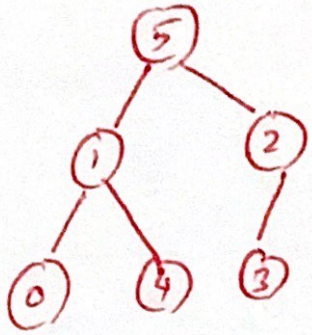
Ancestor Matrix from Binary Tree



In this TREE.

N=6

Nodes are numbered 0 to 5

(1) The Node values are numbered 0 to N-1

(2) N = number of Nodes.

(3) We don't do the naming / numbering or marking.

(4) The Nodes values are numbered like this.

output:- 2D Ancestor MATRIX, such that the Matrix has. 0 to (N-1) Rows, 0 to (N-1) Colums i.e NxN Matrix.

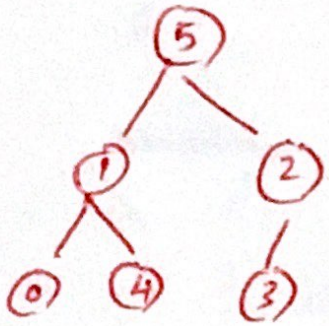
filling of values:- for any cell a(i,j)

i = row Index.

j = column Index.

If i = ancestor of j, Then a(i,j) = 1

else, a(i,j) = 0.



At any a(i,j) ask the question,
is Node numbered (i) an ancestor of
Node numbered (j)

		0	1	2	3	4	5
(leaf)	i = 0	0	0	0	0	0	0
	i = 1	1	0	0	0	1	0
	i = 2	0	0	0	1	0	0
(leaf)	i = 3	0	0	0	0	0	0
(leaf)	i = 4	0	0	0	0	0	0
	i = 5	1	1	1	1	1	0

N x N matrix
6 x 6 Matrix

for (eg) i = 0, j = 3,
is The Node Numbered zero, The ancestor of
Node That is Numbered as (3). The answer is
No. So, it gets a zero.

for (eg) i = 1, j = 4
Is The Node numbered (1), The ancestor of
Node That is Numbered as (4). The answer is
Yes. So, it gets a 1.

As you can see from the ancestor Table, most of its values are filled with zero. So,

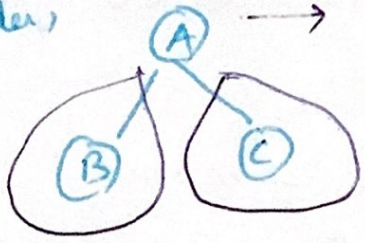
⇒ while an ancestor Matrix is created, it makes sense to create a N×N Matrix, pre-filled with zeroes.

⇒ N = Number of Nodes. So, we would have to find the size of the Binary Tree. (to Build the initial ancestor Matrix)

Size of the Binary Tree.

Its the Count of number of nodes in the Binary Tree.

Consider,

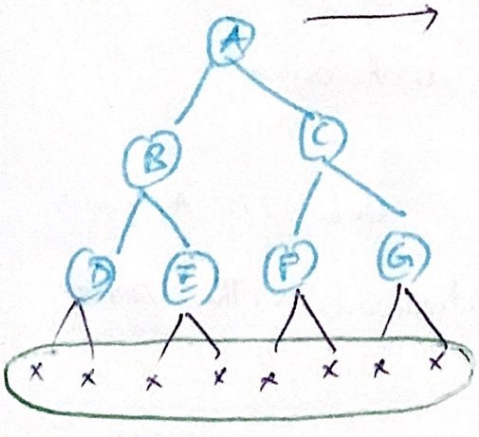


→ with Respect to Node(A), the size can be written as

(size of LEFT) + 1 + (size of RIGHT)

↓ Counting itself.

Consider,



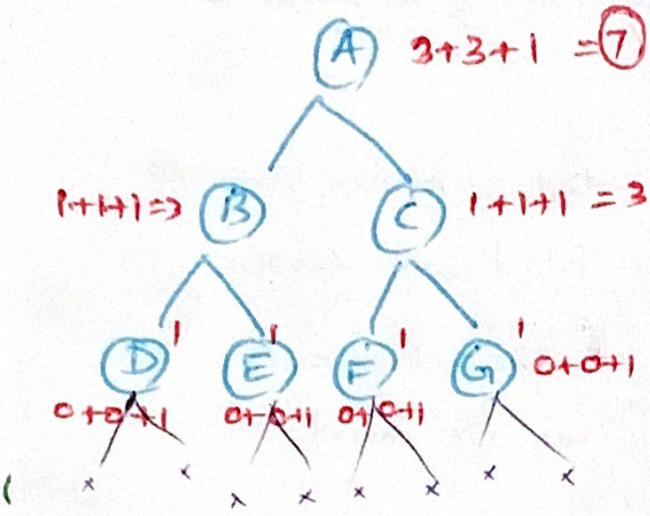
→ size of Left + 1 + size (Right)

size of size (B) + 1 + size (C)

size (D) + 1 + size (E)

size (F) + 1 + size (G)

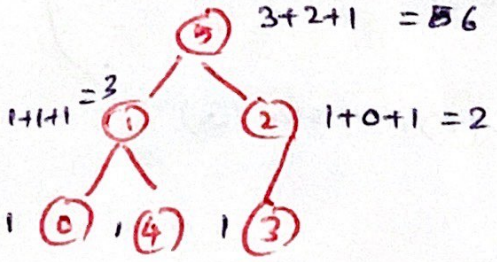
⇒ They all Contribute 0.




```
Size (node) {
        if (!node) {
                return 0
        }
        return size (node.left) + 1 + size (node.right)
}
```

The above recursive function can be used.
To determine the size of Binary Tree &
Thereby Construct an initial ancestor.
Matrix of (size x size) dimension.

In our Tree,



So a 6x6 2D Array is initialized with the zeros

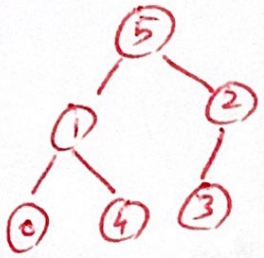
Now The Tree has To Be Traversed to fill in The Ancestor Matrix.
a simple pre-order Traversal would Also Do,

Root → LEFT→RIGHT

✓

Naturally Traverses The Root ①st (ancestor ①st)
giving a chance To store The Node
into an Array of Ancestors.

for (4)



pre-order:-

5, 1, 0, 4, 2, 3.

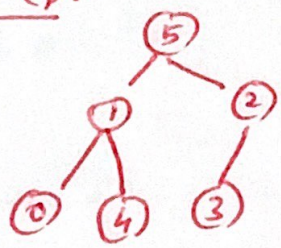
So when procesing Node (4),
⇒ Node 1, 5 can be stored
⇒ Ancestor Can be stored.

When procesing Node (3)
⇒ Node 2, 5 are stored.
⇒ Node ① That was previously stored
was popped out as its SubTrees.
(0), (4) have finished procesing.

for any Node
→ push (store Temporarily)
→ proces Left SubTree
→ proces Right SubTree
→ remove it from Ancestor List

```
function constructAncestorMatrix(node, ancestorArr, ancestorMat){
    if(!node){
        return;
    }
    ancestorArr.forEach(ancestor => {
        ancestorMatrix[ancestor.val][node.val] = 1
    });
    ancestorArr.push(node);
    constructAncestorMatrix(node.LEFT, ancestorArr, ancestorMat);
    constructAncestorMatrix(node.RIGHT, ancestorArr, ancestorMat);
    ancestorArr.pop();
    return ancestorMatrix;
}
```

For eg:-



Pre-order:-

5 1 0 4 2 3

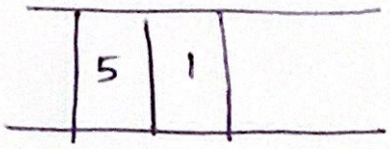
	0	1	2	3	4	5
0						
1					1	
2						
3						
4						
5					1	

for The Node,

(4).

When Node = (4),

Ancestor Array has



Now, we iterate over these ancestors Array.

We know that (5) & (1) are ancestors of (4) per pre-order.

So, we have to fill.

Matrix [5] [4] ⟶ is (5) An ancestor of (4) → Yes — 1

Matrix [1] [4] ⟶ is (1) an ancestor of (4) → Yes — 1

While manually filling, we filled asking The question.

→ is The Node numbered i=x an ancestor of
Node number j=y (Row wise)

But while filling via The program, its filled as.

→ Node (A) is ~~ancestor~~ descendent of Node (B)
→ Node (A) is descended of Node (C)

} Column wise.