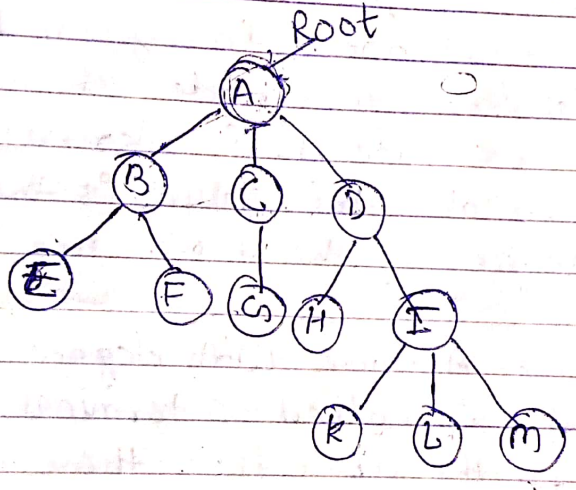
Both tree and graphs comes in non-linear data Structure in which each node or item may be connected with two or more other nodes or items in non-linear arrangement.

UNIT - 3

ee → A tree is a non-linear data Structure in which items are ranged in a sorted sequence. it is used o represent - hierarchical relationship xisting amongst several data items.

The graph theoretic definition tree is: it is a finite set of one r more data items (nodes) such that

1. There is a special data item called the root of the tree.
2. And its remaining data items are Partitioned into number of mutually exclusive subsets, each of which is itself a tree. And they are called Subtrees.



Tree Terminology →

depth = level + 1

1. Root
2. Node
3. Degree of a node
4. Degree of tree
5. Path
6. Terminal node
7. Non-terminal node
8. Siblings
9. level
10. edge
11. Depth
12. Forest

Root → it is specially designed data items in a tree. It is the first in the hierarchical arrangement of data items. In the above tree A is root.

Node → Each data item in a tree is called a node. It is the basic structure in a tree. it specifies the data information and links to other data items. There are 13 nodes in the tree

Degree of a node → it is the number of subtrees of a node in a given tree.
Eg: The degree of node A is 3.

Degree of tree → it is the maximum degree of nodes in a given tree.
In the Eg of tree the node A has degree 3 and another node I is also having its degree 3. In all this value is the maximum So, the degree of the above tree is 3.

Terminal node → A node with degree zero is called a terminal node or a leaf. In the above tree there are 7 terminal nodes. they are E, J, G, H, K, L and M

Non-terminal node → Any node (except the root node) whose degree is not zero is called non-terminal node. Non-terminal nodes are the intermediate nodes in traversing the given tree from its root node to terminal node. Eg - non terminal

DATE : PAGE NO. :

blings → The children nodes of a given Parent node are called Siblings. ey are also called brothers.

el → The entire tree structure is levelled in such a way that the root node is ways at level 0. Then, its immediate children e at level 1 and their immediate children ll be at level n+1. ~~I~~ 4 levels in Example aximum no. of links traversed in depth first search order

ge → It is connecting line of two nodes. That is, the line drawn from one de to another node is ~~called~~ called an ge.

th → It is a sequence of consecutive edges from the source node to the stination node. In the above tree, the th between A and J is given by the ode Pairs

(A,B), (B,F) and (F,J)

pth → it is the maximum level of any node in a given tree. In the bove tree, the root node A has the maxi- um level. That is the number of levels ne can descend the tree from its root o the terminal nodes. The term height is also used to denote the depth aximum no. of nodes traversed in depth first search order

rest → it is a set of disjoint trees. In a ee, if you remove its root node then it become a forest. Forest with three ee in the given Example.

→ Binary tree of height h is ~~given~~ $2^h - 1$

maximum no of nodes = $2^h - 1$

height = 5 , n = 31

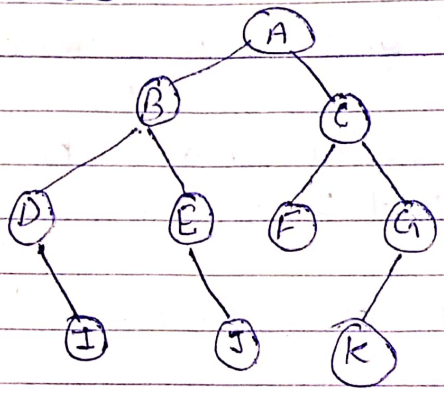
DATE : PAGE NO. :

Binary Tree → A binary tree T is defined as a finite set of elements called nodes, such that -

(a) T is empty (called the null tree or empty tree) or

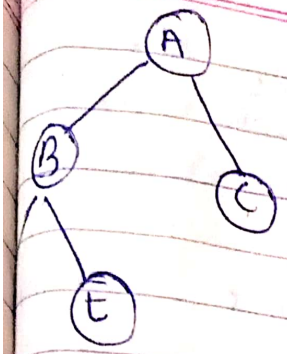
(b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2

In binary tree the maximum degree of any node is at most two. That means, there may be a zero degree node or a one degree node and two degree node. figure given below

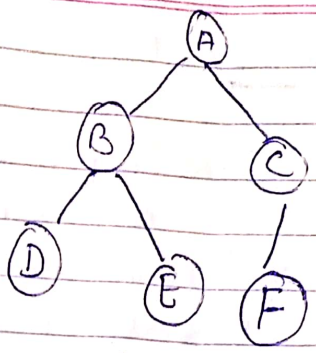


Strictly Binary Tree → A binary tree is called a strictly binary tree if every nonleaf node in the binary tree has nonempty left and right subtree. This means each node in a binary tree will have either 0 or 2 children. A strictly binary tree with n leaves always contain exactly 2n-1 nodes. figure given figure (a) is strictly binary tree but fig (b) is not strictly binary tree.

→ a binary tree cannot accomodate more than 2^{i-1} nodes at level i.

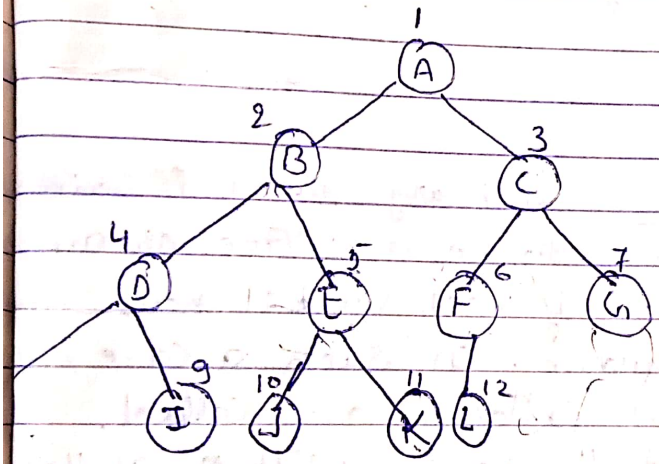


(a) SBT



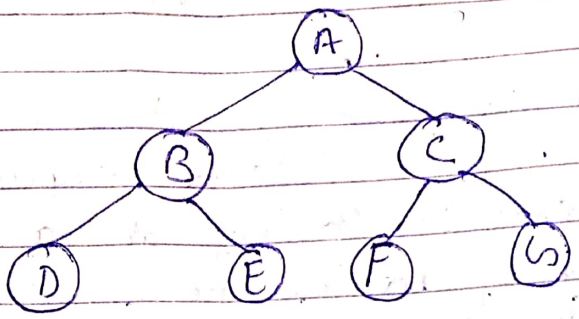
(b)

mplete Binary Tree → The tree T is said to be complete if ll its levels, except possibly the last, have aximum number of possible nodes, and if all e nodes at the last level appear as far ft as possible.



ith this labeling, one can easily determine he children and parent of any node K in y complete tree Tn, specifically the left and ght children of the node K are, respectively *K and 2*K+1, and the parent of K s the node $\lfloor K/2 \rfloor$. for example, the children node 5(K) are 2×5 = 10 and 2×5+1 = 11 and its arent is the node $\lfloor K/2 \rfloor \frac{5}{2} = 2$.

Full Binary Tree → Consider any binary tree T, Each node of T can have at most two children. Accordingly, one can show that level r of T can have at most 2^r nodes called Full Binary Tree.



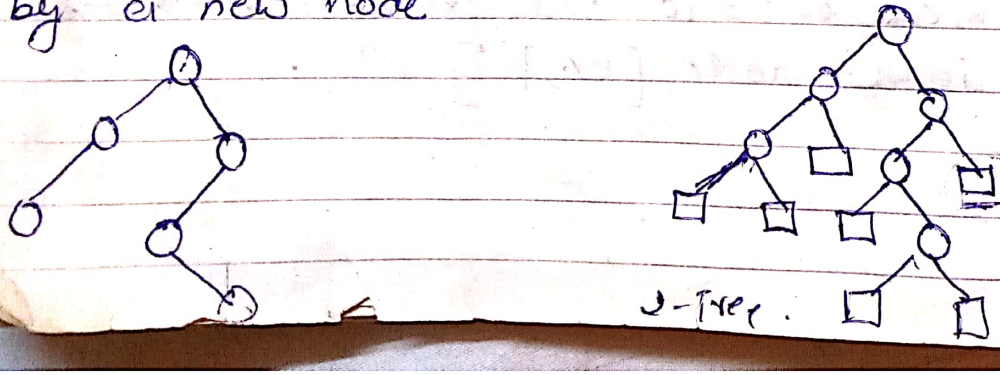
level $r = 2$, no of nodes $2^r = 2^2 = 4$

$2^0 = 1$

$2^1 = 2$

Extended Binary Tree → A binary tree T is said to be a 2-Tree or an extended binary tree if each node N has either 0 or 2 children. In such a case, the nodes with 2 children are called internal nodes, and the nodes with 0 children are called external nodes.

The term "exteded Binary tree" comes from the following operation. Consider any binary tree T, such as the tree in fig. Then T may be "converted" into a 2-tree by replacing each empty subtree by a new node



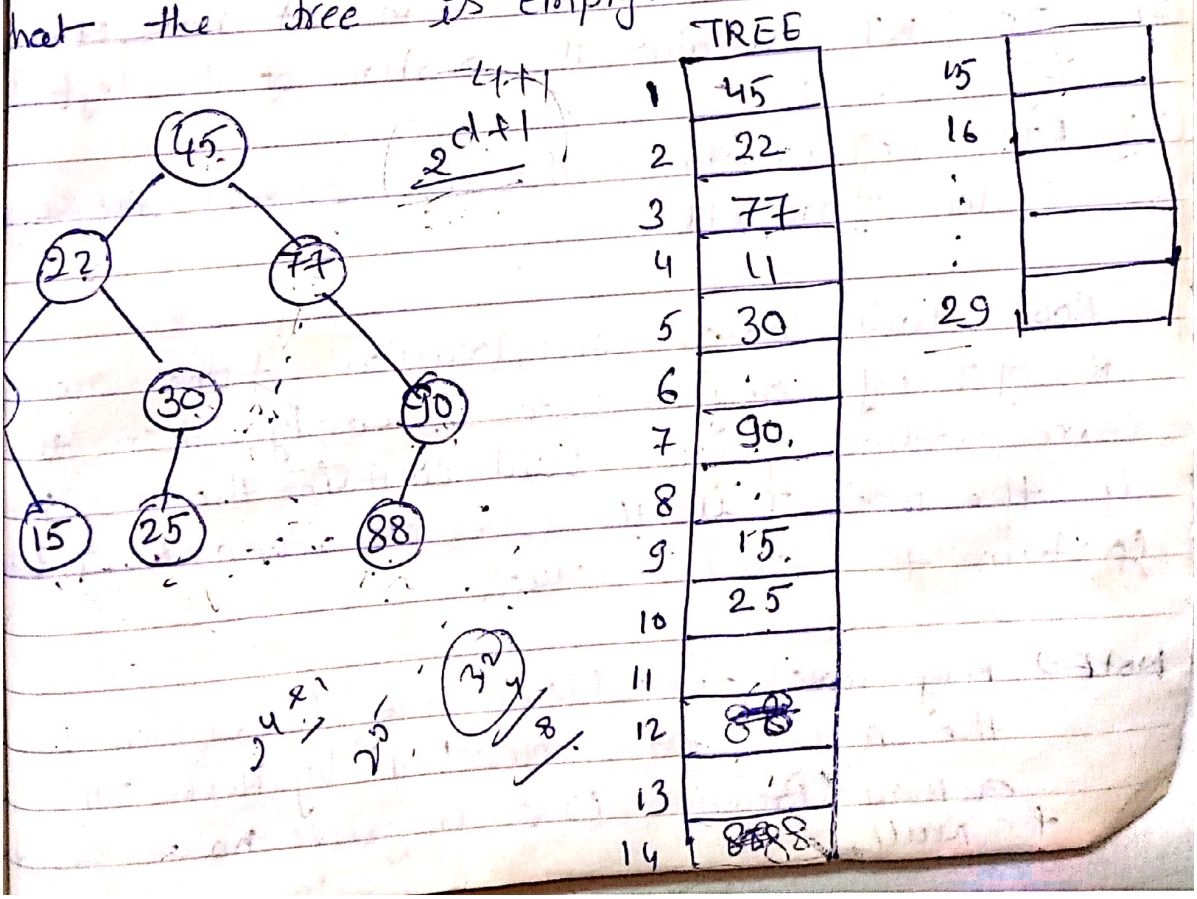
ary Tree Representation →

Array Representation of Binary Tree →

uppose T is a binary tree that is complete r nearly complete. Then there is an efficient ay of maintaining T in memory called e Sequential Representation of T. This epresentation uses only a single linear ray TREE as follows.

) The root R of T is stored in TREE[1].
if a node N occupies TREE[K], then its left child is stored in TREE[2*K] and its right child is stored in Tre TREE[2*K+1].

Again NULL is used to indicate an empty. ubtree. In Particular TREE[1] = NULL indicates hat the tree is empty.



Observe that we require 14 locations in the array TREE even though T has only 9 nodes. In fact, if we included null entries for the successors of the terminal nodes, then we would actually require TREE[29] for the right successor of TREE[14]. Generally speaking the sequential representation of a tree with depth d will require 2^{d+1} elements

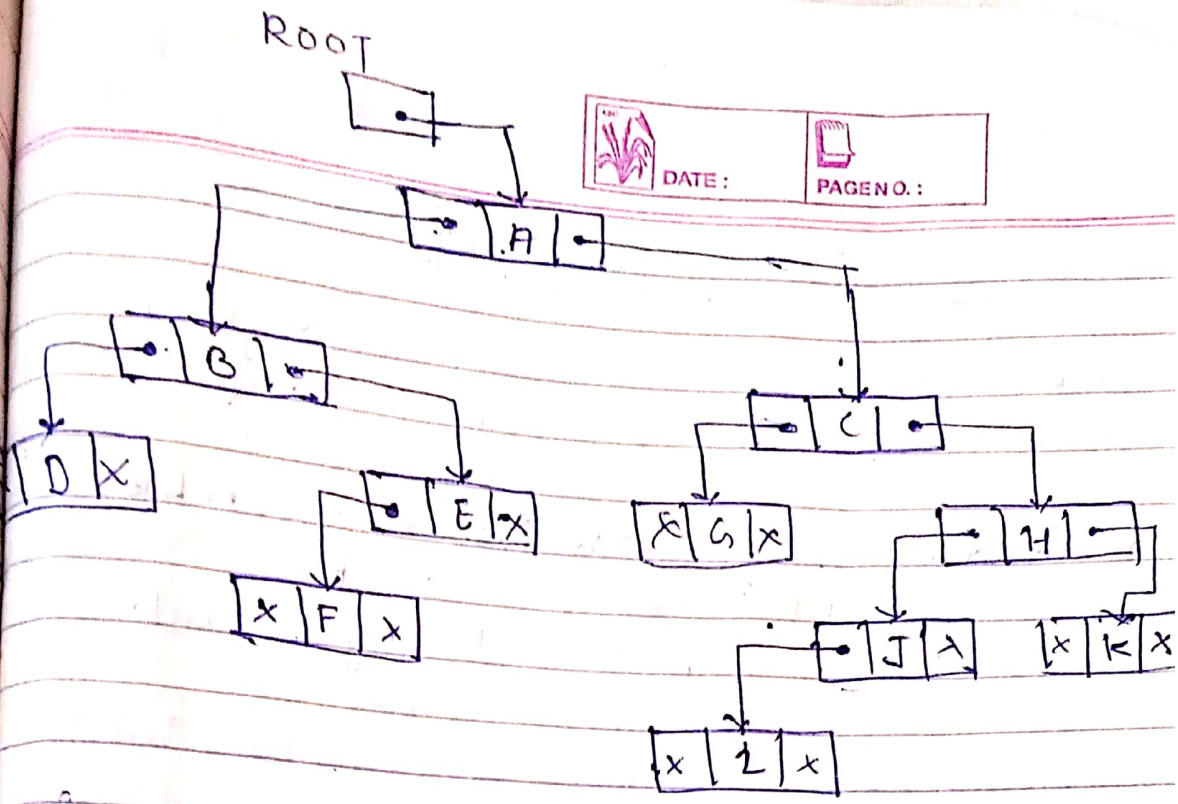
Linked Representation of Binary Tree →

Consider a binary tree T. Unless otherwise stated or implied, T will be maintained in memory by means of a linked list representation which uses three parallel arrays, INFO, LEFT and RIGHT, and a pointer variable ROOT as follows. First of all, each node N of T will correspond to a location K such that:

(1) INFO[K] contains the data at node N.
(2) LEFT[K] contains the location of the left child of node N.
(3) RIGHT[K] contains the location of the right child of node N.

ROOT will contain the location of the root R of T. If any subtree is empty, then the corresponding pointer will contain the null value. If the tree T itself is empty, then ROOT will contain the null value.

NOTE → Any invalid address may be chosen for the null pointer denoted by NULL. In actual practice, 0 or negative no is used for null.



	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	

Traversing Binary Trees →

There are three standard ways of traversing a binary tree T with Root R

① Preorder → (depth first order) (NLR)
- (1) Process the root
- (2) Traverse the left Subtree of R in Preorder
- (3) Traverse the right Subtree of R in Preorder.

② Inorder → Traverse (LNR)
- (1) Traverse the left subtree of R in inorder
- (2) Process the root R.
- (3) Traverse the right subtree of R in inorder.

③ Postorder → (LRN)
- (1) Traverse the left Subtree of R in Postorder
- (2) Traverse the Right Subtree of R in Postorder
- (3) Process the root R.

⇒ Catalan Number gives the Number of distinct trees containing n nodes it is given by $Cat(n) = \frac{\lfloor 2n}{\lfloor n \rfloor \lfloor n+1 \rfloor} = \frac{\lfloor 2n}{n \lfloor n+1}$

ration on Binary Tree →

ing the array implementation.

```
#define maxnum 100
Struct nodetree {
    int info;
    int left;
    int right;
};
Struct nodetree node[maxnum]
```

ng linked ~~list~~ array representation.

```
Struct- nodetree {
    int info;
    Struct node *left, *right;
} tree;
Struct nodetree *P;
```

iary Search tree → A binary search tree is a binary tree which of either empty or satisfies the following les.

The value of the key in the left child or left subtree is less than the value of the root.

The value of the key in right child or right subtree is more than or equal to the value of the root.

Note → Binary Search Tree inorder traversal of T will yeid a sorted listing of elements of T.

Searching and Inserting In Binary Search Tree →

Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the binary search tree T or insert ITEM as a new node in its appropriate Place in the tree.

(a) Compare ITEM with the root node N of the tree:
   (i) if ITEM < N Proceed to the left ch of N
   (ii) if ITEM > N, Proceed to the Right ch of N.

(b) Repeat Step (a) until one of the foll occurs.
   (i) we meet a node N such that ITEM = N. In this case the Search is Successful.
   (ii) we meet an empty Subtree, which indicates that the Search is unsuccessful, and we insert ITEM in Place of the empty Subtree.

In other words, Proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

binary Search tree T is in memory and an ITEM of information is given. This Procedure finds the location LOC of ITEM in T and also the location PAR of the Parent of ITEM.
There are Three special cases:

1. LOC = NULL and PAR = NULL will indicate that the tree is empty.
2. LOC ≠ NULL and PAR = NULL will indicate that ITEM is the root of T.
3. LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

[Tree empty?]
if ROOT = NULL, then: Set LOC := NULL and PAR := NULL and Return.
[ITEM at root?]
if ITEM = INFO[ROOT], then: Set LOC := ROOT and PAR := NULL, and Return.
[Initialize Pointers PTR and SAVE].
if ITEM < INFO[ROOT], then:
Set PTR := LEFT[ROOT] and SAVE := ROOT
Else
Set PTR := RIGHT[ROOT] and SAVE := ROOT
Repeat Steps 5 and 6 while PTR ≠ NULL.
[ITEM found?]
if ITEM = INFO[PTR], then Set LOC := PTR and PAR := SAVE, and Return.
if ITEM < INFO[PTR], then:
Set SAVE := PTR and PTR := LEFT[PTR]
[c
Set SAVE := PTR and PTR := RIGHT[PTR].

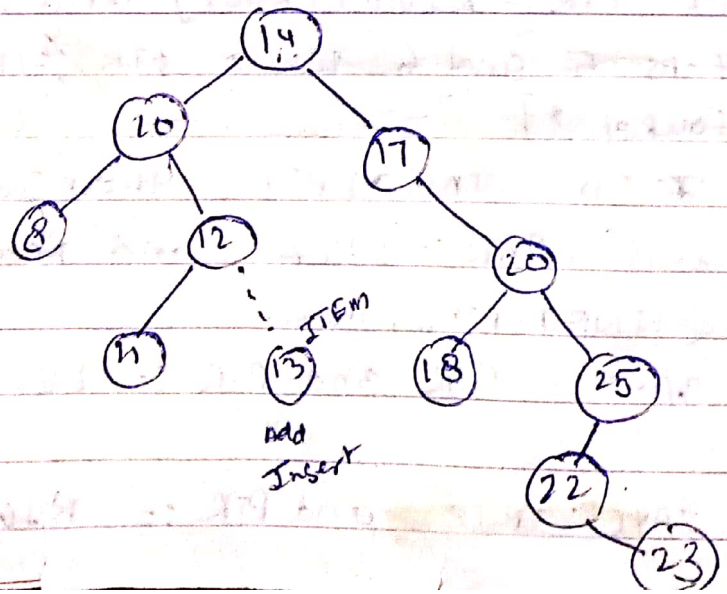
7) [Search Unsuccessful]. Set LOC := NULL and PAR := SAVE.

8) Exit.

Insertion Algorithm →

INSBST [INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM]

1. Call FIND [INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR]
2. if LOC ≠ NULL then Exit.
3. [Copy ITEM into new node in AVAIL list]
   (a) if AVAIL = NULL, then: write: OVERFLOW, Exit.
   (b) Set NEW := AVAIL, AVAIL := LEFT [AVAIL]
       INFO [NEW] := ITEM.
   (c) Set LOC := NEW, LEFT [NEW] := NULL and RIGHT [NEW] := NULL
4. [Add ITEM to tree].
   if PAR = NULL, then:
   Set ROOT := NEW.
   Else if ITEM < INFO [PAR], then:
   Set LEFT [PAR] := NEW.
   Else
   Set RIGHT [PAR] := NEW.
5. Exit.



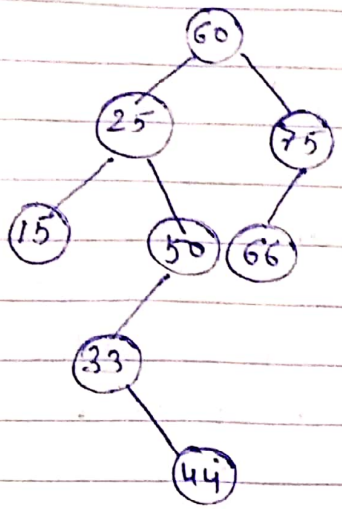
eleting In A Binary Search Tree →

ppose T is a Binary Search tree and
ppose ITEM of Information is given.
The deletion algorithm first
es Procedure to FIND the location of
e Node N which contains ITEM and
so the location of the Parent node P(N).
e Way N is deleted from the tree
epends Primarily on the number of childre
of Node N. There are three cases:

1 → N has no children, Then N is deleted from T by simply replacing the location of N in the Parent node P(N) by the null Pointer.

2 → N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

3 → N has two children. Let S(N) denote the inorder Successor of N. (The reader can verify that S(N) does not have a left child) Then N is deleted from T by first deleting S(N) from T (by using case 1 and case 2) and then replacing node N in T by the node S(N).



	INFO	LEFT	RIGHT
1	33	8	10
2		5	
3	60	1	7
4	66	0	0
5		6	
6		0	
7	75	4	
8	15	0	0
9	44	0	0
10	50	9	0

CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This Procedure deletes the node N at location LOC, where N does not have two children. The Pointer PAR gives the location of the Parent of N, or else PAR = NULL indicates that N is the root node. The Pointer CHILD gives the location of the only child of N or else CHILD = NULL indicates N has no children.

1. [Initialize CHILD]
   if LEFT[LOC] = NULL and RIGHT[LOC] = NULL
   then: Set CHILD := NULL
   Else if LEFT[LOC] ≠ NULL, then:
   Set CHILD := LEFT[LOC]
   Else
   Set CHILD := RIGHT[LOC].

2. if PAR ≠ NULL, then:
   if LOC = LEFT[PAR], then:
   Set LEFT[PAR] := CHILD

Else
Set RIGHT[PAR] := CHILD
Else
Set ROOT := CHILD
3. Return.

CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This Procedure will delete the node N at location LOC, where N has two children. The Pointer PAR gives the location of the Parent of N, or else PAR = NULL indicates that N is the root node. The pointer SUC gives the location of the inorder Successor of N and PARSUC gives the location of the Parent of the inorder Successor.

1. [Find SUC and PARSUC]
   (a) Set PTR := RIGHT[LOC] and SAVE := LOC.
   (b) Repeat while LEFT[PTR] ≠ NULL:
       Set SAVE := PTR and PAR := LEFT,
       PTR := LEFT[PTR]
   (c) Set SUC := PTR and PTR := LEFT[PTR].

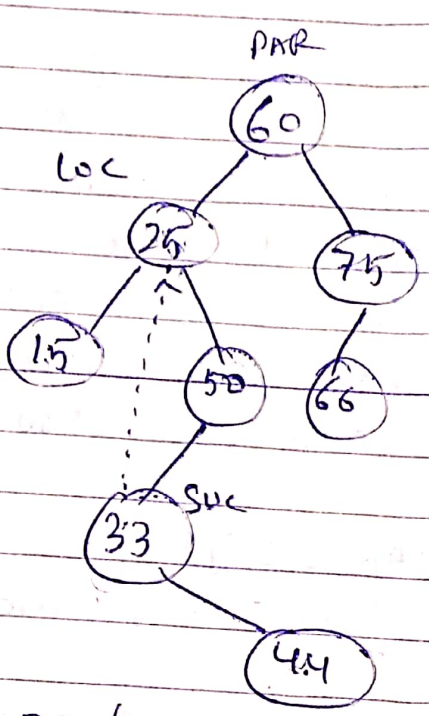
2. [Delete inorder Successor, using Procedure]
   Call CaseA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)

3. [Replace node N by its inorder Successor]
   (a) if PAR ≠ NULL, then:
       if LOC = LEFT(PAR), then:
       Set LEFT[PAR] := SUC

Else
    Set RIGHT[PAR] := SUC.
Else
    Set ROOT := SUC.

(b) Set LEFT[SUC] := LEFT[LOC] and
    RIGHT[SUC] := RIGHT[LOC].

4. Return.



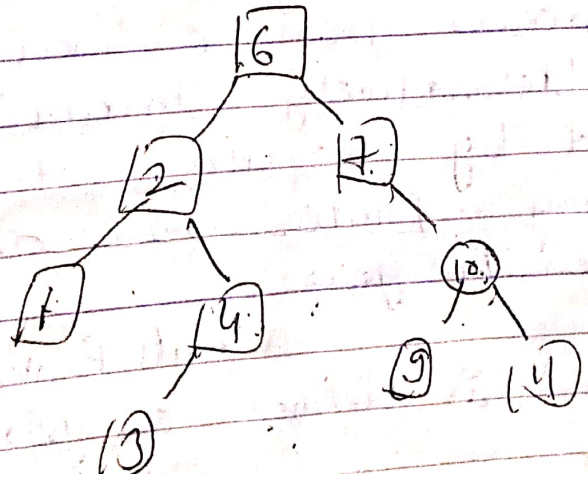
DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)
A binary search tree T is in memory and an ITEM of information is given. This algo deletes ITEM from the tree.

1. [Find the locations of ITEM and its Parent using
    Call Find(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. [ITEM in tree?]
    If LOC = NULL, then: Write: ITEM not in tree and Exit.
3. [Delete node containing ITEM]

if RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL
then,
Call CaseB(INFO, LEFT, RIGHT, ROOT, LOC, PAR
Else:
Call CaseA(INFO, LEFT, RIGHT, ROOT, LOC, PA
4. [Return deleted node to the AVAIL list]
Set LEFT[LOC] := AVAIL and AVAIL := LO
5. Exit

6 2 1 4 3 7 10 9 11

1 2 3 4 6 7 9 10 11



DATE : PAGE NO.:

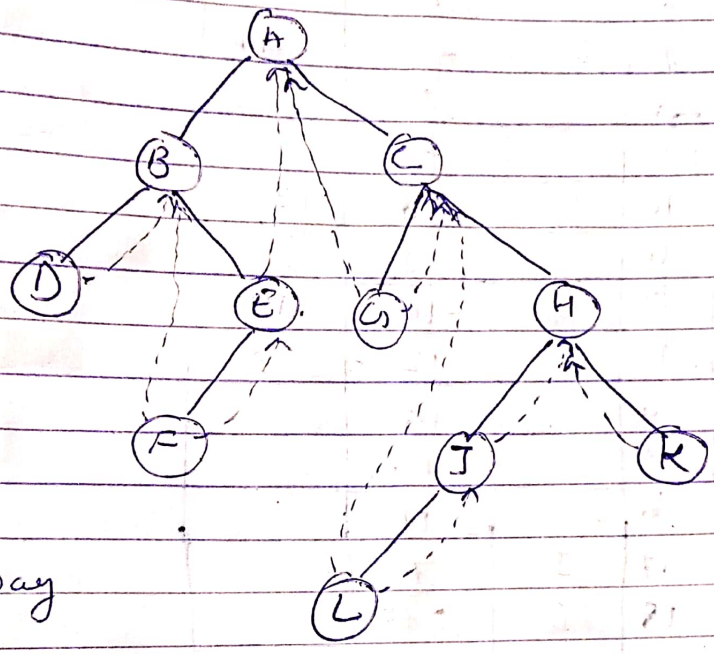
Threaded binary Tree → In a linked Representation of a binary tree, the number of null links (Null Pointers) are actually more than the non-null Pointer. Approximately half of the entries in the Pointer fields LEFT and RIGHT will contain null elements. This space may be more efficiently used by replacing the null entries by some other type of information, specifically, we will replace certain null entries by Special pointers which point to nodes higher in the tree. These special pointers are called threads and binary trees with such pointers are called ~~threaded~~ threaded trees.

The threads in a threaded tree must be distinguished in some way from ordinary Pointers. The threads in a diagram of a threaded tree are usually indicated by dotted lines. In computer memory, an extra 1-bit TAG field may be used to distinguish threads from ordinary pointers or alternatively threads may be denoted by negative integers when ordinary pointers are denoted by Positive Integers.

A Null Pointer replacement scheme is defined as follow.

If the left link (child) of a node P is normally equal to NULL then this link is replaced by

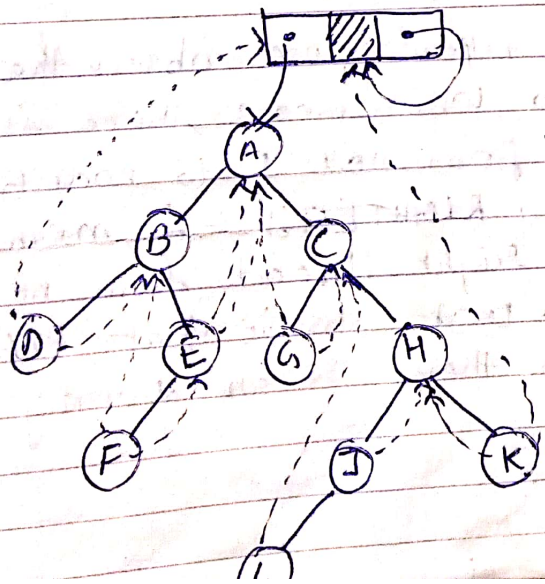
a pointer to the node which - immediately precedes node p in inorder traversal, And similarly if the right link (rchild) of a node p is normally equal to NULL, then this link is replaced by a Pointer to its inorder Successor node. ~~A threaded~~



Two-way

Inorder Traversal.

D, B, F, E, A, G, C, J, H, K.



Two-Way ~~Header~~ Threading With header node

Memory Representation in Threaded Binary Tree →

HEAD [20]

AVAIL [8]

	INFO	LEFT	RIGHT
1	K	-6	-20
2	C	3	6
3	G	-5	-2
4		14	
5	A	10	2
6	H	17	1
7	L	-2	-17
8		9	
9		4	
10	B	18	13
11		19	
12	F	-10	-13
13	E	12	-5
14		15	
15		16	
16		11	
17	J	7	
18	D	-20	
19		0	
20		5	

INFO[20] as a Header node. Observe that LEFT[12] = -10, which means there is a left thread from node F to node B. Analogously, RIGHT[17] = -6, means there is a right thread from node J to node H. Last observe that RIGHT[20]=20 which means there is an ordinary right

Conversion of General tree To Binary Tree

It is convenient to represent binary tree than represent the general trees in Programs because in case of a general tree, the number of edges emanating from a node at any given time is unpredictable. Therefore, it is difficult difficult to manage the node space. In case of binary tree, each node has a predictable maximum of two sub-tree pointers.

⇒ A general tree can be converted into an equivalent binary tree. This conversion process is called the natural correspondence between general tree and binary tree. The algorithm is as follows.

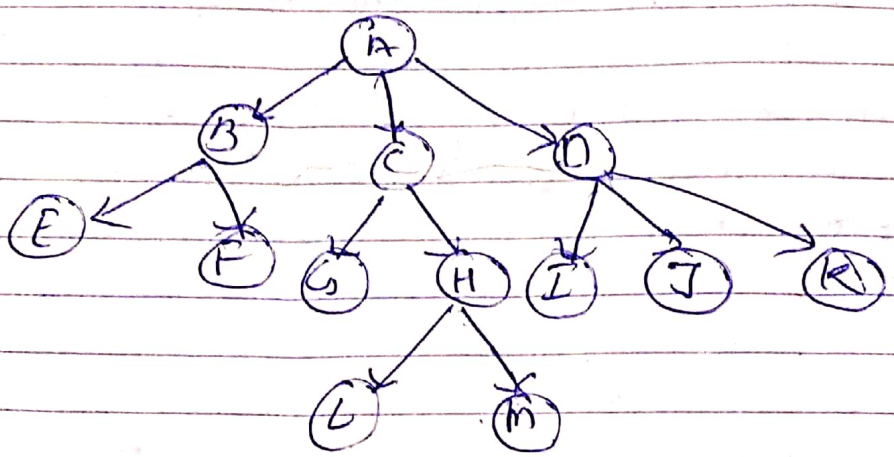
(a) Insert edges connecting Sibilings from left to right at same level.

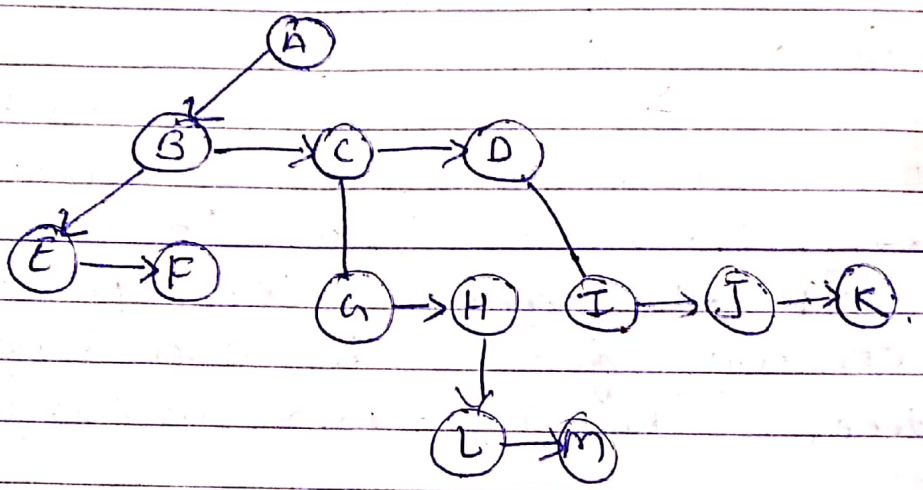
(b) Delete all edges of a Parent to its children except to its left most offspring.

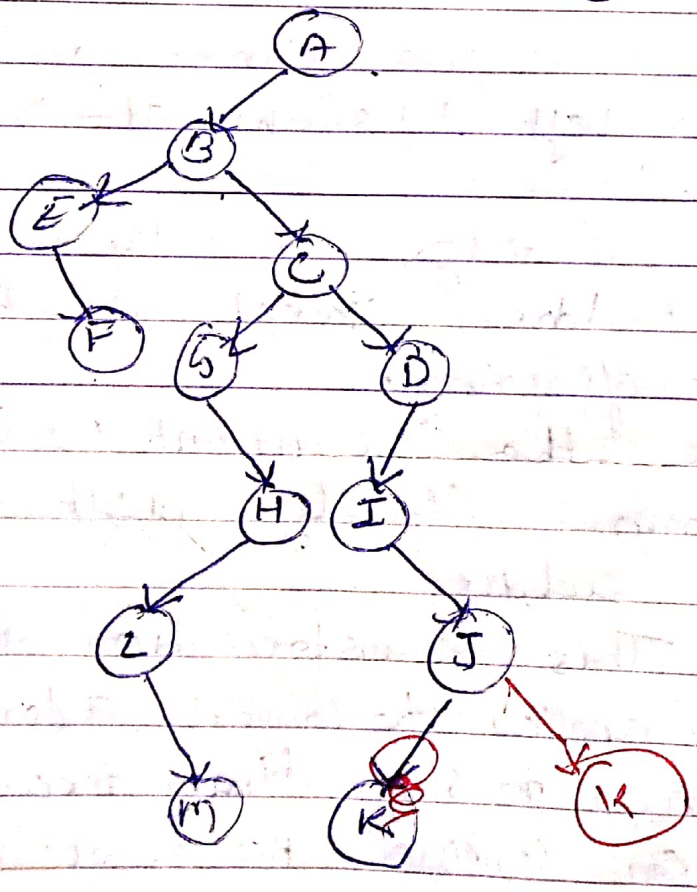
(c) Rotate the resultant tree 45° to mark clearly left and Right subtree.

This transformation algorithm can also be applied to convert a forest of general trees to single binary tree. In this case we can consider the root of the first general tree in the forest to be

the root of the binary tree and the 2a roots of other general trees can be considered as its sibilings.



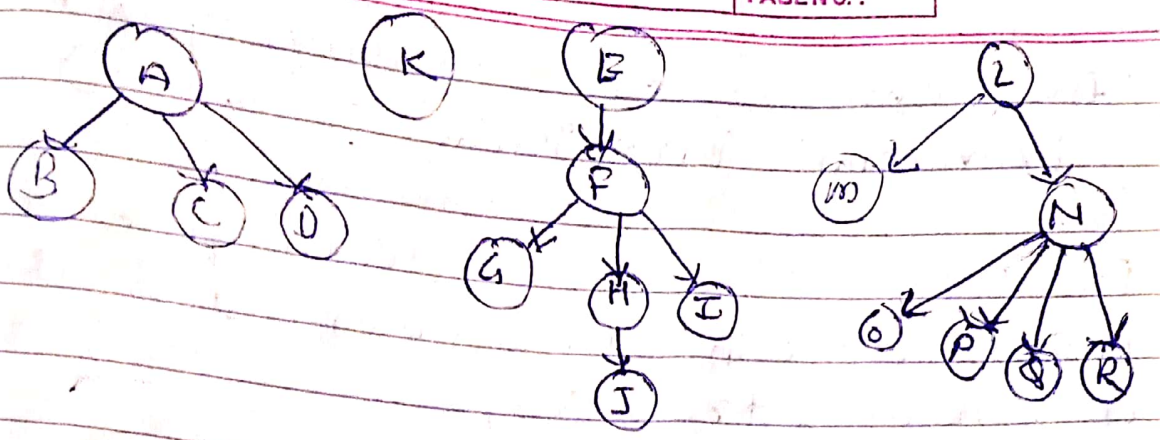


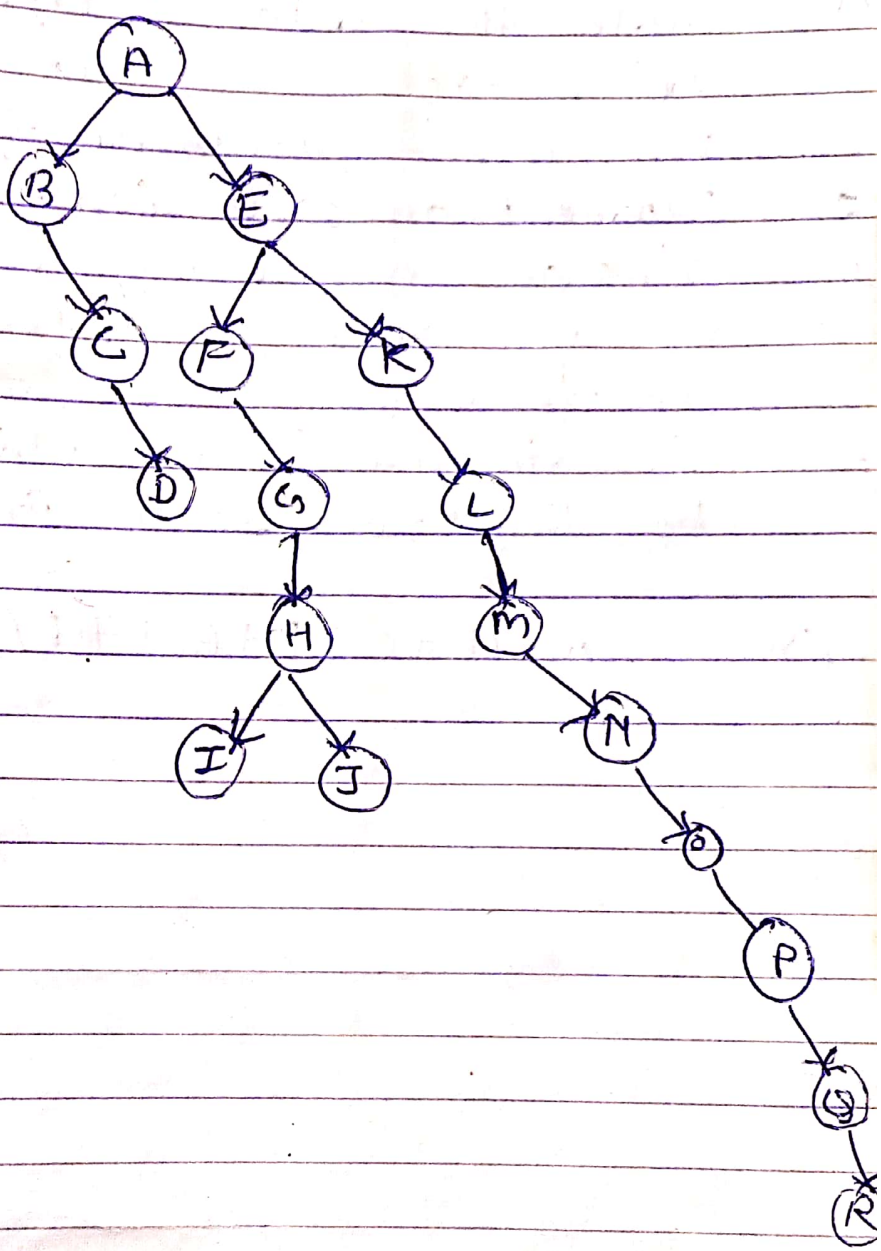


Convert forest into Binary Tree

DATE :

PAGE NO. :





DATE : PAGE NO.:

Technique of Conversion of An Expression into Binary Tree. →

Divide and conquer technique is used to convert an expression into a binary tree. Steps are

(1.) Note the order of Precedence. All expressions in Parentheses are to be evaluated first.
2. Exponentiation will come next.
3. Division and multiplication will be the next in order of Precedence.
4. Substraction and addition will be the last to be Processed.

Ex A+(B+C*D+E)+F/G

DATE : PAGE NO. :

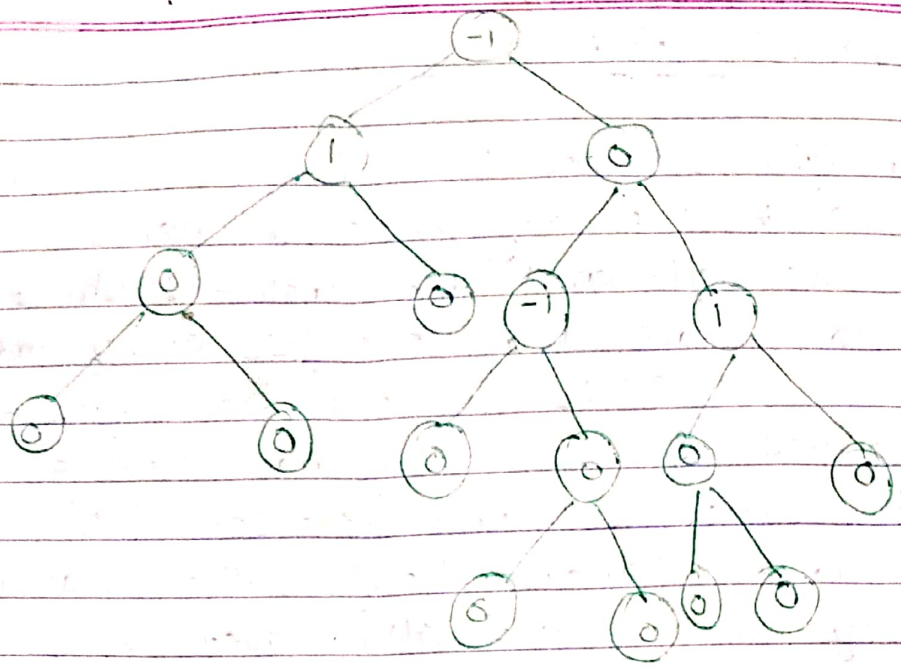
Height Balanced Trees → AVL

A Height-balanced tree is a binary tree in which the difference in heights b/w the left and the right-subtrees is not more than one for every node.

This Property was first described in 1962 by two Russian mathematicians G. M. Adel'son-Vel'skii and E. M. Landis. Therefore, the resulting binary trees and also called AVL trees in their honor.

In order to maintain the height balanced Property. A Balance Factor that indicates the difference in heights of the left and right-Subtrees. Each node in a balanced binary tree has a balance of 1, -1 or 0 depending on whether the height of its left Subtree is greater than, less than or equal to the height of its right Subtree. A Value of 1 indicates that the left child is heavier and there is a Path from root to leaf in the left child Subtree of length n, whereas the longest Path in the right child subtree is length n-1. A balance factor of 0 will indicate that the longest Paths in the two child Subtree are equal while a Value of -1 indicates that the right child Possesses the longest Path.

Balanced Binary Tree



Insertion of a Node ⟹ We can insert a node into an AVL tree by using the insertion algorithm for binary search tree in which we compare the key of the new node with that in the root and then insert the new node into left subtree or right subtree depending on whether it is less than or greater than that in the root.

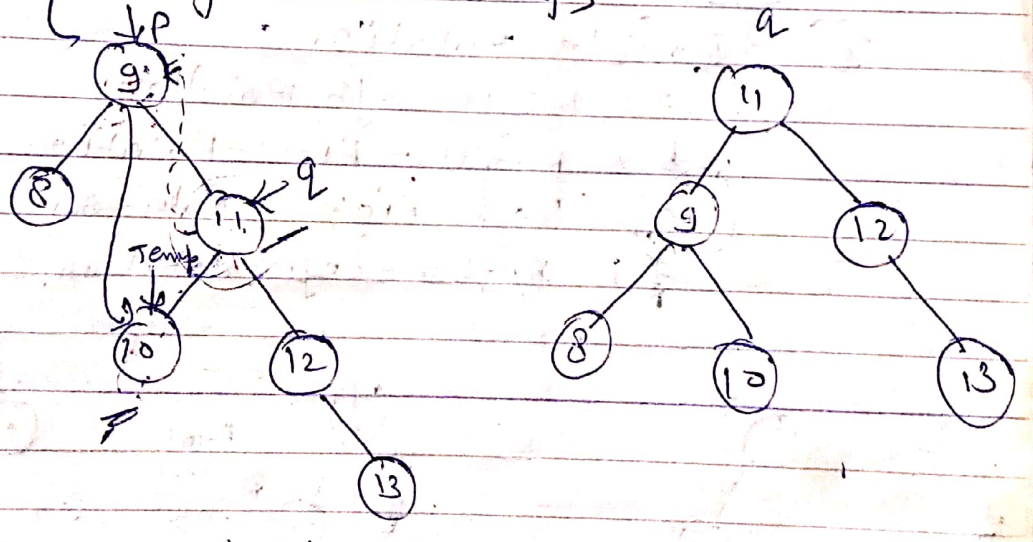
It is quite Possible that the new node can be inserted without changing the height of the subtree in which case neither the height nor the balance of the root will be changed. It is also Possible that the height of a subtree increases when we insert a new node, but it may be the lighter subtree

that has grown so only the balance factor of the root will change.

The only case that causes problem is when the new node is added to the subtree of the root which is heavier than the other subtree and the height is increased. This would cause the balance factor of the root to be 2 or -2. Whereas the AVL condition allows the balance factor to be -1, 0, or 1.

We can define the algorithm for left Rotation of a subtree rooted at P as.

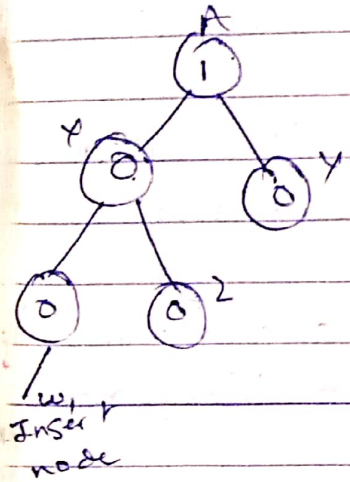
```
q = right-(P);
temp = left-(q);
left(q) = P;
right-(P) = temp;
```



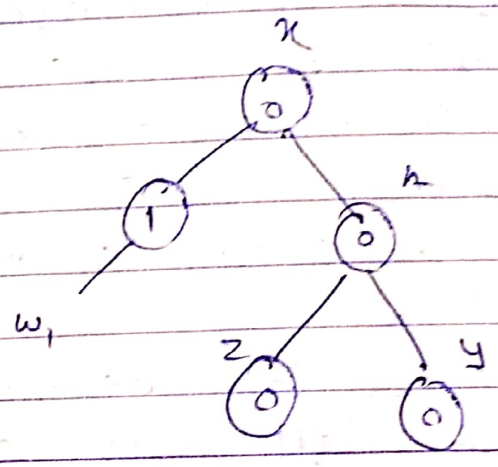
Befor rotation

Similarly we can write algorithm for right rotation of a subtree rooted at P as follows

```
q = left(P);
temp = right(q);
right(q) = P;
left(P) = temp;
```



Befor Rotation

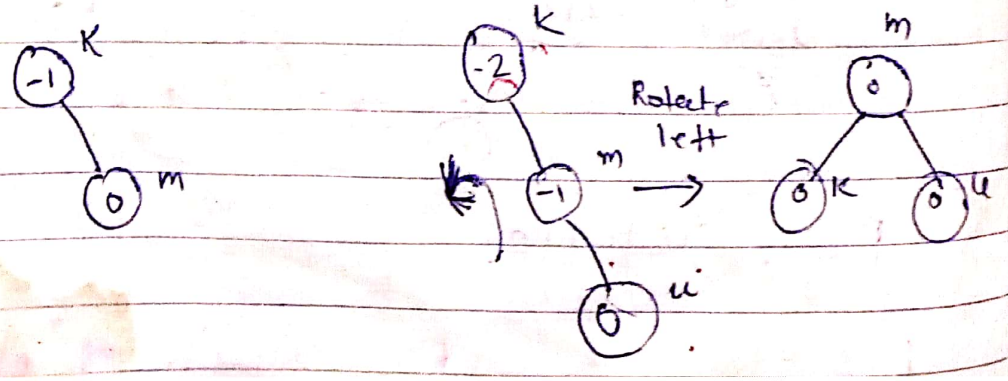


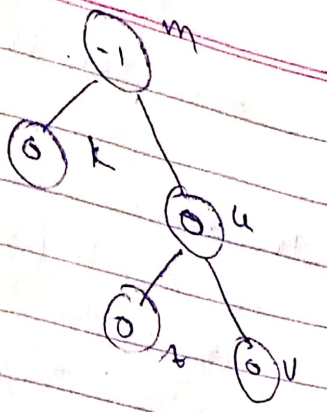
After Rotation.

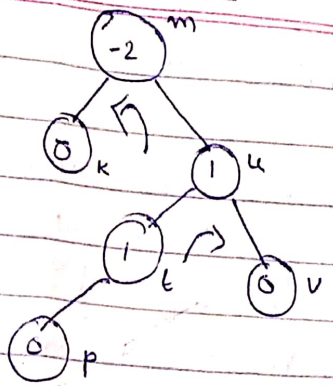
Ex Type of Rotation (~~Double R~~

1. Single Rotation
2. Double Rotation
   - (a) ~~Left~~ - Left Rotation
   - (b) Right - Right Rotation
   - (c) Left - Right Rotation
   - (d) Right - Left Rotation.

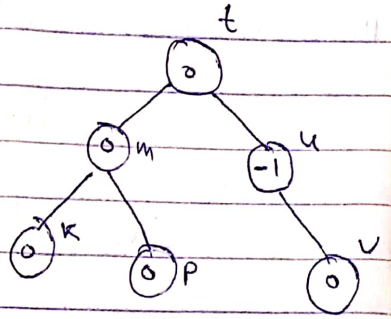
Ex







Double Rotation first we do Right Rotation and then left Rotation. this is called Right - left Rotation.



→ Delete a Node →

The root will be the only node at level 1. Each subsequent level will be as full as possible i.e. 2 nodes at level 2, 4 nodes at level 3 and so on i.e in genral there will be 2^{l-1} node at level 'l', therfore the No. of nodes from level 1 through level h-1 will be

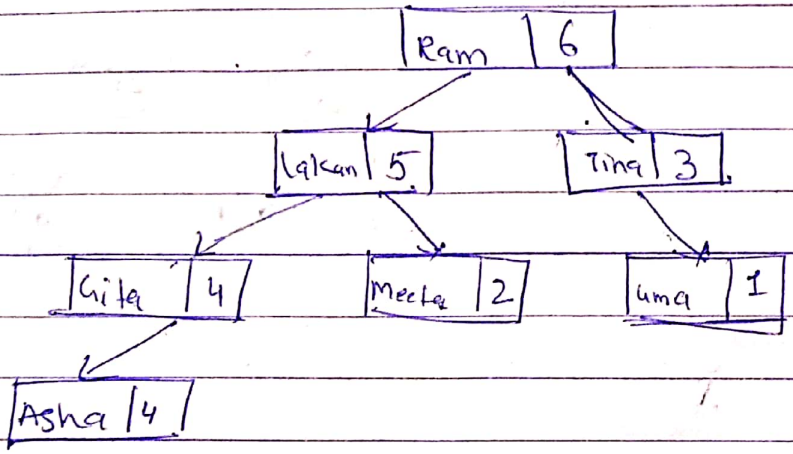
$$1 + 2 + 2^2 + 2^3 \cdots + 2^{h-2}$$

$$= \boxed{2^{h-1} - 1}$$

No. of level h may range from a single node to maximum of 2^{h-1} node

Weight Balanced Tree → A weight-balanced tree is a tree whose each node has an information field which contains the name of the node and the number of times the node has been visited.

For Example, Consider the tree given in figure. This is a balanced tree which is organized according to the number of accesses.



The rules for Putting a node in a weight-balanced tree are expressed recursively as follows:

1) The first node of tree or Subtree is the node with the highest count of no. of times it has been accessed.
2) The left Subtree of the tree is composed of nodes with value laxically less than the first node
3) The right subtree of the tree is composed of nodes with value laxically higher than the first node.

DATE : PAGE NO. :

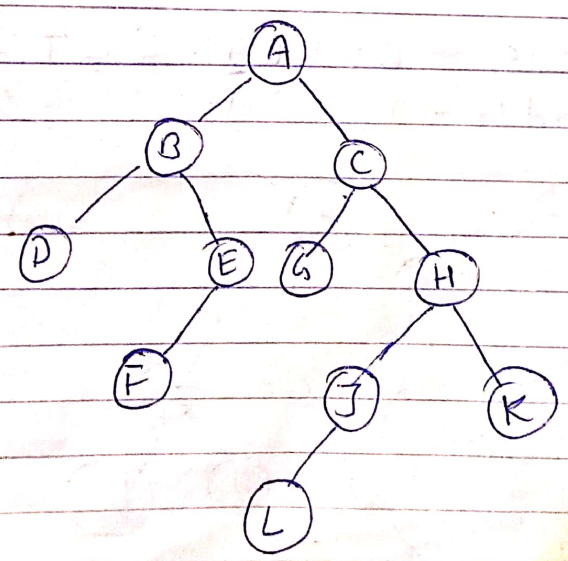
Let us now formulate steps to search and insert a node in a weight balanced tree. if a requested node is found, its weight is incremented by one. if the accessed node is not found, it is added at the appropriate leaf position and its weight is initialized to one.

A general Algorithm to access a node in a weight balanced tree is a follows:

(1) Search for the given target key and determine the search Path followed.

(2) if the given target key is found then increase its weight by one. rebalance the tree, if needed.
else insert new node.
Set its weight to one.

Tree Traversals →



Inorder → Left - Root - Right.

D B F E A G C L J H K

Post order → Left - Right - Root

D F E B G L J K H C A

Pre order → i.e. Root - left - Right

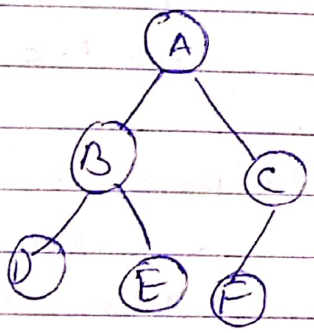
A B D E F C G H J L K.

Ex

Preorder - A B D E G F

Postorder - D E B F C A

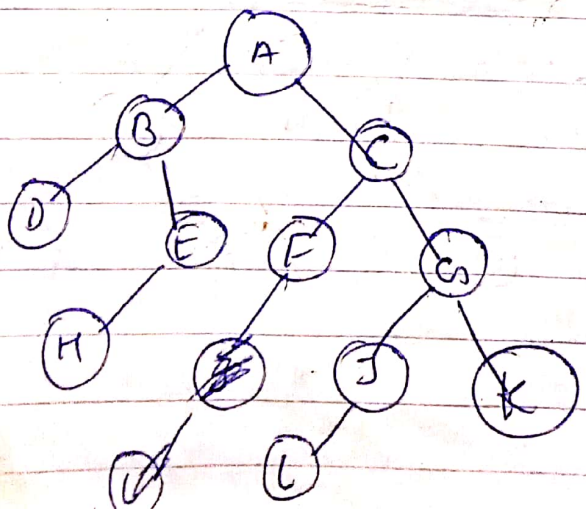
Inorder - D B E A F C



Ex

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

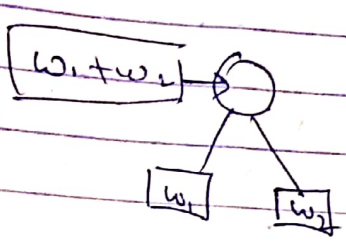
Preorder - A B D E H C F G J L K



Huffman Algorithm →

① Suppose there are n weights $w_1, w_2, \ldots w_n$

② Take two minimum weights among the n given weights, suppose w_1 and w_2 are first two minimum weights then subtree will be



③ Now the remaining weights will be $w_1+w_2, w_3+w_4, \ldots w_n$

④ Create all subtree at the last weight.

Huffman Algorithm constructing the tree from the bottom up rather than top down approach.

$(A+B) * C/D$

$D/C * (B+A)$

D	(	D	1
/	C/	/DC	
C			
*	(/*		