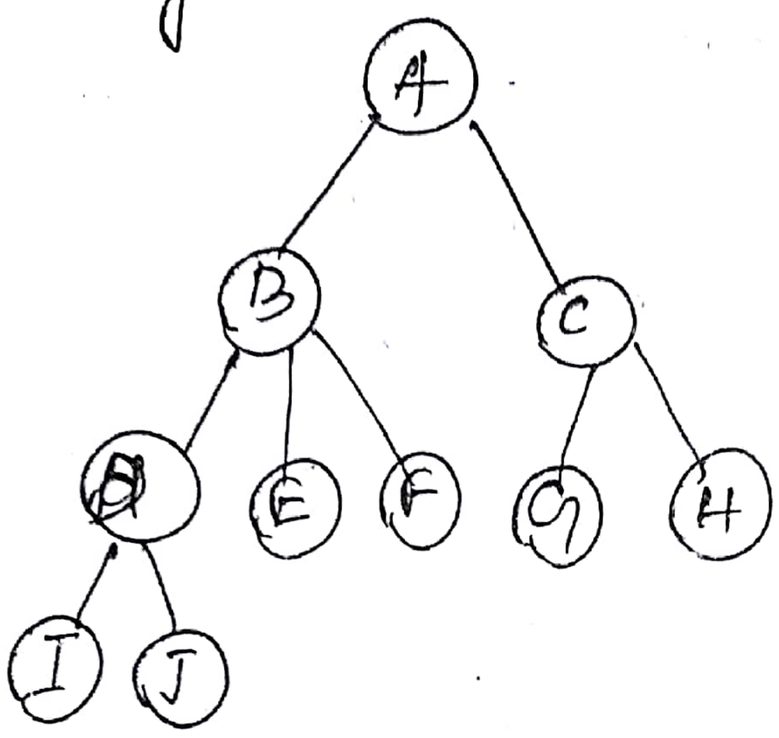
IKEE

Demerit of linked list :-
> It is a linear DS. Every node has information of next or previous node only. So the searching in linked list is sequential which is very slow of O(n)
> So use of tree will help in fast retrival & searching of $O(\log_2 n)$

Def² of tree :- > Tree is a non-linear datastructure
> Tree is an acyclic simple connected graph, contains no loop and no cycles.
> There is no more than one edge between any pair of nodes.
> Tree is used to represent hierarchical relationship existing among several data elements.



TREE TERMINOLOGY :-
(1) NODE, (2) Root, (3) Edge, (4) Parent NODE, (5) Child Node (6) Leaf/Terminal Node, (7) Level, (8) Height/depth (9) sibling (10) Path, (11) Ancestor & Descendent, (12) subtree. (13) Degree. (14) Forest.

(1) Node :- Each element of tree is called a node. In fig, 10 nodes are present.

(2) Root :- It is a specially designed node that does not have any parent. It is the first node in the hierachical arrangement.

(3) Edge :- Edge is a connecting link of two nodes

(4) Parent Node :- Parent of node is the immediate predecessor of a node. In fig B is parent of D, E, F.

(5) Child Node :- All the immediate successor of node are called child node. In the fig; G, H are children of E.

(6) Leaf / Terminal Node :-

A node that does not have any child is called leaf node or terminal node. Except leaf nodes all nodes in tree are non terminal nodes.

* (7) Level :- Level of any node is defined as the distance of that node from the root.
> Root node is at distance 0 from itself so it is at level 0.
> Then, its immediate children are at level 1, and their immediate children are at level 2 and so-on up to the terminal nodes.
> If a node is at level n then its children will be at level n+1.

(8) Depth : The depth of node n_i is the length of the unique path from the root to n_i.
> Thus root is at depth 0.
> In fig: depth of B is 1, H is 2.

(9) Height :- The height of node n_i is the length of the longest path from n_i to a leaf. Thus all leaves are at height 0.
> The height of root is the height of tree.

* (10) Path :- Path is a sequence of consecutive edges from source node to destination node.
> In fig the path between A to J is given by node pairs (A, B), (B, D), (D, J)

(11) Siblings :- Two or more nodes which have same parent is called brothers or siblings.
> All siblings are at same level, but it is not necessary that nodes at same level are siblings.

(12) Ancestor & descendant :-
Any nodes Na is said to be the ancestor of node Nm if node Na lies in the unique path from root node to the node Nm.

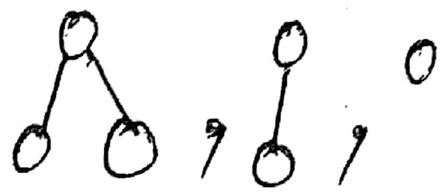
Ex:- B. is the ancestor of I, J
y of Na is the acestor of Nm then Nm is the descendem of node Na.

(13) Subtree :- A tree may be devided into subtree which can further be devided.
> The 1st node of the subtree is called root of the subtree.

(14) Degree :- The number of subtree or children of a node is called its degree.
> ex fig degree Node B is 3, C is 2
> degree of leafnode is 0.

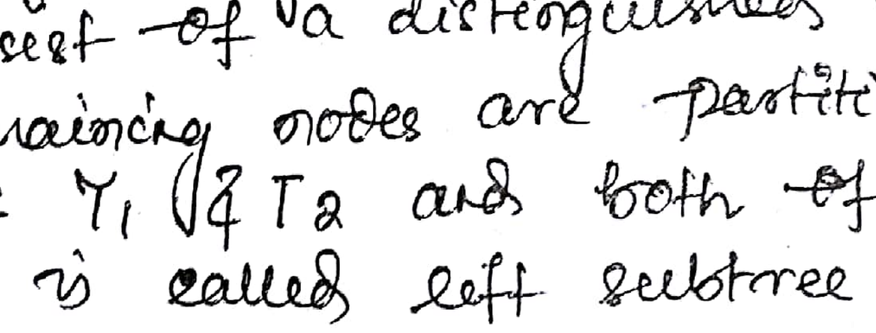
(15) Forest :- A forest is a set of n disjoint tree where n>=1 if the root of a tree is removed we get a forest consisting of its subtree..

Binary Tree :-
> In a binary tree no node can have more than two children ie node can have 0,1, two children.

Ex :- 

> A binary tree is a finite set of nodes that is
i) either empty
ii) consist of a distinguished node called root and remaining nodes are partitioned into two disjoint sets T1 & T2 and both of them are binary tree.
iii) T1 is called left subtree
iv) T2 is called right subtree.

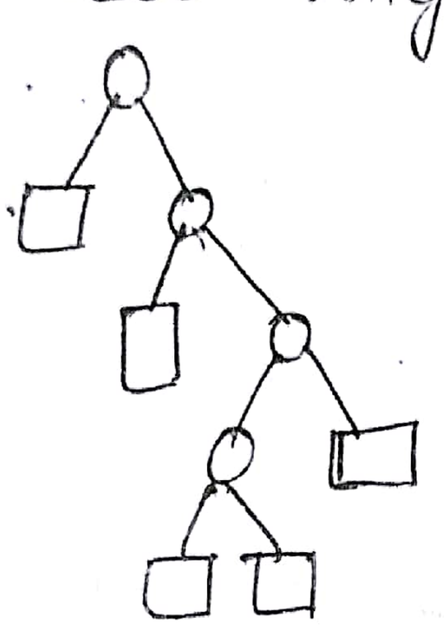
Strictly binary tree :-
A binary tree is a strictly binary tree if each node in the tree is either 0 ~~leaf node~~ or has exactly two children. ie there is no node with one child.

Ex:- 

Note:- A strictly binary tree with 'n' leaves always contains 2n-1 ~~leaves~~ nodes.

Extended binary tree :-
In a binary tree each empty subtree is replaced by a special node the resulting tree is extended binary tree.

> We can convert a binary tree to an extended binary tree by adding special nodes to leaf nodes & nodes that have only one childs.



Note :

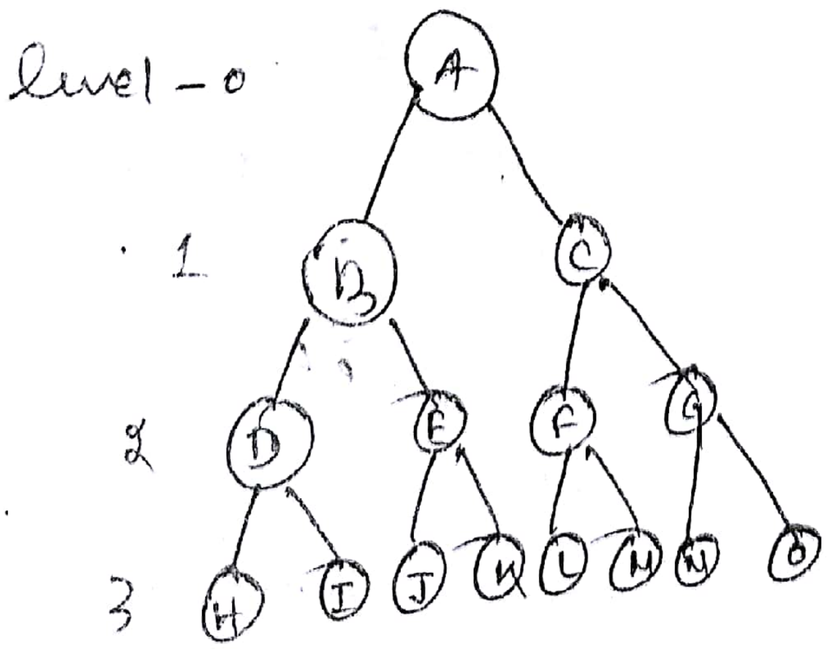
> The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its parent.

> Depth of a binary tree is the maximum level of any leaf in the tree. This equals the length of the longest path from the root to any leaf.

Complete binary tree :- (Full binary tree)

> A complete binary tree is a binary tree in which all leaf nodes / terminal nodes present at same level.

> All nonterminal node must contain two children.



> if binary tree contains n nodes at level l then it contains at most 2^n nodes at level $l+1$

> The binary tree contain one node at level 0, ie 2^0 two node at level 1, ie 2^1 and so.
So at level 'l' it contains at most 2^l nodes.

~~A complete binary tree of depth 'd' contain exactly 2^d nodes (as depth tree is the maximum level)~~

let T_n = Total number of nodes in a complete binary tree.

$$= 2^0 + 2^1 + 2^2 \ldots\ldots + 2^d$$

$$= \sum_{i=0}^{d} 2^i$$

{ formula G.P $\frac{a(r^n-1)}{r-1}$ $a=1$, $r=2$, $n=d+1$

$$= \frac{1(2^{d+1}-1)}{2-1}$$

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

As all leaves nodes in such a tree are at level d, contains 2^d leaves & non-leave node or terminal nodes are $2^d - 1$.

And total is $2^d + 2^d - 1 = 2^{d+1} - 1$

If the number of nodes, t_n, in a complete binary tree is known, we can compute the depth d, from the equation $t_n = 2^{d+1} - 1$ (put $t_n = n$)

$\Rightarrow n = 2^{d+1} - 1$

$\Rightarrow n+1 = 2^{d+1}$

$\Rightarrow \log(n+1) = d+1$

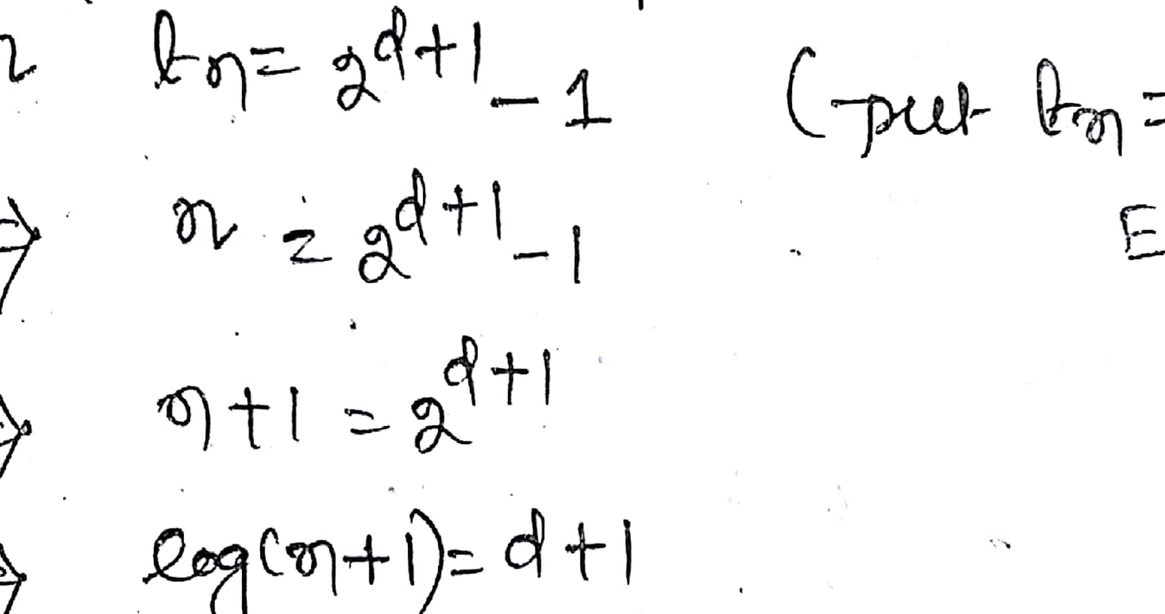
$\Rightarrow \boxed{d = \log(n+1) - 1}$

Ex — on a complete binary tree if total no. of nodes is 31, then calculate the depth.

$d = \log(31+1) - 1$

$= \log_2 32 - 1$ level/depth

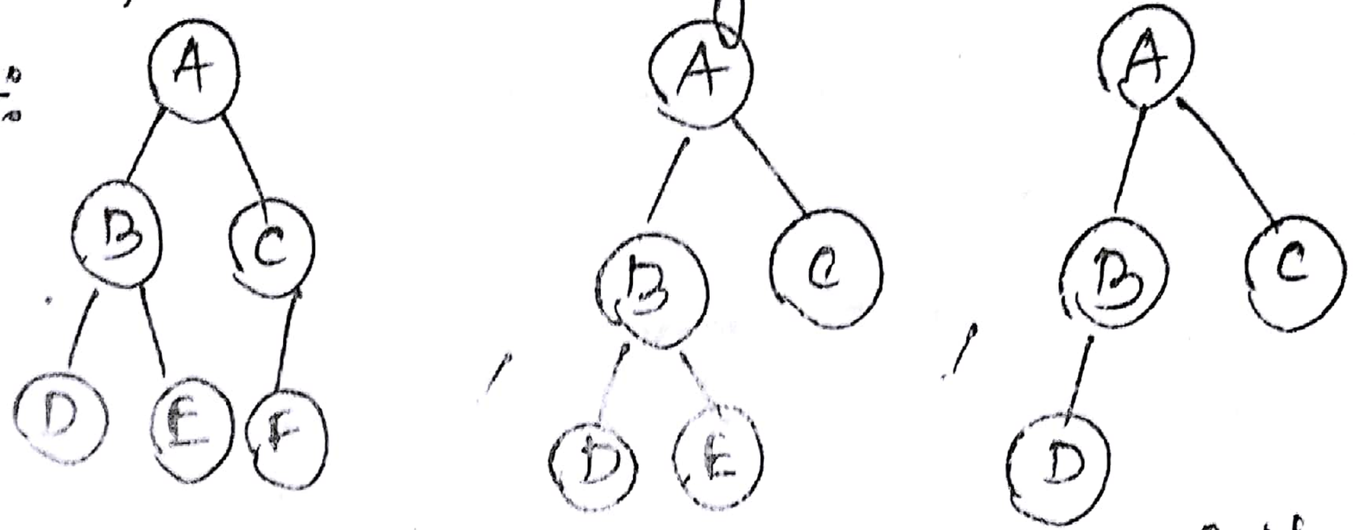
$= 5 - 1 = 4$



(7 Maximum no. of nodes on any level i is 2^i, $i \geq 0$)

Almost complete binary tree :-
1) Nodes are filled in the next level if the present/parent level is filled.
2) on the new level, the order of filling of nodes is from left to right.

Ex :-



Representation of Binary tree :- It can be represented
1) Array representation
2) Linked list representation

(1) Array Representation :-

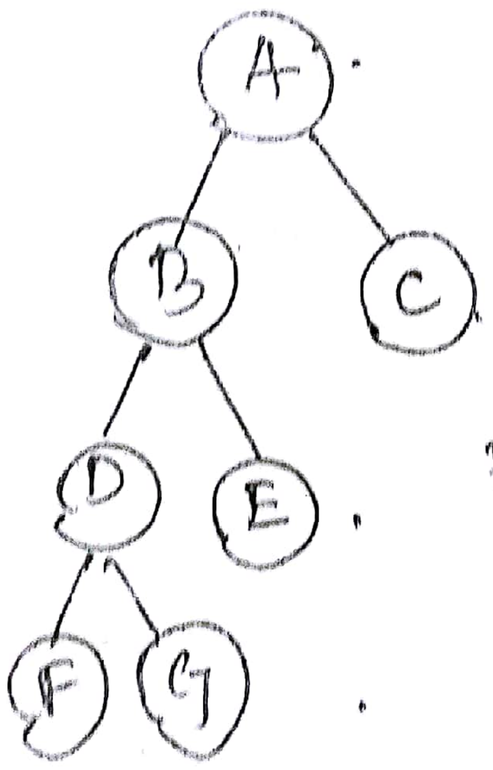
An array can be used to store the nodes of ~~array~~ a binary tree. The nodes stored in an array are accessible sequentially

~~Size of array is the Maximum no. of nodes~~

Root is always stored at index 0, Then in successive memory location the left/right child will be stored

If a node occupies tree[i], Then its left child is stored in tree[2i+1], its right child is stored in tree[2i+2]

Parent is stored at $\lfloor (i-1)/2 \rfloor$



⇒

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

Ex :- left child of (B) = $2 \times i + 1 = 2 \times 1 + 1 = 3$

left child of (D) = $2 \times i + 1 = 2 \times 3 + 1 = 7$

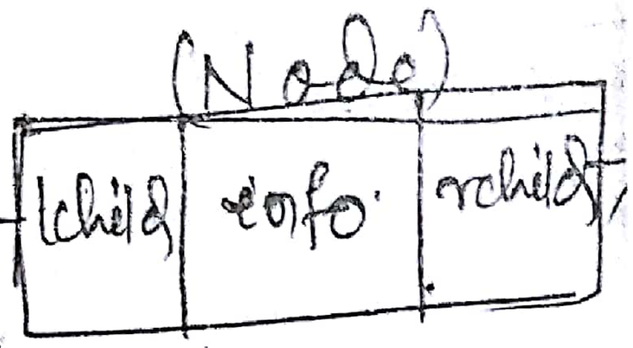
right child of (B) = $2 \times i + 2 = 2 \times 1 + 2 = 4$

Parent of (D) = $\lfloor (i-1)/2 \rfloor = \lfloor (3-1)/2 \rfloor$

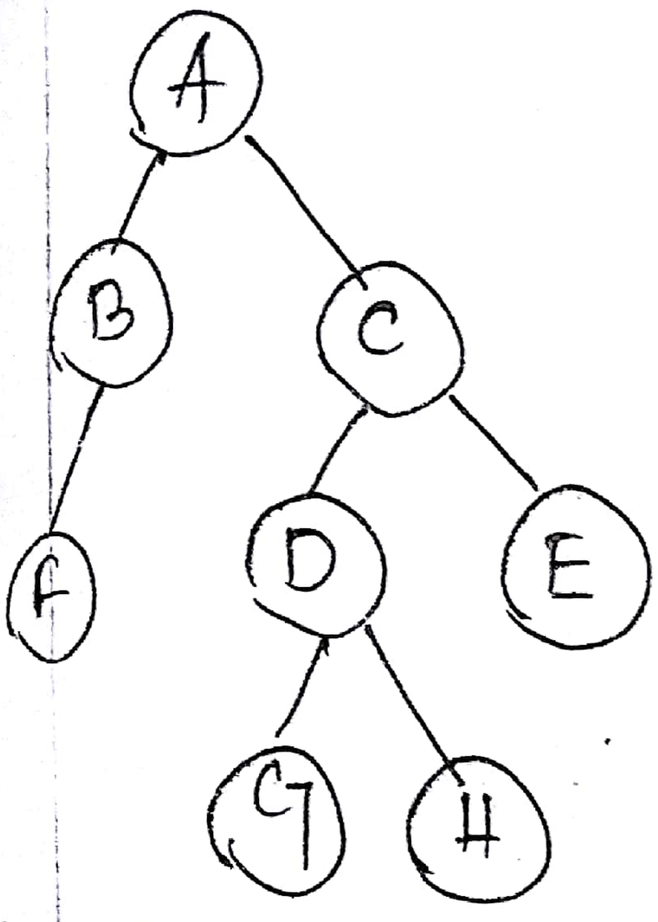
91 Parent of (E) = $\lfloor (i-1)/2 \rfloor = \lfloor 3/2 \rfloor = 1$

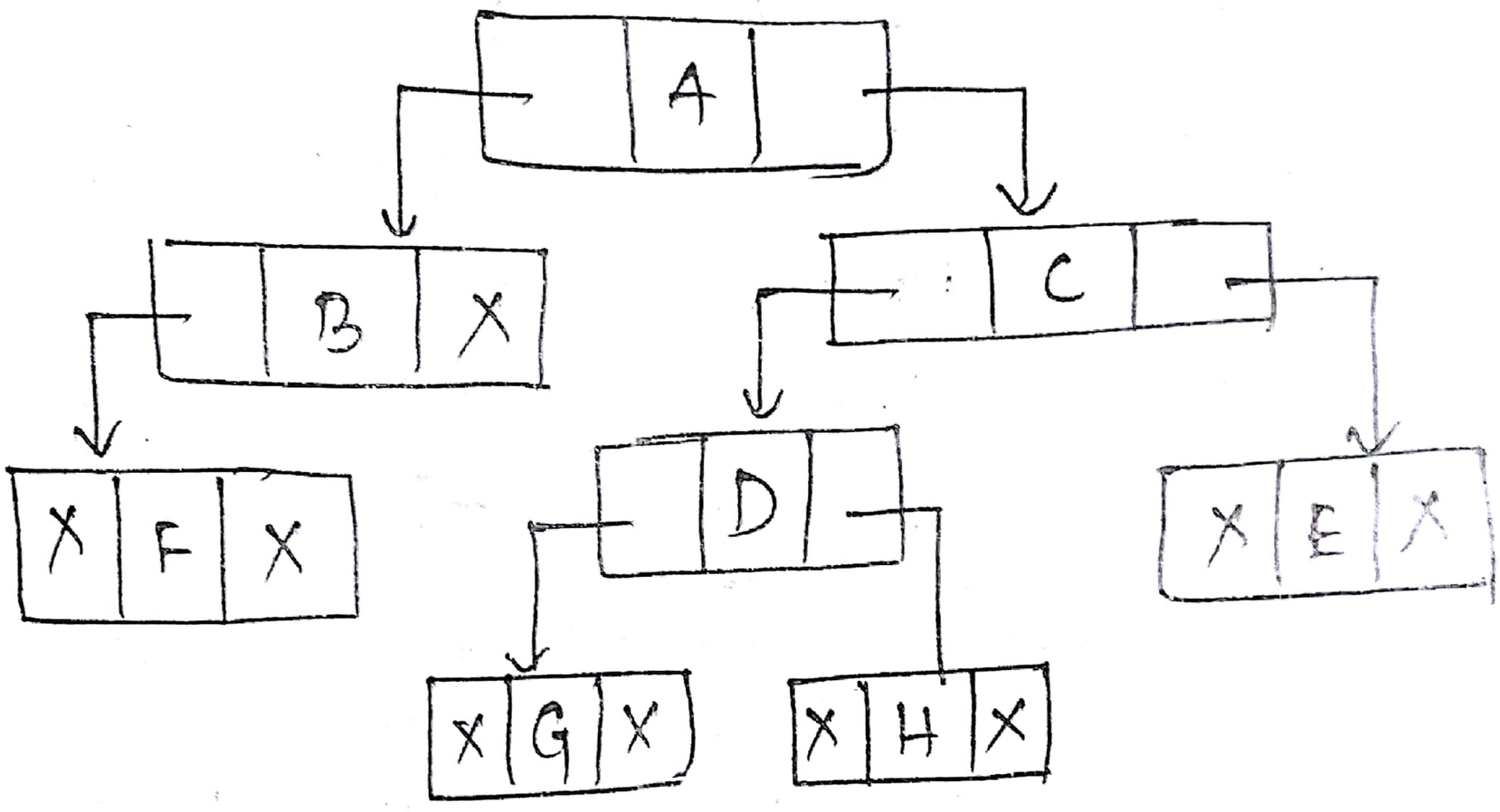
(2) Linked representation of binary tree :-

Node is devided into three fields



(1) info : which is used to store the data
(2) left : left pointer which is used to store the address of the left child.
(3) right : Right pointer which is used to store the address of right child.





Traversing a Binary Tree :-

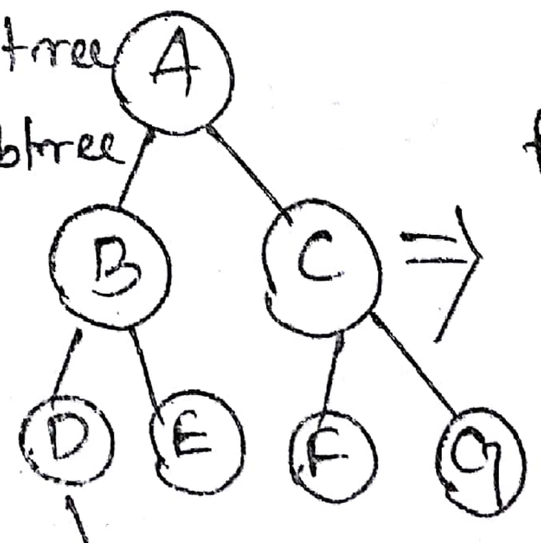
Traversing of binary tree means traversing of nodes left subtree & right subtree.

) There are 3 standard ways of traversing a nonempty binary tree namely (1) Preorder (Node - Left - right)
(2) Inorder (left - Node - right)
(3) Postorder. (left - right - Node)

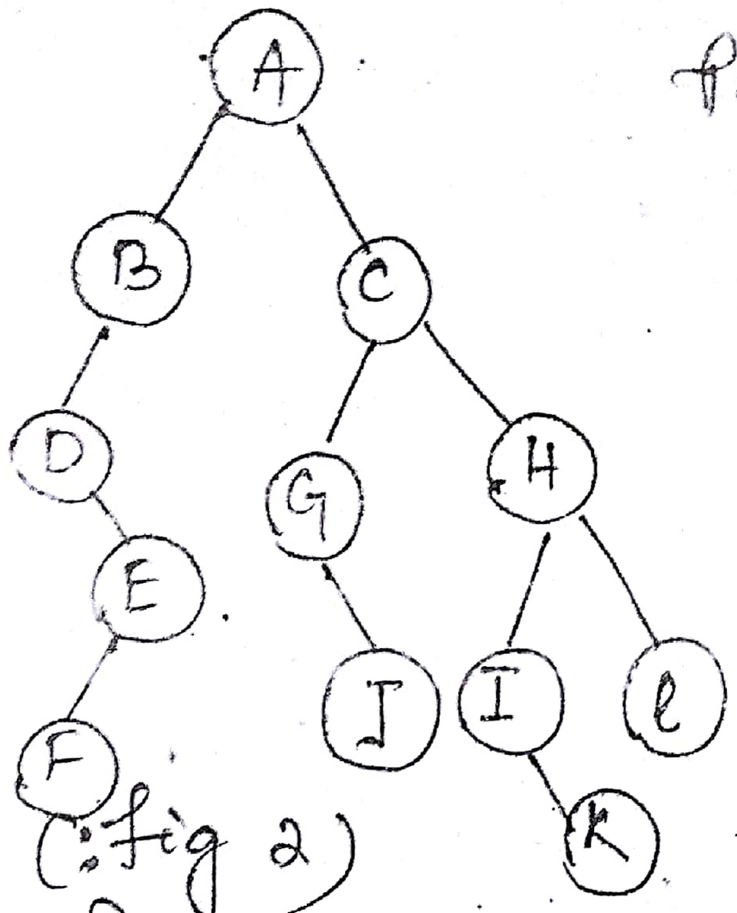
Recursive

Preorder :- 1) Visit root
2) Traverse left subtree in preorder
3) Traverse right subtree in preorder

```
preorder(node *ptr)
{ if(ptr != NULL)
  {
  Pf("%c", ptr->info);
  preorder(ptr->lchild);
  preorder(ptr->rchild);
  }
}
```



Preorder :-
A B D H E C F G



(:fig 2)

preorder ⇒

A B D E F C G J I H K l

inorder:- 1) Traverse left subtree of root in inorder
2) visit root
3) Traverse the right subtree of root in inorder.

'C' function:-

```
inorder(node *ptr)
{ if (ptr != NULL)
  { inorder(ptr->lchild);
    pf("%d", ptr->info);
    inorder(ptr->rchild);
  }
}
```

inorder seq. of fig 2:-

D F E B A G J C I K H l

postorder:- 1) Traverse the leftsubtree of root in postorder
2) Traverse the rightsubtree of root in postorder
3) visit the root.

```
postorder(node *ptr)
{ if (ptr != NULL)
  {postorder(ptr->lchild);
   postorder(ptr->rchild);
   pf("%d", ptr->info);
  }
}
```

postorder seq. of fig 2:

F E D B J G K I l H C A

Non Recursive :-

We will use a stack which will be used to keep the address of the nodes which are to be traversed

preorder :-

'C' function :-

```
void nrec_pre(node *ptr)
{
stack[++top] = ptr;
while(top != -1)
  {
   ptr = stack[top--];
   if(ptr != NULL)
     {
      pf("%d", ptr->info);
      stack[++top] = ptr->rchild;
      stack[++top] = ptr->lchild;
     }
  }
}
```

Algo :-

```
nrec_pre(ptr)
1) set top = top+1
   set stack[top] = ptr
2) Repeat step 3 to 6 till top = -1
3)  set ptr = stack[top]
    set top = top-1
4)  if ptr ≠ NULL
      write ptr->info
5)    set top = top+1
      set stack[top] = ptr->rchi
6)    set top = top+1
      set stack[top] = ptr->lchi
    [end of if structure]
  [end of while loop]
7) exit
```

inorder

```
void nrec_in(node *ptr)
  {
   while(top != -1 || ptr != NULL)
    { if(ptr != NULL)
       {
        stack[++top] = ptr;
        ptr = ptr->lchild;
       }
    else
    {
      ptr = stack[top--];
      pf("%d", ptr->info);
      ptr = ptr->rchild;
    }
  }
}
```

Algo

```
nrec_in(ptr)
1) Repeat step 2 to 6 till
     top ≠ -1 OR ptr ≠ NULL
2)  if ptr ≠ NULL
      set top = top+1
      set stack[top] = ptr
3)    set ptr = ptr->lchild
4)  else
      set ptr = stack[top]
5)    set top = top-1
6)    Write ptr->info
      set ptr = ptr->rchild
    [end of if structure]
  [end of while loop]
7) Exit
```

Postorder:

postorderTraversal(root)

1) set top = 0 , stack[0] = NULL, node = root
   [initially push NULL on to stack & initialize node]
2) Repeat step 3 to 5 while node ≠ NULL
   [push leftmost path onto stack]
3) set top = top + 1
   set stack[top] = node
   [pushes node on to stack]
4) if ~~right[node] ≠ NULL~~ node→rchild ≠ NULL
   [right child exits]
   set ~~top~~ = top+1, stack[top] = ~~node→rchild~~ node→rchild
   [push the negative pointer]
   [end of if structure]
5) set node = node→lchild
   [end of step 2 loop]
6) set node = stack[top].
   set top = top-1
   [pop node from stack]
7) Repeat while node > 0
   [while : non-negative pointer]
   ~~write~~ write node→info
   set node = stack[top]
   set top = top-1
   [end of loop]
8) if node < 0
   [test for negative pointer]
   set node = -node.
   [change to the original value of node]
   go to step 2.
   [end of if structure]
9) Exit.

Construction of binary tree :-

i) construction of tree from preorder & postorder

Preorder: A B D E F C G H J L K

Inorder : D B F E A G C L J H K

Algorithm:

i) In preorder traversal root is the 1st node. So this A is the root.

ii) Find the position of the root Node (A) in the inorder traversal

iii) The nodes precede to root node in the inorder traversal are the nodes of the left subtree of the root node (i.e DBFE) and the nodes succeed to the root node are the nodes in the right subtree of the root node (i.e GCLJH

iv) Now consider two sets of inorder & preorder traversals of the left and right subtrees of the root.

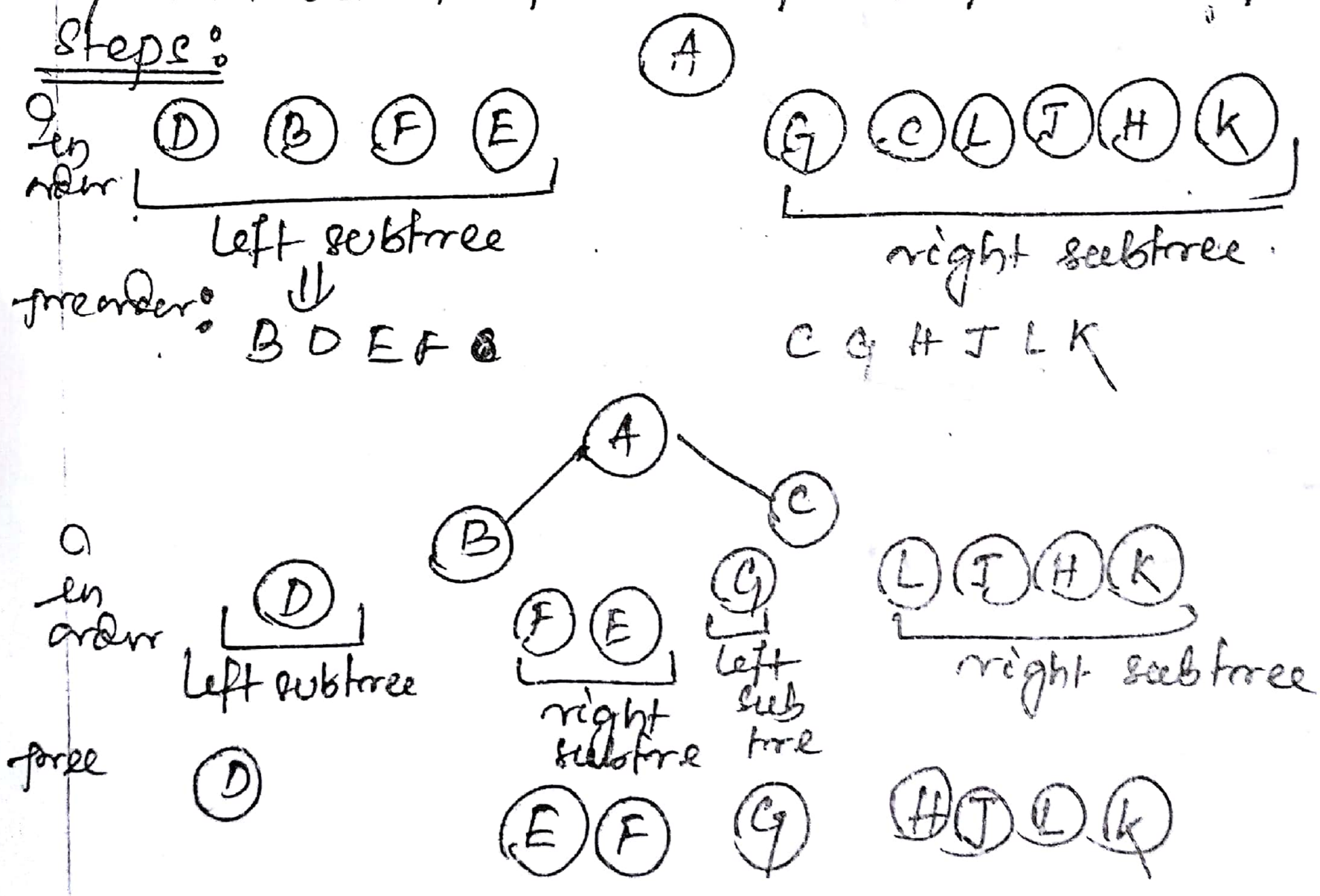
> The 1st set is the nodes appear before root node in inorder DBFE & the combination of those nodes only appear after the root node in preorder traversal (i.e BDEF

> The second set is the nodes appear after root node in inorder traversal is GCLJHK & the nodes in the preorder traversal except the root node & the nodes considered in the ferst set (i.e C G H J L K).

v) Taking the above steps two sets of preorder and inorder traversals for left and right subtree, & repeat step ii to iv till the entire tree is constructed.

~~ii) construction of tree from postorder & inorder.~~

Steps:

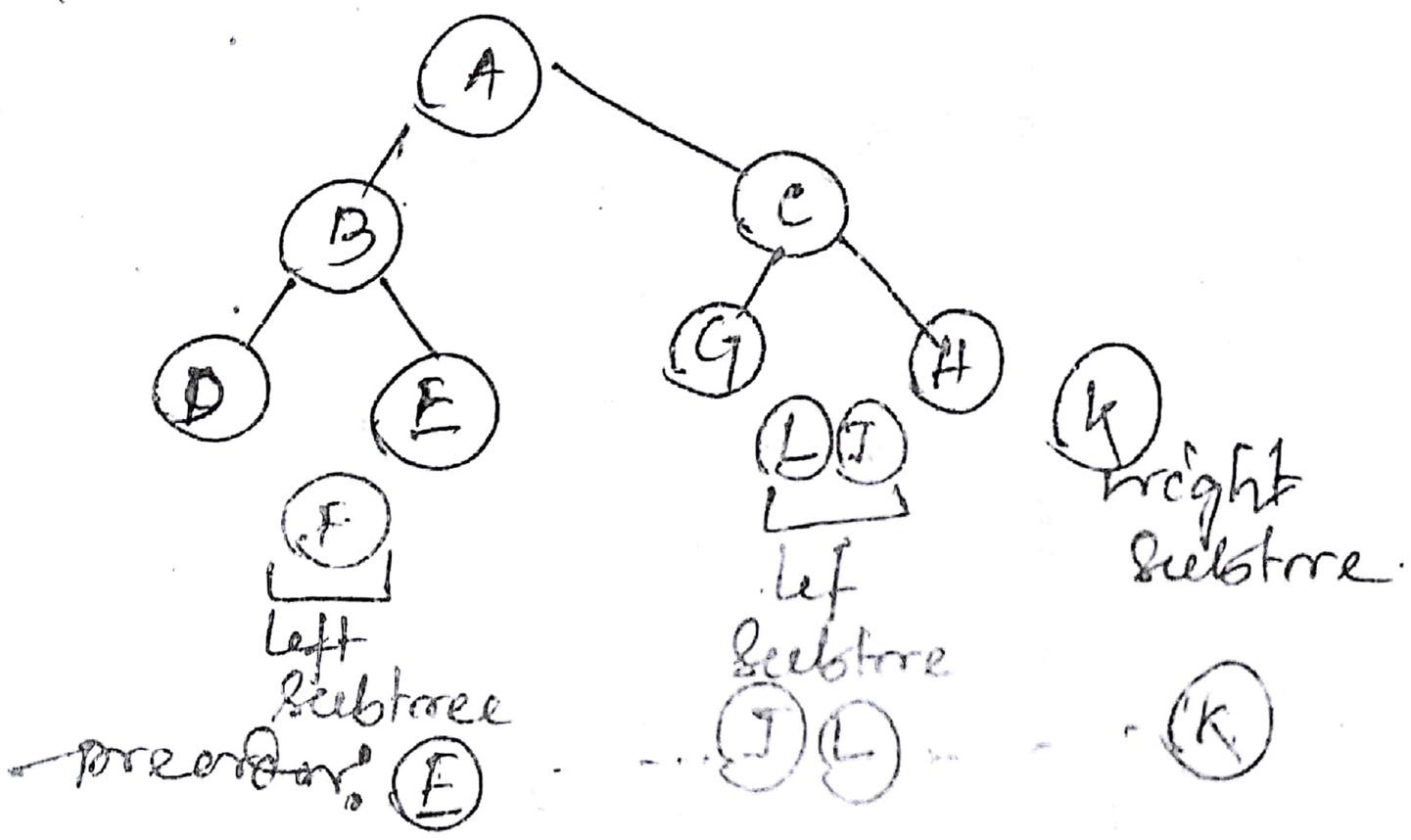


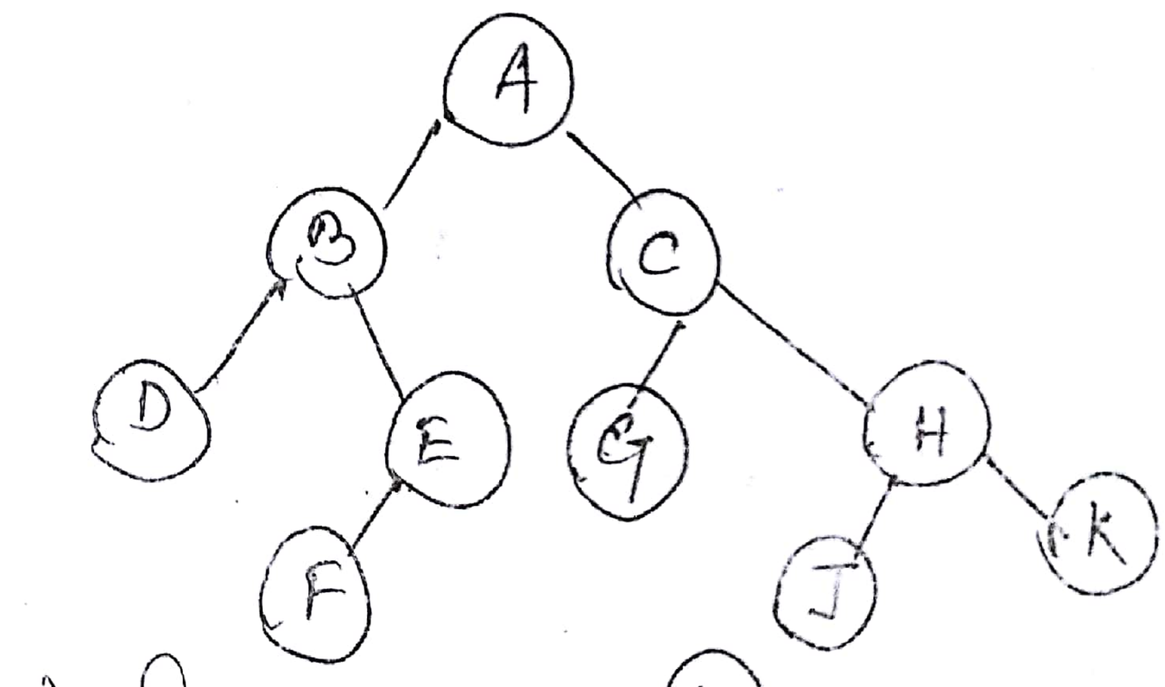
:- Creation of Binary tree using Traversal -:

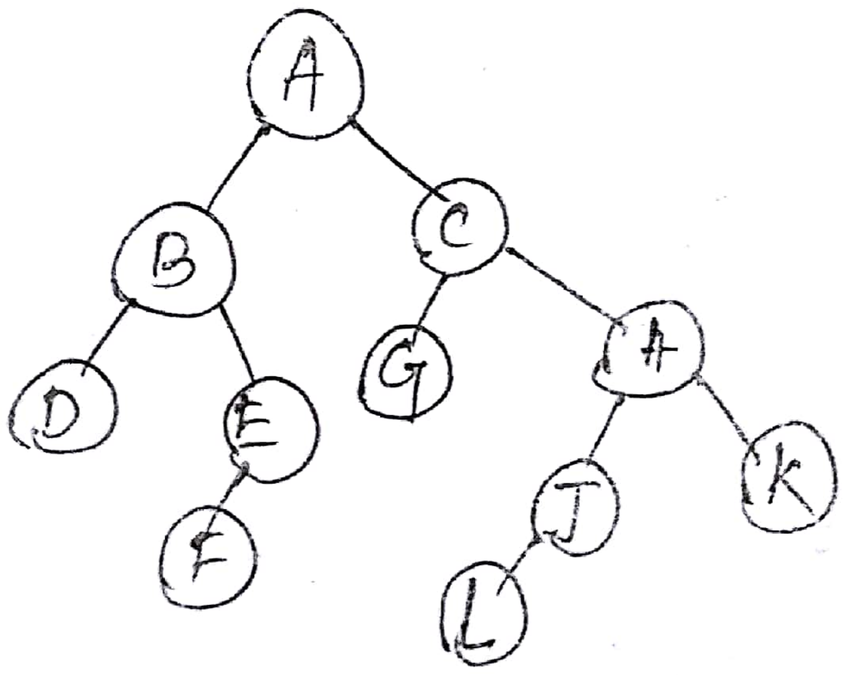
i) From Preorder & Inorder.
ii) From Inorder & postorder.
iii) From preorder & postorder.

~~1) From preorder And Inorder :-~~

~~1) In preorder traversal.~~







postorder -: D F E B G L J K H C A
Inorder -: D B F E A G C L J H K.

construction of binary tree from postorder & inorder.

postorder : H D I E B J F K L G C A

inorder : H D B I E A F J C K G L

Algorithm

1) Last node in the postorder traversal is the root node in the tree (ie A)

2) Finds the position of the root node (ie A) in the above example in inorder traversal. The nodes precede to root node in the inorder traversal are the nodes in the left subtree of the root node (ie H D B I E). The nodes present after the root node in the inorder traversal are the nodes of right subtree.

3) Now consider two sets of inorder and postorder traversals of the left & right subtree of the root.

) 1st set is the nodes appear precede to the root node in inorder traversal (ie H D B I E) and the sequence of same nodes that are in postorder traversal just before the root node is H D I E B.

) 2nd set is the nodes appear after the root node in inorder traversal is F J C K G L & the sequence of same nodes that are in postorder traversal ie J F K L G C

4) Taking above two sets of postorder & inorder traversals for left & right subtrees, & repeat step 2 & 3 till the entire tree is constructed.

Steps are :

Tre: A B D G H K C E F
post: G K H D B E F C A

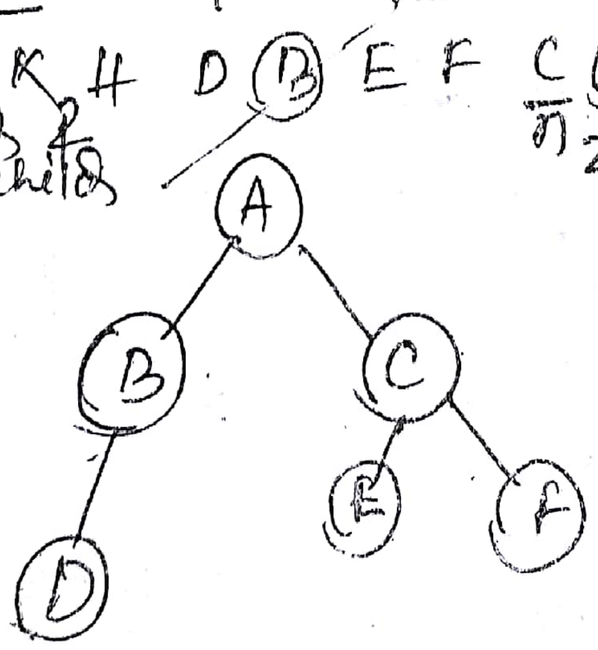
iii) construction of tree from preorder & postorder :-
Algorithm :-
1) The first node in the preorder traversal and last node of the postorder traversal is considered as the root node of the tree (i.e. A)

2) Find the succesor node of the root node in preorder traversal say n1 and predecessor of the root node in postorder traversal say n2. Then there will be two cases.
   i) if n1 = n2 then this node is considered to be the left or right child of the root node, in which the construction of tree is not unique.
   ii) if n1 ≠ n2 then n1 is considered as the left child (i.e. B) and n2 is considered as the right child (i.e C) of the root node

3) Find the position of n2 (i.e C) and n1 (i.e B) in preorder and postorder traversals respectively. Now consider the two sets of preorder and postorder traversals of left and right subtrees of the root. The first set is the nodes appear after n1 & before n2 in preorder traversal (i.e DGHK) and the nodes ~~appear after n~~ precede to node n1 in postorder traversal (i.e GKHD). The second set is the nodes appear after n2 in preorder traversal (i.e EF) and the nodes in between n1 & n2 in postorder traversal (i.e EF).

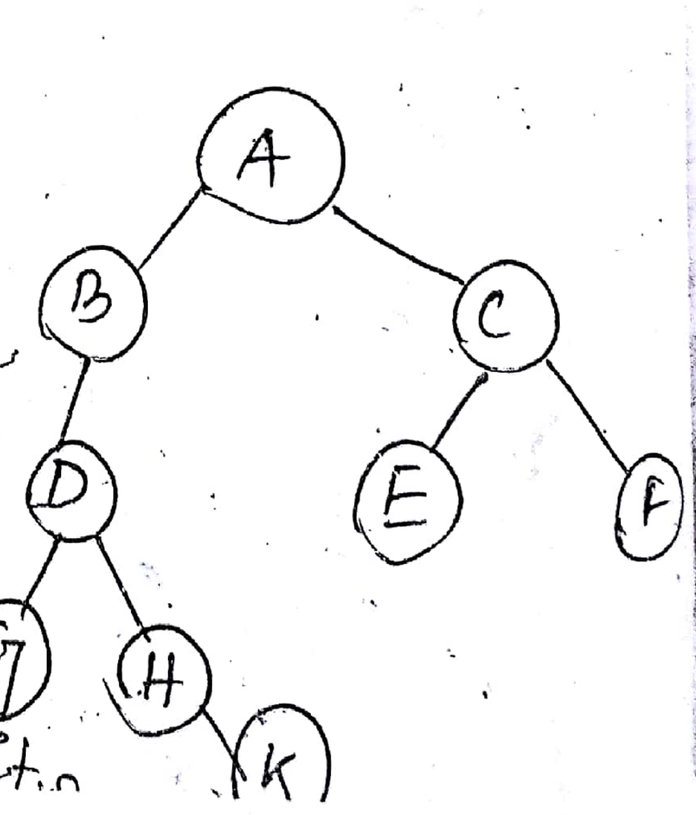
4) Taking these two sets of preorder and postorder traversal for left and right subtrees, & repeat the step 2 & 3 till the entire tree is constructed.

Steps :-
(A) n1 B D G H K (C) E F
⇒ if n1 ≠ n2. G K H D (B) E F C (A)
then n1 is left child & n2 right child
⇒ (B) D G H K (C)
G K H D (B)
n1 = n2
either left or right child of B



C n1 E B
E F C
n2

⇒ D G H K
G K H D
n2



H K either left