Trees

Arrays

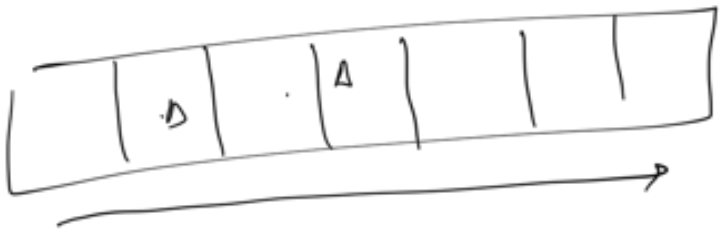
LL

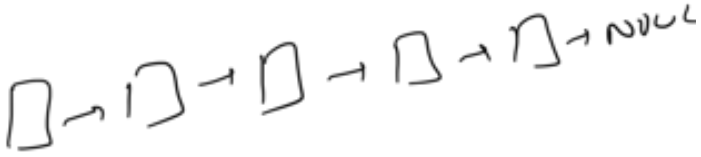
Queue

Stack

Hash Map

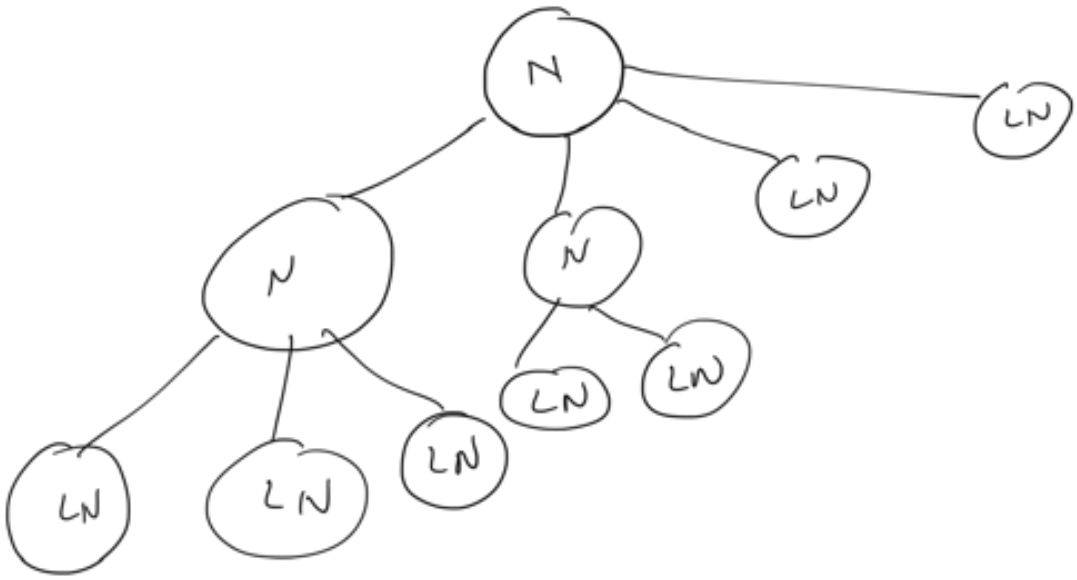
Hash Sets

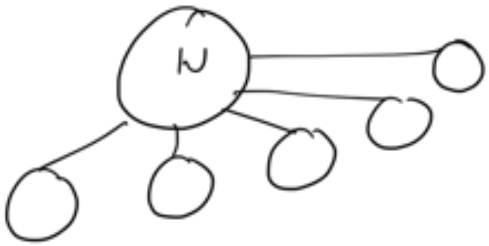




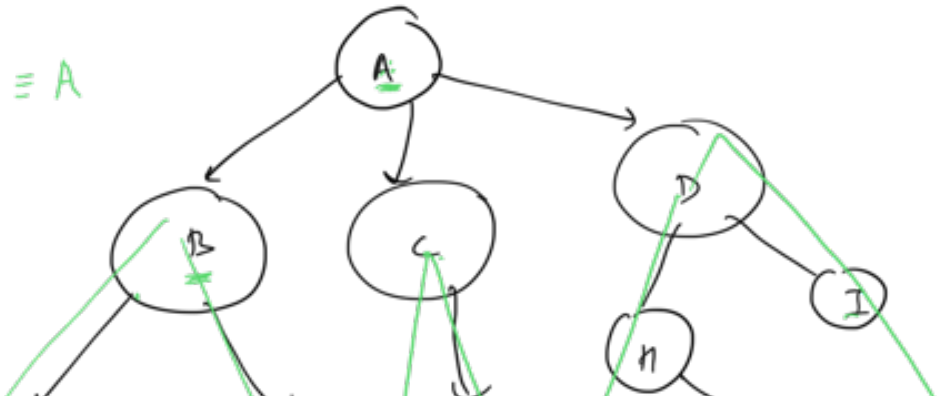
Hierarchial DS

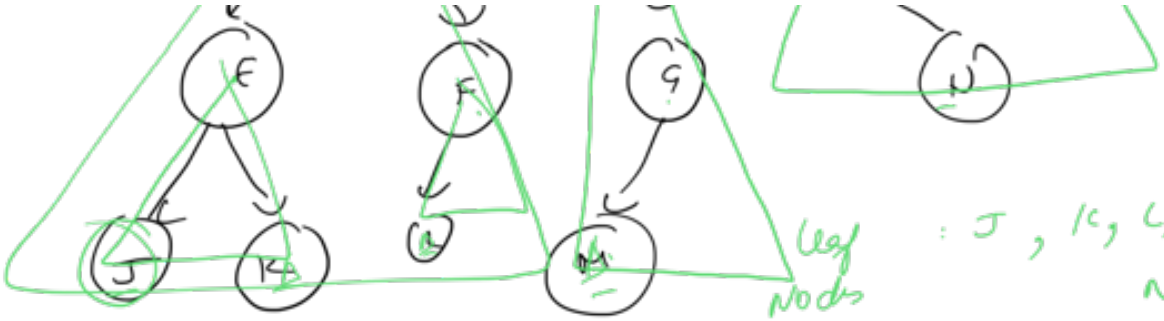
Tree





Root ≡ A



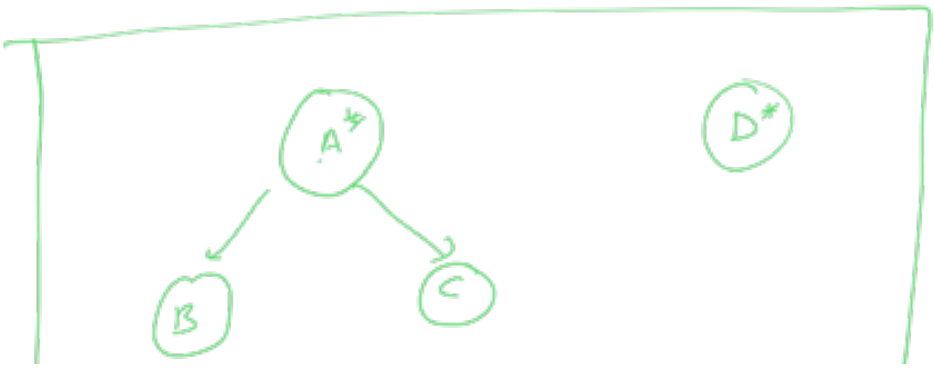


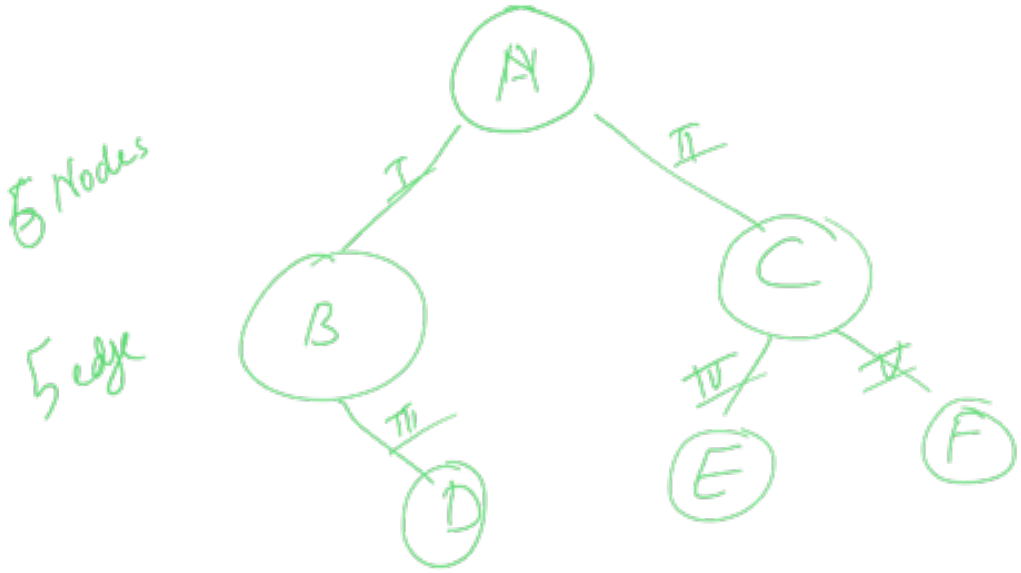
: J, K, L, M, N

I

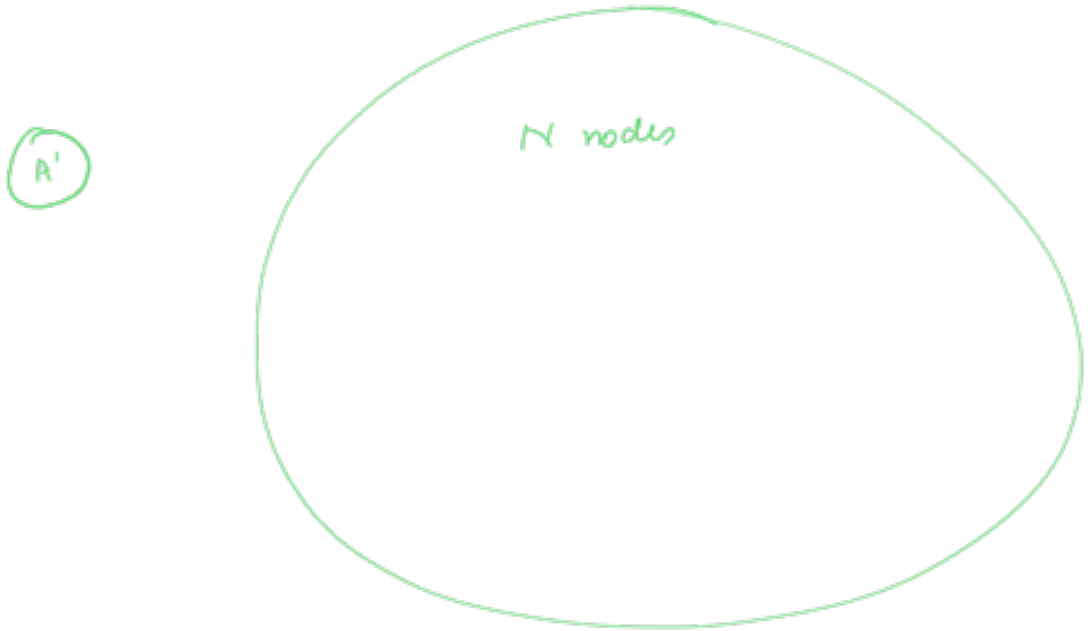
Tree ≡ Recursive

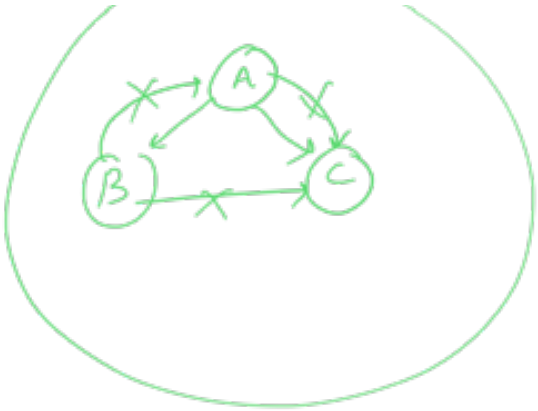
Tree: a collection of nodes, N
in which all of the nodes are connected
and # edges = N-1





Tree ≡ collection of Nodes





0 edge 1 st node
1 edge 2 nd
1 edge 3 rd

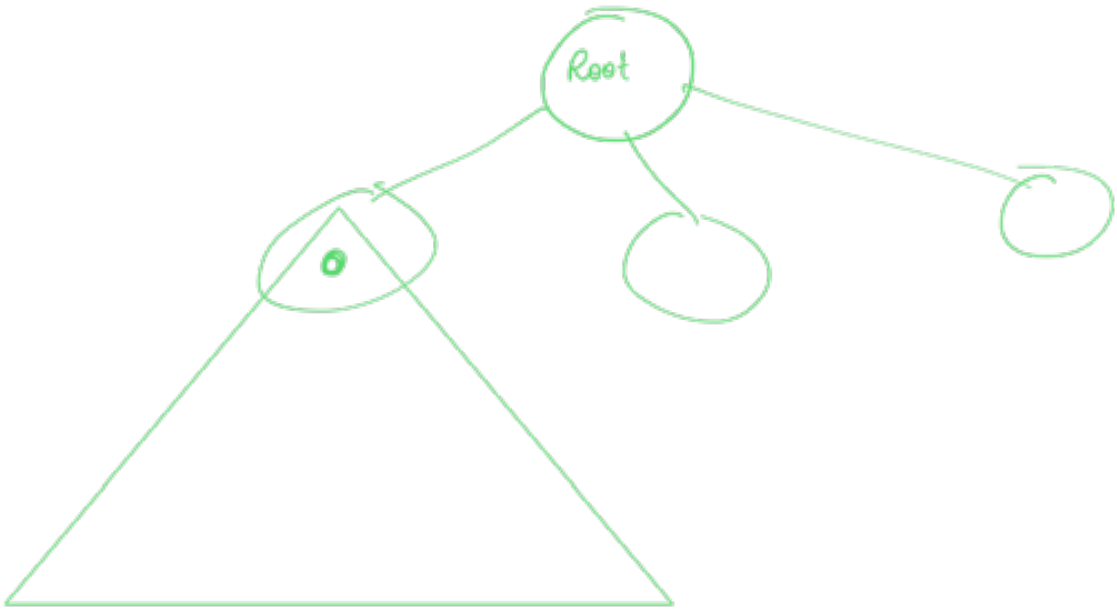
N nodes

atleast N-1 edges

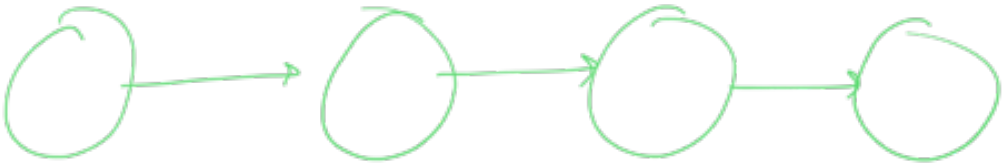
Parent

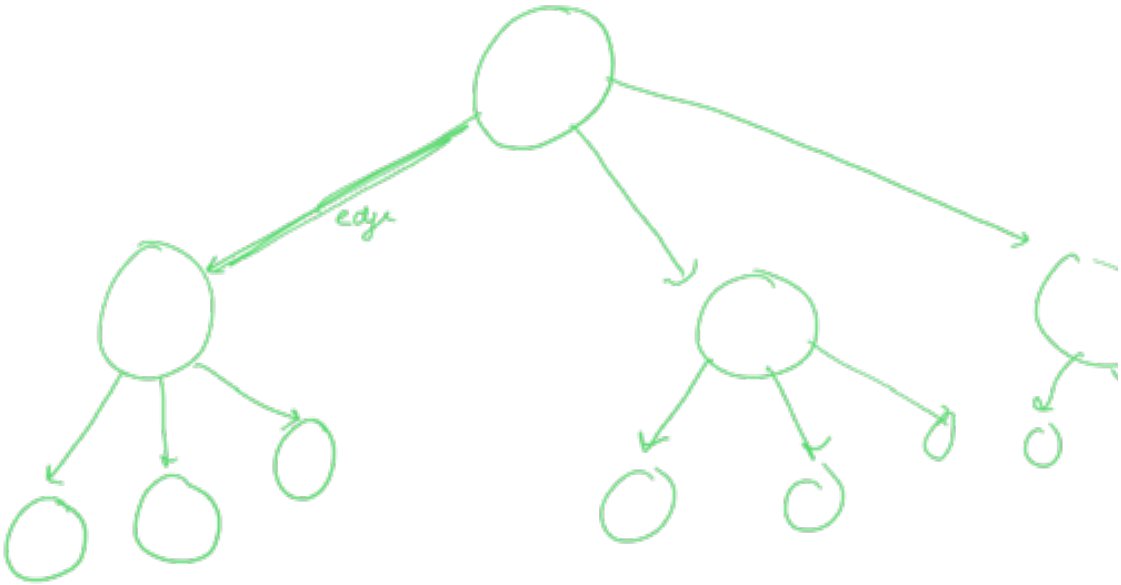
Child

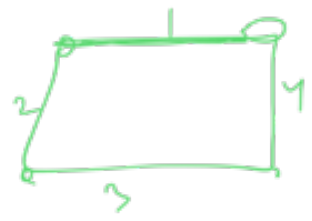
N nodes ≡ N-1 edges



L.L



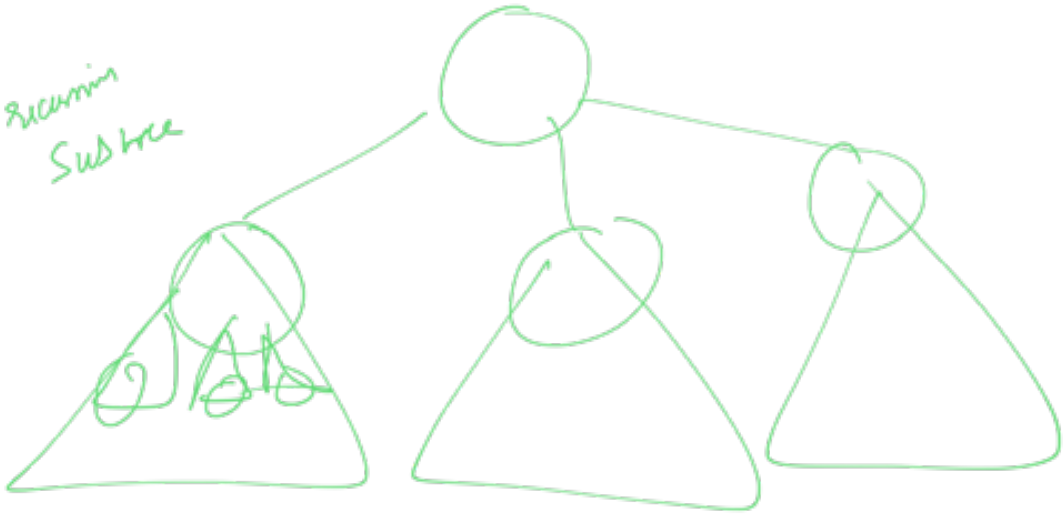


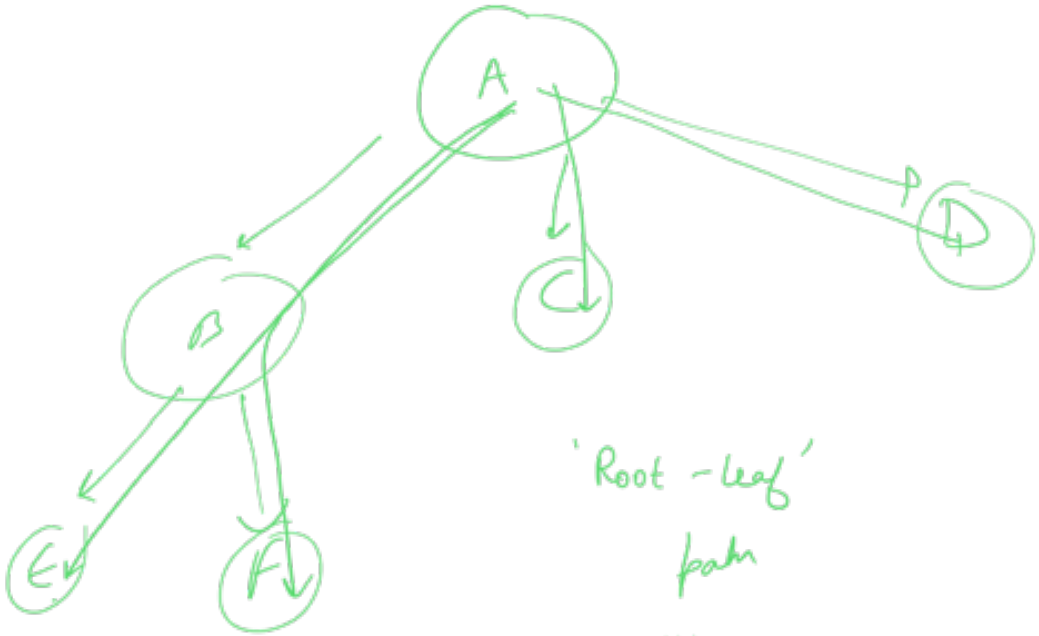


Tree

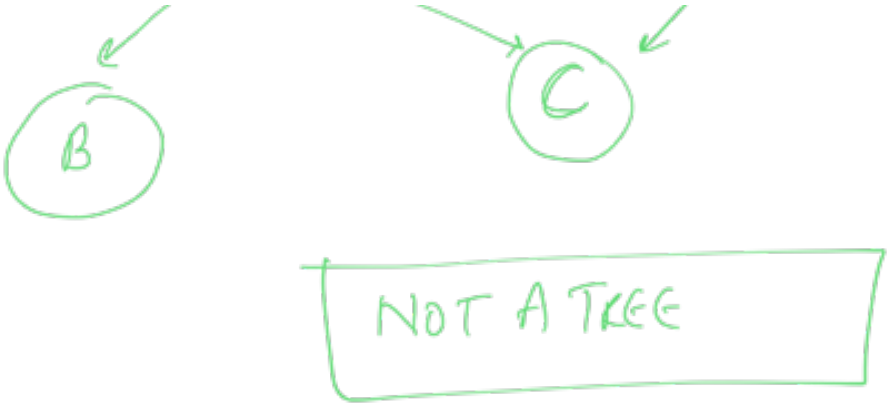
nodes ≡ data pointer

edges → structure

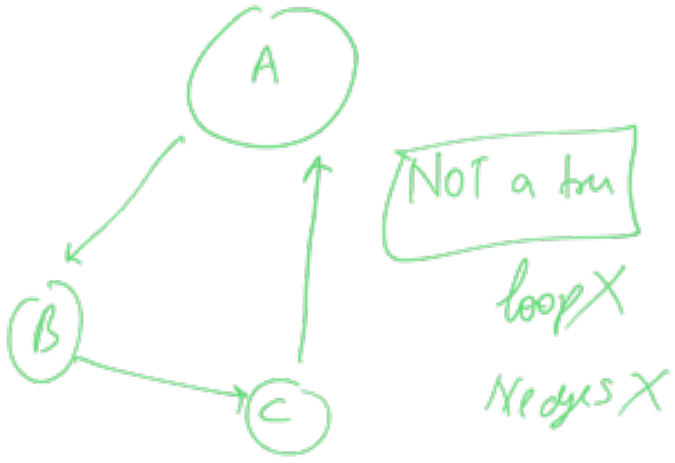


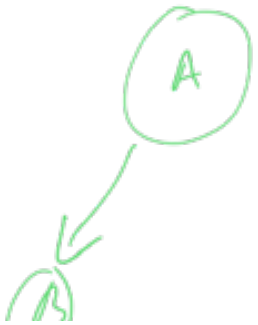


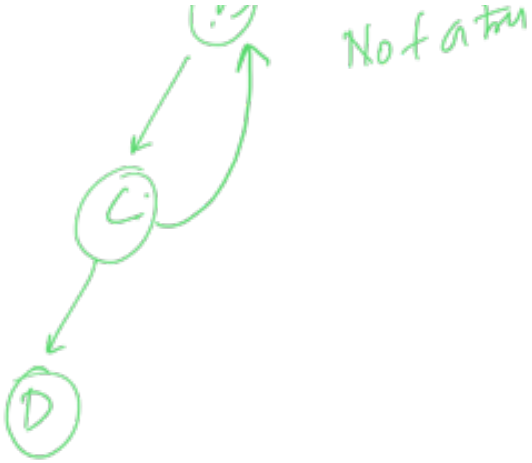
'Root - leaf'
path
|||
LL

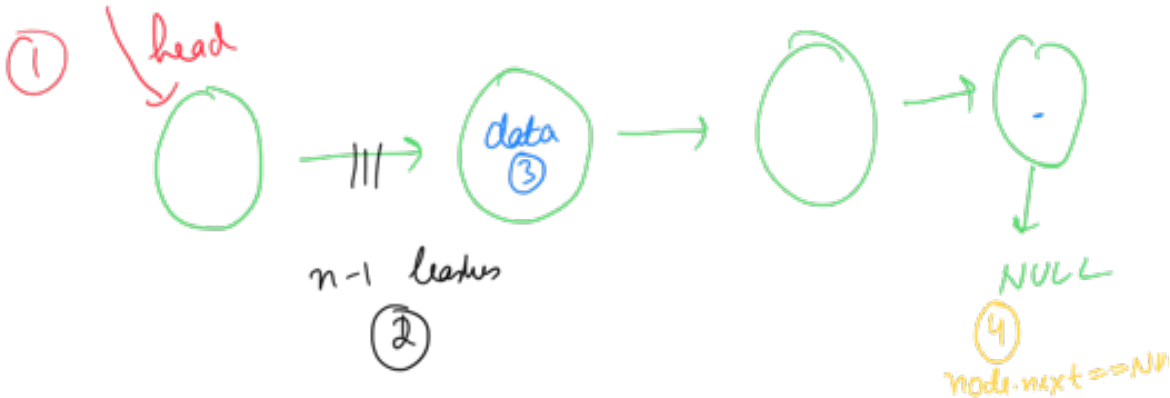


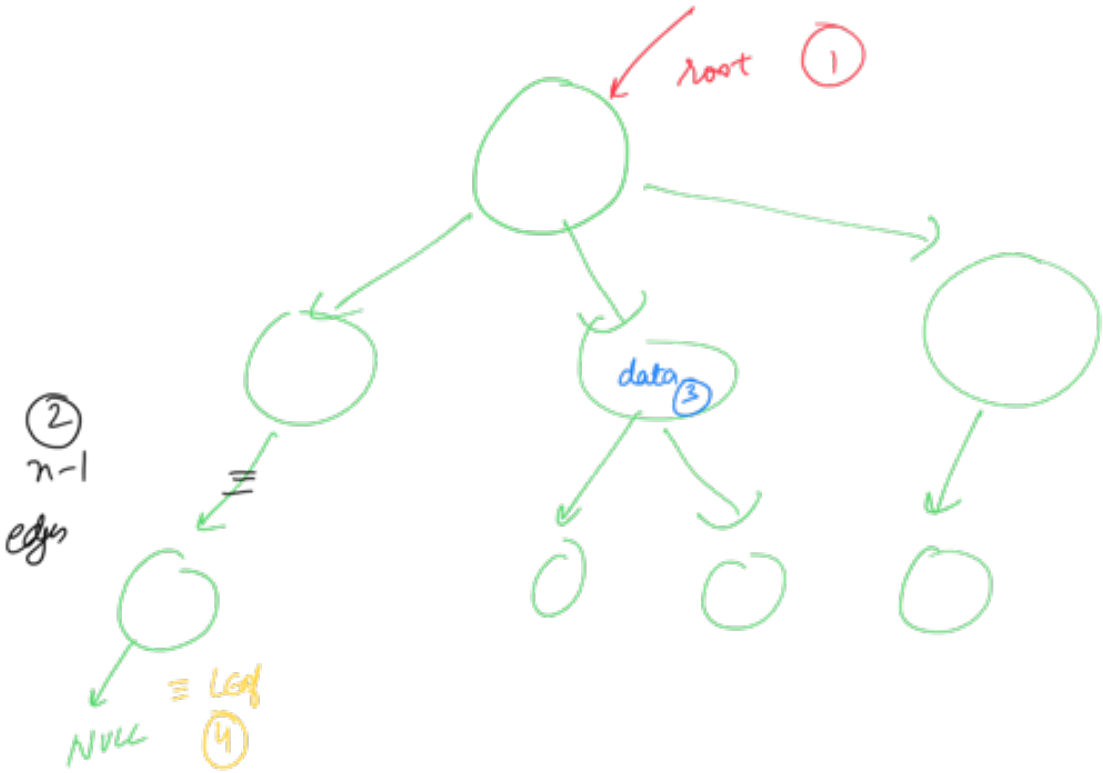
every node must have 1 parent
Other than the root






C
D

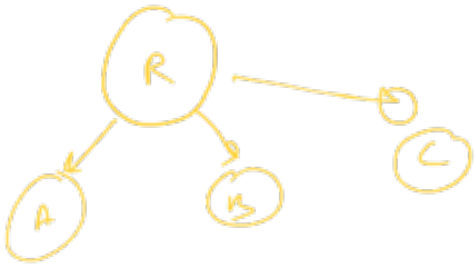

head
data
3
n-1 leaves
2
NULL
4


root
1
data
3
2
n-1
edges
NULL
≡ Leaf
4

Tree is basically hierarchical L.L

L.L is basically a special type of a Tree

≡ Tree is a more general d.s as compared to a LL



Path root to leaf ≡ L.L

L.L is a Tree with every node ==1 child

Class LinkedListNode

{

int data ✓✓

LinkedListNode next

}

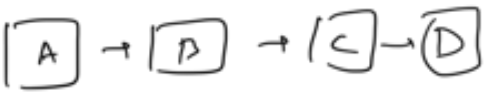
Class TreeNode

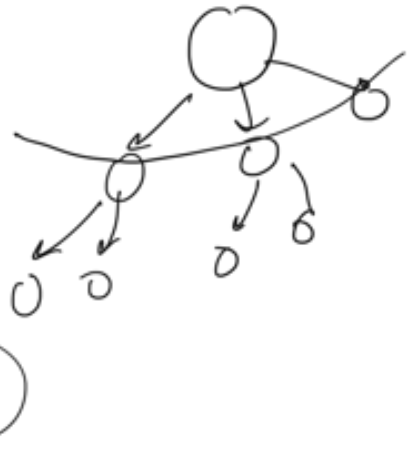
{

int data; ✓✓

TreeNode [] children;

}





Binary Tree

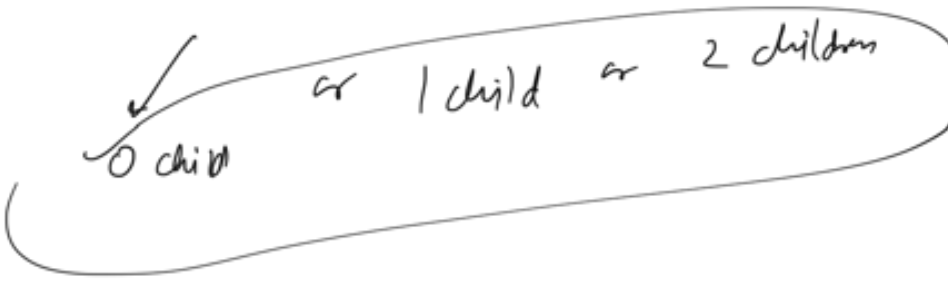
k-ary Tree : max number of children one node can have = k

n - ary Tree with n = 15

Binary Tree

k = 2

every node in the binary tree will have atmost 2 childrens




```
class TreeNode
{
    int data;
    TreeNode[] childrens;
}

class BinaryTreeNode{
    int data;
```

```
TreeNode createNode(i
{ //memory alloc
  TreeNode temp
```

TreeNode leftChild

TreeNode rightChild

}

= new

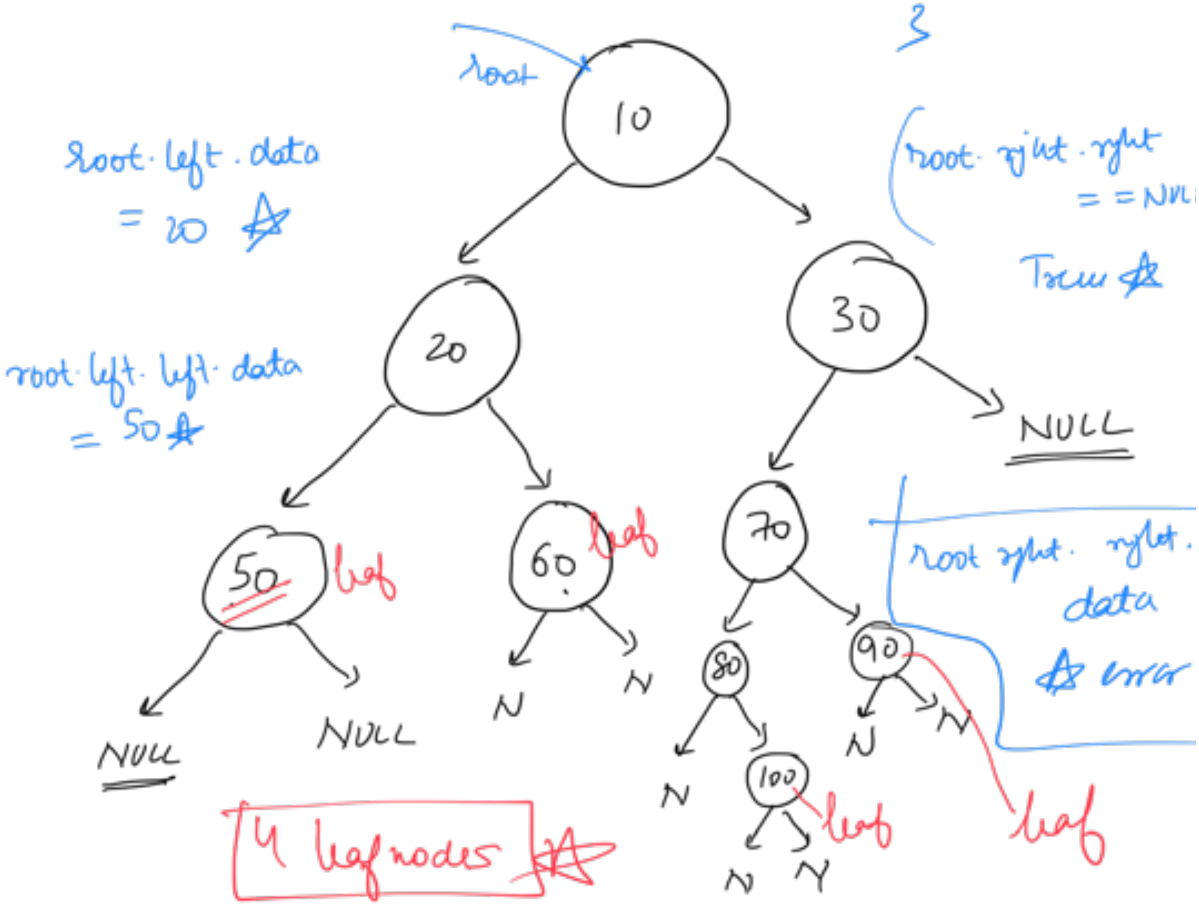
temp.data = v;

temp.leftChild = NU

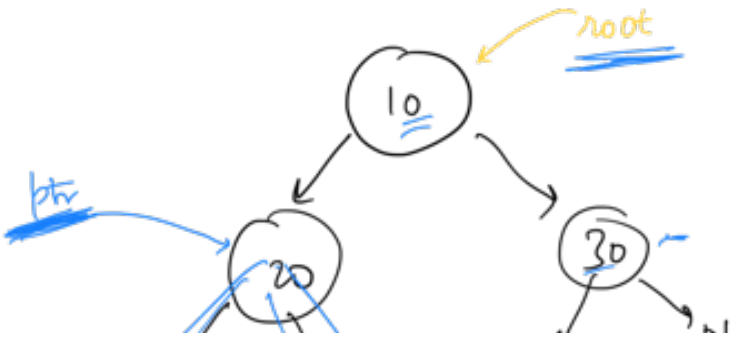
temp.rightChild = N

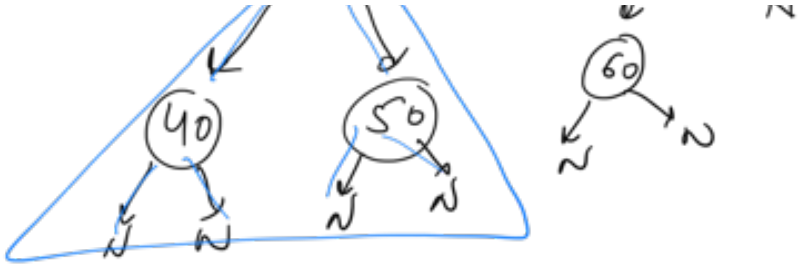
return temp

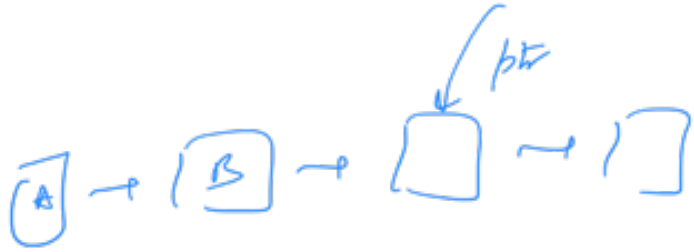
}

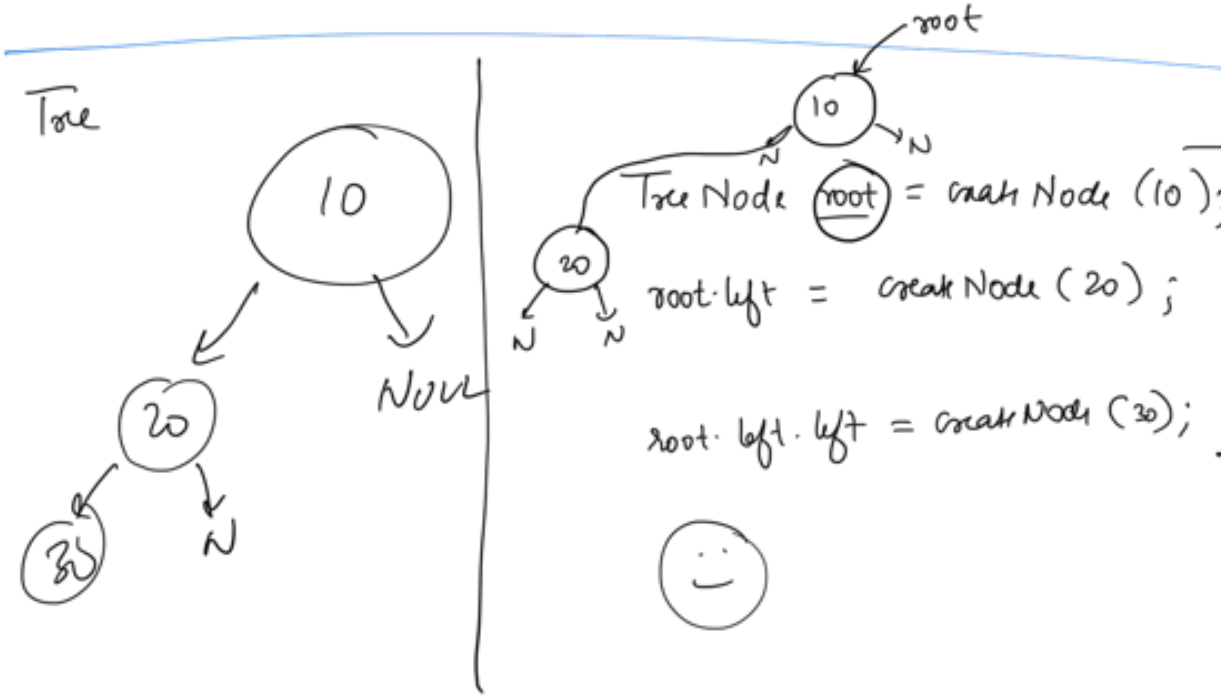


Leaf ≡ node with 0 children

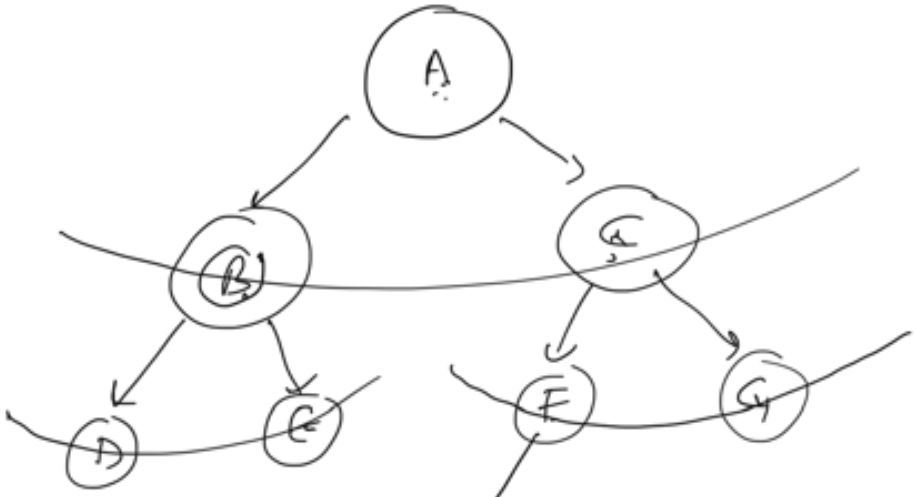






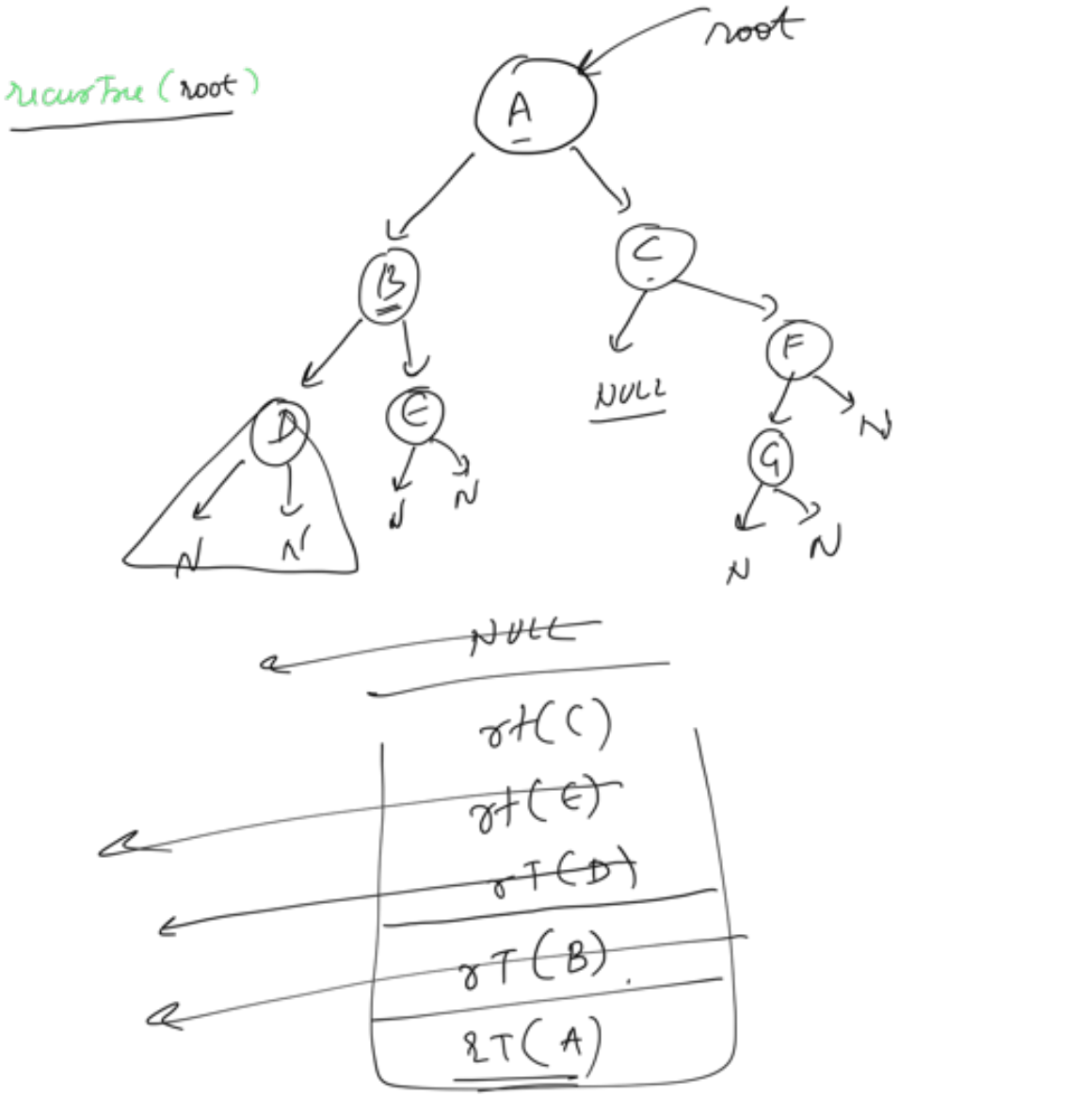


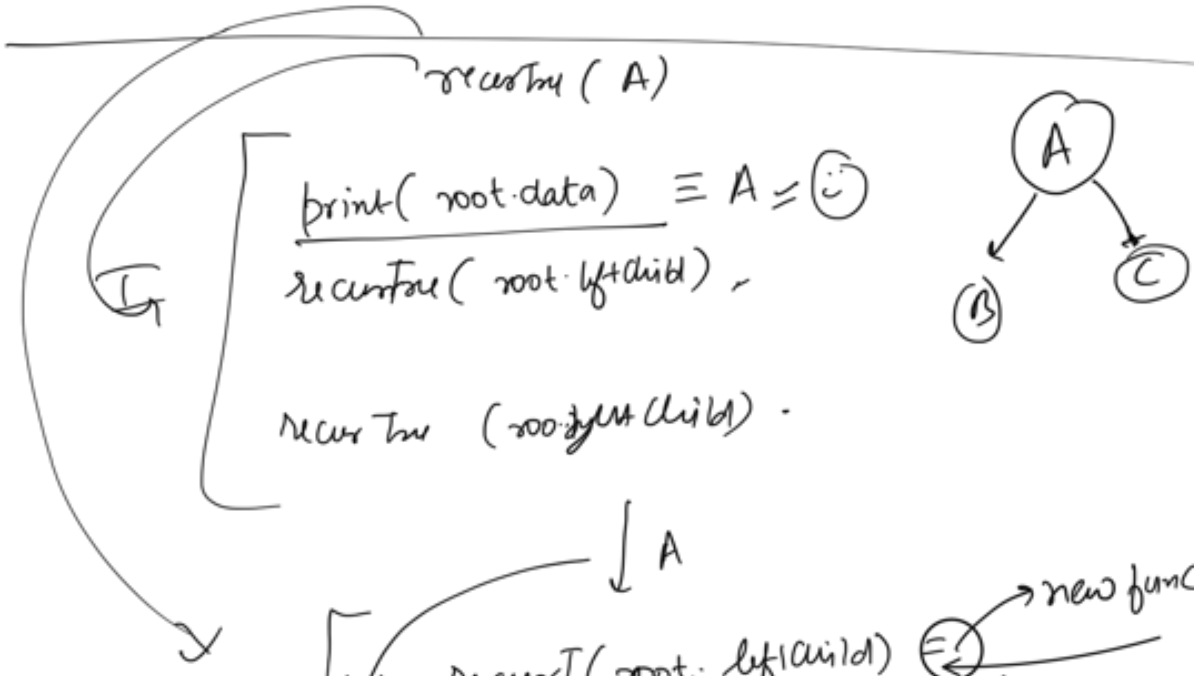

```
void traverseLL (LLNode head)
{
    LLNode temp = head;
    while (temp.next != NULL)
    {
        print(temp.data)
        temp = temp.next
    }
```




```
void recurTree (TreeNode root)
{
    if (root == NULL)        //Sanity
        return;
    if (root.left == NULL AND
        root.right = NULL)
        return
    // recur
    I.
    → recurTree (root.leftChild)
    II
    → recurTree (root.rightChild)
    III →
print (root.data)
}
```

Base Case


recurTree(root)
root
A
B
C
D
E
F
G
NULL
N
NULL
rT(C)
rT(E)
rT(D)
rT(B)
rT(A)


recurTree(A)
print(root.data) ≡ A ≡ :)
recurTree(root.leftChild).
recurTree(root.rightChild).
A
B
C
A
new func

II

101 recurT(root ...)

102 → print (root . data) ≡ A

103 recurT (root . rightchild)

III

recurTree (root leftchild)

recurT (root . rightchild)

print (root . data); ≡ A

---

void recurTree (TreeNode root)

{

// s.c + b.c

if(root == NULL) return

// b.c

if (root.left == NULL AND root.right == NULL

return

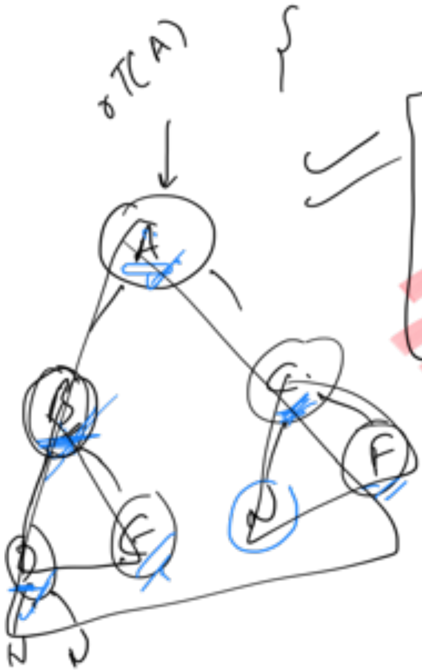
// recursive

print (root . data)

recurTree (root . leftchild)

recurTree (root . rightchild). — only part

}

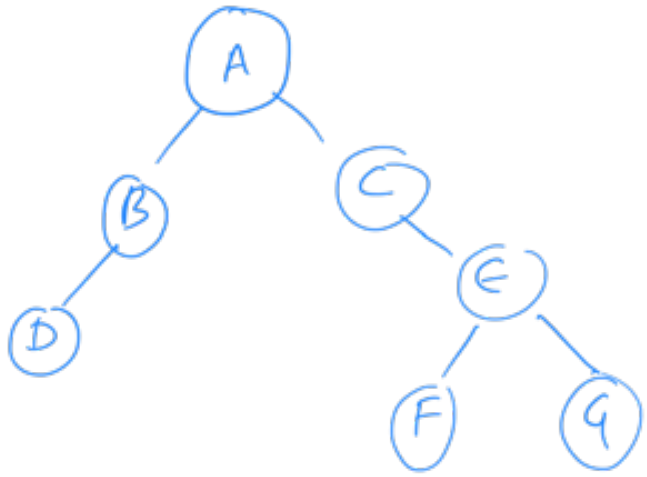


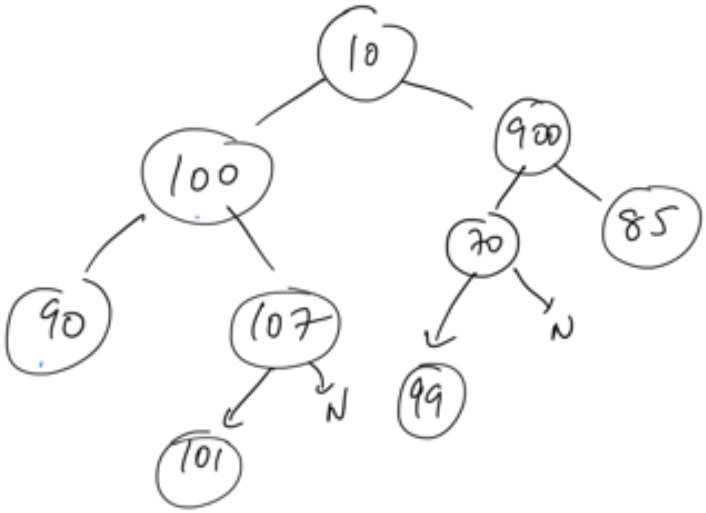
A B D E C F

Preorder Traversal

(recursively print yourself first)

A
B
D
C
E
F
G





10

100

90
107
101
900
70
99
85

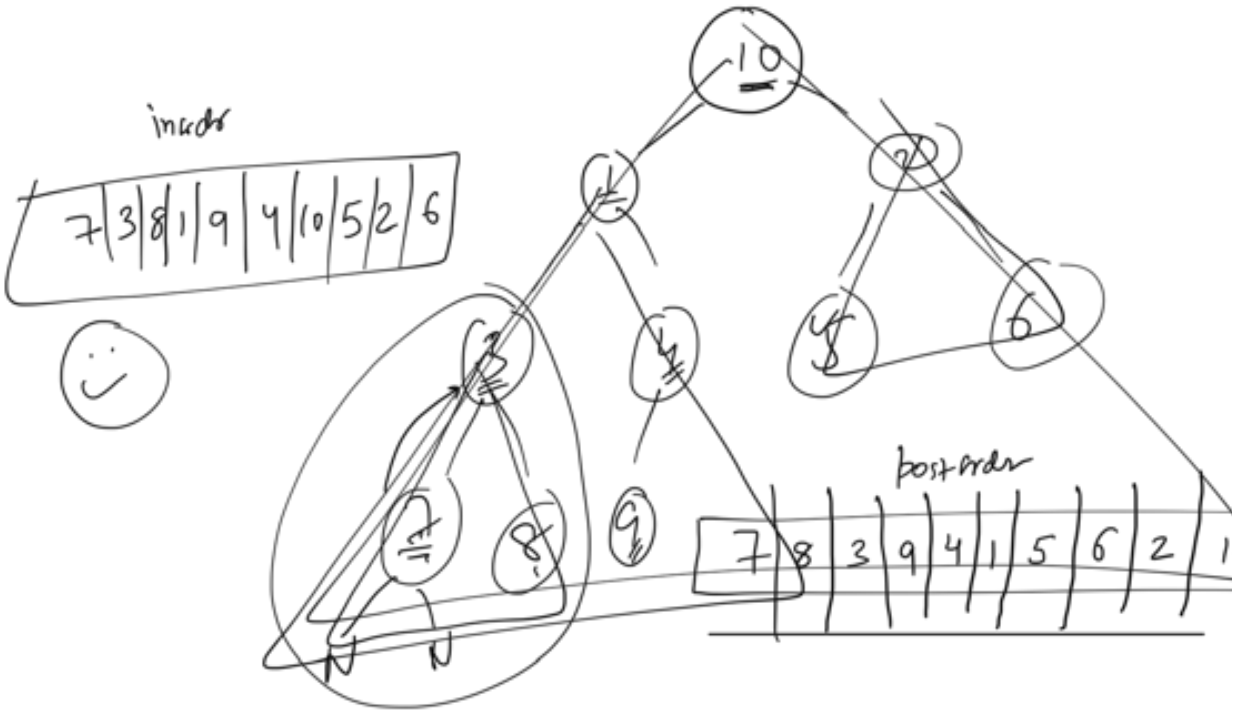
Inorder

// recursive

101 recurTree(root.leftchild)

102 print(root.data) → A

103 recurTree(root.rightchild) →



postorder

//recursive

recurTree(root.leftchild) ✓

recurTree(root.rightchild) ✓

print(root.data) //

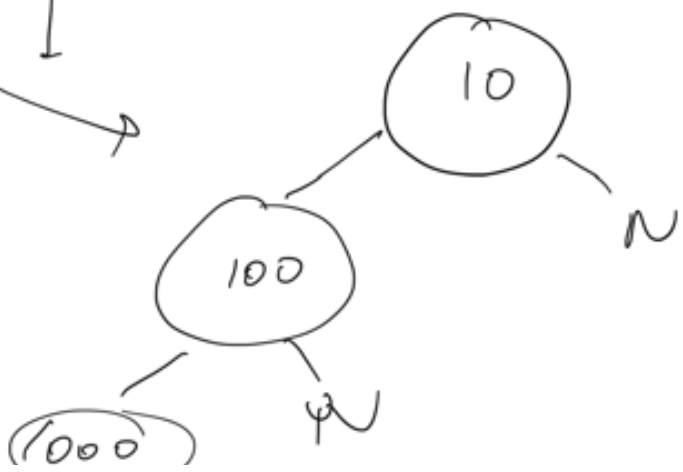
TC = O(N)

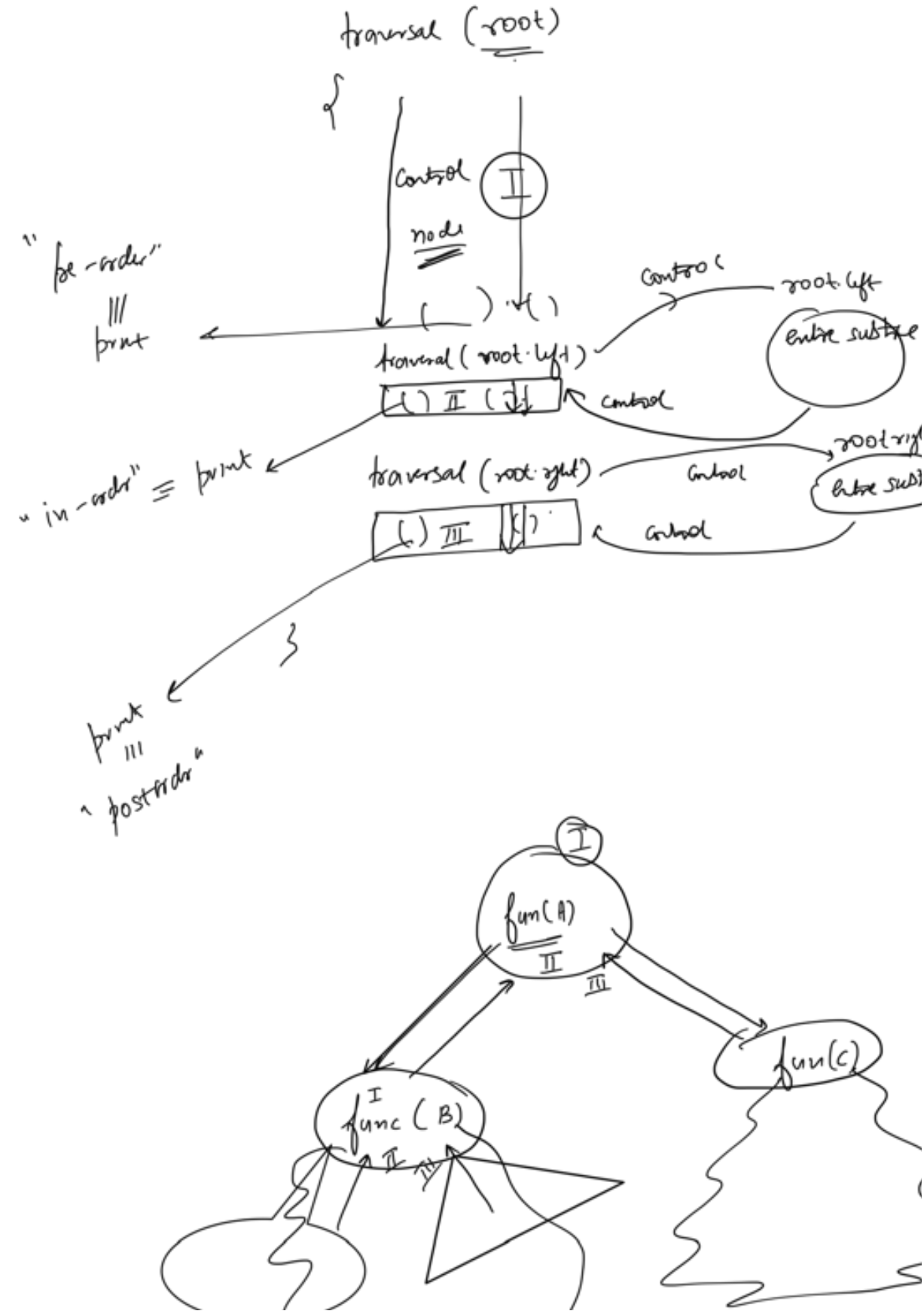
recur S.C = (depth of Tree)
= longest path from root to leaf

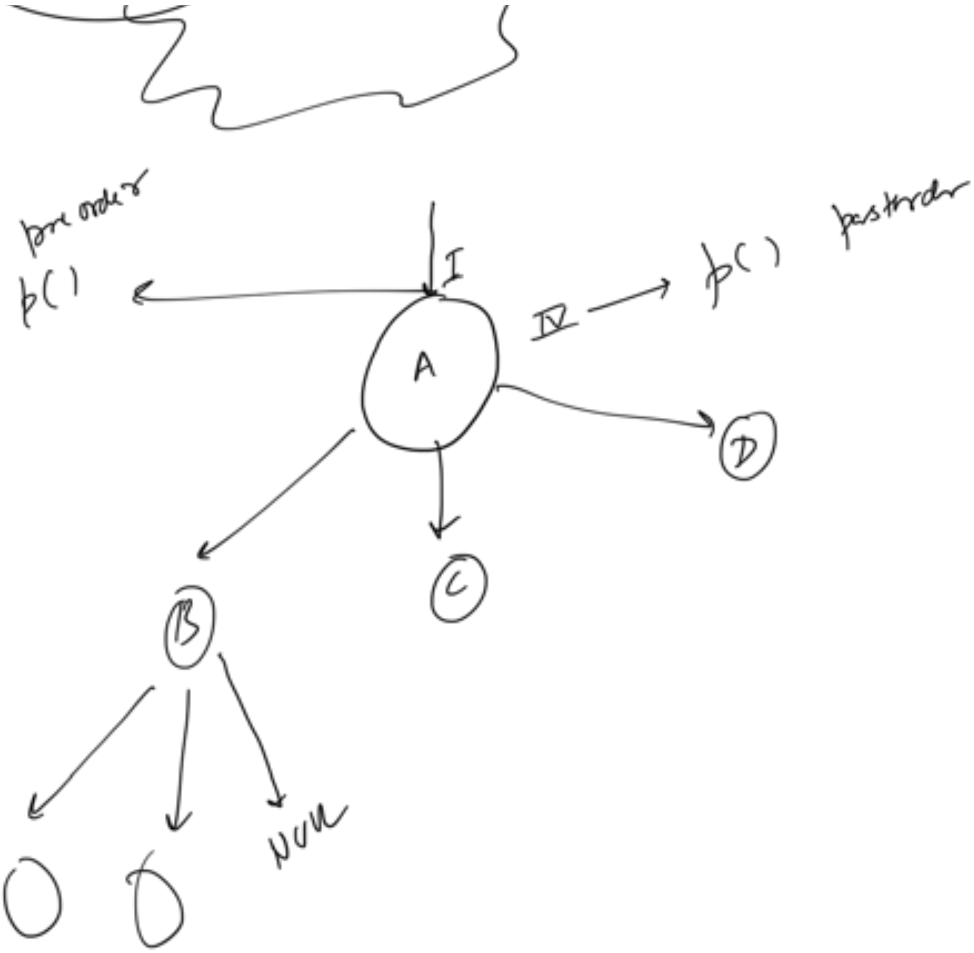
root.data = 10

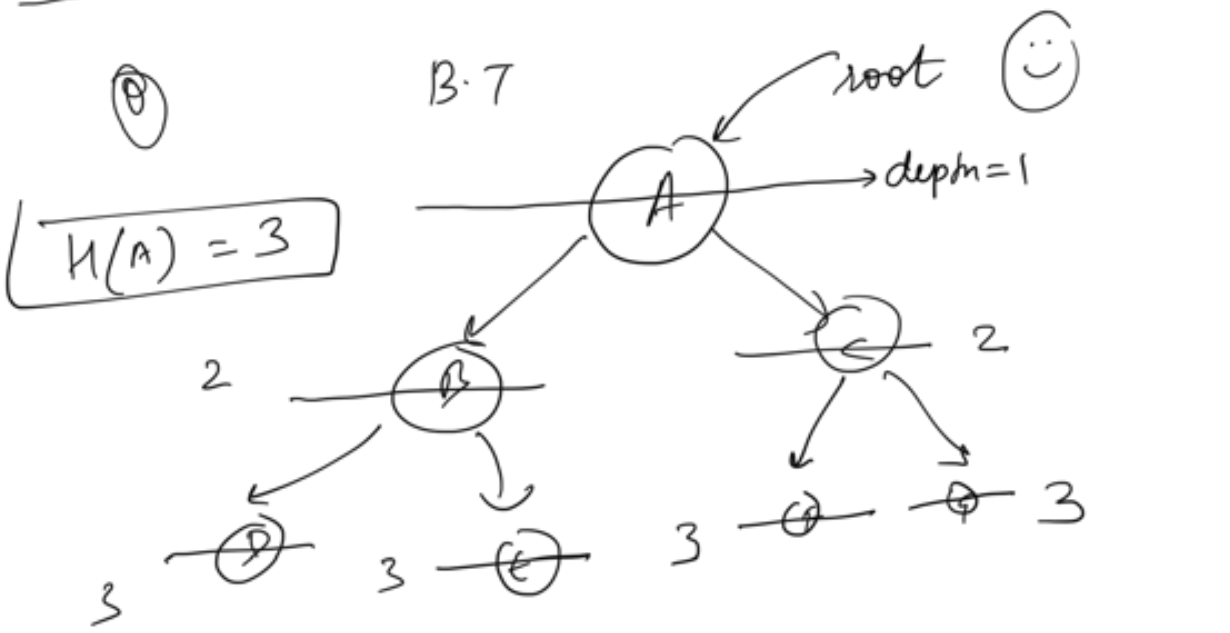
root.left.data = 100

r.l.l.d = 1000

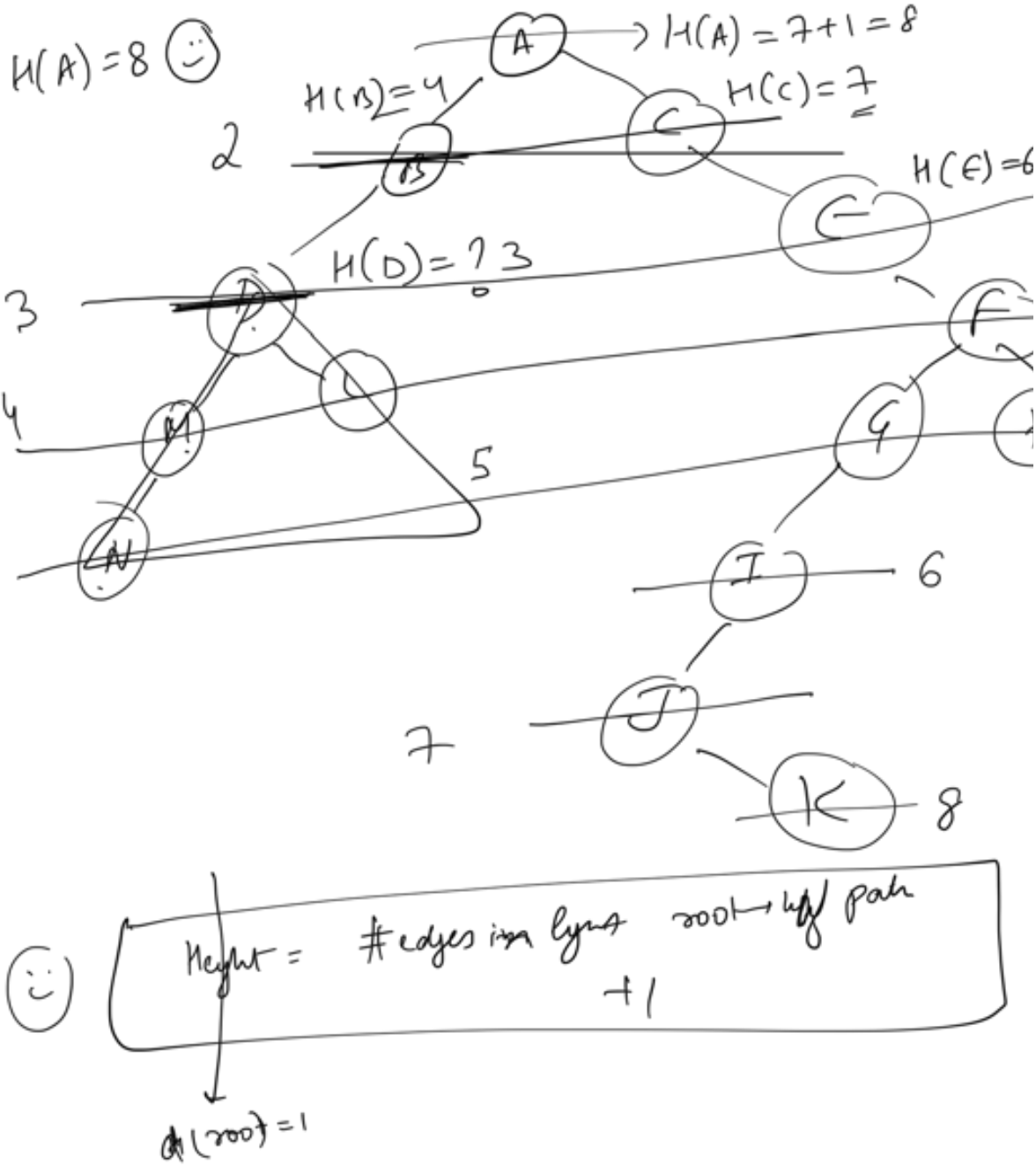



traversal (root)
control
node
"pre-order"
print
traversal (root.left)
control
root.left
entire subtree
"in-order" ≡ print
traversal (root.right)
control
print
"postorder"
I
fun(A)
II
III
fun(B)
I
II
III
fun(C)



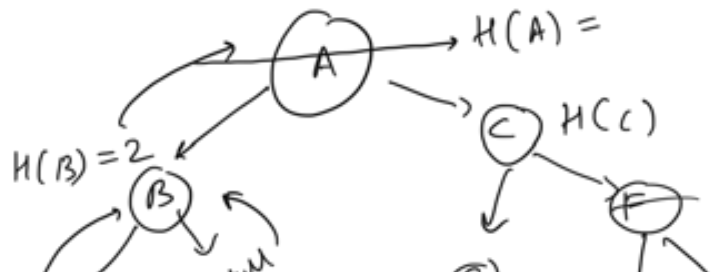


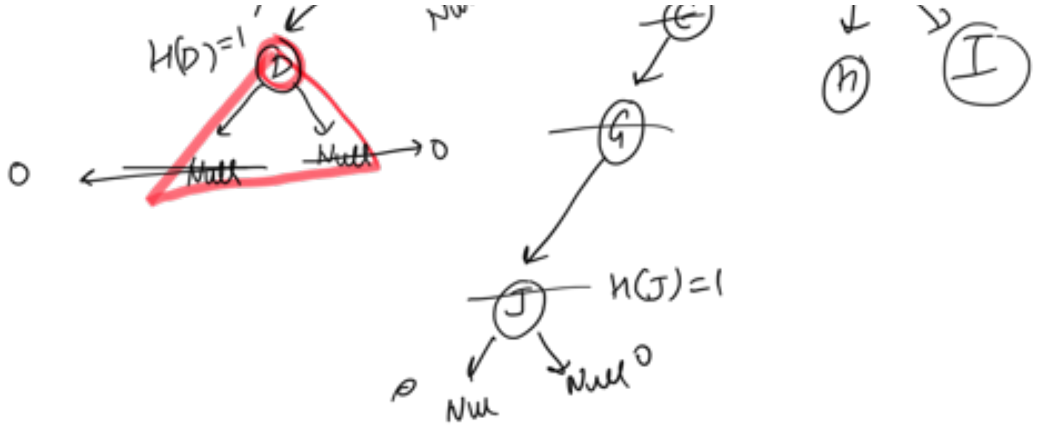
What is height of the Tree?



Height = # edges in longest root → leaf path + 1

H(root) = 1





```
int getHeight ( TreeNode root )
{
    // sanity cum base
    if (root == NULL)
        return 0
```

// Base Case

OR

Option (B)

```
if(root.left == NULL
&&
root.right == NULL
return 1;
```

Option (A)

Base = Sanity

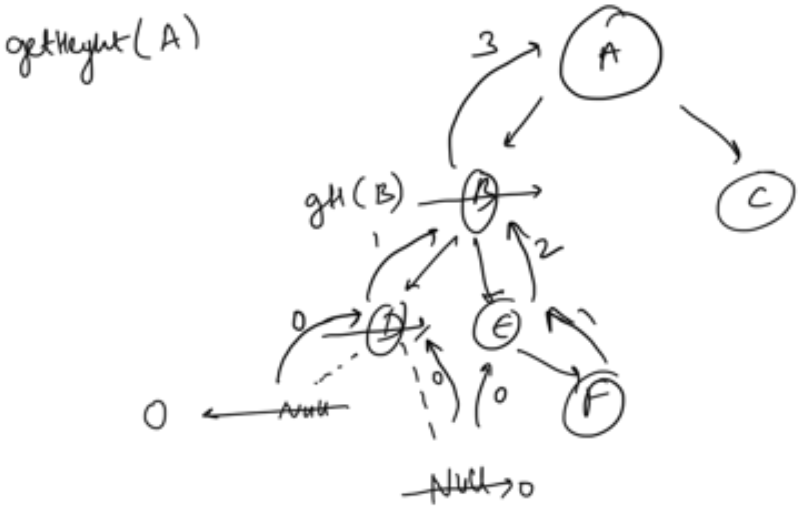
// recursive calls

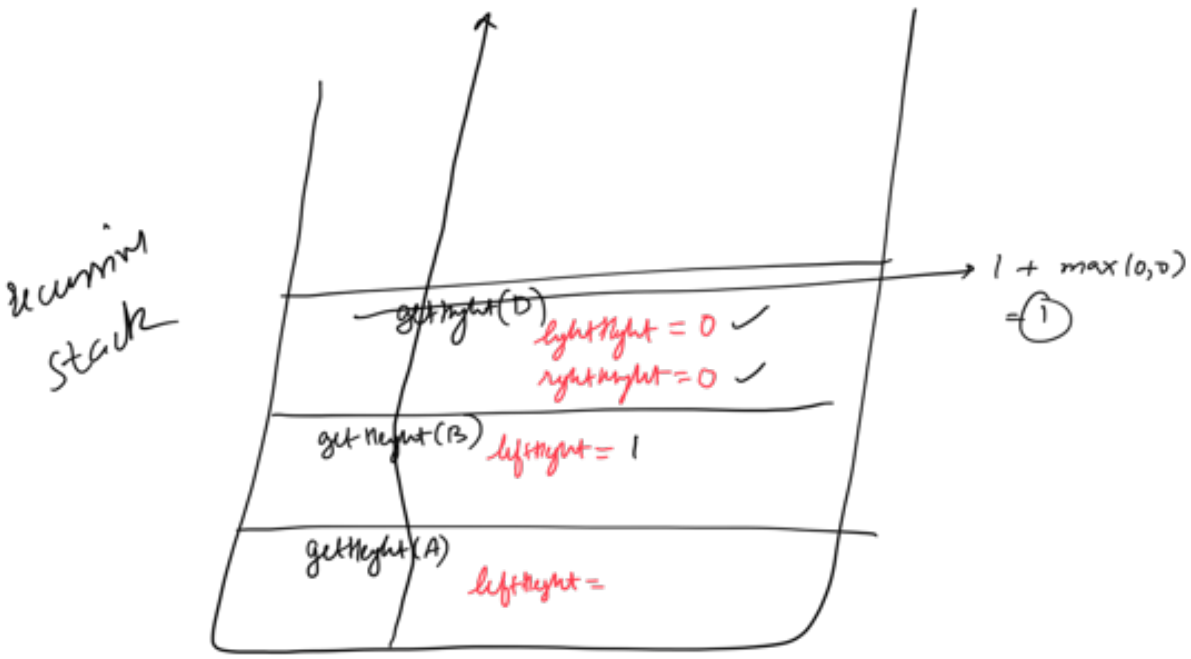
→ int lHight = getHeight (root.left)

→ int rHeight = getHeight(root.right) ≈

return 1 + max(lHeight, rHeight)

}

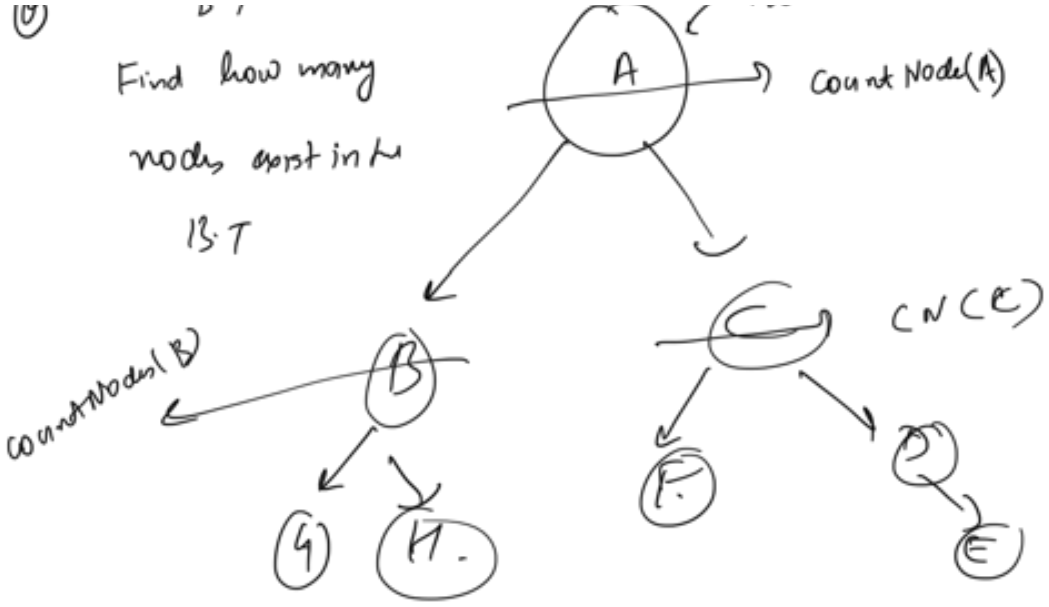




Given

R.T root

Find how many nodes exist in the B.T




```
int countNode(TreeNode root)
{
    // sanity check corner Base
    if (root == NULL)
        return 0;

    // Base Case
    if (root.lf == NULL && root.rgt == NULL)
        return 1

    // recurs
    int lC = countNode(root.lft)
    int rC = countNode(root.rgt)
    return 1+lC+rC;
}
```

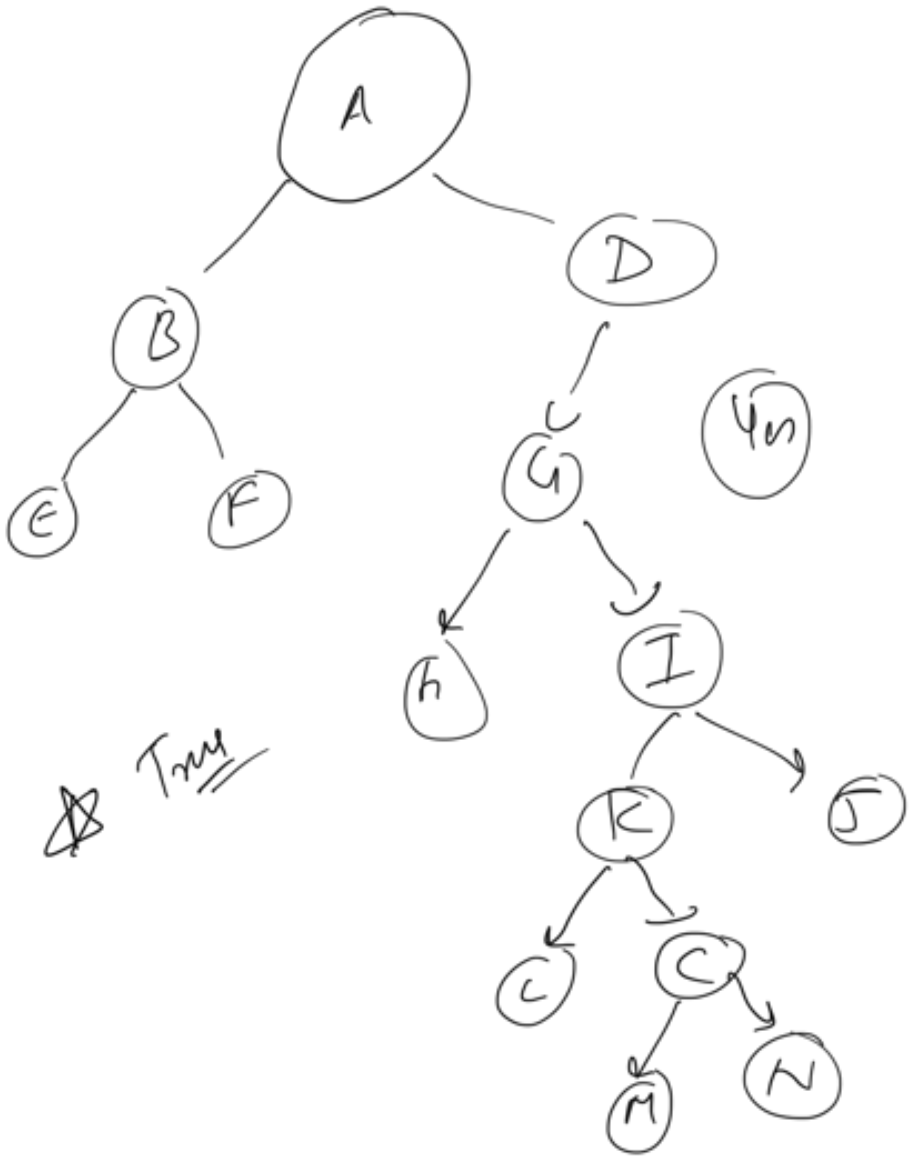
T.C = #state in graph × Time state

O(N) × O(1)

T.C = O(N)

S.C due to RS = O(depth Tree) = O(H)

Q. B.T you want to find if there exists a node whose datavalue = 'c'



Tree

Pre order / Post / In order ≡ T.C = O(N)

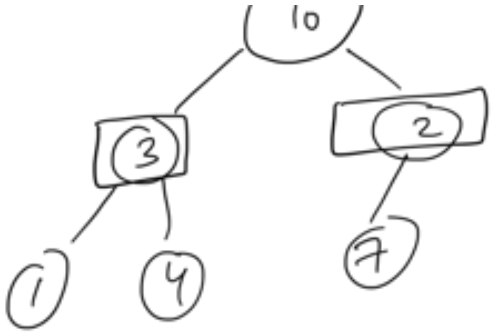
```
boolean checkExistence(TreeNode root)
{
    // sanity check
    if (root == NULL)
        return false;

    // Base Case
    if (root.val == '(')
        return true;

    // Neg B.C
    if (root.left == NULL && root.right == N
        return false

    // rec
    return (checkExistence(root.left)
            ||
            checkExistence(root.right))
}
```

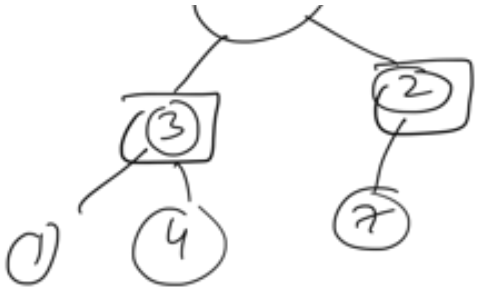
// Positive Base Case

Optional

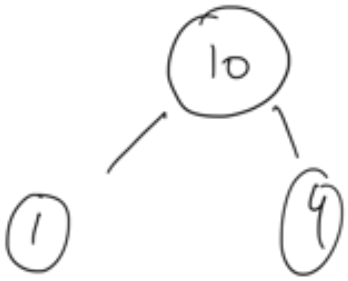
Given two binary trees

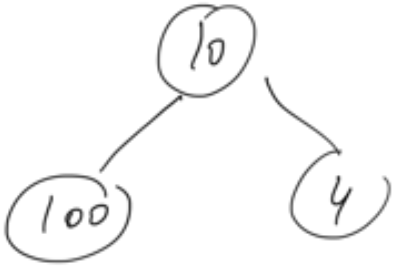
rootA



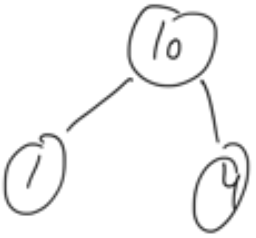


Yes.

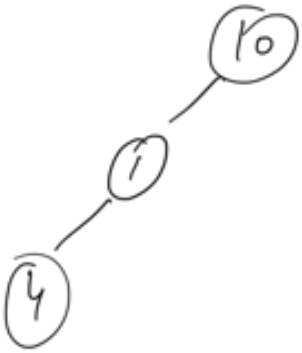




No.

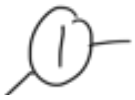


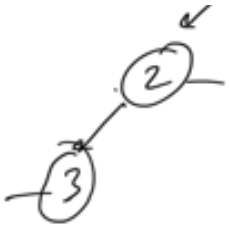
No.



---

* Traversal order is a boiled info in which
* the structure detail get lost
*





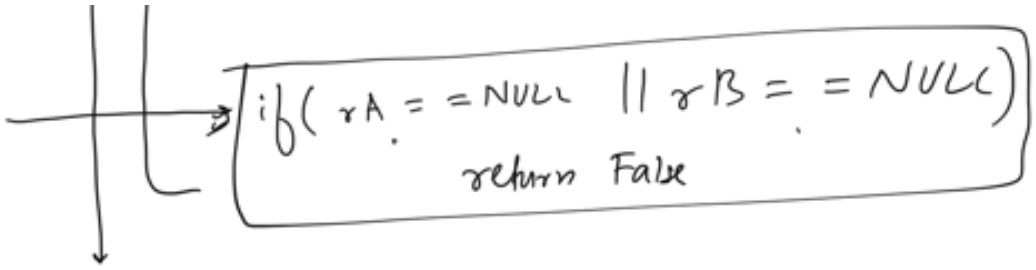
2 3

1, 2, 3

Pre order: 1, 2, 3

Recursion ☺


```
bool isIdentical(TreeNode *A, TreeNode *B)
{
    // sanity check cum B.C
    if( A == NULL and B == NULL)
        return True;
```



// Base Case (Negative Base Case)

if ((rA.data != rB.data)) ✓

return False;

// Base Case (Positive B.C)

if (rA.data == rB.data)

return True

// recursion

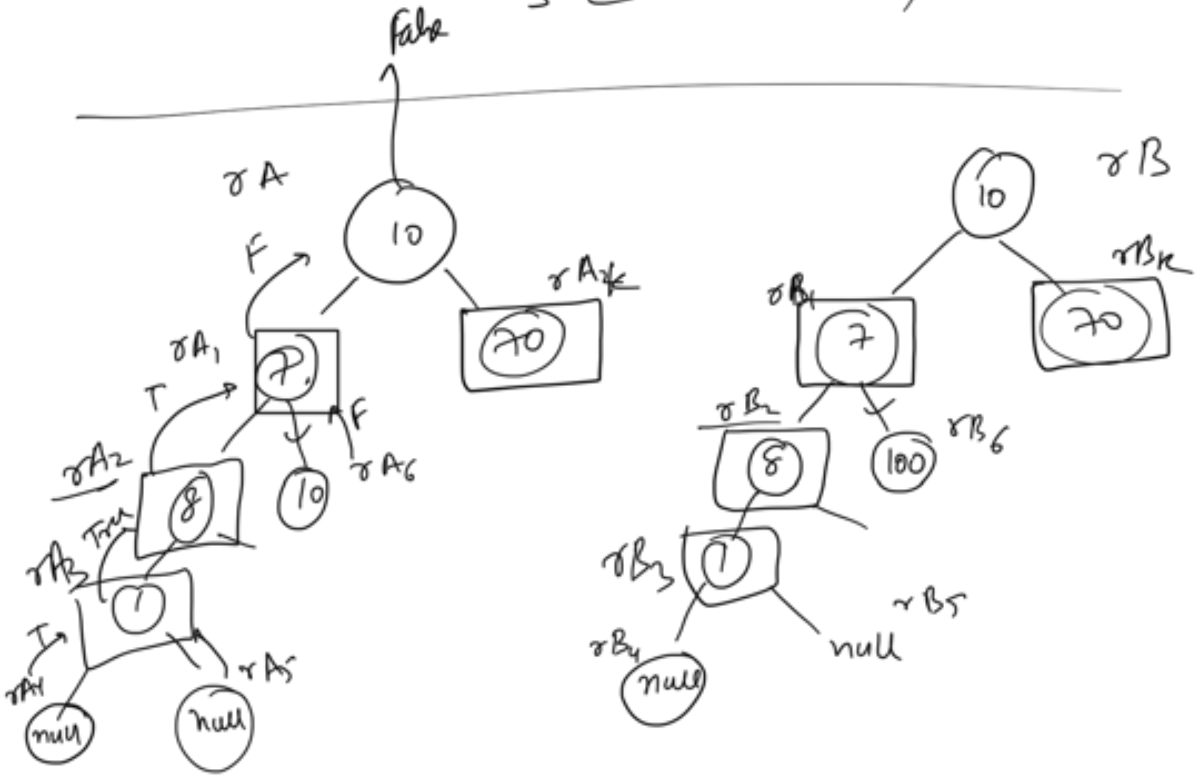
return (isIdentical (rA.left, rB.left) →
and ≡
(isIdent (rA.right, rB.right)
R.S.T

}

T.C = # states × O(1)
O(N)

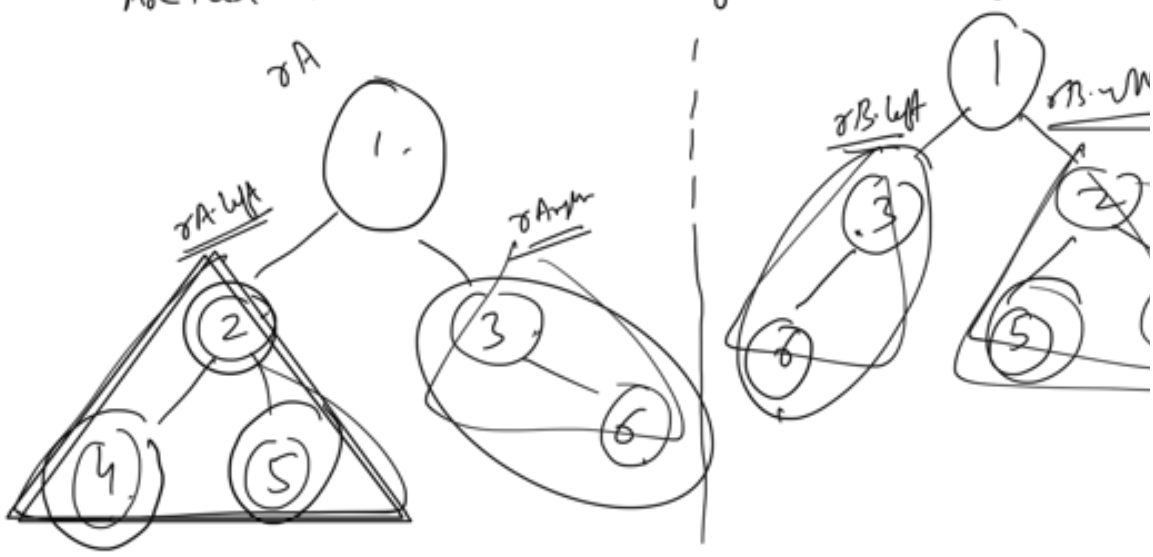
= O(|N|)

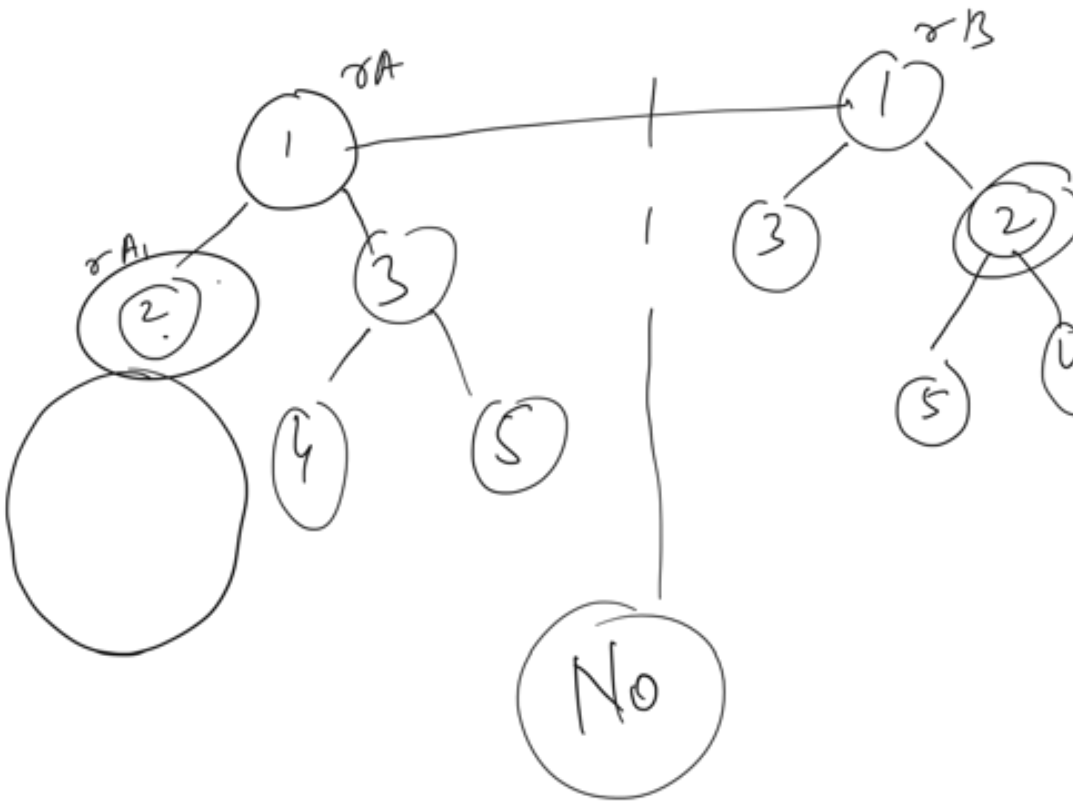
S.C = O(H)



Q Given 2 Trees B.T

Are these two trees mirror images of each other.



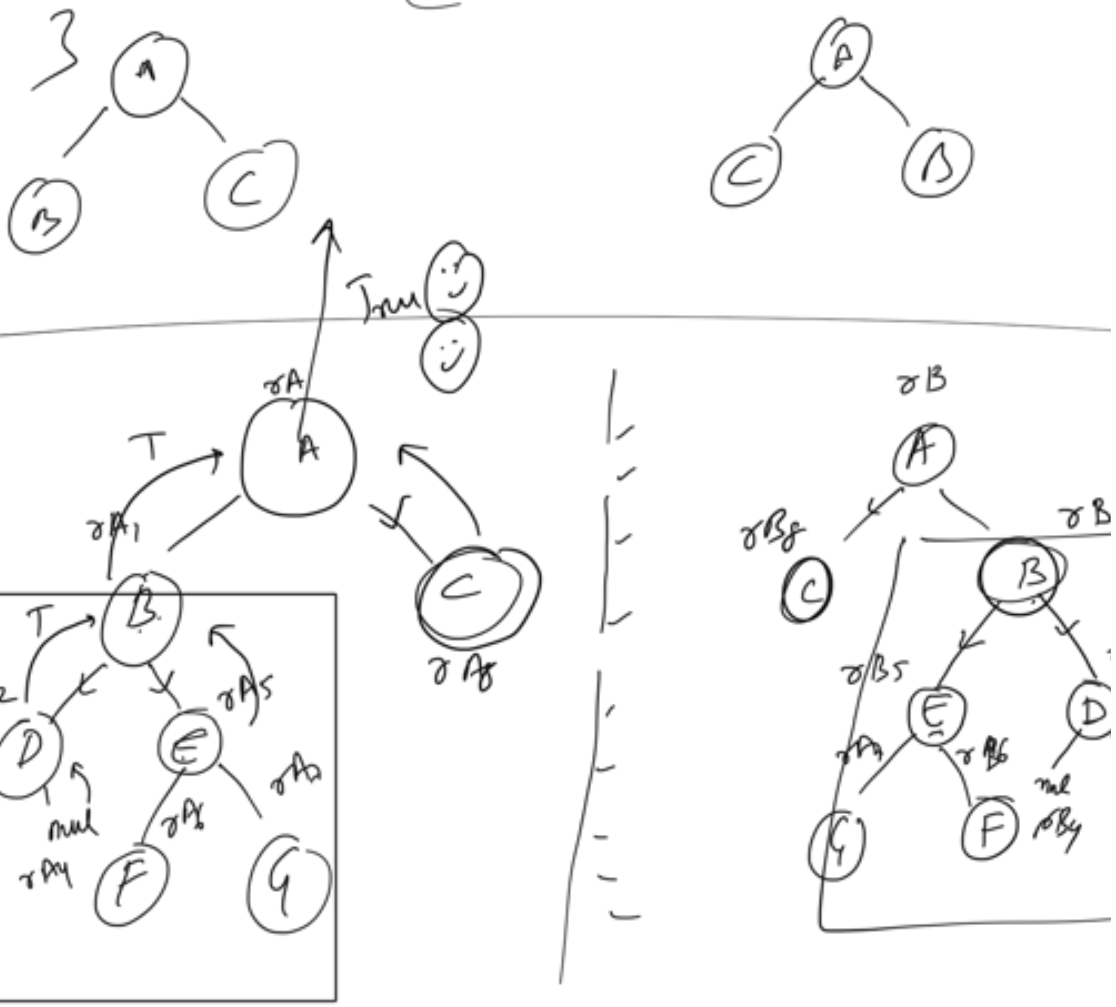


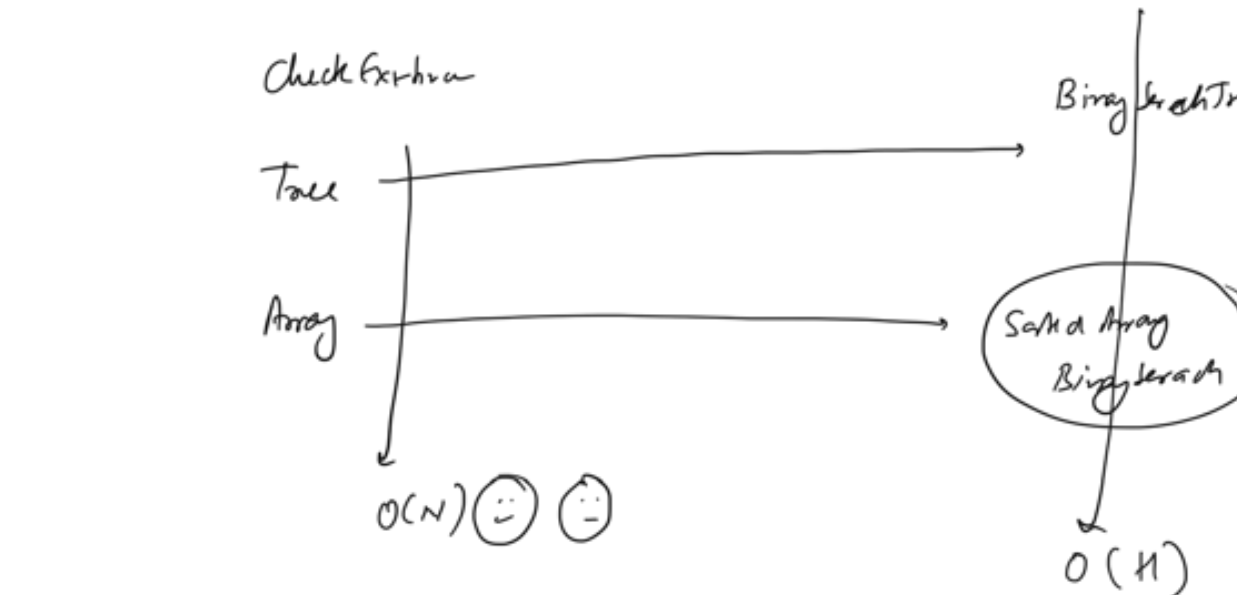

```
bool isMirror (TreeNode rootA, TreeNode rootB)
{
    // sanity check
    if (rootA == null and rootB == null)
        return true;

    if (rootA == null || rootB == null)
        return false

    if ( rootA.val != rootB.val )
        return false
```

// Recurs

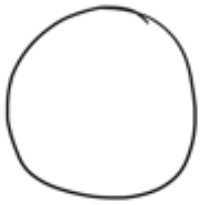
return

isMirror(rootA.left , rootB.right)
and
isMirror(rootA.right , rootB.left)





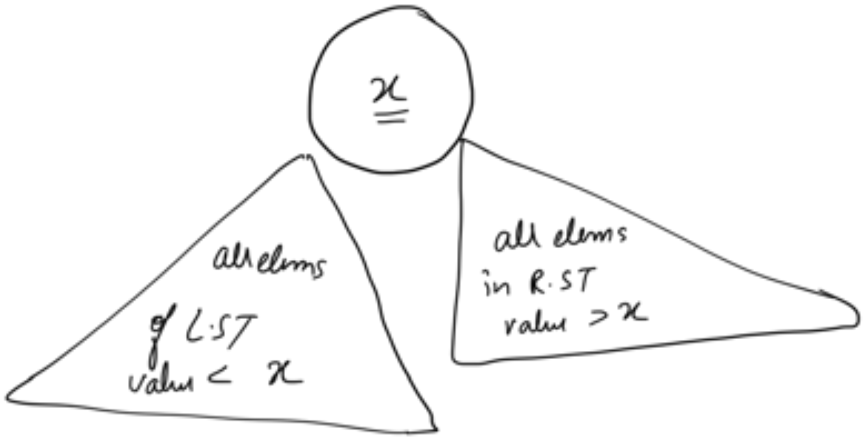
Binary Search Trees

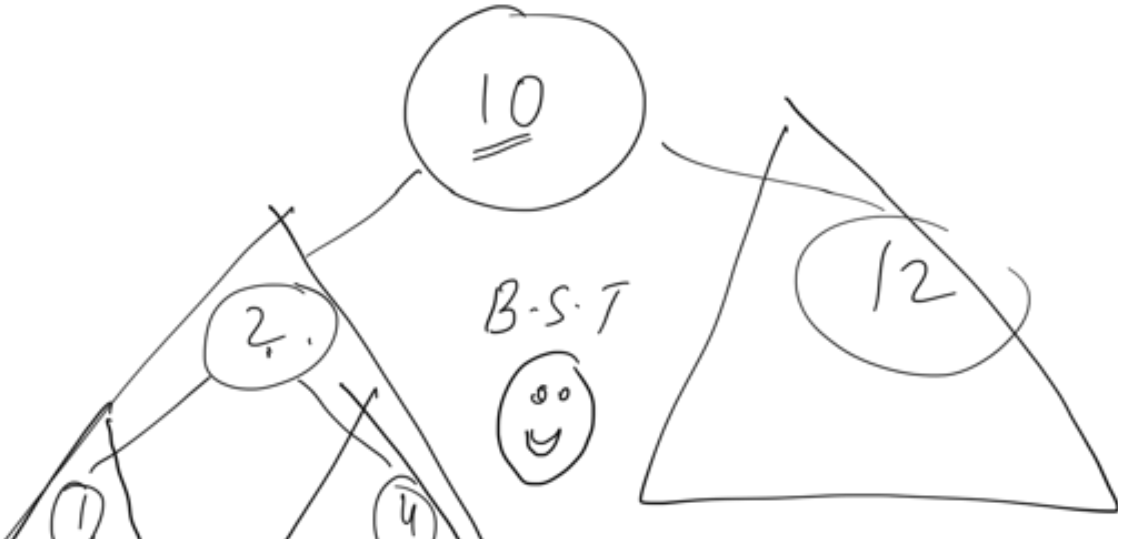


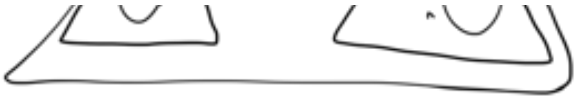
(i) → B.T

(ii) helps in searching

(iii) recursively







Recursion

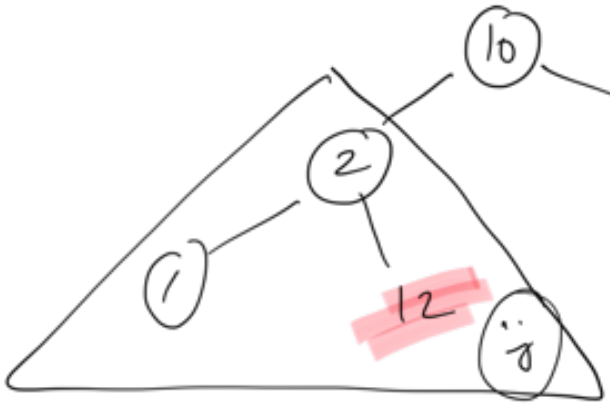
every item in your L.S.T. should be

smaller (or equal to) than your root

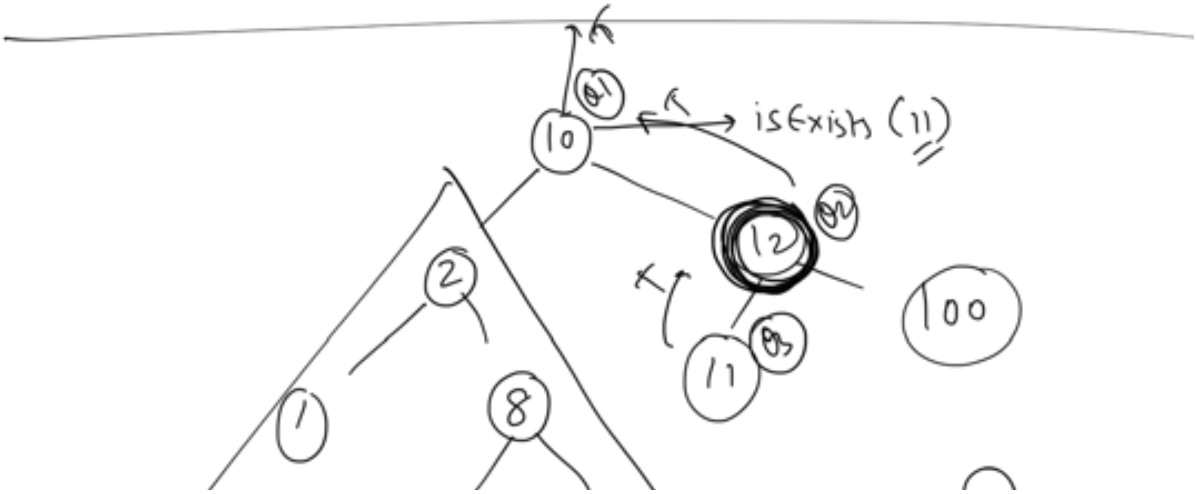
AND

every item in you R.S.T should be

bigger than your root



NOT a B.S.T



T.C = # stack ops × Time for

= O(H) × O(1)

= O(H)

```
bool findInBST (TreeNode root, int val)
{
    if (val == null)
        return false
    if (root == null)
        return false

    if (root.data == val)
        return true;

    // recurs

    if (val > root.data)
        return findInBST(root.right)

    if (val < root.data)
        return findInBST(root.left)
}
```