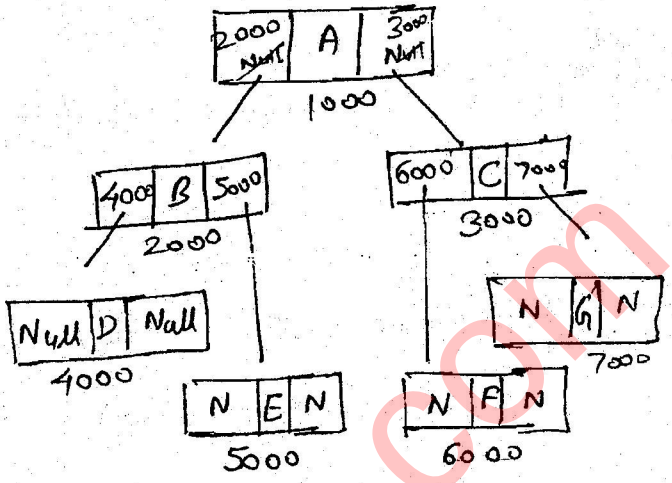
te 1/oct/2017

"Binary Tree" :-

```
Struct Btnode
{ Struct Btnode *lc;
  int data ;
  Struct Btnode *rc;
};
```



Binary Tree

Leftmost child :-

```
while (root -> lc != null)
    root = root -> lc;
```

Height of Tree :- Longest path from root to leaf node or (# of levels - 1)

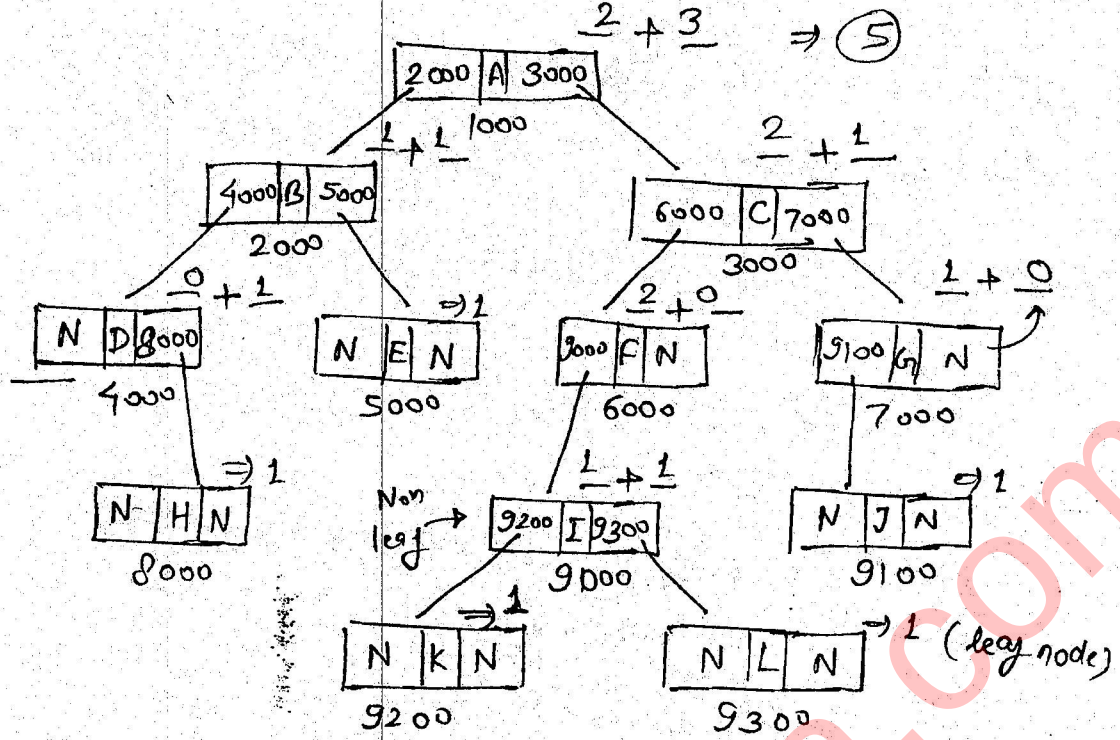
Que :- write a 'C' program to count total number of leaf nodes in the given binary Tree.

i/p :- root

leaf node :- lc = null && rc == null

```
Countleaf ( Struct Btnode *r)
{ 1. If (r==Null) return (0);
  2. If ( r -> lc == null && r -> rc == null)
        return (1);
  3. else
     {
       a = Countleaf (r -> lc);   => T(n/2)   T(n-1)
       b = Countleaf (r -> rc);   => T(n/2)   T(0)
       c = a+b; return (c);       => O(1)     O(1)
     }
}
```

B.C. W.C



Time Complexity (Counting leaf)

Best case :- $2T(n/2) + C$

$= O(n)$

worst case :- $= T(n-1) + C$

$\Rightarrow n \cdot C$

$\Rightarrow O(n)$

} so every case $O(n)$

Stack space :- Best case = $O(\log n)$

worst case = $O(n)$ [are sided tree]

If we take $C = 9$.

Que :- write a 'C' program to count No. of Nonleaf Nodes in the given Binary Tree.

```
Count Nonleaf ( struct Btnode *r)
{
1. If ( r == null)
       return (0);
2. If ( r -> lc == null && r -> rc == null)
       return (0);
3. else
   {
       a = Count Nonleaf (r -> lc);
       b = Count Nonleaf (r -> rc);
       c = a + b + 1;
       return (c);
   }
}
```

O/P :- 7

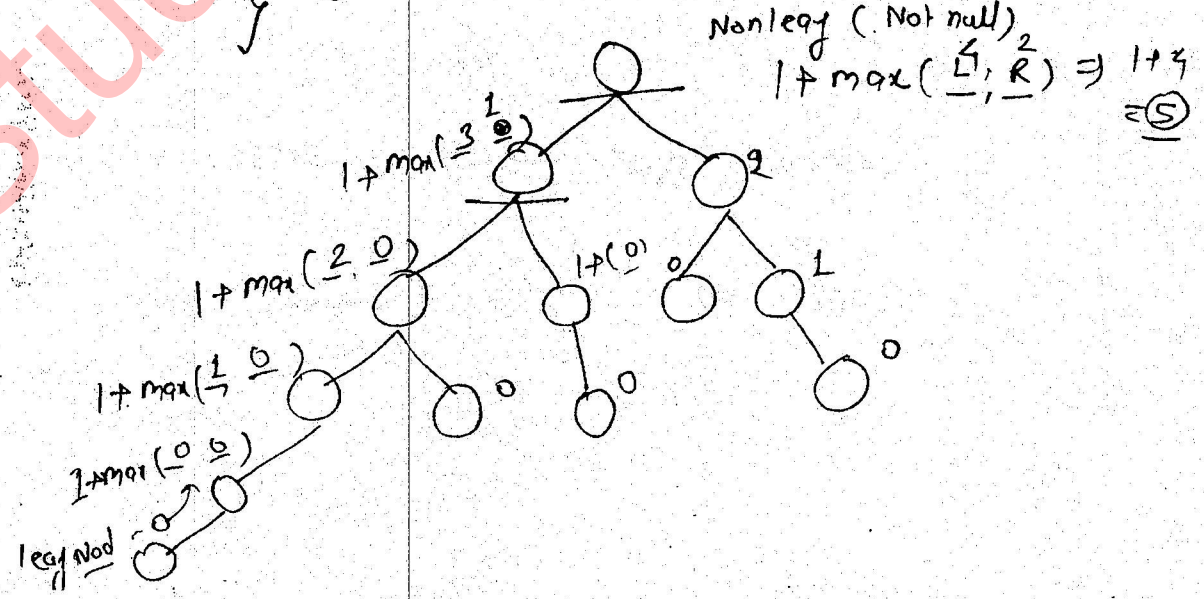
Total nodes :-

```
If ( r -> lc == null && r -> rc == null)
       return (1)        [leaf]
                         Both place count
c = a + b + 1            [Non leaves]
```

Que :- write a C program to find Height of the given Binary tree.

Height of A Node = The Longest distance from that Node to Any leaf Node.

```
height (struct Btnode *r)
{
    if ( r == null)
        return(0);
    if ( r->lc == null && r->rc == null)
        return (0);
    else
    {
    a = height ( r->rc)
    b = height ( r->lc)
    c = 1 + max (a, b);
    return (c);
    }
}
```



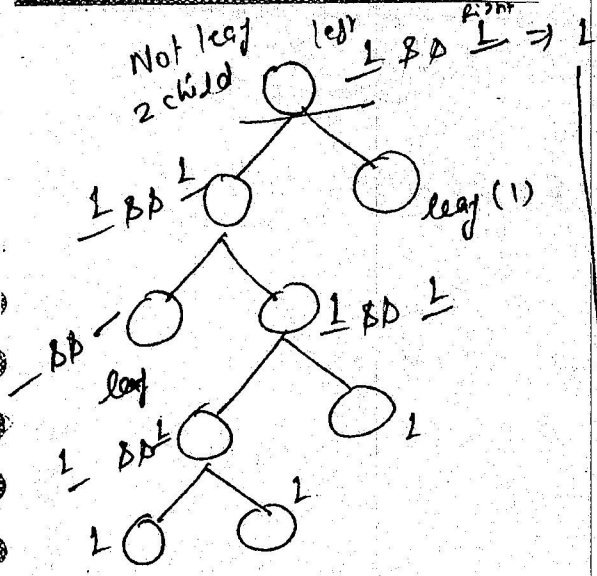
# of levels in tree :-

```
If ( r → rc == null && r → lc == null)
        return (1)
    else
    { same as height ;
    }
```

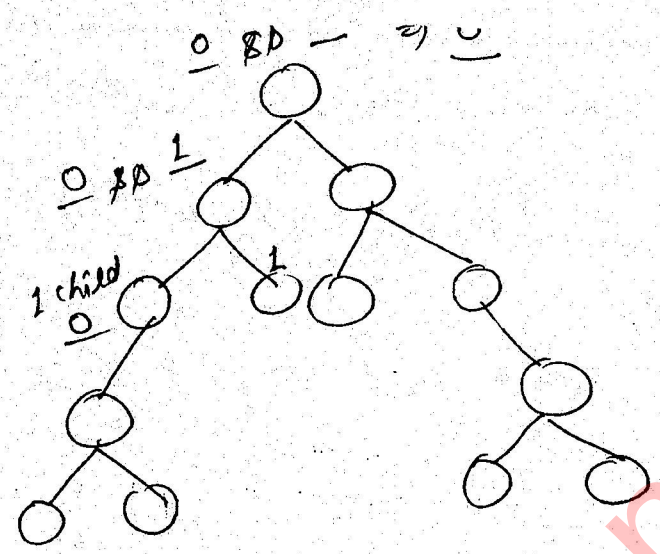
Que :- write a C Program to check given Binary Tree is strict Binary Tree or Not.

Strict Binary Tree :- 0 or 2 child
not 1

```
SBT (r)
{
1.  If (r == null)
        return (1) ;        /* empty Binary Tree also strict */
2.  If (r → lc == null && r → rc == null)
        return (1);         /* leaf Node 0 child */
3.  If ( r → lc != Null && r → rc != Null) /* 2 child */
        return ( SBT (r → lc) && SBT (r → rc)) /* go further */
4.      return (0);         /* Neither leaf Nor 2 child then return (0) */
}
```



Strict Binary Tree



∴ Not Strict Binary Tree

If 3. If (r→lc != null && r→rc != null) → check [may be here also !!]

return (SBT(r→lc) && SBT(r→rc))

Check in left SubTree Strict Binary Tree

Check in Right SubTree Strict Binary.

So It will check, is SBT contained by any SubTree.

**

Que:- write a C Program to check given two Binary Trees are equal or not.

```
Equal (r1, r2)
{
1. If ((r1 == null) && (r2 == null))
       return(1);   /* Both Empty */
2. If ((r1 != null && r2 == null) || (r2 != null && r1 == n
       return(0);   /* 1 empty but
                       other one no
```

3. If ($r_1 \rightarrow data == r_2 \rightarrow data$) /* Both are not empty *, Compare data */

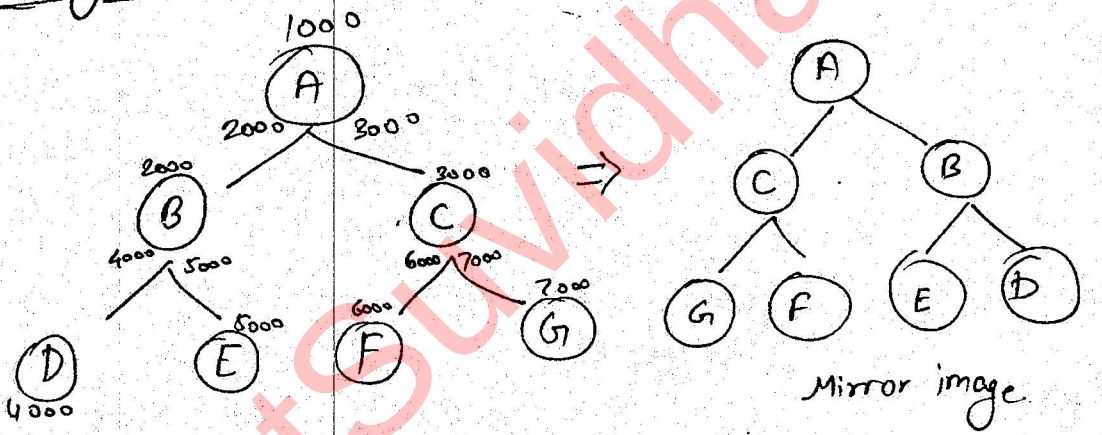
return (Equal ($r_1 \rightarrow lc, r_2 \rightarrow lc$) && Equal ($r_1 \rightarrow rc, r_2 \rightarrow rc$));

4. return (0); /* Data not equal */

}

**

Que :- write a C program to convert the given Binary Tree into its mirror image.



MI (r)

{

1. If (r == null)

return (r);

2. If (r → lc == null && r → rc == null)

return (r);

else

3. { t = r → rc;

r → rc = MI (r → lc);

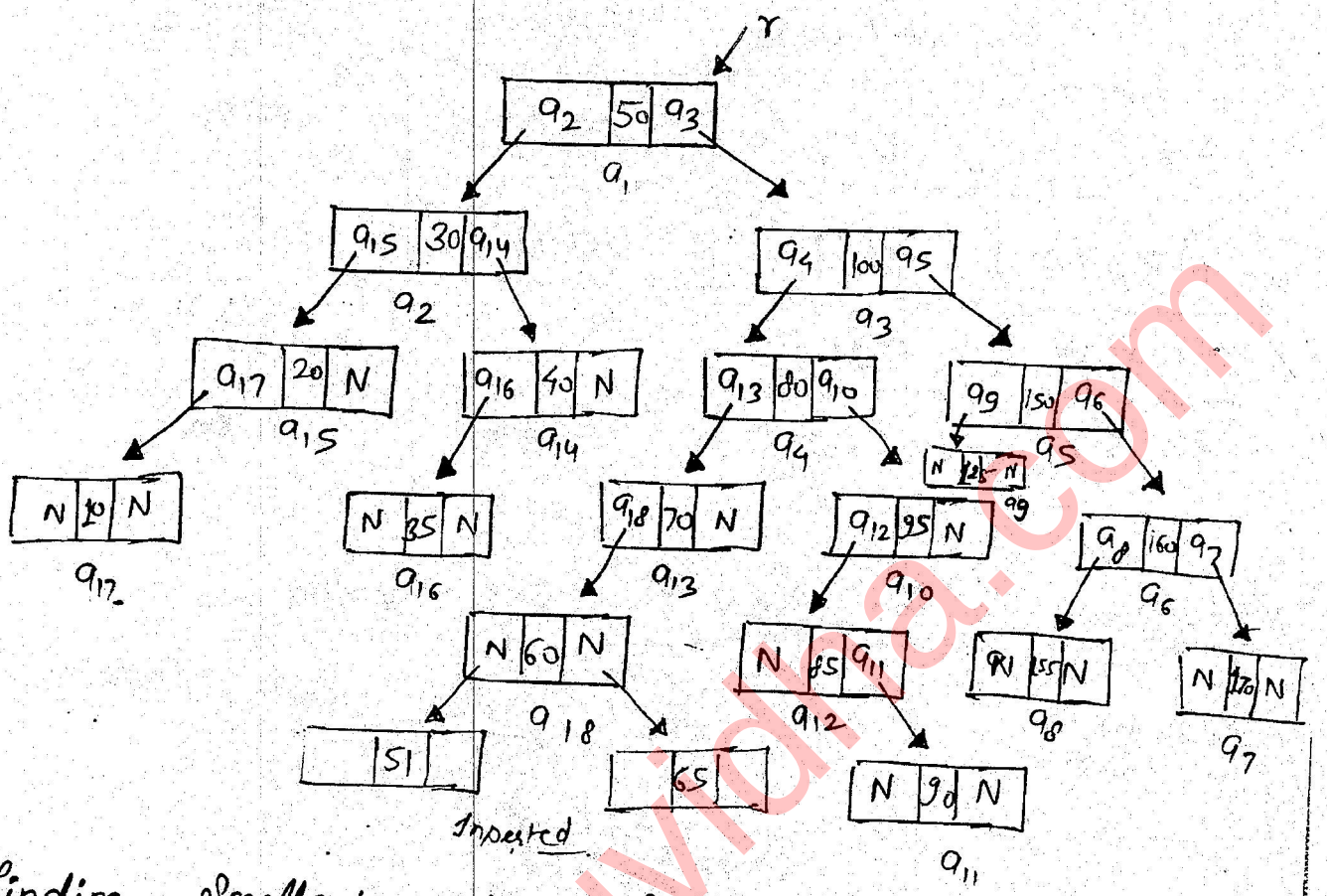
~~r → lc = MI (r → rc);~~

r → lc = MI (t);

} return (r);

Time = O(n) every case

"Binary Search Tree" :-



** finding Smallest Element :-

	Best case	Avg case	worst case	
B.S.T	O(1)	log n	O(n)	
Binary Tree	O(n)	O(n)	O(n)	almost like Normal Arrays (Linear search)
max heap	O(n)	O(n)	O(n)	
AVL Tree	O(log n)	log n	log n	

** finding element x :-

	Best case	Avg case	worst case
B.S.T.	O(1)	O(log n)	O(n)
Binary Tree	O(1)	O(n)	O(n)
max heap	O(1)	O(n)	O(n)
AVL Tree	O(1)	O(log n)	O(log n)

To find Element 'x' not exist :-

	Best case	Avg case	coorst case
B.S.T.	O(1)	O(log n)	O(n)
Binary Tree	O(n)	O(n)	O(n)
max heap	O(1)	O(n)	O(n)
AVL Tree	O(logn)	O(logn)	O(logn)

"Insertion in B.S.T" :- (one insertion)

Step-1st :- create the Node with malloc()

with a pointer P holding its address

Time ⇒ O(1) [Constant]

Step-2nd :- find correct place for Node.

⇒ O(n) [coorst case]

Step-3rd :- Link them. ⇒ O(1) [Constant time]

Time :- O(1) + O(n) + O(1) ⇒ O(n) [coorst case]

O(1) + O(1) + O(1) ⇒ O(1) ⇒ [Best case]

O(1) + O(logn) + O(1) ⇒ O(logn) ⇒ [Avg case]

Note :- In B.S.T. Insertion is always done at place.

:- Deletion :-

Inorder traversal of B.S.T. = Ascending order.

50, 10, 20, 30, 35, 40, 50, 60, 70, 80, 85, 90, 95, 100, 125, 150, 155, 160, 170

51 65

Inorder Predessor (50) ← — — → Successor (inorder for 50)

for 50 inorder successor :- (Right subtree smallest element) i.e 51.

50 inorder Predecessor :- (left subtree Largest element i.e. 40)

finding Successor of element 'x' :-

finding element 'x' ⇒ O(n) worst case (9 time)

find Right subtree smallest element ⇒ ~~can't exceed~~ 6 time } O(n) worst case (a+b) can't exceed n

finding element x ⇒ O(1) B.C.
finding Right subtree smallest ele ⇒ O(1) B.C. } O(1) Best case

Deleting leaf Node (0 kids).

:- find that Node :- O(n) W.C with r (Points) O(1) B.C. } ⇒ O(n) W.C, O(1) B.C, O(log n) Avg case

go with a friend p

P→lc = Null
free(r);
r = Null; } O(1)

Algorithm :-

Deletion Algo :-

Step - 1st :- find 'x' ⇒ O(n) worst case

Step - 2nd :- How many kids you have?

i) 0 - child

⇒ delete simply by changing few links ⇒ O(1)

ii) 1 - child

⇒ ~~Connect~~ simply delete by connecting grand father & grand child. ⇒ O(1)

iii) 2 - children :-

max O(n) {
- Replace data 'x' by Predecessor data & delete Predecessor.
- (or)
- Replace data 'x' by Successor data & delete Successor.

But overall time complexity = O(n) [w.c.]
= O(1) [B.C.]
= O(log n) [Avg case]