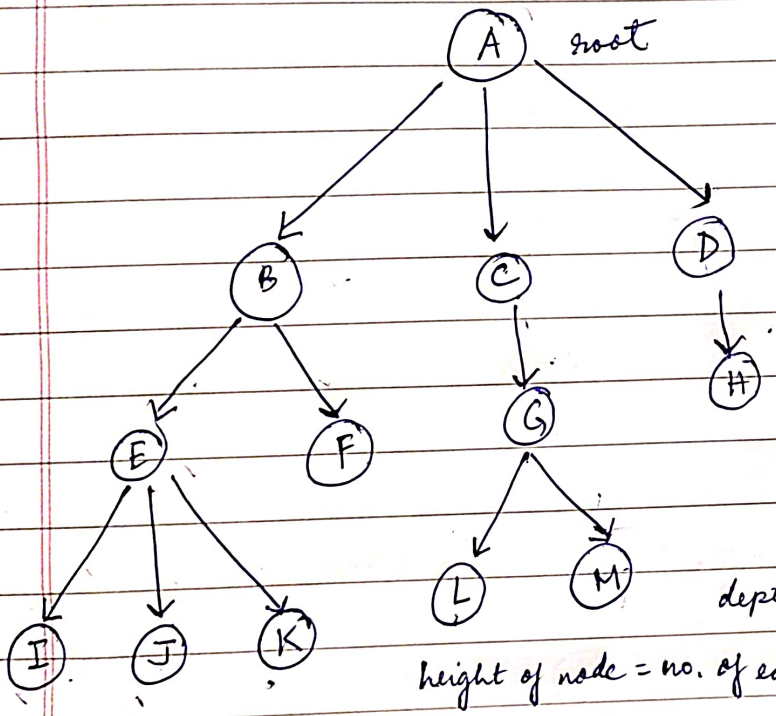
# NON-LINEAR DATA STRUCTURE

Data Structure
- → linear (array, linked list, stack, queue)
- → non-linear (tree, graph)




root = A

B, C, D children of A.

I, J, K, L, M, → leaf node

rest → internal node

non-leaf/internal node
↳ A, B, C, D, E, F, G

degree: no. of children of a node

degree of a tree = max degree among all nodes

depth of node = no. of edges from root to that node

height of node = no. of edges in the longest path from that node to a leaf

Tree can be defined as a collection of entities (nodes) linked together to simulate a hierarchy.

height of tree = no. of edges in the longest path from root to leaf

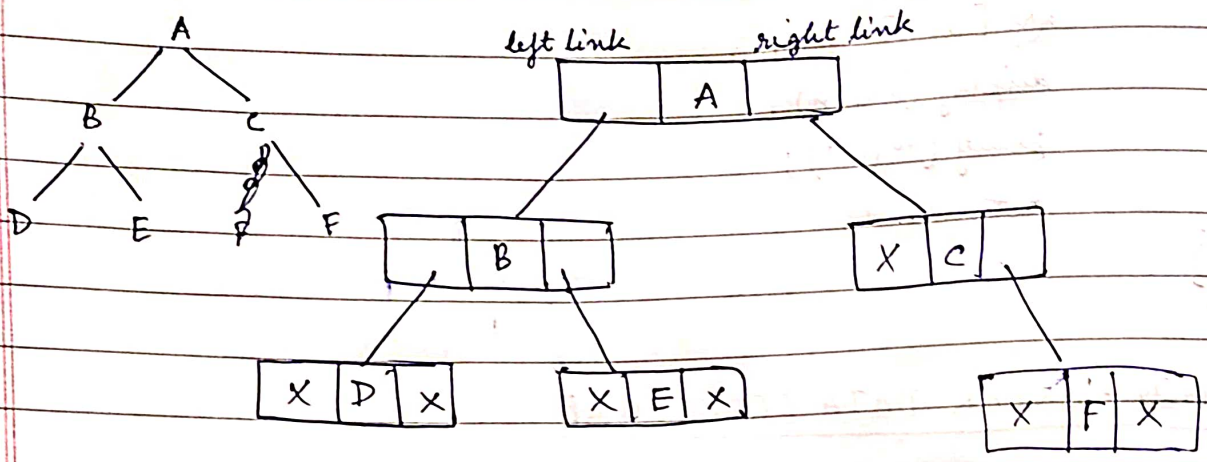
level of node = no. of edges from root to given node = depth of node

level of tree = height of tree = depth of tree = degree of a tree

∴ If there are n nodes, there must be n-1 edges.

depth of tree = max degree among all nodes

BINARY TREE → atmost 2 children




```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

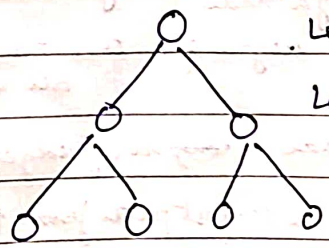
APPLICATIONS

Trees are used to implement the file system, routing protocols.
organizes.

PROS → organizes the data for quick search, as binary search tree, heaptree.

BINARY TREE & ITS TYPES

↳ 0, 1, 2 child / (ren)



| | Maximum children possible |
|---|---|
| Level 0 | $2^0 = 1$ |
| Level 1 | $2^1 = 2$ |
| Level 2 | $2^2 = 4$ |
| ⋮ | ⋮ |
| Level n | $2^n$ |

max no. of nodes possible at any level $i = 2^i$.

maximum no. of nodes for height $h = 2^0 + 2^1 + 2^2 + \ldots + 2^h$

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

min. no. of nodes for height $h$

$= h + 1$

If max. ~~no. of no.~~ & min. no. of nodes given how to collect height?

Max nodes: $n = 2^{h+1} - 1 \Rightarrow n + 1 = 2^{h+1}$

$\Rightarrow \log_2(n+1) = h+1 \log 2 = h+1$

$\Rightarrow h = \log_2(n+1) - 1$ = ~~min. height~~

min no. of nodes = n

Then, $n = h + 1$.

$h = n - 1$

$n_{max}$

Min height = $\log_2(n+1) - 1$

Max height = $n - 1$

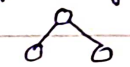
TYPES

- FULL/PROPER/STRICT BT
- COMPLETE BT
- PERFECT BT
- DEGENERATE BT

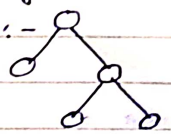
| | Max. nodes | Min. nodes |
|---|---|---|
| BT | $2^{h+1}$ | $h+1$ |
| FULL | " | $2h+1$ |
| ~~DE~~COMPLETE | " | $2^h$ |

| | Min height | Max height |
|---|---|---|
| ~~BT~~ | | |
| BT | $\log_2(n+1)-1$ | $n-1$ |
| FULL | " | ~~$\log_2$~~ $\frac{(n-1)}{2}$ |
| COMPLETE | " | ~~$\log_a(n+1)-1$~~ $\log_2 h$ |

FULL BT → 0/2 children, or each node will have exactly 2 children except for leaf node. eg:-



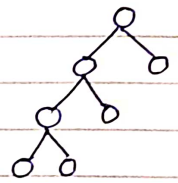
, or



~~no. of~~

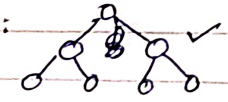
no. of leaf nodes = no. of internal nodes + 1

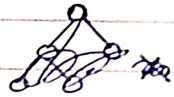
for min nodes →

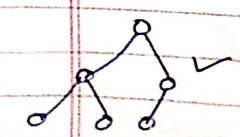


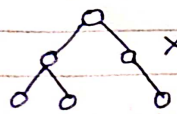

$2h+1$

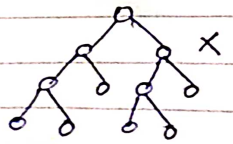
COMPLETE BT → all levels are completely filled (except possibly ~~for~~ the last level (leaf nodes) and the last level has nodes as left as possible. eg:

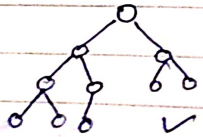


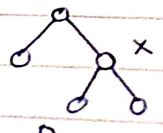


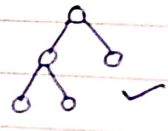


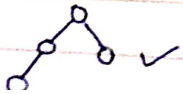










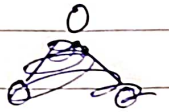


↳ some place it is written nearly complete

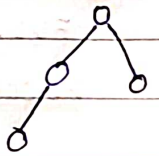
Calculate min nodes for complete BT :-

Height

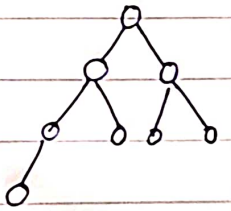
0 — 0 — 1 — Total min nodes = $2^0$



— 1 — $2 = 2^1$



— 2 — $4 = 2^2$



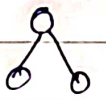
— 3 — $8 = 2^3$

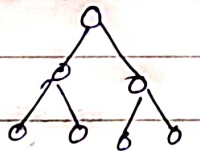
⋮

h — $2^h$

Total min nodes if the min. criteria was considered

3. PERFECT BT — all internal nodes have 2 children & all leaves are at same level.

eg: 



or, maximum node case of.

FULL BT = PERFECT BT.

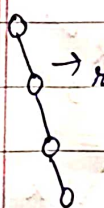
~~max~~ no. of nodes = $2^{h+1} - 1$.

min height = $\log_2(n-1) + 1$

Every PERFECT BT can be FULL & COMPLETE BT but nice versa (X).

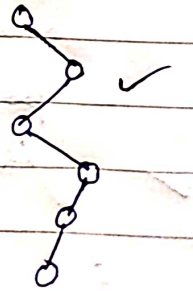
4. DEGENERATE BT — all the ~~n~~ internal nodes have 1 child/

eg:  → also called left skewed BT



→ right skewed BT

or



~~no. of nodes =~~

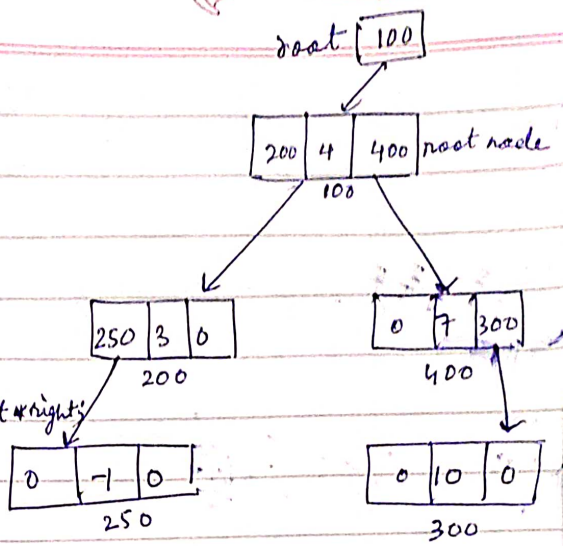
no. of nodes = h + 1

height if nodes given = n − 1

# BINARY TREE IMPLEMENTATION

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

```
struct node
{
    int data;
    struct node *left, *right;
};
```

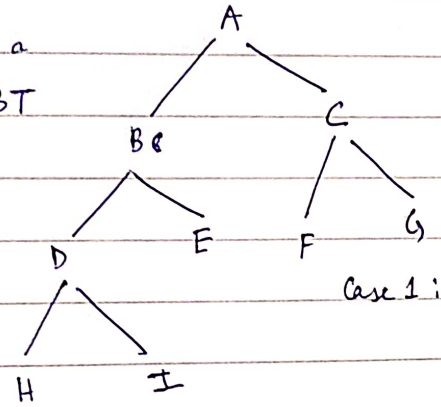



~~void main()~~

{

## ARRAY REPRESENTATION OF BST

should be a complete BT




Fill it levelwise from left to right

Case 1:

| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Case 2:

| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Case 1: If a node is at $i^{th}$ index:-

left child would be at - $[(2*i)+1]$

Right child " " " - $[(2*i)+2]$

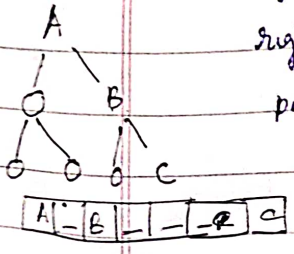
Parent " " " - $\left[\frac{i-1}{2}\right]$ → take lower bound

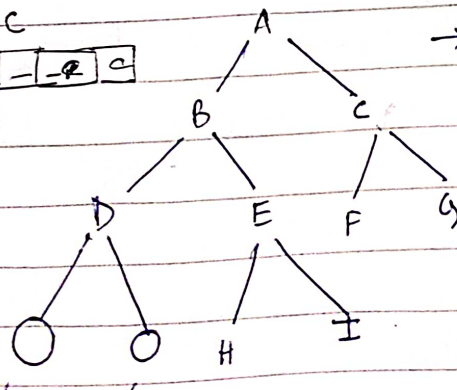
Case 2: If a node is at $i^{th}$ index:-

left child would be at - $(2*i)$

right child " " " - $(2*i)+1$

parent " " " - $\left[\frac{i}{2}\right]$ → take lower value




→

| A | B | C | D | E | F | G | - | - | H | I | Case 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |

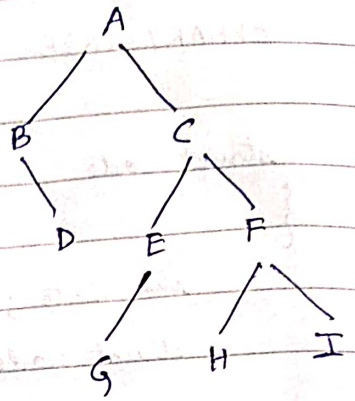
| A | B | C | D | E | F | G | - | - | H | I | Case 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |

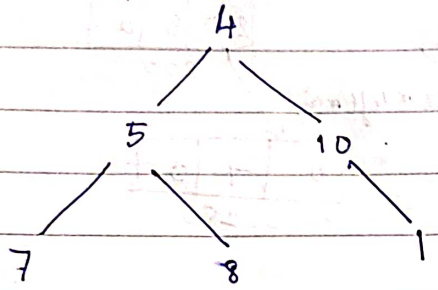
insert NULL for empty child.

left-root-right In order : ~~D B A G E H F I~~ ~~D B A G E C H F I~~ B D A G E C H F I

left right root Post order : D B G E H I F C A .

root left right Pre order : A B D C E G F H I




Inorder : 7 5 8 4 10 1

Pre order : 4 5 7 8 10 1

Post order : 7 8 5 11 10 4

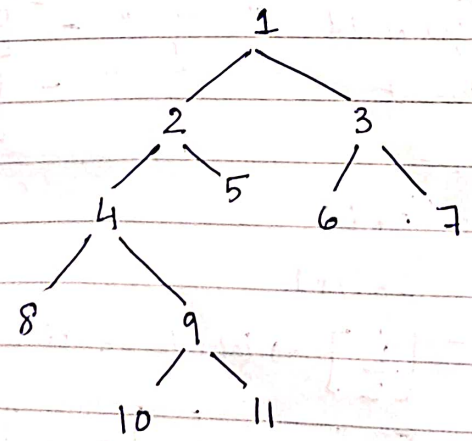
CONSTRUCT A ~~PREORDER~~ BINARY TREE FROM PREORDER & INORDER

Root → next root → then next root

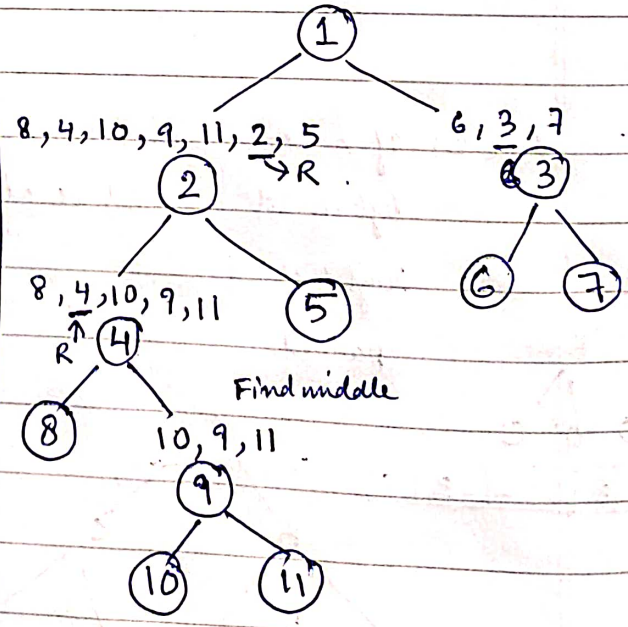
~~PRE~~ PREORDER : 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 (Root L. R)

~~POST~~IN ORDER : 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 (L. Root R)

L — NR — Root — R




Working



## CONSTRUCT A BINARY TREE FROM POST ORDER & INORDER.

~~POST ORDER : 9, 1, 2, 12, 7, 5, 3, 11, 4, 8.~~

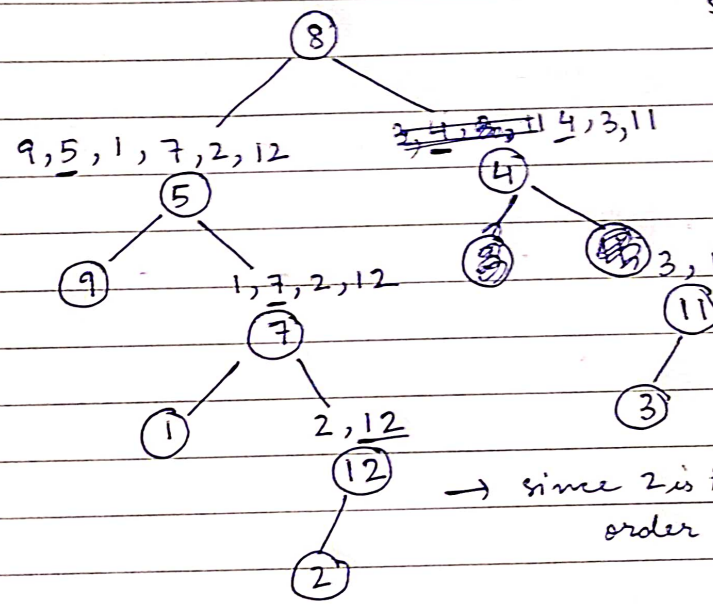
left root  right root  1st Root

POST-ORDER : 9, 1, 2, 12, 7, 5, 3, 11, 4, 8 — (L-R-Root)

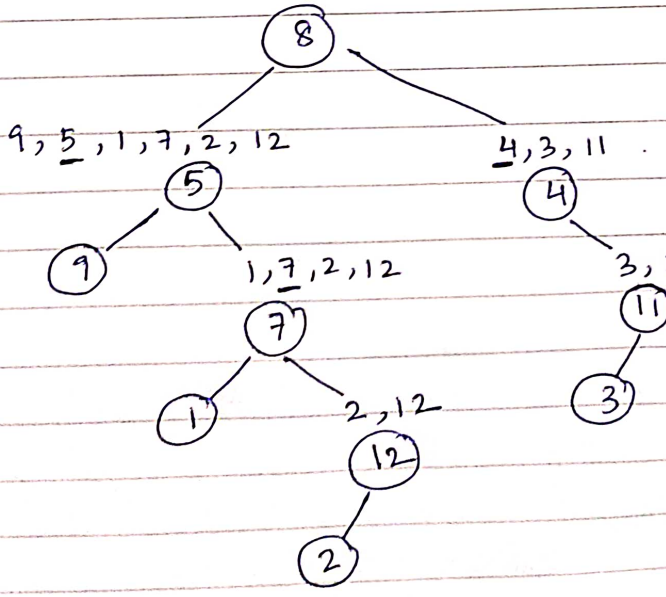
IN-ORDER : 9, 5, 1, 7, 2, 12, 8, ~~3, 4, 3,~~ 11 — (L-Root-R)

Scan right to left in post order to find the root

~~In order; for 3 integers middle element is the root.~~




→ since 2 is to the left of 12 in order.



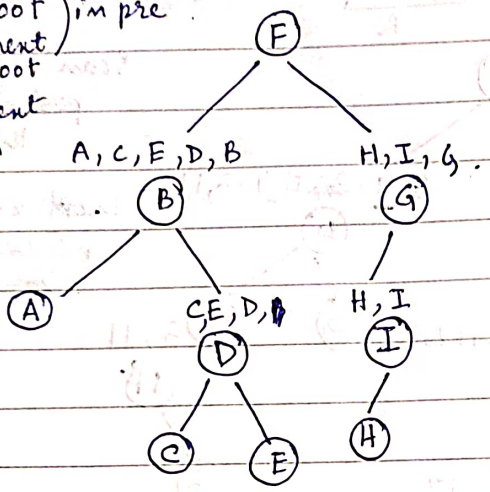

11 comes before 3 in Post order; 3 is to the left of 11

# CONSTRUCT BINARY TREE FROM PREORDER AND POSTORDER TRAVERSAL

PRE-ORDER :- F B A D C E G I H (Root-L-R) possible to construct a unique Full
POST-ORDER :- A C E D B H I G F (L-R-Root) BT

but not possible to construct a unique BT.

1st root then next left root find next root in post ) in pre



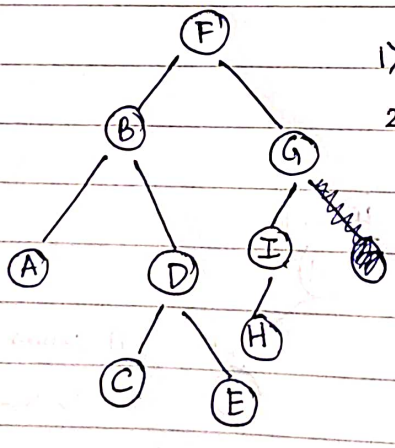

Pre – B A D C E
Post – A C E D B

Pre – D C E
Post – C E D

Pre : G I H
Post : H I G

Pre : I H
Post : H I

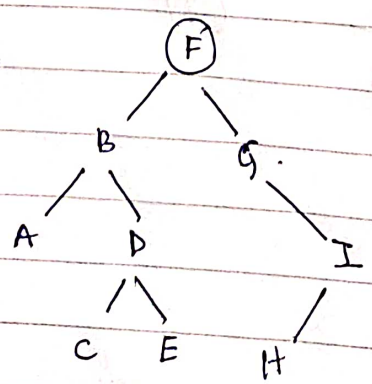
Trick.




1> Scan pree order left to right. Maybe ~~later~~ letter B.
2> ~~So~~ B is to the left of F in Post.

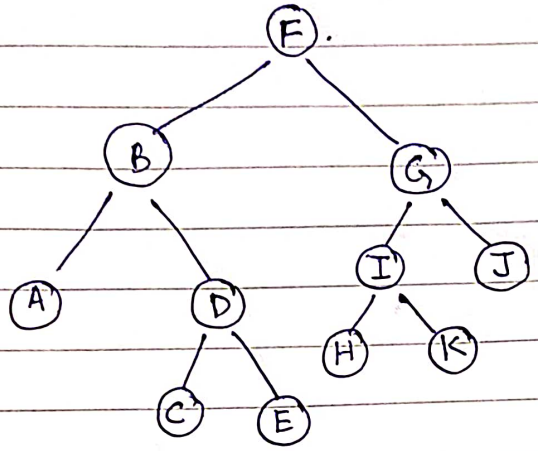
Not unique.

Exception :



Pre : F B A D C E G I H K J (Root L R)

Post : A C E D B H K I J G F (L R Root)




unique full BT.
either 0 or 2 children.

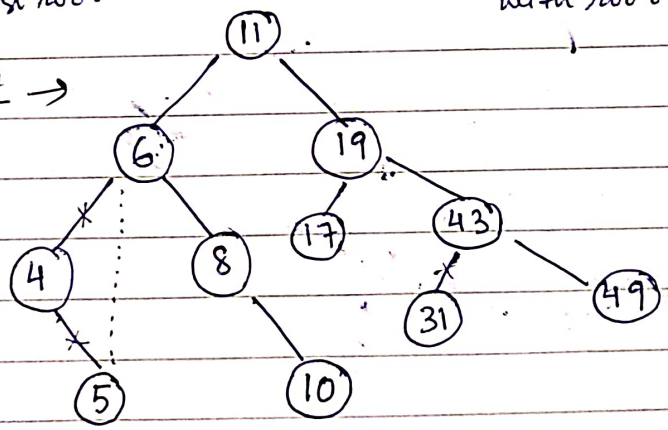
DRAW BINARY TREE BY INSERTING THE FOLLOWING NUMBERS FROM LEFT TO RIGHT.

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

↑ 1st root

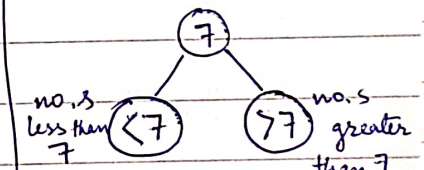
Scan left to right & compare with root

Insertion →

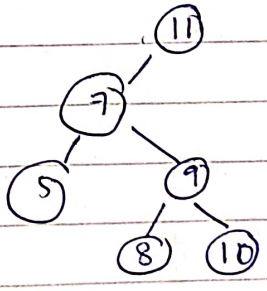



ex: 31 → i) 31 > 11 so right subtree
ii) 31 > 19 " " "
iii) 31 < 43 " left " .

0, 1, 2 .




eg :




~~Del~~ Deletion. →

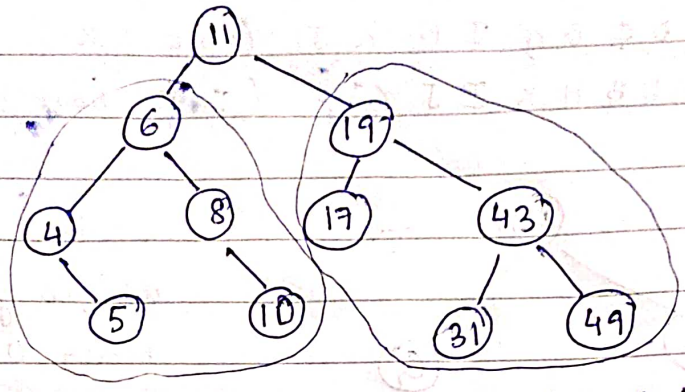
i) 0 child
ii) 1 child
iii) 2 children.

i) 0 child - eg - 31 has 0 child, so delete 31 or free the node.

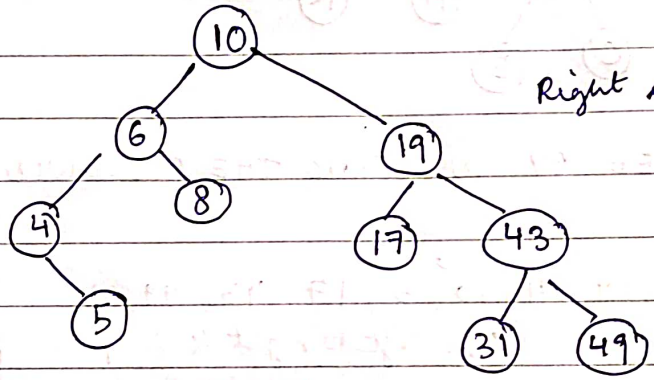
ii) 1 child - eg - 4 has 1 child, 4 will be replaced by 5 or any left/right subtree.

iii) 2 children - eg - 11 has 2 children → replace with inorder predecessor
↳ or replace with " successor.

Deletion :- eg 11- with 2 children.

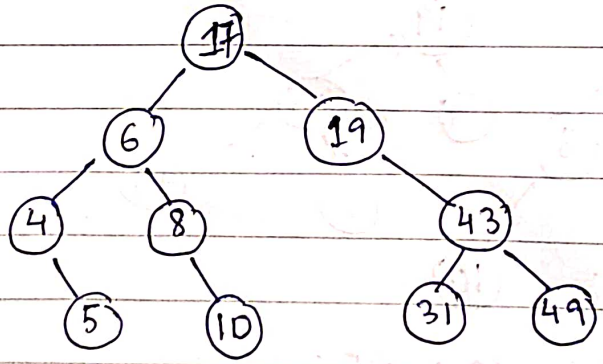



i) INORDER PREDECESSOR: ~~eg #~~ largest integer in left subtree.
here, 10.




Right subtree will be same.

ii) INORDER SUCCESSOR: ~~Large~~ smallest element in the right subtree.




Inorder would always be in sorted order.

4, 5, 6, 8, 10, 11, 17, 19, 31, 43, 49.

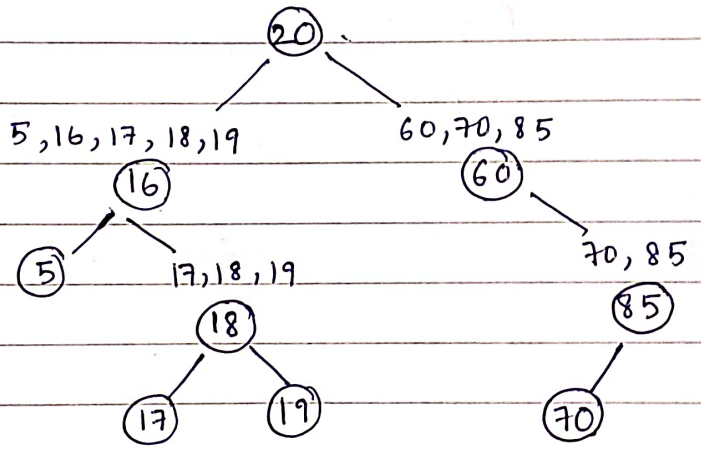
↑ inorder predecessor

↑ inorder successor.

CONSTRUCTION OF ~~BINARY~~ BST WHEN ONLY PREORDER/POSTORDER IS GIVEN.

Given, 1. PREORDER :- 20, 16, 5, 18, 17, 19, 60, 85, 70. (Root - L - R)

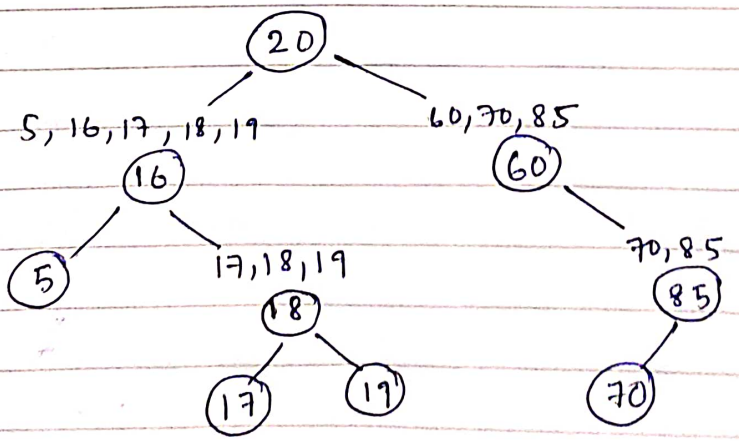
INORDER :- 5, 16, 17, 18, 19, 20, 60, 70, 85 (L - Root - R)
(Traversal will be in ascending order for Binary Search tree)




---

Given., 2. POST ORDER - ~~17, 19, 18~~ 5, 17, 19, 18, 16, 70, 85, 60, 20 (L-R-Root)
R

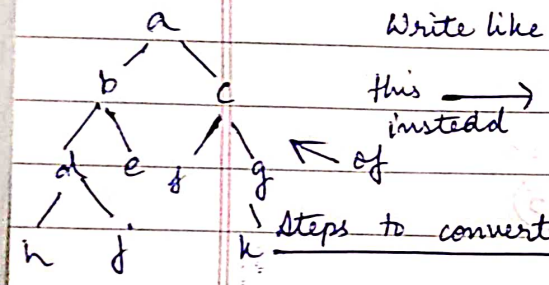
INORDER - 5, 16, 17, 18, 19, 20, 60, 70, 85. (L-Root-R)
R



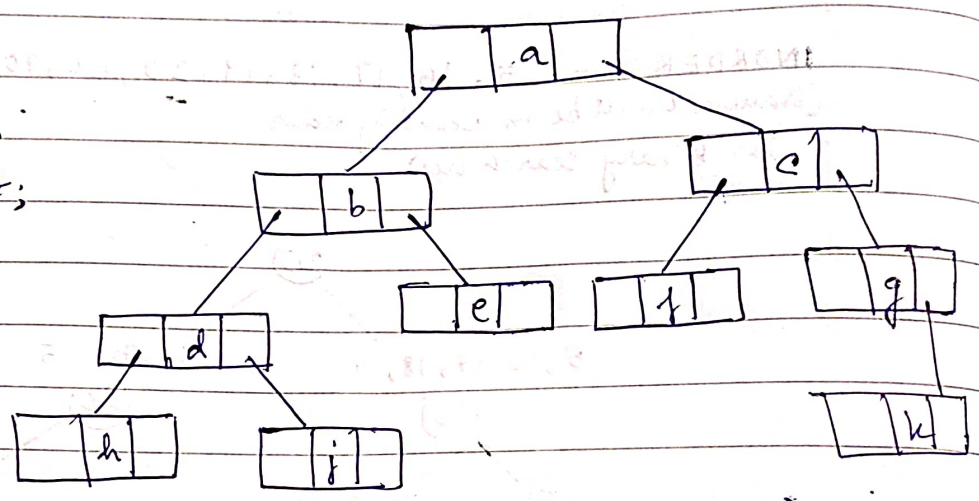
# THREADED BINARY TREE

→ we write the full node

```
struct node
{
    int data;
    struct node * left;
    struct node * right;
}
```




Write like this instead of




Steps to convert.

1. ~~Keep~~ Keep the leftmost and the rightmost NULL pointers as 'NULL'.
2. Change all other NULL pointers as →
   - Left pointer = Inorder Predecessor
   - Right pointer = Inorder successor.