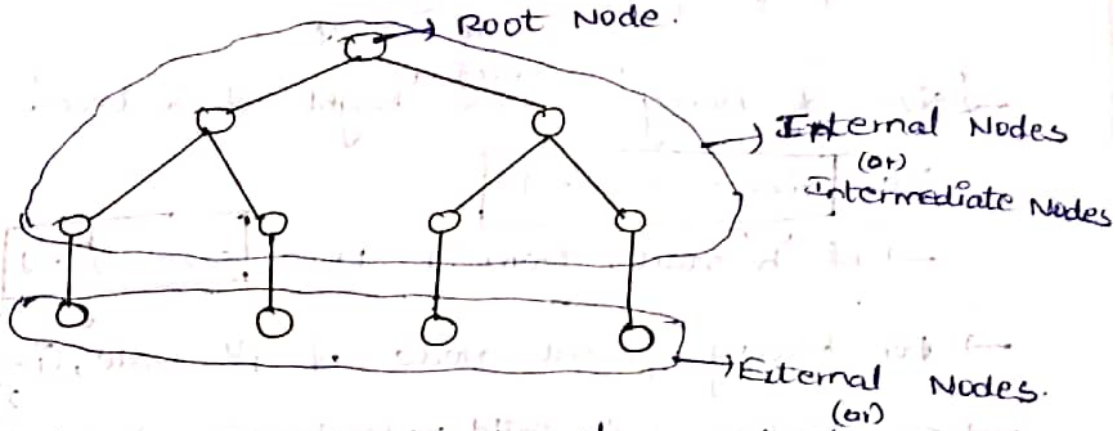
25-04-2023

UNIT-IV · TREES

* Tree :-

→ A 'Tree' is a collection of nodes and there is a one distinct node is called 'Root Node.'

→ A 'Root Node' having no parent Nodes.



Internal Nodes:- Having child Nodes

External Nodes:- Having no child Nodes.

→ If, In a given tree any Node having two child Nodes then it is called 'Binary Tree.'

→ If a tree has any Node containing more than two child nodes then it is called 'M-way Tree.'

* Depth of a Node :-

→ No. of 'Ancestors' is called Depth of a Node.

Ancestors :- Parent (or) grand parent Nodes.

Successors / Descendents :- child (or) grand child Nodes.

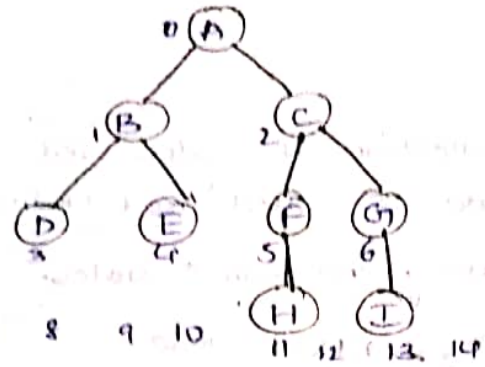
* Height of a Tree :-

→ 'Maximum of depth of a Node' is called the 'Height of a Tree.'

* Sub-Tree :-

→ The 'node' and its Descendents' is known as 'Sub-Tree.'

* Representation of a Binary Tree using Arrays:



→ 'Size of Array' is using the 'height of a tree'.

i.e., $\boxed{\text{size} = 2^{h+1} - 1}$

→ If 'h' starts from '1' then $\boxed{\text{size} = 2^{h} - 1}$

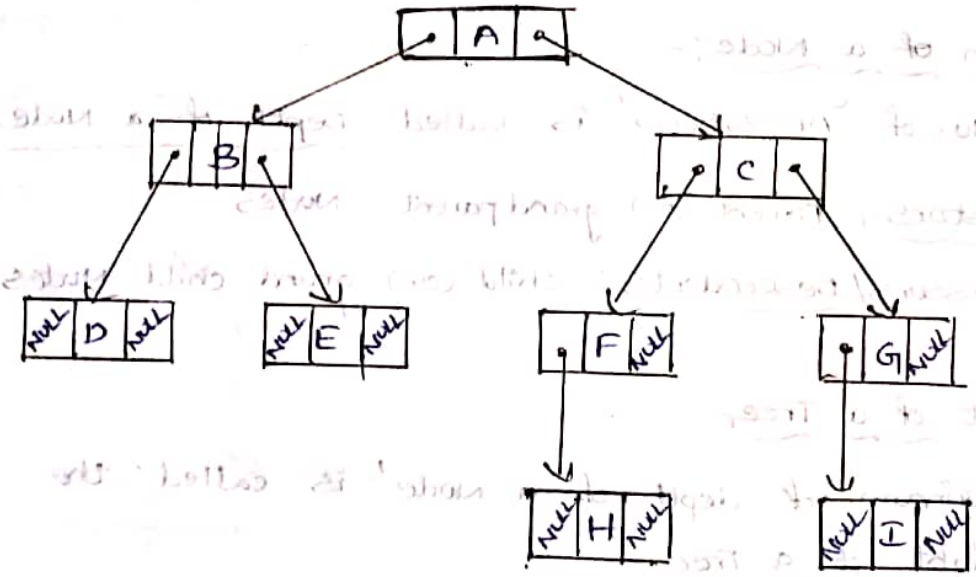
→ For finding parent Node of i^{th} Node, $\frac{(i-1)}{2}$.

→ For finding Left child node location, '2i+1'

→ For finding Right child node location, '2i+2'.

A	B	C	D	E	F	G	-1	-1	-1	-1	H	-1	I	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

* Representation of a Binary Tree using Linked List:



Full Binary Tree:-

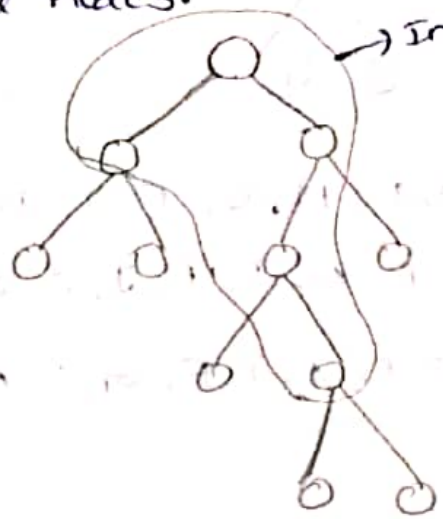
→ A Node containing either '2' or '0' Nodes in the entire tree then it is called 'Full Binary Node'.

Note: For Full Binary tree:

→ If there are 'l' exteral nodes then there are 'l-1' Internal Nodes.

→ If there are 'i' internal nodes then there are 'i+1' external nodes.

Ex:



Internal = i = 5

External = i+1 = 6.

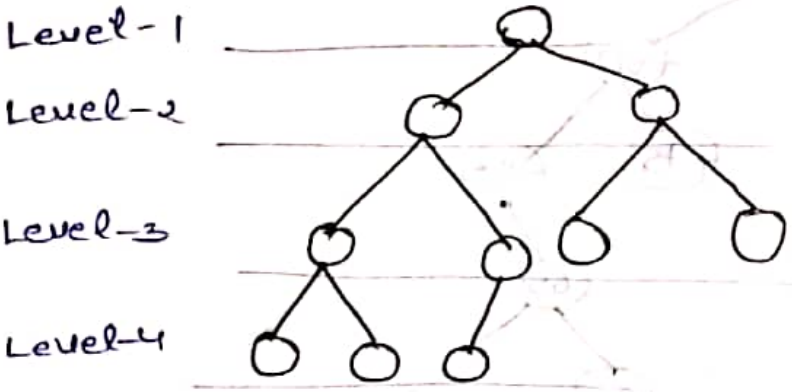
Or)

External = 6 = l

Internal = l-1 = 5

Total = 2(l)-1 = 12-1 = 11

*complete Binary Tree:



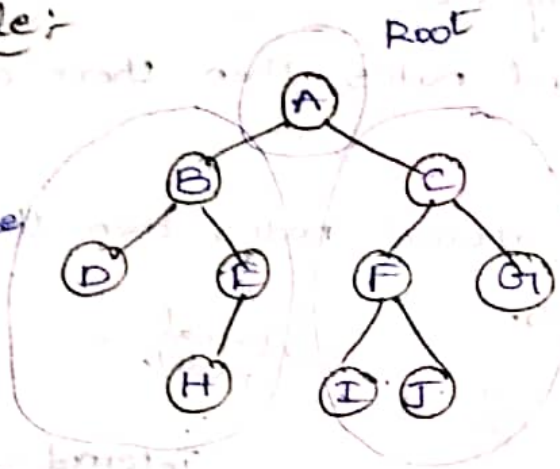
→ Leaf Level will completes by left to right manner.

*Tree Traversals:

→ There are three types.

1. Level order traversal.
2. Depth first traversal.
   - (i) Pre-order traversal [Root, Left subtree, Right stree
   - (ii) In-order traversal. [L.S.T, Root, R.S.T]
   - (iii) Post-order traversal. [L.S.T, R.S.T, Root].

* Example :-

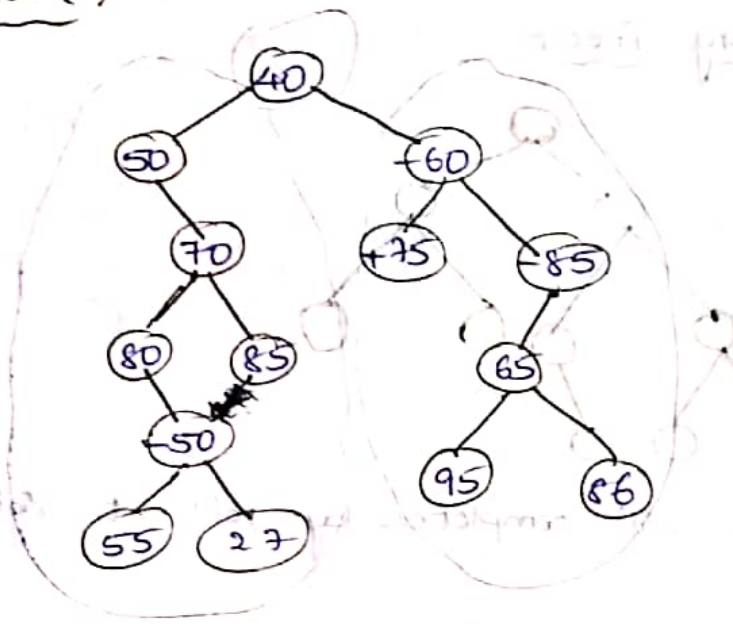


Pre-order :- A, B, D, E, H, C, F, I, J, G.

In-order :- D, B, H, E, A, I, F, J, C, G.

Post-order :- D, H, E, B, I, J, F, G, C, A.

* Example-2 :-



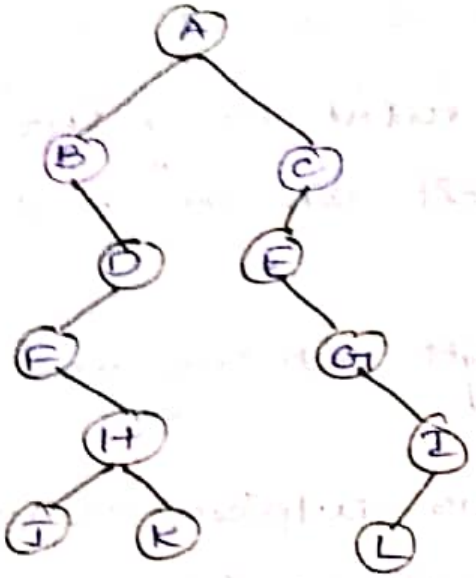
Pre-order :- 40, 50, 70, 80, ~~85~~, -50, 55, 27, 85, -60, 75, ~~+~~

~~In-order :-~~ 65, 95, 86.

Post-order :- ~~55~~ 27, -50, 80, 85, 70, 50, 75, ~~+~~95, 86, 65, -85, -60, 40.

In-order :- 50, 80, ~~70~~ 85, 55, -50, 27, 70, 85, 40, 75, -60, 95, 65, 86, -85.

* Example-3 :-

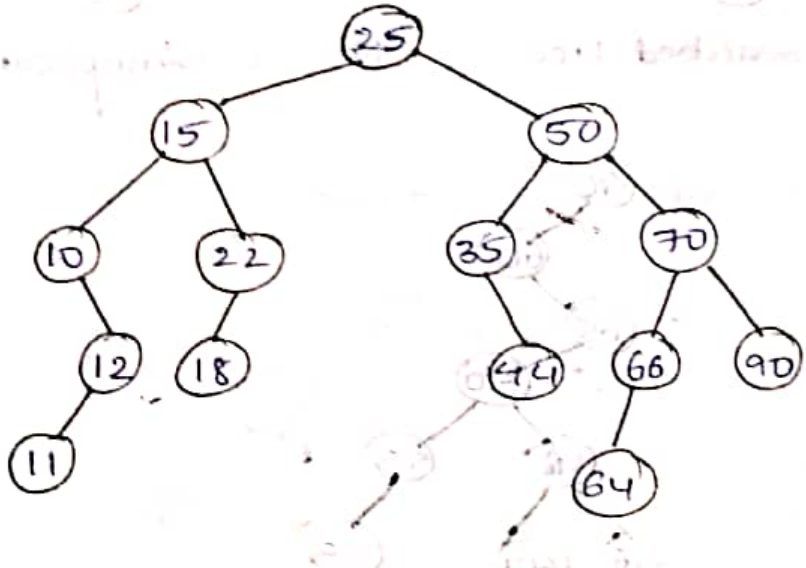


Pre-order :- A, B, D, F, H, J, K, C, E, G, I, L.

In-order :- B, D, J, H, K, D, A, E, G, L, I, C.

Post-order :- J, K, H, F, D, B, L, I, G, E, C, A.

* Example-4 :-



Pre-order :- 25, 15, 10, 12, 11, 22, 18, 50, 35, 44, 70, 66, 64, 90.

In-order :- 10, 11, 12, 15, 18, 22, 25, 35, 44, 50, 64, 66, 70, 90

Post-order :- 11, 12, 10, 18, 22, 15, 44, 35, 64, 66, 90, 70, 50, 25

* Binary Searched Tree [BST]

Properties:-

(1) It should be a rooted Binary tree.

(2) The values of Left sub tree must be less than root Node value.

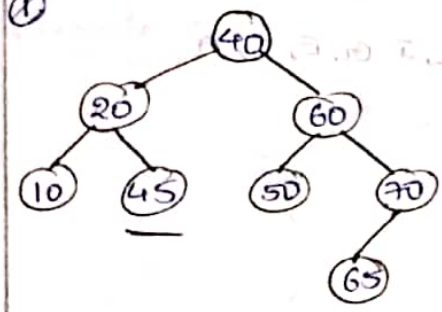
(3) The values of Right sub tree must be greater than root Node value.

(4) It doesnot allows Dulpicate values.

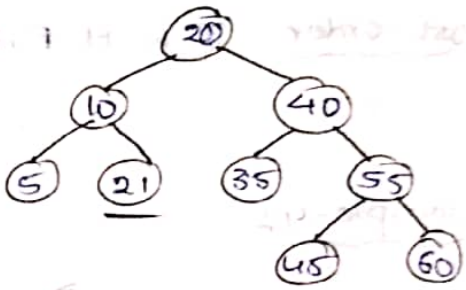
(5) Properties (2), (3) will be followed by every subtree

Examples:-

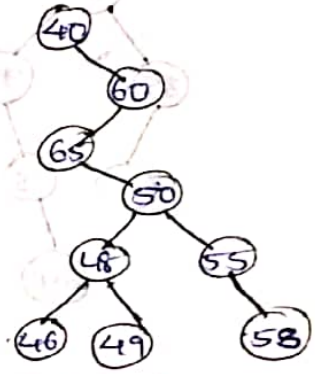
(1)



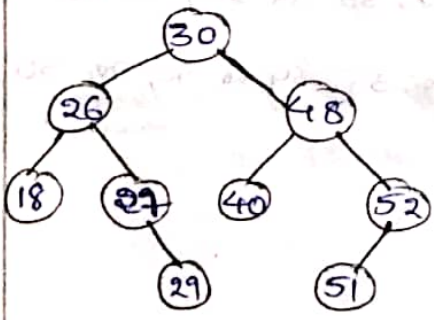
Not a Binary searched tree.



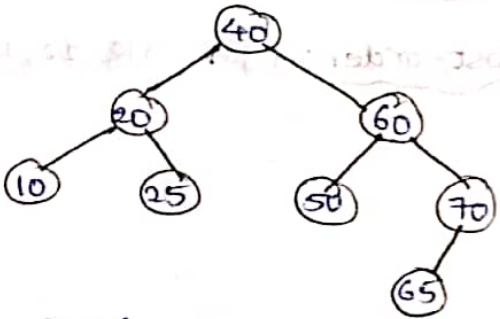
Not a Binary searched tree



It is a Binary searched tree.



It is a Binary Search Tree.



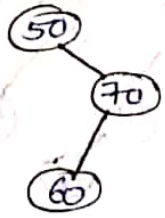
It is a Binary searched tree

* Create a B.S.T using the following sequence of values:

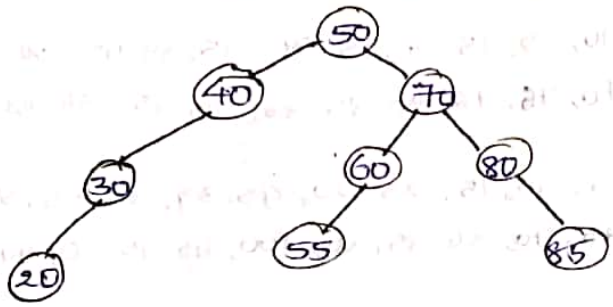
50, 70, 60, 55, 80, 85, 40, 30, 20, 45, 48, 65, 95, 86, 76, 46, 36, 26, 16, 6.

N) → The value inserting at first will be root value.

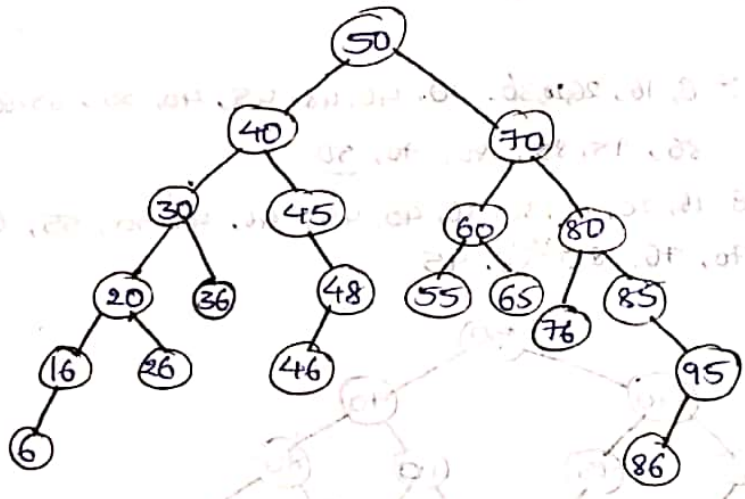
→ compare the inserting value with root value and assign accordingly.



→ If Inserting value is less than root, then traverse Left otherwise traverse Right.



→ Likewise complete the Entire tree step by step.



→ In this way we will create a Binary searched tree satisfying the properties.

Pre-order :- 50, 40, 30, 20, 16, 6, 26, 36, 45, 48, 46, 70, 60, 55, 65, 80, 76, 85, 95, 86.

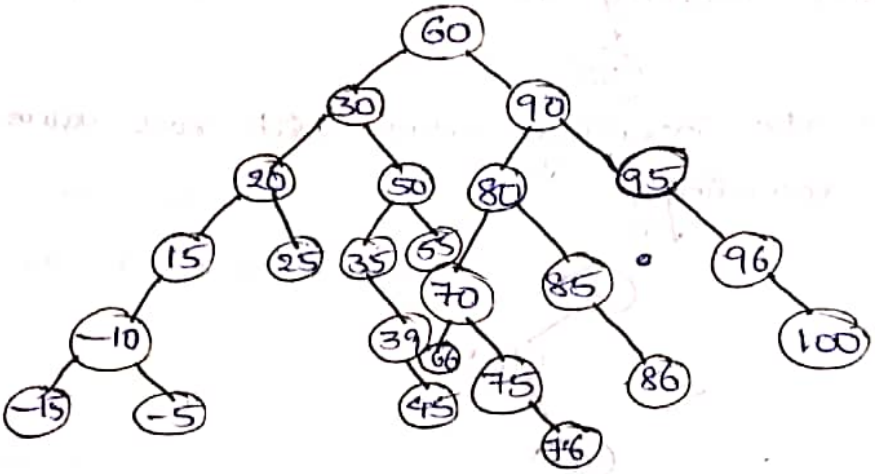
In-order :- 6, 16, 20, 26, 30, 36, 40, 45, 46, 48, 50, 55, 60, 65, 70, 76, 80, 85, 86, 95.

Post-order :- 6, 16, 26, 20, 36, 30, 46, 48, 45, 40, 55, 65, 60, 76, 86, 95, 85, 80, 70, 50.

* Create a B·S·T using the following values:

60, 90, 30, 50, 20, 80, 70, 85, 75, 95, 25, 15, 35, 39, 45, 55, 96, 86, 76, 66, −10, −5, −15, 100.

A)



Pre-order :- 60, 30, 20, 15, −10, −15, −5, 25, 50, 35, 39, 45, 55, 90, 80, 70, 66, 75, 76, 85, 86, 95, 96, 100.

In-order :- −15, −10, −5, 15, 20, 25, 30, 35, 39, 45, 50, 55, 60, 66, 70, 75, 76, 80, 85, 86, 90, 95, 96, 100.

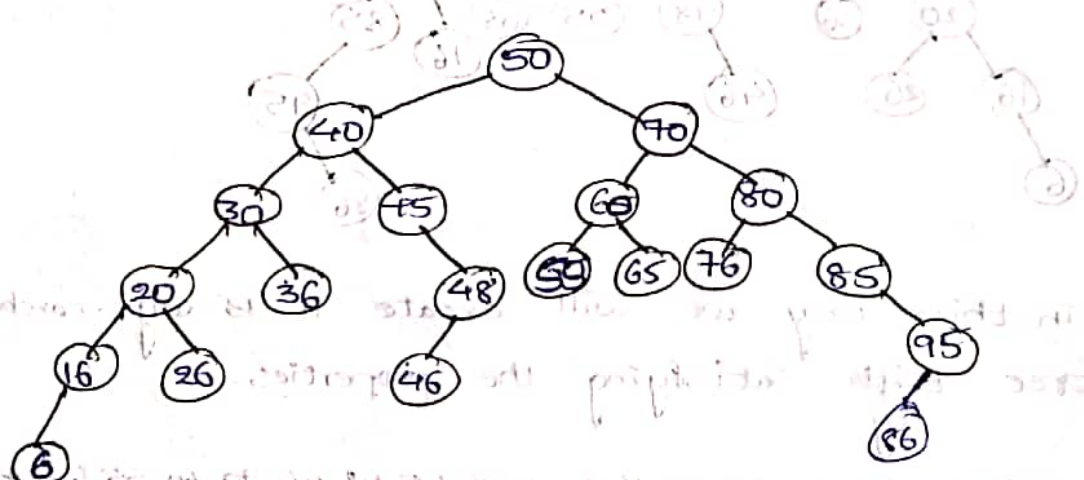
Post-order :- −15, −5, −10, 15, 25, 20, 45, 39, 35, 55, 50, 30, 66, 76, 75, 70, 86, 85, 80, 100, 96, 95, 90, 60.

* Create a B·S·T using its In-order & post-order traversals?

Post-order :- 6, 16, 26, 20, 36, 30, 46, 48, 45, 40, 55, 65, 60, 76, 86, 95, 85, 80, 70, 50.

In-order :- 6, 16, 20, 26, 30, 36, 40, 45, 46, 48, 50, 55, 60, 65, 70, 76, 80, 85, 86, 95.

A)



* Structure of a Binary search Tree

```
struct node
{
  struct node *left;
  int data;
  struct node *right;
} * root = NULL, *p, *temp;
```

root = always points to root / center of a tree.
P = to traverse the tree.
temp = to insert / delete a Node.

* Algorithm for creation :-

```
struct node *newNode (int key);
```

step-1 :- create a new node i.e. temp.

step-2 :- set temp → data = key,
set temp → left = NULL,
set temp → right = NULL.

step-3 :- return temp.

* Algorithm for insertion :-

```
struct node * Insert (struct node *root, int key);
```

step-1 :- if (root == NULL).
(i) return newNode (key)

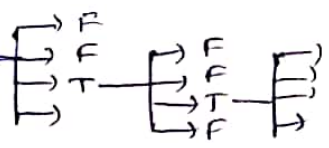
step-2 :- else if (key < root → data).
(ii) root → left = insert (root → left, key).

step-3 :- else if (key > root → data)
(i) root → right = insert (root → right, key).

step-4 :- return root.

Main function :-

root = insert (root, 40).
(or)
root = insert (root, x)



* Algorithm for Traversal :- [In-order Traversal].

void Inorder display (struct node *root);

step-1 :- if (root != NULL)

step-2

(i) Inorder (root → left).

(ii) Display "root → data"

(iii) Inorder (root → right)

step-2 :- EXIT.

* Algorithm for Pre-order Traversal :-

void Preorder (struct node *root);

step-1 ; if (root != NULL)

(i) Display "root → data".

(ii) Preorder (root → left)

(iii) Preorder (root → right)

step-2 :- EXIT.

* Algorithm for Post-order Traversal :-

void Postorder (struct node *root);

step-1 :- if (root != NULL)

(i) Postorder (root → left)

(ii) Postorder (root → right)

(iii) Display "root → data"

step-2 :- EXIT.

* Deletion :-

case-1 :- If deleting node is 'Leaf node' then discard it directly.

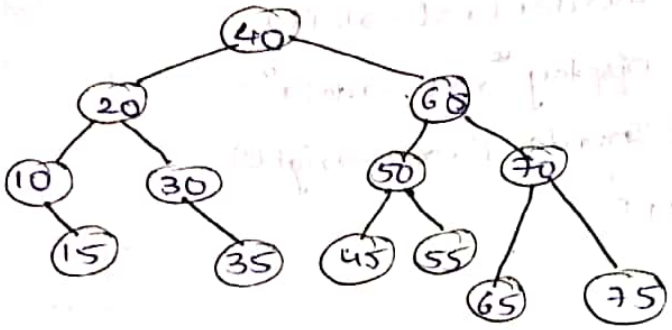
case-2 :- If deleting node contains 'one child node' then replace it with child node [either left or right]

case-3 :- If deleting node contains 'two nodes (or) two sub trees' then substitute with Inorder predecessor or successor. (Maximum value). (or) (min value).

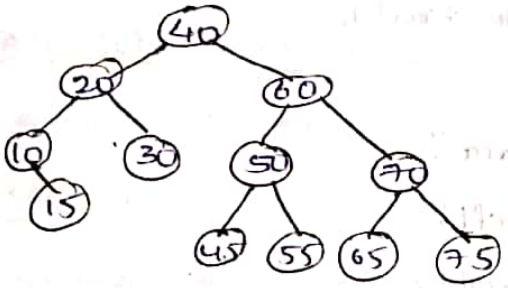
Inorder predecesor :- Maximum value of left sub tree

Inorder successor :- Minimum value of right sub tree

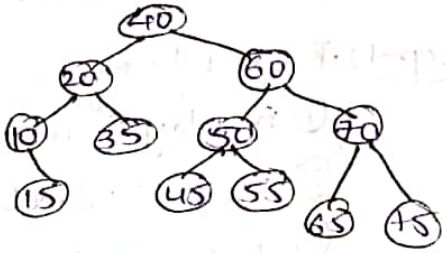
Ex :-



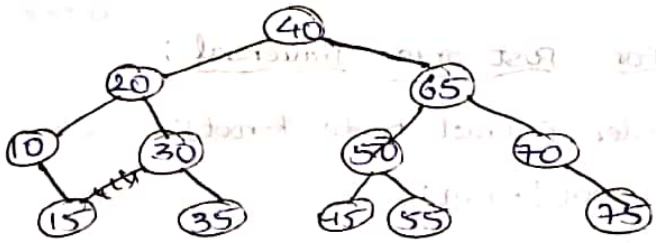
case-1 :- Delete 35



case-2 :- Delete 30



case-3 :- Delete 60



* Algorithm for Deletion :-

struct node * deleteNode (struct node * root, int key);

step-1 :- if (root == NULL)

(i) return NULL

step-2 :- else if (key < root → data)

(i) root → left = deleteNode (root → left, key)

step-3 :- else if (key > root → data)

(i) root → right = deleteNode (root → right, key)

step-4 :- else

(i) if (root → left == NULL)

(a) set temp = root → right

(b) free(root)

(c) return temp.

(ii) else if (root → right == NULL)

(a) set temp = root → left

(b) free (root)

(c) return temp.

(iii) set temp = minValue BST (root → right)

(iv) Set root → data = temp → data.

(v) root → right = deleteNode (root → right, temp → data).

Step-5 :- return root.

* Algorithm for Inorder successor :-

struct node * minValue BST (struct node * root);

step-1 :- set p = root

step-2 :- while (P != NULL && P → left != NULL)

(i) set p = p → left.

step-3 :- return p.

* Algorithm for searching :-

struct node * searchBST (struct node * root, int key);

step-1 :- if (root → data == key OR root == NULL)

(i) return root.

step-2 :- else if (key < root → data)

(i) return searchBST (root → left, key);

step-3 :- else

(i) return searchBST (root → right, key);

* Algorithm for counting total no. of nodes :-

int totalNodesBST (struct node * root)

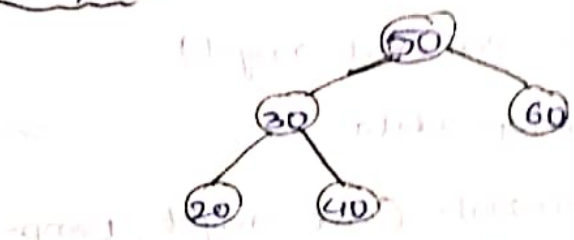
step-1 :- if (root == NULL)

ⓘ return 0.

step-2 :- else

ⓘ return totalNodesBST (root → left) +
totalNodesBST (root → right) + 1.

Example :-



t.N.BST (50) ← 1. False
2. return tNBST (30) + tNBST (60) + 1

1. False | 2. return tNBST (20) + tNBST (40) + 1

1. False | 2. return tNBST (NU + tNBST (Nu + 1

* Algorithm to find internal Nodes in B.S.T :-

int internalnodesBST (struct node * root)

step-1 :- if (root == NULL)

ⓘ return 0

step-2 :- else if (root → left == NULL && root → right == NULL)

ⓘ return 0

step-3 :- else

ⓘ return internalnodesBST (root → left) +
internalnodesBST (root → right) + 1

* Algorithm to find External Nodes:-

int externalBST (struct node * root)

Step-1 :- if (root == NULL)

(i) return 0

Step-2 :- else if (root → left == NULL && root → right == NULL)

(i) return 1

step-3 :- else

(i) return externalBST (root → left) + externalBST (root → right) + 1

* Algorithm for Height of a BST :-

int height BST (struct node *root);

step-1 :- if (root == NULL)

(i) return 0

step-2 :- else

(i) LH = height BST (root → left)

(ii) RH = height BST (root → right)

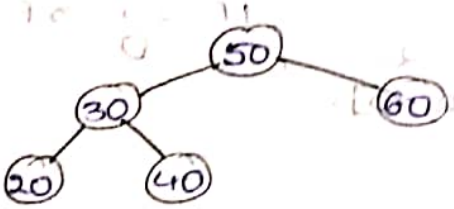
(iii) if (LH > RH)

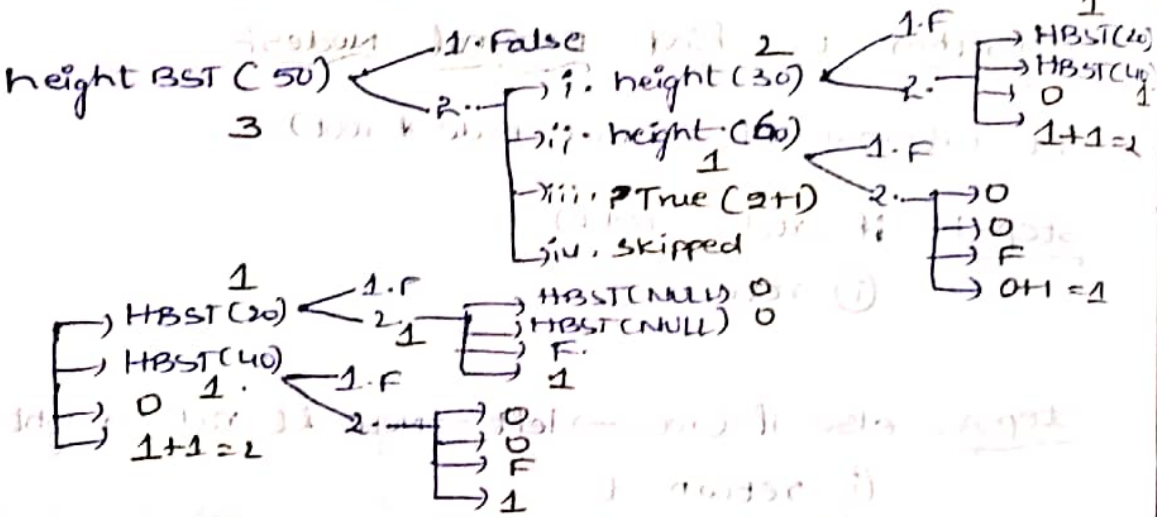
(a) return LH + 1

(iv) else

(a) return RH+1

Example :-





∴ Height of given tree = 3 − 1 = 2. [∵ root node starts from '0'].

Note :-

→ Most of the tree root Node is starts from '0'. so, we need to reduce '1' to get exact height of the Tree [BST].

* Heap Tree :-

→ Heap Tree follows the properties of complete Binary Tree. [Nodes fill from Left to right. by Level by Level]

Types of Heap tree :

① Max Heap tree.

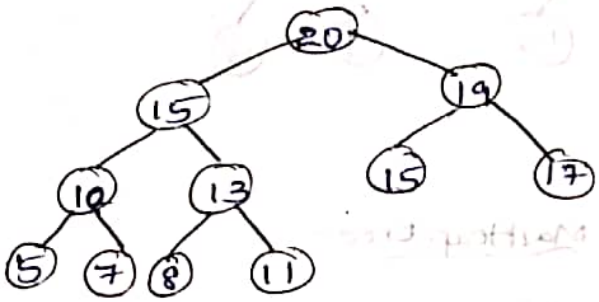
② Min Heap tree.

① Max Heap tree :-

→ The 'root node value' is more than all node values.

→ If we consider any sub tree, the 'parent node value' is must greater than 'child node values'.

Ex :-

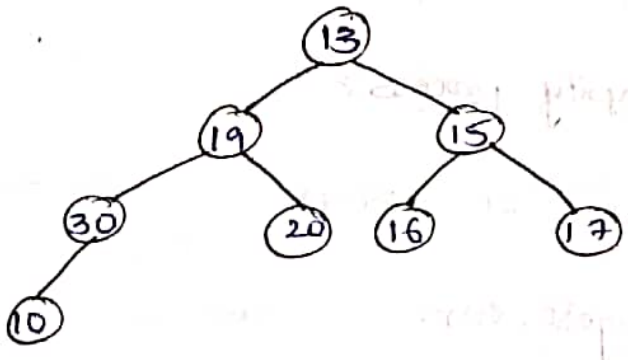


② Min Heap tree :-

→ The root node value is 'smaller' than all node values.

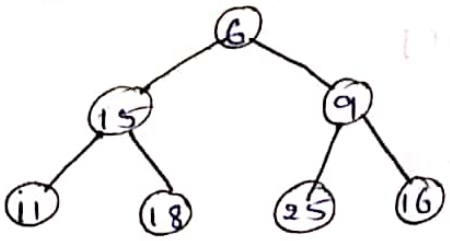
→ If we consider any subtree, the 'parent node value' is must smaller than 'child node values'.

Ex :-



* Ex:- 6, 15, 9, 11, 18, 25, 16.

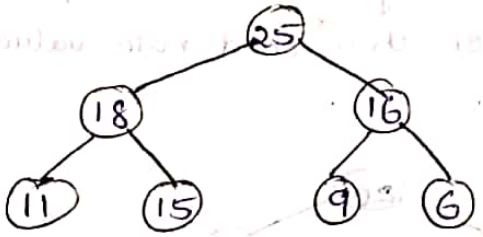
A) Complete Binary Tree :-



Heapify :- heapify(a, n, i).

$i = \frac{n}{2} - 1 = \frac{7}{2} - 1$. & $i >= 0$.

We will build complete max Heap tree by using Heapify procedure:



* Algorithm for MaxHeap tree:-

BuildMaxheap(a, n);

step-1 :- Declare i.

step-2 :- for(i = $\frac{n}{2}$ - 1; i >= 0; i--)

① heapify(a, n, i)

step-3 :- EXIT.

* Algorithm for Heapify process:

heapify(a, n, i);

void heapify(int a[], int n, int i)

step-1 :- Declare largest, l, r.

step-2 :- set largest = i.

step-3 :- compute l = 2xi+1 , r = 2xi+2 .

step-4 :- If (l < n && a[l] > a[largest])

(i) set largest = l ;

step-5 :- if (r < n && a[r] > a[largest])

(i) set largest = r ;

step-6 :- if (largest != i)

(i) swap (&a[i], &a[largest])

(ii) heapify (a, n, largest).

step-7 :- EXIT .

* Algorithm to Build MinHeapTree :-

step-1 :- Declare i.

step-2 :- for(i = $\frac{n}{2}$ - 1; i>=0; i--)

(i) heapify (a, n, i)

step-3 :- EXIT

heapify procedure :-

step-1 :- Declare smallest, l, r.

step-2 :- set smallest = i.

step-3 :- compute l = 2*i+1

(ii) r = 2*i+2

step-4 :- if(l<n && a[l] < a[smallest])

(i) smallest = l.

step-5 :- if(r<n && a[r] < a[smallest])

(i) smallest = r.

step-6 :- if(smallest != i)

(i) swap(&a[i], &a[smallest])

(ii) heapify(a, n, smallest)

step-7 :- EXIT.

* Algorithm for Heapsort :-

step-1 :- Declare i.

step-2 :- for(i = $\frac{n}{2}$ - 1; i>=0; i--)

(i) heapify (a, n, i).

step-3 :- for(i=n-1; i>=0; i--)

(i) swap(&a[0], &a[i])

(ii) heapify(a, i, 0)

step-4 :- EXIT.

* Expression Tree :-

→ It is a special kind of Binary Tree which is used to represent expressions.

→ An Expression is composed of 'variables', 'constants' and 'operators' which are arranged according to rules.

Example :-

→ An 'Expression tree' is a representation of an expression in the form of tree like data structure.

→ Operands are leaves [Leaf Nodes] in the tree.

→ Operators are Internal nodes in the expression tree.

* Example :-

Convert the following Infix expression into Binary expression tree :- a * b/c + e/f * g + K - x * y.

A)

Operator		Precedence & Associativity
^	→	Right to left
*, /	→	Left to right
-, +	→	Left to right.

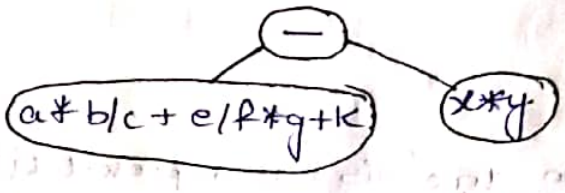
Rules :-

→ Highest preference operators are stored near to the Leaf nodes. Because they are going to be evaluated first.

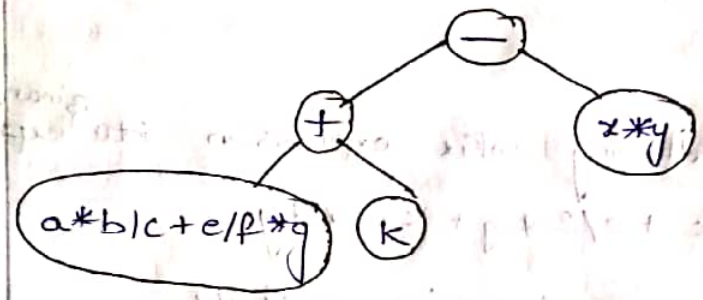
→ The lowest preference operator will be stored in Root Node and it is evaluated at last.

A) Given expression is a * b/c + e/f * g + k - x * y

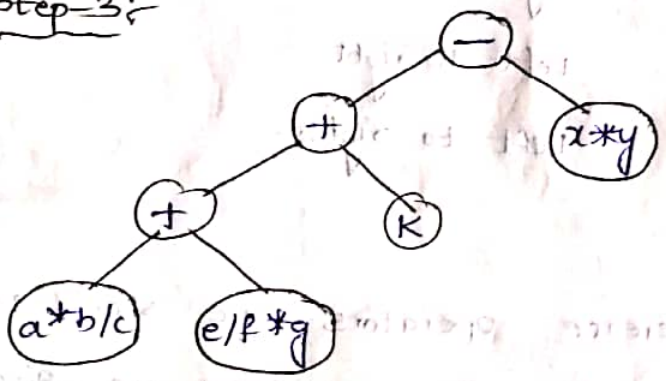
Step-1 :- check out & find the operator which is having least preference. as per associativity and store it in root node.



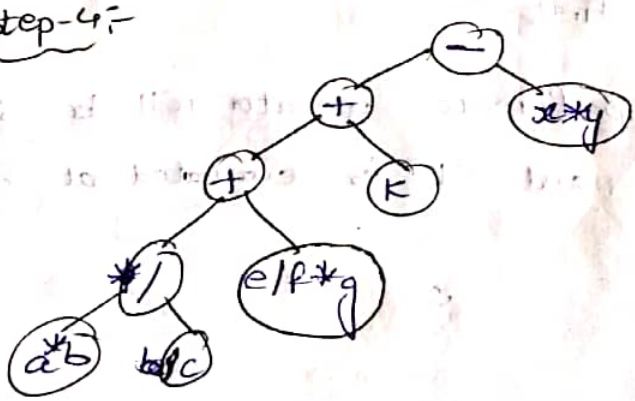
Step-2 :- Repeat the same process [find the operator with minimum precedence]. in either Right subtree (or) left subtree.



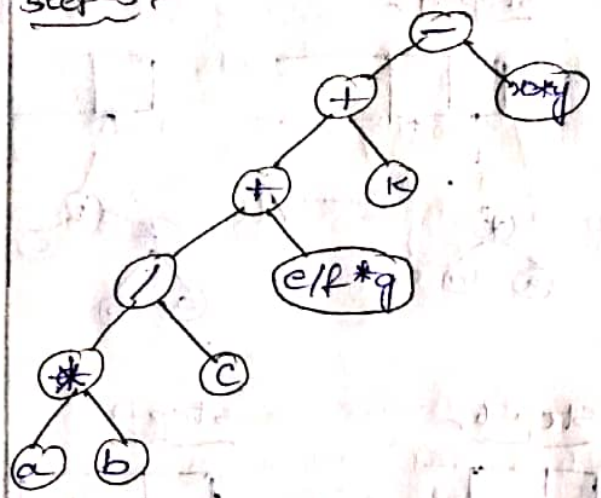
Step-3 :-



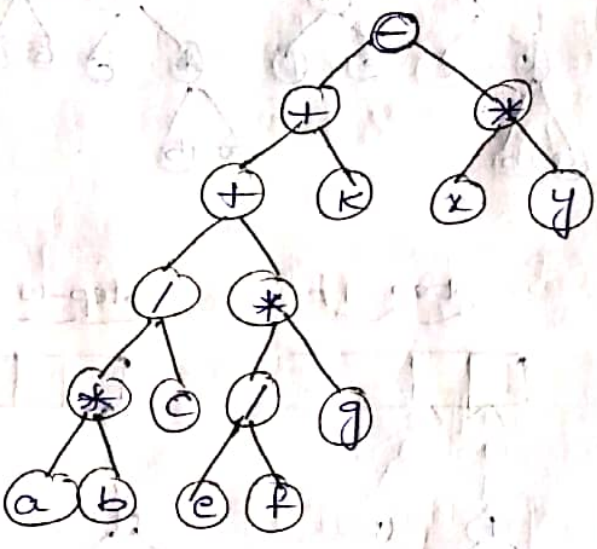
Step-4 :-



step-5:-



step-6:-



This is the required expression Tree.

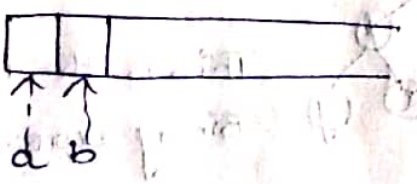
Prefix expression:- -++/*abc*/efgk**xy

Post expression:- ab*c/ef/g*+k+xy*-

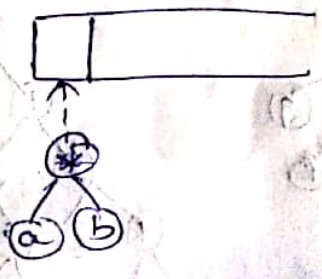
* convert the postfix expression to Binary expression tree?

ab*c/ef/g*+k+xy*-

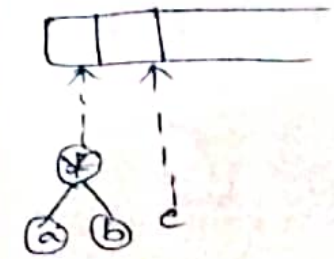
A) step-1:-



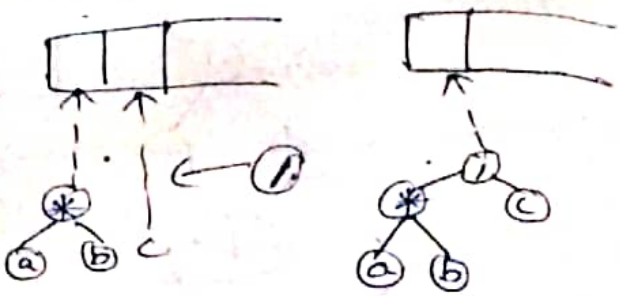
step-2:-



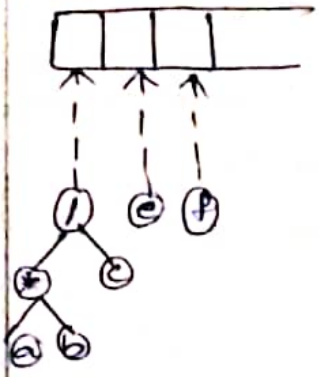
step-3 :-



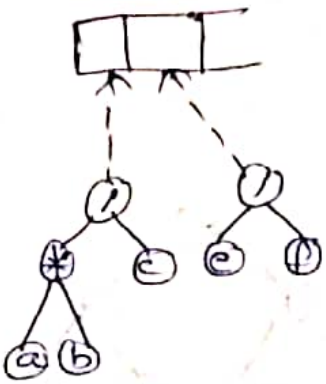
step-4 :-



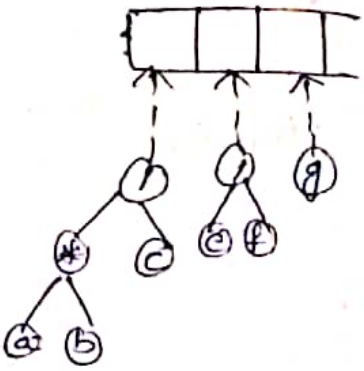
step-5 :-



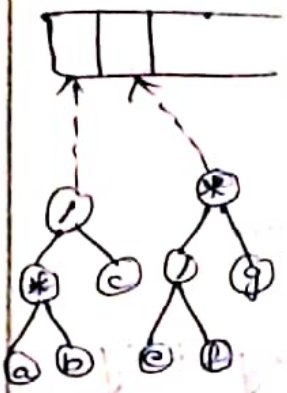
step-6 :-



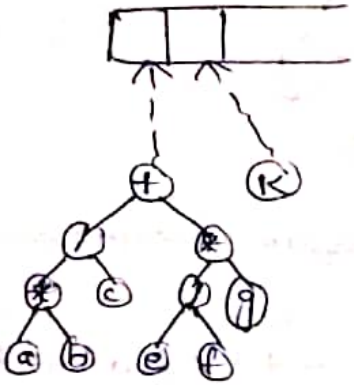
step-7 :-



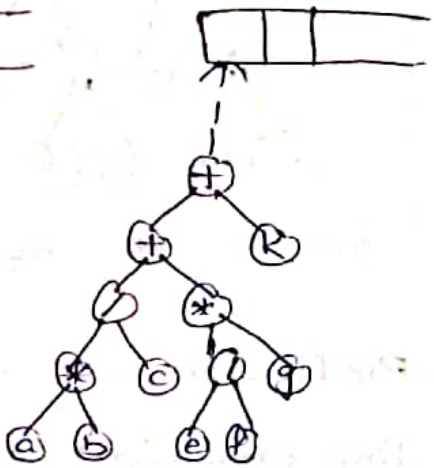
step-8 :-



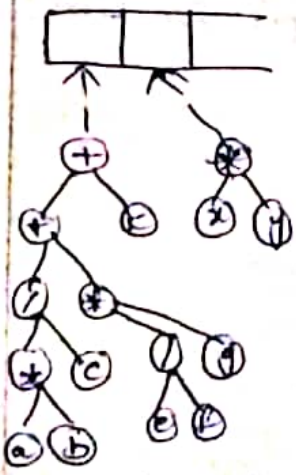
step-9 :-



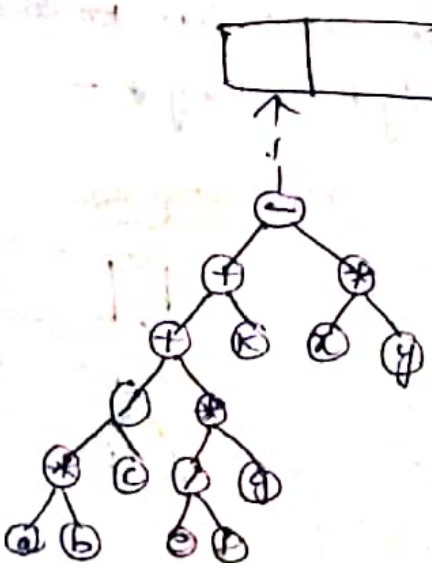
step-10 :-



step-11 :-



step-12 :-

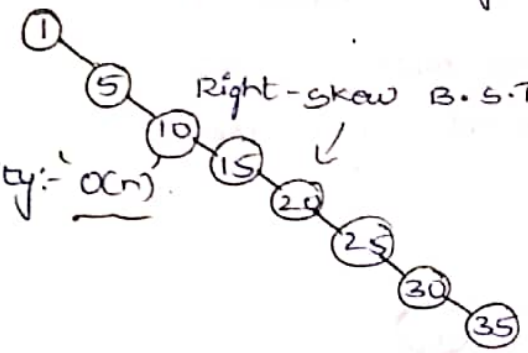


This is the required Binary expression tree

* AVL Tree(s) :- [Self Balanced B.S.T]

→ The main drawback of B.S.T is, when we want to create a B.S.T using sequence of values which are in increasing order / Decreasing order then we will get Linear List. [skew tree].

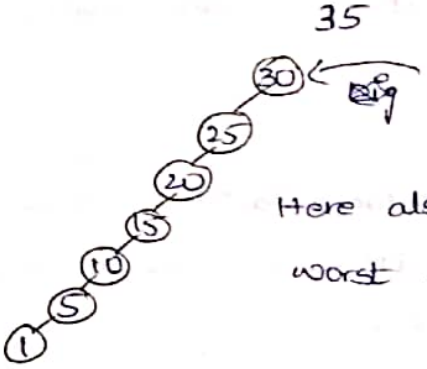
Ex:- 1, 5, 10, 15, 20, 25, 30, 35 [Increasing order].



Right-skew B.S.T

Time complexity:- O(n).

Ex:- 35, 30, 25, 20, 15, 10, 5, 1 [Decreasing order]



Left skew B.S.T

Here also, the time complexity for worst case is 'O(n)'. [For searching]

→ To overcome this Limitation, 'Adelson', 'Velski' and 'Landis' these three persons gave a concept with their name called 'AVL Tree.'

→ It is also known as "Balanced Binary search Tree."

Properties of AVL Tree :-

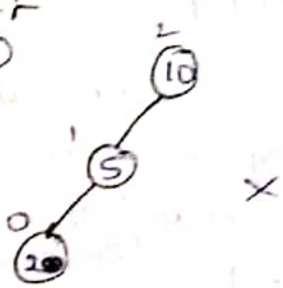
1. It should follow the all properties of 'Binary Tree' [Having atleast two child nodes].
2. It should follow the all properties of 'Binary search Tree'.
3. **Balance factor = Left subTree's Height − Right subTree's height**

→ At any point of time for a given tree, the height of tree will be '0, −1, 1'.

→ The difference b/w heights of Leftsubtree and Rightsubtree should not be more than '1'.

Ex:-

(1)



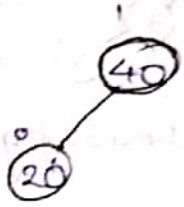
Height of Leftsubtree = 2
Height of Right subtree = 0
∴ Balance factor = 2 - 0 = 2
∴ It is not a AVL Tree.

(2)

Balance factor = 1 - 0
= 1
∴ It is an AVL Tree.

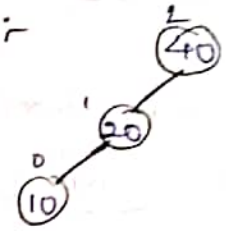
Example:-

step-1:-



Balance factor = 1 - 0 = 1
∴ It is an AVL Tree.

step-2:-



Balance factor = 2 - 0 = 2
∴ It is not an AVL Tree.

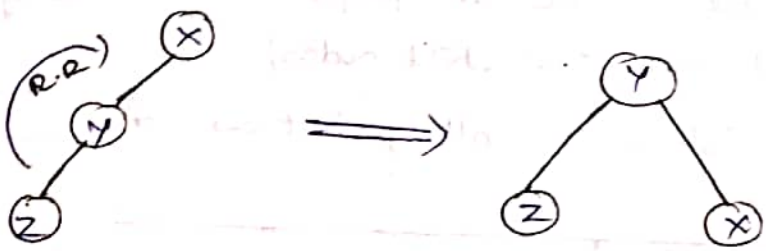
* Insertion:-

Case-1:- "L-L Insertion"

→ 'L-L Insertion' means inserting a node for at 'Left subtree of Left subtree'.

→ Here we will apply 'Right Rotation' [RR]

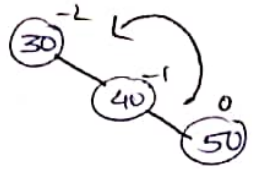
Ex:-



Case-2 :- R·R Insertion"

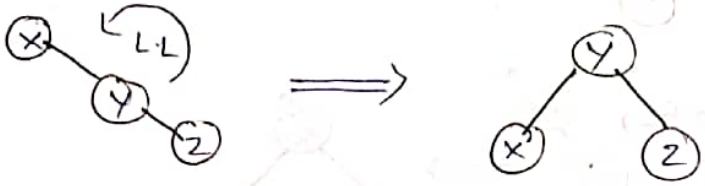
→ Insertion at Right subtree of Right subtree.

Ex :-



Balance factor = -2.
Not AVL Tree.

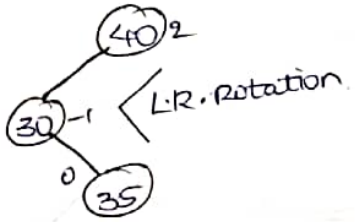
→ So, we'll use L·L [Left Rotation] here.



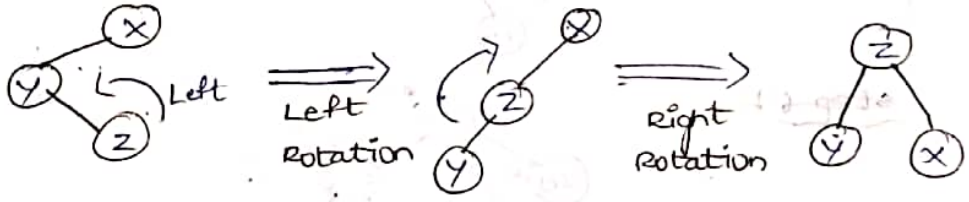
Case-3 :- "R·L Insertion"

→ Insertion at Right subtree of Leftsubtree.

Ex :-



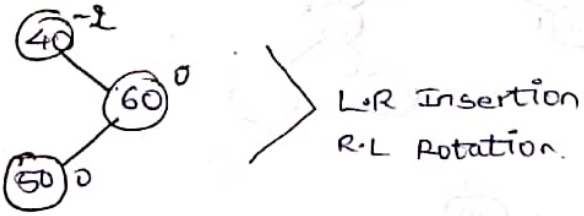
→ So we'll apply L·R Rotation here.



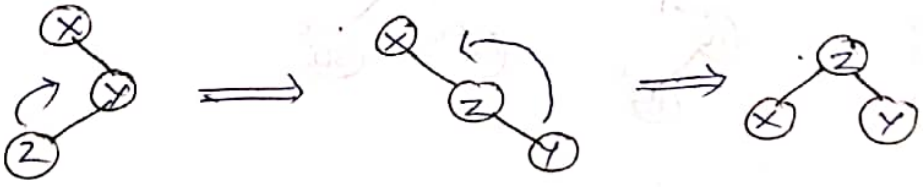
Case-4 :- "L·R Insertion"

→ Insertion at Left subtree of Right subtree.

Ex :-



→ We need to apply Right Rotation first then Left rotation.

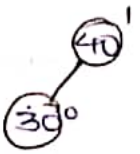


* Create an AVL Tree by using the following values

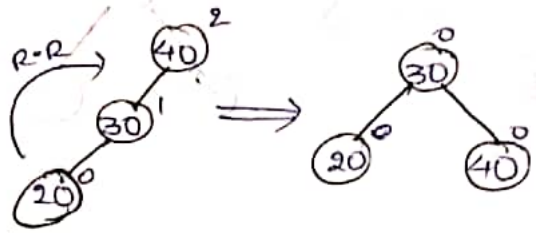
40, 30, 20, 50, 45, 10, 15, 5, 55, 60, 65, 70, 68, 80, 75, 85, 90.

A) step-1 :-

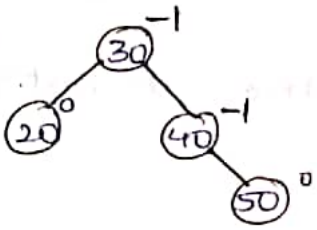
step-2 :-



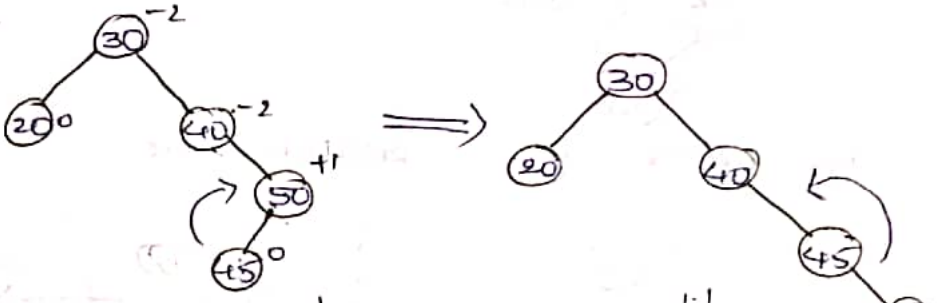
step-3 :-



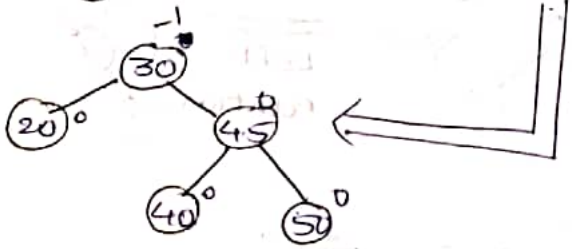
step-4 :-



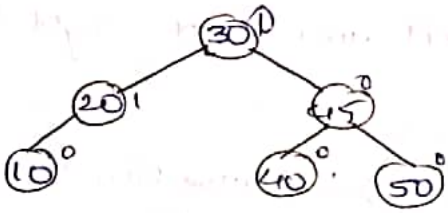
step-5 :-



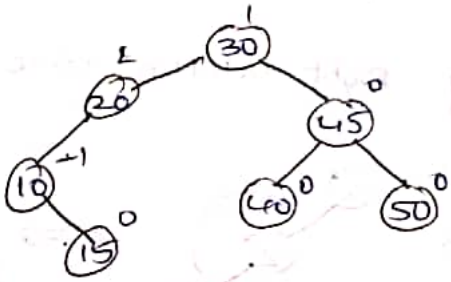
step-6 :-



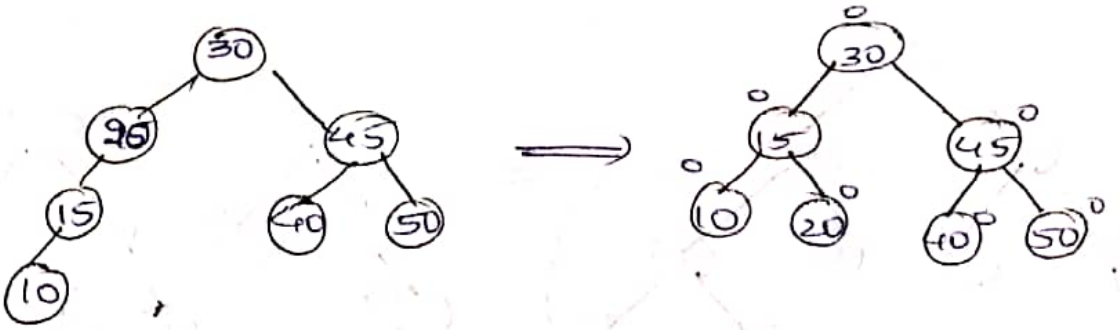
step-7 :-



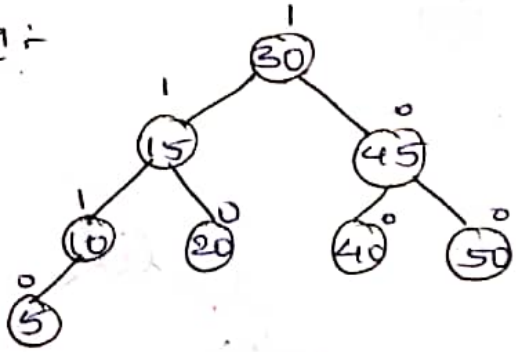
step-8 :-



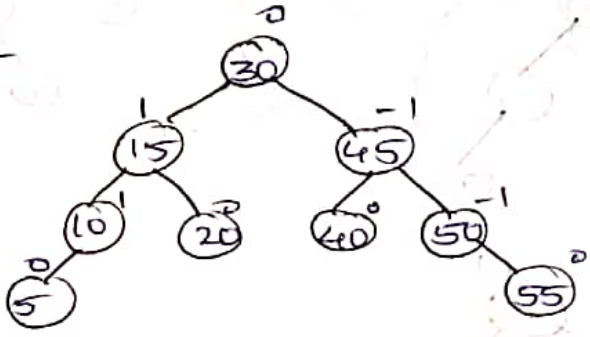
→ Then we need to apply L-R Rotation



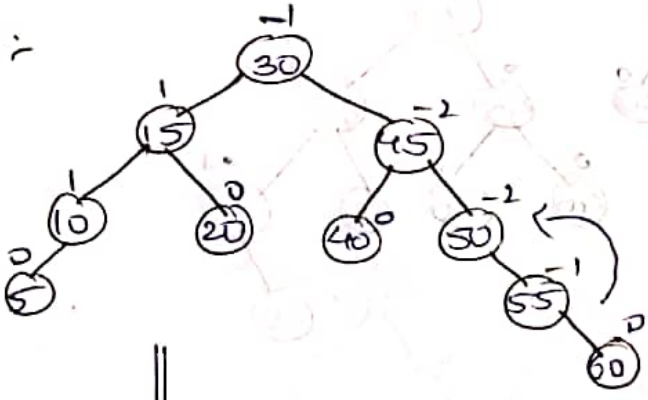
step-9 :-

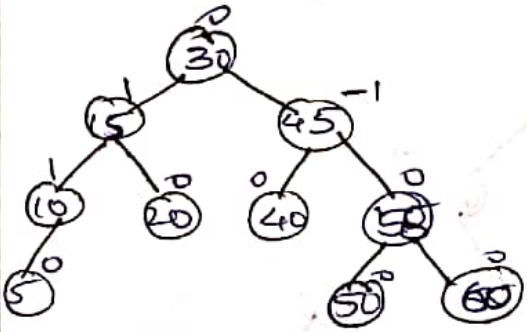


step-10 :-

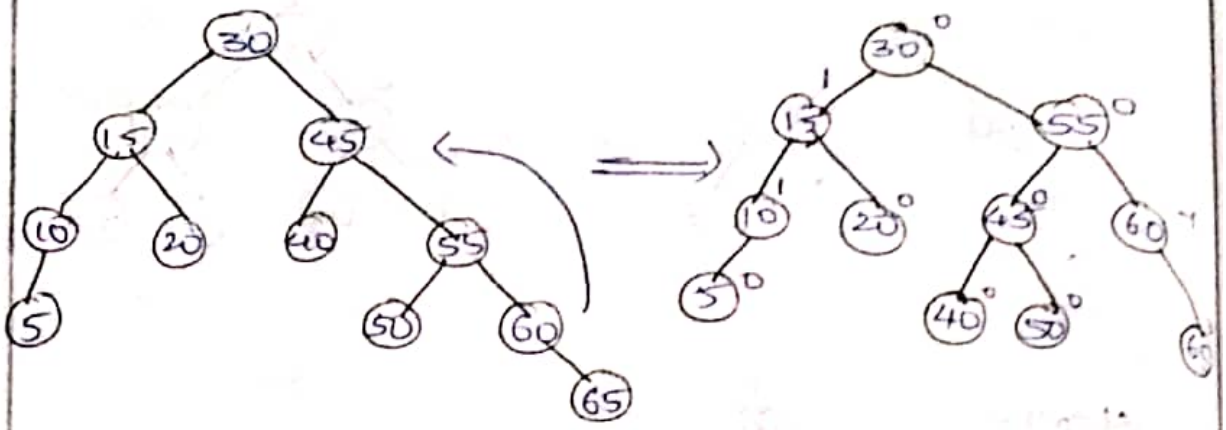


step-11 :-

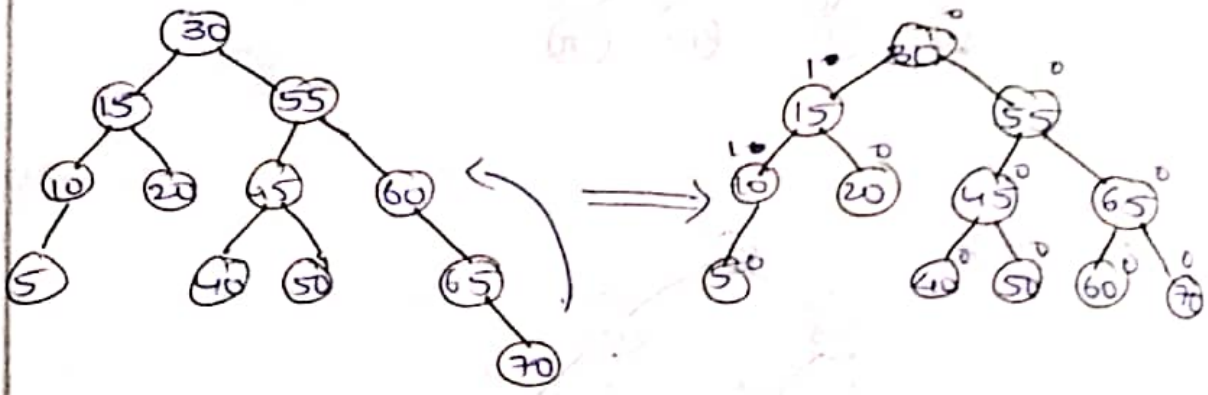




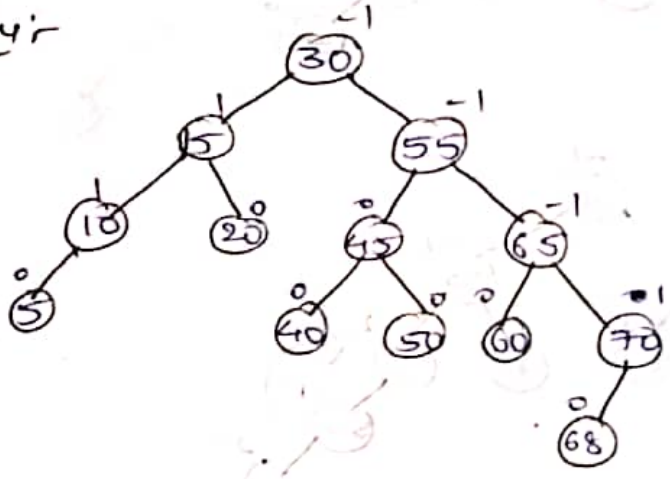
step-12 :-



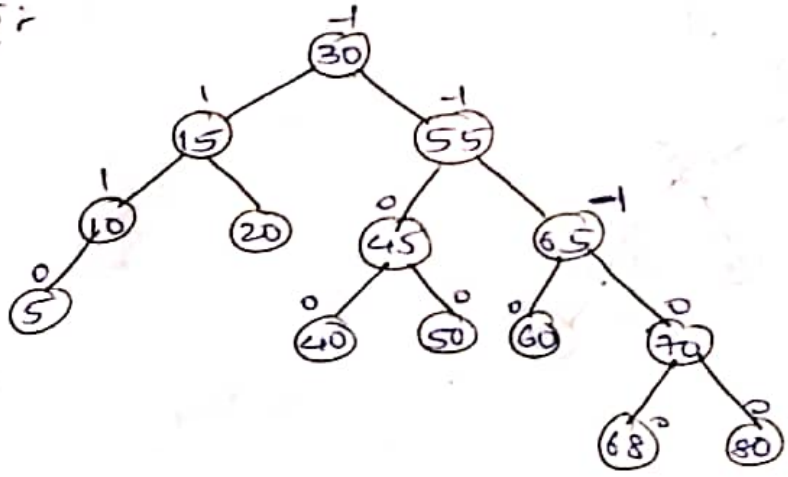
step-13 :-



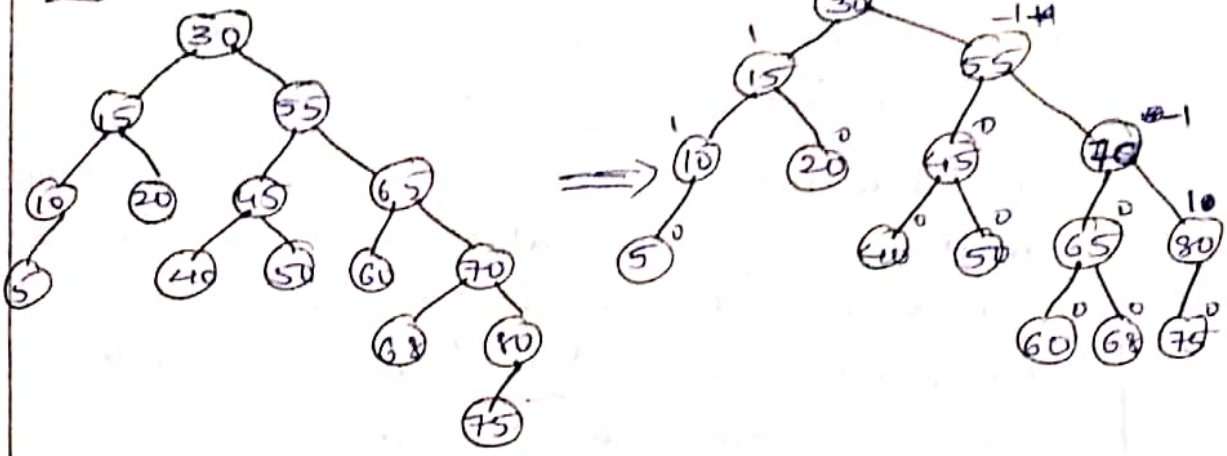
step-14 :-



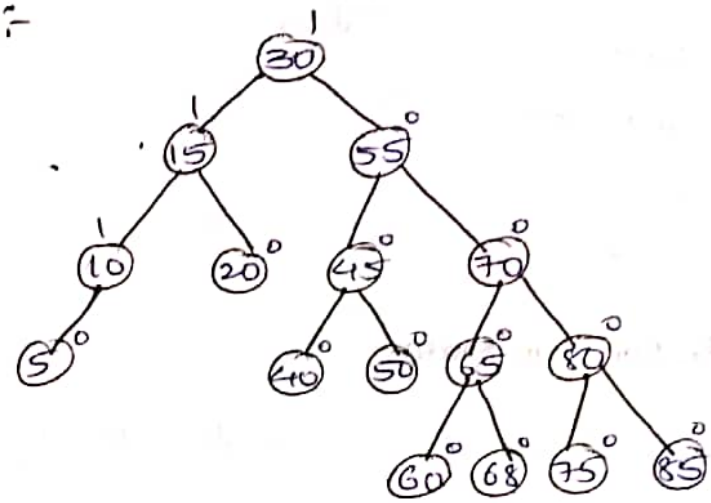
step-15 :-



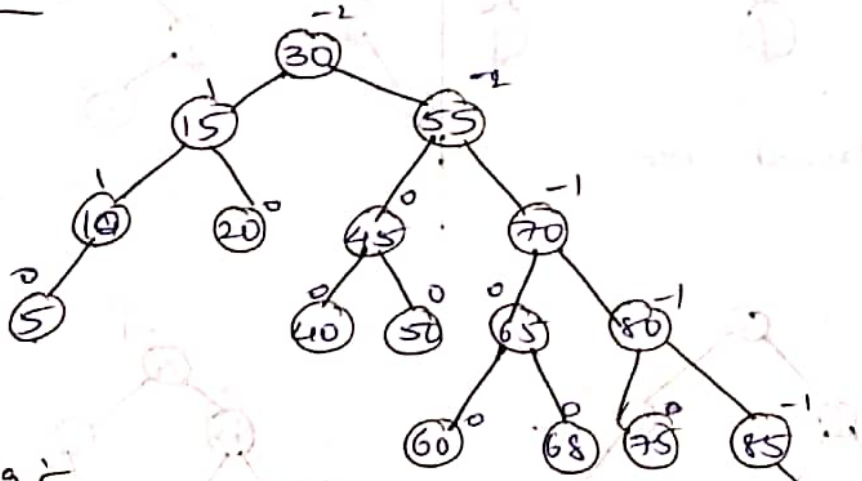
step-16 :-



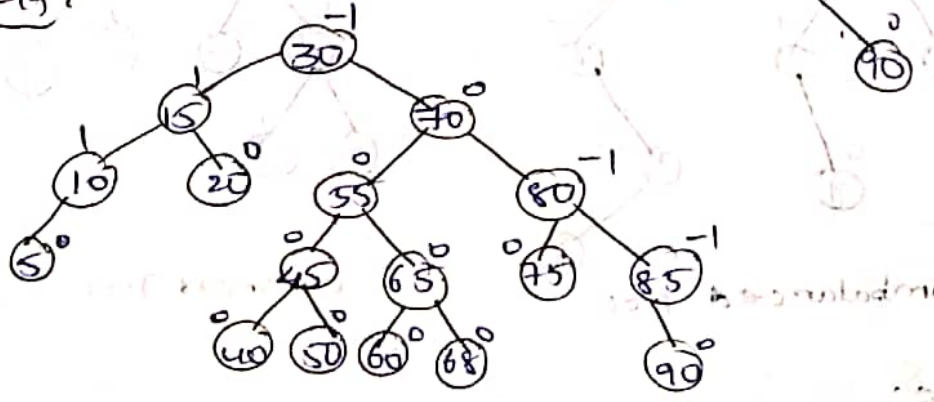
step-17 :-



step-18 :-



Step-19 :-

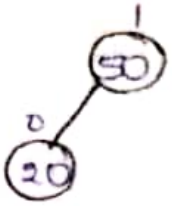


* Create an AVL Tree with the following values by using suitable Rotations

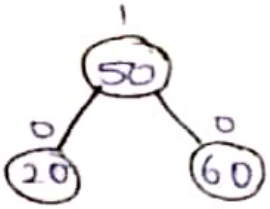
50, 20, 60, 10, 8, 15, 32, 46, 11, 48.

A) step-1 :-

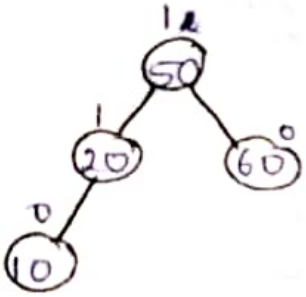
step-2 :-



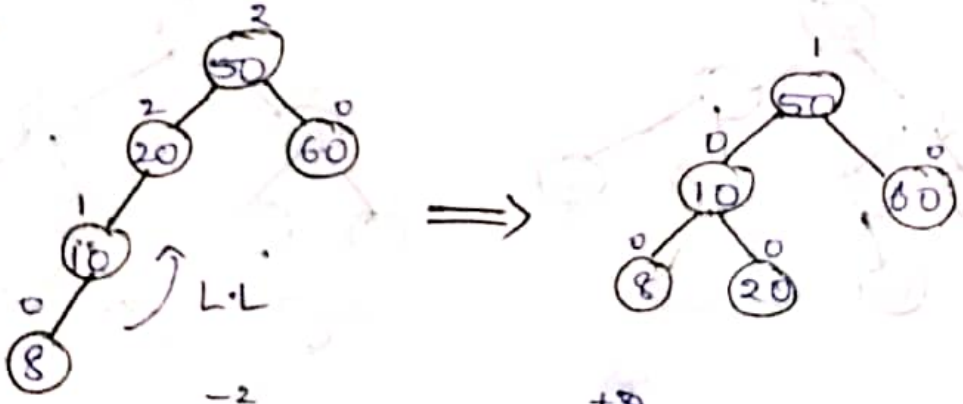
step-3 :-



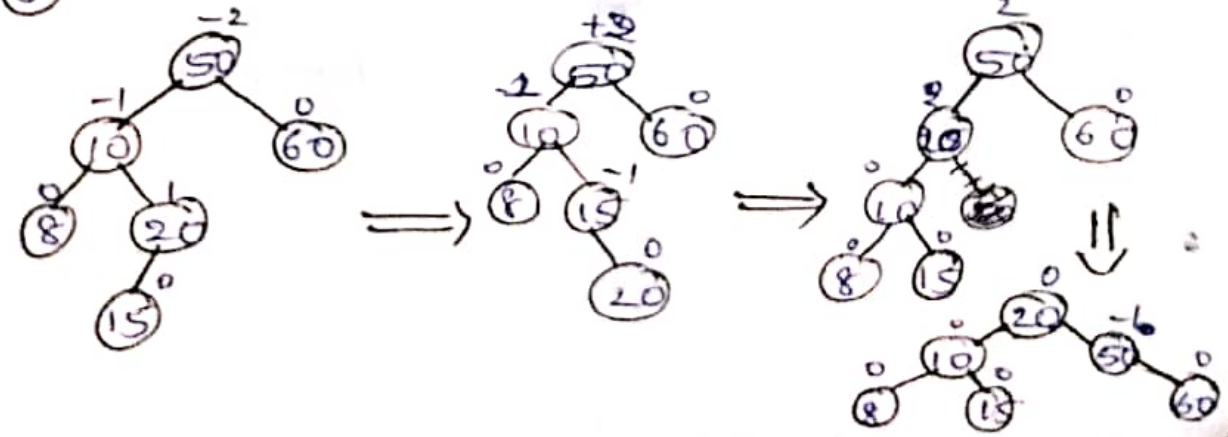
step-4 :-



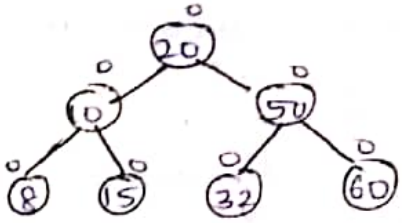
step-5 :-



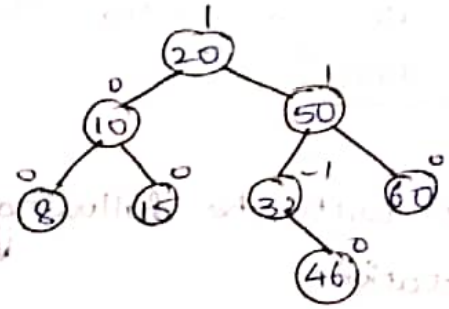
step-6 :-



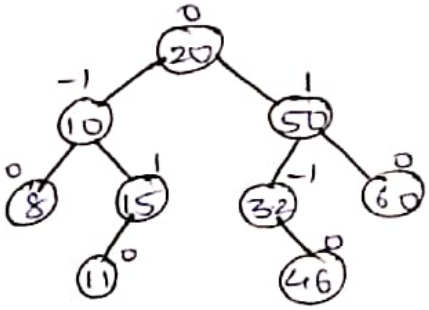
Step-7 :-



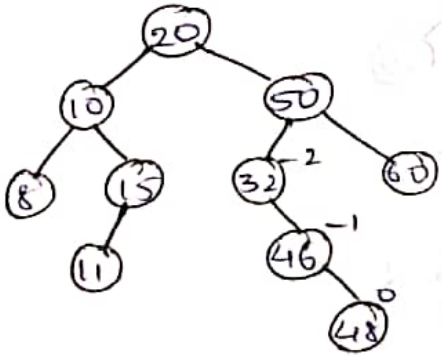
Step-8 :-

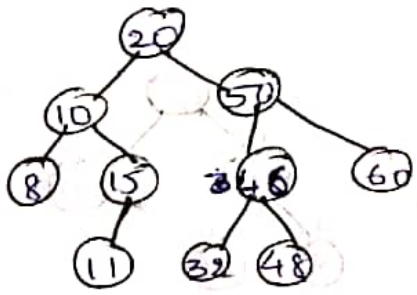


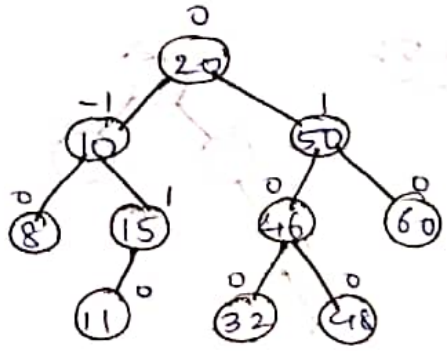
Step-9 :-



step-10 :-



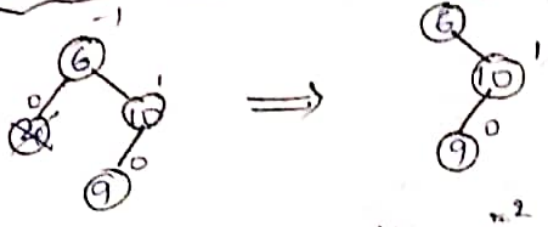




Final Tree. [AVL Tree]

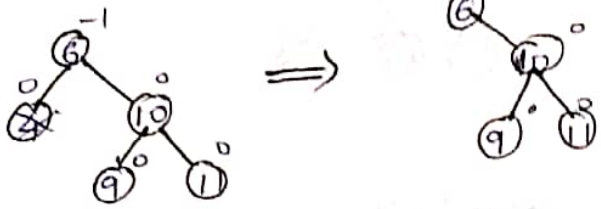
* Delete Operation in AVL Tree:-

(1)



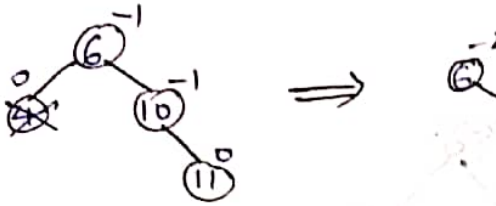
L_{-1} Notation
[R-L rotation]

(2)



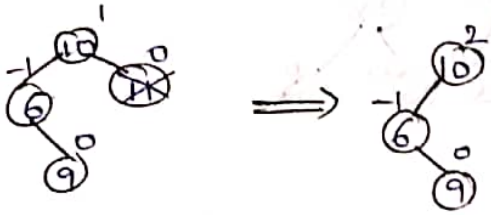
L_0 [R-R].

(3)



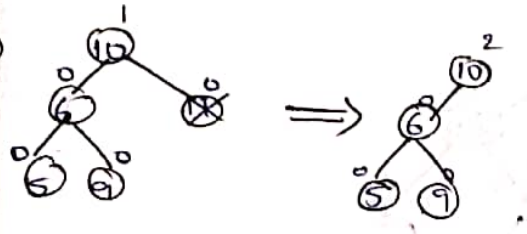
$L_{-1} \rightarrow$ R-R rotation

(4)



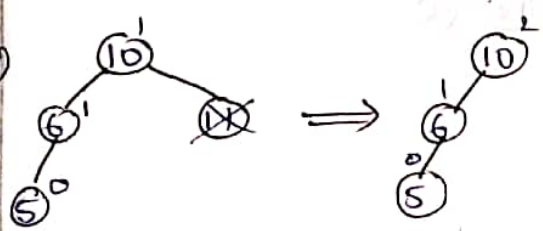
$R_{-1} \rightarrow$ L-R rotation.

(5)



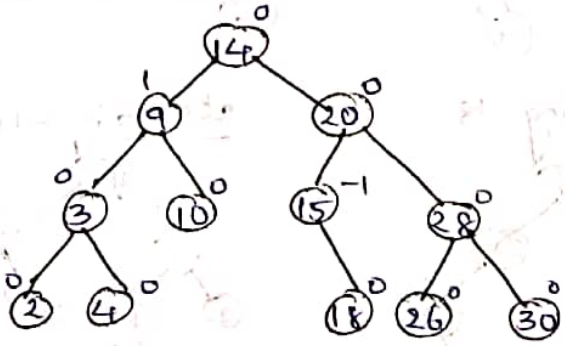
$R_0 \rightarrow$ L-L rotation.

(6)

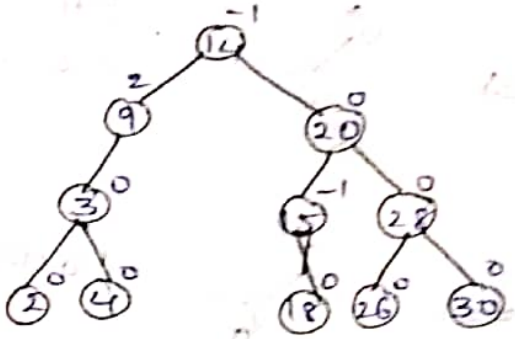


$R_{+1} \rightarrow$ L-L rotation

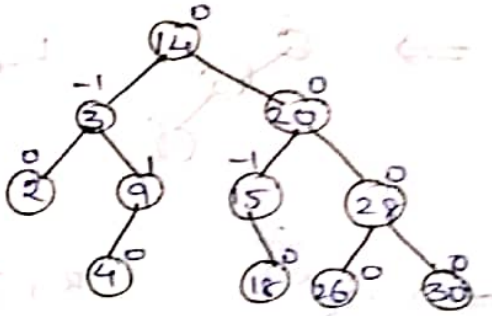
* Delete 10, 30, 15, 18 from the following tree.



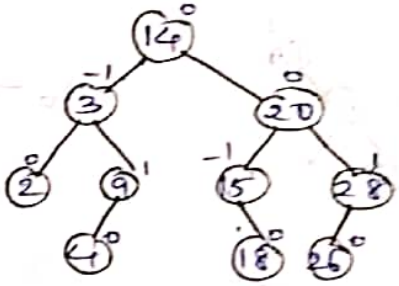
A) Step-1 :- Deleting 10.



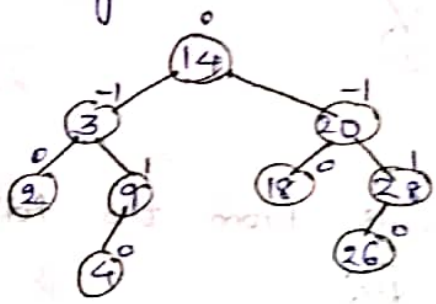
Step-2 :- R_0 [L-L Rotation]



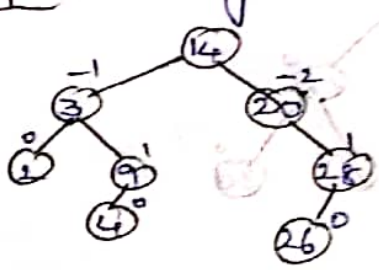
Step-3 :- Deleting 30. [No problem].



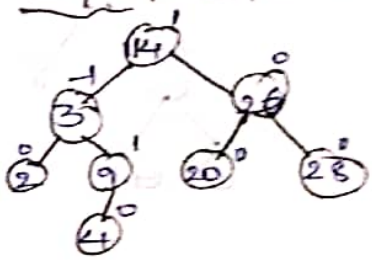
Step-4 :- Deleting 15.



Step-5 :- Deleting 18.



step-6 :- L+1 [R-L rotation]



* B-Tree :-

→ It is used to fetch the Records from 'Disk' to 'Main memory' very fastly and easily.

→ In a 'm-way tree', a node can stores 'm-1 records'

* Properties of B-Tree :-

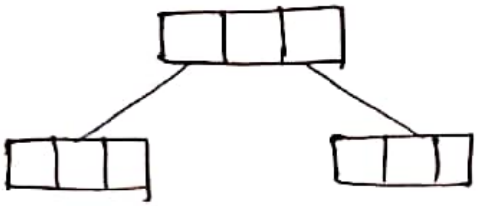
① Each Node, if order of B-Tree is 'm' then each node have 'm children' and 'm-1 ways'

② Except root node, all other nodes can have atleast 'm/2 children'. $\lceil m/2 \rceil$.

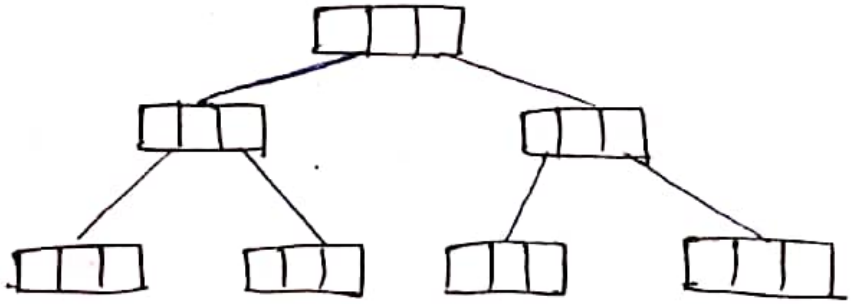
(or)

Except root node, all other nodes should have atleast 'm/2 keys' $\lceil m/2 \rceil$

③ If a root is non-terminal node, then it should have atleast '2 children'.



④ All Leaf nodes must be at same level.



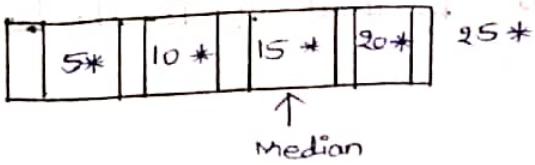
⑤ It should follow the properties of B.S.T.

* Create a B-Tree with the following values.

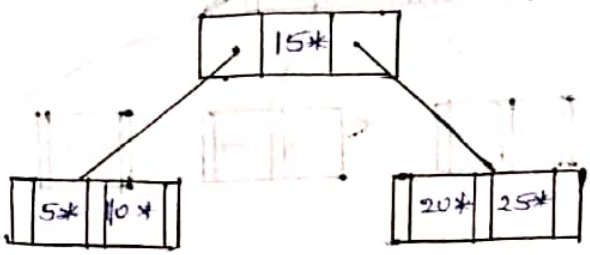
5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 95, 90, 85, 80, 75, 70, 65, 60, 55, 42, 52, 62, 72, 82, 92, 12, 22, 32, 102.

A) Let us assume the order of B-Tree as 5.

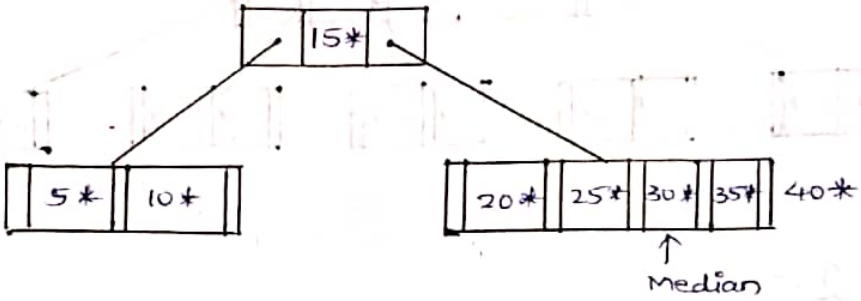
step-1 :-



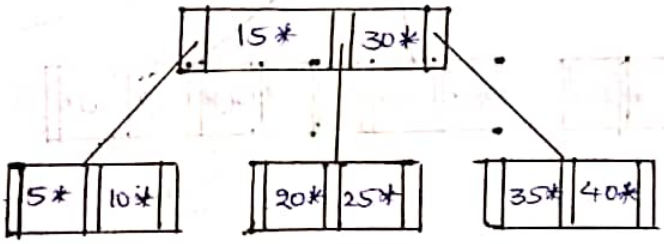
step-2 :-



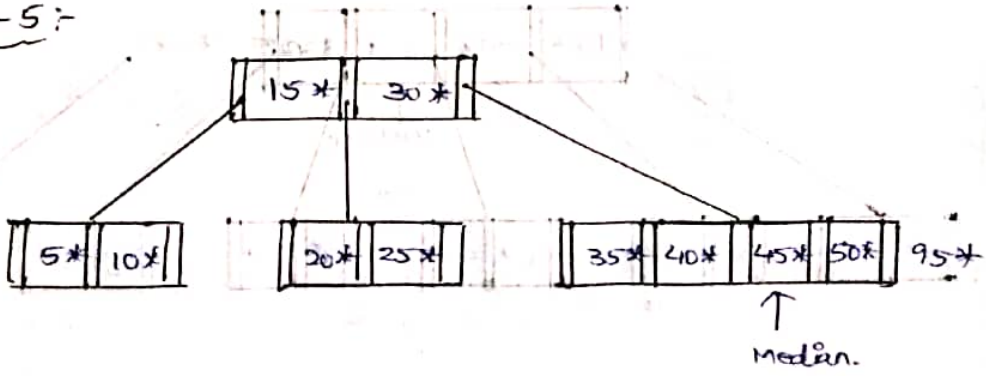
step-3 :-



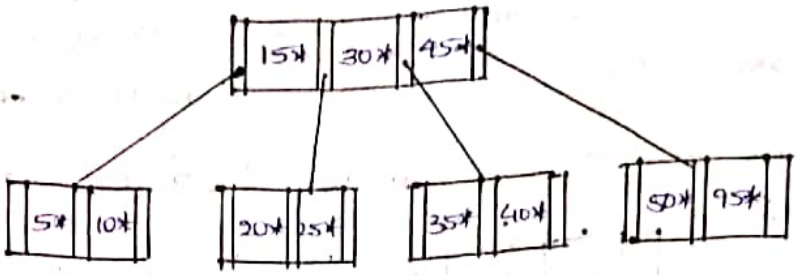
step-4 :-



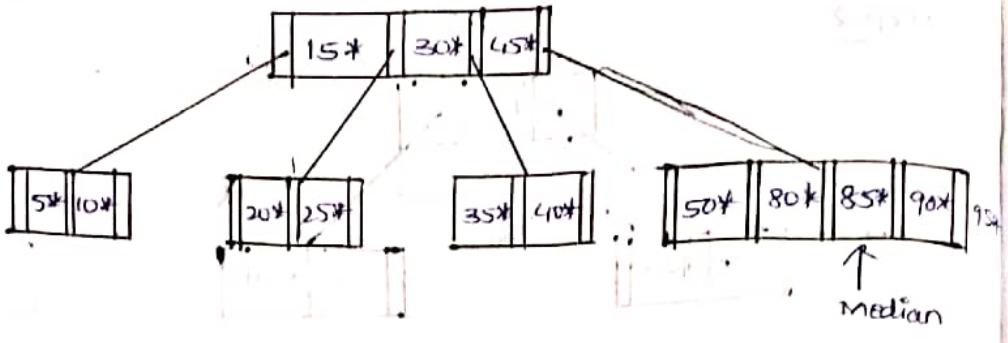
step-5 :-



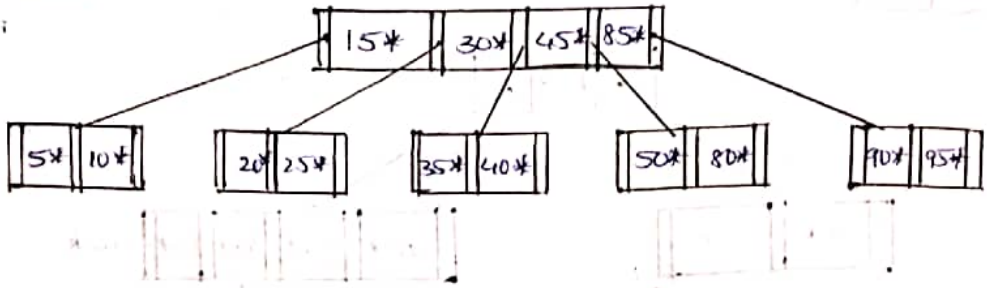
step-6 :-



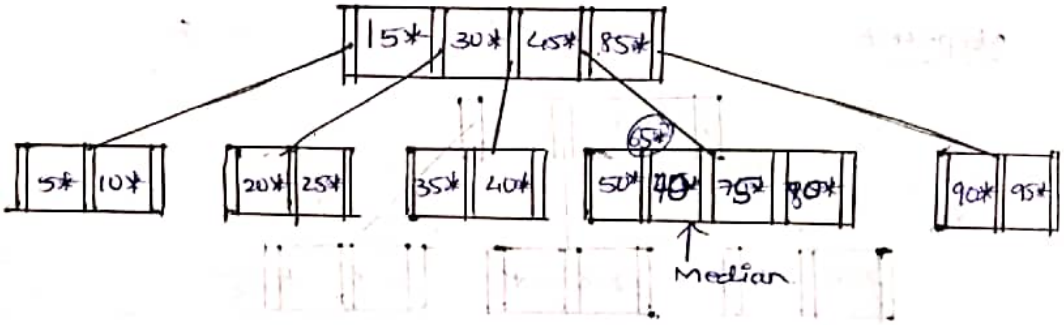
step-7 :-



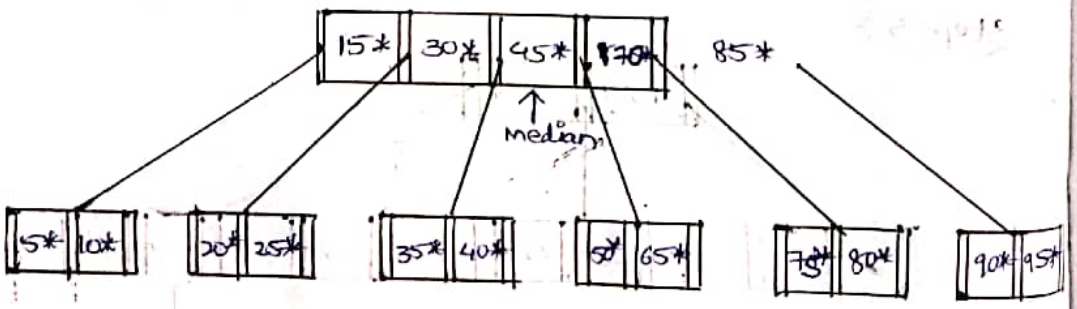
step-8 :-



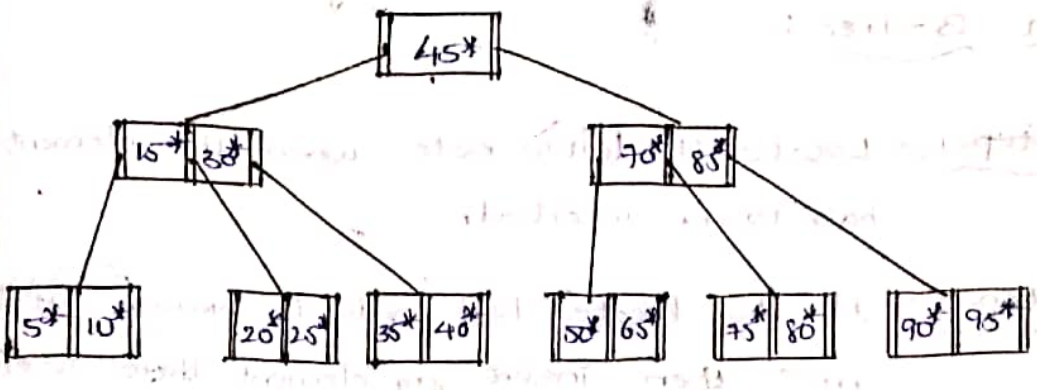
step-9 :-



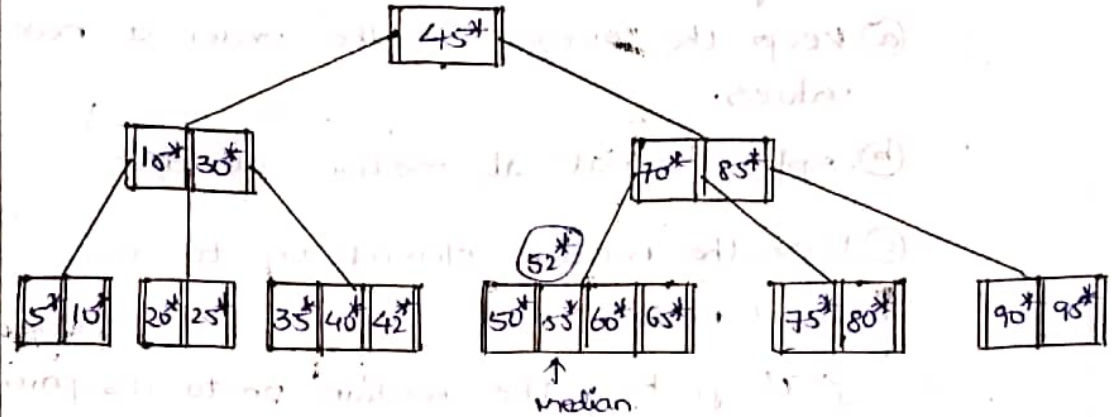
step-10 :-



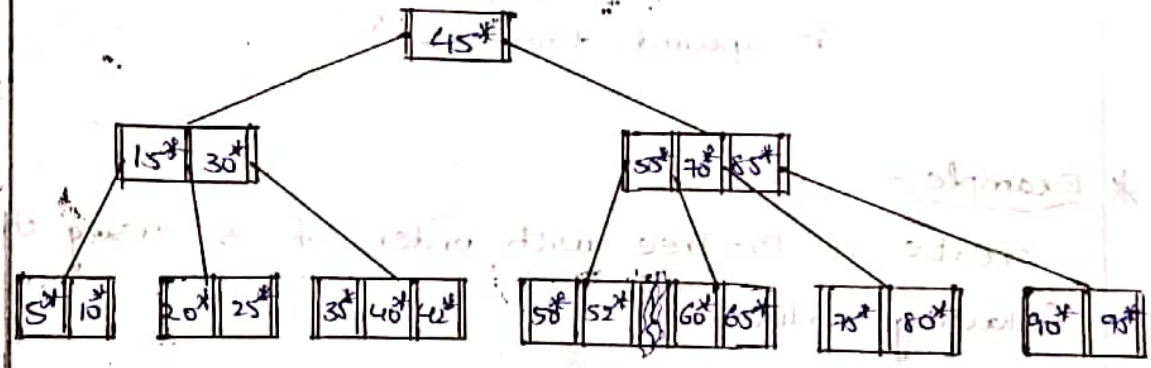
Step-11 :-



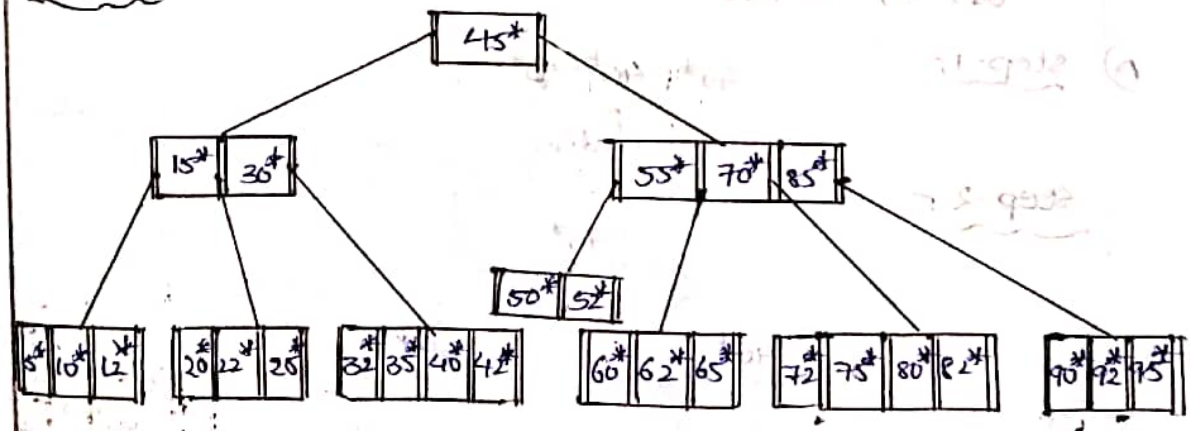
Step-12 :-



Step-13 :-



Step-14 :-



* Algorithm for creation / Inserting an element into a B-Tree :-

Step-1 :- Locate the leave node where the element has to be inserted.

Step-2 :- If the located leaf node is having sufficient space then insert an element there itself.

Step-3 :- If the located leaf node become full then following the below steps.

(a) keep the element in the order of node values.

(b) split the node at median element.

(c) Push the median element on to its parent node.

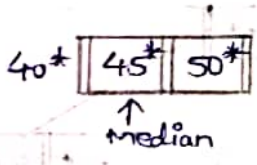
(i) If pushing the median on to its parent node, if the parent node already became full then propagate the above process in upward direction.

* Example :-

create a B-Tree with order of 3 using the following values.

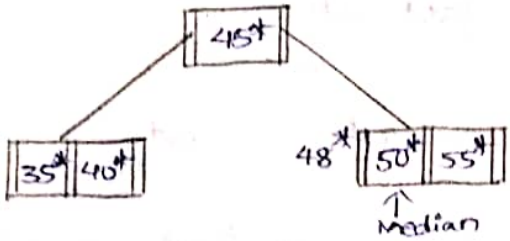
50, 45, 40, 35, 55, 48, 60, 65, 25, 30, 15, 10, 5, 68, 70, 75, 80

A) Step-1 :-

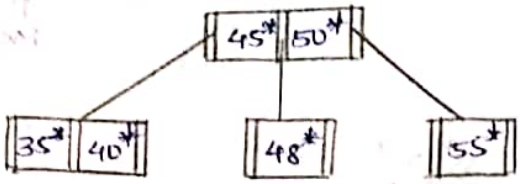


Step-2 :-

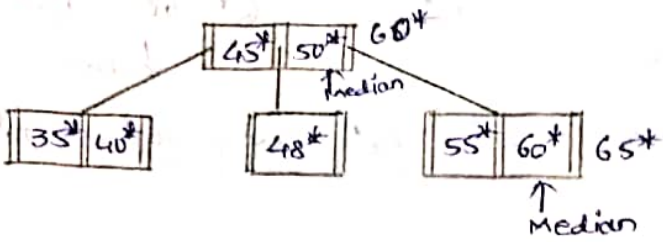
step-3 :-



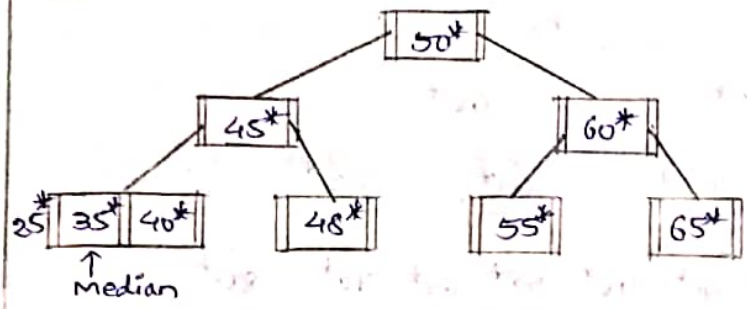
step-4 :-



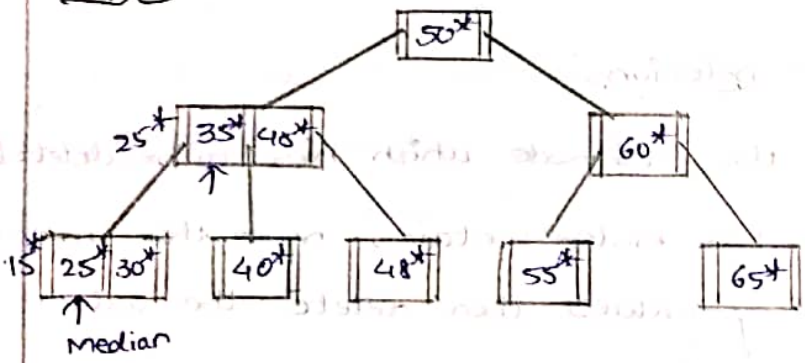
step-5 :-



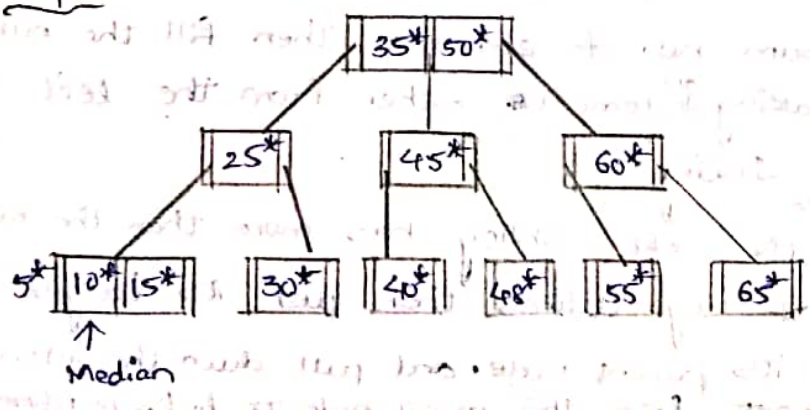
Step-6 :-



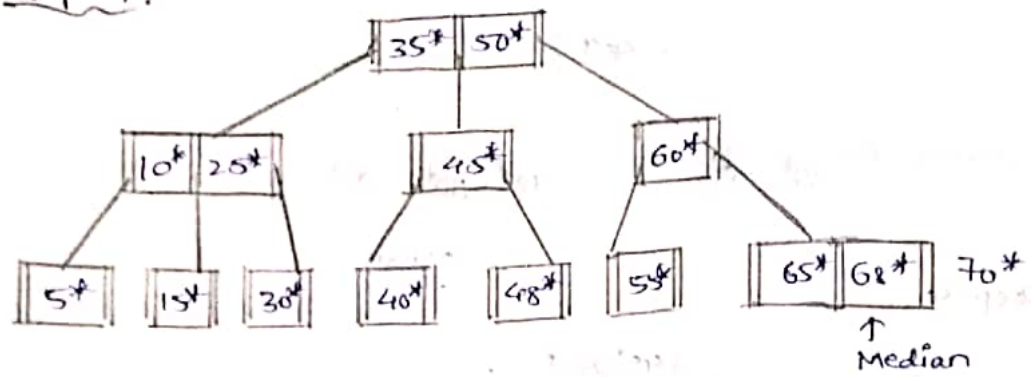
Step-7 :-



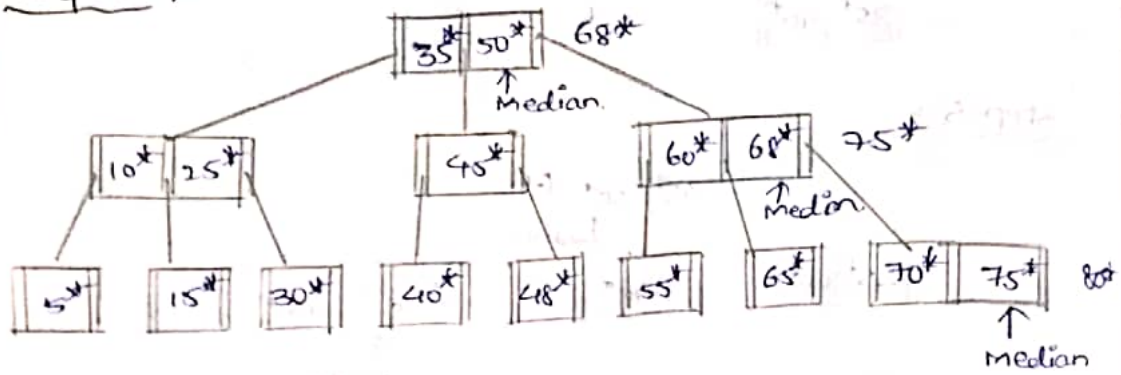
Step-8 :-



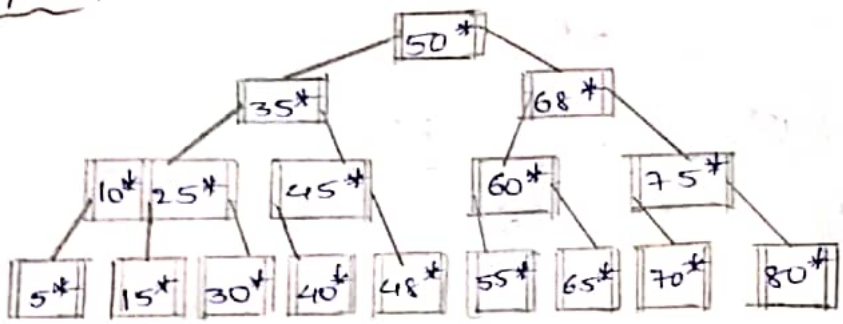
Step-9 :-



Step-10 :-



Step-11 :-



* Algorithm for Deletion :-

step-1 :- Locate the leaf node which has to be deleted.

step-2 :- If the Leaf Node contains more than minimum no. of key values then delete the value.

step-3 :- else if the leaf node doesnot contains even minimum no. of elements then fill the node by taking an element either from the left or right sibling.

(a) If the left sibling has more than the minimum no. of key values then push its Largest key into its parent node. and pull down the intervening element from the parent node to leaf node where key is deleted.

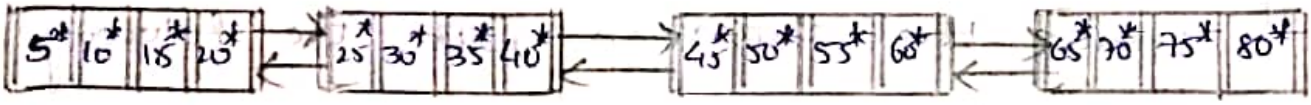
(b) else if the right sibling has more than minimum no. of key values then push its smallest key into its parent node & pull down the intervening element from the parent node to leaf node where the key is deleted.

Step-4 (c) else if both left & right siblings contains only the minimum no. of elements then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node.

If the pulling the intervening element from the parent node leaves it with less than the minimum no. of keys in the node then propagate the process in upward direction. Thereby Reducing the height of the B-Tree.

* B+ Tree :-

→ Let us assume the leaf nodes of B-Tree with order-5 as follows.



→ The above is called as B+ Tree.

Advantages :

→ It is used to retrieve the range of records easily.

Ex:- ① Employees details from ID 5000 to 10,000.

② Accessing a value from 20 to 60.

* Red-Black Tree :-

→ It is used to minimize the no. of Rotations in an AVL Tree.

Properties :-

(1) Any node in the tree should be coloured either with red or with black.

(2) Root node must be in Black colour.

(3) Red-Black Tree shouldnot contains Red-Red sequence in any path from root node to leaf node.

(4) The no. of black nodes from any path from root node to leaf node must be equal.

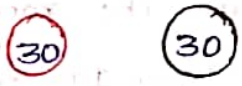
(5) It should follow the properties of B.S.T.

* creation of R.B.Tree :-

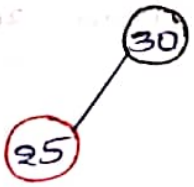
create the R-B Tree with the following values

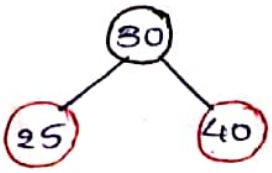
30, 25, 40, 28, 45, 50, 48, 55, 60, 42, 35, 20, 15, 10, 65

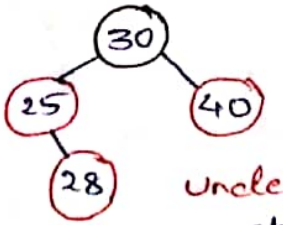
A).-In Insertion, always we are going to insert a value in red colour.

Step-1 :- 

But Root node is Black.

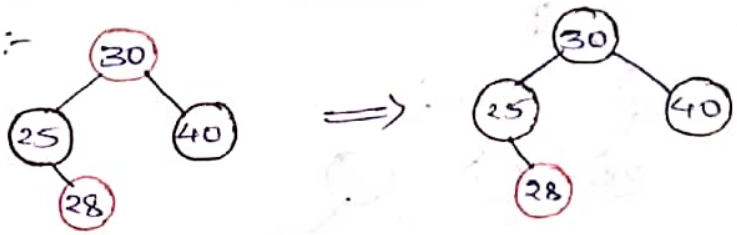
step-2 :- 

step-3 :- 

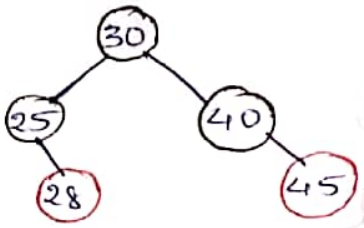
step-4 :- 

Uncle Node case - We need to change Parent node color.

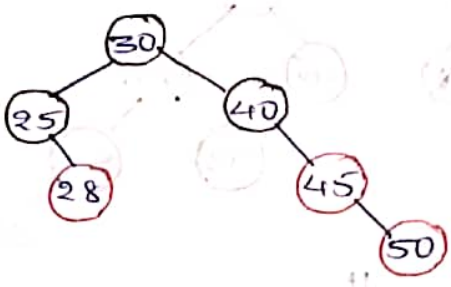
step-5 :-



step-6 :-

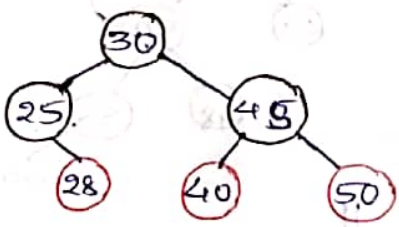


step-7 :-

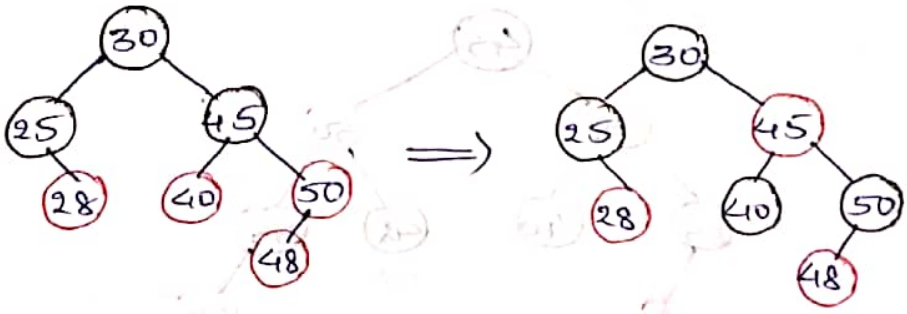


→ Red-Red sequence occured & uncle-Node is also absent.

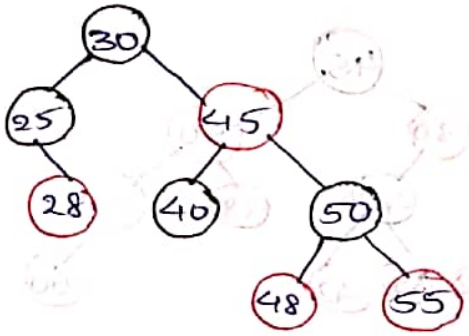
→ so we need to apply rotation.



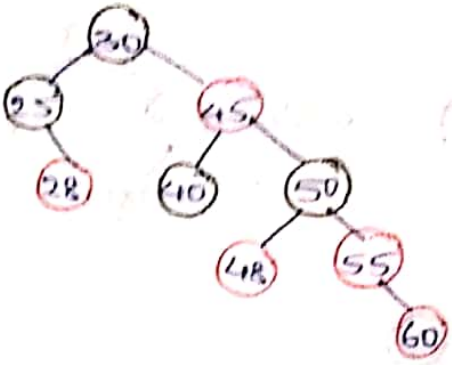
step-8 :-

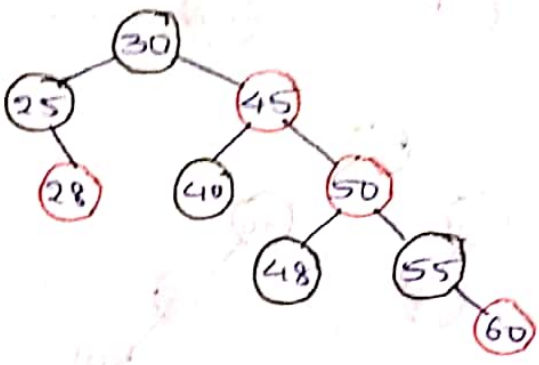


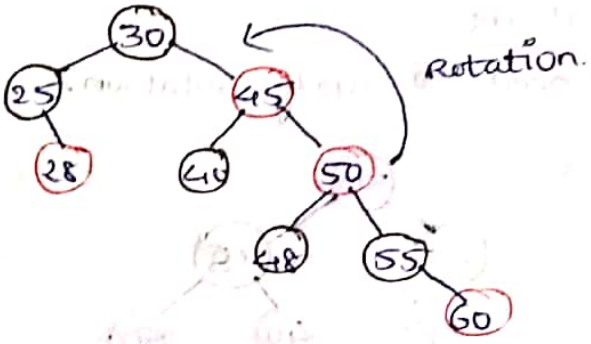
step-9 :-

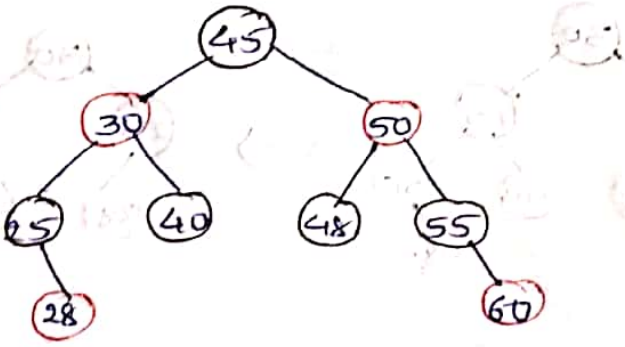


Step-10 :-

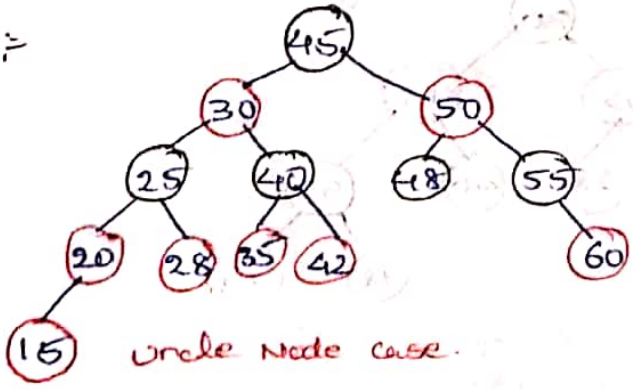






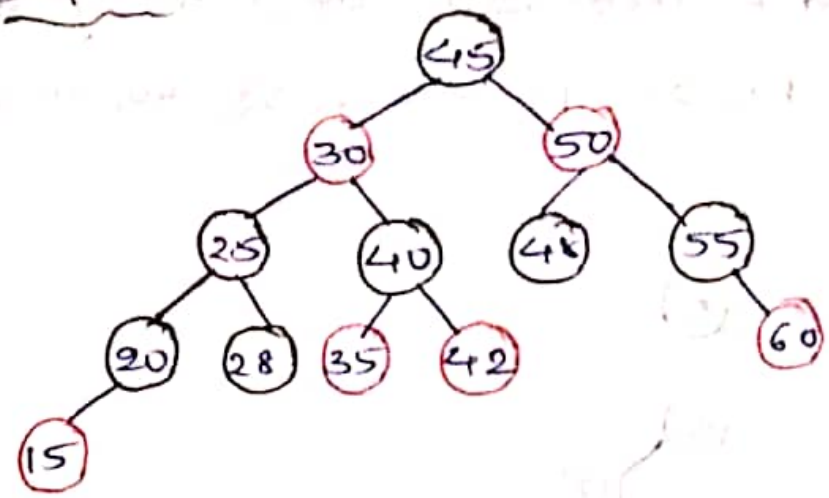


Step-11 :-

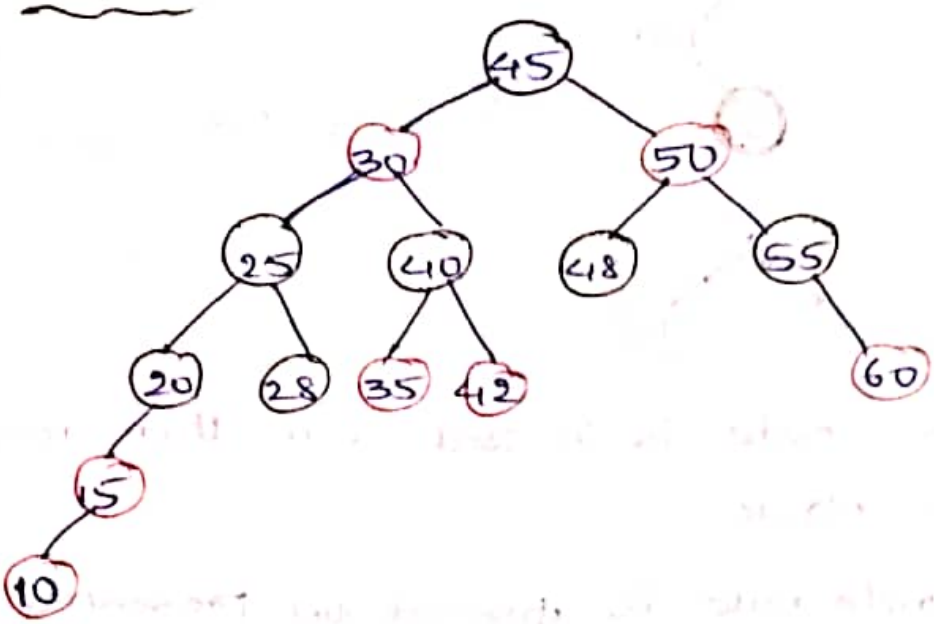


Uncle Node case.

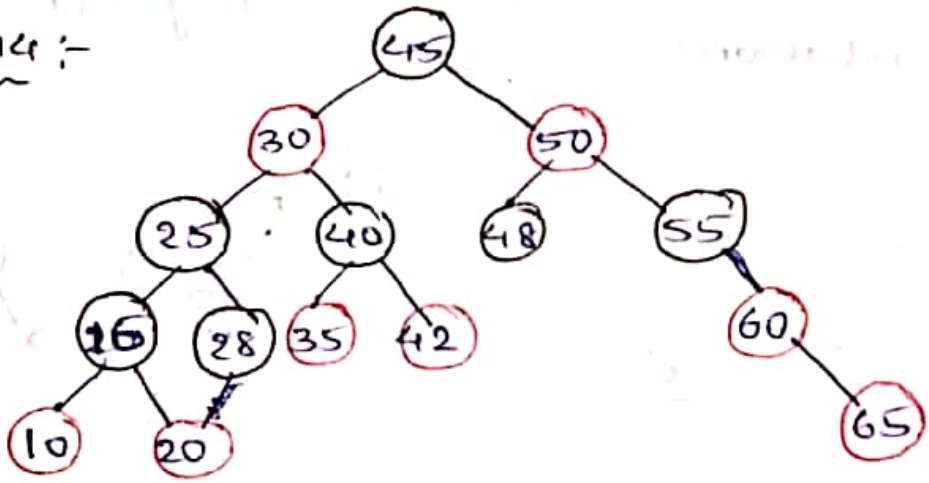
step-12 :-



step-13 :-



step-14 :-



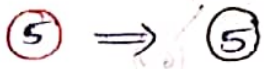
Rotation.

* Create a Red-Black Tree with the following values

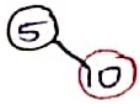
5, 10, 8, 9, 12, 1, 15, 25, 18, 30, 35, 32, 40, 50, 45, 46, 65, 60.

A)

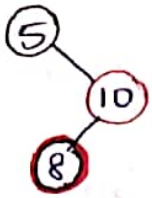
step-1 :-



step-2 :-

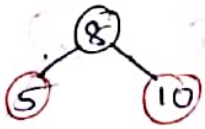


step-3 :-



Red-Red sequence

step-4 :-

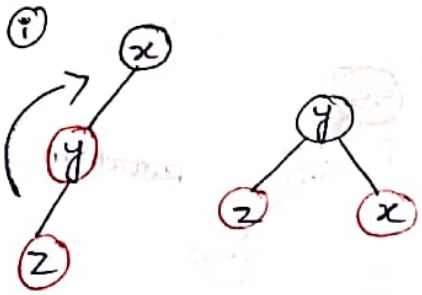


cases :-

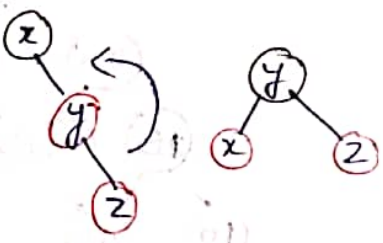
case-1 :- If root node is in Red color then recolor it as black.

case-3 :- If Uncle node is absented (or) presented with black color, then apply appropriate rotation.

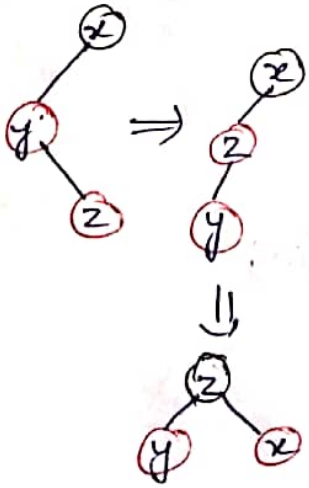
(i)



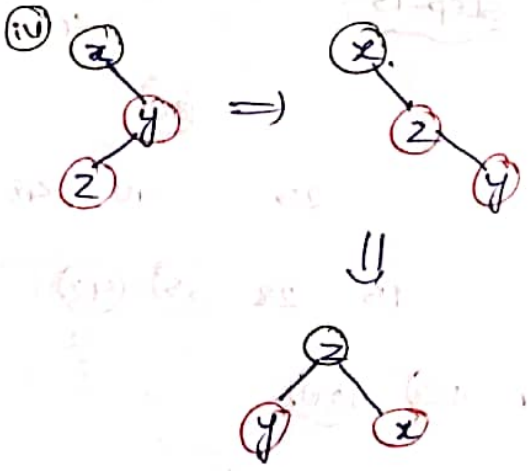
(ii)



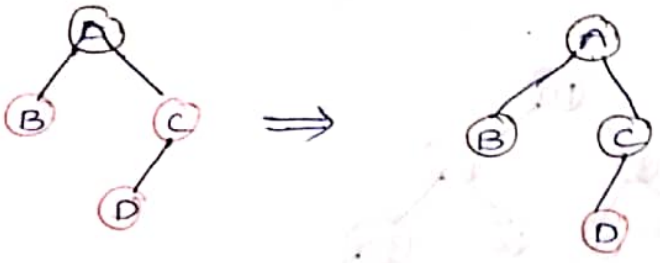
(iii)



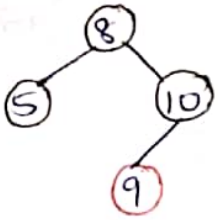
(iv)



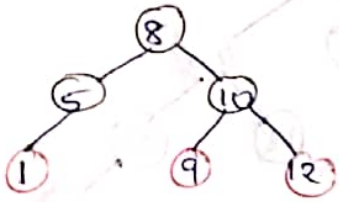
Case-2:- If the uncle node is presented with red color, then recolor its parent node, recolor parent sibling node [uncle node] & recolor their parent node.



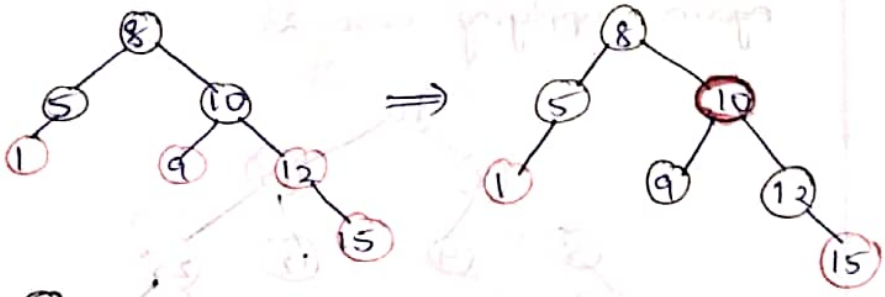
step-5:-



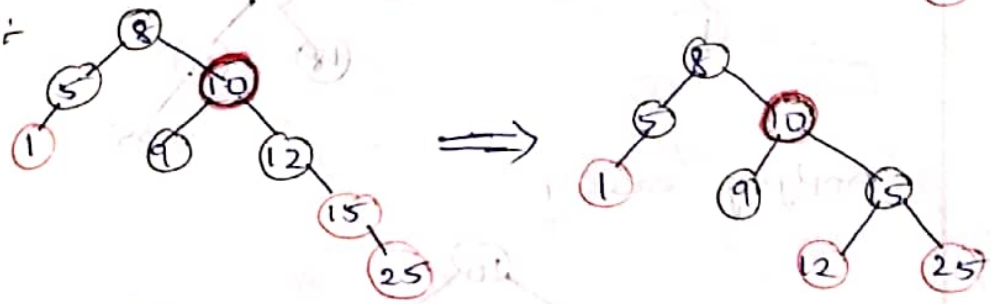
step-6:-



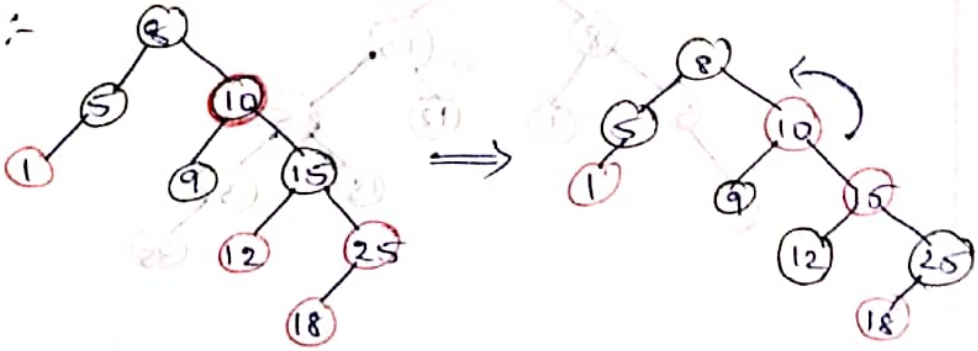
step-7:-

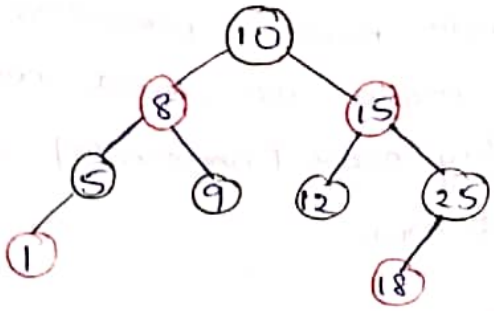


step-8:-

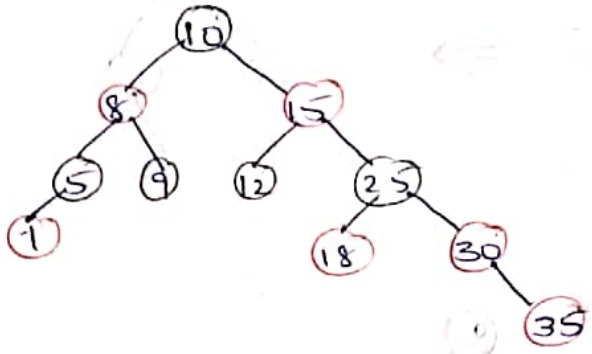


step-9:-

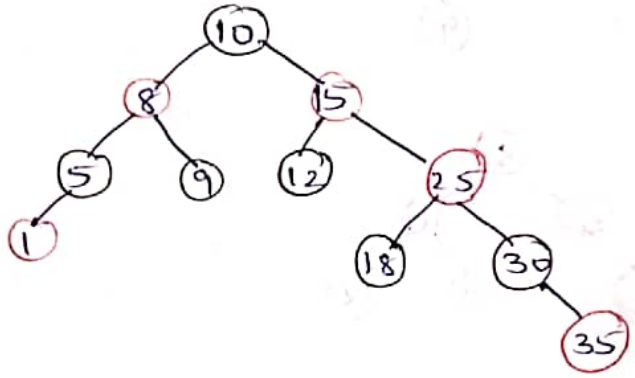




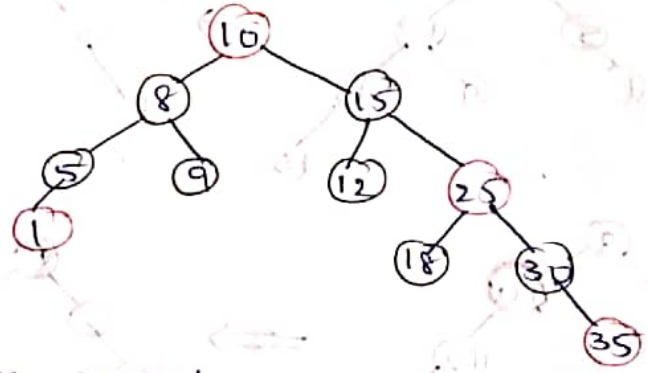
step-10 :-



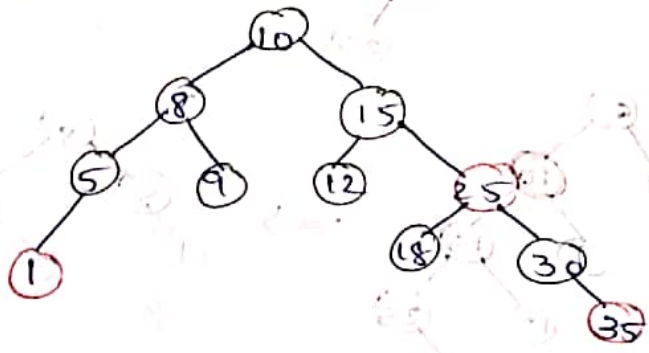
Applying case-2.



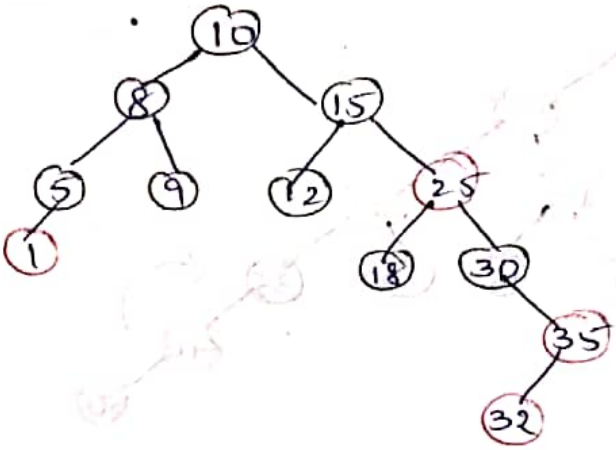
Again applying case-2.



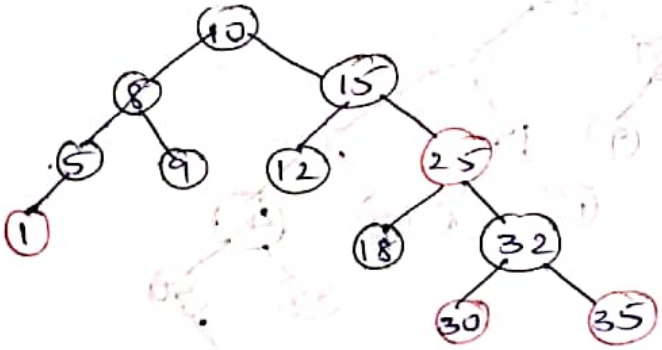
Applying case-1



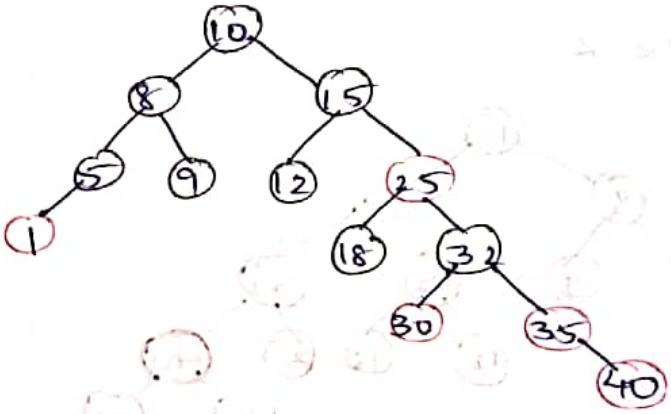
step-11 :-



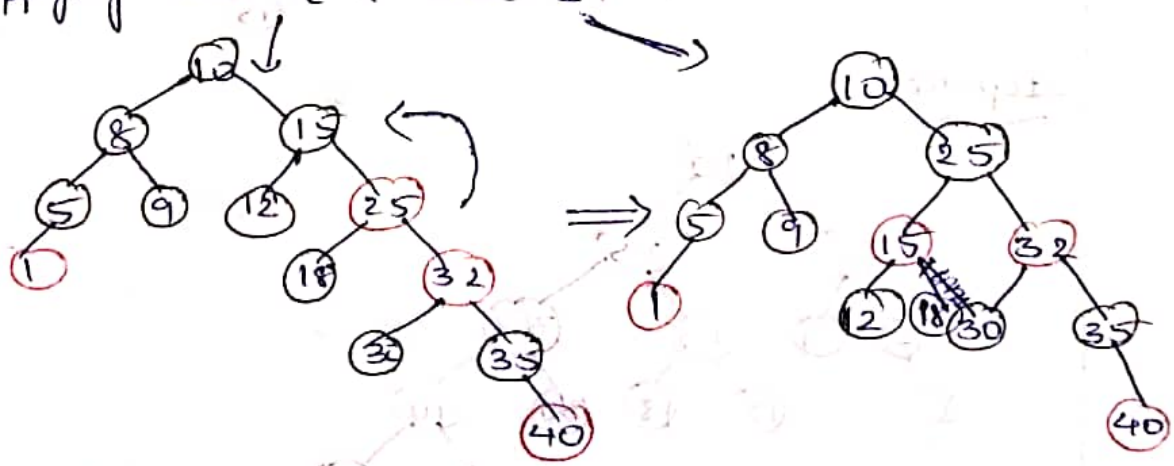
Applying case-3, [2 - Rotations].



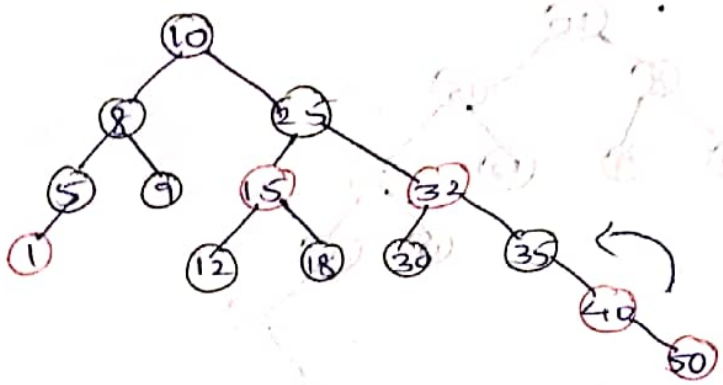
step-12 :-



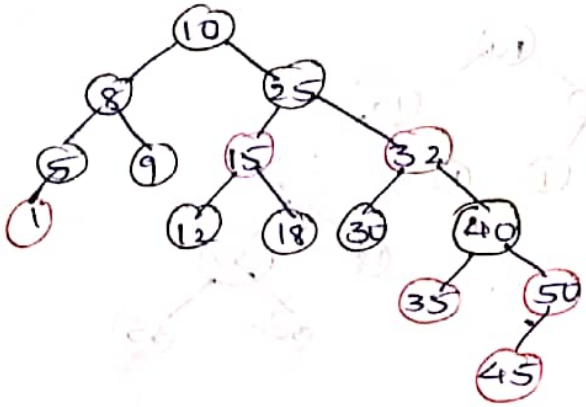
Applying case-2 & case-3



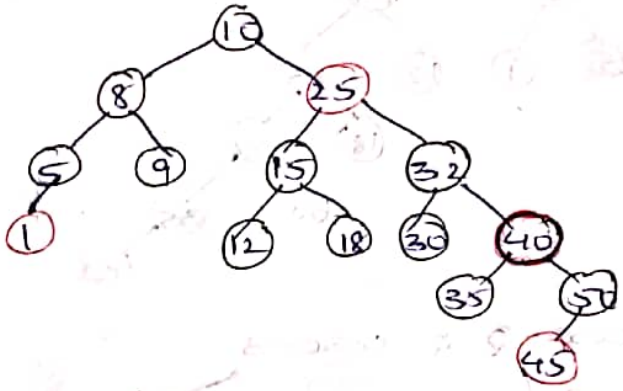
step-13:



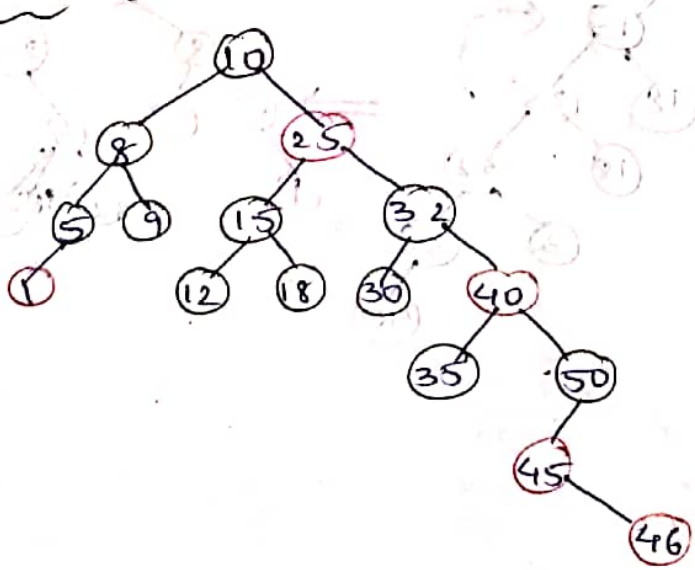
Applying case-3:



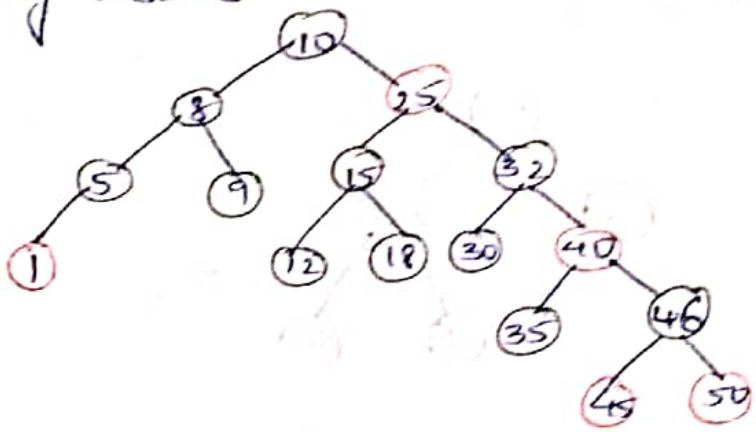
Applying case-2.



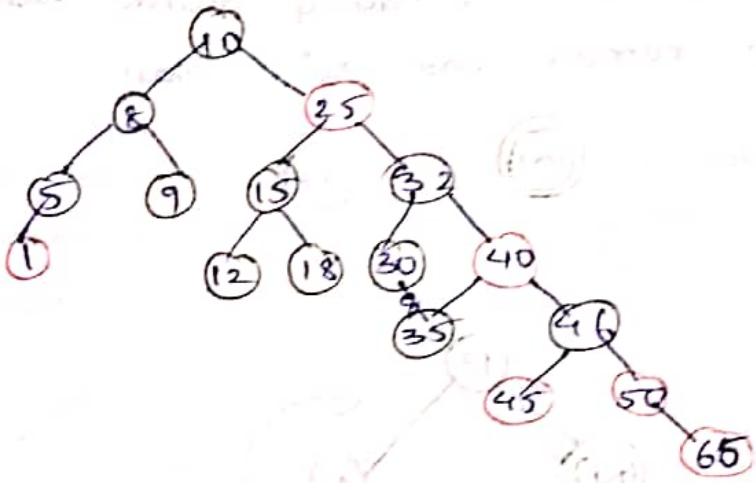
step-14:



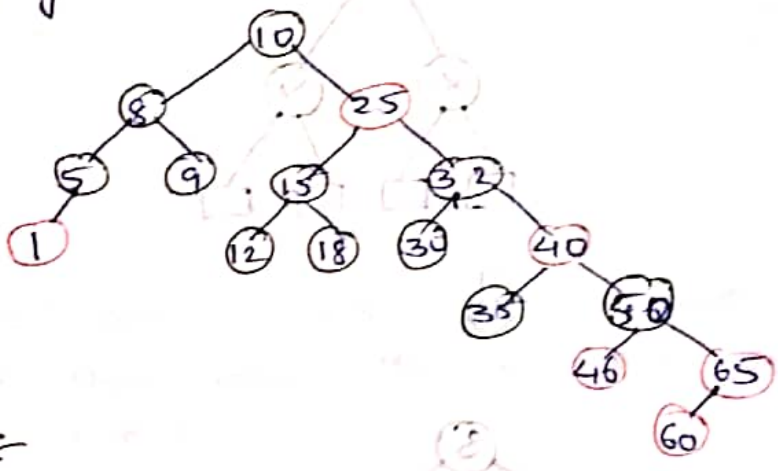
Applying case-3 :



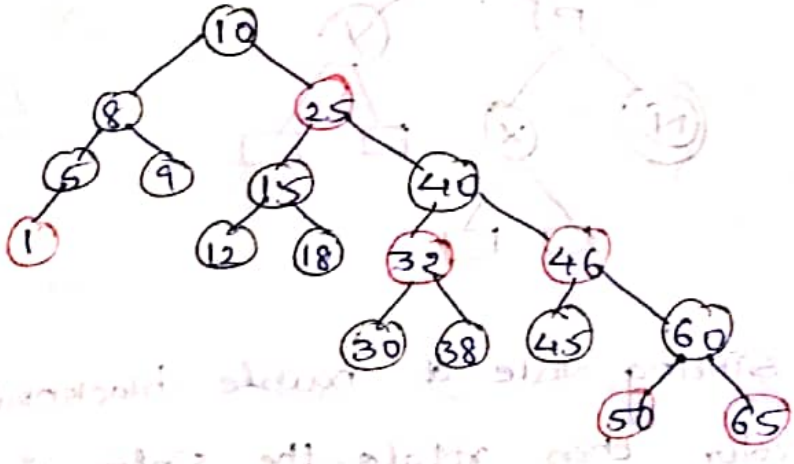
step-15 :-



Applying case-3 :

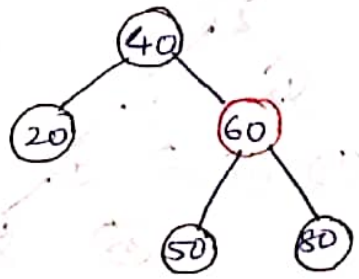


step-16 :-



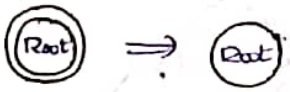
* Deletion operation :-

Ex:-

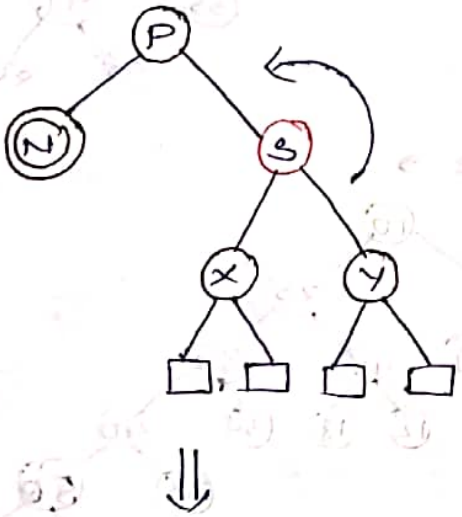


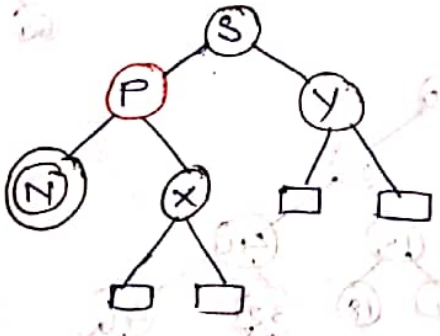
case-1 :-

→ If root node is having double black colour then remove one Black colour.



case-2 :-

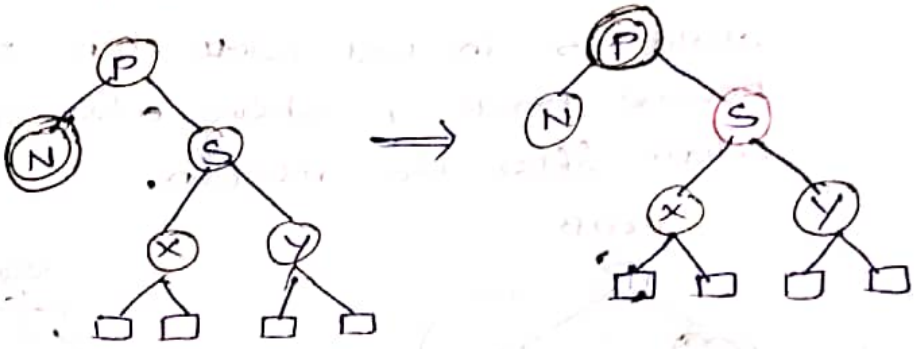




→ If sibling Node of Double blacknode is in Red color then rotate the sibling in L.L direction and append Red color to parent Node.

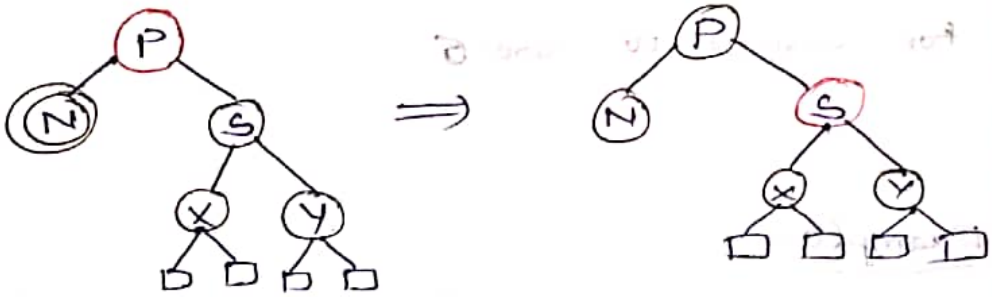
case-3 :-

→ If all are in black color, then make parent node as double black node & append red color to sibling node.



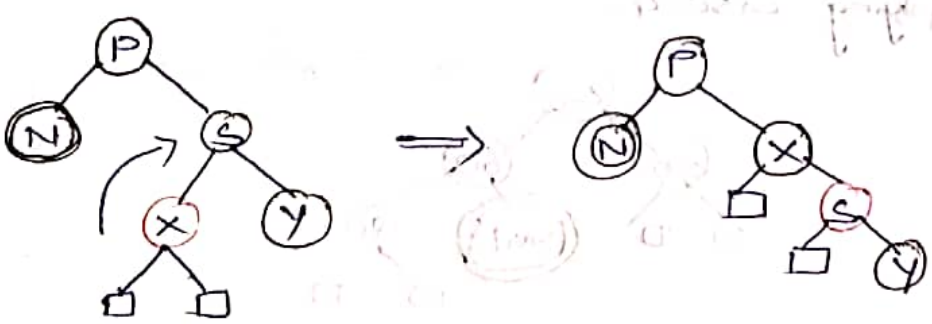
case-4 :-

→ If all are in black color except parent node then make parent color as black [by moving one black color from N] and sibling as red.



case-5 :-

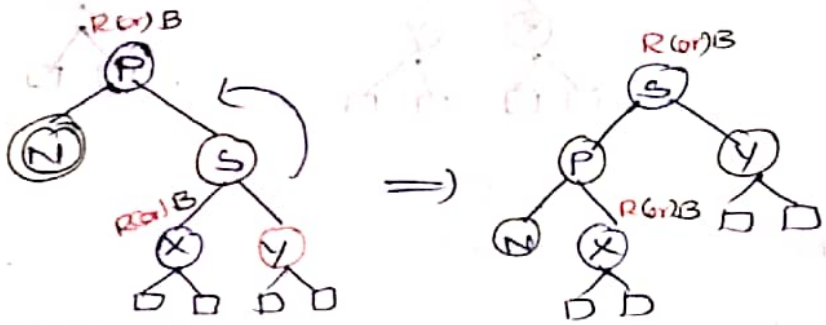
→ If all are in black colour except sibling's left child then rotate the x and s nodes and give red color to s.



→ This is not Terminating case since Double Black Node is still present.

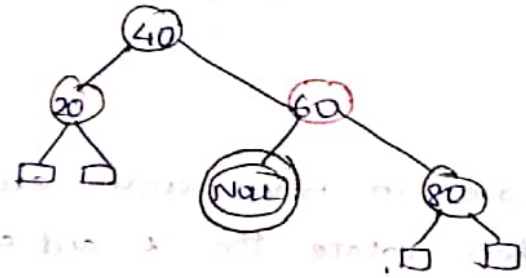
Case-6 :-

→ If parent and sibling's Left childs are in either red or Black colour and sibling's right child is in Red colour then parent node colour become black & sibling colour is taken parent's colour after the rotation.

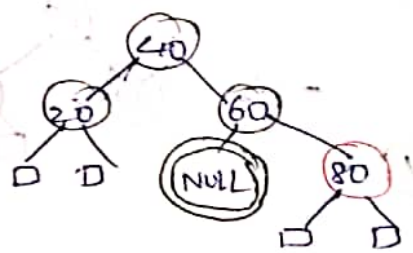


Case-7 to 11 :- These are mirror cases [symmetric] for case-2 to case-6.

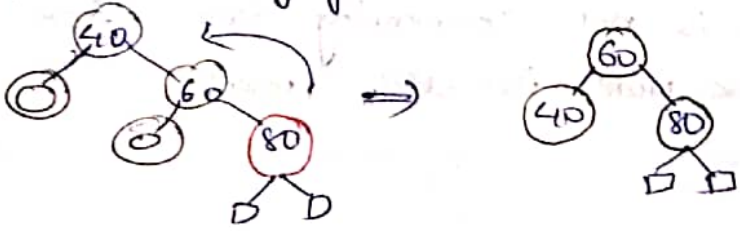
* Example :-



A) Applying case-4 :-

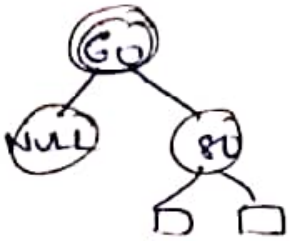


Deleting 20 & Applying case-6.

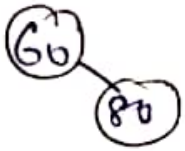


Deleting 40

Applying case-3.



Applying case-1



Deleting 80.

(or)

Deleting 60

* Applications of Trees :

(1) Used to sorti the elements.

(2) storing different/unique records [No duplicates].

(3) To store the data in Hierarchial structure.

(4) Less Accessing Time.