching a Red-Black Tree- Inserting into a Red Black Tree- Deletion from a Red-Black Tree- Joining Red-Black Trees, Splitting a Red-Black tree.

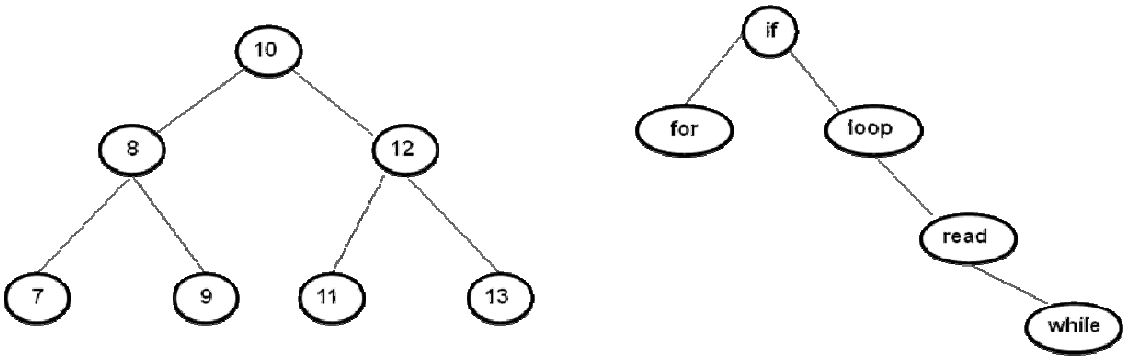
BINARY SEARCH TREE

A Binary search tree is a binary tree that is either empty or in which every node contains a key and satisfies the following conditions.

1. The key in the left child of a node (if it exists) is less than the key in its parent node.
2. The key in the right child of a node is greater than the key in its parent node.
3. The left and right subtrees of the root are again binary search trees.

No two entries in a binary search tree may have equal keys.

Ex) Binary Search Trees



(a) (b)

Figure 1) Binary Search Trees

If we want to search an element in binary search tree, first that element is compared with root node. If element is less than root node then search continue in left subtree. If element is greater than root node then search continue in right subtree. If element is equal to root node then print search was successful (element found) and terminate search procedure.

```
algorithm search(t,x)
{
   if  (t==0) then return 0;
   else
        if  (x = t->data) then return t;
        else
            if  ( x< t->data) then return search (t->left, x);
            else
                 return search (t->right, x);
}
```

OPTIMAL BINARY SEARCH TREE

Consider the two search trees in figure 2.

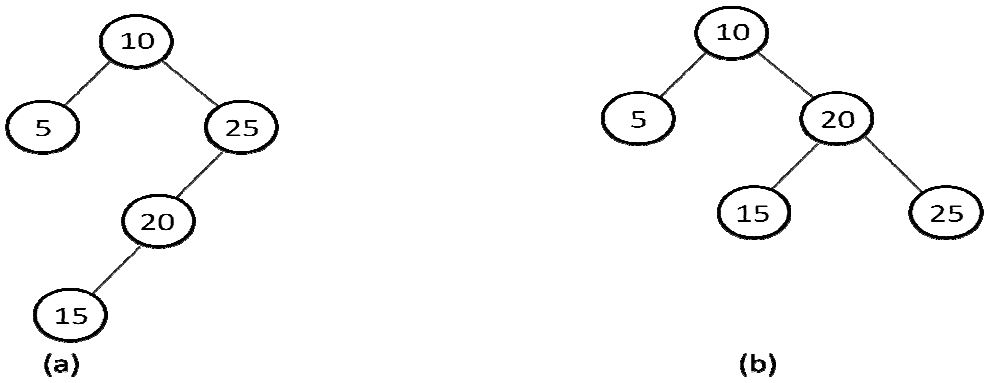


Figure 2) Two Binary search Trees

The second of tree requires at most three comparisons to decide whether the element being sought is in the tree.

The first binary tree may require at most 4 comparisons

As for as the worst case search time is concerned, the second binary tree is more desirable than the first.

To search for a key in figure (a) one comparison for 10 two comparisons for 5 and 25, three for 20 and four for 15. Assume that each key is searched for with equal probability. The average number of comparisons for a successful search is 2.4.

Total comparisons of figure (a) = 1+2+2+3+4=12
Average comparisons of figure (a) =12/5=2.4

Total comparisons of figure (b) = 1+2+2+3+3=11
Average comparisons of figure (b) =11/5=2.2

Thus the figure (b) has better average behavior.

In evaluating binary search trees, it is useful to add a special “square” node at every null link. Every binary tree with n nodes have n+1 null links. And therefore will have n+1 square nodes. We shall call these nodes external nodes. The remaining nodes are called internal nodes.

Each time a binary search tree is examined for an identifier that is not in the tree , the search terminates at external node. Since all such searches are unsuccessful searches external nodes will also referred as failure nodes. A binary tree with external nodes added is defined as extended binary tree.

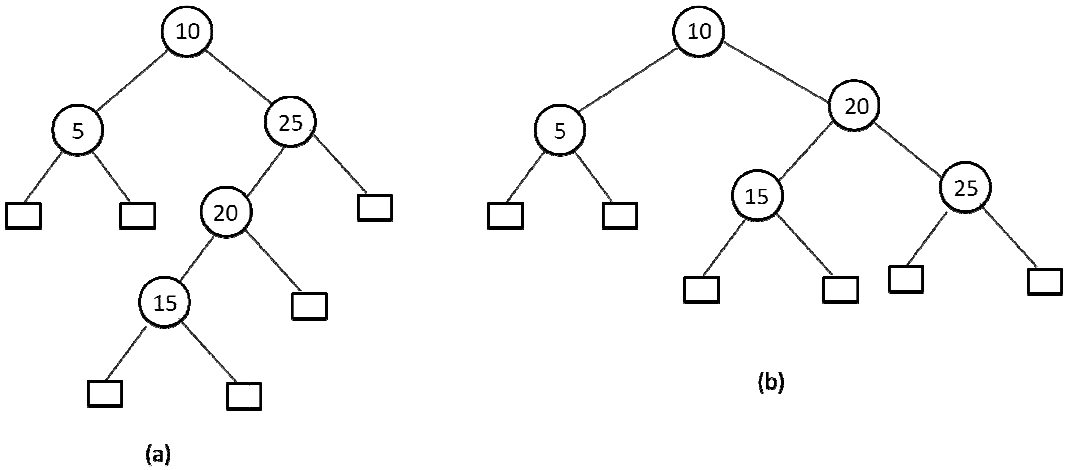


Figure 3) Extended Binary search trees

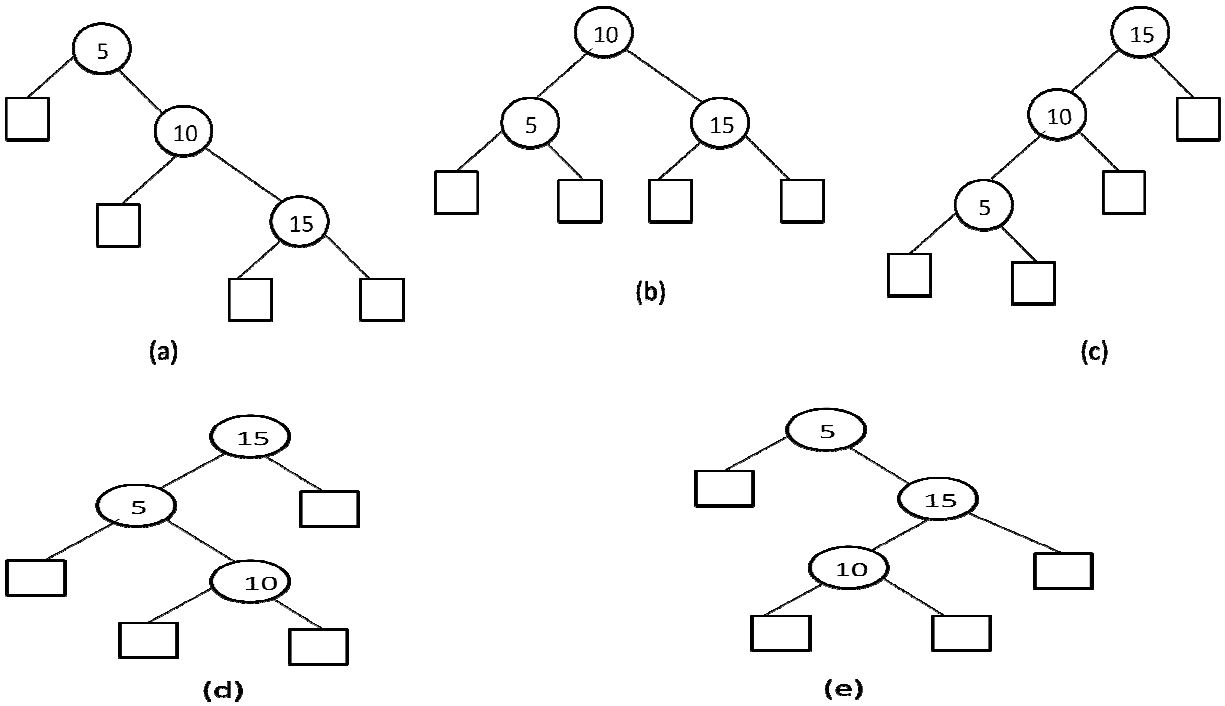
We define the external path length of a binary tree to be the sum over all external nodes of the lengths of the paths from the root to those nodes.

Internal path length is the sum over all internal nodes of lengths of the paths from the root to those nodes.

For figure (3a) Internal path length I = 0+1+1+2+3=7
External path length E = 2+2+4+4+3+2=17

For figure (3b) Internal path length I = 0+1+1+2+2=6
External path length E = 2+2+3+3+3+3=16

For Example the possible binary search trees for the identifier set (a1, a2, a3 = 5, 10, 15). The number of identifiers n=3.



For Example the possible binary search trees for the identifier set (a1, a2 ,a3 = do, if, stop). The number of identifiers n=3.

The number of possible binary search trees = $\frac{1}{n+1}{}^{2n}C_n = \frac{1}{3+1}{}^{2x3}C_3$

$= \frac{1}{4}{}^{6}C_3 = \frac{1}{4} \times \frac{1\times2\times3\times4\times5\times6}{1\times2\times3\times1\times2\times3} = \frac{1}{4} \times 20 = 5$

Figure 4) shows the possible BST for the key set (a1,a2,a3) = (do, if,stop)

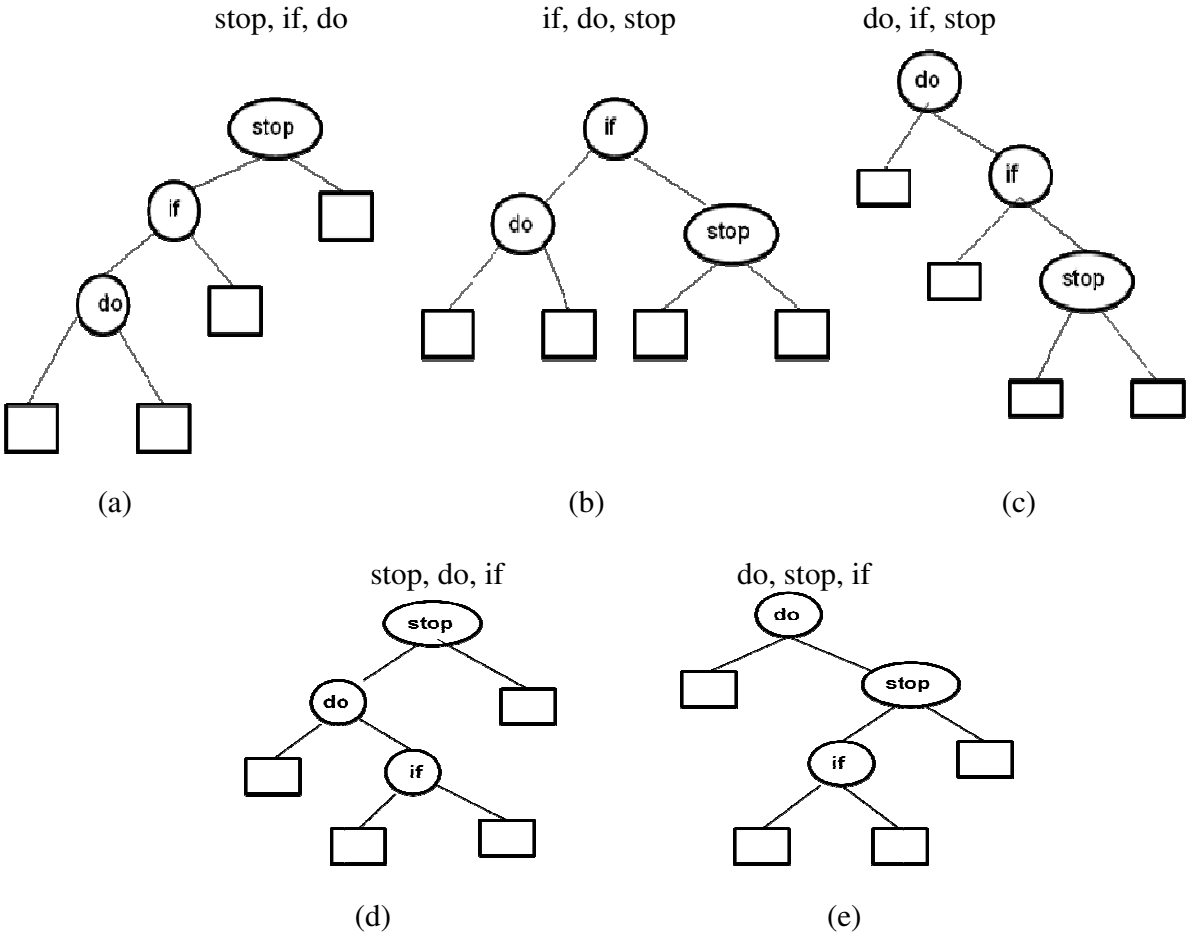


Figure 4) possible extended binary search trees with the given 3 identifiers

With equal probabilities pi=qi=1/7 for all i, j we have.

The cost of binary search tree for (a1, a2, a3) = $cost(T) = \sum_{1\le i\le n} p(i) \times level(a_i)$
Since unsuccessful searches will also be made, we should include cost of these searches in our cost measure. $\sum_{1\le i\le n} q(i) \times (level(\text{failure node i}) - 1)$

$$cost(T) = \sum_{1\le i\le n} p(i) \times level(a_i) + \sum_{0\le i\le n} q(i) \times (level(E_i) - 1)$$

$$p(i) = q(i) = \frac{1}{7}$$

$$a)\ 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{15}{7}$$

$$b)\ 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} + 2 \times \frac{1}{7} = \frac{13}{7}$$

$$c)\ 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{15}{7}$$

$$d)\ 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{15}{7}$$

$$e)\ 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 1 \times \frac{1}{7} + 2 \times \frac{1}{7} + 3 \times \frac{1}{7} + 3 \times \frac{1}{7} = \frac{15}{7}$$

In the above binary search tree the cost of the tree (b) is minimum. Hence it is optimal binary search tree.

If the probability p1=0.5 p2=0.1 p3=0.05 q0=0.15 q1=0.1 q2=0.05 and q3=0.05 we have

Cost of (a)=2.65.
Cost of (b)=1.9.
Cost of (c)=1.5.
Cost of (d)=2.05.
Cost of (e)=1.6.

Tree (c) is optimal with assignments of p's and q's.

A binary search tree have maximum of two Childs. i.e. 0, 1 or 2 Childs.

p(i) - probability of searching an internal node.

Q(i) – probability of searching an external node.

A successful search is terminated at an internal node denoted by circle (O) and unsuccessful search is terminated at an external node by square (□).

$$\text{cost}(T) = \sum_{1 \le i \le n} p(i) \times level(a_i) + \sum_{0 \le i \le n} q(i) \times (level(E_i) - 1)$$

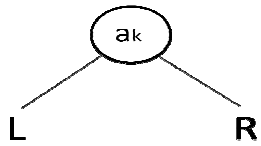


Figure (5) An optimal binary search tree T_{ij}

T_{ij} is an optimal binary search tree. T_{ij} has 2 subtrees L and R. L is left subtree and contains the keys $a_{i+1},..,a_{k-1}$ and R is right sub tree and contains the keys a_{k+1}, .., a_j.

The cost C_{ij} of T_{ij} is

$C_{ij} = P_k$ + cost(L) + cost(R)+weight(L)+Weight(R)
Where

w(i, j) = p(j) + q(j) + w(i, j-1) weight of t_{ij}

$c(i, j) = \min_{i<k\le j}\{c(i,k-1)+c(k, j)\}+w(i, j)$ cost of t_{ij}

r(i,j) = k Root of t_{ij}

Example 2) Let n=4 and (a1, a2, a3, a4) = (do, if , read, while) -- internal nodes
p(1:4)=(3,3,1,1) and
q(0:4)=(2,3,1,1,1)

External nodes (E_0, E_1, E_2, E_3, E_4)
As there are 4 internal nodes (a1, a2, a3, a4) = (do, if , read, while)

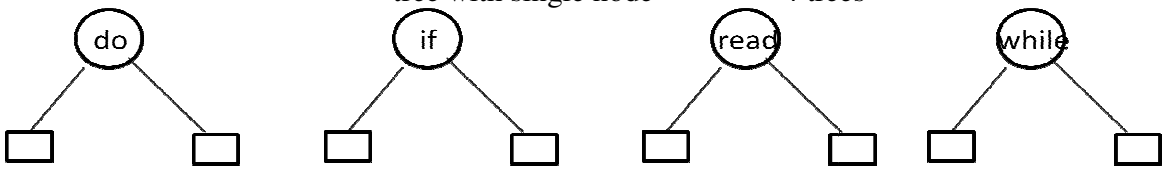
the number of possible binary search trees = $\frac{1}{n+1}{}^{2n}C_n$ = 14

w(i, j) = p(j) + q(j) + w(i, j-1) weight of t_{ij}

$c(i, j) = \min_{i<k\le j}\{c(i,k-1)+c(k, j)\}+w(i, j)$ cost of t_{ij}

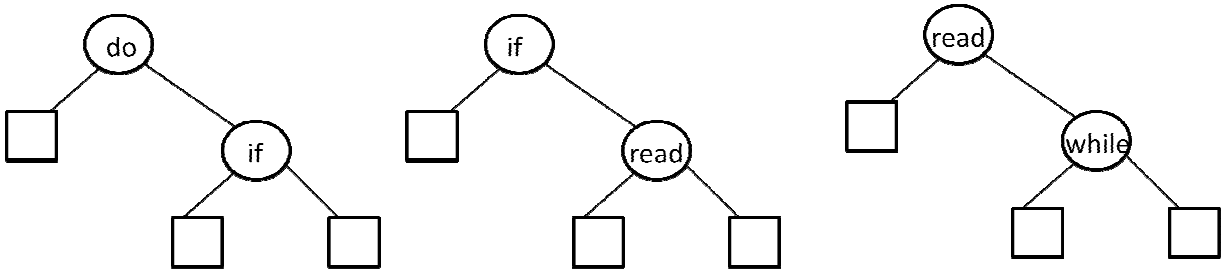
r(i,j) = k root of t_{ij}

---------- tree with single node ---------- 4 trees



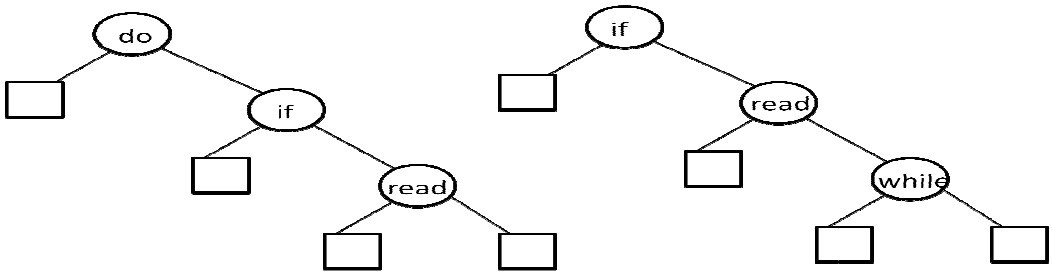
$W_{01} = E_0\, a_1\, E_1$
$W_{12} = E_1\, a_2\, E_2$
$W_{23} = E_2\, a_3\, E_3$
$W_{34} = E_3\, a_4\, E_4$

---------- tree with 2 nodes ---------- 3 trees



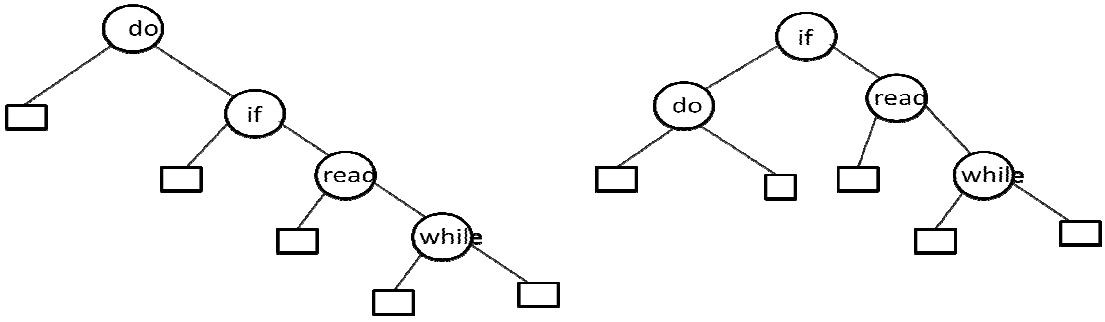
$W_{02} = E_0\, a_1\, E_1\, a_2\, E_2$
$W_{13} = E_1\, a_2\, E_2\, a_3\, E_3$
$W_{24} = E_2\, a_3\, E_3\, a_4\, E_4$

---------- tree with 3 nodes ---------- 2 trees



$W_{03} = E_0\, a_1\, E_1\, a_2\, E_2\, a_3\, E_3$
$W_{14} = E_1\, a_2\, E_2\, a_3\, E_3\, a_4\, E_4$

---------- tree with 4 nodes ---------- 1 tree



$W_{04} = E_0a_1E_1a_2E_2a_3E_3a_4E_4$

In order to solve above problem, first we will draw one table by taking 'I' corresponds to columns. The cells in the table can be indicated as $W_{j,j+i}$, $C_{j,j+i}$,$R_{j,j+i.}$

	0	1	2	3	4	
0	$W_{00}=2$ $C_{00}=0$ $R_{00}=0$	$W_{11}=3$ $C_{11}=0$ $R_{11}=0$	$W_{22}=1$ $C_{22}=0$ $R_{22}=0$	$W_{33}=1$ $C_{33}=0$ $R_{33}=0$	$W_{44}=1$ $C_{44}=0$ $R_{44}=0$	External nodes E0,E1,E2,E3,E4
1	$W_{01}=$ $C_{01}=$ $R_{01}=$	$W_{12}=$ $C_{12}=$ $R_{12}=$	$W_{23}=$ $C_{23}=$ $R_{23}=$	$W_{34}=$ $C_{34}=$ $R_{34}=$		do, if, read, while
2	$W_{02}=$ $C_{02}=$ $R_{02}=$	$W_{13}=$ $C_{13}=$ $R_{13}=$	$W_{24}=$ $C_{24}=$ $R_{24}=$			do, if if,read read,while
3	$W_{03}=$ $C_{03}=$ $R_{03}=$	$W_{14}=$ $C_{14}=$ $R_{14}=$				do,if,read if,read,while
4	$W_{04}=$ $C_{04}=$ $R_{04}=$					do,if,read,while

Principle of optimality was applied.
The first row is initiated as W(i,i)=q(i) R(i,i)=0 C(i,i)) = 0
It means

$W_{00} = q(0) = 2$ $C_{00} = R_{00} = 0$
$W_{11} = q(1) = 3$ $C_{11} = R_{11} = 0$
$W_{22} = q(2) = 1$ $C_{22} = R_{22} = 0$
$W_{33} = q(3) = 1$ $C_{33} = R_{33} = 0$
$W_{44} = q(4) = 1$ $C_{44} = R_{44} = 0$

The remaining values of cells can be calculated using equations as follows

w(i, j) = p(j) + q(j) + w(i, j-1)

---

w(0,1)=p(1) + q(1) + w(0,0) = 3+3+2 = 8
w(1,2) = p(2) + q(2) + w(1,1) = 3+1+3 = 7
w(2,3) = p(3) + q(3) + w(2,2) = 1+1+1 = 3
w(3,4) = p(4) + q(4) + w(3,3) = 1+1+1 = 3

---

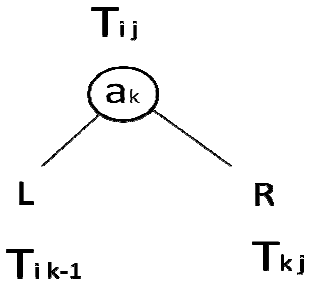
w(0,2) = p(2) + q(2) + w(0,1) = 3+1+8 = 12
w(1,3) = p(3) + q(3) + w(1,2) = 1+1+7 = 9
w(2,4) = p(4) + q(4) + w(2,3) = 1+1+3 = 5

---

w(0,3) = p(3) + q(3) + w(0,2) = 1+1+12 = 14
w(1,4) = p(4) + q(4) + w(1,3) = 1+1+9 = 11

---

w(0,4) = p(4) + q(4) + w(0,3) = 1+1+14 = 16



Cost = Cost of Left Tree + cost of Right Tree + weight of Root Node

$$c(i,j)=\min_{i<k\le j}\{c(i,k-1)+c(k,j)\}+w(i,j)$$

C(0,1) = min { c(0,0) + c(1,1) } + w(0,1) = 0 + 0 + 8 = 8 when k=1
R(0,1)=1

C(1,2) = min { c(1,1) + c(2,2) } + w(1,2) = 0 + 0 + 7 = 7 when k=2
R(1,2)=2

C(2,3) = min { c(2,2) + c(3,3) } + w(2,3) = 0 + 0 + 3 = 3 when k=3
R(2,3)=3

C(3,4) = min { c(3,3) + c(4,4) } + w(3,4) = 0 + 0 + 3 = 3 when k=4
R(3,4)=4

When k=2
C(0,2) = min { c(0,1) + c(2,2) } + w(0,2) = 8 + 0 + 12 = 20

When k=1
C(0,2) = min { c(0,0) + c(1,2) } + w(0,2) = 0 + 7 + 12 = 19

C(0,2) = min{20,19} = 19
r(0, 2) = 1

When k=3 or 2
C(1,3) = min { c(1,1) + c(2,3), c(1,2) + c(3,3) } + w(1,3) ={ 0 + 3, 7 + 0} + 9 = 12
r(1, 3) = 2

When k=3
C(2,4) = min { c(2,2) + c(3,4) } + w(2,4) = 0 + 3 + 5 = 8

When k=4
C(2,4) = min { c(2,3) + c(4,4) } + w(2,4) = 3 + 0 + 5 = 8

r(2,4)=3

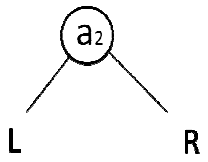
When k = 1 or 2 or 3
C(0,3) = min { c(0,0) + c(1,3), c(0,1) + c(2,3), c(0,2)+c(3,3) } + w(0,3)
= min{12,11,19} + 14
= 11 + 14 = 25
r(0,3)=2

When k = 2 or 3 or 4
C(1,4) = min { c(1,1) + c(2,4), c(1,2) + c(3,4), c(1,3)+c(4,4) } + w(1,4)
= min {0+8, 7+3, 12+0} + 11
= 8 + 11 = 19
r(1,4)=2

When k = 1 or 2 or 3 or 4
C(0,4) = min { c(0,0) + c(1,4), c(0,1) + c(2,4), c(0,2)+c(3,4), c(0,3)+c(4,4) } + w(0,4)
= min{19, 16, 23, 25} + 16
= 16 + 16 = 32
R(0,4)=2

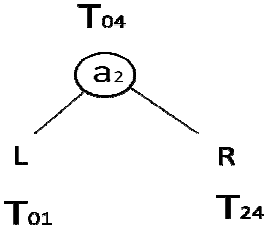
	0	1	2	3	4
0	$W_{00}=2$ $C_{00}=0$ $R_{00}=0$	$W_{11}=3$ $C_{11}=0$ $R_{11}=0$	$W_{22}=1$ $C_{22}=0$ $R_{22}=0$	$W_{33}=1$ $C_{33}=0$ $R_{33}=0$	$W_{44}=1$ $C_{44}=0$ $R_{44}=0$
1	$W_{01}=8$ $C_{01}=8$ $R_{01}=1$	$W_{12}=7$ $C_{12}=7$ $R_{12}=2$	$W_{23}=3$ $C_{23}=3$ $R_{23}=3$	$W_{34}=3$ $C_{34}=3$ $R_{34}=4$	
2	$W_{02}=12$ $C_{02}=19$ $R_{02}=1$	$W_{13}=9$ $C_{13}=12$ $R_{13}=2$	$W_{24}=5$ $C_{24}=8$ $R_{24}=3$		
3	$W_{03}=14$ $C_{03}=25$ $R_{03}=2$	$W_{14}=11$ $C_{14}=19$ $R_{14}=2$			
4	$W_{04}=16$ $C_{04}=32$ $R_{04}=2$				

Now observe the tables last cell i.e. 4th row 0th column, contains $r_{04} = 2$ i.e. $r_{04} = a_2$ (2 corresponds to second node a_2).



R04=k, then k=2
To build OBST R(0,4)=2=k=2

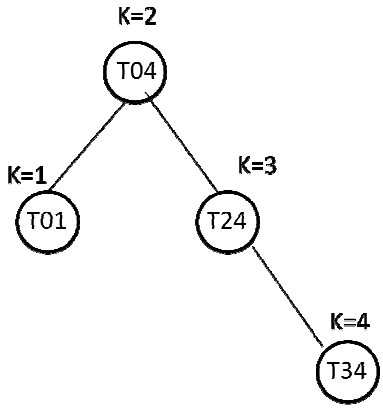
Hence a_2 become the root node
Let T be OBST $T_{i,j} = T_{i,k-1}$, and $T_{k,j}$

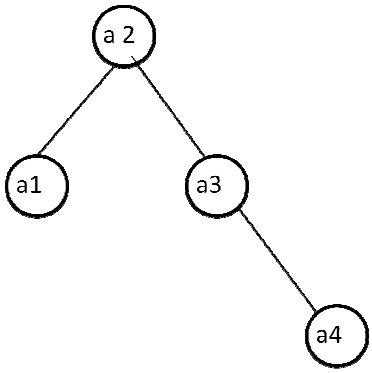


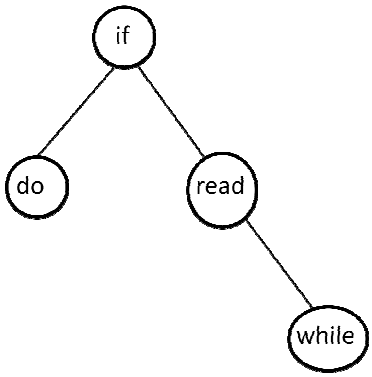
T_{04} is divided into two parts i.e. T01 and T24
T_{01}=r(0,1)=1=k=1
T_{24}=r(2,4)=3=k=3

T_{01} is divided into two parts T_{00} and T_{11} where k=1
T_{24} is divided into two parts T_{22} and T_{34} where k=3
T_{34} is divided into two parts T_{33} and T_{44} where k=4

Since r00, r11, r22, r33, r44 = 0 these are external nodes and can be neglected.
Let T be the optimal binary search tree



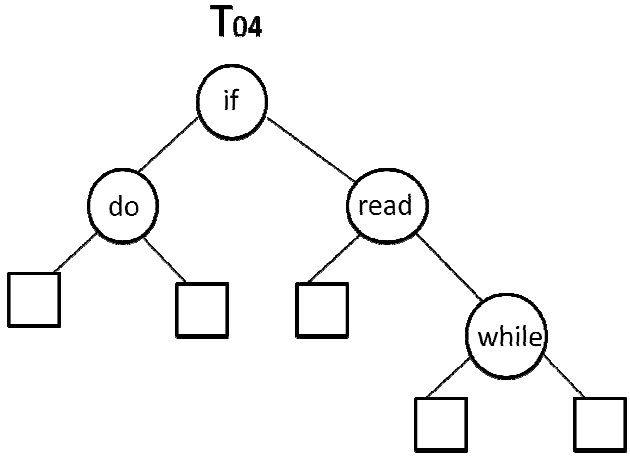





```
Algorithm OBST(p,q,n)
{
  for i:=0 to n-1 do
  {
     w[i,i]:=q[i];
      r[i,i]:=0;
      c[i,i]:=0;
      w[i,i+1]:=q[i]+q[i+1]+p[i+1];
      r[i,i+1]:=i+1;
     c[i,i+1]=q[i]+q[i+1]+p[i+1];
   }
     w[n,n]:=q[n];
      r[n,n]:=0;
      c[n,n]:=0.0;

    for m:=2 to n do
    for i:=0 to n-m do
     {
         j:=i+m;
         w[i, j]:=w[i, j-1]+p[j]+q[j];
         k:=find(c, r, i, j);
         c[i, j]:=w[i, j]+c[i, k-1]+c[k, j];
         r[i,j]:=k;
     }
         write(c[0,n],w[0,n],r[0,n];
}

algorithm find(c, r, i, j)
{
    min := ∞;
    for m := r[i, j-1] to r[i+1, j] do
        if ((c[i, m-1]+c[m, j]) < min) then
         {
              min:=c[i, m-1]+c[m, j];
             l:=m;
         }
     return l;
}
```



Time Complexity O(n log(n))

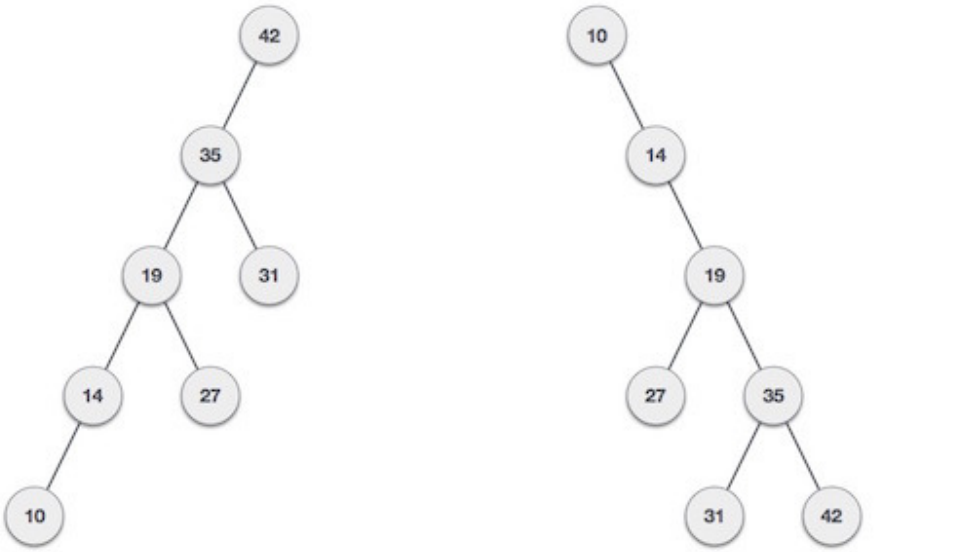
AVL Trees

Tree is one of the most important data structure that is used for efficiently performing operations like insertion, deletion and searching of values. However, while working with a large volume of data, construction of a well-balanced tree for sorting all data is not feasible. Thus only useful data is stored as a tree, and the actual volume of data being used continually changes through the insertion of new data and deletion of existing data. You will find in some cases where the NULL link to a binary tree to special links is called as threads and hence it is possible to perform traversals, insertions, deletions without using either stack or recursion. In this chapter, you will learn about the Height balance tree which is also known as the AVL tree.

What is AVL Tree:

AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson, Velsky, and Landis and hence given the short form as AVL tree or Balanced Binary Tree.

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



If input 'appears' non-increasing manner

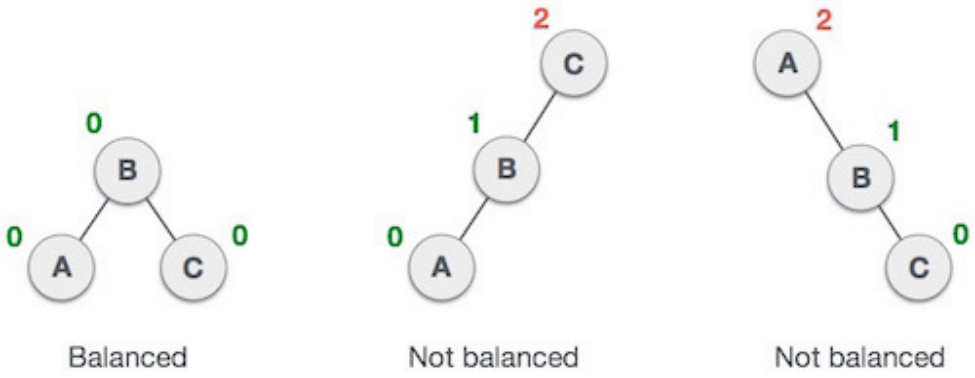
If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velsky** and **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1.

This difference is called the **Balance Factor**.

Consider the following trees. Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-subtree) – height(right-subtree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

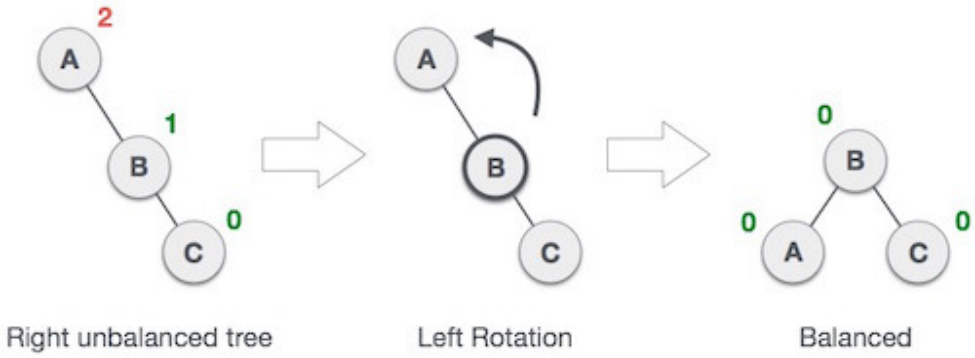
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

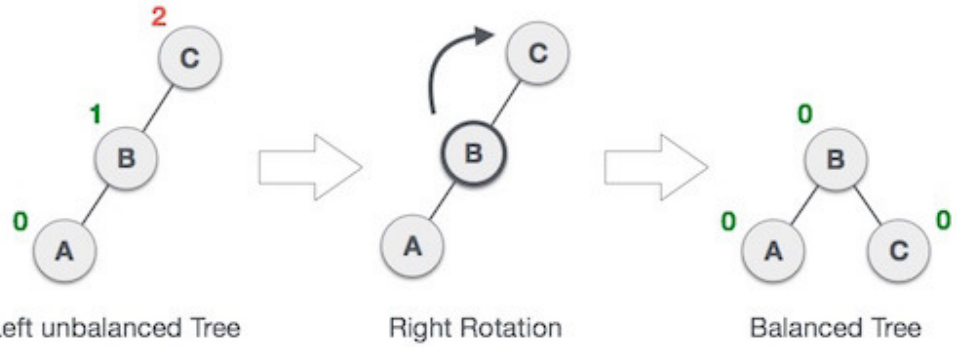
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

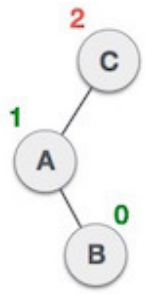
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

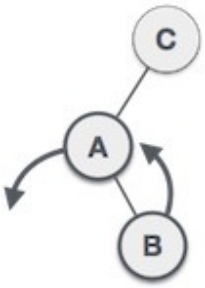
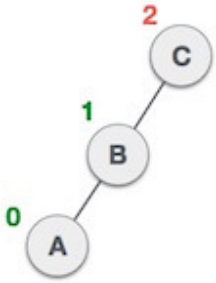
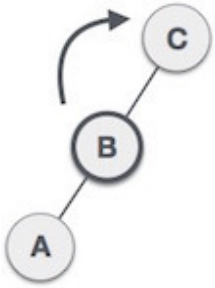
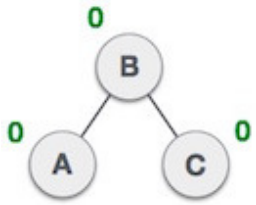


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

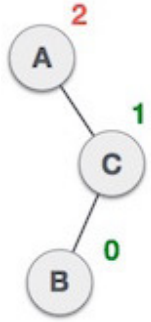
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

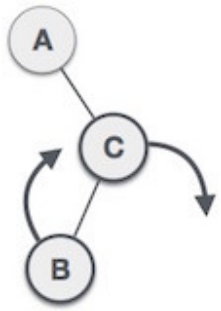
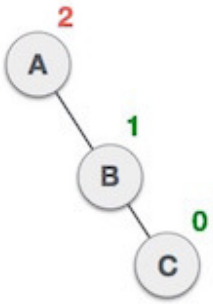
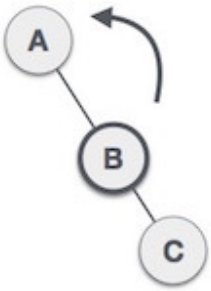
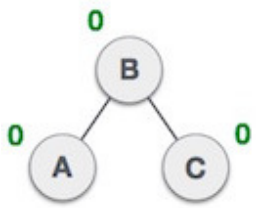
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.

	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

An AVL tree can be defined as follows:

Let T be a non-empty binary tree with T_L and T_R as its left and right subtrees. The tree is height balanced if:

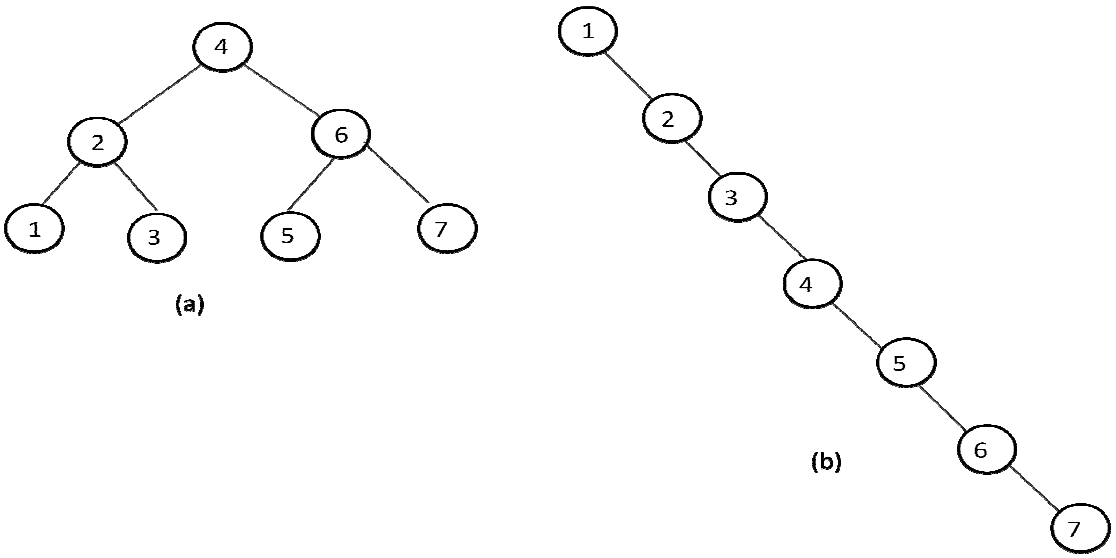
- T_L and T_R are height balanced
- h_L - h_R <= 1, where h_L - h_R are the heights of T_L and T_R

The Balance factor of a node in a binary tree can have value 1, -1, 0, depending on whether the height of its left subtree is greater, less than or equal to the height of the right subtree.

Advantages of AVL tree

Since AVL trees are height balance trees, operations like insertion and deletion have low time complexity. Let us consider an example:

If you have the following tree having keys 1, 2, 3, 4, 5, 6, 7 and then the binary search tree will be like the second figure:



To insert a node with a key Q in the binary tree, the algorithm requires seven comparisons, but if you insert the same key in AVL tree, from the above 1st figure, you can see that the algorithm will require three comparisons.

Representation of AVL Trees

```
Struct AVLNode
{
   int data;
   struct AVLNode *left, *right;
   int balfactor;
};
```

Algorithm for Insertion

Step 1: First, insert a new element into the tree using BST's (Binary Search Tree) insertion logic.

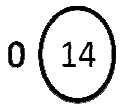
Step 2: After inserting the elements you have to check the Balance Factor of each node.

Step 3: When the Balance Factor of every node will be found like 0 or 1 or -1 then the algorithm will proceed for the next operation.

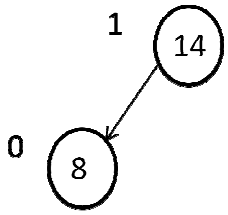
Step 4: When the balance factor of any node comes other than the above three values then the tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced and then the algorithm will proceed for the next operation.

Ex 1) Construct AVL Tree for the following numbers 14, 8, 12, 46, 23, 5, 77, 88, 20

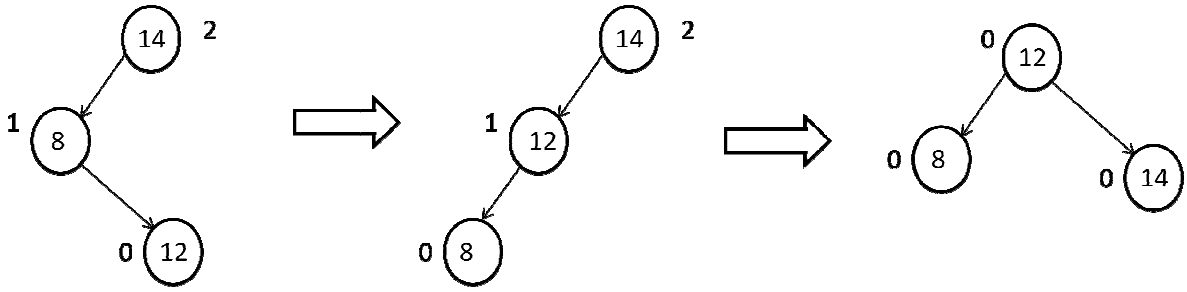
Insert (14)



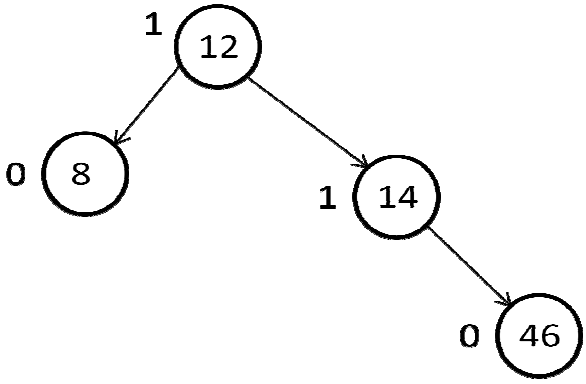
Insert (8)



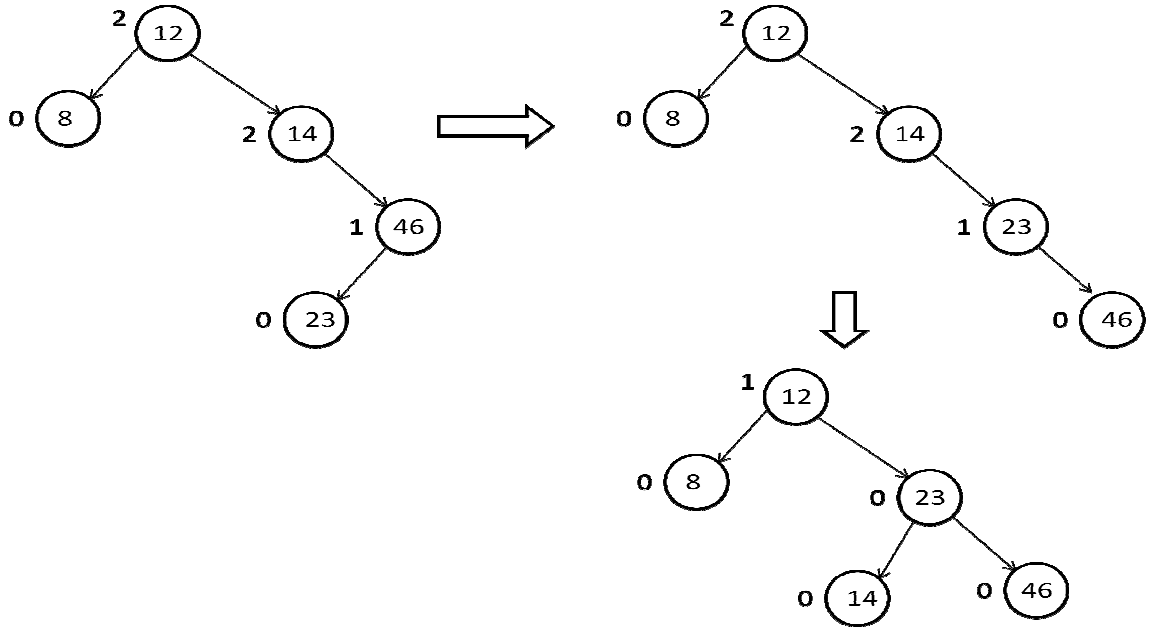
Insert (12)



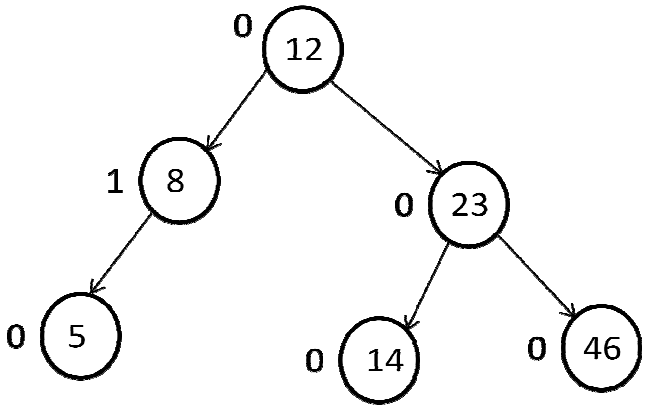
Insert (46)



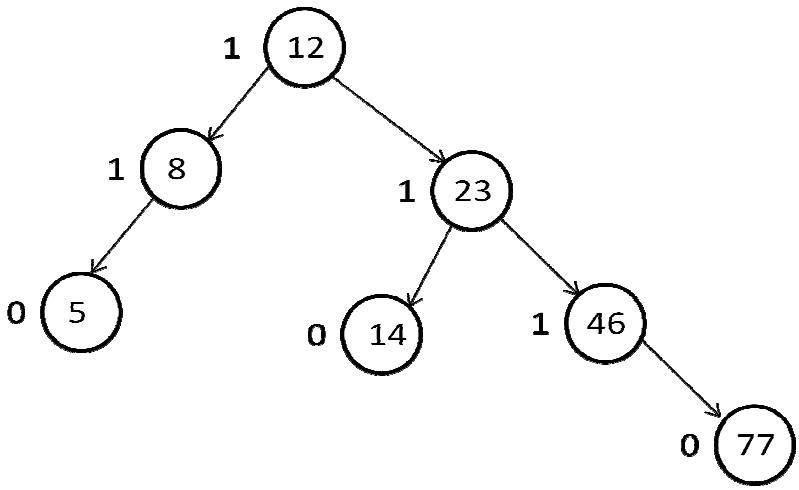
Insert (23)



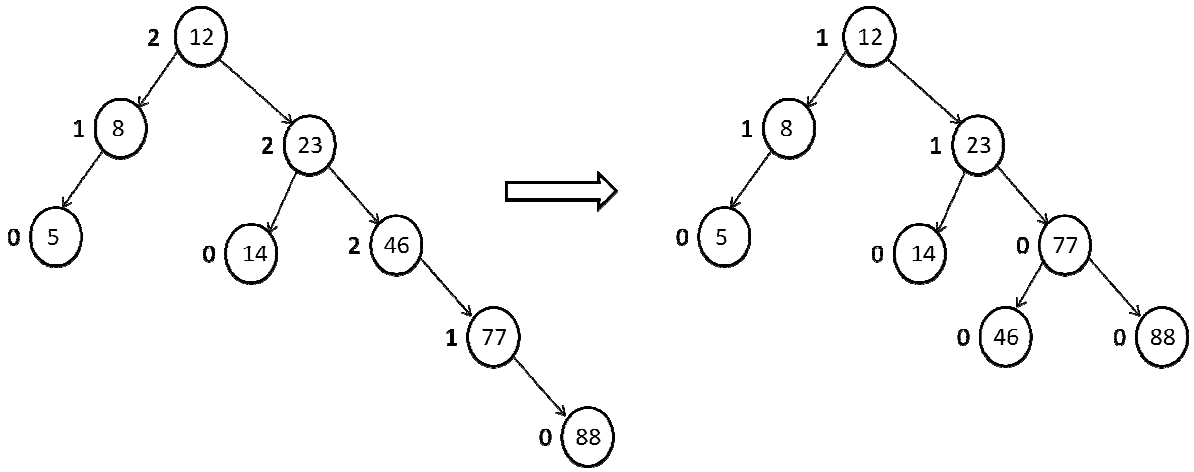
Insert (5)



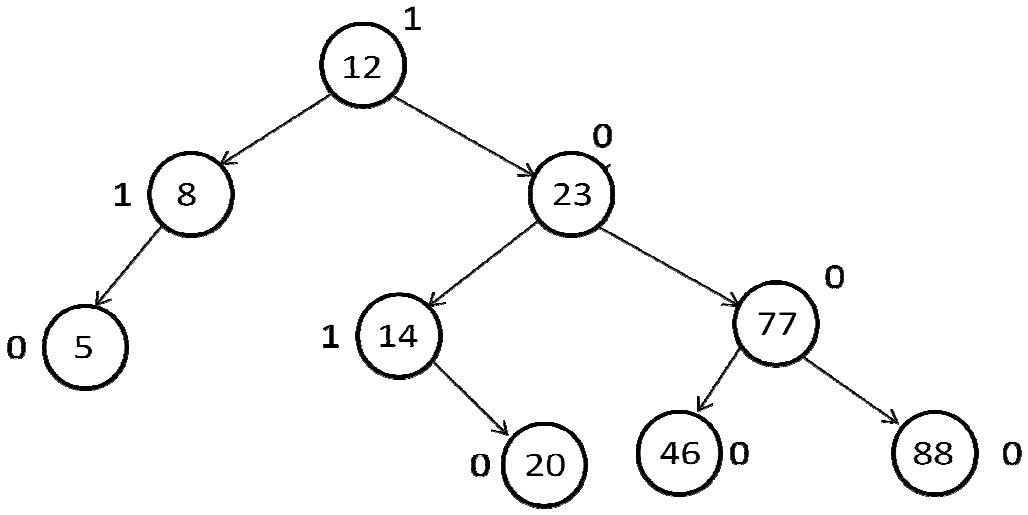
Insert (77)



Insert (88)

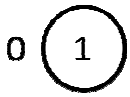


Insert (20)

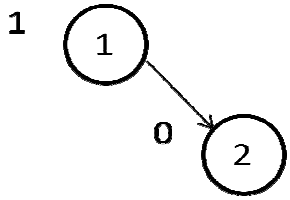


Ex 2) Construct AVL Tree for the following numbers 1, 2, 3, 4, 8, 7, 6, 5, 11, 10, 12

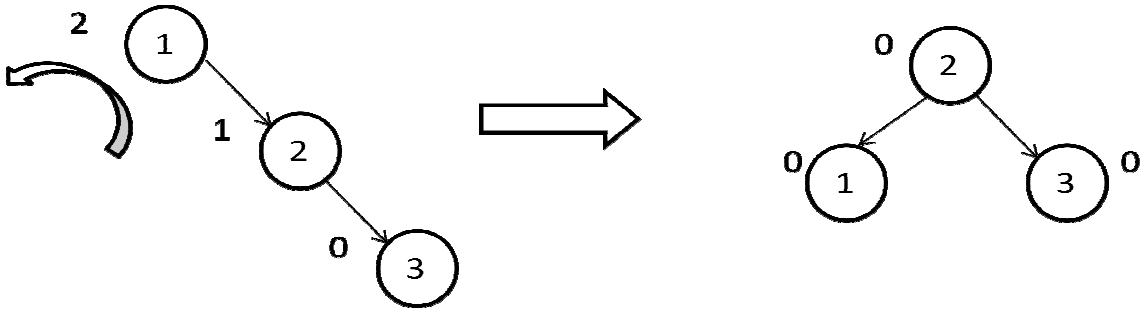
Insert (1)



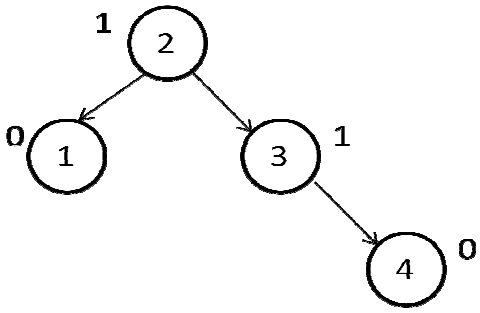
Insert (2)



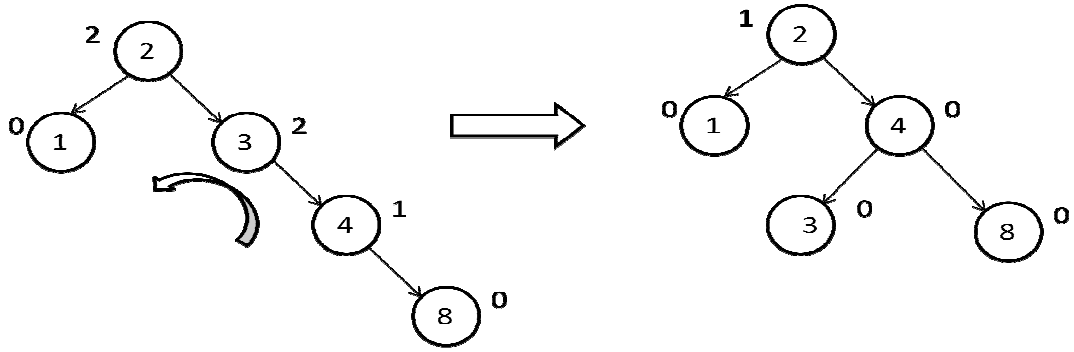
Insert (3)



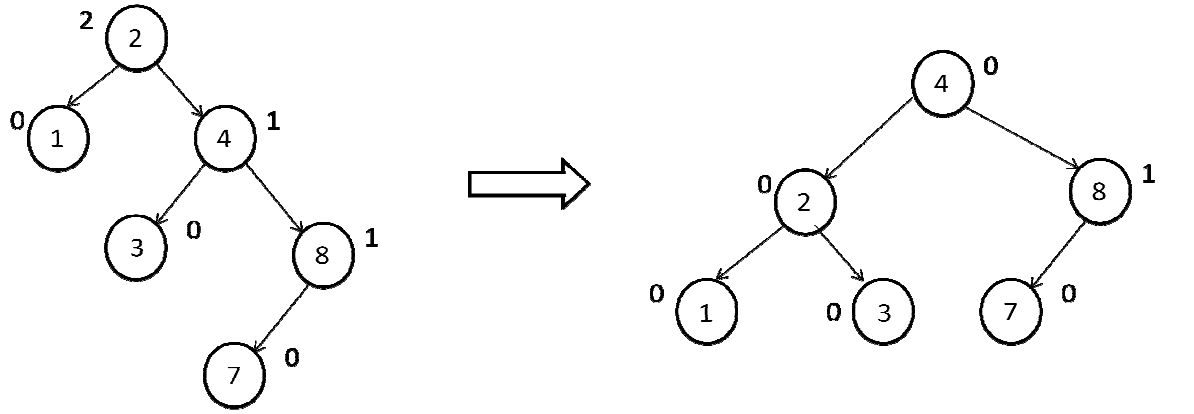
Insert (4)



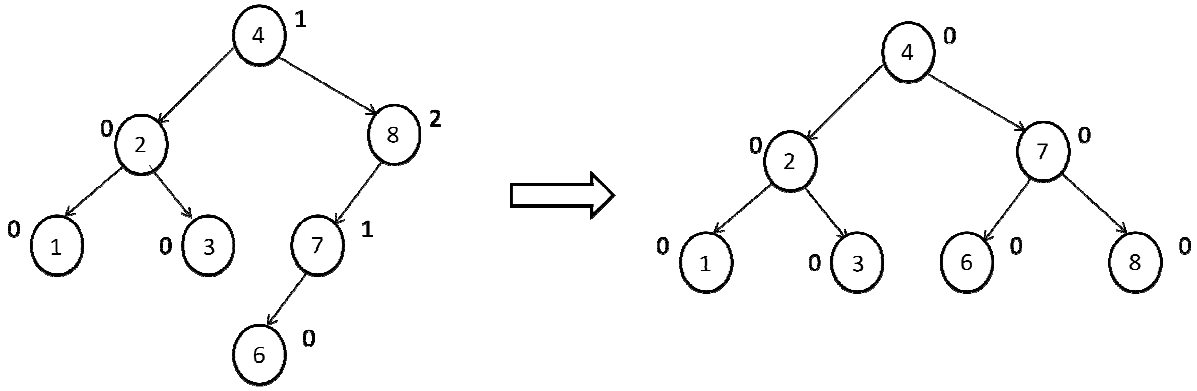
Insert (8)



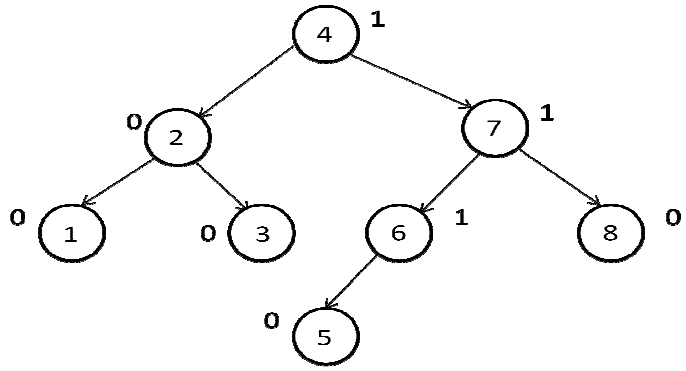
Insert (7)



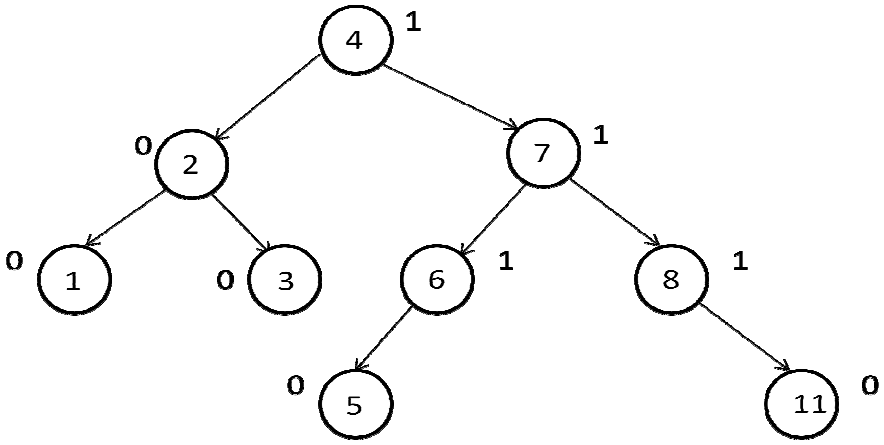
Insert (6)



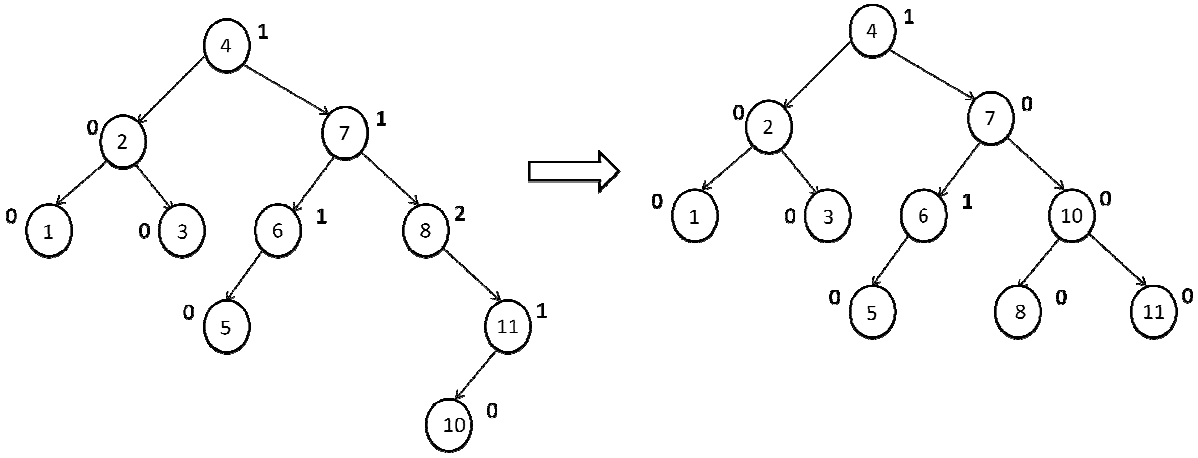
Insert (5)



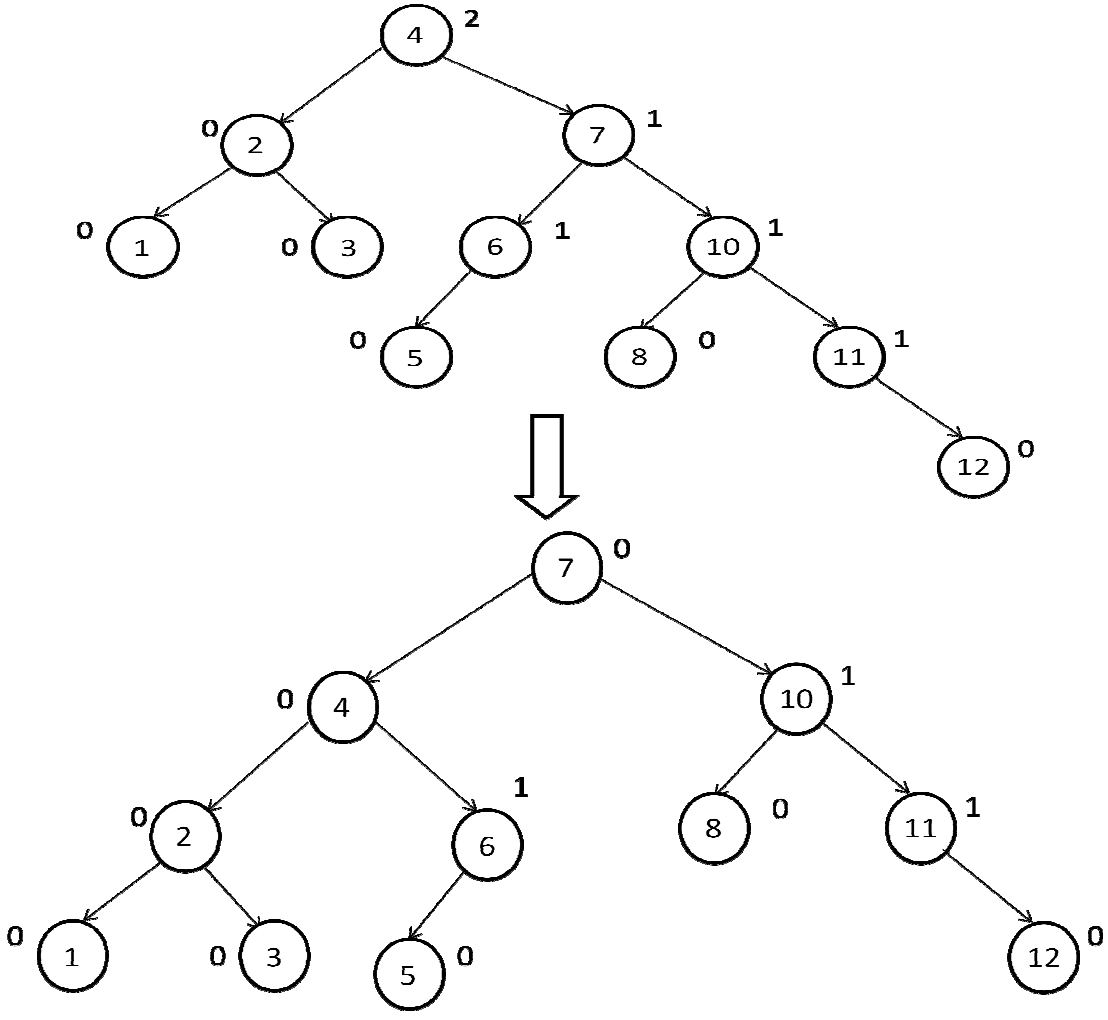
Insert (11)



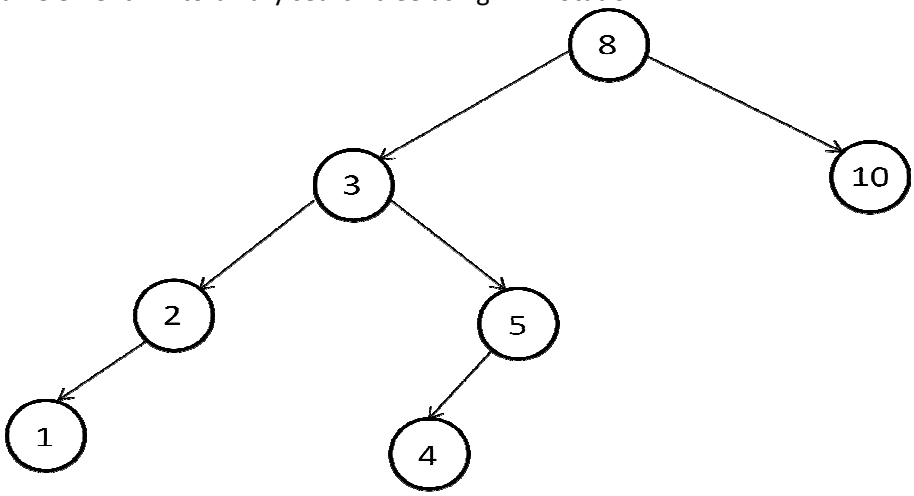
Insert (10)



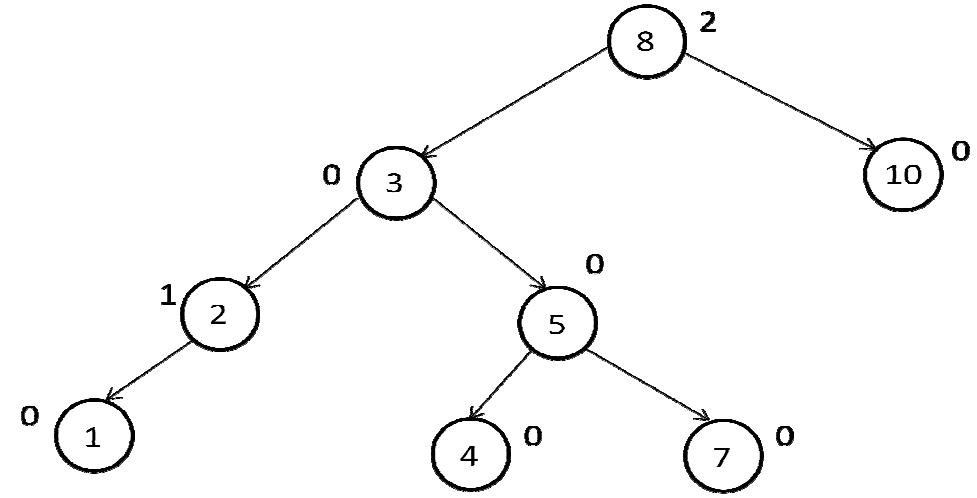
Insert (12)

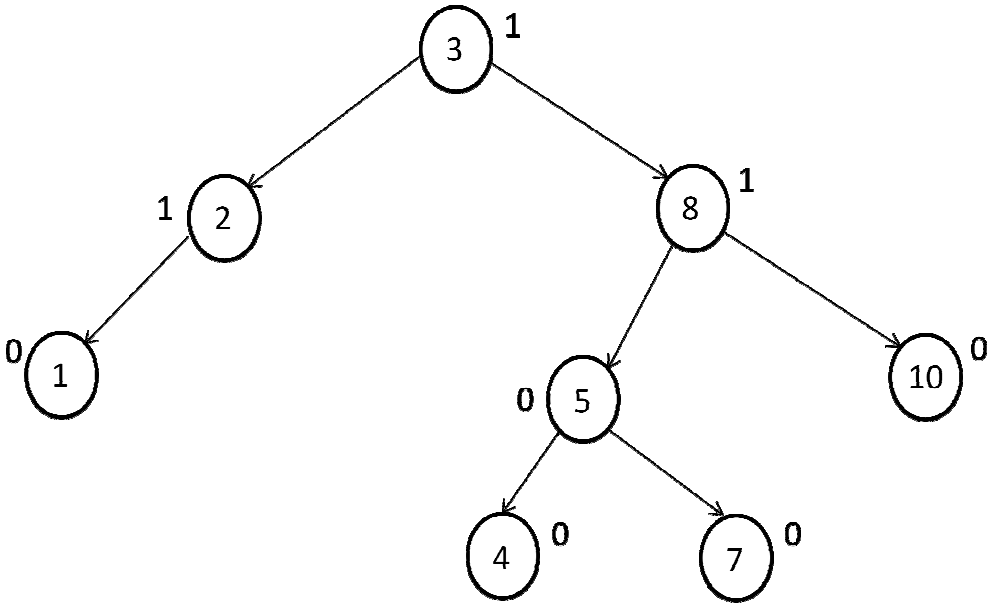


EX 3) Construct a binary search tree for data 8, 10, 3, 2, 1, 5, 4 ,6
Insert an element 7 into binary search tree using AVL rotation.



Insert 7





Algorithm for Deletion

We will try to understand this algorithm using an example but before that let's go over the major steps of this algorithm. Note that this algorithm is a bottom-up algorithm and hence height restoration of the tree proceeds from leaves to root.

This algorithm is basically a modification of the usual BST deletion algorithm.

The steps of this algorithm are :

1. Use general BST deletion algorithm to delete given key from the AVL tree.

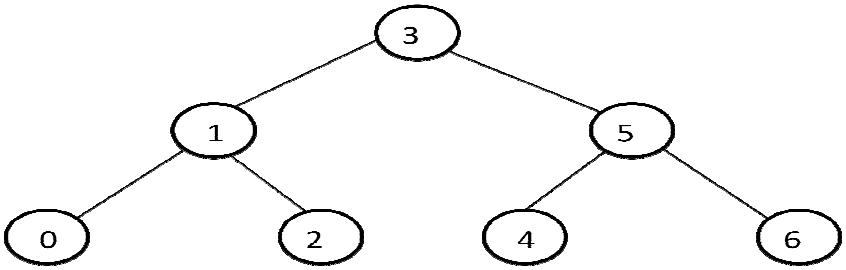
2. After deletion, update the height of the current node of the AVL tree.

3. Now check the 'balance' at the current node by getting the difference of height of left sub-tree and height of right sub-tree.

3a. If 'balance' > 1 then that means the height of the left sub-tree is greater than the height of the right sub-tree. This indicates left-left or left-right case(discussed in the previous post). To confirm if this is left-left or left-right case, we check the balance of left sub-tree of current node. If it greater than 0 then that confirms left-left case, if it less than 0 then that confirms left-right case. If it is equal to 0, then this we can either consider this as a left-left case or as a left-right case. In this implementation, we consider this as left-left case for efficiency. For left-left case, we do a right rotation at the current node. For the left-right case, we do a left rotation at left child of current node followed by a right rotation at the current node itself.

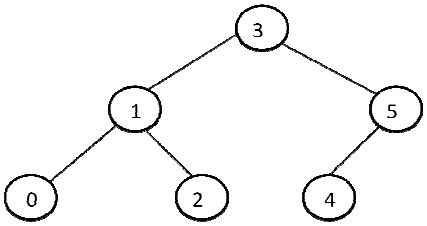
3b. If 'balance' < -1 then that means the height of the right sub-tree is greater than the height of the left sub-tree. This indicates right-right or right-left case(discussed in the previous post). In this case, if balance of right sub-tree of current node is less than 0 then this confirms right-right case, if it is greater than 0 then this confirms right-left case. If it is equal to 0, then we can either consider this as a right-right case or a right-left case. In this implementation, we will consider this as a right-right case. In right-right case, we do a left rotation at the current node. In right-left case, we do a right rotation at the right child of current node followed by left rotation at the current node itself.

Example walk-through: Let's delete key sequence [6,5,4] from the below AVL tree and see how the height balance is maintained throughout.

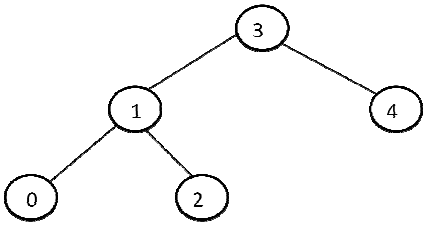


Delete keys 6,5 and 4:

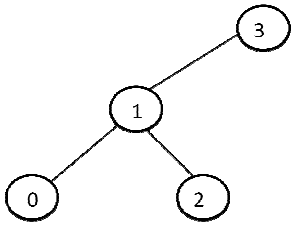
Delete (6)



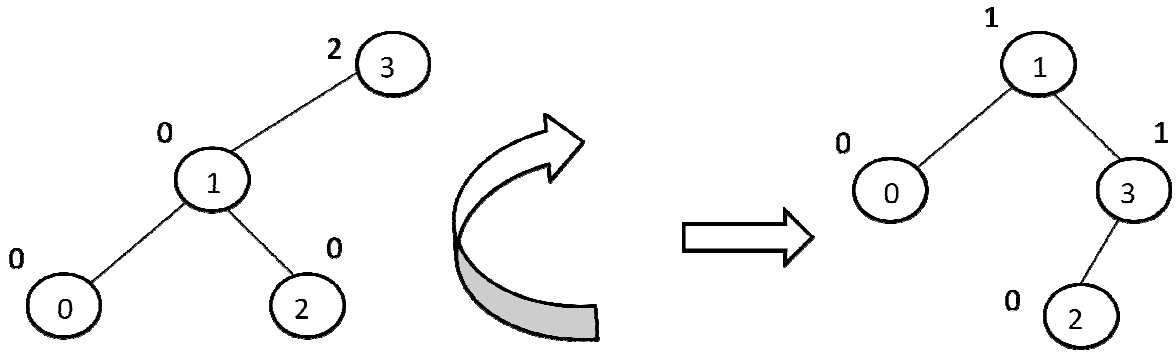
Delete (5)



Delete (4)



At this point, there is a height imbalance at node 3 with the left-left case. We do a right rotation at node 3.



As you can confirm this tree satisfies height balance property for AVL tree.

Program for AVL Tree in C

```
//-------------------------------------------------------------------------
// Program to create, insert, delete and print AVL tree
//-------------------------------------------------------------------------
#include<stdio.h>
typedef struct node
{
        int data;
        struct node *left,*right;
        int ht;
}node;

node *insert(node *,int);
node *Delete(node *,int);
void preorder(node *);
void inorder(node *);
int height( node *);
node *rotateright(node *);
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);
int BF(node *);

int main()
{
        node *root=NULL;
        int x,n,i,op;

        do
        {
                printf("\n1)Create:");
                printf("\n2)Insert:");
                printf("\n3)Delete:");
                printf("\n4)Print:");
                printf("\n5)Quit:");
                printf("\n\nEnter Your Choice:");
                scanf("%d",&op);

                switch(op)
                {
                        case 1: printf("\nEnter no. of elements:");
                                scanf("%d",&n);
                                printf("\nEnter tree data:");
                                root=NULL;
                                for(i=0;i<n;i++)
                                {
                                        scanf("%d",&x);
                                        root=insert(root,x);
                                }
                                break;
```

```
                        case 2: printf("\nEnter a data:");
                                scanf("%d",&x);
                                root=insert(root,x);
                                break;

                        case 3: printf("\nEnter a data:");
                                scanf("%d",&x);
                                root=Delete(root,x);
                                break;

                        case 4: printf("\nPreorder sequence:\n");
                                preorder(root);
                                printf("\n\nInorder sequence:\n");
                                inorder(root);
                                printf("\n");
                                break;
                }
        }while(op!=5);

        return 0;
}

node * insert(node *T,int x)
{
        if(T==NULL)
        {
                T=(node*)malloc(sizeof(node));
                T->data=x;
                T->left=NULL;
                T->right=NULL;
        }
        else
                if(x > T->data)                 // insert in right subtree
                {
                        T->right=insert(T->right,x);
                        if(BF(T)==-2)
                                if(x>T->right->data)
                                        T=RR(T);
                                else
                                        T=RL(T);
                }
                else
                        if(x<T->data)
                        {
                                T->left=insert(T->left,x);
                                if(BF(T)==2)
                                        if(x < T->left->data)
                                                T=LL(T);
                                        else
                                                T=LR(T);
                        }
```

```
                T->ht=height(T);
                return(T);
}

node * Delete(node *T,int x)
{
        node *p;

        if(T==NULL)
        {
                return NULL;
        }
        else
                if(x > T->data)                 // insert in right subtree
                {
                        T->right=Delete(T->right,x);
                        if(BF(T)==2)
                                if(BF(T->left)>=0)
                                        T=LL(T);
                                else
                                        T=LR(T);
                }
                else
                        if(x<T->data)
                        {
                                T->left=Delete(T->left,x);
                                if(BF(T)==-2) //Rebalance during windup
                                        if(BF(T->right)<=0)
                                                T=RR(T);
                                        else
                                                T=RL(T);
                        }
                        else
                        {                                       //data to be deleted is found
                                if(T->right!=NULL)
                                {       //delete its inorder succesor
                                        p=T->right;

                                        while(p->left!= NULL)
                                                p=p->left;
                                                T->data=p->data;
                                                T->right=Delete(T->right,p->data);
                                        if(BF(T)==2)//Rebalance during windup
                                                if(BF(T->left)>=0)
                                                        T=LL(T);
                                                else
                                                        T=LR(T);\
                                }
                                else
                                        return(T->left);
                        }
                T->ht=height(T);
                 return(T);
```

```
}

        int height(node *T)
        {
                int lh,rh;
                if(T==NULL)
                        return(0);

                if(T->left==NULL)
                        lh=0;
                else
                        lh=1+T->left->ht;

                if(T->right==NULL)
                        rh=0;
                else
                        rh=1+T->right->ht;

                if(lh>rh)
                        return(lh);
                return(rh);
        }

        node * rotateright(node *x)
        {
                node *y;
                y=x->left;
                x->left=y->right;
                y->right=x;
                x->ht=height(x);
                y->ht=height(y);
                return(y);
        }

        node * rotateleft(node *x)
        {
                node *y;
                y=x->right;
                x->right=y->left;
                y->left=x;
                x->ht=height(x);
                y->ht=height(y);

                return(y);
        }


        node * RR(node *T)
        {
                T=rotateleft(T);
                return(T);
        }
```

```
node * LL(node *T)
{
        T=rotateright(T);
        return(T);
}

node * LR(node *T)
{
        T->left=rotateleft(T->left);
        T=rotateright(T);

        return(T);
}

node * RL(node *T)
{
        T->right=rotateright(T->right);
        T=rotateleft(T);
        return(T);
}

int BF(node *T)
{
        int lh,rh;
        if(T==NULL)
                return(0);

        if(T->left==NULL)
                lh=0;
        else
                lh=1+T->left->ht;

        if(T->right==NULL)
                rh=0;
        else
                rh=1+T->right->ht;

        return(lh-rh);
}

void preorder(node *T)
{
        if(T!=NULL)
        {
                printf("%d(Bf=%d)",T->data,BF(T));
                preorder(T->left);
                preorder(T->right);
        }
}

void inorder(node *T)
{
        if(T!=NULL)
```

```
        {
                inorder(T->left);
                printf("%d(Bf=%d)",T->data,BF(T));
                inorder(T->right);
        }
}
```

Output

```
1)Create:2)Insert:3)Delete:4)Print:5)Quit:
Enter Your Choice:1
Enter no. of elements:4
Enter tree data:7 12 4 9

1)Create:2)Insert:3)Delete:4)Print:5)Quit:
Enter Your Choice:4
Preorder sequence:
7(Bf=-1)4(Bf=0)12(Bf=1)9(Bf=0)

Inorder sequence:
4(Bf=0)7(Bf=-1)9(Bf=0)12(Bf=1)

1)Create:2)Insert:3)Delete:4)Print:5)Quit:
Enter Your Choice:3
Enter a data:7

1)Create:2)Insert:3)Delete:4)Print:5)Quit:
Enter Your Choice:4

Preorder sequence:
9(Bf=0)4(Bf=0)12(Bf=0)

Inorder sequence:
4(Bf=0)9(Bf=0)12(Bf=0)

1)Create:2)Insert:3)Delete:4)Print:5)Quit:
Enter Your Choice:5
```

Red - Black Trees

Red - Black Tree is a self balanced Binary Search Tree in which every node is colored either RED or BLACK. The remaining properties satisfied by a red-black tree are best stated in terms of the corresponding extended binary tree.

In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

Property #1: Red - Black Tree must be a Binary Search Tree.

Property #2: The ROOT node must be colored BLACK.

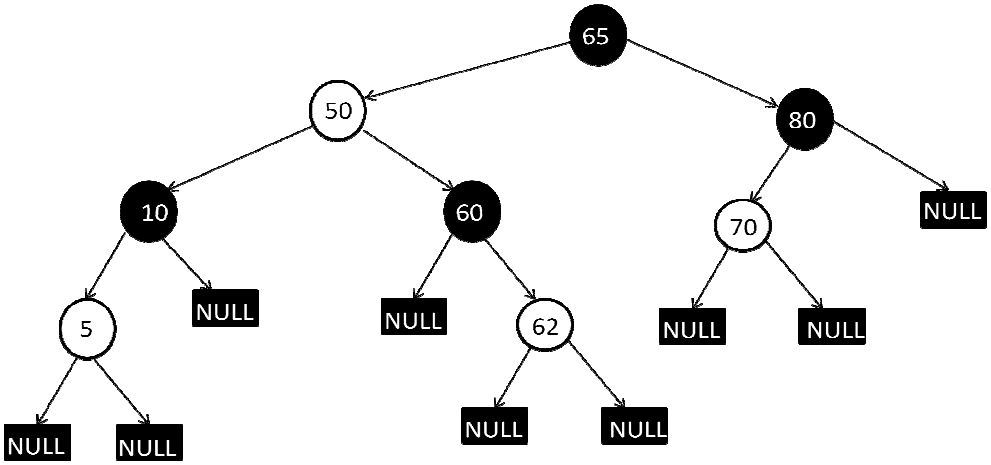
Property #3: The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).

Property #4: In all the paths of the tree, there should be same number of BLACK colored nodes.

Property #5: Every new node must be inserted with RED color.

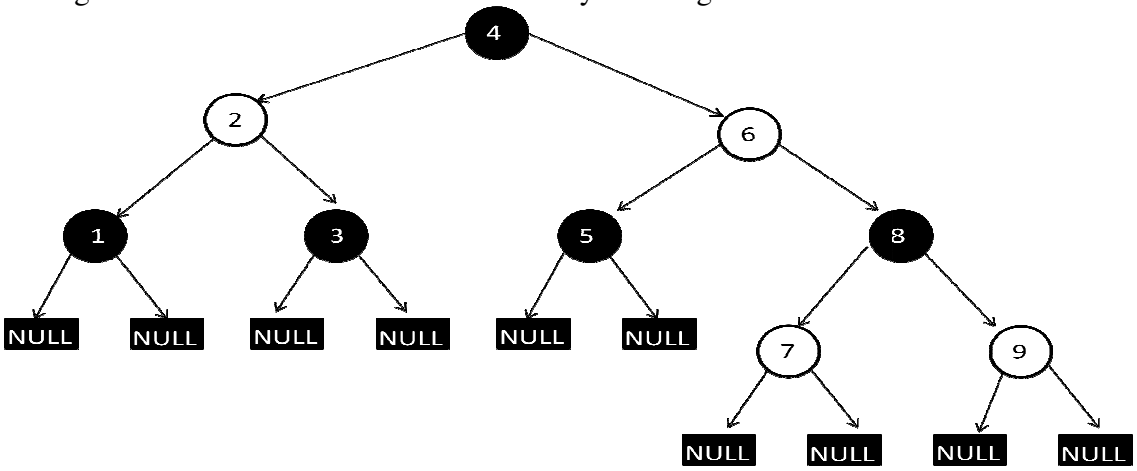
Property #6: Every leaf (i.e. NULL node) must be colored BLACK.

Example



A Red Black Tree

Following is a Red-Black Tree which is created by inserting numbers from 1 to 9.



This tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

Rank of a Node: The **rank of a node in a red-black tree** is the number of **black nodes** on any path from the **node** to any NULL pointer in its subtree.

Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operations to make it Red Black Tree.

1. **Recolor**
2. **Rotation**
3. **Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

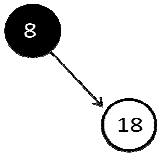
Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

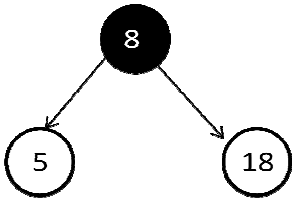
Example : Insertion into a Red Black tree. Create a red Black tree by inserting sequence of numbers 8, 18, 5,15, 17, 25, 40 and 80

Insert (8) : Tree is empty. So insert new node as root node with black color.

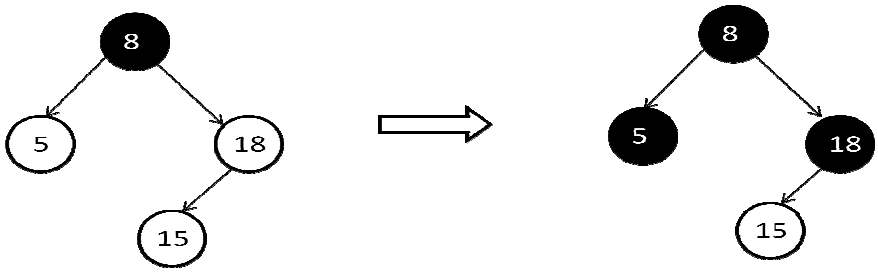
Insert (18) : Tree is not empty. So insert new node with red color.



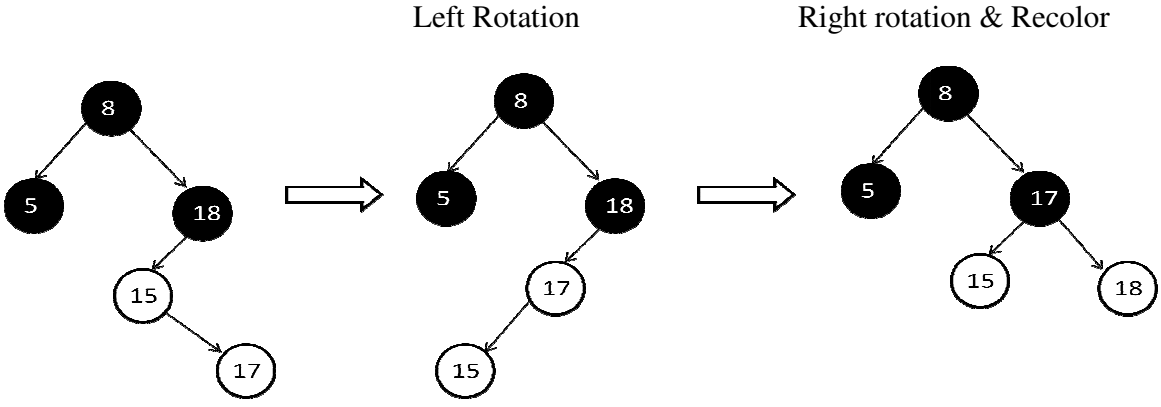
Insert (5) : Tree is not empty. So insert node with red color.



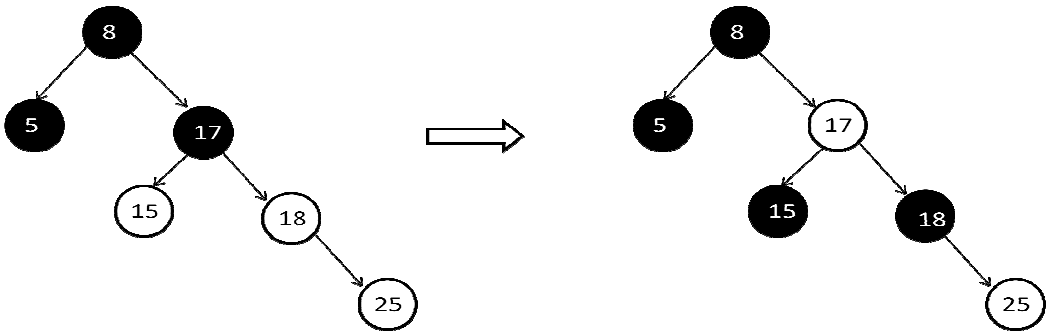
Insert (15) : Tree is not empty. So insert node with red color. Here, there are two consecutive red nodes 18 and 15. The newnode's parent sibling color is Red and parent's parent is root node. So we use Recolor to make it Red Black tree.



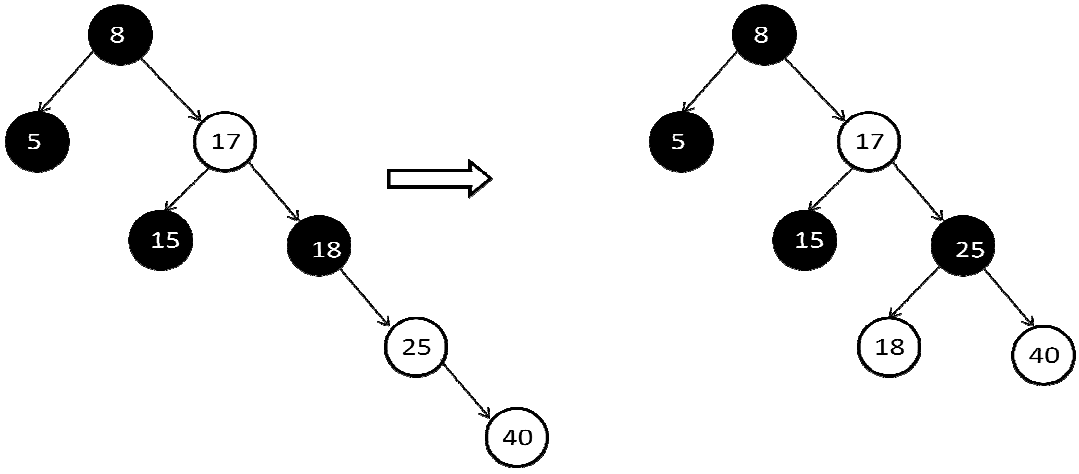
Insert (17) : Tree is not empty. So insert node with red color. Here, there are two consecutive red nodes 15 and 17. The new node's parent sibling is NULL. So we need Left rotation Right rotation and recolor.



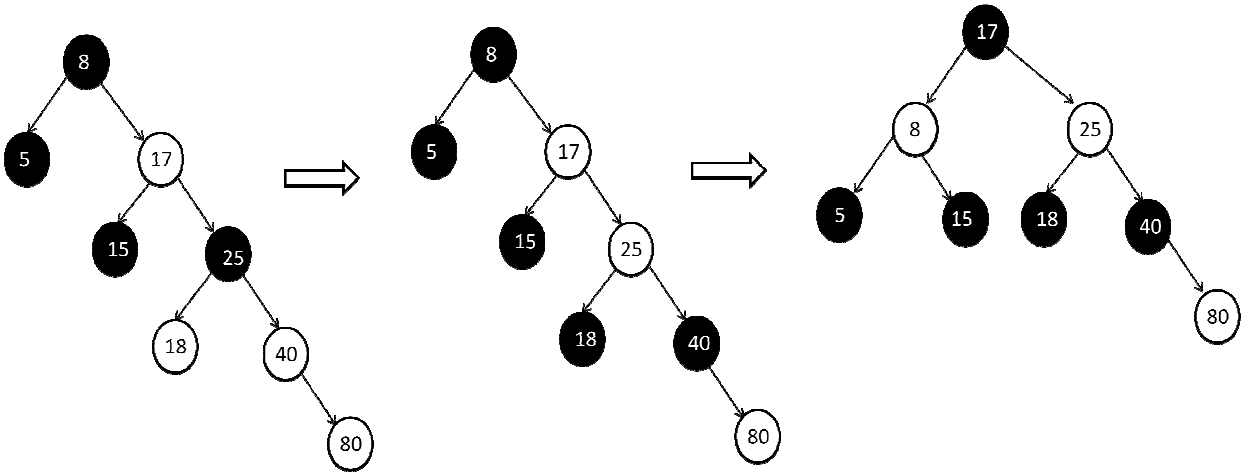
Insert (25): Tree is not empty. So insert node with red color. Here there are two consecutive nodes with Red color 18 and 25. The New node's parent sibling color is Red and Parent's parent is not Root node. So, we use recolor and recheck. After recolor operation, the tree satisfying all Red Black tree properties.



Insert (40) : Tree is not empty. So insert node with red color. Here, there are two consecutive nodes with Red color 25 and 40. The new nodes parent sibling is NULL. So we need a rotation and recolor. Here we use LL Rotation and recheck.



Insert (80) : Tree is not empty. So insert node with red color. Here, there are two consecutive nodes with Red color 40 and 80. The new node's parent sibling color is red. Parent's parent is not root. So, we use recolor and recheck.

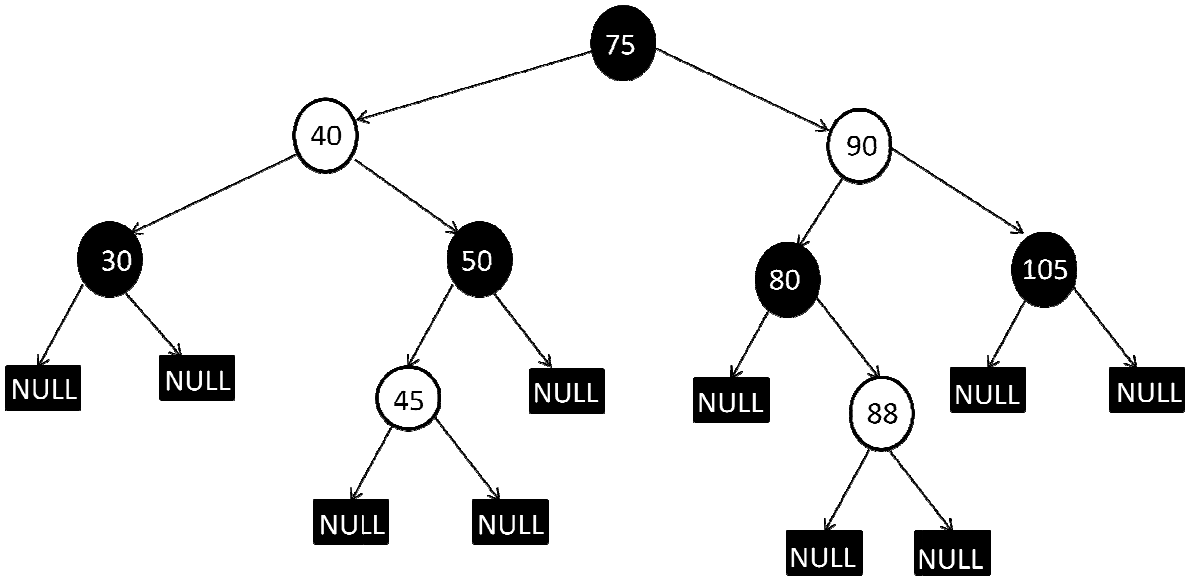


After recolor again there are two consecutive red nodes (17 and 25). The newnode's parent sibling color is Black. So, we need rotation. We use Left Rotation and recolor. Finally above tree is satisfying all the properties of Red Black tree and it is perfect red Black tree.

Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red-Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

The height of a Red-Black tree is O(logn) where (n is the number of nodes in the tree).



While representing the red black tree color of each node should be shown. In this tree leaf nodes are simply termed as null nodes which means they are not physical nodes. It can be checked easily in the above-given tree there are two types of nodes in which one of them is red and another one is black in color. The above-given tree follows all the properties of a red black tree that are

1. It is a binary search tree.
2. The root node is black.
3. The children's of red node are black.
4. All the paths from root node to external node contain same number of black nodes.

For Example: Consider path 75-90-80-88-null and 75-40-30-null in both these paths 3 black nodes are there.

Advantages of Red Black Tree

1. Red black tree are useful when we need insertion and deletion relatively frequent.
2. Red-black trees are self-balancing so these operations are guaranteed to be O(log n).
3. They have relatively low constants in a wide variety of scenarios.

Insertion in Red black Tree

- Every node which needs to be inserted should be marked as red.
- Not every insertion causes imbalancing but if imbalancing occurs then it can be removed, depending upon the configuration of tree before the new insertion is made.
- In Red black tree if imbalancing occurs then for removing it two methods are used that are: **1) Recoloring** and **2) Rotation**

To understand insertion operation, let us understand the keys required to define the following nodes:

1. Let u is newly inserted node.
2. p is the parent node of u.
3. g is the grandparent node of u.
4. Un is the uncle node of u.

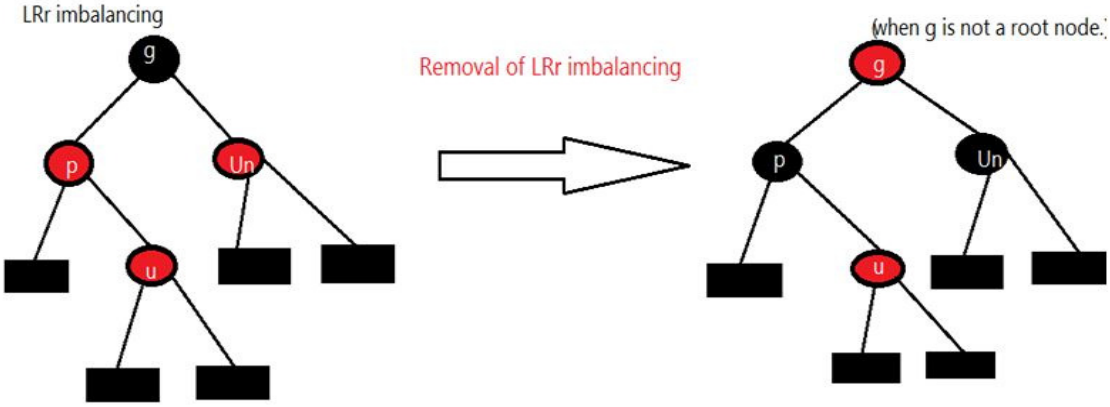
When insertion occurs the new node is inserted which is already a balanced tree. But after inserting many nodes there can be an imbalancing condition occurs which violates the property of the red black tree. So in this case first we try to recolor first then we go for a rotation.

Case 1)

The imbalancing is concerned with the color of grandparent child i.e. Uncle Node. If uncle node is red then there are four cases then in this, by doing recoloring imbalancing can be removed.

1) LRr imbalance

In this Red Black Tree violates its property in such a manner that parent and inserted child will be in a red color at a position of left and right with respect to grandparent. Therefore it is termed as Left Right imbalance.



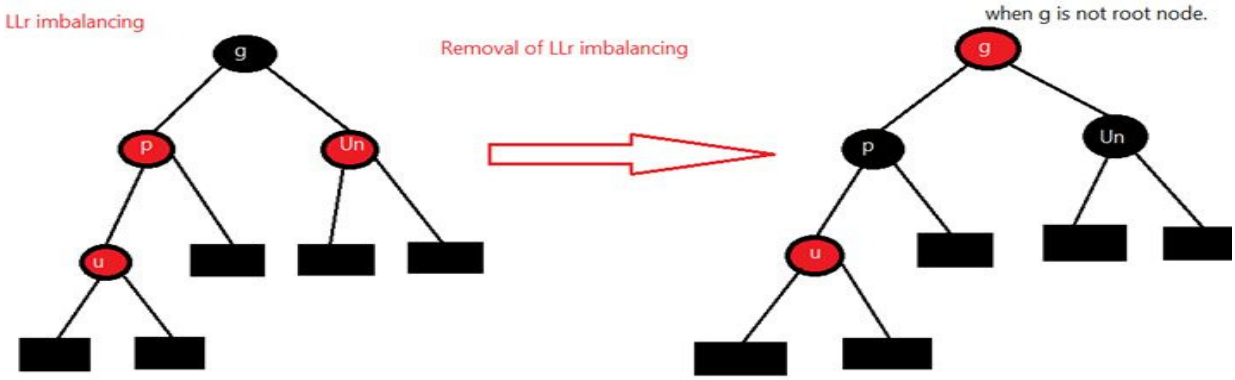
Removal of LRr imbalance can be done by:

i. Change color of p from red to black.
ii. Change color of Un from red to black.
iii. Change color of g from black to red, provided g is not a root node.

Note: If given g is root node then there will be no changes in color of g.

2) LLr imbalance

In this red black tree violates its property in such a manner that parent and inserted child will be in a red color at a position of left and left with respect to grandparent. Therefore it is termed as LEFT LEFT imbalance.



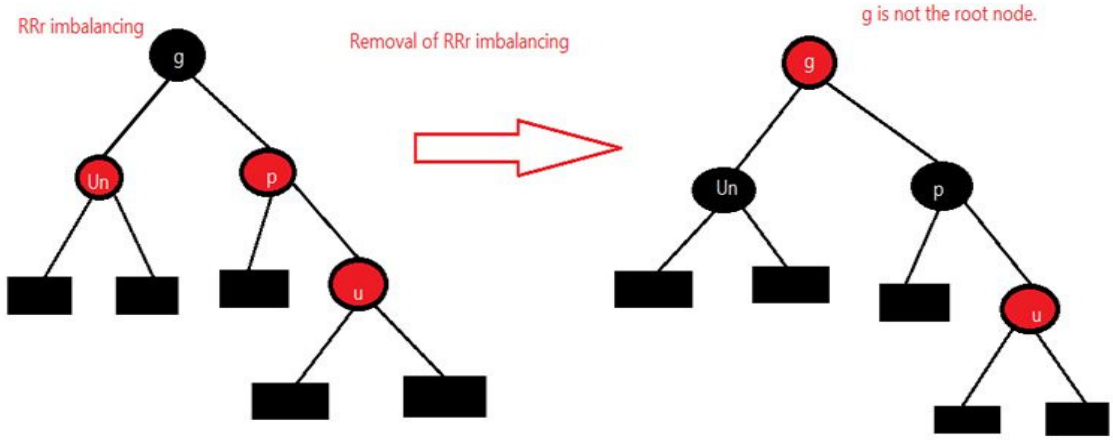
Removal of LLr imbalance can be done by:

i. Change color of p from red to black.
ii. Change color of Un from red to black.
iii. Change color of g from black to red, provided g is not a root node.

Note: If given g is root node then there will be no changes in color of g.

RRr imbalance

In this red black tree violates its property in such a manner that parent and inserted child will be in a red color at a position of right and right with respect to grandparent. Therefore it is termed as RIGHT RIGHT imbalance.



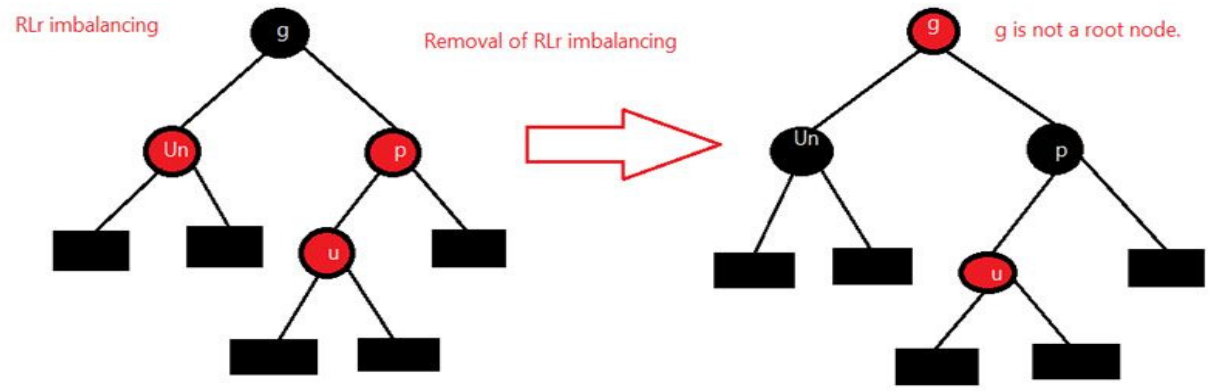
Removal of RRr imbalance can be done by:

i. Change color of p from red to black.
ii. Change color of Un from red to black.
iii. Change color of g from black to red, provided g is not a root node.

Note: If given g is root node then there will be no changes in color of g.

4) RLr imbalance

In this red black tree violates its property in such a manner that parent and inserted child will be in a red color at a position of right and left with respect to grandparent. Therefore it is termed as RIGHT LEFT imbalance.



Removal of RLr imbalance can be done by:

i. Change color of p from red to black.
ii. Change color of Un from red to black.
iii. Change color of g from black to red, provided g is not a root node.

Note: If given g is root node then there will be no changes in color of g.

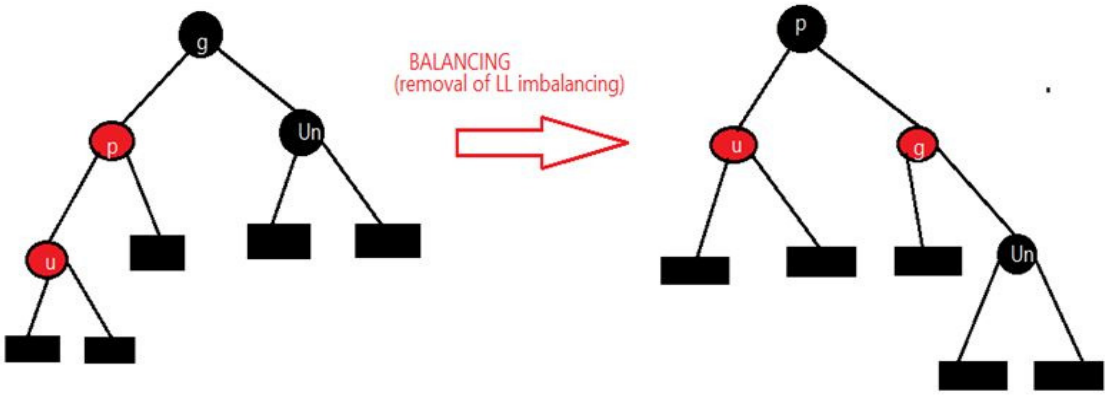
Case 2)

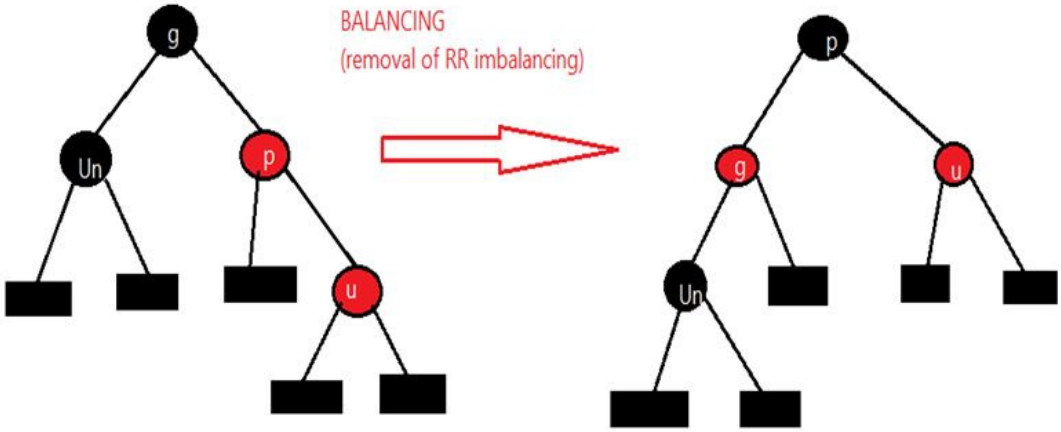
The imbalancing can also be occurred when the child of grandparent i.e. uncle node is black. Then now also four cases will be arises and in this case imbalancing can be removed by using rotation technique.

1. LR imbalancing
2. LL imbalancing
3. RR imbalancing
4. RL imbalancing

LL and RR imbalancing can be removed by following two steps i.e. are:

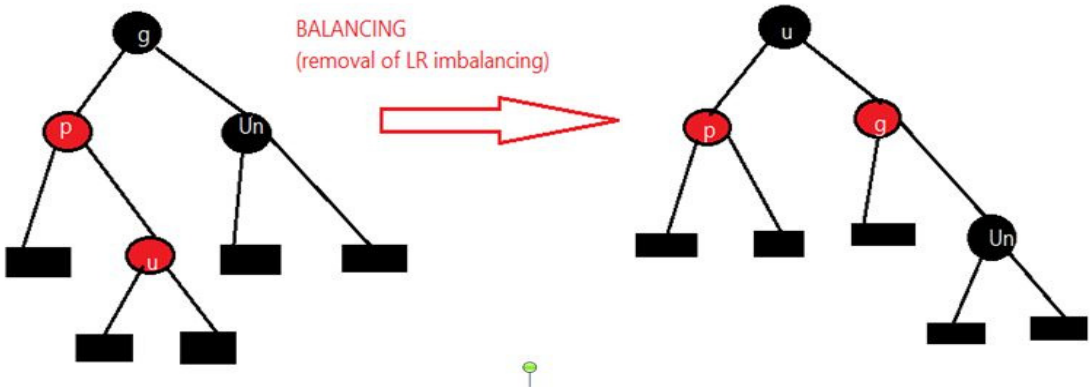
a. Apply single rotation of p about g.
b. Recolor p to black and g to red.

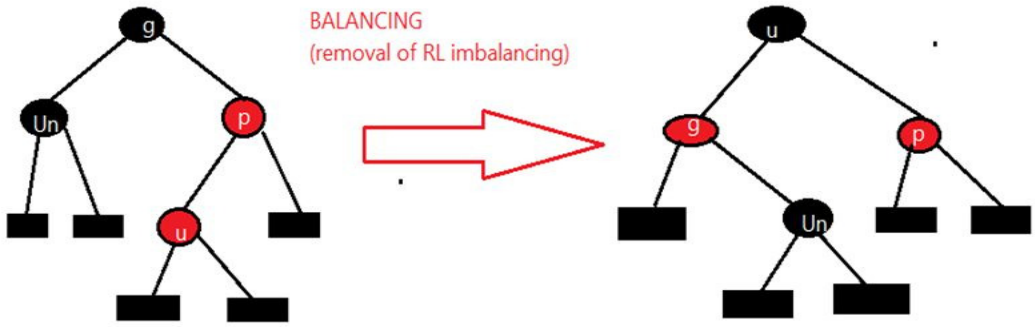




LR and RL imbalancing can be removed by following steps:

a. Apply double rotation of u about p followed by u about g.
b. For LR imbalancing recolor u to black and recolor p and g to red.
c. For RL imbalancing recolor u to black and recolor g and p to red.





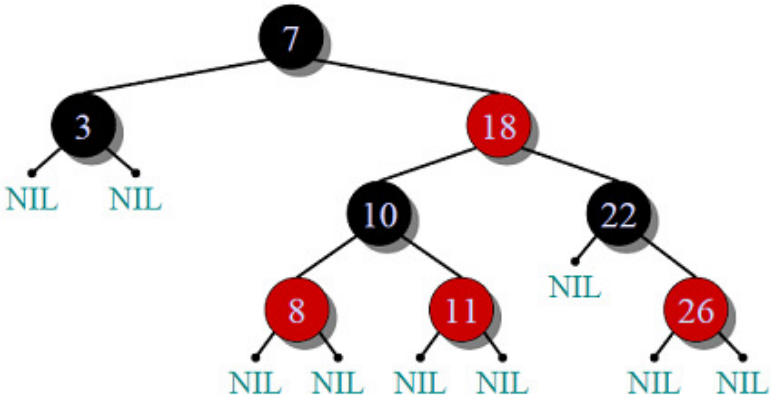
Note:

1. While inserting any node its color is by default red.
2. If uncle node is NULL then it will be consider as black.

Deleting a node from RED BLACK TREE

A **red–black tree** is a kind of **self-balancing binary search tree**. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Deleting a value in Red Black tree takes **O(log N) time complexity** and **O(N) space complexity**.



The figure depicts the basic structure of Red Black Tree.

Algorithm to Delete a node in Red Black Tree:

1) Perform standard Binary Search Tree delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.

3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.

3.2) Do following while the current node u is double black and it is not root. Let sibling of node be s.

a) If sibling s is black and at least one of sibling's children is red, perform rotation(s). Let the red child of s be r. This case can be divided in four subcases depending upon positions of s and r.

   i. Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

ii. Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

iii. Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)

iv. Right Left Case (s is right child of its parent and r is left child of s)

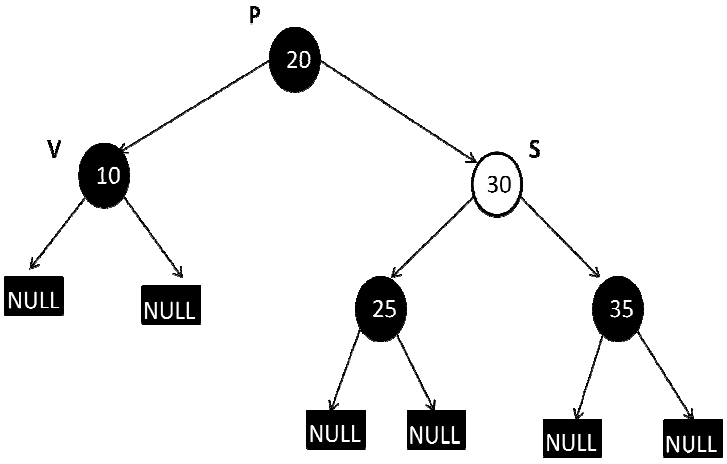
b) If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

c) If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

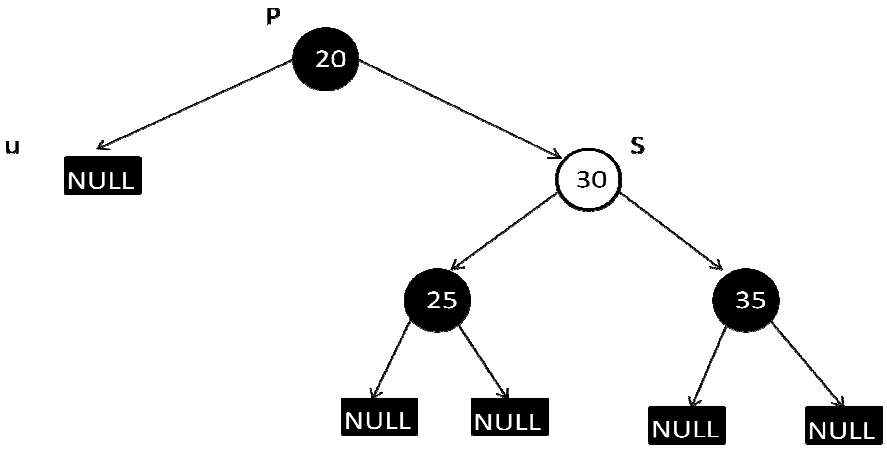
i. Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

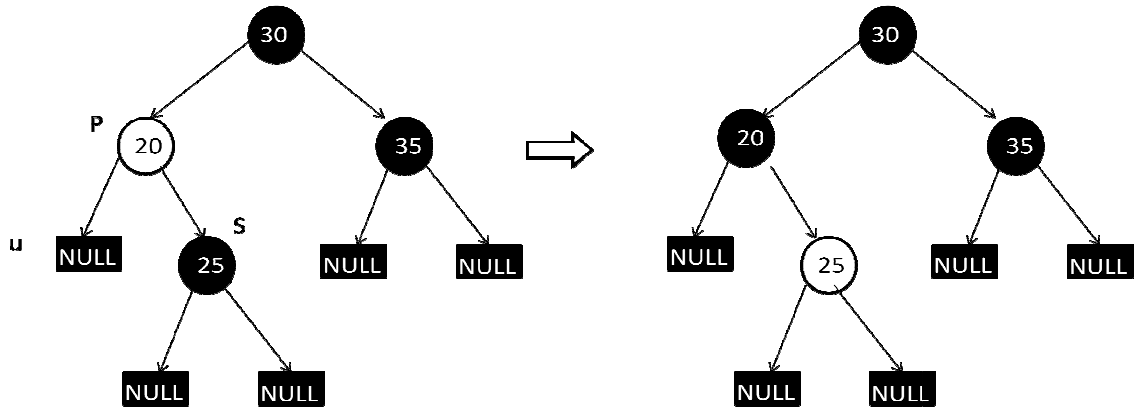
ii. Right Case (s is right child of its parent). We left rotate the parent p.

3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).



Delete Node 10




30
P
20
35
u
NULL
S
25
NULL
NULL
NULL
NULL
30
20
35
NULL
25
NULL
NULL
NULL
NULL

JOINING RED BLACK TREES

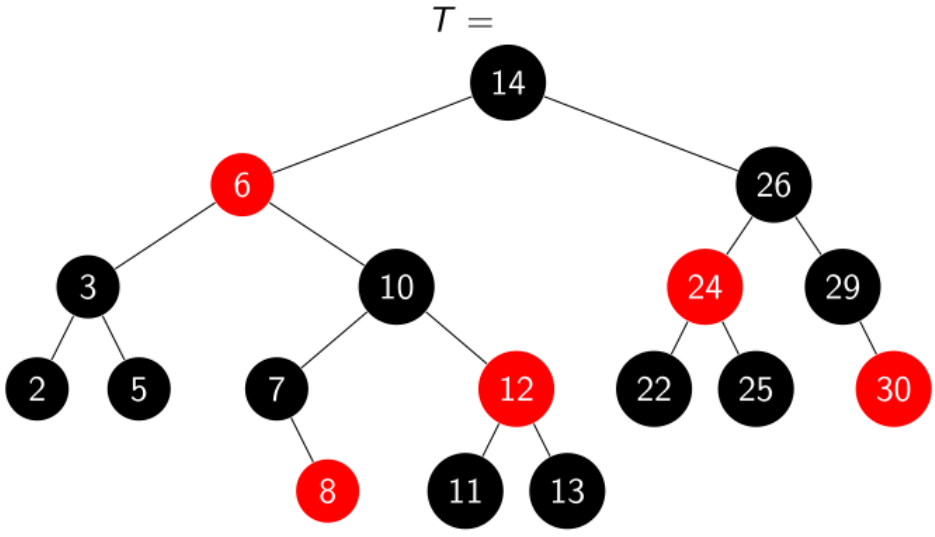
In addition to the single-element insert, delete and lookup operations, several set operations have been defined on red-black trees: union, intersection and set difference. Then fast *bulk* operations on insertions or deletions can be implemented based on these set functions. These set operations rely on two helper operations, *Split* and *Join*. With the new operations, the implementation of red-black trees can be more efficient and highly-parallelizable. This implementation allows a red root.

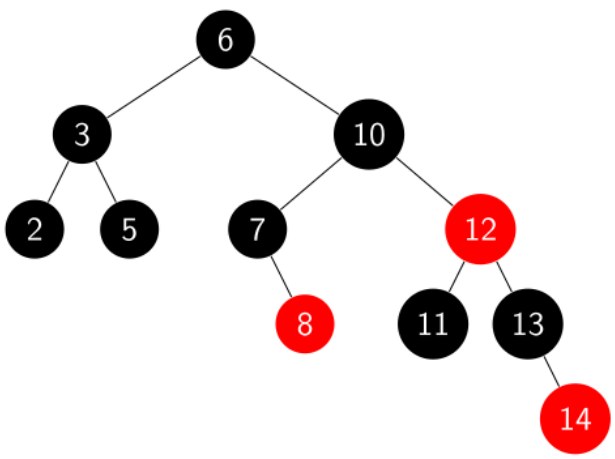
Join: The function *Join* is on two red-black trees t_1 and t_2 and a key k and will return a tree containing all elements in t_1, t_2 as well as k. It requires k to be greater than all keys in t_1 and smaller than all keys in t_2. If the two trees have the same black height, *Join* simply create a new node with left subtree t_1, root k and right subtree t_2. If both t_1 and t_2 have black root, set k to be red. Otherwise k is set black. Suppose that t_1 has larger black height than t_2 (the other case is symmetric). *Join* follows the right spine of t_1 until a black node c which is balanced with t_2. At this point a new node with left child c, root k (set to be red) and right child t_2 is created to replace c. The new node may invalidate the red-black invariant because at most three red nodes can appear in a row. This can be fixed with a double rotation. If double red issue propagates to the root, the root is then set to be black, restoring the properties. The cost of this function is the difference of the black heights between the two input trees.

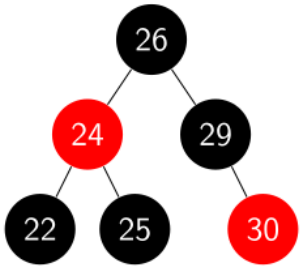
SPLITTING RED BLACK TREES

Split: To split a red-black tree into two smaller trees, those smaller than key x, and those larger than key x, first draw a path from the root by inserting x into the red-black tree. After this insertion, all values less than x will be found on the left of the path, and all values greater than x will be found on the right. By applying *Join*, all the subtrees on the left side are merged bottom-up using keys on the path as intermediate nodes from bottom to top to form the left tree, and the right part is asymmetric. For some applications, *Split* also returns a boolean value denoting if x appears in the tree. The cost of *Split* is O(log n), order of the height of the tree. This algorithm actually has nothing to do with any special properties of a red-black tree, and thus is generic to other balancing schemes such as AVL trees.

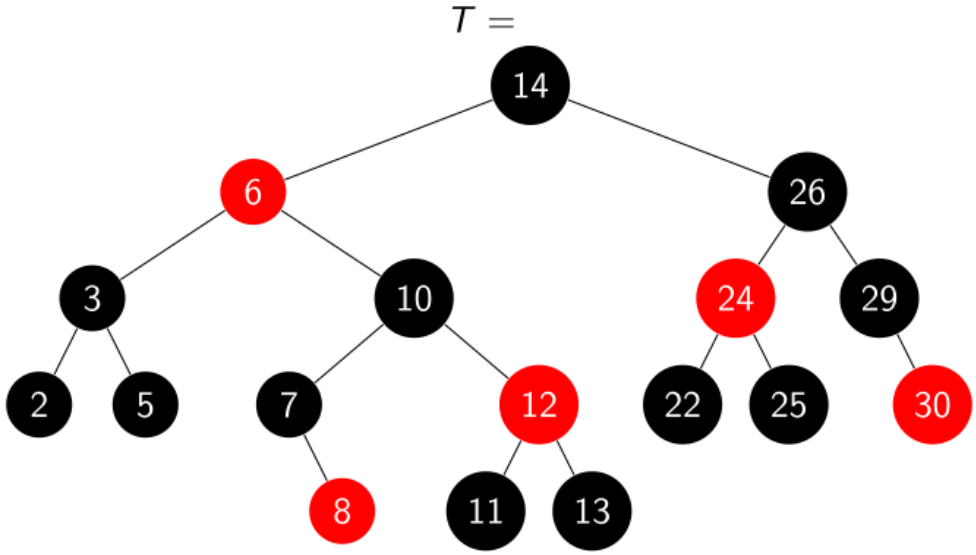
Ex 1)



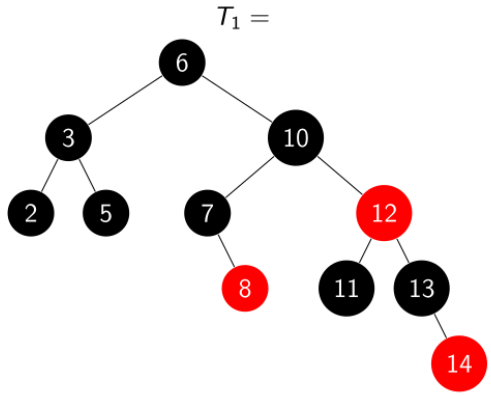




Ex 2)

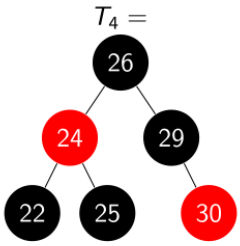


I = (1, 4, 9, 15, 16, 17, 18, 19, 20, 21, 23, 27, 28)

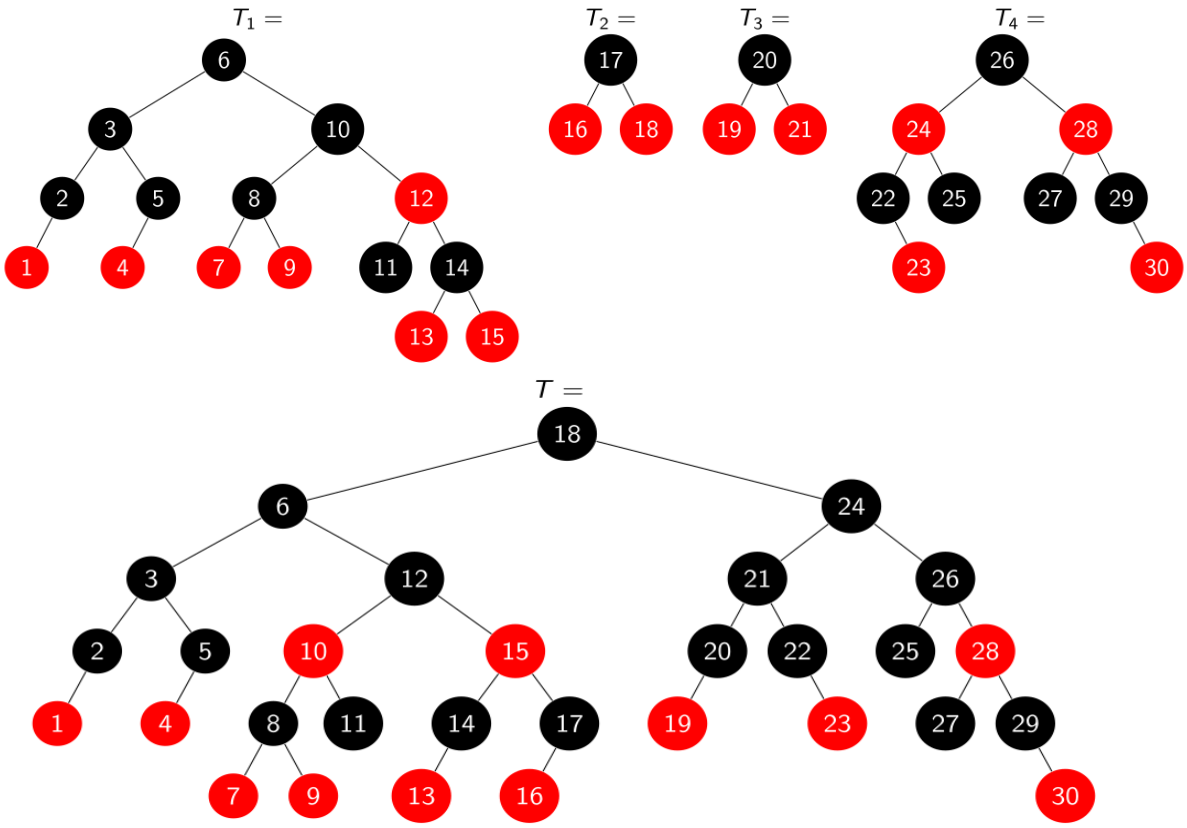


T2 =

T3 =



I_1=(1,4,9,15) I_2=(16,17,18) I_3=(19,20,21) I_4=(23,27,28)


$T_1 =$
6
3
10
2
5
8
12
1
4
7
9
11
14
13
15
$T_2 =$
17
16
18
$T_3 =$
20
19
21
$T_4 =$
26
24
28
22
25
27
29
23
30

$T =$
18
6
24
3
12
21
26
2
5
10
15
20
22
25
28
1
4
8
11
14
17
19
23
27
29
7
9
13
16
30

```
//-------------------------------------------------
// Implementing Red-Black Tree in C
//-------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
enum nodeColor
{
  RED,
  BLACK
};
struct rbNode
{
     int data, color;
     struct rbNode *link[2];
};
struct rbNode *root = NULL;
struct rbNode *createNode(int data)
{
    struct rbNode *newnode;
    newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
    newnode->data = data;
    newnode->color = RED;
    newnode->link[0] = newnode->link[1] = NULL;
    return newnode;
}



void insertion(int data)
{
     struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
     int dir[98], ht = 0, index;
     ptr = root;
     if (!root)
    {
        root = createNode(data);
        return;
    }
  stack[ht] = root;
  dir[ht++] = 0;
  while (ptr != NULL)
  {
    if (ptr->data == data)
    {
      printf("Duplicates Not Allowed!!\n");
      return;
    }
    index = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    ptr = ptr->link[index];
    dir[ht++] = index;
  }
```

```
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED))
{
 if (dir[ht - 2] == 0)
 {
  yPtr = stack[ht - 2]->link[1];
  if (yPtr != NULL && yPtr->color == RED)
  {
   stack[ht - 2]->color = RED;
   stack[ht - 1]->color = yPtr->color = BLACK;
   ht = ht - 2;
  }
  else
  {
   if (dir[ht - 1] == 0)
   {
    yPtr = stack[ht - 1];
   }
   else
   {
    xPtr = stack[ht - 1];
    yPtr = xPtr->link[1];
    xPtr->link[1] = yPtr->link[0];
    yPtr->link[0] = xPtr;
    stack[ht - 2]->link[0] = yPtr;
   }
   xPtr = stack[ht - 2];
   xPtr->color = RED;
   yPtr->color = BLACK;
   xPtr->link[0] = yPtr->link[1];
   yPtr->link[1] = xPtr;
   if (xPtr == root)
   {
    root = yPtr;
   }
   else
   {
    stack[ht - 3]->link[dir[ht - 3]] = yPtr;
   }
   break;
  }
 }
 else
 {
    yPtr = stack[ht - 2]->link[0];
    if ((yPtr != NULL) && (yPtr->color == RED))
   {
       stack[ht - 2]->color = RED;
       stack[ht - 1]->color = yPtr->color = BLACK;
       ht = ht - 2;
   }
  else
   {
```

```
          if (dir[ht - 1] == 1)
          {
            yPtr = stack[ht - 1];
          }
          else
          {
            xPtr = stack[ht - 1];
            yPtr = xPtr->link[0];
            xPtr->link[0] = yPtr->link[1];
            yPtr->link[1] = xPtr;
            stack[ht - 2]->link[1] = yPtr;
          }
          xPtr = stack[ht - 2];
          yPtr->color = BLACK;
          xPtr->color = RED;
          xPtr->link[1] = yPtr->link[0];
          yPtr->link[0] = xPtr;
          if (xPtr == root)
          {
            root = yPtr;
          }
          else
          {
            stack[ht - 3]->link[dir[ht - 3]] = yPtr;
          }
          break;
        }
      }
    }
    root->color = BLACK;
}
void deletion(int data)
{
    struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
    struct rbNode *pPtr, *qPtr, *rPtr;
    int dir[98], ht = 0, diff, i;
    enum nodeColor color;
    if (!root)
    {
      printf("Tree not available\n");
      return;
    }
    ptr = root;
    while (ptr != NULL)
    {
      if ((data - ptr->data) == 0)
        break;
      diff = (data - ptr->data) > 0 ? 1 : 0;
      stack[ht] = ptr;
      dir[ht++] = diff;
      ptr = ptr->link[diff];
    }
    if (ptr->link[1] == NULL)
```

```
{
  if ((ptr == root) && (ptr->link[0] == NULL))
  {
    free(ptr);
    root = NULL;
  }
  else if (ptr == root)
  {
    root = ptr->link[0];
    free(ptr);
  }
  else
  {
    stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
  }
}
else
{
  xPtr = ptr->link[1];
  if (xPtr->link[0] == NULL)
  {
    xPtr->link[0] = ptr->link[0];
    color = xPtr->color;
    xPtr->color = ptr->color;
    ptr->color = color;
    if (ptr == root)
    {
      root = xPtr;
    }
    else
    {
      stack[ht - 1]->link[dir[ht - 1]] = xPtr;
    }
    dir[ht] = 1;
    stack[ht++] = xPtr;
  }
  else
  {
    i = ht++;
    while (1)
    {
      dir[ht] = 0;
      stack[ht++] = xPtr;
      yPtr = xPtr->link[0];
      if (!yPtr->link[0])
        break;
      xPtr = yPtr;
    }
    dir[i] = 1;
    stack[i] = yPtr;
    if (i > 0)
      stack[i - 1]->link[dir[i - 1]] = yPtr;
    yPtr->link[0] = ptr->link[0];
```

```
    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = ptr->link[1];
    if (ptr == root)
    {
      root = yPtr;
    }
    color = yPtr->color;
    yPtr->color = ptr->color;
    ptr->color = color;
  }
}
if (ht < 1)
  return;
if (ptr->color == BLACK)
{
  while (1)
  {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
    if (pPtr && pPtr->color == RED)
    {
      pPtr->color = BLACK;
      break;
    }
    if (ht < 2)
      break;
    if (dir[ht - 2] == 0)
    {
      rPtr = stack[ht - 1]->link[1];
      if (!rPtr)
        break;
      if (rPtr->color == RED)
      {
        stack[ht - 1]->color = RED;
        rPtr->color = BLACK;
        stack[ht - 1]->link[1] = rPtr->link[0];
        rPtr->link[0] = stack[ht - 1];
        if (stack[ht - 1] == root)
        {
          root = rPtr;
        }
        else
        {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        dir[ht] = 0;
        stack[ht] = stack[ht - 1];
        stack[ht - 1] = rPtr;
        ht++;
        rPtr = stack[ht - 1]->link[1];
      }
      if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
        (!rPtr->link[1] || rPtr->link[1]->color == BLACK))
      {
```

```
      rPtr->color = RED;
    }
    else
    {
      if (!rPtr->link[1] || rPtr->link[1]->color == BLACK)
      {
        qPtr = rPtr->link[0];
        rPtr->color = RED;
        qPtr->color = BLACK;
        rPtr->link[0] = qPtr->link[1];
        qPtr->link[1] = rPtr;
        rPtr = stack[ht - 1]->link[1] = qPtr;
      }
      rPtr->color = stack[ht - 1]->color;
      stack[ht - 1]->color = BLACK;
      rPtr->link[1]->color = BLACK;
      stack[ht - 1]->link[1] = rPtr->link[0];
      rPtr->link[0] = stack[ht - 1];
      if (stack[ht - 1] == root)
      {
        root = rPtr;
      }
      else
      {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      break;
    }
  }
  else
  {
    rPtr = stack[ht - 1]->link[0];
    if (!rPtr)
      break;
    if (rPtr->color == RED)
    {
      stack[ht - 1]->color = RED;
      rPtr->color = BLACK;
      stack[ht - 1]->link[0] = rPtr->link[1];
      rPtr->link[1] = stack[ht - 1];
      if (stack[ht - 1] == root)
      {
        root = rPtr;
      }
      else
      {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      dir[ht] = 1;
      stack[ht] = stack[ht - 1];
      stack[ht - 1] = rPtr;
      ht++;
      rPtr = stack[ht - 1]->link[0];
```

```
        }
        if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
          (!rPtr->link[1] || rPtr->link[1]->color == BLACK))
        {
          rPtr->color = RED;
        }
        else
        {
          if (!rPtr->link[0] || rPtr->link[0]->color == BLACK)
          {
            qPtr = rPtr->link[1];
            rPtr->color = RED;
            qPtr->color = BLACK;
            rPtr->link[1] = qPtr->link[0];
            qPtr->link[0] = rPtr;
            rPtr = stack[ht - 1]->link[0] = qPtr;
          }
          rPtr->color = stack[ht - 1]->color;
          stack[ht - 1]->color = BLACK;
          rPtr->link[0]->color = BLACK;
          stack[ht - 1]->link[0] = rPtr->link[1];
          rPtr->link[1] = stack[ht - 1];
          if (stack[ht - 1] == root)
          {
            root = rPtr;
          }
          else
          {
            stack[ht - 2]->link[dir[ht - 2]] = rPtr;
          }
          break;
        }
      }
      ht--;
    }
  }
}

void inorderTraversal(struct rbNode *node)
{
  if (node)
  {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
   return;
}


int main()
{
  int ch, data;
```

```
 while (1)
 {
        printf("1. Insertion\t2. Deletion\n");
        printf("3. Traverse\t4. Exit");
        printf("\nEnter your choice:");
        scanf("%d", &ch);

  switch (ch)
  {
        case 1:    printf("Enter the element to insert:");
                   scanf("%d", &data);
                   insertion(data);
                   break;
        case 2:    printf("Enter the element to delete:");
                   scanf("%d", &data);
                   deletion(data);
                   break;
        case 3:    inorderTraversal(root);
                   printf("\n");
                   break;
        case 4:   exit(0);
        default:  printf("Not available\n");
                  break;
  }
  printf("\n");
 }
 return 0;
}
```

QUESTIONS FROM PREVIOUS UNIVERSITY EXAMINATIONS

1) What is the rank of a node in a Red-Black tree.

The **rank of a node in a red-black tree** is the number of **black nodes** on any path from the **node** to any NULL pointer in its subtree.

2) An extended Binary search tree has 40 failure nodes. How many internal nodes does the tree have? Justify your answer.

39

3) Define internal nodes and External nodes in an extended binary search tree?

4) What is an AVL tree?

Height Balanced Tree. AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson, Velsky, and Landis and hence given the short form as AVL tree or Balanced Binary Tree.

5) What is black height of a node in Red-Black trees?

Black height of a red black tree is number of black nodes on a path from root to a leaf. Leaf nodes are also counted as black nodes. From the properties of red black trees, we can derive, **a Red-Black Tree of height h has black-height >= h/2**.

Black height is an important term used with red-black trees. It is the number of black nodes on any simple path from a node *x* (not including it) to a leaf. Black height of any node *x* is represented by bh(x).

The number of black nodes from a node to any leaf is the same. Thus, the black height of any node counted on any path to any leaf will be unique.

6) Explain the right of left rotation of an AVL tree?

7) Explain the left of right rotation of an AVL tree?

8) Define the Optimal binary search tree?

9) List the properties of Red-Black trees?

10) What is an Extended Binary search tree?

11) What is the maximum number of nodes in an AVL tree given a height h.

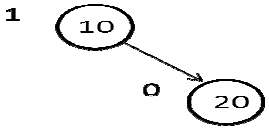
12) What is an AVL search tree? How do we define the height of it? Explain about the balance factor associated with a node of an AVL tree.

13) Construct AVL tree for the following numbers 14, 8, 12, 46, 23, 5, 77, 88, 20

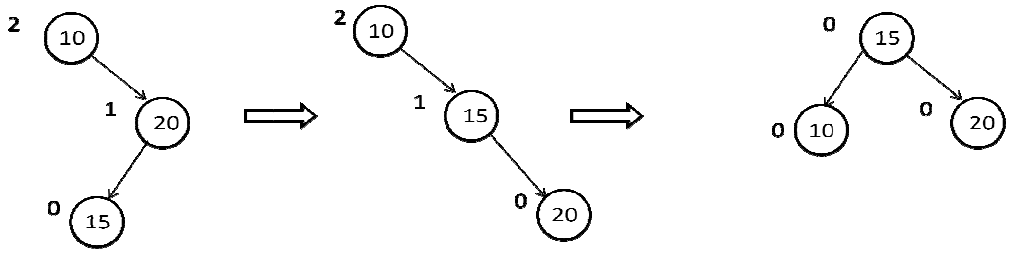
14) Insert the following sequence of elements into an AVL tree, starting with an empty tree 10, 20, 15, 25, 30, 16, 18, 19

Insert(10)

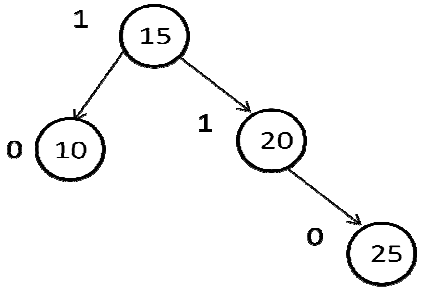
insert(20)



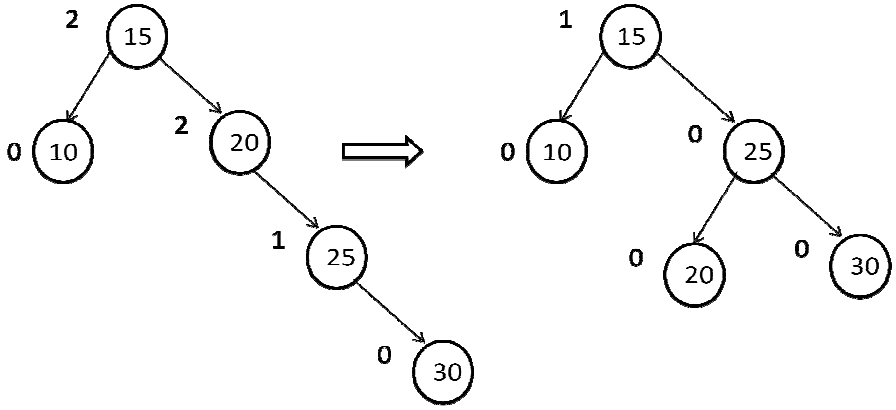
Insert(15)



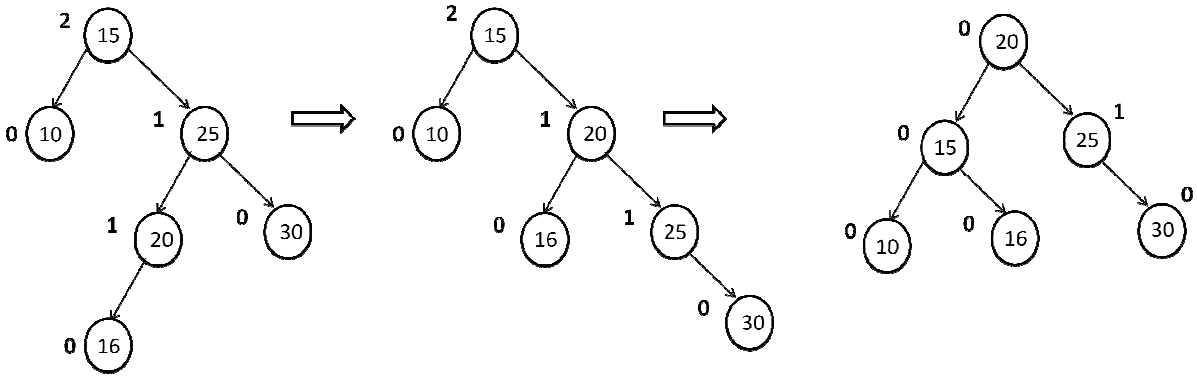
Insert(25)



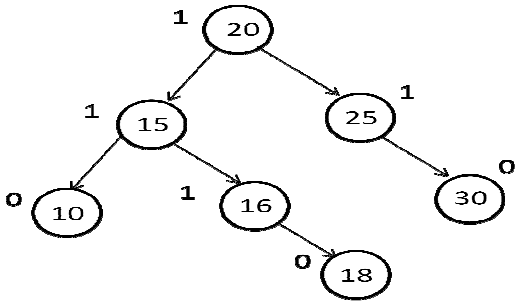
Insert(30)



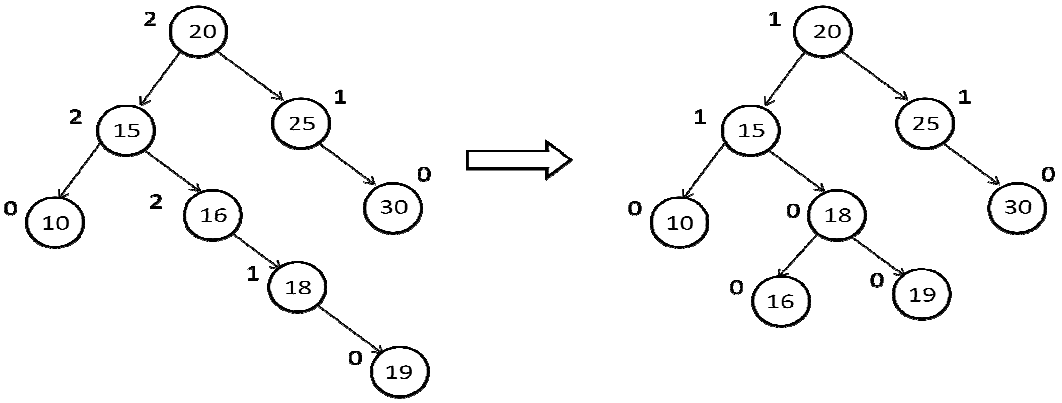
Insert(16)



Insert(18)

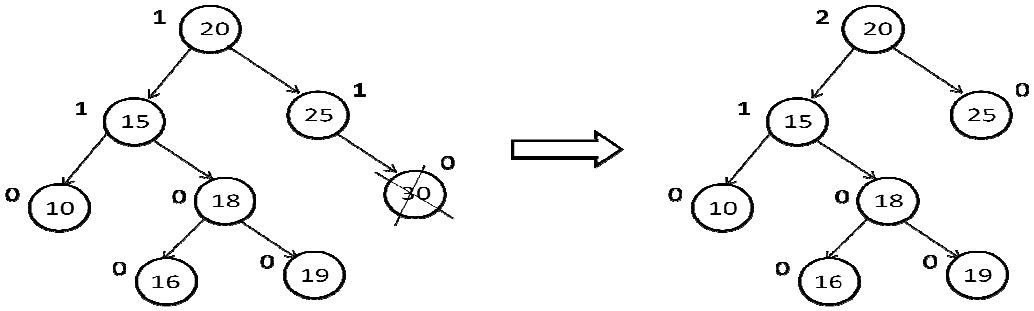


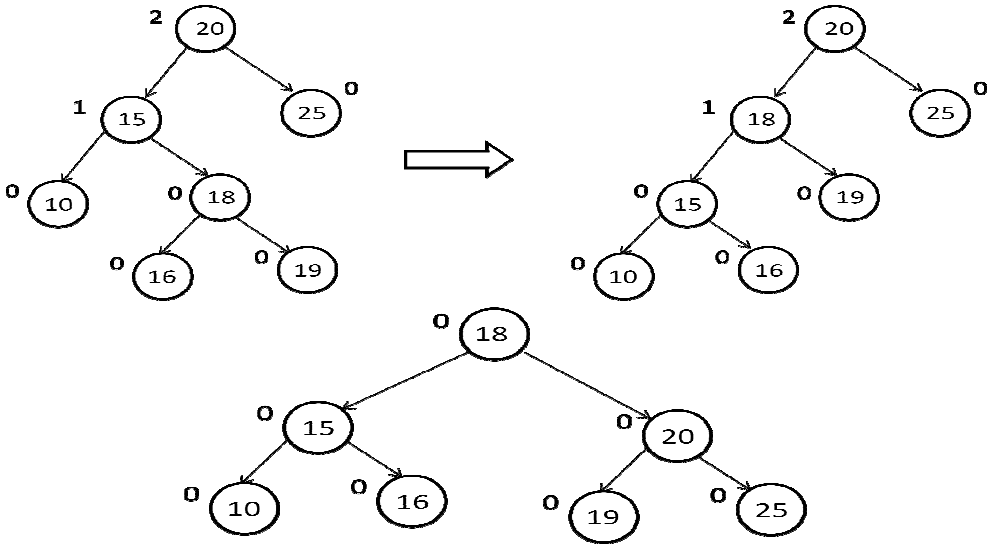
Insert(19)



15) Delete 30 in the AVL tree you got?

Delete(30)





16) Construct AVL tree for the following numbers 1, 2, 3, 4, 8, 7, 6, 5, 11, 10, 12

17) Define balanced binary search tree. Construct binary search tree for the data 8, 10, 3, 2, 1, 5, 4, 6, Insert an element 7 into binary search tree using AVL rotation.

18) Write a routine for inserting an element into an AVL tree.

19) Construct an optimal binary search tree for the below case: n = 4, (a1, a2, a3, a4) = (10, 15, 20, 25), (p1, p2, p3, p4) = (3, 3, 1,1) and (q0, q1, q2, q3, q4) = (2, 3, 1, 1,1).

20) Compute the worst case height of a red black tree with n elements

O(log n)

21) Create an AVL tree using the following elements as a sequence set. Show the balance factor in the resulting tree 13, 22, 6, 9, 32, 55, 79, 65, 70

22) Insert 42, 43, 46 and 49 in the above constructed AVL tree and show a balanced AVL tree

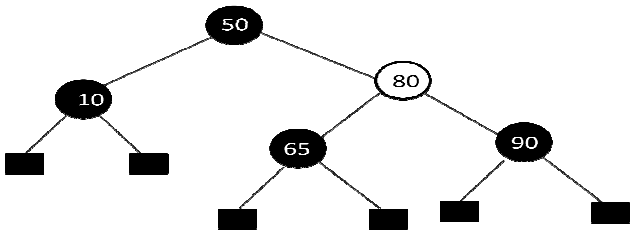
23) Create a Red-black tree by inserting the following sequence of numbers 8, 18, 5, 15, 17, 25, 40 and 80 and explain the process?

24) Write the AVL tree deletion algorithm?

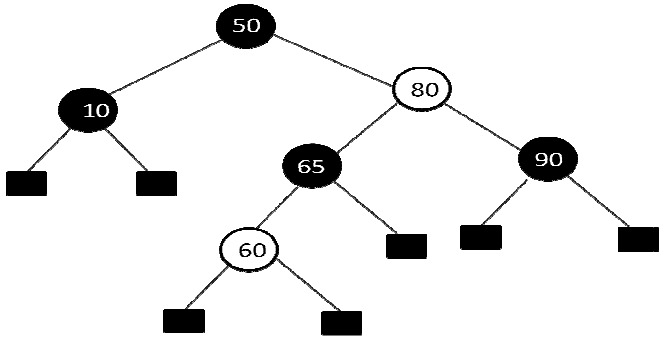
25) How to split the Red-Black tree. Explain with an example?

26) Write the AVL tree insertion algorithm?

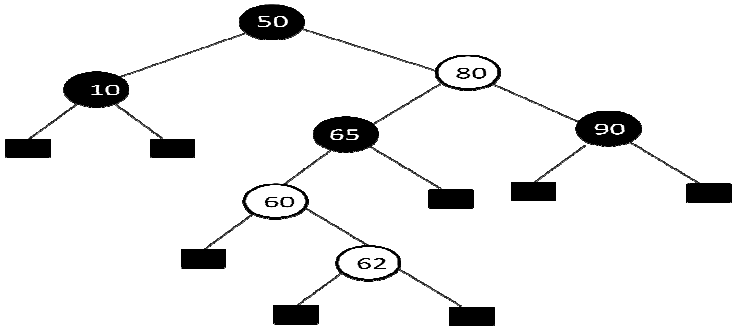
27) Insert 60, 65 and 62 in the following Red-black tree. Show the resultant Red-black tree



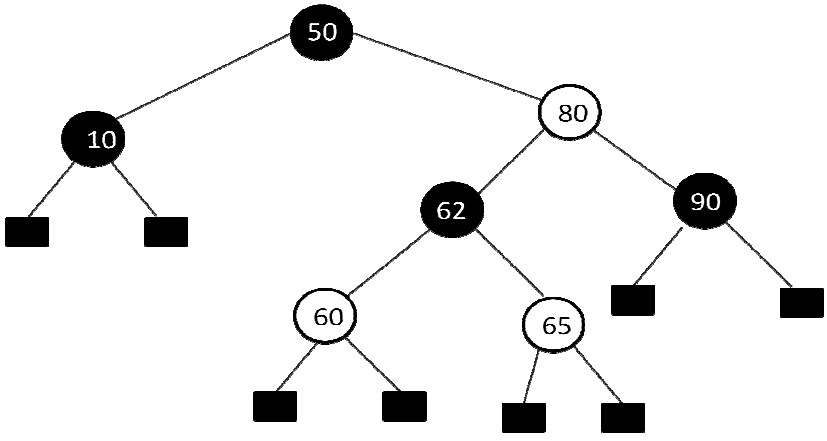
Insert(60)



Insert 65 : 65 is duplicate – duplicate nodes can not be added
Insert 62



60 and 62 are red nodes. Parent sibling of new node is null. So rotate. Now the tree satisfying all the properties of Red Black tree.



28) Explain the step by step process of joining the Red-black trees?

29) If T is a binary tree with n internal nodes, I is its internal path length and E is its external path length, then prove that E=I+2n for n>=0

30) Compare the worst case height of a red-black tree with n nodes and the worst case height of an AVL tree with same number of nodes

31) In an initially empty AVL tree insert the following Keys: DEC, JAN, APR, MAR, JUL, AUG, OCT, FEB, NOV. Draw AVL tree after every insertion and apply rotations where ever necessary

32) With suitable example, explain the procedure for joining the two red-black trees

33) Construct optimal binary search tree for the key set (a1, a2, a3, a4) = (5, 10, 15, 20) with p1=1/20, p2=1/5, p3=1/10, p4=1/20, q0=1/5, q1=1/10, q2=1/5, q3=1/20 and q4=1/20,

34) Briefly discuss about cases of imbalance that might be caused by a red-black tree insertion and corresponding methods to rebalance the tree?

35) With suitable examples, explain different rotations associated with AVL tree insertion.

36) Start with an empty Red-black tree and insert the following keys in the given order. Draw the red-black tree after every insertion and apply rotations and recoloring where ever necessary.: 15,14, 13, 12, 11, 10,9,8,7,6

37) Write about the two way join, three way join and split operations w.r.t. red-black trees.

38) Derive an expression for maximum height of an AVL tree with n nodes.

If there are n nodes in AVL tree, minimum height of AVL tree is floor($\log_2 n$).
If there are n nodes in AVL tree, maximum height can't exceed $1.44*\log_2 n$.

The worst case possible height of AVL tree with n nodes is 1.44*log n.
This can be verified using AVL tree having 7 nodes and maximum height.

using 7 nodes, we can achieve maximum height as 3. Following is the AVL tree with 7 nodes and height 3.