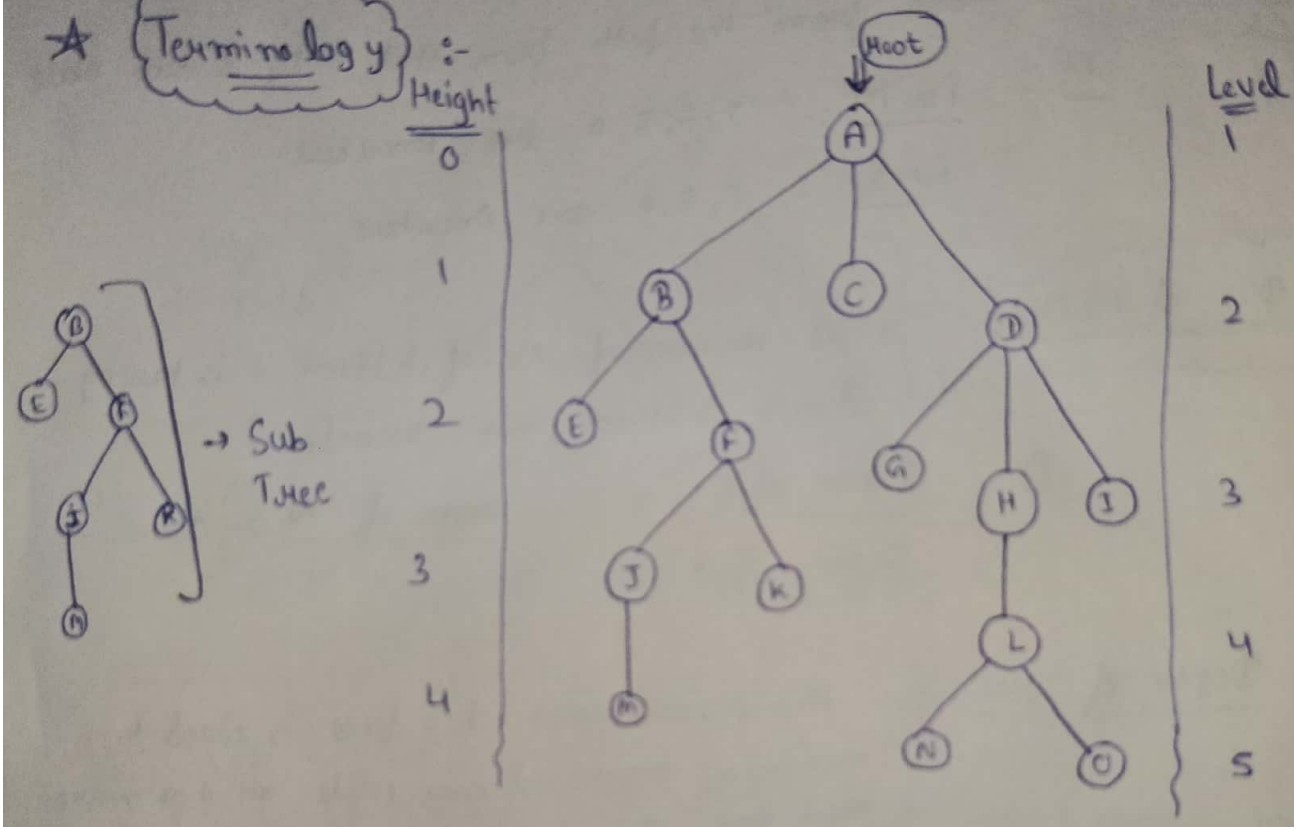
TREES

:- Tree is a Collection of Nodes & edges, if there are 'n' nodes, then there will be 'n-1' edges.

★ **Terminology** :-



1. **Root** :- The very first node on the top of the tree is called Root.
2. **Parent** :- A node is a parent to its very next descendant.

   Eg :- E, F are children of B.
   G, HI are children of D

3. **Siblings** :- Siblings are children of same Parent.

   Eg :- G, H, I are Siblings
   J, K are Siblings
   E, F are Siblings.

Binary Tree

④ **Descendents** :- Descendants are those set of Nodes which can be from a particular Node or under that Node.

Eg :- For B :- E, F, J, k, M are Descendants

⑤ **Ancestors** :- All nodes from the path from that Node to root

Eg :- For M :- J, F, B, A are Ancestors

For k :- F, B, A are Ancestors.

⑥ **Degree of Node** :- It is measure of No. of children it is having (i.e. direct children, not Descendents)

Eg :- Degree of L :- 2 Degree of N :- 0

Degree of J :- 3

Note :- Degree of Tree :- Minimum Degree of a tree is equal to maximum Degree of any Node or it is more than that also.

⑦ **Internal / External Nodes** :- Nodes with degree '0' are known as External nodes / Leaf Nodes. and Nodes whose degree is greater than '0' are known as internal nodes / Non-Leaf Nodes

⑧ **Levels** :- For Level, we just Count No. of nodes Connected along Single path.

⑨ **Height** :- We Count edges to determine Height.

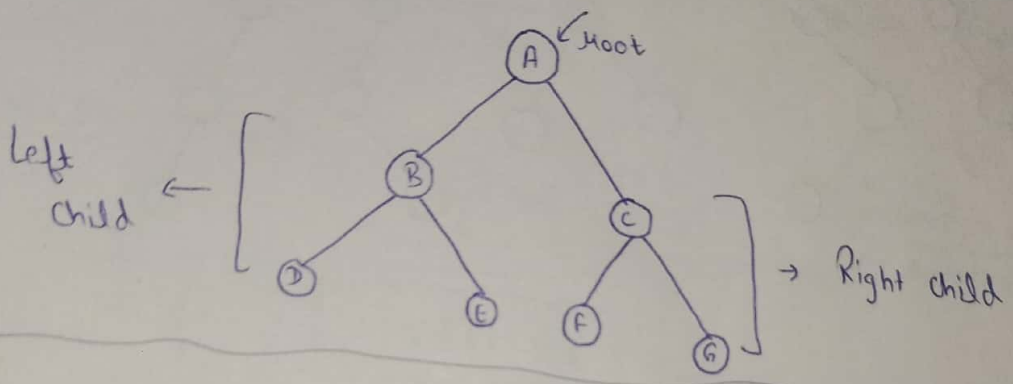
Note :- Levels and Height are used for Analysis.

Binary Tree :-

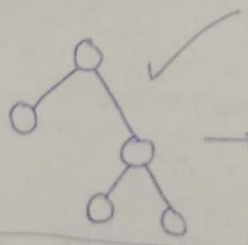
A tree having Degree '2'. it means that every node must have 2 children, can be less than 2 children but not more than 2 children.

i.e. $\deg(T) = 2$

children = {0, 1, 2}

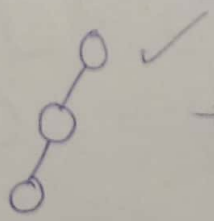


Eg :- (1)



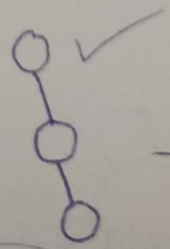
→ It is also a binary tree

Eg :- (2)



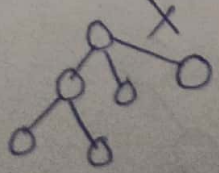
→ It is also a binary tree (name :- Left Skewed Tree)

Eg :- (3)



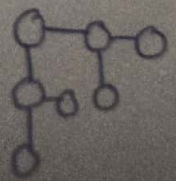
→ It is also a binary tree (name :- Right Skewed Tree)

Eg :- (4)



→ It is not a binary tree.

Eg :- (5)



→ It is also a binary tree.

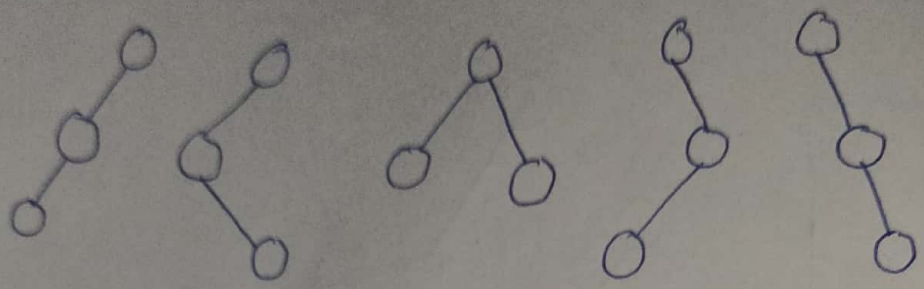
✡ Number of binary Trees :- How many types of bi... Can be formed using gi... Nodes

For n=...

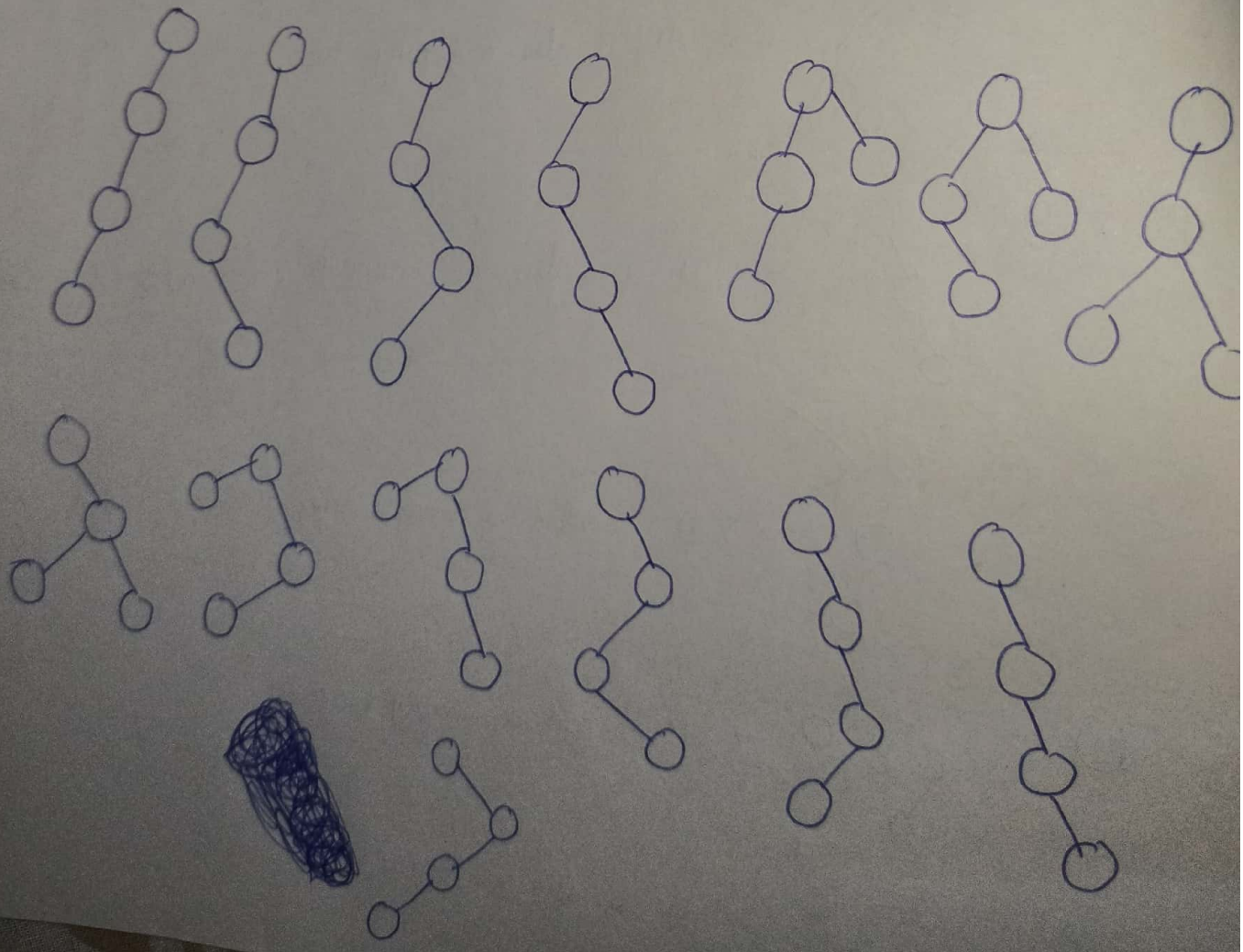
For ...

No...

if n = 3 :- 000



if n = 4 :- 0000



For $n=3$, Total no. of Trees $= T(3) = 5$

For $n=4$, Total no. of Trees $= T(4) = 14$

Now, For 'n' no. of trees, we have a formula

$$T(n) = \frac{2n C_n}{n+1} \longrightarrow \text{Catalan Number formula}$$

For $n=5$, $T(5) = \frac{10 C_5}{6} = \frac{\frac{10!}{5!\,5!}}{6} = \frac{10 \times 9 \times 8 \times 7 \times 6}{6 \times 5 \times 4 \times 3 \times 2 \times 1} = 7 \times 3 \times 2 = 42$

$\therefore T(5) = 42$

$\Rightarrow$ Now Maximum Height of a Tree :- We know Height starts from '0' onwards.

If $n=3$:- Maximum Possible Height of a tree $= 2$

No. of trees with maximum Height $\Rightarrow 4 = 2^2$

If $n=4$:- No. of trees with maximum Height $\Rightarrow 8 = 2^3$

For $n=5$:- No. of trees with maximum Height $\Rightarrow 16 = 2^4$

For 'n' :- No. of trees with maximum Height $\Rightarrow 2^{n-1}$

⇒ One another formula for Catalan Number :-

n	0	1	2	3	4	5	6
$T(n) = \frac{2nC_n}{n+1}$	1	1	2	5	14	42	?

Now, $T(6) = (1 \times 42) + (1 \times 14) + (2 \times 5) + (5 \times 2) + (14 \times 1)$

$= 132$

Now, Verify :- $T(6) = \frac{12C_6}{7} = \frac{12 \times 11 \times 10 \times 9 \times 8 \times 7}{7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}$

$= 11 \times 2 \times 3 \times 2$

$= 11 \times 12$

$= 132$ ✓

Now, Formula Preparation :-

$T(6) = [T(0) * T(5)] + [T(1) * T(4)] + [T(2) * T(3)] + [T(3) * T(2)] + [T(4) * T(1)] + [T(5) * T(0)]$

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$ ✓

Hence, $T(n) = \frac{2nC_n}{n+1}$ → Combination Catalan Formula

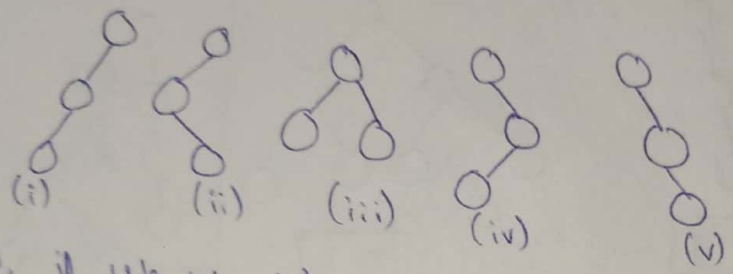
$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-1)$ → Recursive Catalan Formula

Labelled Nodes :- Labeling the Nodes.

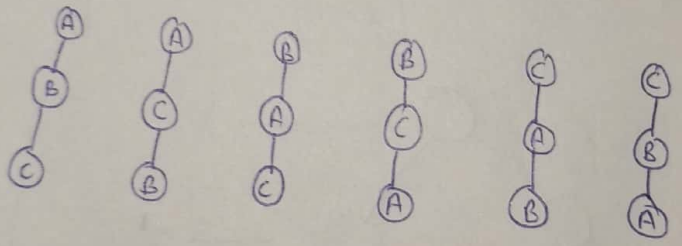
Previously, we done with the unlabelled Nodes, Now will done with labelled Nodes.

Now, For labelled Nodes :-

if n = 3 :- (A) (B) (C)



Now, if we take (i), than



it means, there will be 3! trees of one representation for n=3, Hence, it will be (3! x 5)

So, the formula for Number of trees for Labelled Nodes will be :-

$$T(n) = \frac{2nC_n}{n+1} * n!$$

Shapes

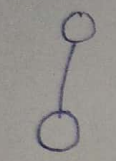
filling Permutations

Height Vs. Nodes :- If the Height of a binary tree is 'h', then how much maximum no. of and minimum no. of Nodes.

Height

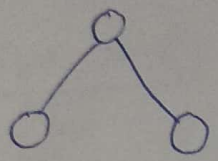
$h = 1$

min



$n = 2$

max.

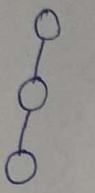


$n = 3$

Height

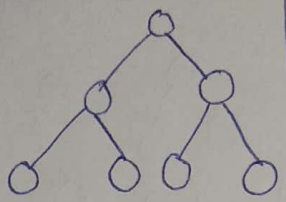
$h = 2$

min



$n = 3$

max



$n = 7$

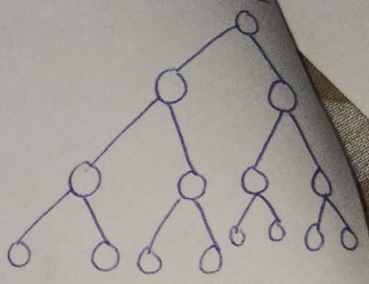
Height

$h = 3$

min

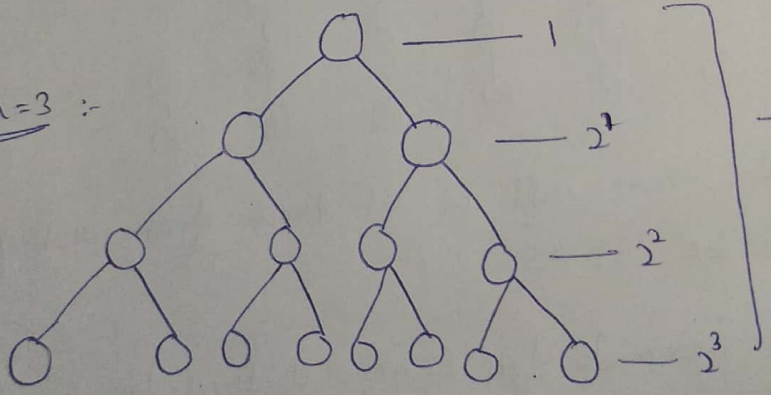
$n = 4$

max



$n = 15$

Now for h = 3 :-



→ Nodes present at different Levels

Total no. of Nodes : $1 + 2 + 2^2 + 2^3 = 15$ → it represents G.P Series

$1 + 2 + 2^2 + 2^3 + \cdots + 2^h \Rightarrow a = 1 ; r = 2$

$a + ar + ar^2 + ar^3 + \cdots + ar^k = \frac{a(r^{k+1} - 1)}{r - 1}$

$\therefore$ Sum $\Rightarrow \frac{1(2^{h+1} - 1)}{2 - 1}$

$\therefore$ **Sum = $2^{h+1} - 1$**

$\therefore$ **Maximum No. of Nodes = $2^{h+1} - 1$**

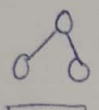
Minimum No. of Nodes = $h + 1$

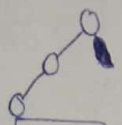
3.6

Now, if Nodes are Given, then what will be maximum and minimum Height"

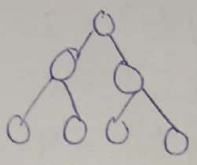
Here we will use the formula of Minimum Nodes, to find maximum Height and formula of maximum Nodes, to find minimum Height.

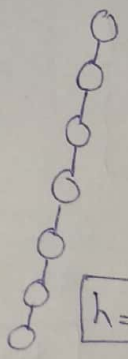
Nodes
$n = 3$

min

$h = 1$

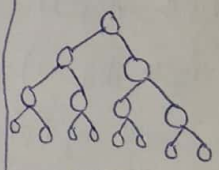
max

$h = 2$

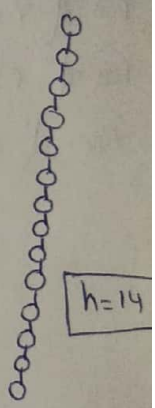
Nodes
$n = 7$

min

$h = 2$

max

$h = 6$

Nodes
$n = 15$

min

$h = 3$

max

$h = 14$

Now, for minimum Height :- We know max. no. of Nodes $= 2^{h+1} - 1$

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

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

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

$h+1 = \log_2(n+1)$

$\boxed{h = \log_2(n+1) - 1}$

For maximum Height :-
We know, min no. of Nodes
$n = h+1$
$\therefore \boxed{h = n-1}$

Eg :- $n = 15$

min :- $h = \log_2(15+1) - 1$
$h = \log_2(16) - 1$
$h = 4 - 1$
$\boxed{h = 3}$

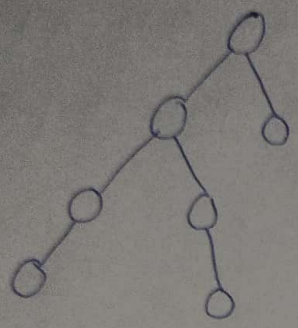
max :- $h = n-1 \Rightarrow \boxed{h = 14}$

Note :- ① No. of Nodes in a binary tree :- $\boxed{h+1 \leq n \leq 2^{h+1} - 1}$

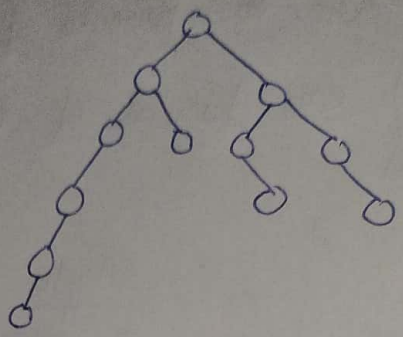
② Height of a binary tree :- $\log_2(n+1) \leq h \leq n+1$

$\therefore$ Space Complexity of binary tree ranged from $\boxed{O(\log n) \text{ to } O(n)}$

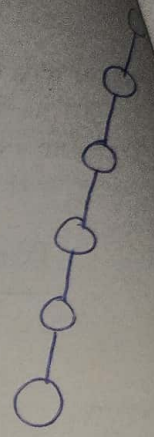
★ Relation b/w Internal (Leaf) and External (Non-Leaf) Nodes



No. of nodes with deg(2) = 2
No. of Nodes with deg (1) = 2
No. of Nodes with deg (0) = 3



deg (2) = 3
deg (1) = 5
deg (0) = 4



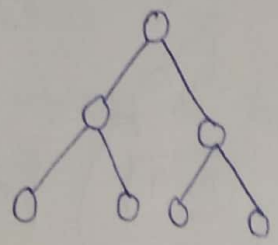
deg (2) = 1
deg (1) = 4
deg (0) = 2

∴ We can see here, that

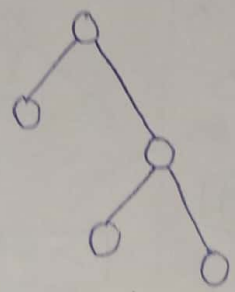
$$\deg(2) = \deg(0) - 1$$

i.e. No. of nodes with deg (2) = No. of Nodes with deg (0) - 1

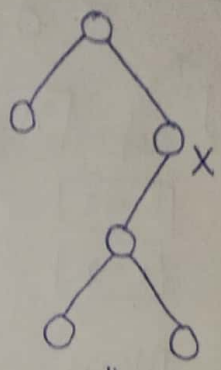
Strict Binary Tree :- We know that a generally binary tree, a node can have a degree 0, 1, 2 but a strict binary tree have degree only 0, 2, it means a node can have exactly '0' children or exactly '2' children, doesn't have '1' children, so strict means it is strictly binary.



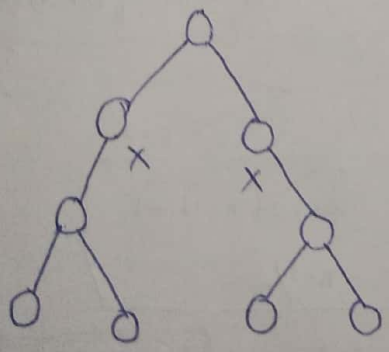
Strict binary tree.



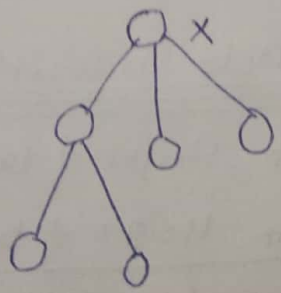
Strict binary tree.



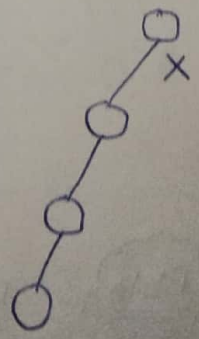
Not a strict binary tree.



Not a strict Binary tree.

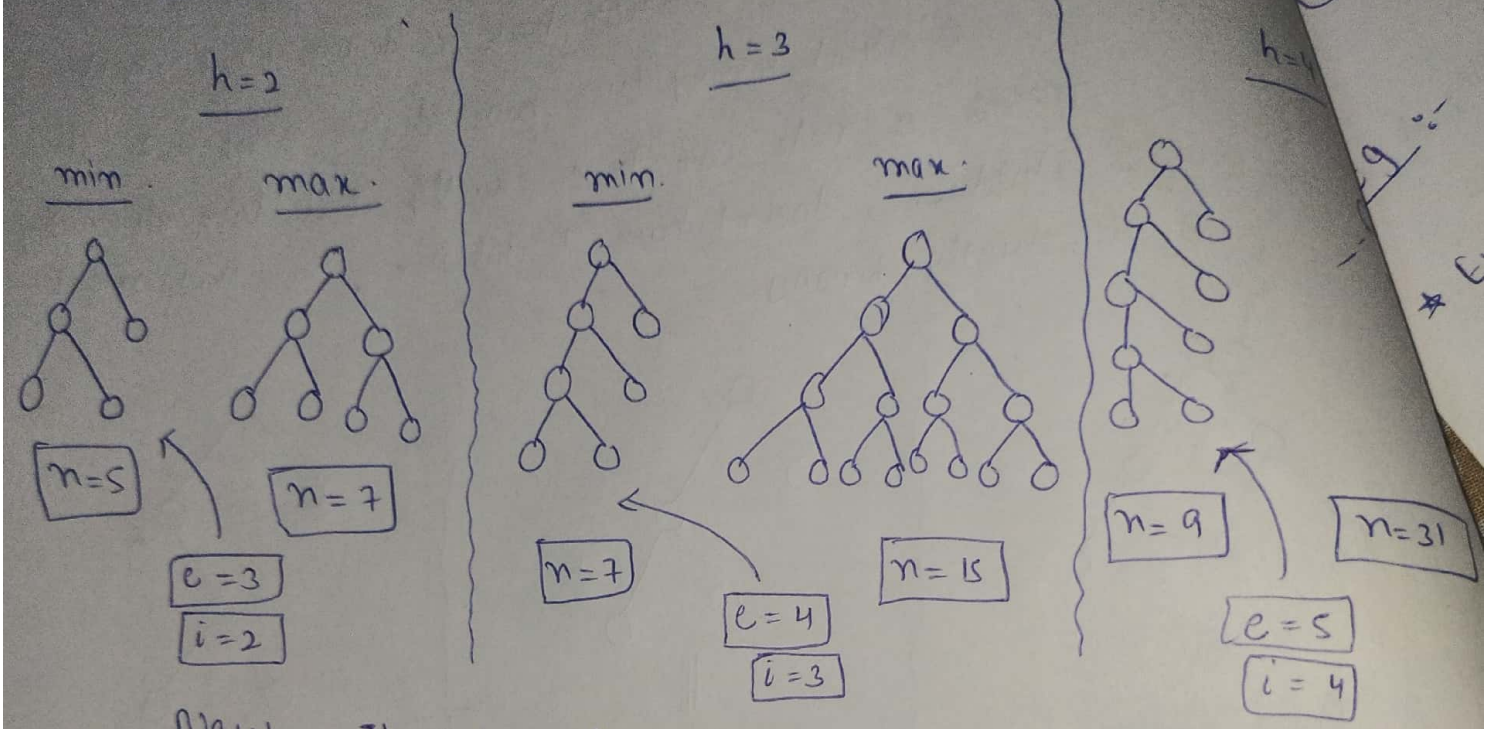


Not a strict Binary tree.



Not a strict Binary tree.

★ Height vs. Nodes :-



Now, **If Height is Given :-**

Minimum no. of Nodes = $2h + 1$

Maximum no. of Nodes = $2^{h+1} - 1$ (as Same of binary tree)

If 'n' Nodes is Given :-

Minimum Height of tree, $h = \log_2(n+1) - 1$

Maximum Height of tree, $h = n - 1/2$

Note :- Height Ranges from $O(\log n)$ to $O(n)$ i.e. $\log_2(n+1) - 1 \leq h < \frac{n}{2}$

★ Relation b/w Internal and External Nodes :-

$e = i + 1$

Where, e = no. of external nodes

i = no. of inter...

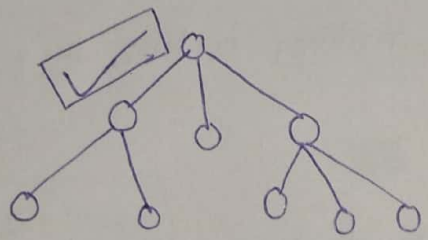
m-ary Trees :- Here 'm' is the degree of the tree, Degree of 'm' means, every node have atmost 'm' children not more than 'm' children.

Eg :-

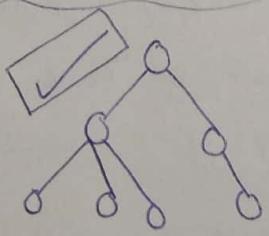
3-ary tree

* Every node can have degree - 3, 2, 1, 0.
* Capacity of these nodes is max. 3.

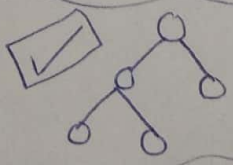
(i)



(ii)



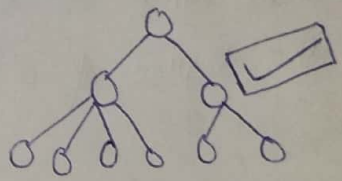
(iii)



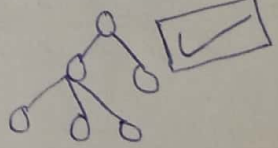
4-ary tree

* Every node can have degree - 4, 3, 2, 1, 0.
* Capacity of these nodes is max. 4.

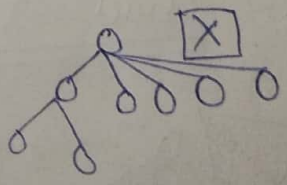
(i)



(ii)

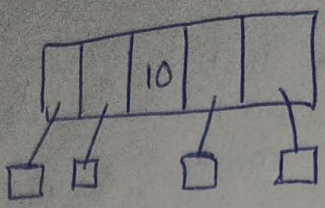


(iii)



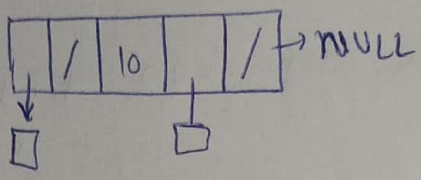
Note :- Degree of a tree is Pre-decided, ie. we cant decide the degree of a tree by just looking on it, ie. See in 3-ary tree fig (iii), it is a binary tree also, but we decleared it as 3-ary tree, so it will be a 3-ary tree.

Note :- More explanation :- Let us take a Node,



It can have 4 children, so it means it has capacity so it will be a 4-ary tree.

Now, Let us take

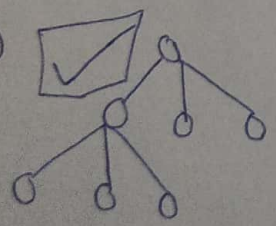


Now, It have 2 children, but it has capacity of 4, so, it doesn't mean that it is a binary tree, it is 4-ary tree, bcz it can capacity of 4. Hence, that's why we said, if we are making a tree, then we will decide the degree of a tree.

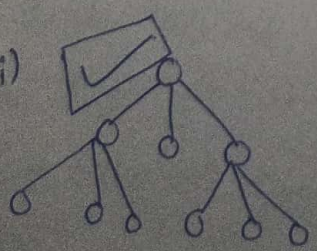
★ Strict m-ary tree :- In this type of tree, a node can have either '0' children or have exactly 'm' children.

Eg :- Strict 3-ary tree {0, 3}

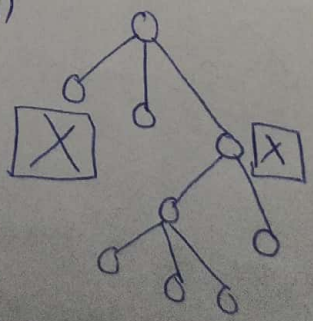
(i)



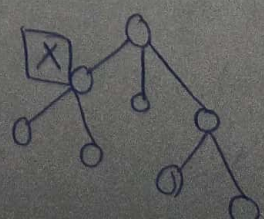
(ii)



(iii)



(iv)

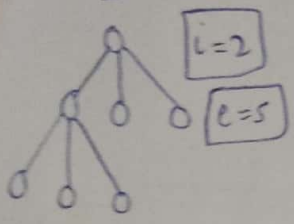


Analysis of n-Ary Trees :- Height vs. Nodes formulas

Strict 3-ary Trees :-

h = 2

min

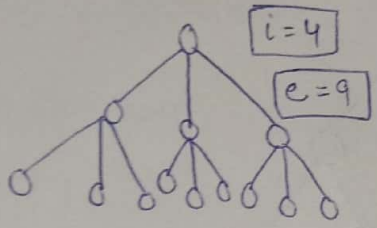


$n = 3+3+1$

$n = 2\times3+1$

$n = 7$

max.

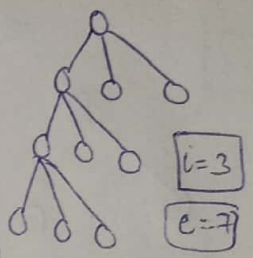


$1 + 3 + 3^2$

Here, 2 → height
3 → degree

h = 3

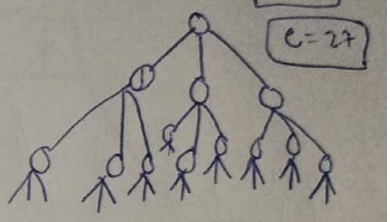
min



$n = 3\times3+1$

$n = 10$

max.



$1 + 3 + 3^2 + 3^3$

Here, 3 → degree

form :- $1 + m + m^2 + \cdots + m^h = \frac{(m^{h+1} - 1)}{m-1}$

Now, If Height is given :-

minimum no. of Nodes :- $n = mh + 1$

maximum no. of Nodes :- $n = \frac{m^{h+1} - 1}{m-1}$

Now, If 'N' nodes are given :-

Maximum Height of tree :- $h = \frac{n-1}{m}$

Minimum Height of tree :- $h = \log_m [n(m-1)+1] - 1$

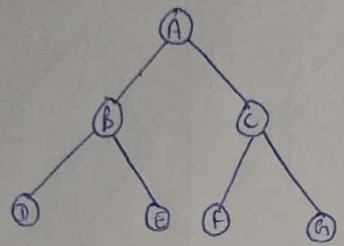
⇒ Relation b/w Internal and External Nodes :- for 3-ary :- $e = 2i+1$

for m-ary :- $e = (m-1)i+1$

* Representation of a Binary Tree :-

(1) ARRAY REPRESENTATION :-

For storing a binary tree, we have to store two things, one is the elements and other is relation between them.



T

A	B	C	D	E	F	G
1	2	3	4	5	6	7

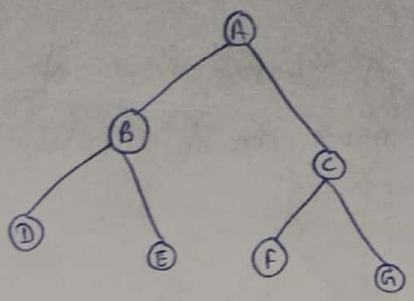
Element	Index	L child	R child
A	1	2	3
B	2	4	5
C	3	6	7

- **For A :-** It is present at index 1, its left child (B) is present at 2 and right child (C) is present at 3.
- **For B :-** It is present at index 2, its left child (D) is present at 4 and right child (E) is present at 5.
- **For C :-** It is present at index 3, its left child (F) is present at 6 and right child (G) is present at 7.

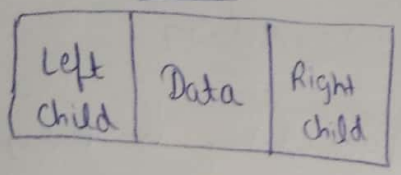
Formula :- "If any Element is at index 'i', then its left child will be at $(2*i)$ and right child be at $(2*i)+1$"

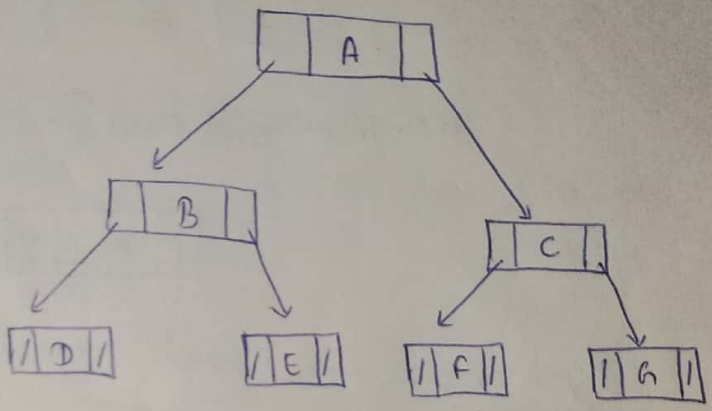
For Finding Parent :- $\left\lfloor \frac{i}{2} \right\rfloor$ → floor value

Linked Representation :- Here we will using Nodes.



Node → Structure





n = 7

No. of Null Pointers = 8

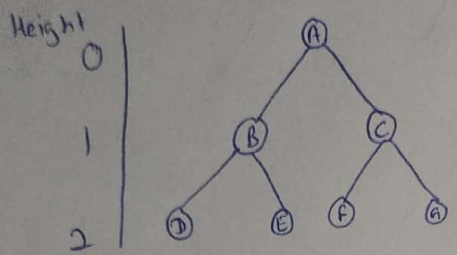
```
Struct Node ●→ (Self-Referential Structure)
{
    Struct Node * L child;
    int data;
    Struct Node * R child;
}
```

If we Representant a binary tree using Linked Lists, than always there will be (n+1) null pointers where, n → "no. of nodes"

Note :- Mostly we will use this type of Representation for algorithms.

* Full vs. Complete Binary tree :-

- **Full Binary Tree** :- A Binary tree of Height 'h', has ... mum no. of Nodes is a full bina...



Maximum no. of Node $\Rightarrow n = 2^{h+1} - 1 = 2^{2+1} - 1 = 8 - 1 \Rightarrow 7$

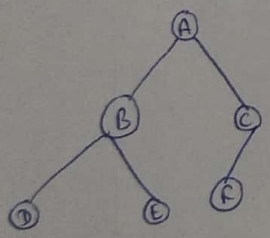
T

A	B	C	D	E	F	G
1	2	3	4	5	6	7

→ Representation in Array

- **Complete Binary Tree** :-

(i)

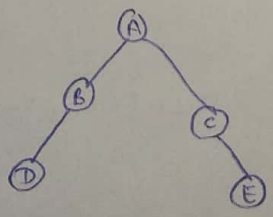


Representation in Array :-

T

A	B	C	D	E	F		
1	2	3	4	5	6	7	8

(ii)



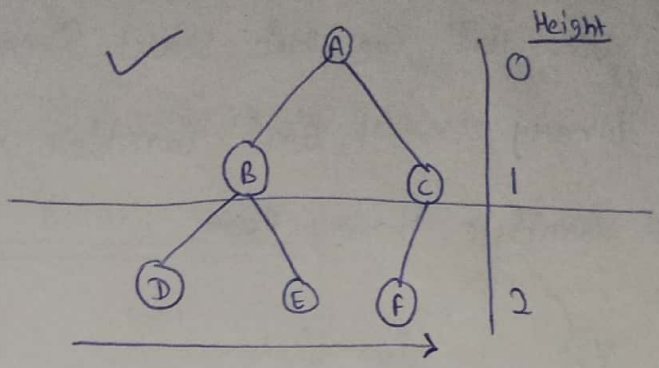
T

A	B	C	D	-	-	E
1	2	3	4	5	6	7

1.) In fig (i), we can see that, there are no blank spaces in b/w two elements while storing in an Array, so it is a Complete Binary tree.

2.) In fig (ii), we can see, there are blank spaces b/w two elements (i.e. D & E) while storing in an Array, so, it is not a Complete Binary tree.

Definition :- A Complete Binary tree of height 'h', will be a full binary tree at height 'h-1'.



Note :- 1.) Must be full Binary tree at height 'h-1'.
2.) Elements will be stored from left to right (i.e. in an array) without skipping.

Q- Why we need Complete Binary tree?

A- We know, we learnt Arrays and we use them implementing Stacks and Queues and everytime we dont have blanks spaces in blw Elements, if any element is deleted, we will shift the elements and occupy that place, then how we can blank spaces while representing a binary tree. thats why we use Complete Binary tree bcz it is suitable for an Array.

Imp Note :- 1.) A Full binary tree is always a Complete Binary tree.
2.) But a Complete Binary tree need not to be a Full Binary tree.

☆ Strict vs. Complete Binary tree :- Just for Diff. Binary

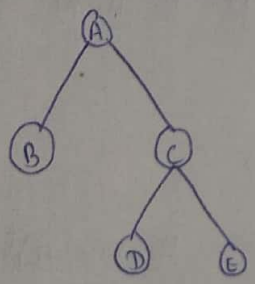
At some places, we will see that Strict Binary tree is called as Complete binary tree and Complete binary tree known as almost Complete Binary tree.

Strict vs. Complete

Strict → Complete

Complete → almost Complete

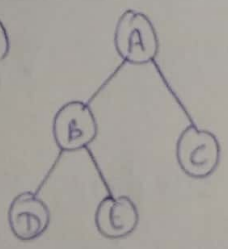
(i)



	1	2	3	4	5	6	7
T	A	B	C	-	-	D	E

✓ Strict
✗ Complete

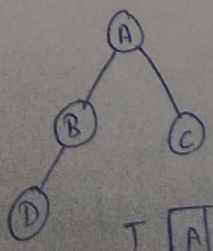
(ii)



T	A	B	C	D	E		

✓ Strict
✓ Complete

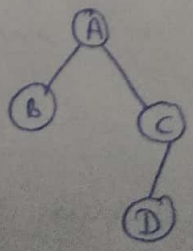
(iii)



T	A	B	C	D

✗ Strict
✓ Complete

(iv)



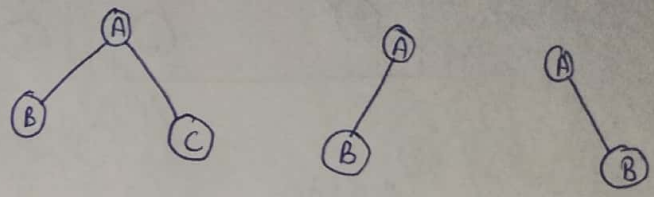
T	A	B	C	-	-	D

✗ Strict
✗ Complete

Binary Tree Traversal :-

Traversing means visiting on all nodes, if a data structure is linear than we can traverse left to right, or right to left.

But for Binary trees



① Preorder :- Visiting a node first, than Performing Preorder on Left-Subtree and than on Right Subtree.

⇒ visit (node) → Preorder (Left Subtree) → Preorder (Right Subtree)

② Inorder :- Perform Inorder on Left Subtree, than visiting a node, than Inorder on Right Sub tree.

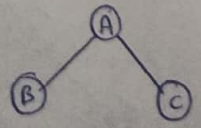
⇒ Inorder (left) → visit (node) → Inorder (right)

③ Postorder :- Perform Postorder on Left Sub tree, Perform Postorder on Right Subtree, than visit a node.

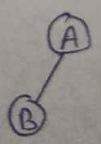
⇒ Postorder (left) → Postorder (Right) → visit (node)

④ Level order :- visit all the nodes, level by level.

Eg:-

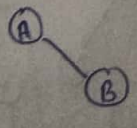


Pre :- A, B, C
In :- B, A, C
Post :- B, C, A
Level :- A, B, C



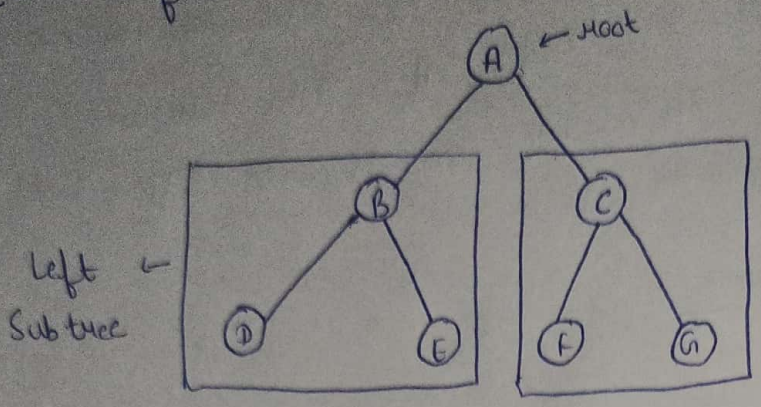
Pre :- A, B
In :- B, A
Post :- B, A
Level :- A, B

Same bcz not having right child



Pre :- A, B
In :- A, B
Post :- B, A
Level :- A, B

Eg :- To find a Traversal using Definitions



Preorder :- A, (B, D, E), (C, F, G)

⇒ A, B, D, E, C, F, G

Inorder :- (D, B, E), A, (F, C, G)

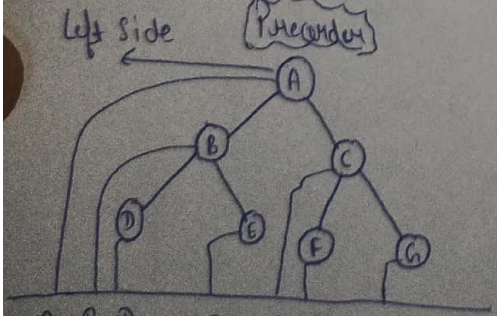
⇒ D, B, E, A, F, C, G

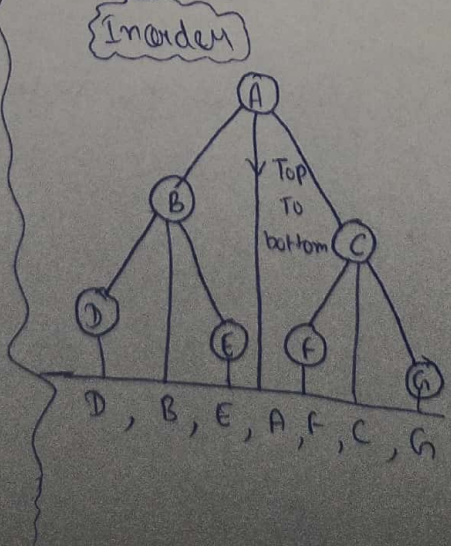
Post order :- (D, E, B), (F, G, C), A

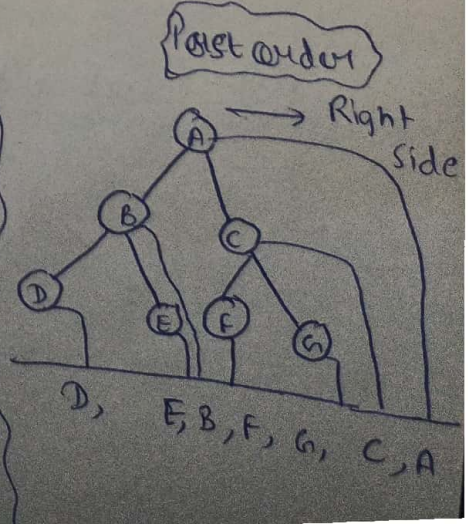
⇒ D, E, B, F, G, C, A

Level by Level :- A, B, C, D, E, F, G

★ Binary Tree Traversal Easy method -1 :-

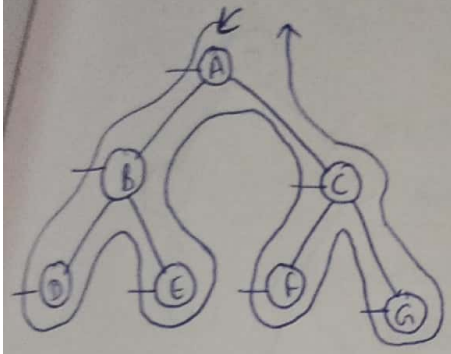






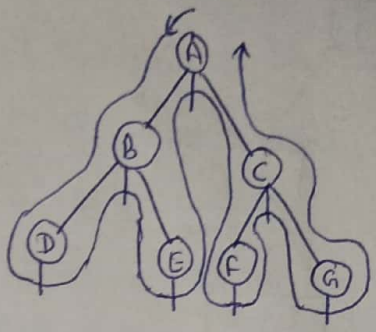
Binary Tree Traversal Easy method - 2 :-

Preorder



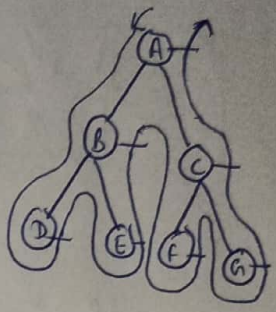
A, B, D, E, C, F, G

Inorder



D, B, E, A, F, C, G

Postorder

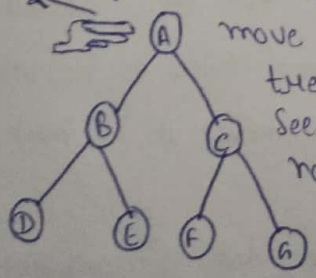


D, E, B, F, G, C A

☆ Binary Tree Traversal Easy method - 3 :-

Preorder

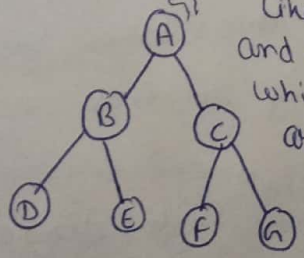
Put ur finger like this and move around tree and see which nodes are visible.



A, B, D, E, C, F, G

Inorder

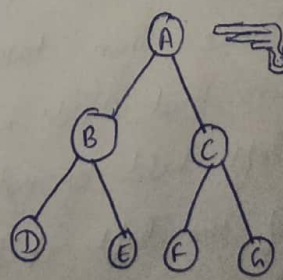
→ Put ur finger like this and See which nodes are visible



D, B, E, A, F, C, G

Post order

→ Put ur finger like this and then See. and move along the boundary.

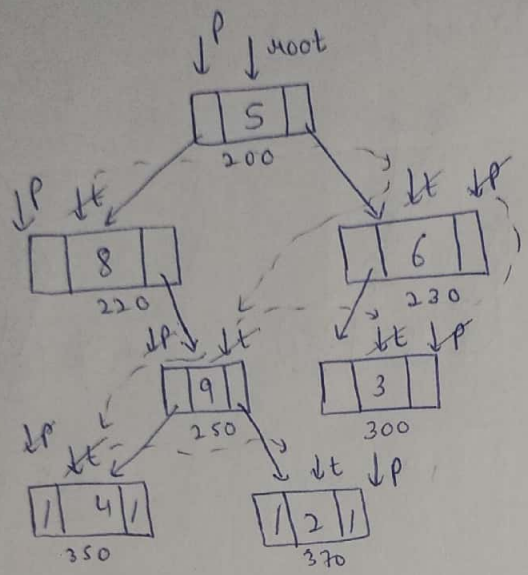


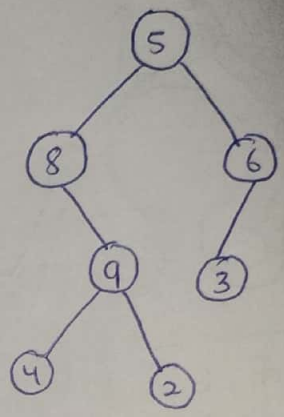
D, E, B, F, G, C, A

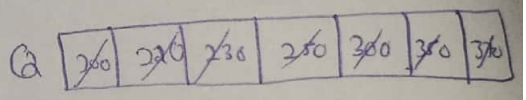
Note: "This method is applicable on any Binary tree"

☆ Creating a Binary Tree :-

We will Generate it level by level, we will use Queue Structure here.







Procedure :-

- First of all, we have to know that there is Left / Right Child is there or not and if it is present then we can attach it and create it
- We will Generate them Level by Level.
- Now, we need a Queue data structure having front & Rear pointer and all the mechanism we know.
- Create "Root" Node and Store Value in it and Store its address in Queue. Then take out this address from Queue and take a pointer on that Node, Standing on that node, ask whether it is having left child or not
- If the left child is there then create a node by taking temporary pointer and store its address in Queue.
- If the Right child is there then create a node by taking temporary pointer and store its address in Queue. and this Procedure going on.

Note :- "Here we have to ask whether left child is there or not, if yes then create it, same for right child, so instead of asking, we simply take value of left child and if value entered is -1, then there is no child

Program :-

```
Void Create ( )
{
    node *p, *t;
    int x;
    Queue q;
printf (" Enter root value");
Scanf ("%d", &x);
root = malloc (...);
root → data = x;
root → lchild = root → rchild = 0;
enqueue (root);
While (! isEmpty (q))
  {
    P = dequeue (&q);
    printf ("Enter Left child : \n");
    Scanf ("%d", &x);
     if (x != -1)
     {
        t = malloc(...) ; // creating node
        t → data = x;
        t → lchild = t → rchild = 0;
        P → lchild = t;
        enqueue(t);
     }
    printf (" Enter Right child : \n");
    Scanf ("%d", &x);
     if (x != -1)
     {
        t = malloc (...);
        t → data = x;
        t → lchild = t → rchild = 0;
        P → rchild = t;
        enqueue (t);
     }
}
}
```

→ for left child

→ for Right child

⇒ Code for creating Binary Tree :-

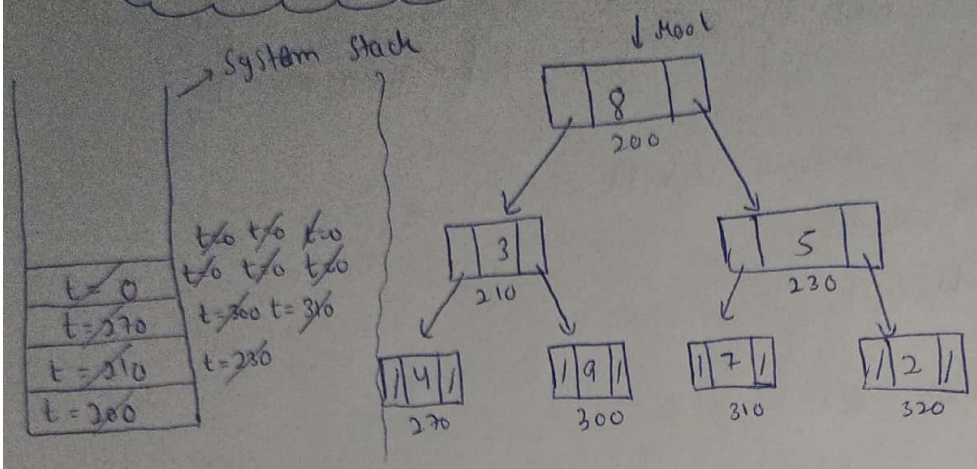
```
Struct Node
{
  Struct Node *lchild;
  int data;
  Struct Node *rchild;
};
Struct Queue
 {
   int Size
   int front;
   int Rear;
   Struct Node **Q; // *Q is for Array and **Q is
                       type (node). [ie (Struct Node *) *
 };
Struct Node *root = NULL ; // Global, so that we can use it
                             anywhere
 int main ()
{
  tree create ();
  printf (" Preorder traversal is : ");
  Preorder (root);
  printf ("\n");
  printf (" Inorder traversal is : ");
  Inorder (root);
  printf ("\n");
  printf (" Postorder traversal is : ");
  Postorder (root);
 }
```

```
void Preorder (struct Node *p)  // Recursive function
{
  if(p)
    {
     printf ("%d_", P -> data);
     Preorder (P -> lchild);
     Preorder (P -> rchild);
    }
}
void Inorder (struct Node *p)  // Recursive function
   {
     if (p)
      {
        Inorder (P -> lchild);
        printf ("%d_", P -> data);
        Inorder (P -> rchild);
      }
   }
   void Postorder (struct Node *p) // Recursive function
     {
       if (p)
        {
          Postorder (P -> lchild);
          Postorder (P -> rchild);
          printf ("%d_", P -> data);
        }
     }
     void create (struct Queue *q, int size)
      {
        q -> size = size;
        q -> front = q -> Rear = 0;
        q -> Q = (struct Node *) malloc (q -> size * sizeof (struct Node*));
      }
```

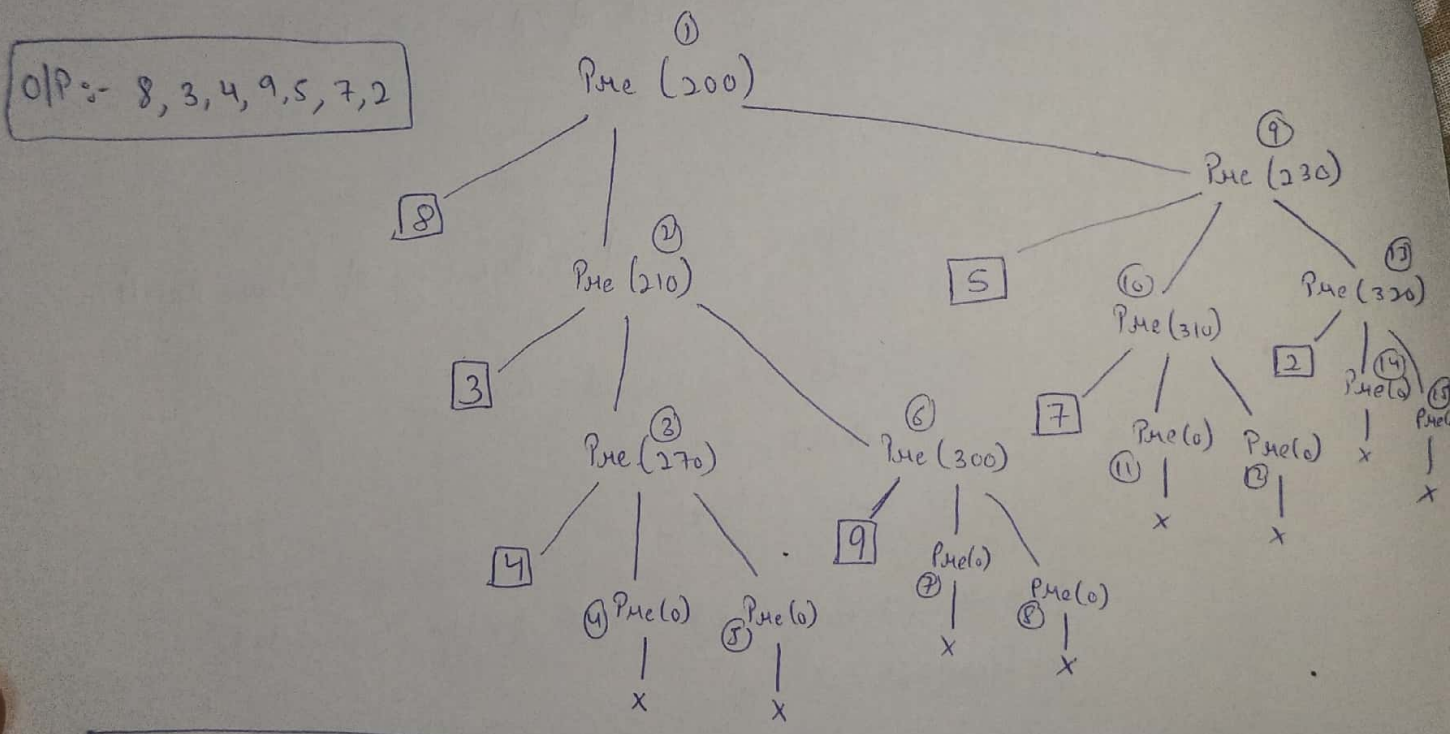
```
Void enQueue (struct Queue *q, struct Node * x)
{
    if (q->Rear == q->size -1)
        printf ("Queue is full");
    else
    {
        q->Rear ++;
        q->Q[q->Rear] = x;
    }
}
Struct Node * deQueue (struct Queue *q)
{
    Struct Node *x = NULL;
    if (q->front == q->Rear)
        printf ("Queue is Empty");
    else
    {
        q->front ++;
        x = q->Q[q->front];
    }
    return x;
}
int isEmpty (struct Queue q)
{
    return q.front == q.Rear; // if q.front = q.Rear then it
                              will return true else false
}
Void tree create ( )
{
    Struct Node *p, *t;
    int x;
    Struct Que ue q;
    Create (&q, 100);
```

```
printf("Enter the root value: \n");
scanf("%d", &x);
root = (struct Node *) malloc (sizeof (struct Node));
root → data = x;
root → lchild = root → rchild = NULL;
enqueue (&q, root);
while (! isEmpty (q))
{
    P = dequeue (&q);
    printf ("Enter left child of %d \n", P → data);
    scanf ("%d", &x);
    if (x != -1)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t → data = x;
        t → lchild = t → rchild = NULL;
        P → lchild = t;
        enqueue (&q, t);
    }
    printf ("Enter right child of %d \n:", P → data);
    scanf ("%d", &x);
    if (x != -1)
    {
        t = (struct Node *) malloc (size of (struct Node));
        t → data = x;
        t → lchild = t → rchild = NULL;
        P → rchild = t;
        enqueue (&q, t);
    }
}
}
```

☆ Preorder Traversal Explanation :-




```
Void Preorder (n
{
  if (t != NULL)
  {
    printf ("%d", t→d
    Preorder (t→lchild)
    Preorder (t→rchild)
  }
}
```



Total no. of Calls :- 15 Now, we can see that these calls are equal to sum of nodes and null pointers (i.e. $n+n+1 = 2n+1$)

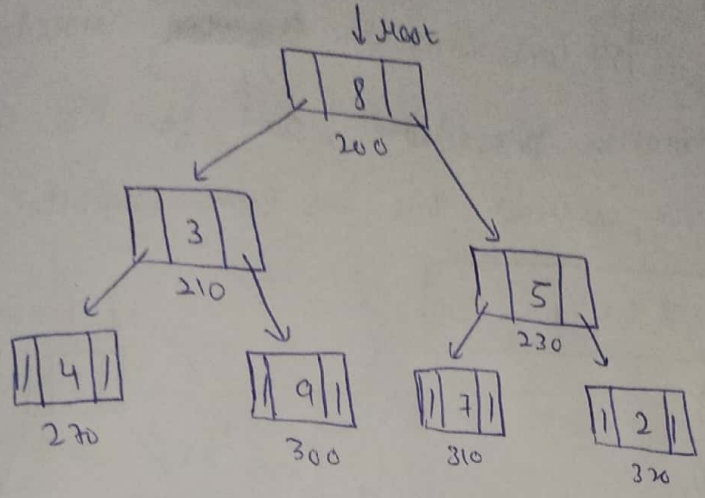
∴ Total no. of Calls = $2n+1$

No. of activation Records :- No. of levels + 1
Height of tree + 2

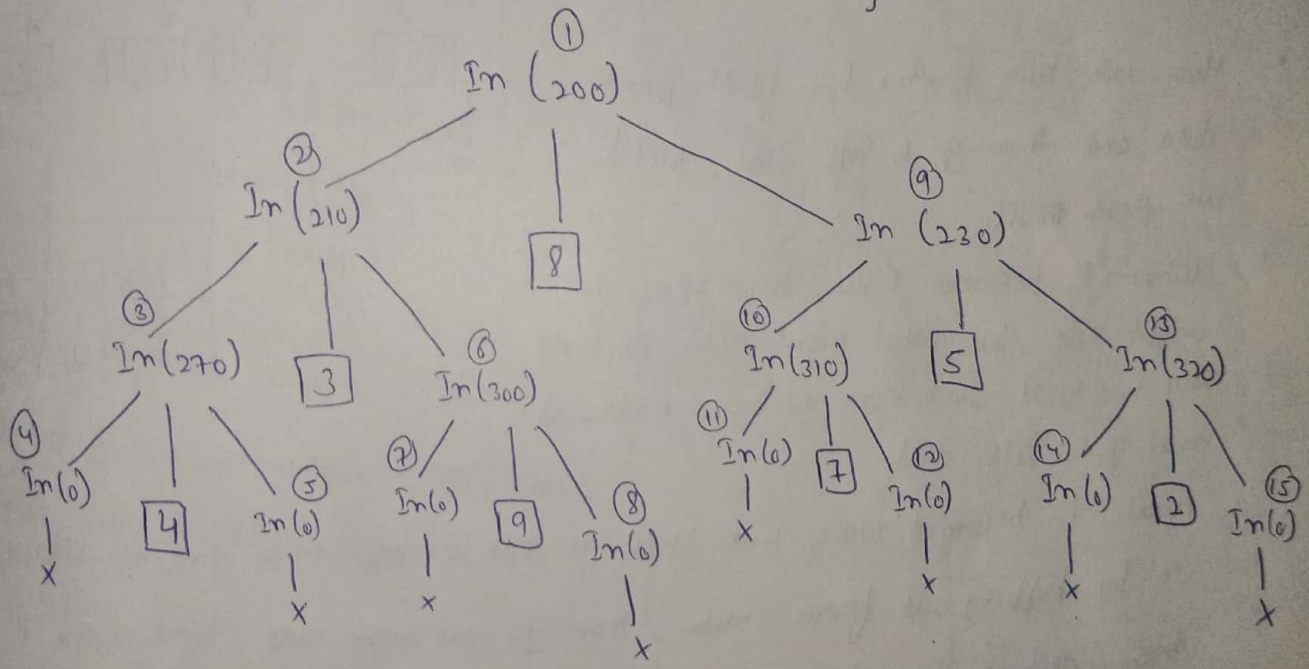
Time complexity ∝ No. of Calls (bcz printing is done only).

Time Complexity :- $O(n)$

Inorder Traversal Explanation :-




```
Void Inorder (Node *t)
{
  if (t != NULL)
  {
    Inorder (t → lchild);
    printf ("%d", t → data);
    Inorder (t → rchild);
  }
}
```



o/p :- 4, 3, 9, 8, 7, 5, 2

Total no. of Calls :- $2n+1$ $(O(n))$

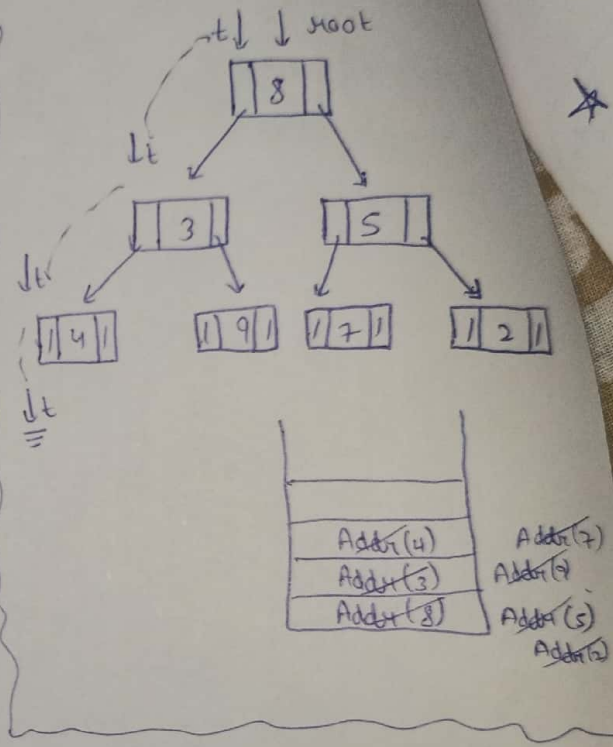
Here, We can See that it just prints the value and printing depends upon no. of Calls, So, Time Complexity :- $O(n)$

Iterative Preorder :- In some situations we require for converting an recursive Procedure to iterative procedure, and for this we need our own Stack, without this, we can't convert.

O/P :- 8, 3, 4, 9, 5, 7, 2

Procedure :-

```
Print (data);
Pre (l child);
Pre (r child);
```



- Here we have pointer t, first print data and then go to left side, until we reach NULL.
- When 't' becomes NULL, then move to right side, but whose right side, so that's why we need Stack here to store addresses then go to left child
- Now, 't' becomes NULL, but Stack is not Empty, then give its address to 't' by popping out from Stack, then go on right side, and again 't' becomes NULL but Stack is not Empty, so again popping out.
- Do this again & again until, 't' becomes NULL and there will be Nothing in the Stack.

Code :-

```
void Preorder (Node *t)
{
    Struct stack st;
    while (t!=NULL || !isEmpty (st))
    {
        if (t!=NULL)
        {
            printf ("%d", t->data);
            push (&st, t);
            t = t->lchild;
```

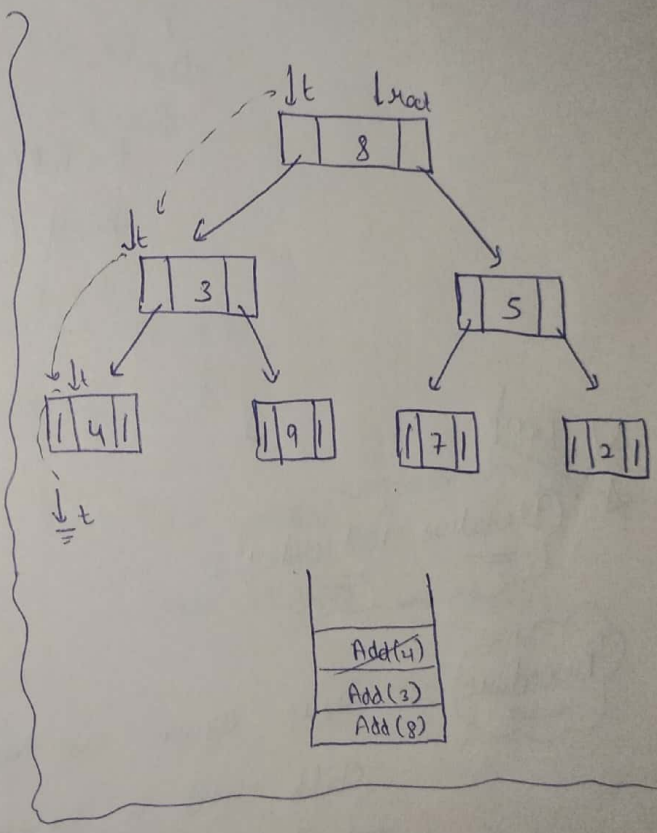
```
else
  {
    t = Pop(st);
    t = t -> rchild;
  }
 }
}
```

Time Complexity :- O(n)

★ Iterative Inorder :-

Procedure :-

```
In (lchild);
Print (data);
In (rchild);
```



- Now, Here again, we take a pointer 't' on root node, Now move this to left child until it becomes NULL.
- Now, When it become NULL, Popping out the address, (i.e. Address which we store in the Stack during moving to left child).
- Then after popping that address and give it to 't', print that data and then move to right child
- Do this again & again until 't becomes NULL and Stack is empty.

Code :-

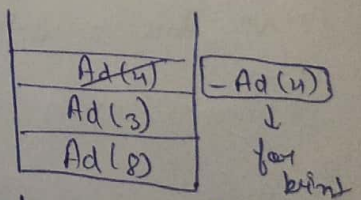
```
Void Inorder (Node *t)
{
  Struct Stack St;
  while (t != NULL || !isEmpty(St))
  {
    if (t != NULL)
    {
      Push (&St, t);
      t = t -> lchild;
    }
    else
    {
      t = Pop(&St);
      printf ("%d", t -> data);
      t = t -> rchild;
    }
  }
}
```

Time Complexity :- O

V. Imp

☆ Iterative Post order :-

Procedure :-



- Here again we have a pointer 't' and move to left child until it becomes NULL and Push the Address into the Stack.
- When t becomes NULL, Pop out and again Push the address and then go to right child and when again 't' becomes NULL then again Pop out, But this will be confusing bcz same Address so we push the negative address for printing the data for differentiate by storing it in temporary variable.

```
Void Postorder (Node *t)
{
    Struct Stack st;   → For this we have to create a long int
    long int temp;  // for storing Negative address type stack
    While (t!=NULL || !isEmpty(st))
    {
        if (t!=NULL)
        {
            Push (&st, t);
            t = t→lchild;
        }
        else
        {
            temp = Pop (&st);
            if (temp > 0)
            {
                Push (&st, -temp);
                t = ((Node *) temp) → rchild;
            }
            else
            {
                printf ("%d", ((Node *) temp)→data);
                t = NULL;
            }
        }
    }
}
```

Temp is having an address but it is an integer value, so we type cast it into address and then move to right child.

★ Code for All Iterative traversals :-

```
Struct Node
{
    Struct Node *lchild;
    int data;
    Struct Node *rchild;
}

Struct Stack
{
    int size;
    int top;
    long int *s;
};

Struct Queue
{
    int size;
    int front;
    int Rear;
    Struct Node **Q;
}

Struct Node * root = NULL;

Void StackCreate (struct Stack * st, int size)
{
    st -> size = size;
    st -> top = -1;
    st -> s = (long int *) malloc (st -> size * sizeof (long int));
}
```

```
void Push (struct Stack *st, long int x)
{
    if (st->top == st->size-1)
        printf ("Stack overflow\n");
    else
    {
        st->top++;
        st->s[st->top] = x;
    }
}
long int Pop (struct Stack *st)
{
    long int x=-1;
    if (st->top ==-1)
        printf (Stack underflow\n");
    else
    {
        x= st->s[st->top--];
    }
    return x;
}
int isemptyStack (struct Stack st)
{
    if (st.top==-1)
        return 1;
    return 0;
}
int isfull Stack (struct Stack st)
{
    if (st.top == st.size-1)
        return 1;
    return 0;
}
```

```
Void Preorder1 (struct Node *P)
{
    Struct Stack St;
    StackCreate (&St, 100);
    While (P || isemptyStack (St))
    {
        if (P != NULL)
        {
            printf ("%d -", P->data);
            Push (&St, P);
            P = P->lchild;
        }
        else
        {
            P = Pop (&St);
            P = P->rchild;
        }
    }
}

Void Inorder1 ( struct Node *P)
{
    Struct Stack St;
    StackCreate (&St, 100);
    while (P || isemptyStack (St))
    {
        if (P != NULL)
        {
            Push (&St, P);
            P = P->lchild;
        }
        else
        {
            P = Pop (&St);
            printf ("%d -", P->data);
            P = P->rchild;
        }
    }
}
```

```
Void Postorder1 (Struct Node *P)
{
    Struct Stack st;
    StackCreate (&st, 100);
    long int temp; // for storing negative address
    while (P!=NULL || !isEmptyStack (st))
    {
        if (P!=NULL)
        {
            Push (&st, (long int) P);
            P = P -> lchild;
        }
        else
        {
            temp = Pop(&st);
            if (temp >0)
            {
                Push (&st, -temp);
                P = ((Struct Node *) temp) -> rchild;
            }
            else
            {
                printf("%d-", ((Struct Node *) -temp) ->data);
                P=NULL;
            }
        }
    }
}
```

```
Void create (struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = 0;
    q->Q = (struct Node **) malloc (q->size * sizeof (struct
```

```
Void enQueue (struct Queue *q, struct Node * x)
{
    if (q->rear == q->size-1)
        printf ("Queue is full");
    else
    {
        q->rear ++;
        q->Q [q->rear] = x;
    }
}
```

```
Struct Node * deQueue (struct Queue *q)
{
    struct Node * x = NULL;
    if (q->front == q->rear)
        printf ("Queue is Empty");
    else
    {
        q->front ++;
        x = q->Q [q->front];
    }
    return x;
}
```

```
int isEmpty (struct Queue q)
{
    return q.front == q.rear;
}
```

```
void levelorder (struct Node *P)  // for level order traversal, we will use queue for this
{ struct queue q;
  create (&q, 100);
  printf ("%d_", P->data);
  enqueue (&q, P);
  while (!isEmpty (q))
   {
     P= dequeue (&q);
     if (P->lchild)
       {
         printf ("%d_", P->lchild->data);
         enqueue (&q, P->lchild);
       }
     if (P->rchild)
       {
         printf ("%d_", P->rchild->data);
         enqueue (&q, P->rchild);
       }
   }
}
Void treecreate ()
{
    struct Node *P, *t;
     int x;
     struct queue q;
     create (&q, 100);
     printf (" Enter the root value : \n");
     scanf (" %d", &x);
     root = (struct Node *) malloc (sizeof (struct Node));
```

```
root -> data = x;
root -> lchild = root -> rchild = NULL;
enqueue (&q, root);
while (!isempty (q))
{
    P = dequeue (&q);
    printf ("Enter the left child of %d : \n", P->data);
    scanf ("%d", &x);
    if (x != -1)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t -> data = x;
        t -> lchild = t -> rchild = NULL;
        P -> lchild = t;
        enqueue (&q, t);
    }
    printf ("Enter right child of %d : \n", P -> data);
    scanf ("%d", &x);
if (x != -1)
  {
    t = (struct Node *) malloc (sizeof (struct node));
    t -> data = x;
    t -> lchild = t -> rchild = NULL;
    P -> rchild = t;
    enqueue (&q, t);
  }
}
}
```

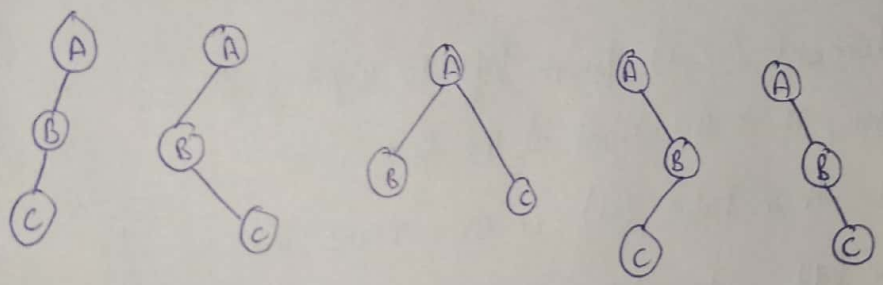
```
int main ()
{
  tree create ();
printf ("Preorder Traversal
Preorder (root);
printf ("Inorder Traversal is
Inorder (root);
printf ("Postorder Traversal
Postorder (root);
printf ("Levelorder Traver
levelorder (root);
}
```

Can we Generate Tree From Traversals

"It means that can we create a unique tree from traversals or not" Actually we find traversals from tree but can we generate a tree from traversals.

$n = 3$ (A) (B) (C)

Suppose, it have preorder traversal with 3-nodes



preorder :- ABC

postorder :- CBA

We can see that there are multiple trees from same preorder but we want a single tree, so we can't generate a tree.

Conclusion :- • From preorder, we can't generate a single tree. and same for Inorder & postorder

Imp. Can we Generate a tree from two traversals,

1) From preorder & postorder, we cant generate a unique tree as shown above.

preorder, postorder, Inorder → Total no. trees $\left(\frac{2nC_n}{n+1}\right)$

preorder + postorder → 1+ (more than one)

Imp :- "We can Generate tree from these two traversals" :-

- preorder + Inorder
- postorder + Inorder

For Generating a Tree, Inorder Traversal is must, because it allows us to visit left child, then root then right child, so we can know whats in Left side / whats in Right side.

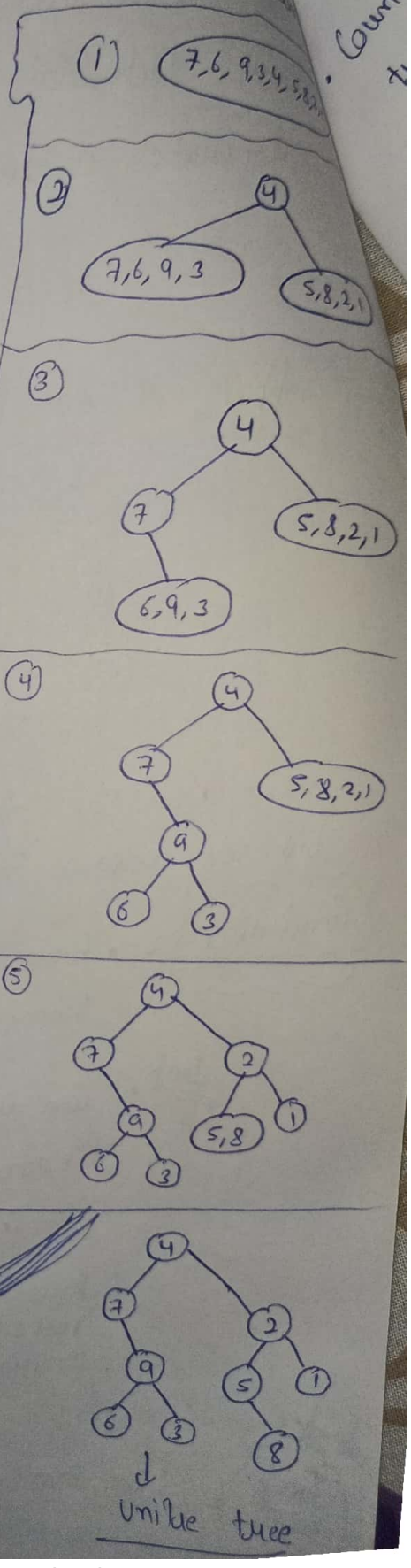
Eg:-

Preorder → 4, 7, 9, 6, 3, 2, 5, 8, 1

Inorder → 7, 6, 9, 3, 4, 5, 8, 2, 1

- Create a node, take all the elements of inorder.
- Now, Scan through Preorder by taking one element at a time.
- Take first element (i.e. 4) from left to right and we know inorder, it is the root of the tree, so search for (4), then take (4) in one node, then which are on left side, take them on Left Side and which are not right side, Take them on right side, (i.e. Search then split)
- Now, we will search (7), first we will check it in Left side, as we search in left side first in Preorder, if not found, then go on right side.
- Then splitting goes on and this procedure goes on until last.

Note :- Now, the resultant tree will be the unique tree for this Preorder & inorder traversal, Hence no more trees can be generated from this Preorder & Inorder.

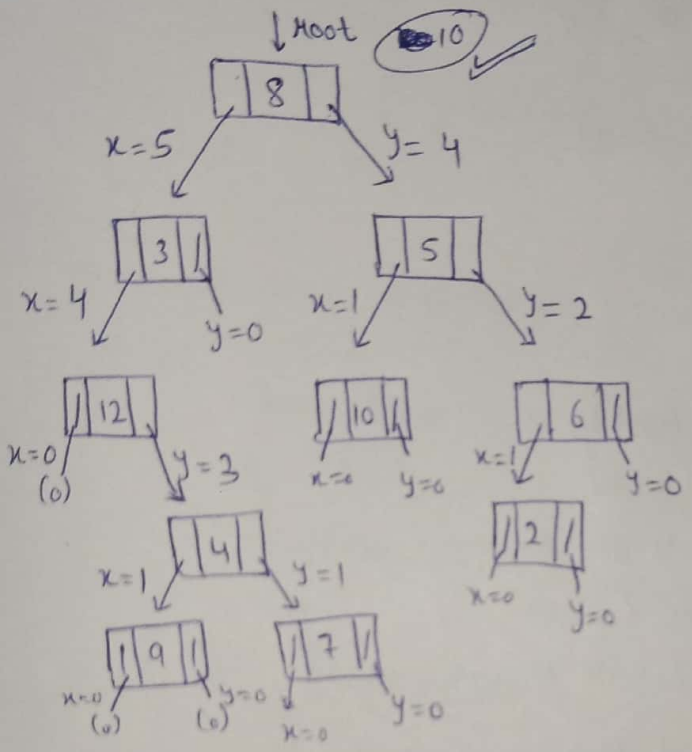


Height and Count of a tree (Binary)

① Counting Total no. of Nodes
② Height of a Tree
③ Nodes with degree 2
④ Counting no. of leaf nodes
⑤ Nodes with degree 1

:-

1. Counting no. of Nodes in Binary tree.




```
int Count (Node *P)
{
    int x, y;
    if (P != NULL)
    {
    1. x = Count (P→ lchild);
    2. y = Count (P→ rchild);
    3. Return x+y+1;
    }
    Return 0;
}
```

- This Procedure is working in Postorder form and most of the times we will use Postorder while processing

Code :- "In this Code, there are some more functions for student challenges".

```
Struct Node
{
    Struct Node * lchild;
    int data;
    Struct Node * rchild;
};
```

```
Struct Queue
  {
    int Size;
    int front;
    int Rear;
    Struct Node **Q;
  };
Struct Node * Root = NULL;
int main ( )
  {
    tree Create ( );
    int m = Count (Root);
    int n = height (Root);
    int P = Count node with degree 2 (Root);
    int L = leaf nodes (Root);
    int H = Count node with degree 1 (Root);
    printf (" Total no. of Nodes : %d \n", m);
    printf (" Height of Tree : %d \n", n-1);
    printf (" No. of nodes with degree two : %d \n", P);
    printf (" No. of leaf nodes : %d \n", L);
    printf (" No. of Nodes with degree one : %d \n", H);
  }
void Create (Struct Queue *q, int Size)
  {
    q → Size = Size;
    q → front = q → Rear = 0;
    q → Q = (Struct Node * *) malloc (q → Size * sizeof (struct
                                                    node *))
  }
```

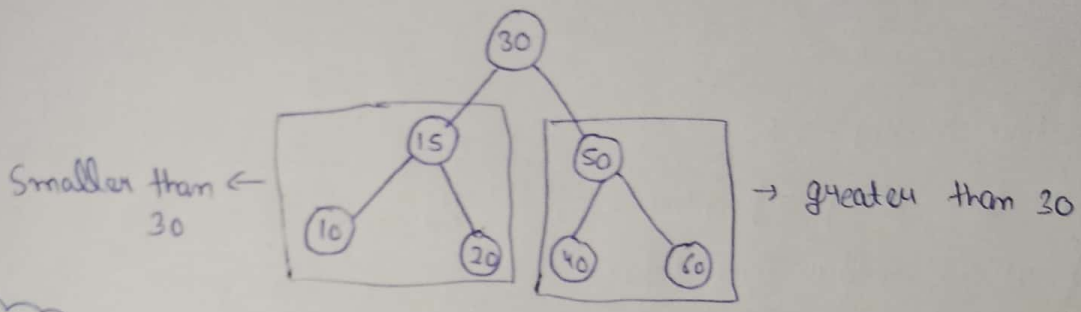
```
void enqueue (struct Queue *q, struct Node *x)
{
  if (q->Rear == q->Size-1)
      printf ("Queue is Full");
  else
    {
      q->Rear++;
      q->Q[q->Rear] = x;
    }
}
struct Node * dequeue (struct Queue *q)
  {
     struct Node *x = NULL;
     if (q->front == q->Rear)
         printf ("Queue is Empty");
     else
       { q->front++;
         x = q->Q[q->front];
       }
     return x;
  }
int isEmpty (struct Queue q)
   {
     return q.front == q.Rear;
   }
Void treeCreate ()
{
    struct Node *p, *t;
    int x;
    struct Queue q;
    Create (&q, 100);
```

```
printf ("Enter the root value : \n");
scanf ("%d", &x);
root = (struct Node *) malloc (size of (struct Node));
root → data = x;
root → lchild = root → rchild = NULL;
enqueue (&q, root);
while (! isEmpty (q))
 {
    P = dequeue (&q);
    printf ("Enter left child of %d : \n", P→ data);
    scanf ("%d", &x);
    if (x!=-1)
    {
      t = (struct Node *) malloc (size of (struct Node));
      t → data = x;
      t → lchild = t → rchild = NULL;
      P → lchild = t;
      enqueue (&q, t);
    }
    printf ("Enter right child of %d : \n", P → data);
    scanf ("%d", &x);
    if (x!=-1)
    {
      t = (struct Node *) malloc (size of (struct Node));
      t → data = x;
      t → lchild = t → rchild = NULL;
      P → rchild = t;
      enqueue (&q, t);
    }
 }
}
```

```
int Count (struct Node *P) // for Counting Total no. of Nodes
{ if (P)
   return  Count (P->lchild) + Count (P->rchild) +1;
   else
   return 0;
}
int Height (struct Node *P) // for Height of a tree
{
   int x=0, y=0;
     if (P==0)
     return 0;
     x= Height (P->lchild);
     y= Height (P->rchild);
     if (x>y)
       return x+1;
     else
       return y+1;
   }
int CountNode with degree 2 (struct Node *P) // for Counting Nodes with degree 2.
{
   int x,y;
   if (P!=NULL)
     {
        x= Count Node degree 2 (P->lchild);
        y= Count Node degree 2 (P->rchild);
        if (P->lchild && P->rchild)
           return x+y+1;
        else
        }
           return x+y;
     else
        return 0;
}
```

```
int leafnodes(struct Node *P)   // counting No. of leaf nodes
{
    int x, y;
    if (P != NULL)
    {
        x = leafnodes(P->lchild);
        y = leafnodes(P->rchild);
        if (P->lchild == 0 && P->rchild == 0)
            return x+y+1;
        else
            return x+y;
    }
    else
        return 0;
}

int Countnodewithdegree1(struct Node *P)   // for Counting Nodes with degree 1.
{
    int x, y;
    if (P != NULL)
    {
        x = Countnodewithdegree1(P->lchild);
        y = Countnodewithdegree1(P->rchild);

        if (P->lchild != NULL ^ P->rchild != NULL)
        // it means (P->lchild == NULL && P->rchild != NULL  !!
        //           P->lchild != NULL && P->rchild == NULL)
        // to reduce this large condition, we use exclusive (OR) (^) operator.
            return x+y+1;
        else
            return x+y;
    }
    else
        return 0;
}
```

BINARY SEARCH TREES



Introduction :- It is a binary tree in which for every node, all the elements in its left sub-tree are smaller than that node and all the elements in its right sub-tree are greater than that node

This binary tree is helpful for searching thats why it is known as Binary Search tree, like if we want to search for 40, it is greater than 20, so it will be on right side, then we search for 40, we see 50, so 40 is less than 50, so we go on left side and there we find 40.

So, we find / search in less time like Binary Search, but we do binary Search in Array and Here we search in a tree, thats why it known as Binary Search tree. And Search time depends upon Height of tree.

Properties :- (1) This tree doesn't have Duplicates.

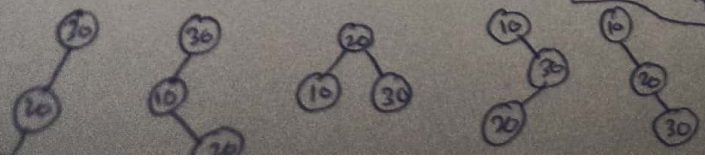
(2) If we take inorder traversal, we get Sorted list (10, 15, 20, 30, 40, 50, 60).

Imp

(3) No. of BST for 'n' nodes :- we know, if we take inorder traversal we get sorted order, Let us take n=3, 10 20 30

Inorder :- 10 20 30

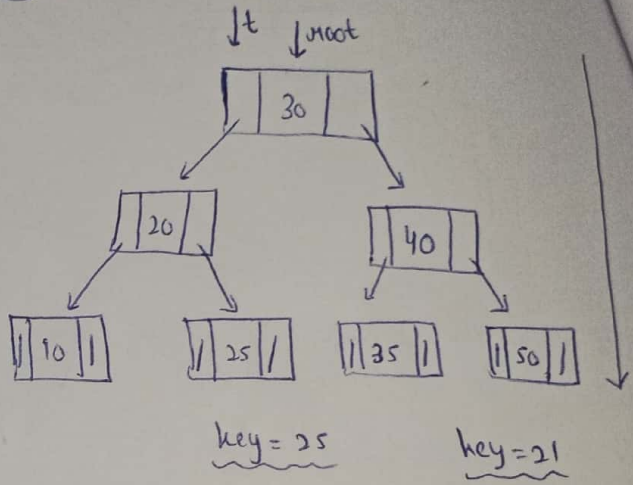
So, No. of different trees = 5 $\Rightarrow$ $T_n = \frac{2nC_n}{n+1}$



Note :- If we change order of these nodes, we didn't get same inorder, thats why Every shape can be only filled in one way

Recursive

★ Searching in a Binary Search tree



• For Succesful Search

① Let us take a key = 25

② Check it with using a pointer 't', if it is smaller than data then move 't' on right side and if data is greater than move it to left side.

③ Time taken for Searching depends upon height of tree

$O(h)$, $\log n \leq h \leq n$

• For Unsucceful Search

① Movement of 't' pointer is same as above process

② but if 't' becomes NULL, it means it reaches the end of the tree So, element is not found

③ Time taken is $O(n)$ [i.e. maximum]

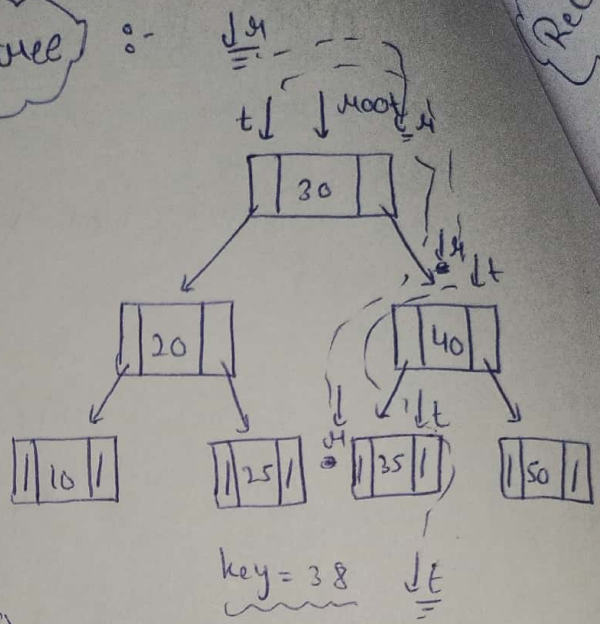
Recursive approach for Searching :- [Tail Recursion]

```
Struct Node * RSearch (Node *t, int key)
{
    if (t == NULL)
        return NULL;
    if (key == t->data)
        return t;
    else if (key < t->data)
        return RSearch (t->lchild, key);
    else
        return RSearch (t->rchild, key);
}
```

* Iterative approach for Searching :- V.Imp Note :- We know, when we convert a recursive function into an iterative function, we may require a stack, but here we don't need any stack and the main point is that in tail recursion, we never need a stack.

```
Node * ISearch (Node *t, int key)
{
    while (t != NULL)
    {
        if (key == t->data)
            return t; // if key is found
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    return NULL; // if key is not found
}
```

☆ Inserting in a Binary Search tree :-



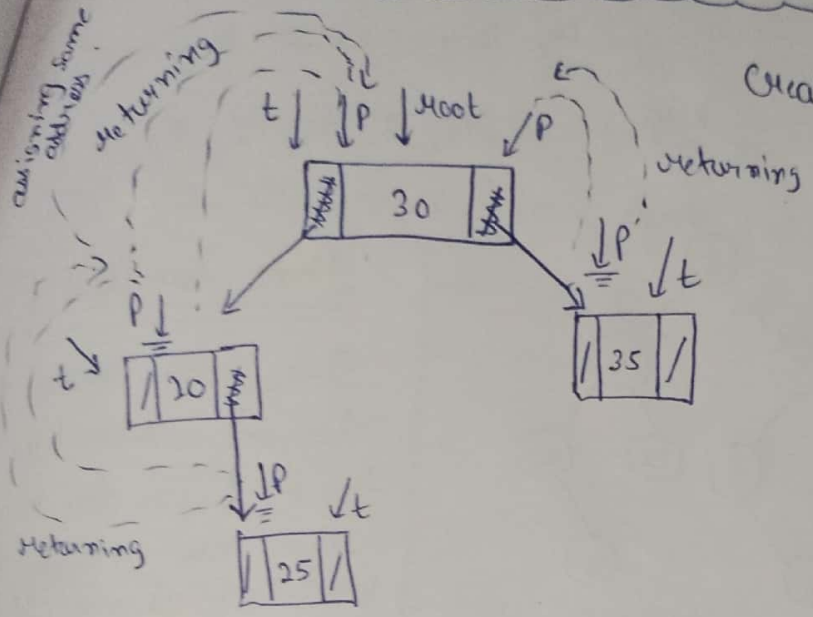
① First step, is to search for key value, is it already present or not, bcz we know there are no duplicates in a binary search tree

② We already know how to search, so Search 38, while searching, we can see, it is not there and 't' become NULL, so, where 't' terminates that place will be for the key = 38, Here it is right child of 35, but Now for insertion, we need a pointer on '35', so we need a tailing pointer, which follows 't' and when 't' becomes NULL, it points to '35'.

```
void Insert (Node *t, int key)
{
  Node *r = NULL, *p;
  while (t != NULL)
  {
    r = t;
    if (key == t->data)
        return; // match is found
    else if (key < t->data)
        t = t->lchild;
    else
        t = t->rchild;
  }
  p = malloc(...);
  p->data = key;
  p->lchild = p->rchild = NULL;
  if (p->data < r->data) r->lchild = p;
  else r->rchild = p;
}
```

Time Complexity :-
Actually time is only for Searching, thats why time is of $O(\log n)$.

Recursive Insert in Binary Search tree :- we can use this function also for creating a BST (Binary Search tree)




```
Node * Insert (Node*P, int key)
{
    Node *t;
    if (P == NULL)
    {
        t = malloc (...);
        t→data = key;
        t→lchild = t→rchild = NULL;
    }
    if (key < P→data)
        P→lchild = Insert (P→lchild, key);
    else if (key > P→data)
        P→rchild = Insert (P→rchild, key);
    else
    return t;
}
```

```
int main()
{
    Node * root = NULL;
    root = Insert (root, 30);
        Insert (root, 20);
        Insert (root, 35);
```

Procedure :- ① First we create, a root Node and insert it using 't' pointer and its value will assigned to root.

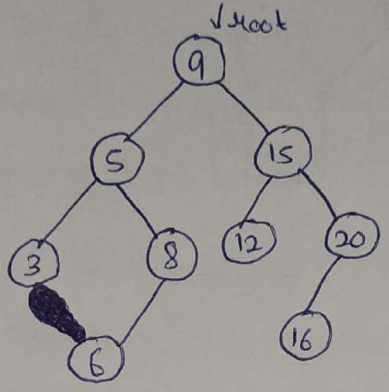
② Now, P will not NULL, and let we insert '20', it is less than root, so, call this function again by passing P→lchild, but it is NULL, so create a new Node and now this new node returns, and after returning 'P' was at root node, so, make P→lchild assigned with result of that function.

③ Now, All calls are working on this Procedure.

Creating a Binary Search tree :-

Generating a ... from given set of ... in same order they are giv...

keys :- 9, 15, 5, 20, 16, 8, 12, 3, 6



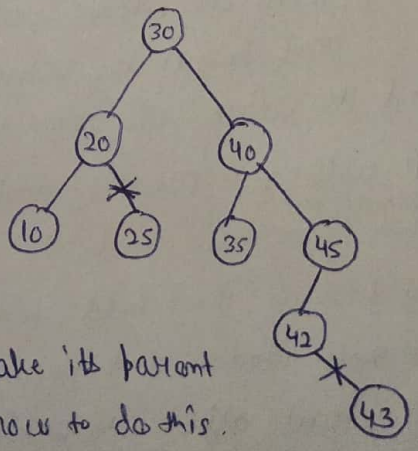
Time Complexity :-

Since we insert 'n' elements which take O(n) time & we also do searching which take O(log n) time and we search at every step, so Time Complexity will be

O [n log(n)]

V. Imp

☆ Deleting from Binary Search tree :-

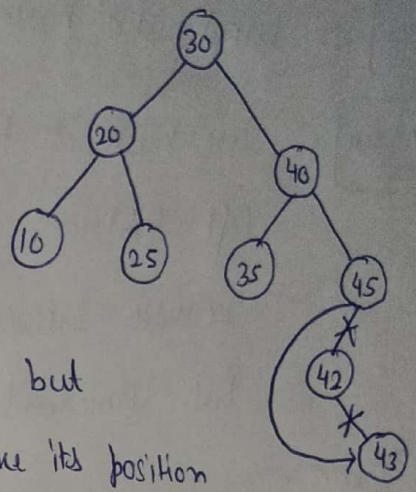


(1) Condition - 1 :-

(a) key = 25, Search for the key and then delete it. Now if we want to delete (25), we want to make its parent (20) NULL, as we already know how to do this.

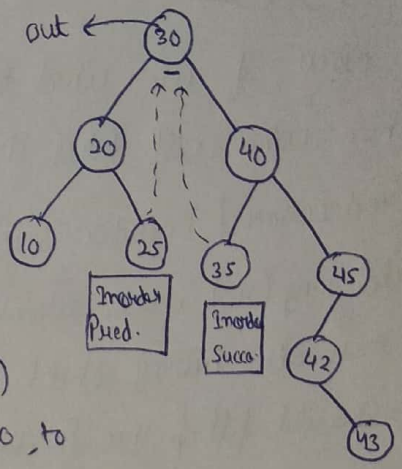
(b) key = 43, Search for key and then delete it and make its parent (42) as NULL.

Condition :- 2



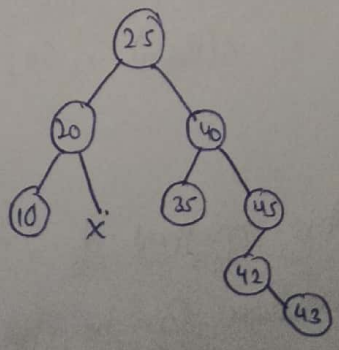
key = 42, than Search for it,
if we want to delete (42), than we have to modify the link of its parent but (42) also have child, than its child will take its position

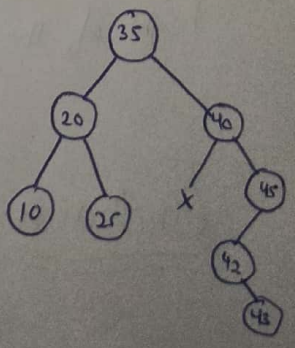
Condition :- 3



Now, if we want to delete key = 30, than who will take its place either (20) | (40) than who will take position of (25) Either (10) | (25) or who will take position of (40) Either (35) | (45) than again for us & so, on, so, to avoid this method, For (30) found Inorder Predecessor, if we perform Inorder than which Node Comes before (30) [i.e. 25] and found which Node Comes after (30) that will Inorder Succesor, so either Inorder Predecessor will take its place | Inorder Succesor will take its place.

Hence, Results Can be,



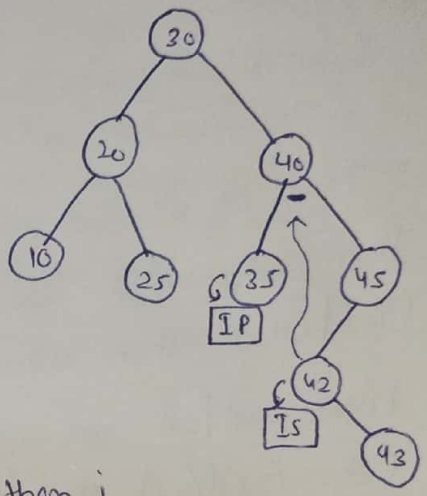


Note :- To find Inorder Predecessor for deleting a node, go to its left Subtree, than go rightmost node, than it will be Inorder Predecessor and for Inorder Succesor go to right sub tree and than leftmost child/node.

* **Ques** :- What we have to take Inorder Predecessor / Inorder Successor

Answer :- We have to take Acc. to the Height of the right-Subtree / left Subtree, if height of right subtree is longer then we take Inorder Successor otherwise we will take inorder Predecessor, But Randomly, we can take anyone.

Condition -4 :-



Now, if we want to delete key = 40, then who will takes its place, Inorder Predecessor / Inorder Succesor, Suppose after deleting (40), we want (42) to be here, but it is also having child, it is not a leaf node, then i should fill up this Position also with its inorder Predecessor / Succesor, but we take Inorder Predecessor / succesor to avoid multiple modifications in the tree, but here if we take, then we have to make multiple modifica

So, Here, Total time for Searching will be $O(\log n)$ and for modifications depends upon Inorder Predecessor / succesor further depends upon height of the tree is $O(\log n)$, so, Total Time taken $[O \log(n)]^2$.

Conclusion :- This method of Inorder Predecessor / Inorder Successor Can be applicable to any of the Condition, So we will use this to make Code.

Code for Creating, Searching, Inserting / Deleting in a Binary Search Tree

```
Struct Node
{
    Struct Node * lchild;
    int data;
    Struct Node * rchild;
} * root = NULL; // globally Accessible

int main ()
{
    void insert (int key);
    void inorder (struct Node *P);
    Struct Node * Search (struct Node * t, int key);
    Struct Node * Rinsert (struct Node *P, int key);
    Struct Node * delete ( struct Node *P, int key);
    root = Rinsert (root, 28);
    Rinsert (root, 50);
    insert (10);
    insert (20);
    insert (5);
    insert (8);
    delete (root, 8);
    delete ( root, 20);
    inorder (root);
    Struct Node *k = Search (root, 10);
    if (k != NULL)
     printf (" Element %d is found \n", k->data);
    else
     printf (" Element is not found");
}
```

```
Void insert (int key)   // Iterative insertion
{
    Struct Node *t = root;
    Struct Node *r, *p;
    if (root == NULL)
    {
        P = (Struct Node *) malloc (size of (Struct Node));
        P→data = key;
        P→lchild = P→rchild = NULL;
        root = P;
    }   Return;
    while (t != NULL)
    {
        r = t;
        if (key < t→data)
            t = t→lchild;
        else if (key > t→data)
            t = t→rchild;
        else
            Return;
    }
    P = (Struct Node *) malloc (sizeof (Struct Node));
    P→data = key;
    P→lchild = P→rchild = NULL;
    if (key < r→data)
        r→lchild = P;
    else if (key > r→data)
        r→rchild = P;
}
```

```
struct Node * Rinsert (struct Node *P, int key) // Recursive insert
{
    struct Node *t = NULL;

    if (P == NULL)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t->data = key;
        t->lchild = t->rchild = NULL;
        return t;
    }
    if (key < P->data)
        P->lchild = Rinsert (P->lchild, key);
    else if (key > P->data)
        P->rchild = Rinsert (P->rchild, key);
    else
        return P;
}
Void inorder (struct Node *P)
{
    if (P)
    {
        inorder (P->lchild);
        printf ("%d_", P->data);
        inorder (P->rchild);
    }
}
```

```
{
    while (t!=NULL)
    {
        if (key == t->data)
            return t;
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    return NULL;
}

int height (struct Node *p) // for calculating Height
{
    int x, y;
    if (p==NULL)
        return 0;
    x = height (p->lchild);
    y = height (p->rchild);
    return x>y ? x+1 : y+1;
}

Struct Node * inPre (struct Node *p) // for finding inorder Predecessor
{
    while (p && p->rchild != NULL)
        p = p->rchild;
    return p;
}

Struct Node * in succ (struct Node *p)
{
    while (p && p->lchild != NULL)
        p = p->lchild;
    return p;
}
```

```
struct Node * Delete (struct Node *P, int key) // For deletion
{
    struct Node *q;
    if (P== NULL)
        return NULL;
    if ( P→lchild == NULL && P→rchild == NULL)
    {
        if (P== root) // For root Node
            root = NULL;
            free (P); // For leaf Node
        return NULL;
    }
    if (key < P→data)
        P→lchild = Delete (P→lchild, key);
    else if (key > P→data)
        P→rchild = Dlete (P→rchild, key);
    else
    {
        if (height (P→lchild) > height (P→rchild))
        {
            q = inPre (P→lchild); // Calling Recursively
            P→data = q→data;
            P→lchild = Delete (P→lchild, q→data);
        }
        else
        {
            q = inSucc (P→rchild);
            P→data = q→data;
            P→rchild = Delete (P→rchild, q→data);
        }
    }
    return P;
}
```

Generating BST from Preorder

In Binary [redacted] trees, we know we can generate by

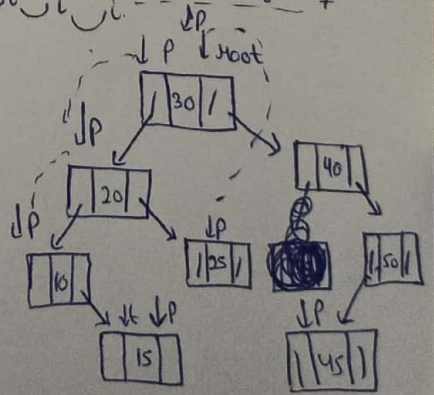
1) Preorder + Inorder
2) Post[redacted]order + Inorder

But in Binary Search tree, we can generate by the above process also, but we can create through either Preorder / Postorder bcz we know, Inorder of Binary Search tree gives sorted order, so if we have Preorder / Postorder only then we will generate inorder and then we can generate BST, "But with only Preorder / Postorder we can generate BST ☺ [inorder].

Eg :- Just using Preorder :-

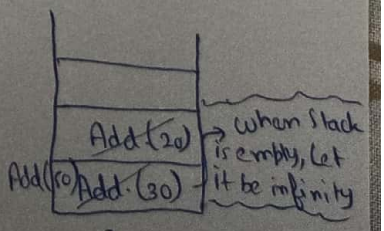
Pre

30	20	10	15	25	40	50	45
0	1	2	3	4	5	6	7



Procedure :-

- ★ For this we need a stack.
- ★ Initial step, take Pre(0), and make a node & point root to it and take temp. pointer P also.
- ★ Now, Repeating Steps, Now see Next value if it is less than P→data, then take a temp-pointer 't' and create a node, Set the data and make P→left point to it and push address of P.
- ★ Again, creating Nodes and pushing the Addresses, Now when we reach Pre(3), Now check it is less than 20 but greater than 10, so attach it with right of 10, but dont Push address of 10.
- ★ Again, for 25, Now check is it lying b/w Add(20) → 20 and 15, it is not lying So, it will not be right child of 15, so pop out Add(20), assigned to P and make it, its right child.



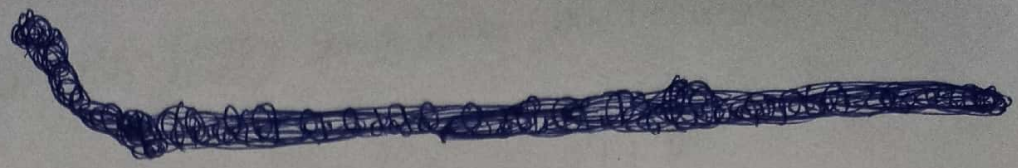
As we input

:. As we are doing Preorder traversal, So we have to give input in preorder traversal. (like :- 30 20 10 40 50)

```
Struct Node
{
    Struct Node *lchild;
    int data;
    Struct Node *Rchild;
} *Root = NULL;

Struct Stack
{
    int size;
    int top;
    Struct Node **S;
};
int main ()
{
    int n, i;
    printf ("Size of an Array : \n");
    Scanf ("%d", &n);
    int Pre [n];
    Void CreatePre (int Pre [], int n);
    Void Inorder (struct Node *P);
    for (i=0; i<n; i++)
    {
        scanf ("%d", &Pre[i]);
    }
    CreatePre (Pre, n);
    Inorder (Root);
}
```

```
Void StackCreate (struct Stack *st, int Size)
{
    st -> Size = Size;
    st -> top = -1;
    st -> s = (struct Node * *) malloc (st -> Size * sizeof (struct Node *));
}
Void Push (struct Stack *st, struct Node *x).
{
    if (st -> top == st -> Size -1)
        printf ("Stack overflown\n");
    else
      {
        st -> top++;
        st -> s [st -> top] = x;
      }
}
Struct Node *PoP (struct Stack *st)
   {
      Struct Node *x;
      if (st -> top == -1)
          printf ("Stack underflown\n");
      else
        {
          x= st -> s [st -> top --];
        }
      return x;
   }
int isEmpty (struct Stack st)
   { if (st.top == -1)
        return 1;
     return 0;
   }
```

```
Struct Node * stacktop (struct Stack St)
{
    if (! isempty (st))
    {
        Return St.s[st.top];
    }
    Return NULL;
}

Void createPre (int Pre[], int n)
{
    Struct Stack st;
    Stack create (&st, 100);
    Struct Node *t, *P;
    int i=0;
    Root = (struct Node *) malloc (size of (Struct Node));
    Root → data = Pre[i++];
    Root → lchild = Root → rchild = NULL;
    t = Root;
    while (i<n)
    {
        if (Pre[i] < t → data)
        {
            P = (struct Node *) malloc (size of (Struct Node));
            P → data = Pre[i++];
            P → lchild = P → rchild = NULL;
            t → lchild = P;
            Push (&st, t);
            t = P;
        }
```



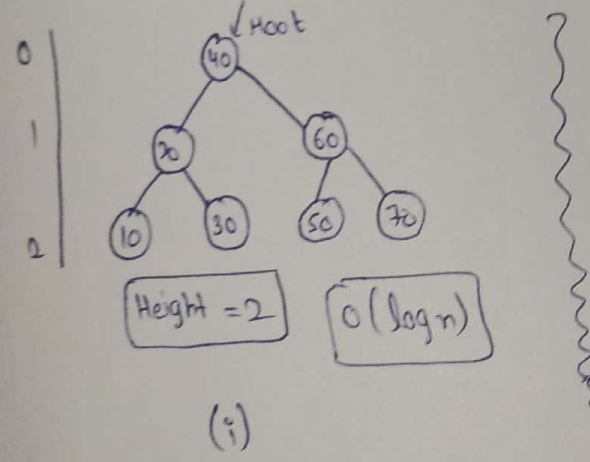
```
else
{
  if (Pre[i] > t->data && Pre[i] < (isEmpty(st) ? 32767 : StackTop(st)->data))
    {
      P = (struct Node *) malloc (sizeof (struct Node));
      P->data = Pre[i++];
      P->lchild = P->rchild = NULL;
      t->rchild = P;
        t = P;
    }
      else
        t = Pop (&st);
    }
}

Void Inorder (struct Node *P)
  {
    if (P)
     {
        Inorder (P->lchild);
        printf ("%d", P->data);
        Inorder (P->rchild);
     }
  }
```

Drawbacks of Binary Search Tree

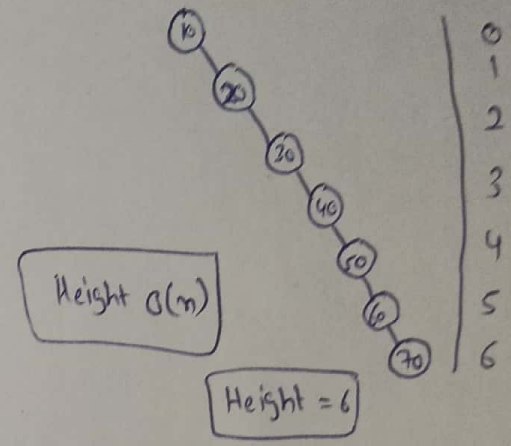
Suppose we have two set of keys and we have to generate binary search tree for them.

keys :- 40, 20, 30, 60, 50, 10, 70
keys :- 10, 20, 30, 40, 50, 60, 70

} both sets have same keys but are having different order



(i)



Height = 6

(ii)

Now, we can see in (i), height is $O(\log n)$ and for (ii) height is $O(n)$ for the same set of keys, Now, The Drawback is that the minimum Height is $O(\log n)$ and maximum Height is $O(n)$ but it is not depending upon no. of nodes, it depends upon how we insert the keys, it means height depends upon order of insertion.

- So, can we control the Height of Binary Search tree?

⇒ No, bcz suppose we are making a program for user, so we dont know how he/she will use, so, we can't control it, but "Binary Search trees itself can control its height, which we called as A-V-L Trees, [i.e. AVL Trees are height balanced Binary Search trees], we will learn about further them.

AVL TREES

These are Height balanced trees and Height can be balanced by using Balance factor.

Balance factor = Height of left Subtree - Height of Right Subtree

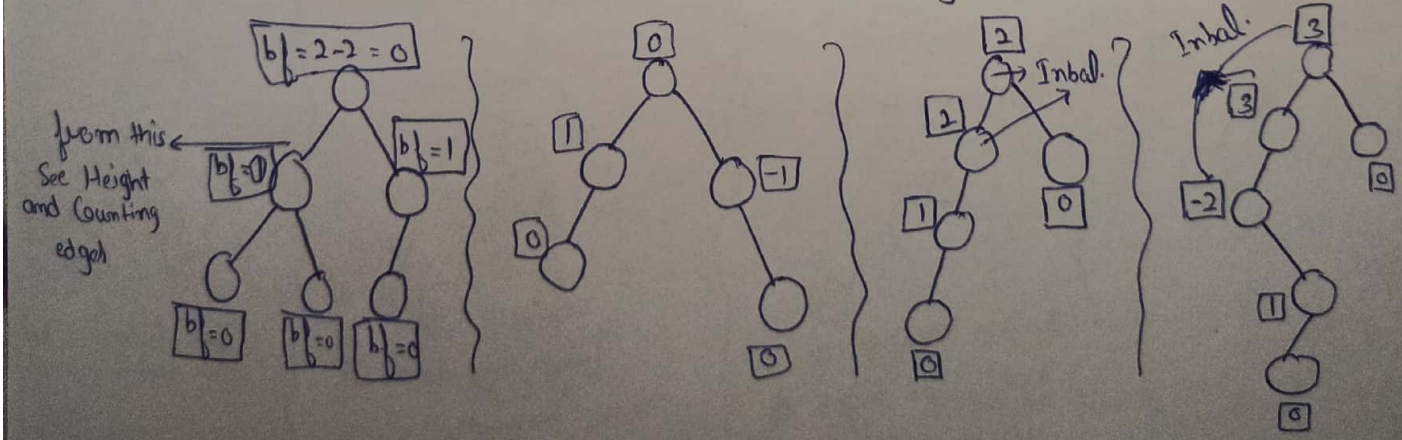
$$\boxed{bf = hl - hr} = \{-1, 0, 1\} \Rightarrow \text{valid balance factor}$$

On every node, we will calculate these, in an AVL Tree.

$$bf = |hl - hr| \leq 1$$

if $\boxed{bf = |hl - hr| > 1}$ → Node will be inbalanced

Note :- If any node is inbalanced, then AVL Tree will be inbalanced and we will make it balanced using/by doing rotations



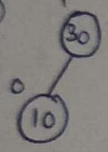
Note :- We can't / Never get $\boxed{bf = 3|-3}$, so last example is incorrect.

★ Inserting in AVL With Rotations :-

When we insert AVL Trees, some nodes become imbalanced or we can say tree will become imbalanced & hence, we perform rotations, and these rotations are as follows :

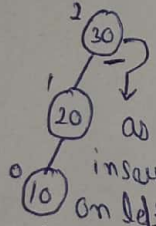
1. LL - Rotation
2. RR - Rotation
3. LR - Rotation
4. RL - Rotation

Initial



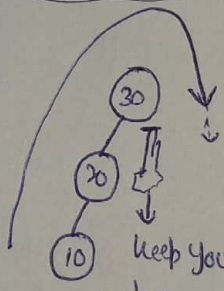
After Inserting

let, we insert(10)



as we insert a node on left of left so, it will be named as LL-imbalanced.

Perform Rotation (LL)

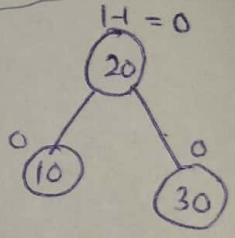


Keep your finger here and pull the thread

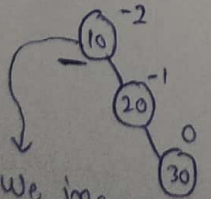
Name of Rotation :-

LL - Rotation

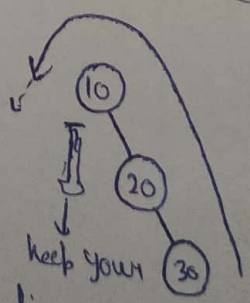
After Rotation(LL)



let we insert (30)



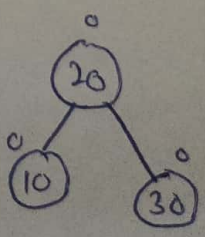
as we insert a node on right of right, so it will be named as RR-imbalanced



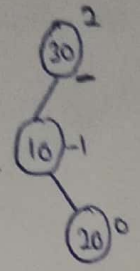
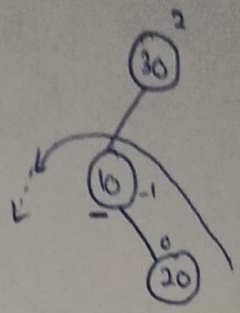
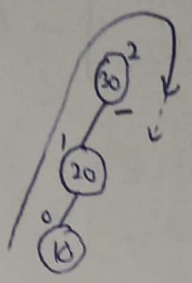
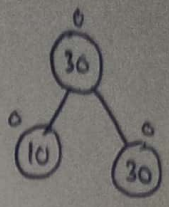
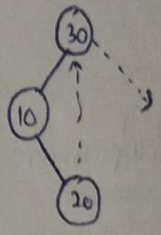
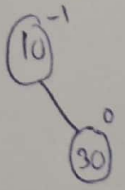
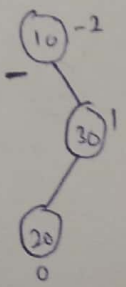
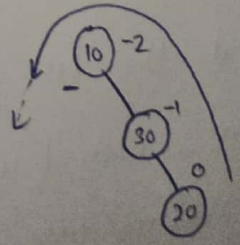
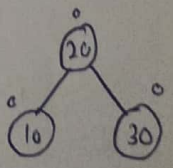
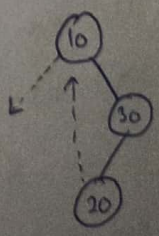
finger here and pull the thread.

Name of Rotation :-

RR - Rotat...



After Rotation(RR)

Initial	After Inserting	Perform Rotation	After Rotation
	Lets insert (30)  Now, Here we insert 20, but it will insert on Left to right, So it will known as **LR-imbalance**	  First left Rotation then right Rotation, actually it is a single Rotation but two step involves, so we call it as double Rotation.	 Single step we can see as :- 
D] 	let insert (20)  Now, Here we insert 20, but it will insert on Right to left, So it will known as **R-L imbalance**	  First right Rotation then Left Rotation.	 

Note :- (1) A node on which we perform rotation, its degree become zero (0).

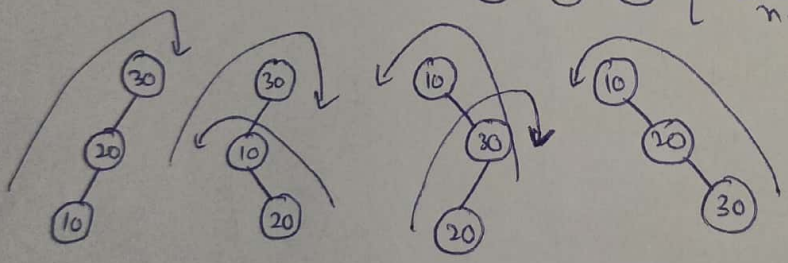
(2) Rotations will performed only on three nodes always, eg. if a tree is degree (having 1000 nodes) rotation will performed only 3-nodes.

LL - Rotation
RR - Rotation
→ Single Rotation

LR - Rotation
RL - Rotation
→ Double Rotation (dont count it as 2 rotations, actually it involves 2 steps).

Important / Logical observation :- We know, from three nodes, we can create 5 BSTs.

$n = 3$ (10) (20) (30) $\left[\frac{2nCn}{n+1}\right]$ → Catalan's formula

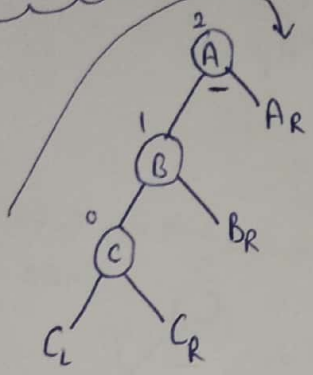


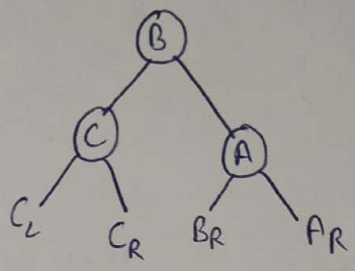
↓
Smaller Height (more Height balanced)

⇒ We can see that the last tree is more height balanced, so why we don't take that one, even if we get the earlier 4 trees convert them to get a height balance tree. This is the origin of AVL trees that is why AVL Trees are more preferable as they are height balanced, and even we can convert other BST to AVL Trees by performing rotations.

General form of AVL Rotations :- Formula of Rotations for Insertion.

For LL Rotation :- May be 1000 Nodes in a tree (Assume)

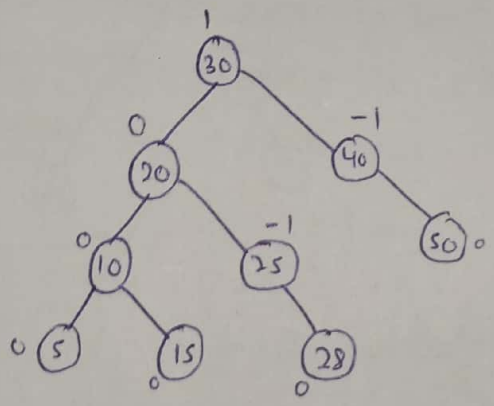




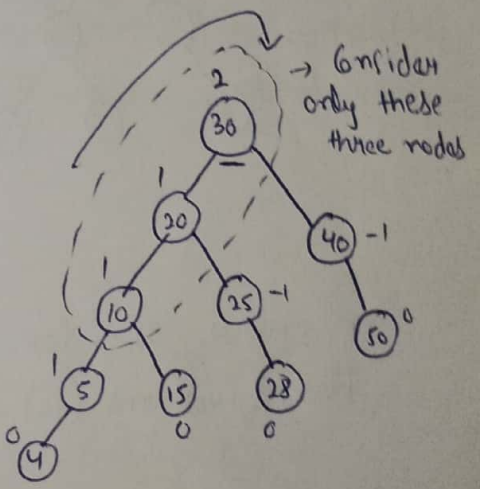
C_L, C_R, A_R → as it is

B_R → becomes Left child of A

Eg :-

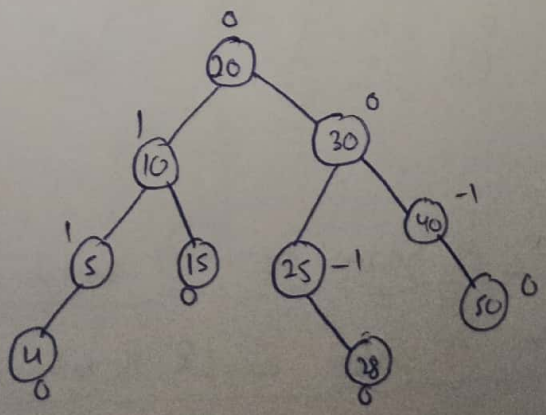


Insert key = 4 ⟹

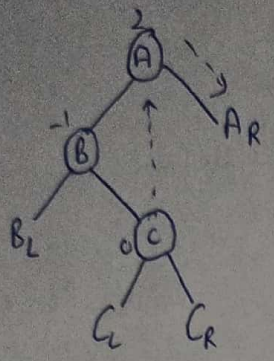


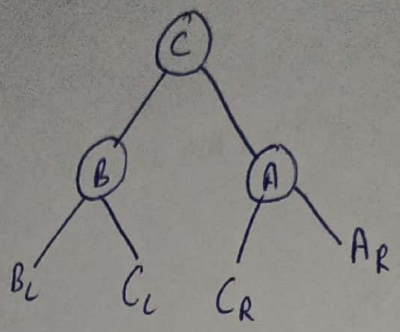
→ Consider only these three nodes

Check, this insertion is LL imbalance at seen from (30), so Now after Rotation.

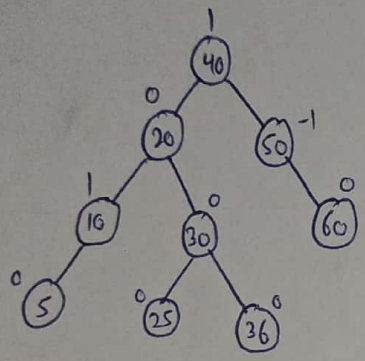


For LR Rotation :-

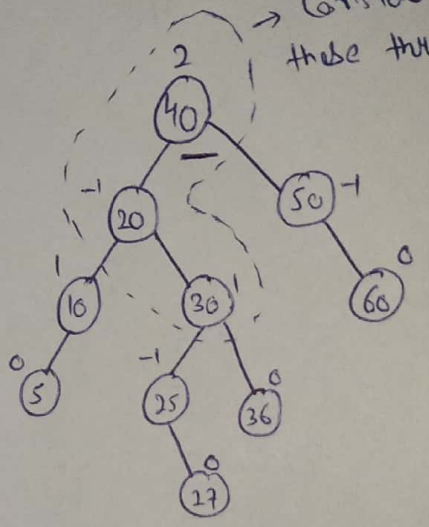




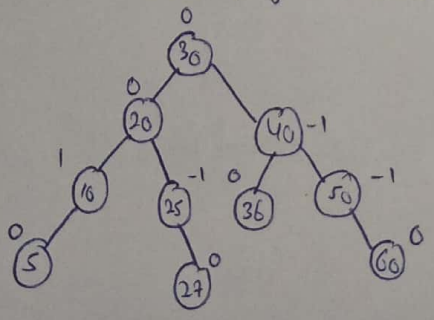
Eg :-



Insert key=27 ⟶



Now, We can see it is LR imbalance, so now, (30) will go to place of (40) and (40) will move on right hand side, so Now After Rotation.



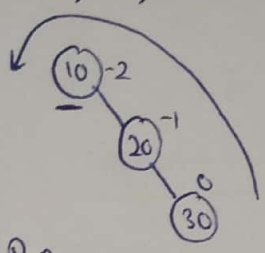
Note :- RR Rotation is Same as LL Rotation.

RL Rotation is Same as LR Rotation.

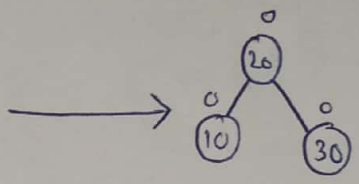
Generating AVL Tree :-

Keys :- 10, 20, 30, 25, 28, 27, 5

Insert 10, 20, 30



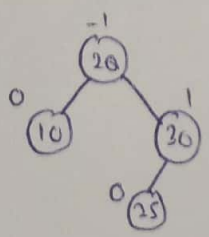
RR - Rotation



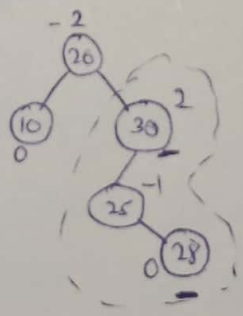
This is the movement we will perform when a node will be imbalanced.

Note :- if we didnt perform this Rotation and proceed with imbalancing then the balance factor may becomes more than (2) or less than (-2) which we dont want it, So thats why we will perform Rotatation when ever any node is imbalanced.

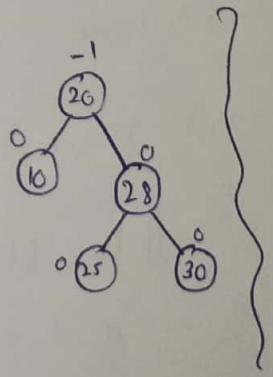
Insert 25



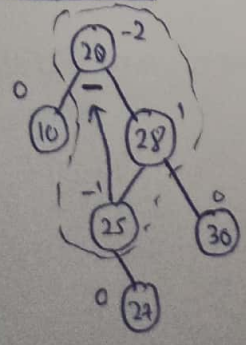
Insert 28



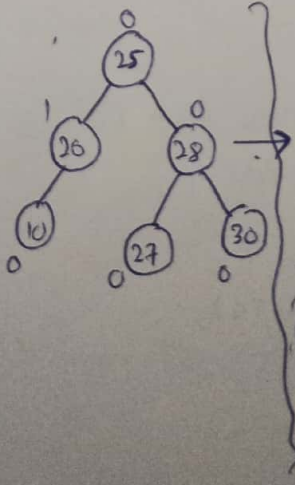
LR Rotation



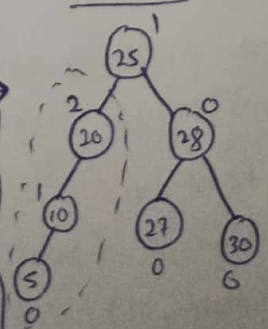
Insert 27



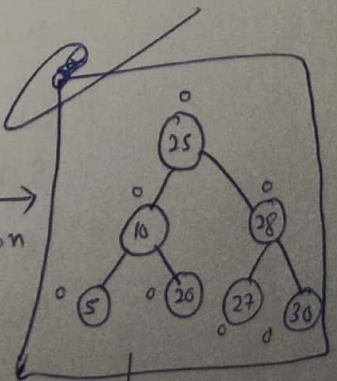
RL Rotation



Insert 5



LL Rotation



Smallest Height Possible

★ Deletion from AVL Tree With Rotations

"Deletion is same as in Binary Search tree."

★ Code for LL, L-R Rotation : RR & RL are same with little change

```
Struct Node
{
   Struct Node * lchild;
   int data;
   int height;
   Struct Node * rchild;
} * root = NULL;

int main ()
{
   // for LL Rotation
      root = rinsert (root, 10);
      rinsert (root, 5);
      rinsert (root, 2);
   // We can see in debug area that (5) will become root and (2) its
      lchild and (10) its rchild
   // For LR Rotation
      root = rinsert (root, 50);
      rinsert (root, 10);
      rinsert (root, 20);
   // We can see in debug area that (20) will become root and 6 its
      lchild and 50 its rchild
}
```

```
int NodeHeight (struct Node * P)
{
    int hl, hr;
    hl = P && P->lchild ? P->lchild -> height : 0;
    hr = P && P->rchild ? P->rchild -> height : 0;
    return hl > hr ? hl+1 : hr+1;
}

int balancefactor (struct Node *P)
{
    int hl, hr;
    hl = P && P->lchild ? P-> lchild -> height : 0;
    hr = P && P-> rchild ? P->rchild -> height : 0;
    return hl - hr;
}

struct Node * LLRotation (struct Node *P)
{
    struct Node *Pl = P->lchild;
    struct Node *Plr = Pl -> rchild;
    Pl -> rchild = P;
    P-> lchild = Plr;
    P-> height = NodeHeight (P);
    Pl-> height = Node Height (Pl);
    if (root == P)
        root = Pl;
    return Pl;
}
```

```
Struct Node *LRRotation (struct Node *P)
{
    Struct Node *Pl = P->lchild;
    Struct Node *PlR = Pl->Rchild;
    Pl->Rchild = PlR->lchild;
    P->lchild = PlR->Rchild;
    PlR->Rchild = P;
    PlR->lchild = Pl;
    P->height = NodeHeight(P);
    Pl->height = NodeHeight(Pl);
    PlR->height = NodeHeight(PlR);
    if (Root == P)
        Root = PlR;
    Return PlR;
}

Struct Node *Rinsert (struct Node *P, int key)
{
    Struct Node *t = NULL;
    if (P == NULL)
    {
        t = (struct Node *) malloc (sizeof (struct Node));
        t->data = key;
        t->height = 1;
        t->lchild = t->rchild = NULL;
        return t;
    }
```

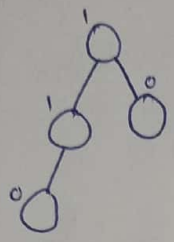
```
if (key < P→data)
    P→lchild = AVLInsert(P→lchild, key);
else if (key > P→data)
    P→rchild = AVLInsert(P→rchild, key);
P→height = NodeHeight(P); // Updating Height of every node
if (balancefactor(P) == 2 && balancefactor(P→lchild) == 1)
    return LLRotation(P);
if (balancefactor(P) == 2 && balancefactor(P→lchild) == -1)
    return LRRotation(P);
return P;
}
```

★ Height Analysis of AVL Trees :- Height Vs. Nodes of Trees

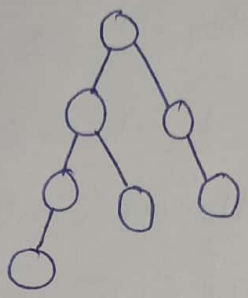
h = 1, n = 1

h = 2, n = 2

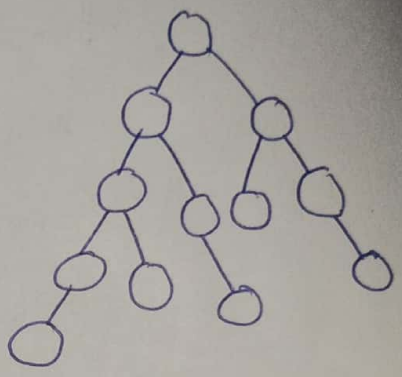
h = 3, n = 4



h = 4, n = 7



h = 5, n = 12



h → height of tree (starting from 1)

n → minimum no. of nodes required

h	1	2	3	4	5	6	7
N(h)	1	2	4	7	12	20	33

$$N(h) = \begin{cases} 0 & 0 \\ 1 & 1 \\ \text{otherwise} & N(h-2) + N(h-1) + 1 \end{cases}$$

This formula is same as formula for fibonacci series and fibbonaci series is famous for its balanced series i.e. $f_{i+1}/f_i = 1.6$, so this proves AVL Trees are balanced

N(h●) → for minimum no. of Nodes

As we know, it is also a full / complete Binary tree so maximum no. of Nodes :- $2^{h●} - 1$ [but in Binary tree we take from (height) 0 but here we take height from 1]

if 'N' nodes are given :- Minimum height :- $\log_2(n+1)$ → logarithmic

Maximum height :- Look in the table and if n = 13, then h = 5 [for 12-19]

General value from table ← $h = 1.44 \log_2(n+2)$ → logarithmic

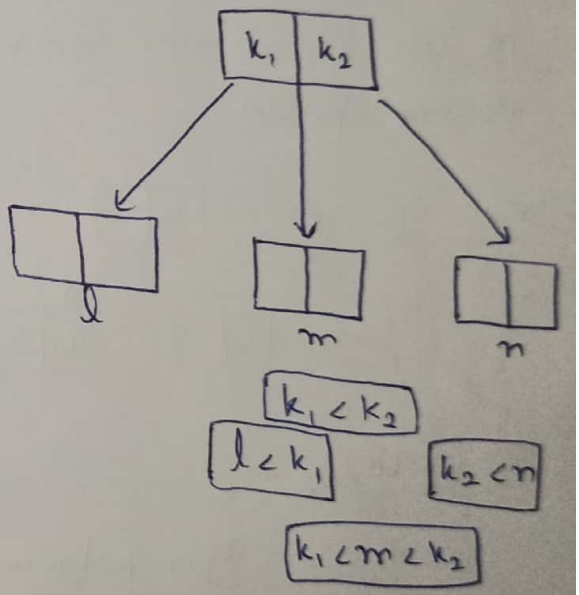
Some Important Trees

★ 2-3 Trees :-

These are search trees like BST but as we know in BST there is one node having two children but in 2-3 trees, there is a pair of node having 3 childrens, thats why they are known as multiway search tree / m-way tree having Degree 3.

- 2-3 trees are height balanced trees thats why known as B-Trees. B-Trees are generally used for height.
- All Leaft Nodes at Same level
- Every node must have $\lceil \frac{n}{2} \rceil = \lceil \frac{3}{2} \rceil = 2$ children

n → degree

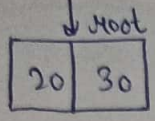


- As it is a search tree, so there will not be duplicates elements present.

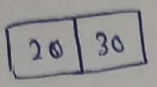
⇒ Creation :- keys :- 20, 30, 40, 50, 60, 10, 15, 70, 80, 90

Insert 10

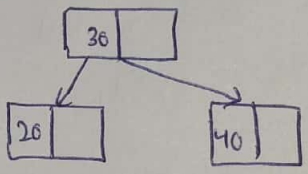
Insert 20, 30



Insert 40 → very imp. step

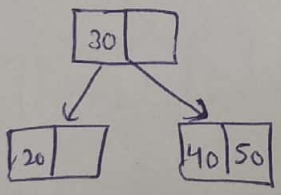


Now, Here we can see that there is no space left, it means degree is 3 but keys are 2 only, then what to do, split the node. one key on left side & one key on right side and one will go upward



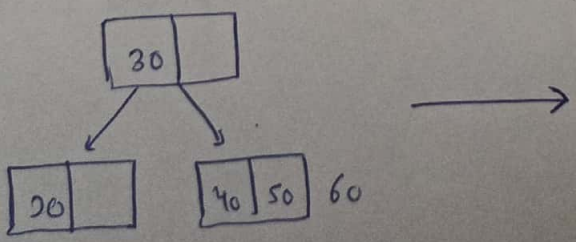
As we see binary search tree grows downward, But 2-3 trees are growing upwards.

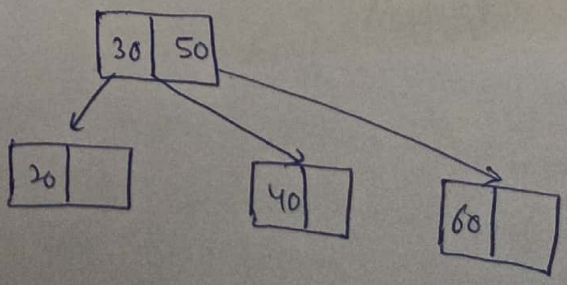
Insert 50 ⇒



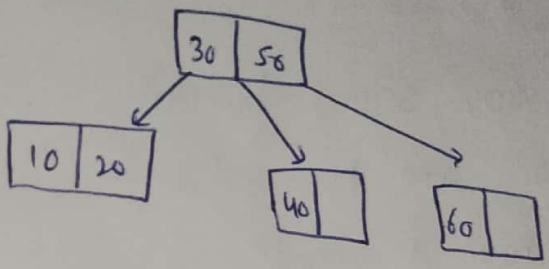
Search for 50, it is not there and > 30 and > 40, so it will inserted after 40 as there is a vacant space available.

Insert 60 ⇒ 60 is > 30, > 40, > 50, but there is no space, so again perform split.

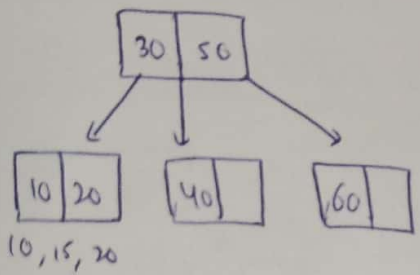


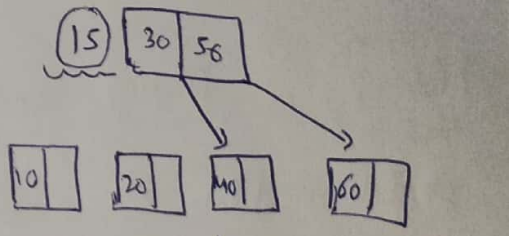


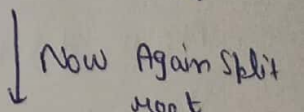
Insert 10 :- it is less than 30 & 20, and there is vacant space also, so shift 20 and insert it here



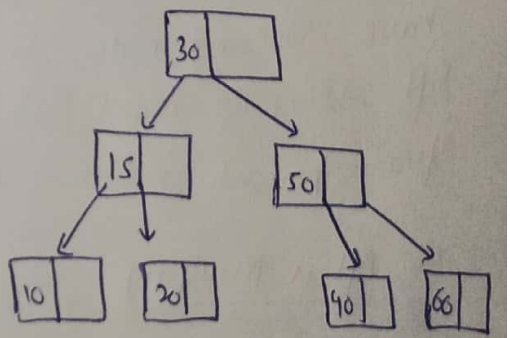
- Insert 15 :- it is less than 30, greather than 10, but < 20, so



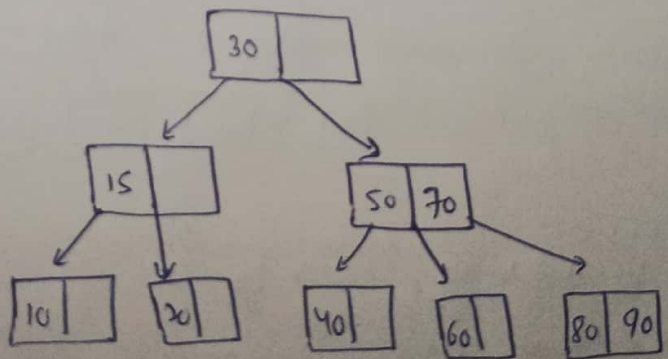




"So insertion of 15 caused 2 splits"

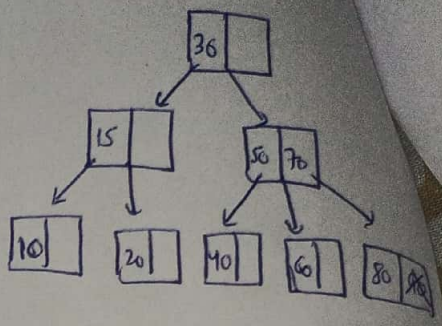


- Insert 70, 80, 90 :-



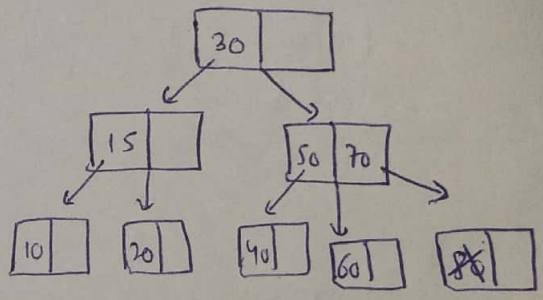
⇒ **Deletion** :-

Case-1 : Simply Delete



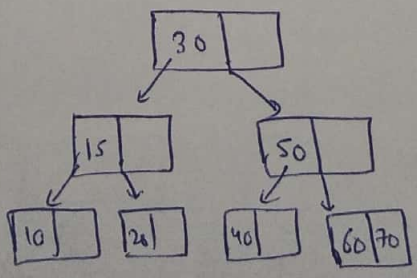
✯ Let us delete 90, Search for 90
Yes we found 90 and than delete it
No changes we have to made.
Simply we can say if the Node is a leaf node, we don't have to make any changes.

Case-2 :- Merge and Delete

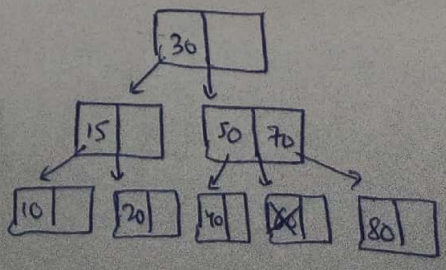


✯ Let us delete 80, Search for 80
Now, delete 80, and we get a vacant Node now, we should not have vacant node, so merge it with left Sibling or right Sibling, so, Remove this node and bring key (70) from parent which is with the Connection.

So, After Merging ⟹

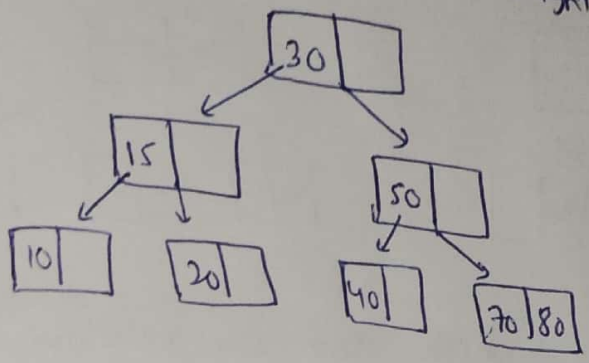


✯ Now, Let us delete 60, Search for 60
Now delete it, Now we can merge it with Left/right Sibling.
Lets first with right Sibling,



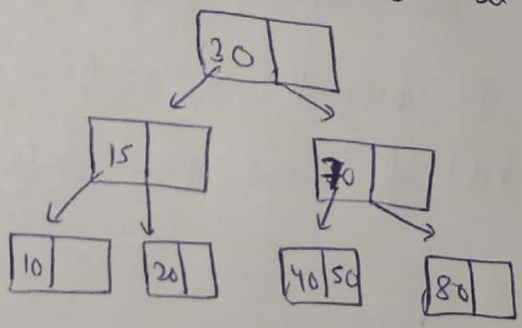
So, After merging with right sibling :- bring 70 from and parent and 80 from right sibling for merge and delete right sibling node.

1)

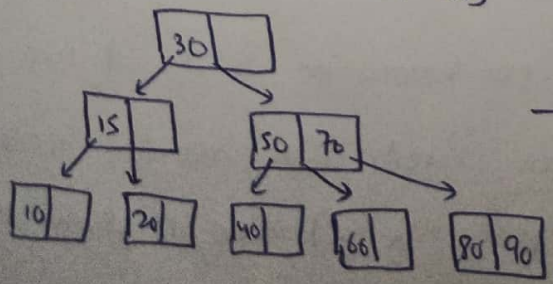


Now, Lets merge with left sibling :- Now deleting 60, So (50) should come down and bring/shift 70 and right sibling will become [middle].

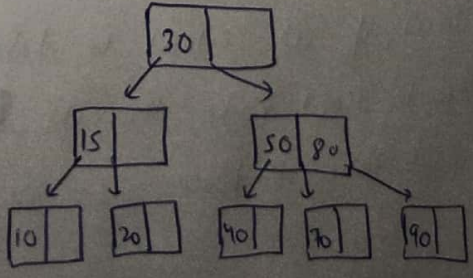
2)



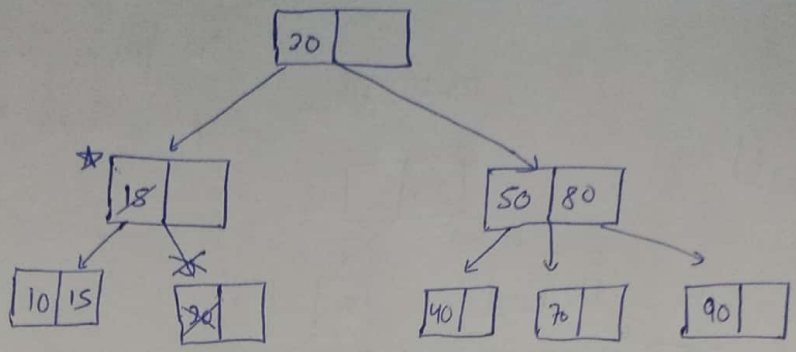
Case - 3 :- Borrow a key :- Let us delete 60 from here, So can we borrow a key from sibling, left sibling has only one key So, we can't borrow from here, but we can borrow from right sibling via parent and bring one key to the parent position and shift the another key



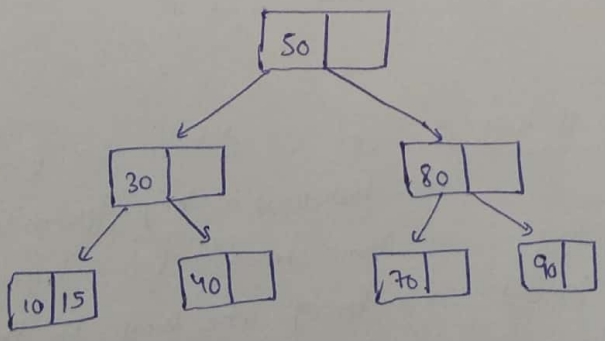
After borrowing →



Important Case :- Let us delete 20, from this tree

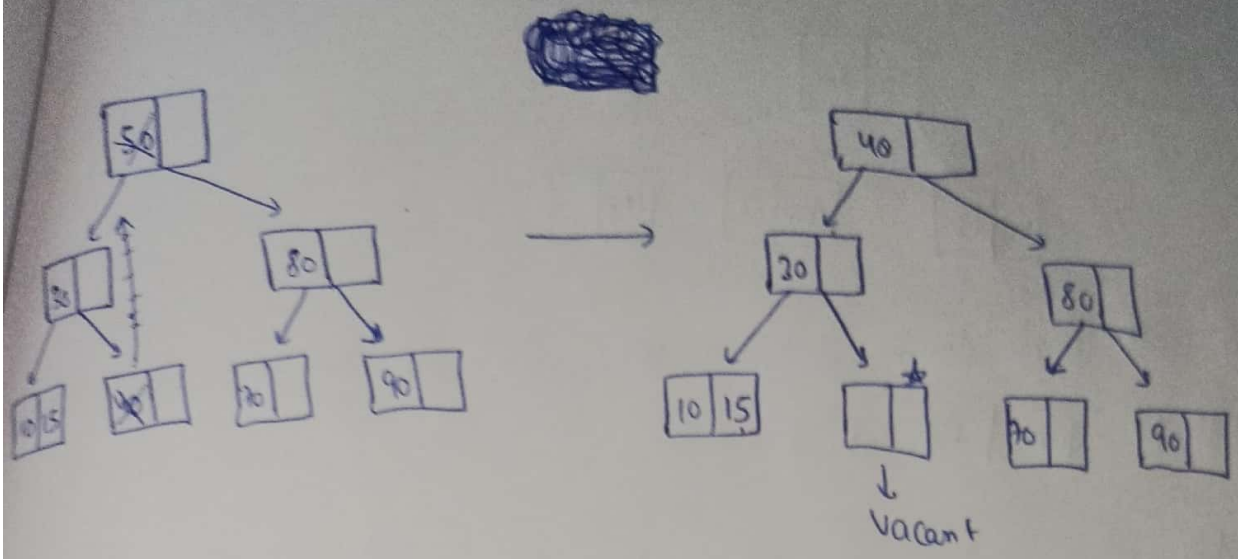


Now, when we delete 20, check can we borrow from left / right Sibling, Here we can't so, merge them, so After merging, we get 15 down but then that node (*) will be vacant, so again check can we borrow, yes we can borrow from right Sibling, so after borrowing, 50 will go upwards, 30 will get down & child of 50 will become middle child of 30 and then shift 80 and make it Connections again.



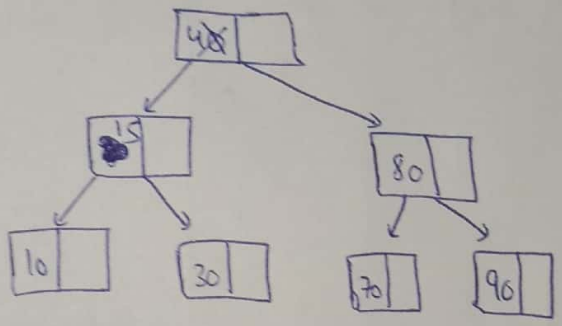
Now, if we want to delete (50), then we know in BST, if we delete root node, then its [redacted] inorder Predecessor / inorder Successor will take its place, so either (40) takes it place or (70) take its place.

Let us Consider that (40) will take its place.

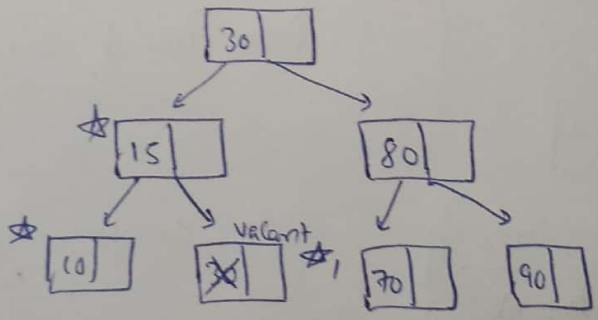


⇒ So, when ★ will become vacant, so borrow from left child

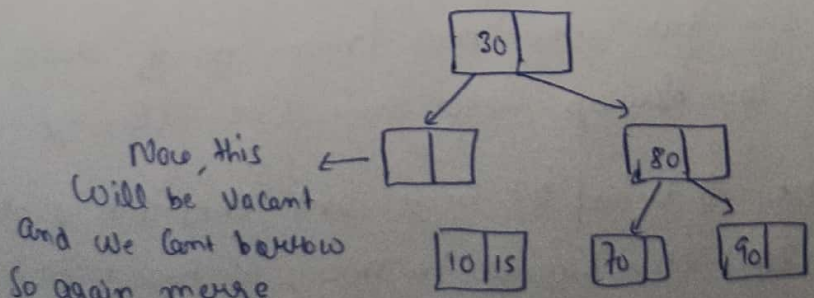
After Borrowing :-



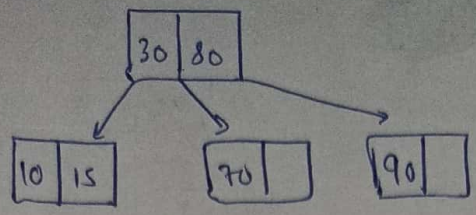
⇒ Now, Lets delete Delete 40, so (30) will take its place



Now, merge ★ & ★, to get

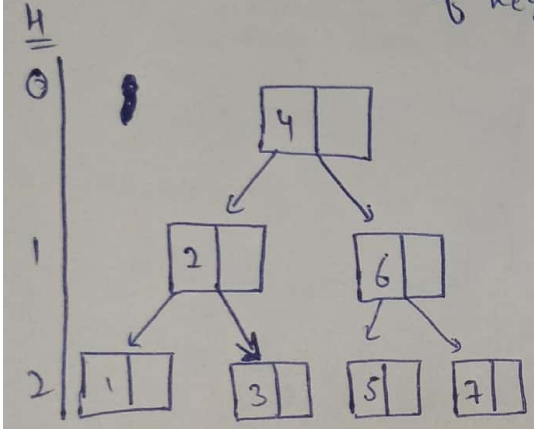


Now, this Will be vacant and we cant borrow so again merge



Here, 30 will come down, & 80 will come with 30 and remaining nodes will be gone / deleted.

⇒ **Analysis** :- Let us take two trees, one with min. no. of keys & min. no. of nodes ; Another with max. no. of keys & max. no. of nodes.

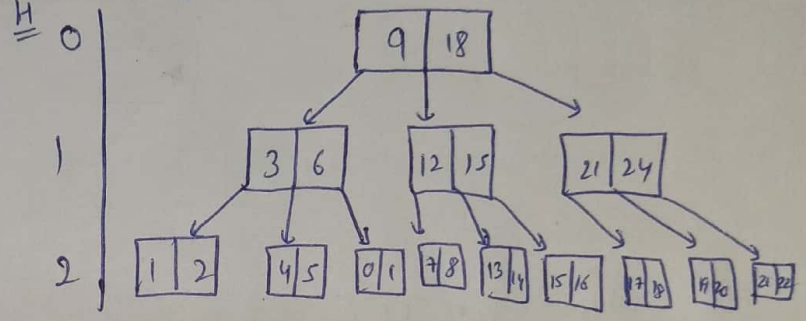


Min. no. of nodes, $n = 1 + 2 + 2^2 + \cdots + 2^h$

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

Max. Height $\Rightarrow$ $h = \log_2(n+1) - 1$

$O(\log_2 n)$



Max. No. of Nodes, $N = 1 + 3 + 3^2 + \cdots + 3^h$

$$N = \frac{3^{h+1} - 1}{2}$$

Min. Height $\Rightarrow$ $h = \log_3[2N+1] - 1$

$O(\log_3 N)$

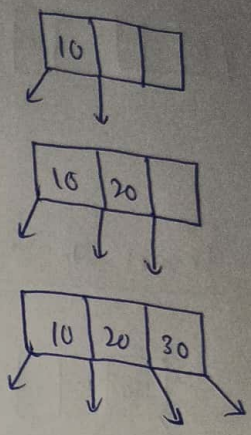
Note :- ① We can see that either max. / min. no. of Nodes, Height of 2-3 trees are always logarithmic (i.e. $O(\log_3 N)$)

Imp ② Actually these are B trees / B^+ trees and they are used in DBMS like oracle, mySQL, etc. because in this, we have to keys in the same node, so we can lastly direct access of more values in a less time as compared to BST (as in BST, one node has only one value)

2-3-4 Trees :-

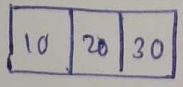
- B- Tree of degree = 4
- Every Node must have $\lceil \frac{4}{2} \rceil \rightarrow$ ceil $= 2$ children
- All leaf nodes are at same level

d = 4

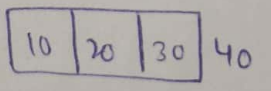


⇒ Creation :- keys → 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110

Insert 10, 20, 30

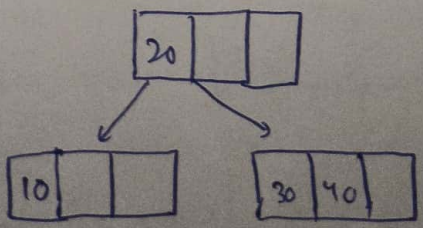


Insert 40



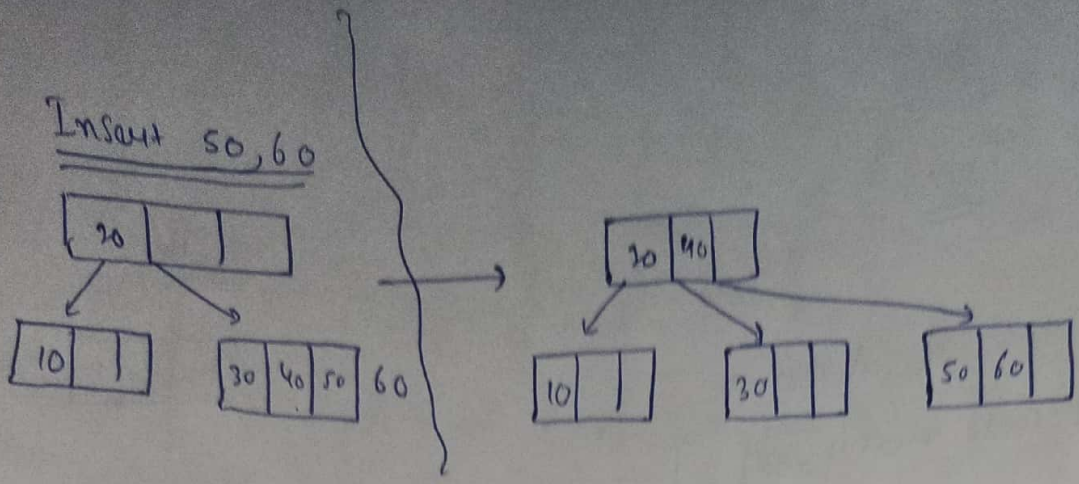
→ Perform splitting

Now, we can see that, there are even no. of keys during splitting which is different from 2-3 trees, and from those one of the node will be at root, so during splitting it may be Left biased / Right biased, i.e. Left nodes can more elements or else right node

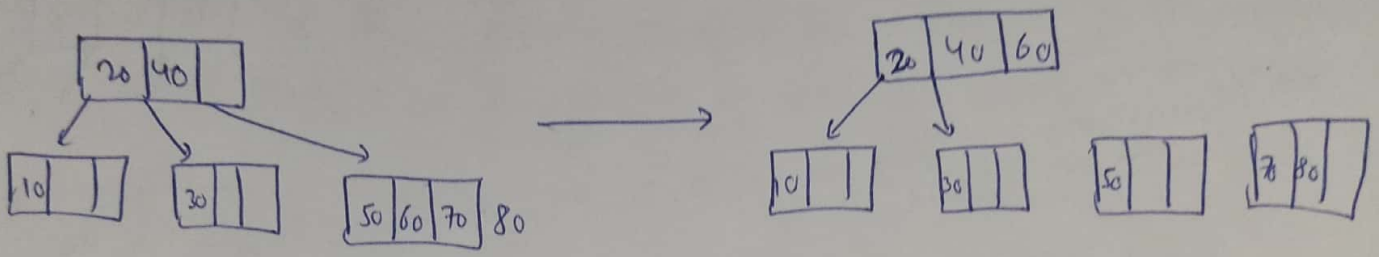


→ Right Biased

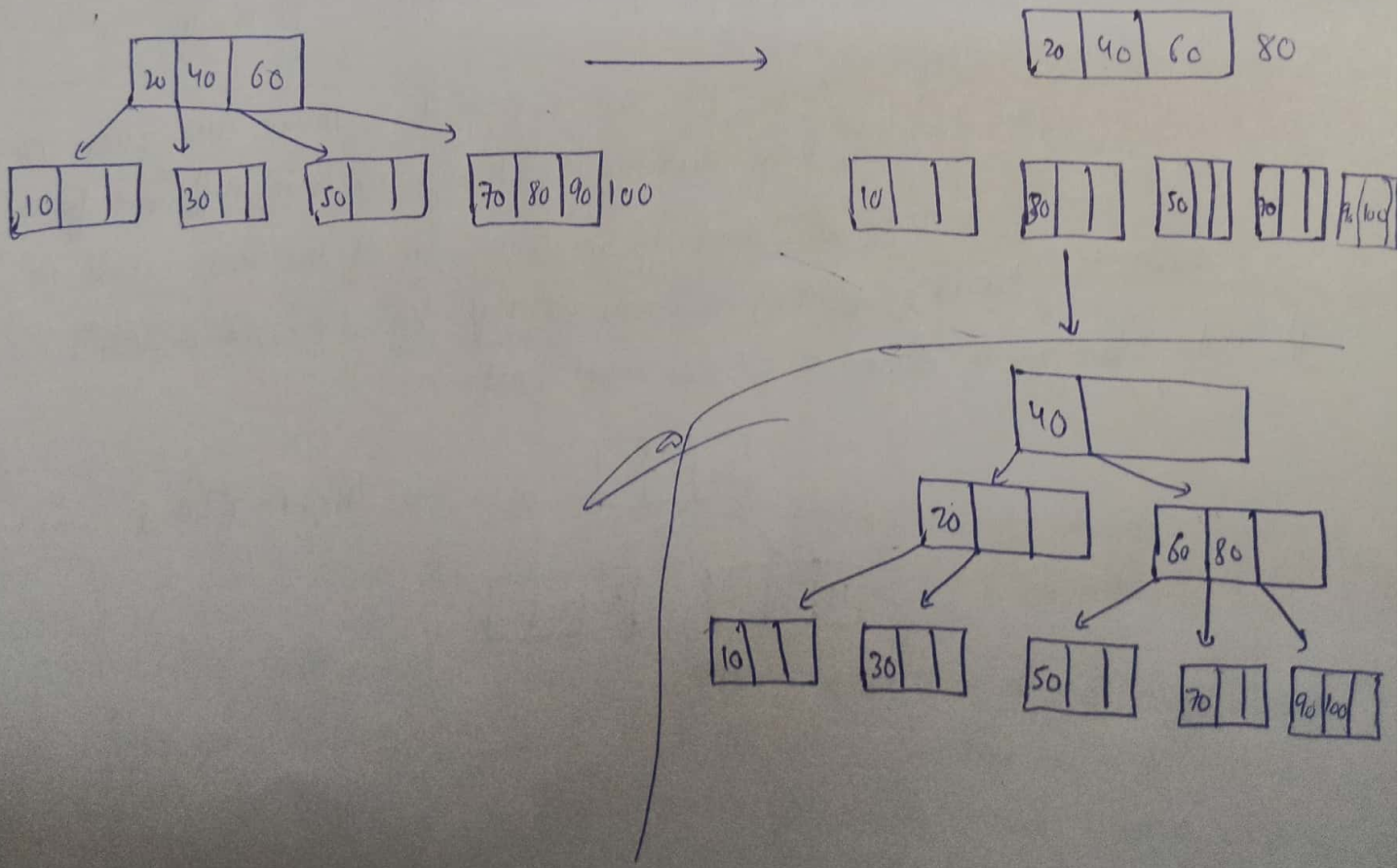
Insert 50, 60



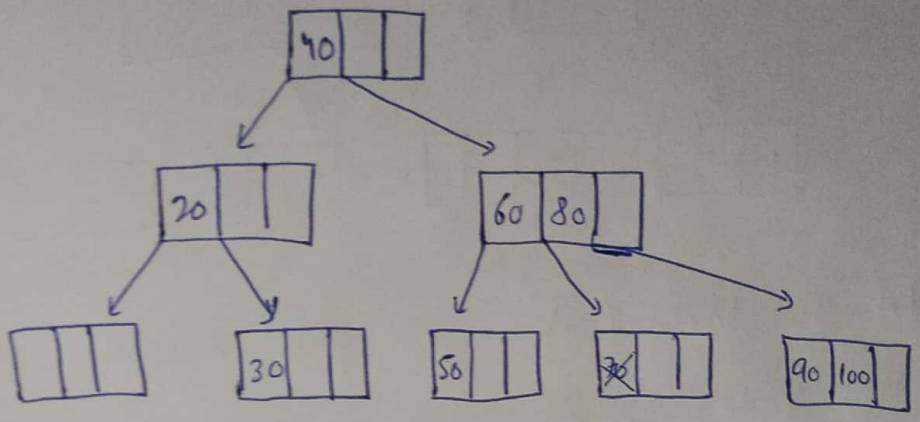
Insert 70, 80



Insert 90, 100

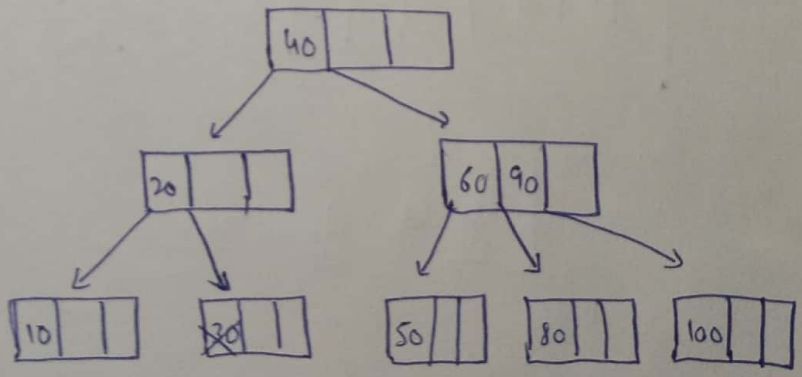


Deletion :- key → 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110

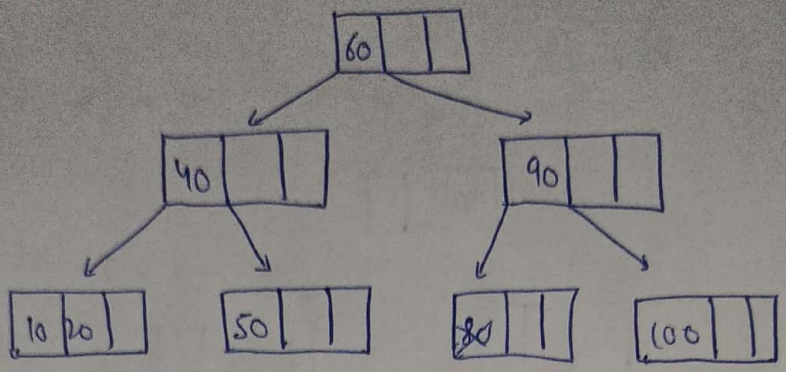


Case 1 :- If we want to delete 100, simply delete it, no changes are required

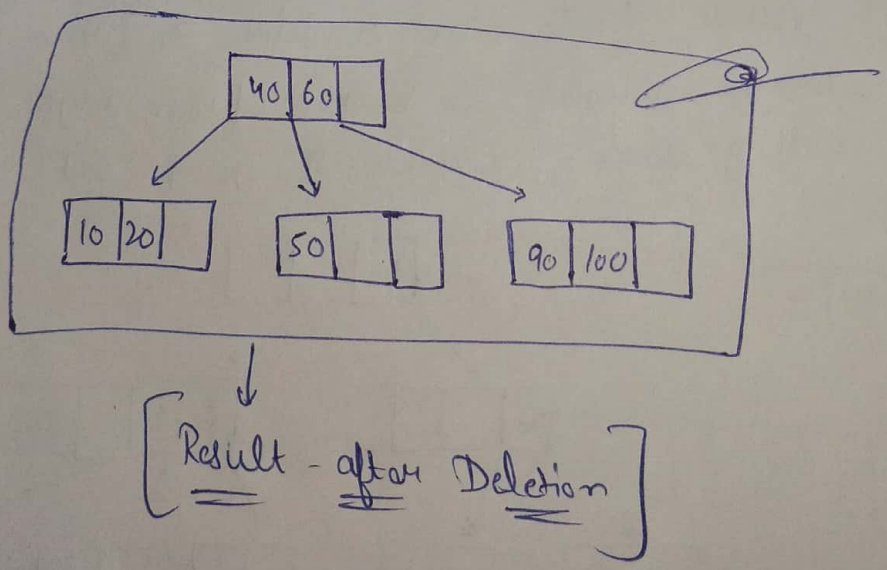
Case 2 :- Now, if we want to delete 70, then the node will become vacant, so borrow from right child, then (80) will go down, & (90) will go up and (100) will be shifted.



Now, let us delete 30, that node will become vacant, so merge with (10), then upper node will becomes vaccant, so try to borrow, so we can borrow from right child, so (40) comes down, (60) goes up and left child of (60) will be right child of (40) and (90) will be shifted and its children will also shifted.



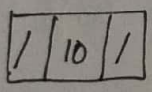
Now, Let us delete (80), So, again merging and (90 & 100) will be on that node, So upper node will be vacant & we cant borrow from left Sibling, So merge again & join (40 & 60), So remaining nodes will be gone / deleted.

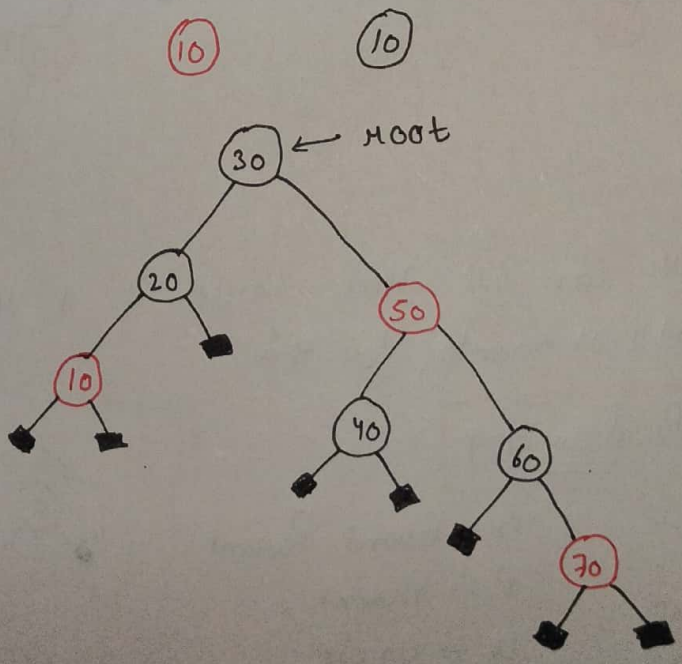


[Result - after Deletion]

☆ Red - Black Trees :- Properties

1. It is a Height Balanced Binary Search tree, Similar to 2-3-4 Tree.
2. Every Node is Either Red or Black.
3. Root of a tree is Black
4. NULL is also Black
5. No. of Blacks on paths from Root to leaf are Same
6. No 2 Consecutive red Nodes, Parent and children of Red or Black.
7. New Inserted Node is Red
8. Height is $\log n \leq h \leq 2(\log n)$





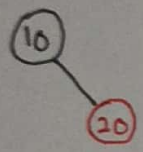
⇒ Creation :- keys :- 10, 20, 30, 50, 40, 60, 70, 80, 4, 8

Note :- Insertion is done just like Binary Search tree.

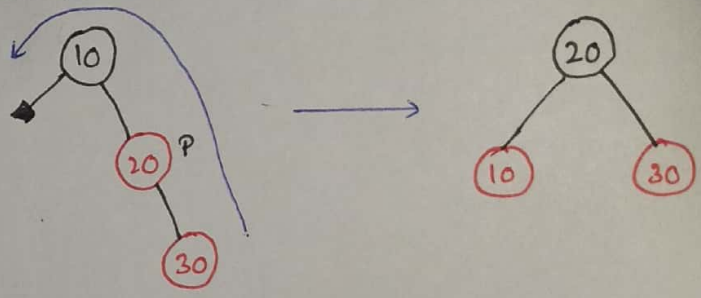
- Insert 10

as we know, this is a newly inserted Node, So its Colour is Red, but its root node also, So we'll change it to black

- Insert 20

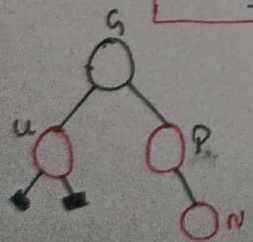


- Insert 30



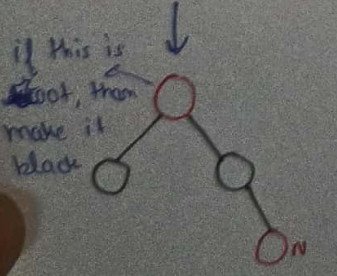
★ So, Here we can See that there is a Red-Red Conflict, So there are two adjustments for this.

Recolouring

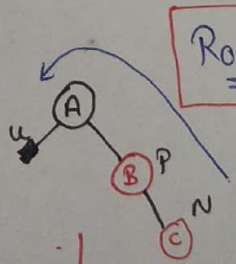


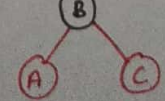
G → Grand Parent
P → Parent
u → uncle
N → New inserted node

Now, if parent is Red & uncle is also Red, then do Recolouring by changing the colours of parent, uncle & Grand parent.

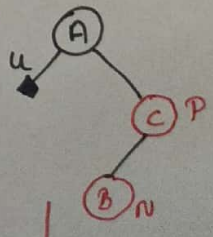


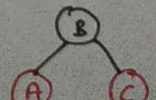
Rotation





Zig-Zig (LL/RR)

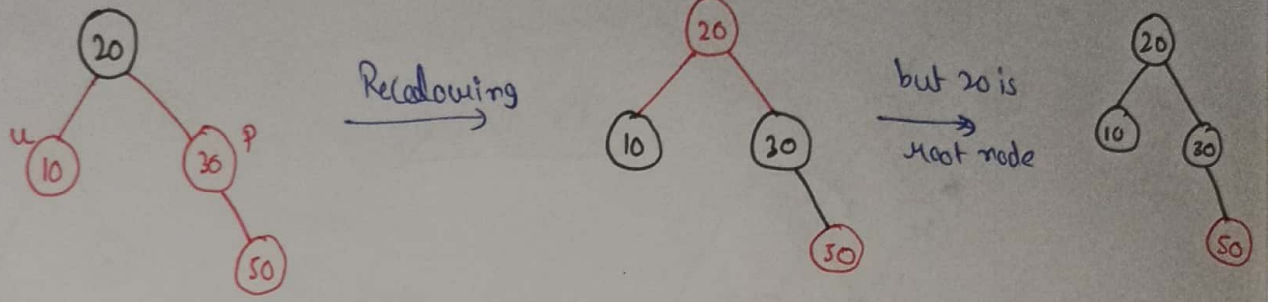




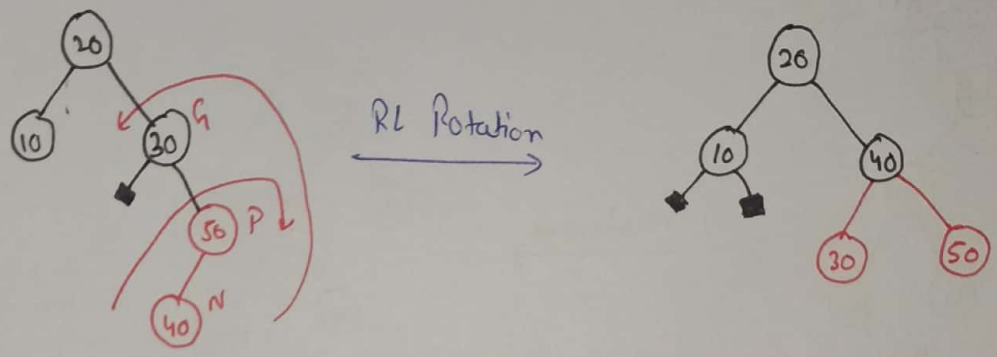
Zig-Zag (LR/RL)

If Uncle is black & parent is Red then perform Rotations as we do in AVL Trees.

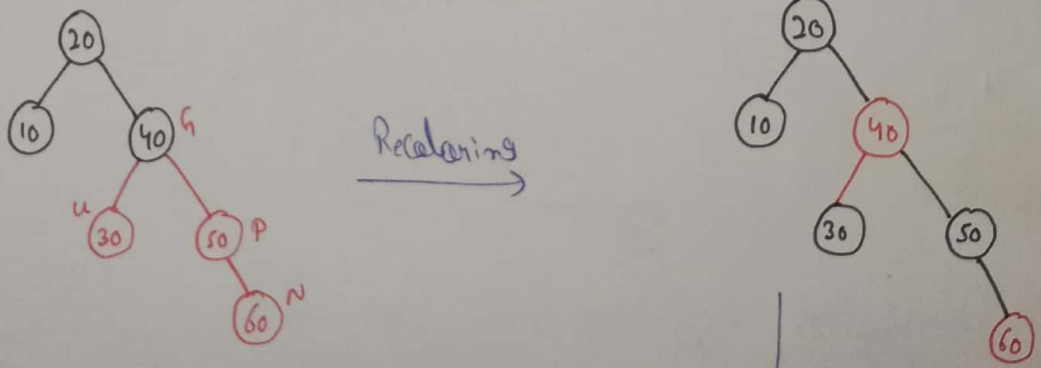
- Insert 50



- Insert 40

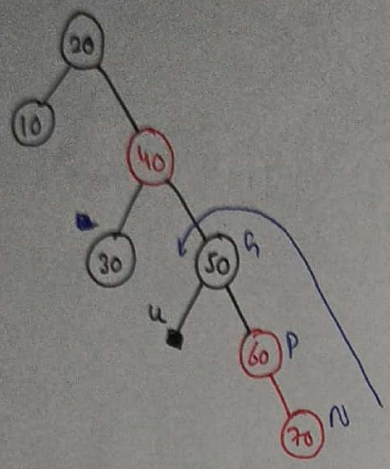


- Insert 60

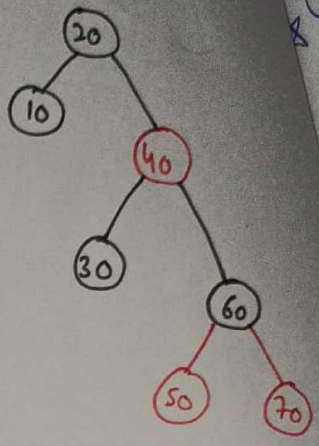


Imp
At this step, we have to check for (40), that is it having a red conflict with its parent / not, Here is no conflict, so move forward

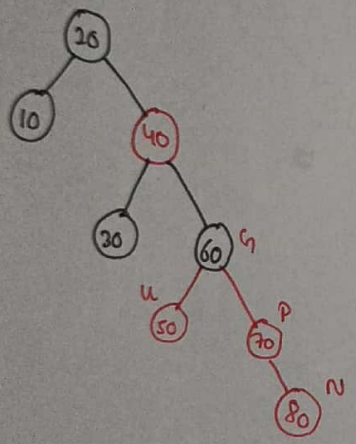
- Insert 70



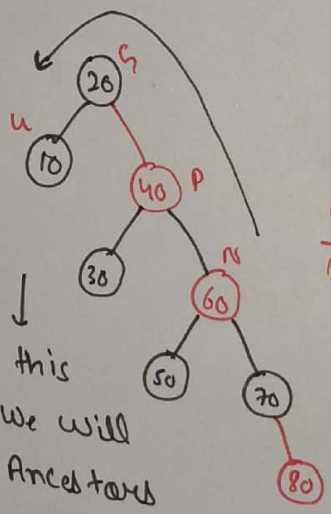
RR Rotation →



- Insert 80

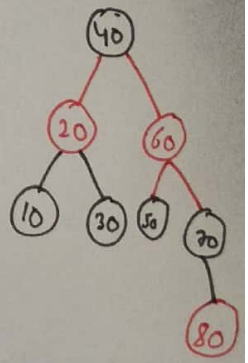


Recoloring →

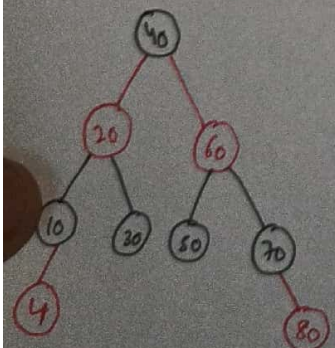


At this Step, we will Check Ancestors

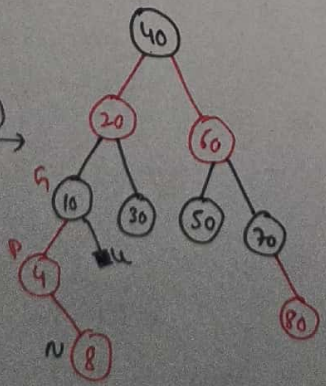
RR Rotation →



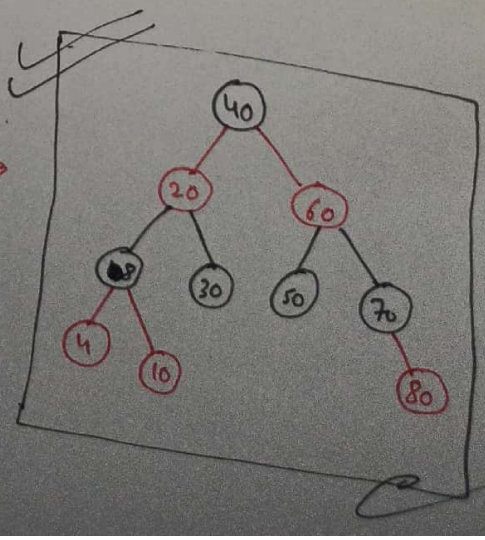
- Insert 4, 8



Insert(8) →



LR Rotation →



Red Black Trees Vs. 2-3-4 Trees :-

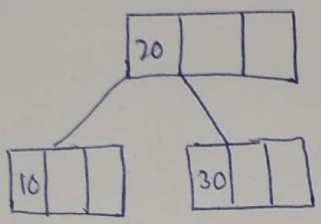
Key :- 10, 20, 30, 50, 40, 60, 70, 80, 4, 8

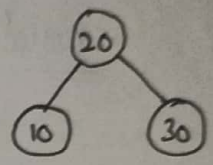
Degree

2-3-4 Tree

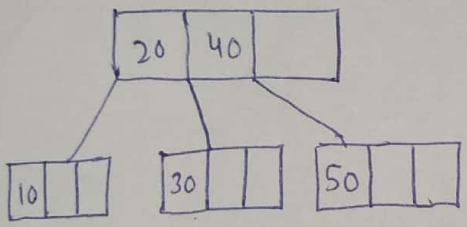
Red-Black tree

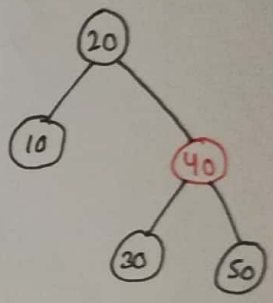
(i)



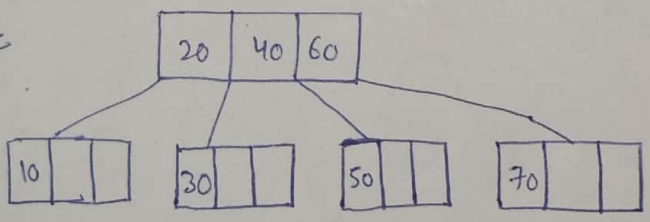


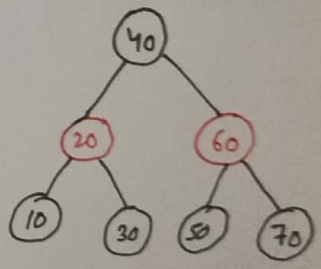
(ii)





(iii)





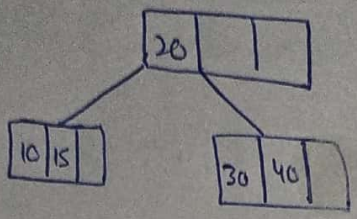
Relation for (i) :- There are three nodes on both trees with one key.

Relation for (ii) :- 1st node is having two keys 20 & 40, but in R-B tree 1st node has just one key but having two keys done is red, it means this (40) is with (20) in single node as in 2-3-4.

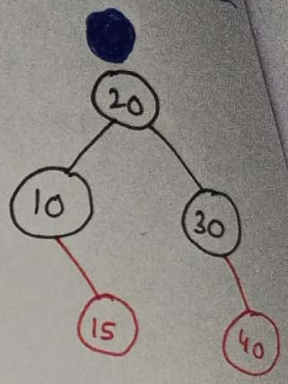
Relation for (iii) :- Single node is having 3 keys and middle key is black Node and (20) & (60) are in red color as they contain with 40 in a single node, so Red Color Nodes are related to its black Colour parent Node.

One more Example :-

2-3-4 Tree

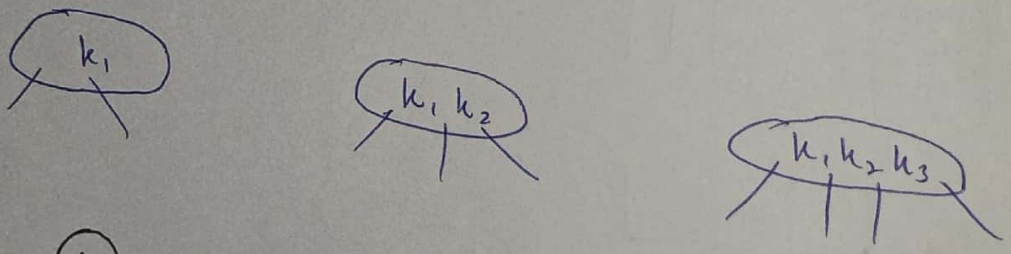


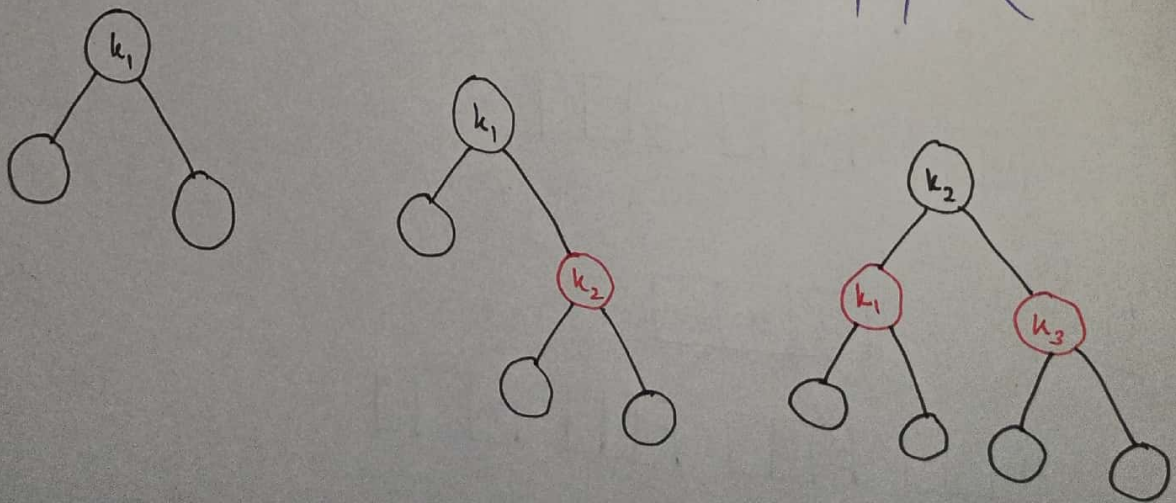
Red-black Tree



Red

⇒ General form :-

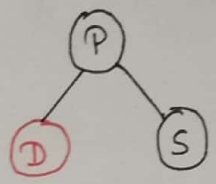




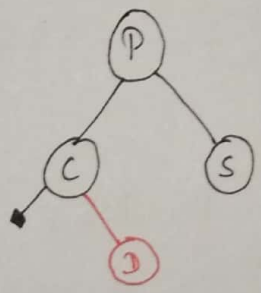
Red - Black Tree Deletion Cases :-

Deletion in Red - Black tree is similar to that of Deletion in Binary Search Trees (so, read them first).

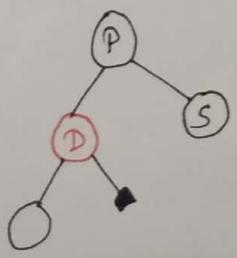
Case 1 :-



Let us Delete a Node (D), Now, it is a Red node having no child, so Simply delete it without make any changes.



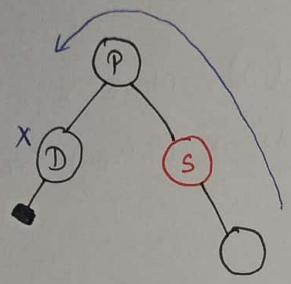
Now, Here if we want to Delete node (D), then again do Nothing.



Now, Here if we wanto Delete node (D), and it is having a one child and definetly it will black, so, bring its child there on its place and definitely it will black.

Note :- We are saying that if a node is red, then simply deleting it blindly without any changes, bcz we know that there is condition that no. of blacks from the root to leaf path must be same, So we don't have to bother about red nodes. thats why we said that if a node is red, then simply delete it.

Case-2 :-

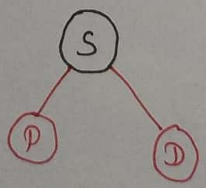


P → parent of D

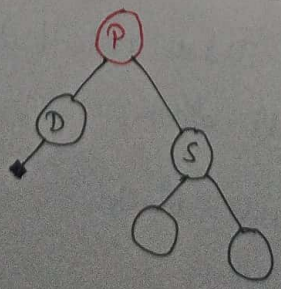
D → Node getting deleted

S → Sibling of Node which is getting deleted

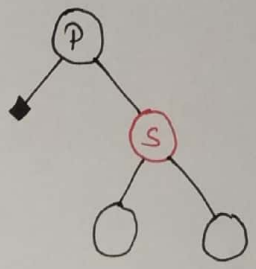
now, In this case if we see that, the Node which have to be Deleted is a black Node and its Sibling is red and parent is black, then do nothing but Simply delete that node and perform Zig-Zig Rotation.



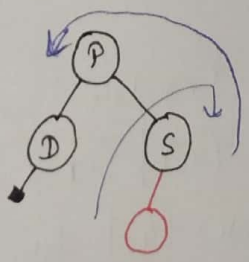
Case-3 :-



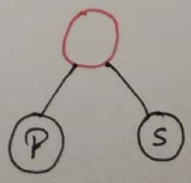
Now, Here, if we can see that the Deleted node is (D) and its sibling is (S) which is also black and it is having both the children as black, than do the recolouring, like, (S) will become red (i.e sibling) and make children of Sibling black and make parent of Sibling also black and the place of Deleted Node will be taken by its child (Here, child is NULL).



Case - 4 :-

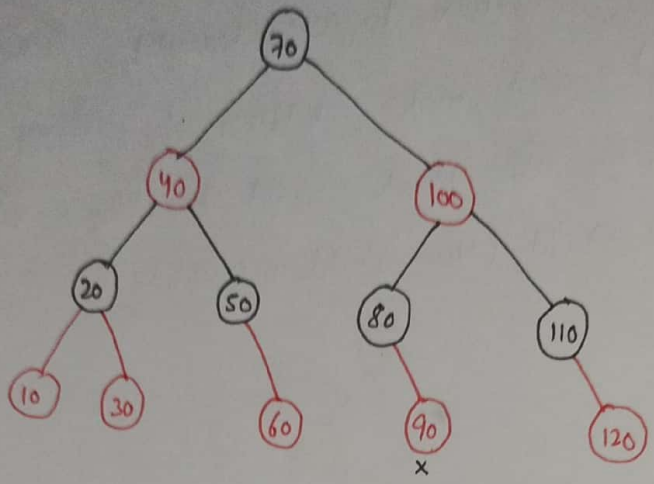


Now, Here if we can see that, the Deleted Node (D) is having a black Sibling (S) but its child is red, than perform Zig-Zag rotation one/two children doesn't matter, bcz both. will be red.

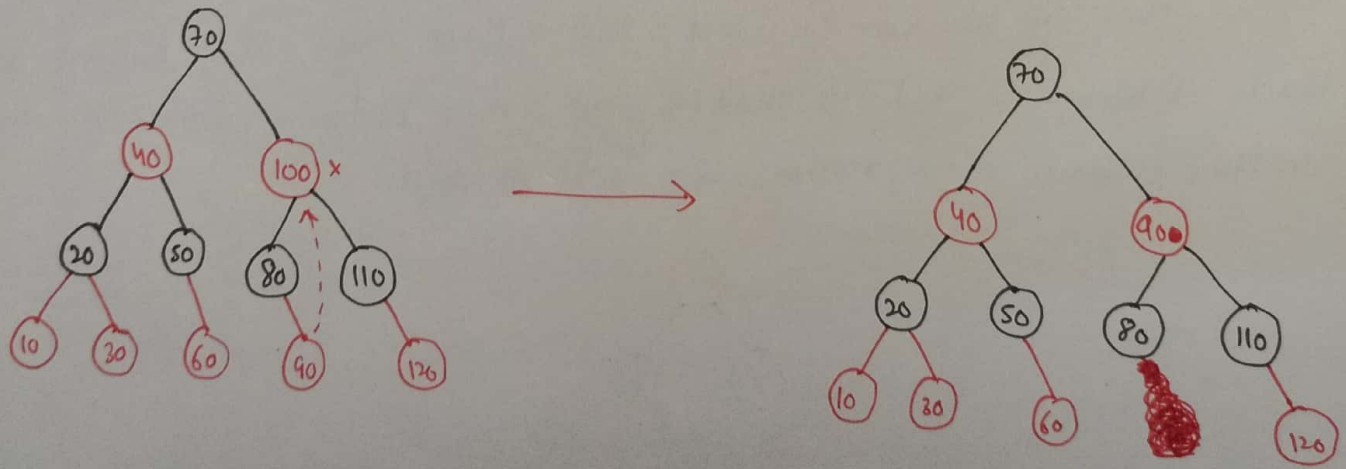


but, if (S) is having right child, than perform Zig-Zag rotation. and if (S) is having two children, than perform any one from Zig-Zig/ Zig-Zag, but more Preferable is Zig-Zig.

Example for Deletion :-

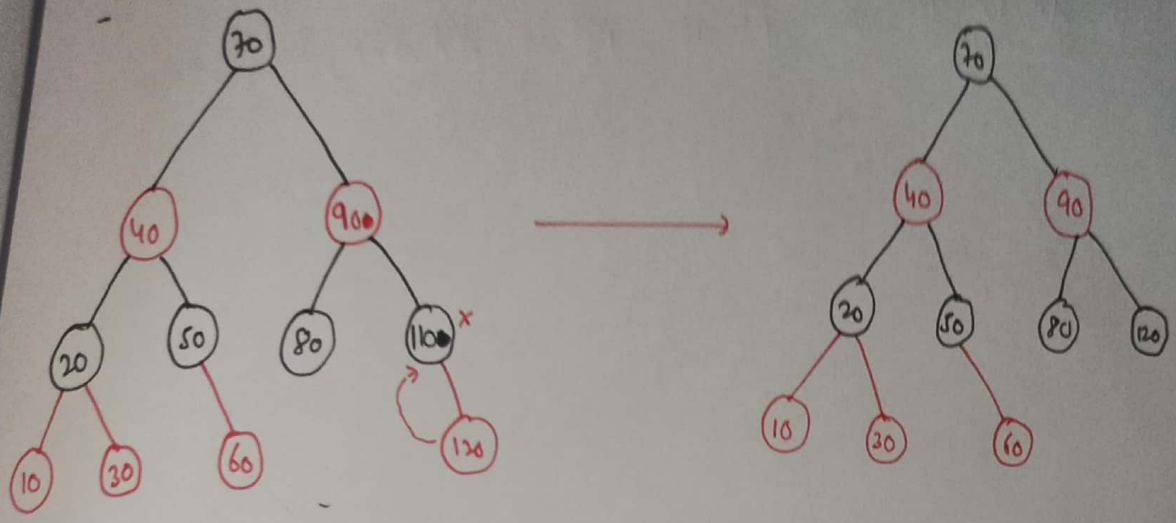


- **Now, Delete 90** :- Simply delete it, Without any change
- **Now, Delete 100** :- Find Inorder Predecessor | Inorder will take its place either (90) | (120)
  Lets take (90) take its place.

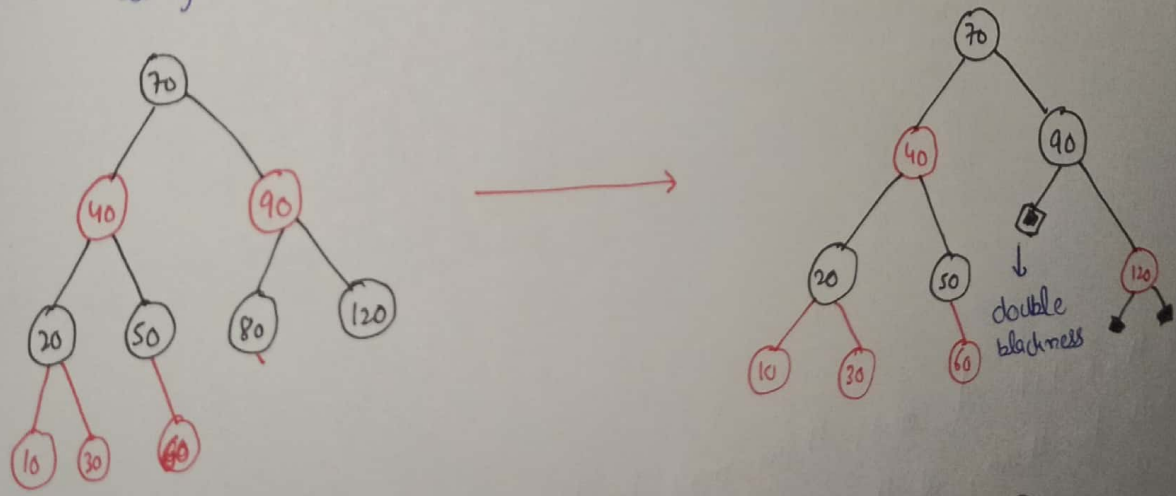


- **Now Delete 110** :- Simply delete 110, and 120 will take its place.

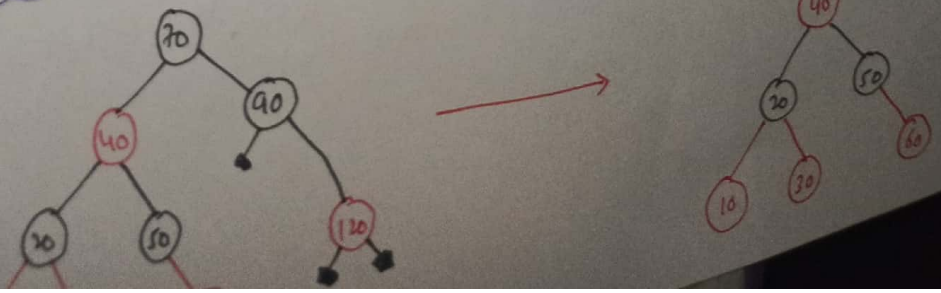
"This is all about deleting Red Nodes".



Now, Delete 80 :- Now, if we delete (80), it is a black color node having NULL child, so when we delete it, (90) left also will be NULL, than it will become double NUL which we called double Blackness. In these type of cases, check sibling Here Sibling is black, so do recoloring, bcz its child are also black (null)



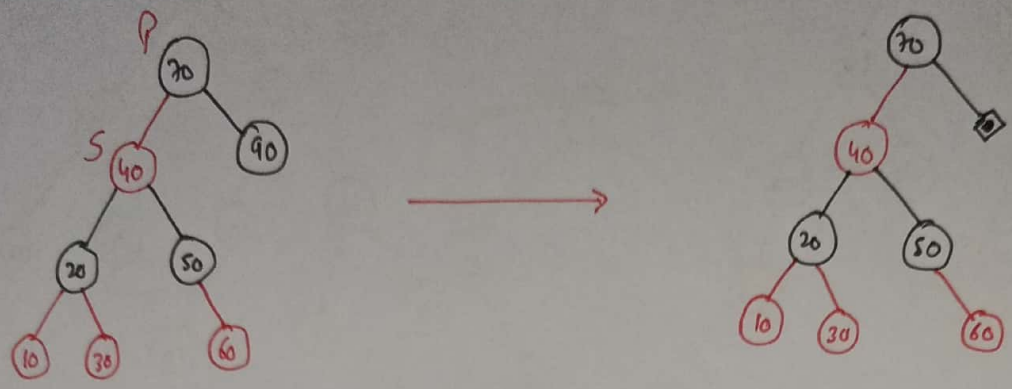
- Now, delete 120 :- Simply delete it.



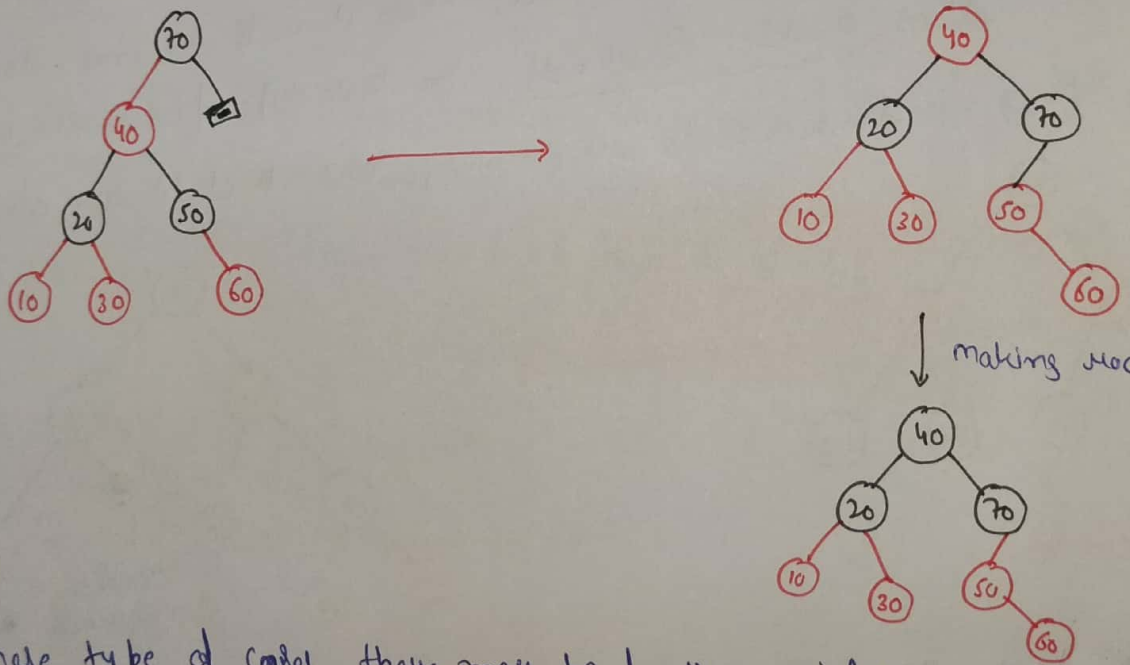
Imp :-

Special Case :-

Now, delete 90 :- Now, After deleting (90), tree will look like-



Now, checking its Sibling, Here its Sibling is (40) having Red Colour so, before Zig-Zig Rotation, and while performing Zig-Zig Rotation, where we shift the node (50), it will become Red Colour (i.e. Important thing here).



In these type of cases, there may be further violations, that we have to check for them, as there is a red-red conflict b/w (50) & (60), so perform Rotation. (Here, we have to perform Zig-Zag Rotation (i.e L-R Here).