Unit-5 (Tree) - Set-1

Tree is a non-linear data structure which allows a parent child relationship, thus allows us to arrange our records, data and files in hierarchical fashion.

Tree is a finite set of one or more data items called nodes such that the special data item called the root of the tree and the remaining data items are partitioned into number of mutually exculitive subset each of which is itself a tree and called subtrees.

a) b) c)

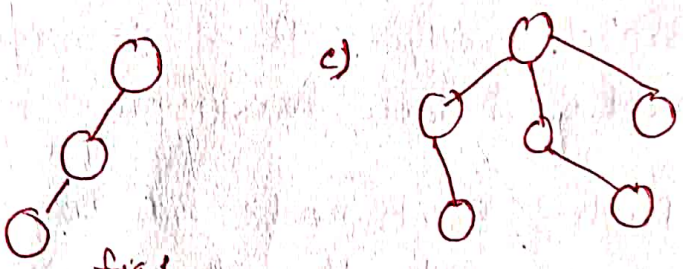


fig 1

Basic Terminology :-

1) **Root**- It is the first node in the tree that has no parent.

2) **Node**- Each data item in a tree is called a node. It specifies the information about the data and the links to other data items.

3) **Parent**- parent of a node is the immediate predecessor of that node.

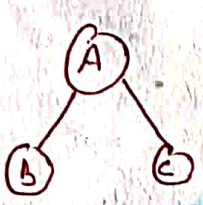


fig.2

A is the parent of B and C.

4) **Child**- The immediate successor of a node are called childs of that node. B and C are childs of A.

5) **Leaf node (External node) or terminal node**- A leaf node is that node, which doesnot have any child node.

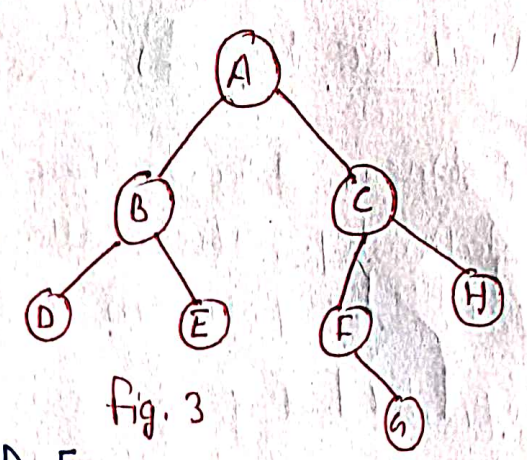


Fig. 3

leaf nodes are D, E, G, H

6) Siblings – The child nodes of a given parent nodes are called siblings.

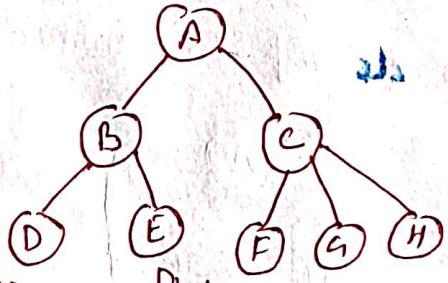
(B, C), (D, E), (F, H) are siblings.

7) Branch or edge – It is a connecting line between two nodes. A node can have more than one edge.

8) Path – Each node has to be reachable from the root through a unique sequence of edges called a path. The number of edges in a path is called the length of path.

9) Degree of a node – It is the number of childs of that node.

Ex→



Degree(A) = 2
Degree(B) = 2
Degree(C) = 3
Degree(D, E, F, G, H) = 0

fig fig 4.

10) Degree of a tree – It is the maximum degree of a node in a given tree. In fig 4. The degree of tree = 3.

11) Level :- The entire tree structure is leveled in such a way that the root node is at level 0, and its immediate children are at level 1 and so on.

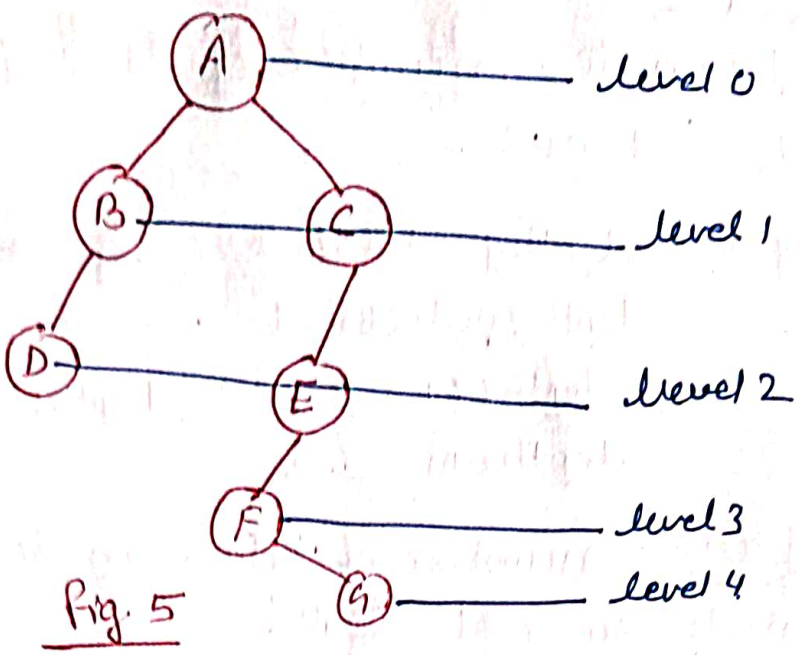


Fig. 5

1) Ancestor- x is said to be ancestor of Y, if Y lies any of the subtree of node x.

3) Descendent- x is say to descendant of Y if x lies any of the subtree of the node Y.

4) Internal node or non-terminal node- Any node whose degree is not zero is called the internal node.

15) Empty or NULL tree- which has no node.

Rooted Tree- which that only one node.

16) Height of node- number of edges in longest path from that node to leaf node.

In fig. 5 Height of (A) - 4
Height of (B) - 1
Height of (C) - 3
Height of (D) - 0
Height of (E) = 2
Height of (F) = 1
Height of (G) = 0

17) Height of tree- No. of edges in longest path from root to leaf nodes.

Height of tree - 4

18) Depth of ~~tree~~ node – number of edges from that node to root node.

in fig 5. depth of node(A) = 0 Depth(E) = 2
depth of node(B) = 1 Depth(F) = 3
depth(C) = 1 Depth(G) = 4
depth(D) = 2

19) Depth of tree – number of edges in longest path from leaf ~~left~~ node to root node.

In fig. 5 depth of tree = 4

	h given		n given	
	max. nodes	min. nodes	max height	min height
Binary tree	$2^{h+1}-1$	$h+1$	$n-1$	$\log_2(n+1)-1$
Full tree	$2^{h+1}-1$	$2 \cdot h+1$	$(n-1)/2$	$\log_2(n+1)-1$
Complete binary tree	$2^{h+1}-1$	2^h	$\log_2 n$	$\log_2(n+1)-1$

Note: max nodes means min height.

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

$2^{h+1} = n+1 \Rightarrow h+1 = \log_2(n+1) \Rightarrow \boxed{h = \log_2(n+1)-1}$

min nodes means max height

$h+1 = n \Rightarrow \boxed{h = n-1}$

same procedure follow for complete binary tree and full tree.

Degenerate tree (Pathological tree) – A tree in which every internal node has one child.

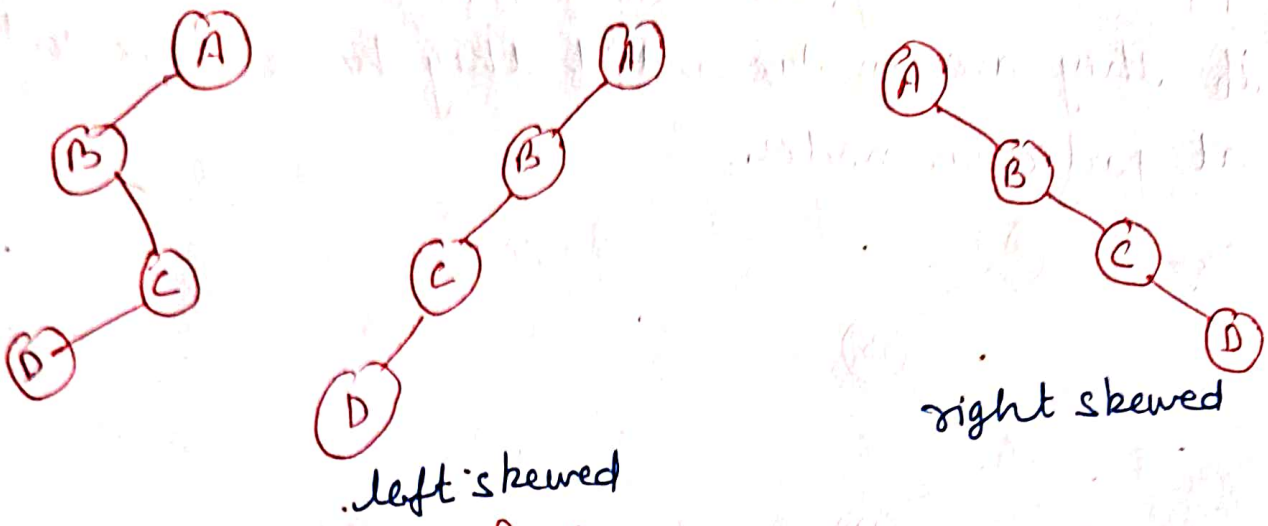
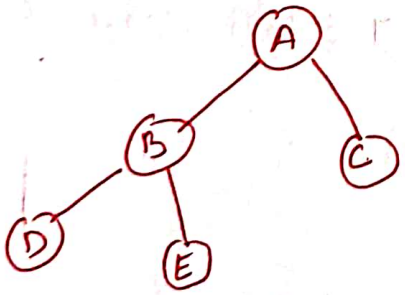


Fig 6

Note: The nodes with same level number are said to belong to the same generation.

Binary Tree – A binary tree T is a tree which contains root node and two disjoint binary trees called the left subtree and right subtree. The degree of binary tree is at most two.

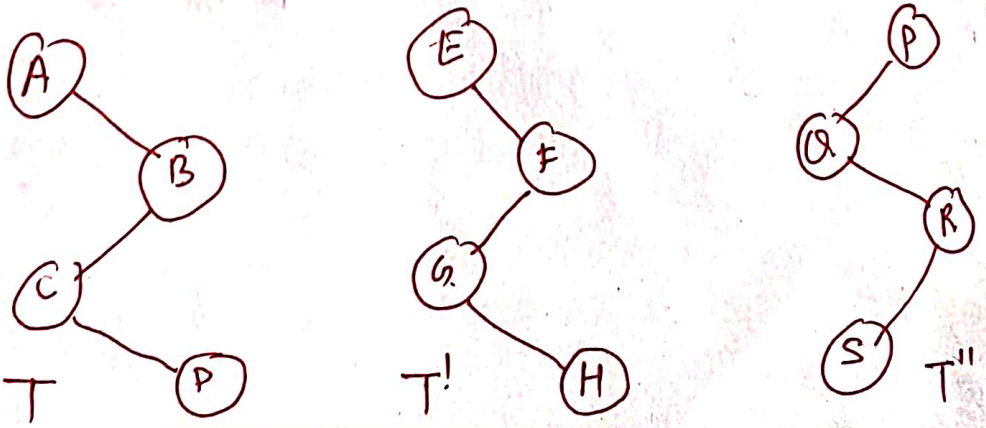
Ex→



max. possible node at any level = 2^l

max. no. of a node in a binary tree with height = $2^{h+1} - 1$.

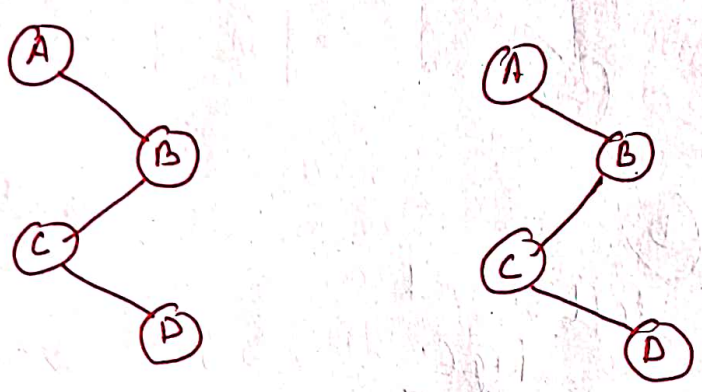
Similar tree – Binary tree T and T' are said to be similar if they have same structure and shape.



T and T' are similar but T & T'' are not similar.

Copy tree- Binary Tree T and T' are said to be copies if they are similar and if they have same contents at particular nodes.

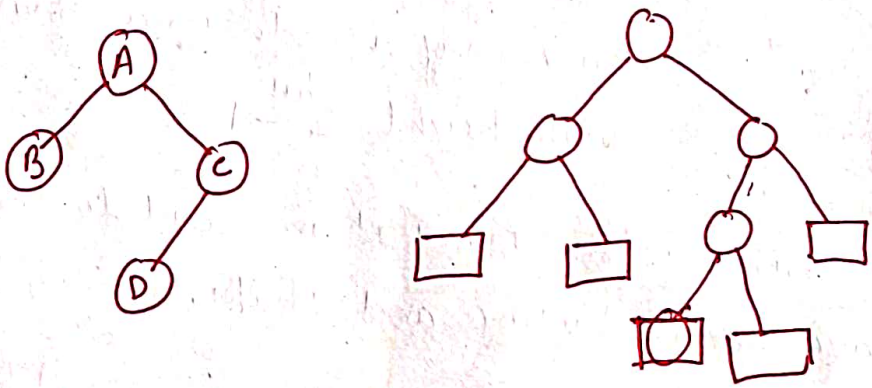
Ex-



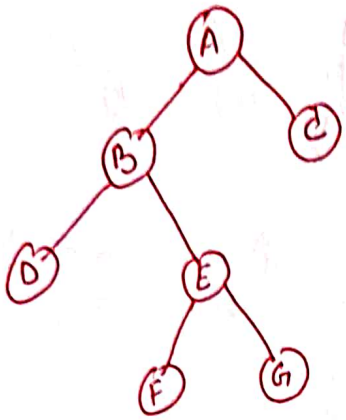
T and T' are copies.

Type of binary tree-

1) **Extended binary tree:-** A binary tree is said to be extended binary tree or 2 tree if each node has either 0 or 2 children. In such a case, the node with two children are called internal node and the node with 0 children are called external node.



Strictly binary tree:- if internal node have it's non empty left and right children then it is called strictly binary tree.

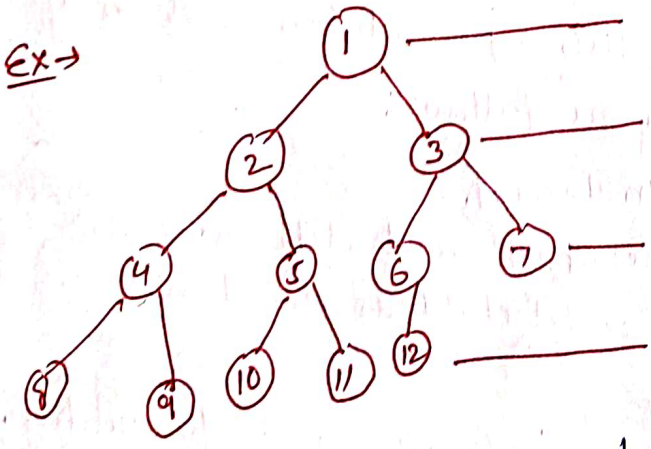


This is also called 2-tree or full tree.

Complete binary tree:- In binary tree each node can have at most two children. Accordingly one can show that level r of T can have at most 2^r nodes.

The tree T is said to be complete if all it's level, except possibly the last, have maximum number of possible node, and if all the nodes at the last level appear as far left as possible. Thus there is a unique complete tree T_n with exactly n nodes.

Ex→



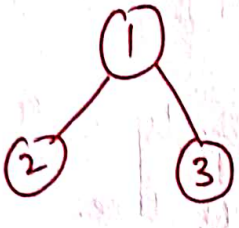
max. possible node = 1
" " " = 2
" " " = 4
" " " = 8

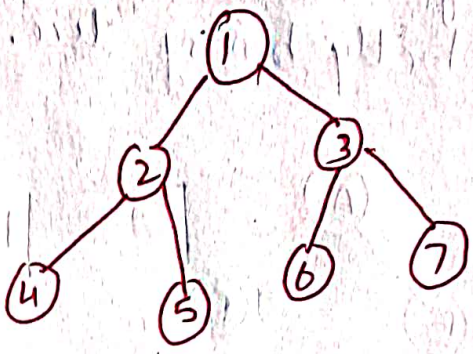
In a complete binary tree for any node k has its parent at $[K/2]$ place. The Depth (height) of complete binary tree with n nodes is given by

$$D_n = \lfloor \log_2 n \rfloor$$

Perfect tree- The tree T is said to be full if all its level have maximum number of possible nodes.

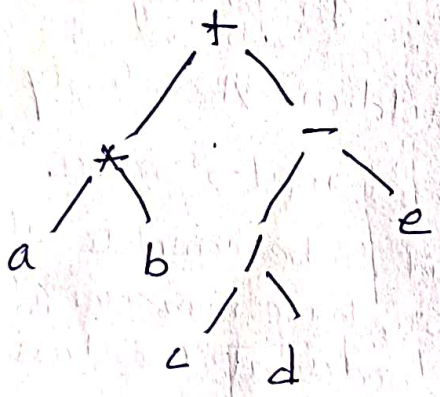
Ex→





Representation of algebric expression in the form of tree:

(a*b) + ((c/d) - e)



Representation of binary tree :-

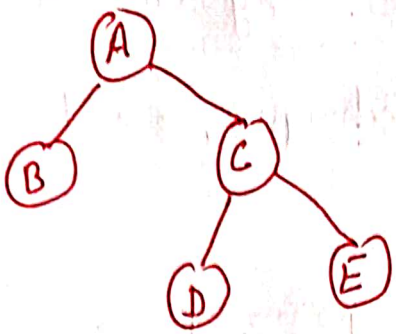
1) Sequential representation (static or array) :-

Let T is a binary tree. This representation uses only a single linear array tree as follows:

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

Again null is used to indicate an empty subtree. If Tree[i] = NULL then tree is empty. If height of a tree is h then size of array to represent binary tree will be $2^{h+1} - 1$.

Ex →

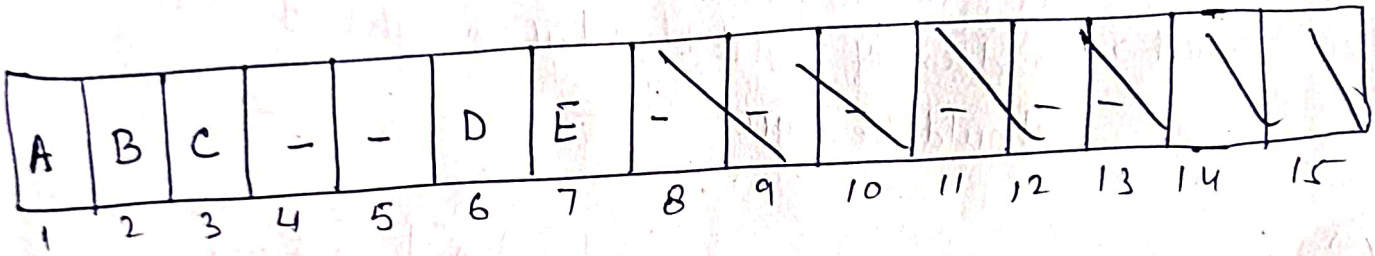


n	h
1	0

height of tree = 2

So size of array = $2^{2+1} - 1 = 7$

A is a root so it will store at T[1].



Advantages: ~~1) Not suitable for normal binary tree but it~~

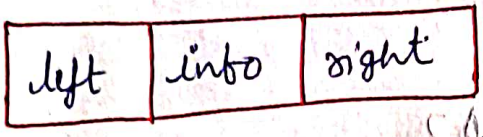
1) Data are stored without any pointer
2) Any node can be calculated from any other node by calculating the index.
3) Programming language which do not support dynamic memory allocation use this type of representation.
4) Simplicity.

Disadvantages-

1) Not suitable for normal binary tree but it is ideal for complete binary tree.
2) most of the array entries are empty, i.e. wastage of memory.
3) Addition and deletion of nodes are inefficient because of the data movement in the array.

» (2) Representation of Tree using Linked List (Dynamic representation)

In this a location technique, a node in a tree contains three fields:



1) left 2) Info 3) right

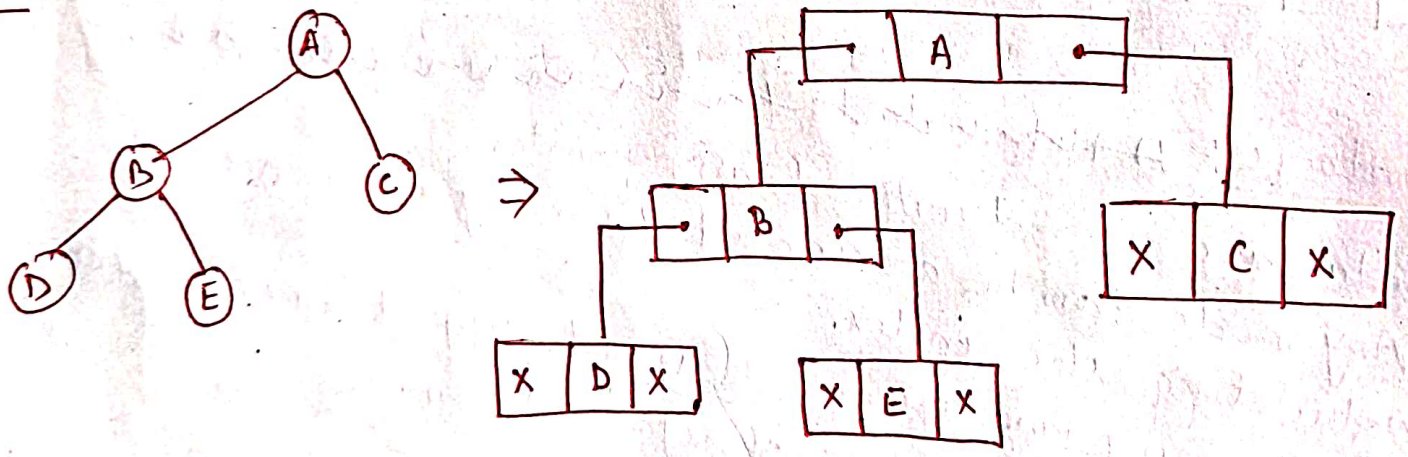
where

Info - information field of a node

left → address of left child node. if no left child then it contains NULL.

right → address of right child. if no right child then right should be NULL.

Ex→




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

Advantages	Disadvantages.
1) No wastage of memory 2) Enhancement of tree is possible 3) Insertion and deletion involve no data movement.	1) In this pointer fields are involved which take more space then just data field. 2) programming language that do not support dynamic memory allocation have difficult to implement tree

path length Extended binary tree:-

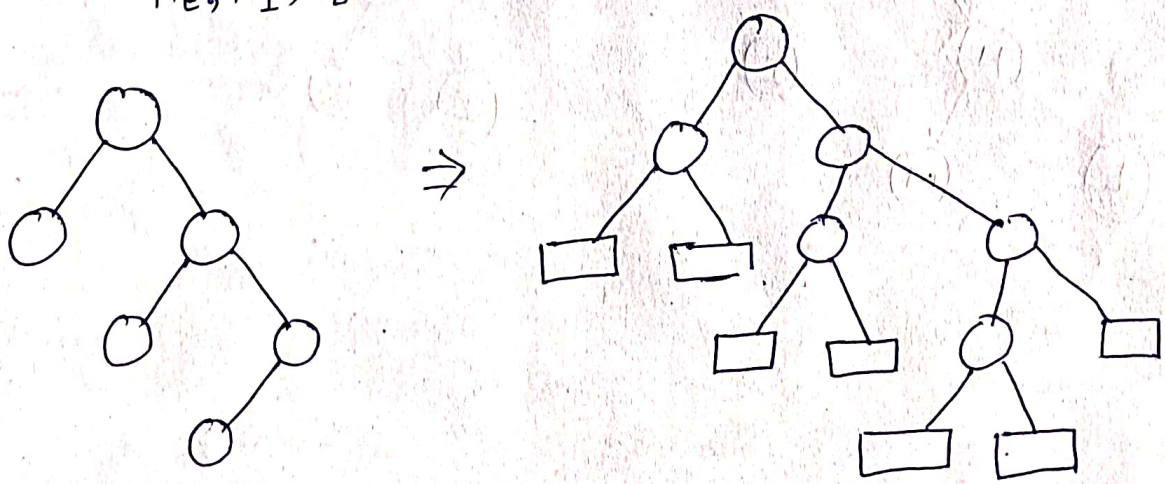
External Path length (L_E) = sum of all path length from each external node to root.

Internal Path length (L_I) = Sum of all path from each internal node to root.

$$N_E = N_I + 1$$

$$L_E = L_I + 2N_I$$

Example- To make Extended binary tree & find out the. N_E, N_I, L_E, L_I.

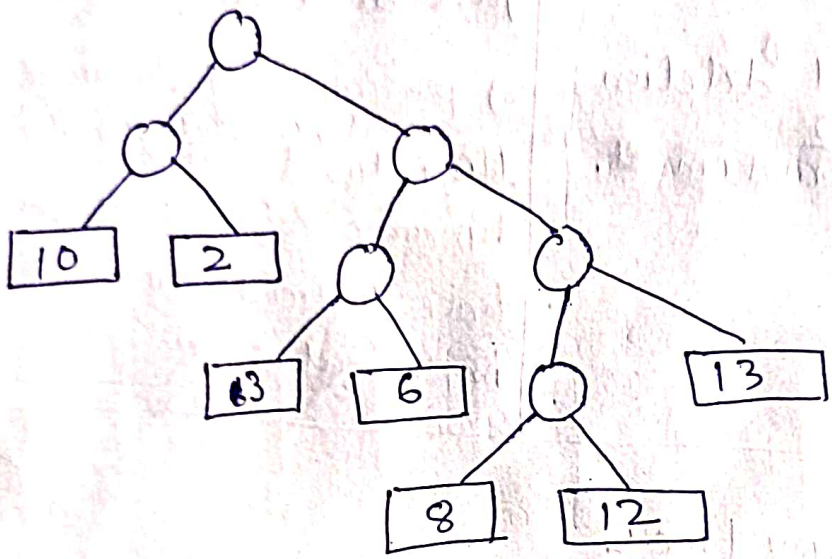


$N_I = 6$ $\quad\quad$ $N_E = 7$

$P_E = 2+2+3+3+3+4+4 = 21$

$P_I = 1+1+2+2+3 = 9$

Weighted External path length :-

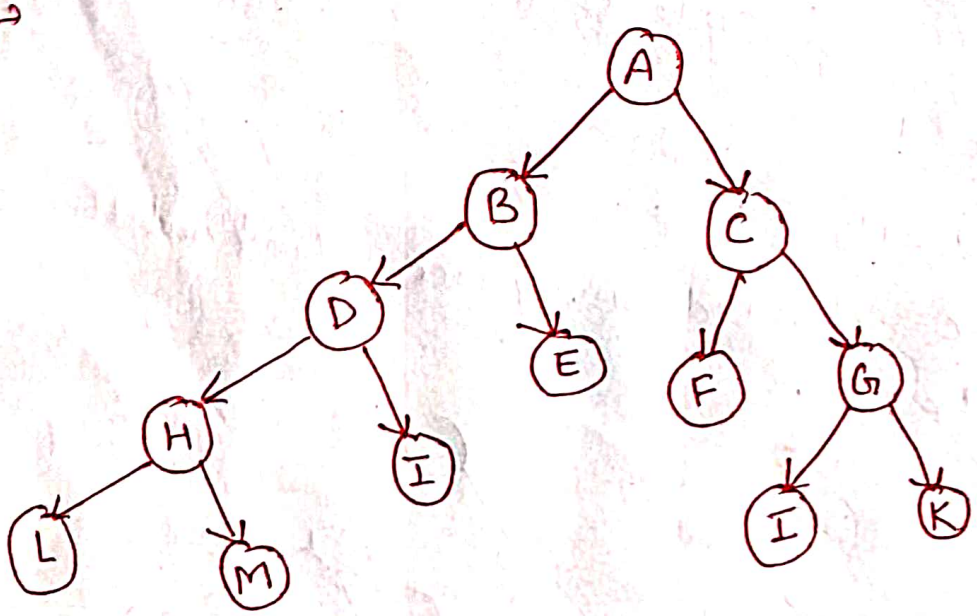


$P_{WE} = 10\times2 + 2\times2 + 3\times3 + 6\times3 + 13\times3 + 8\times4 + 12\times4$

$= 20 + 4 + 9 + 18 + 39 + 32 + 48$

$= 170$

Q:→



Find

1) Right & left descendent of node B.
2) write all siblings
3) H & F are same generation, D & G some generation?
4) leaf nodes of tree.
5) Write internal nodes of a tree
6) Which is root?
7) Depth of tree
8) C is a father of whom?
9) Path length of A-J.
10) Write child of B
11) Which is degree of Tree.
12) Degree of D, F, A.
13) Height of tree.

Tree traversal Algo→

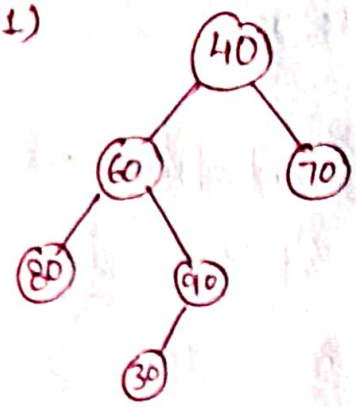
1) Pre-order traversing.
2) In-order traversing.
3) Post order traversing.

Pre-order traversing- We can traverse any binary tree T with Root R in preorder by following rules:

1) Procen the root R
2) Traverse the left subtree of R in Pre order.
3) Traverse the right subtree of R in Prerorder.

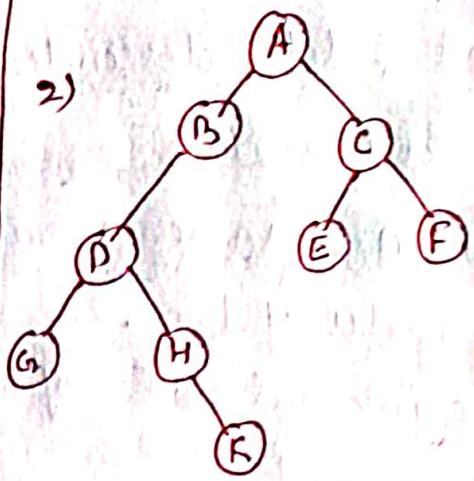
Root, left, right

Ex- 1)



40, 60, 80, 90, 30, 70

2)



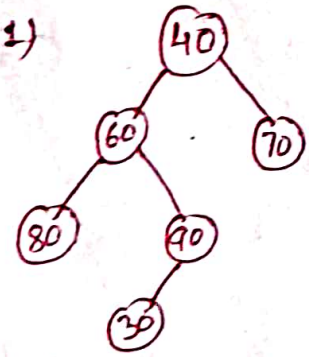
A B D G H K C E F

Inorder traversing → We can traverse any binary tree T with root R in inorder by following rules:

1) Traverse the left subtree
2) Procer the root R
3) Traverse the right subtree

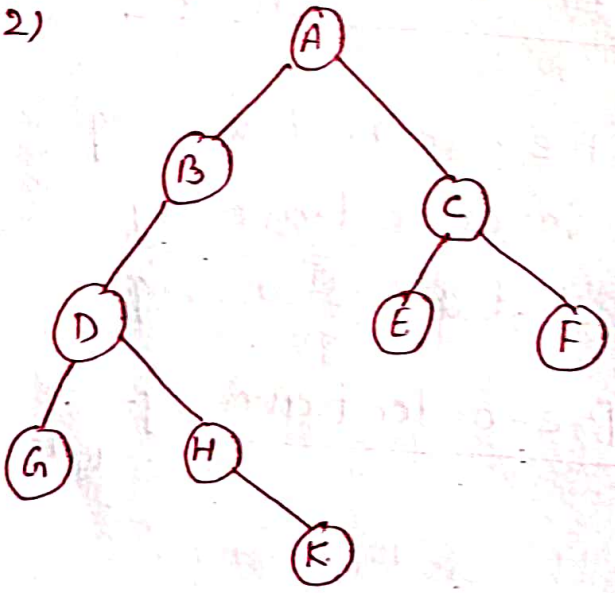
Left, Root, Right

Ex→ 1)



80, 60, 30, 90, 40, 70

2)

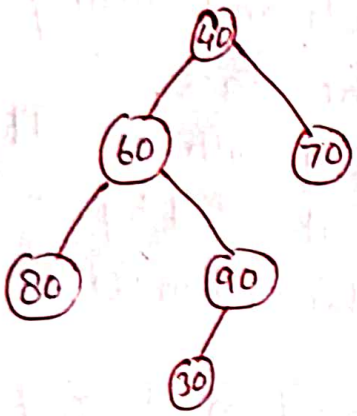


G, D, H, K, B, A, E, C, F

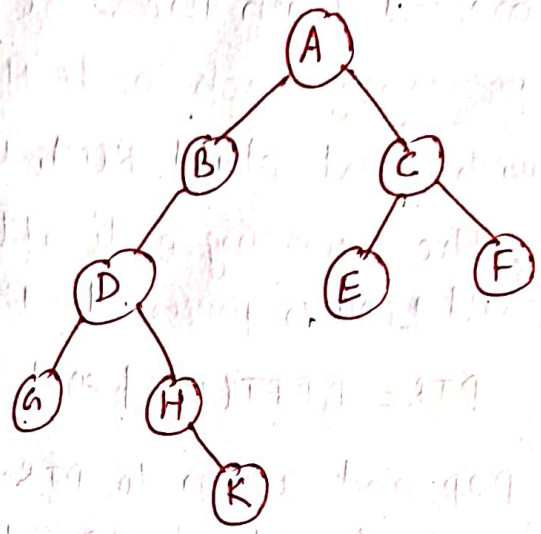
Post order traversing– We can traverse any binary tree T with root R in post order by following rules:

1) Traverse the left subtree of R
2) Traverse the right subtree of R
3) Process the Root R.

Ex→



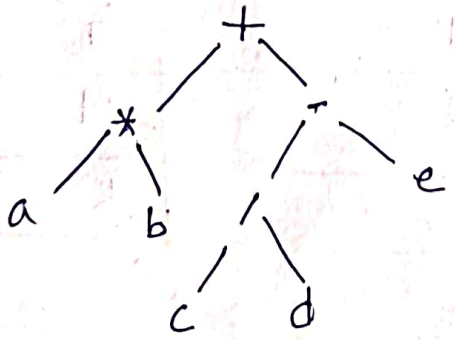
80, 30, 90, 60, 70, 40



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

Q:→ Represent following expression in tree form and also write Pre order, Post order and Inorder.

(a*b) + (c/d - e)



preorder– + , * , a, b, - , / , c, d , e

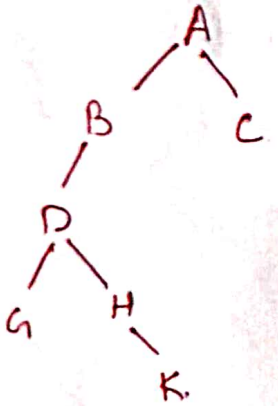
Inorder → a, *, b, +, c, /, d, -, e

postorder – a, b, *, c, d, /, e, -, +

pre order traversing Algorithm

1. Initially push NULL on to stack then set ptr = Root.
2. repeat steps 3 & 4 while (ptr ≠ NULL)
3. proceed down the left most path rooted at PTR, processing each node N on the path and pushing each right child R(N), if any on the stack.
   The traversing ends after a Node N with no left child L(N) is processed. Thus PTR is updated by
   PTR = LEFT[PTR] and traversing stops when LEFT[PTR] = NULL
4. POP and assign to PTR the top element on stack. If PTR ≠ NU then return to step 3. otherwise exit.

Ex→



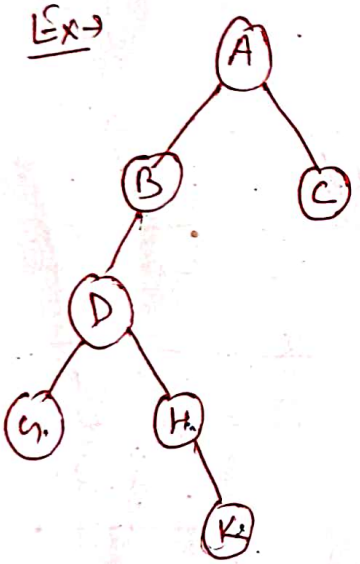
STACK	Processed Node
∅	
∅ C	A
∅ C B	B
∅ C H	D
∅ C H	G
∅ C K	H
∅ C	K
∅	C

pre order → A B D G H K C

Traversing Algorithm for Inorder –

1. Initially push NULL on to stack and then set ptr = Root.
2. repeat steps 3 & 4 until NULL is popped from stack
3. proceed the left most path rooted on PTR, pushing each node N on to stack, and stopping when a Node N with no left child is pushed onto stack.
4. Pop and process the nodes on stack. If NULL is popped then exit. if a node N with right child R(N) is processed. set PTR = right[PTR] and return to step 3.

Ex→



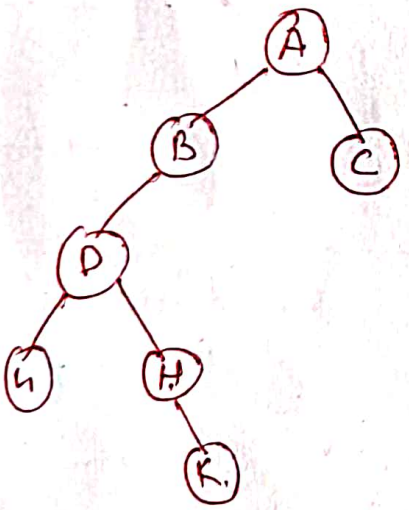
STACK	Processed Node
∅	
∅ A	
∅ A B	
∅ A B D	
∅ A B D G	
∅ A B D	G
∅ A B	D
∅ A B H	
∅ A B	H
∅ A B K	
∅ A B	K
∅ A	B
∅	A
∅ C	
∅	C

Inorder → G D H K B A C

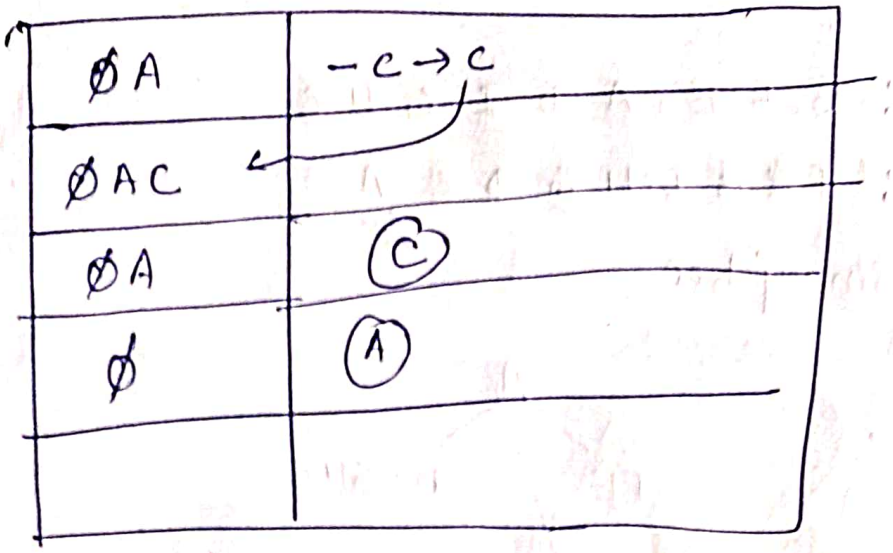
post order traversing Algo→

1. Initially push NULL onto stack. and set PTR = Root
2. repeat steps 3 & 4 unhill NULL is popped from stack.
3. Proceed the left most path rooted at PTR, At each node N of the path, PUSH N onto stack and if N has a right child R(N). PUSH -R(N) onto stack.
4. Pop and process positive nodes on stack. if NULL is popped then exit. If negative node is popped, i.e., if PTR = -N for some node N, set PTR = N. by ~~and~~ assigning PTR = - PTR and return to step3.

Ex→



STACK	processed Node
Ø	
ØA	
ØA-C	
ØA-CB	
ØA-CBD	
ØA-CBD-H	
ØA-CBD-HG	
ØA-CBD-H	G
ØA-CBD	-H → H
ØA-CBDH	
ØA-CBDH-K	
	-K → K
ØA-CBDHK	
ØA-CBDH	K
ØA-CBD	H
ØA-CB	D
ØA-C	B



postorder: G K H D B C A

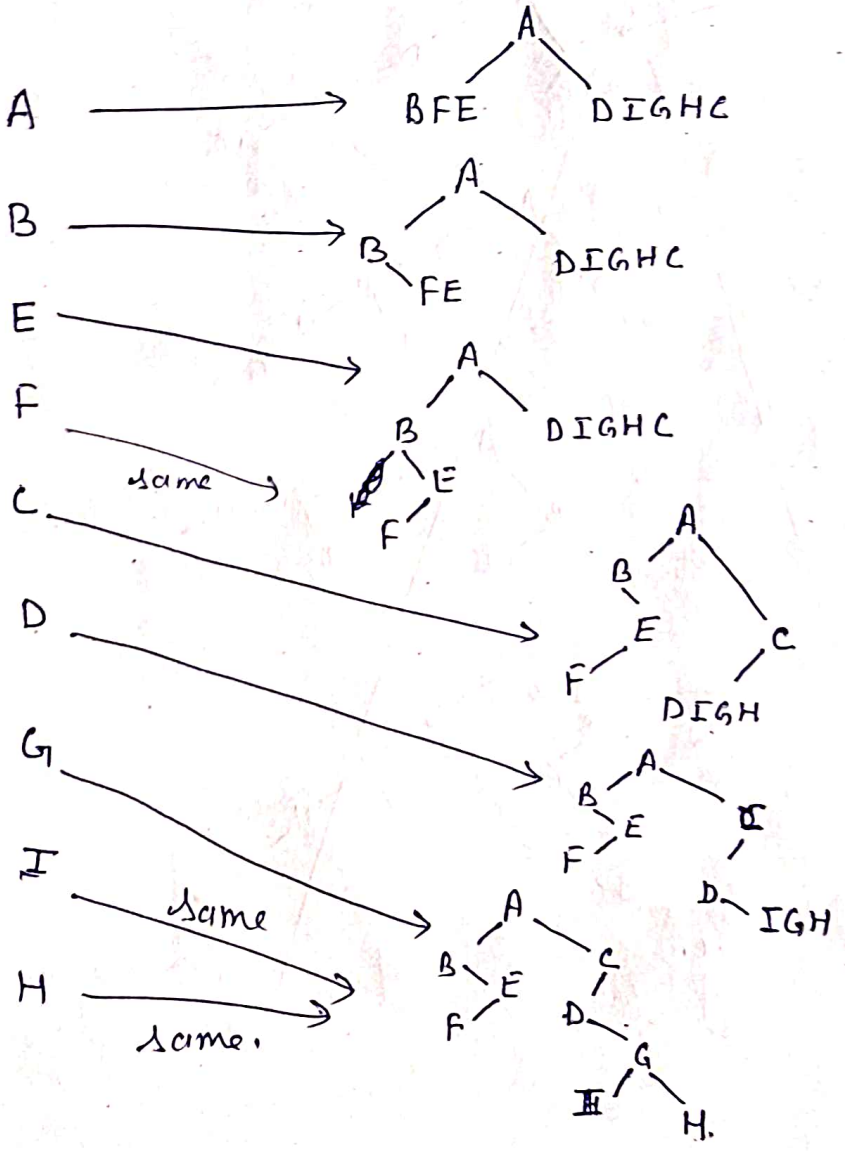
Q:- consider

Inorder: B F E A D I G H C

Preorder: A B E F C D G I H

create binary tree.

Ans-



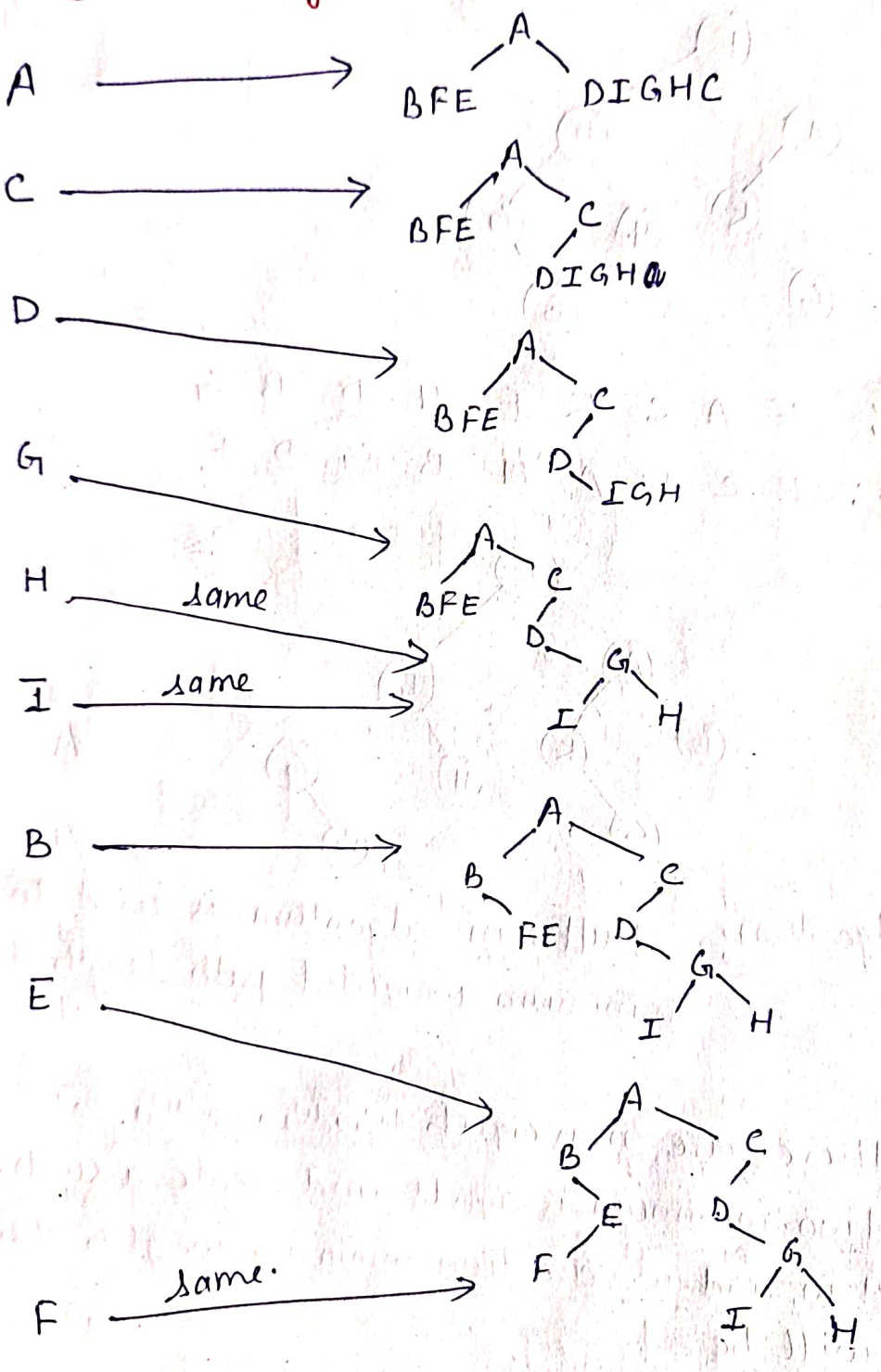
Q:- consider

Inorder: B F E A D I G H C

Postorder: F E B I H G D C A

create binary tree.

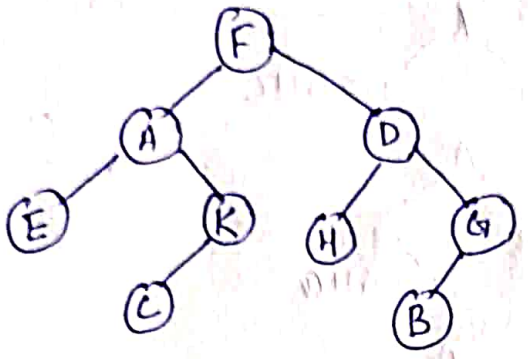
Ans



Practise problems:

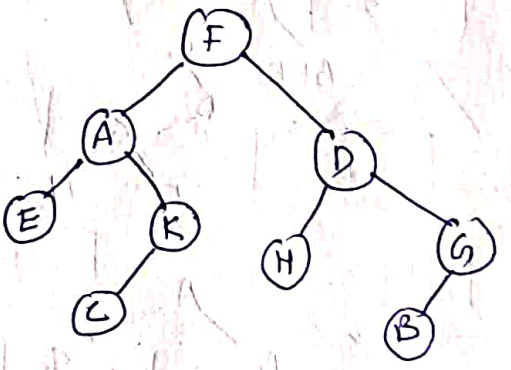
1) In order : E A C K F H D B G
Preorder : F A E K C D H G B
Create binary tree.

Sol:



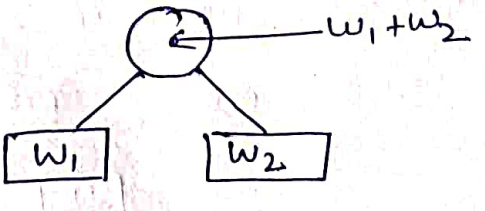
(2) Inorder: E A C K F H D B G
post order: E C K A H B G D F

Sol:



Huffman algorithm:→ Huffman algorithm is used to find minimum weighted path length for a tree.

1. Let there are n weights $w_1, w_2, \ldots w_n$.
2. Take two minimum weights and create a subtree.
Let w_1 and w_2 are two minimum weights then the subtree will be



3. now the remaining n-1 weights will be $w_1+w_2, w_3, w_4, \ldots w_n$.

4. create all subtrees at the last weight.

Example:

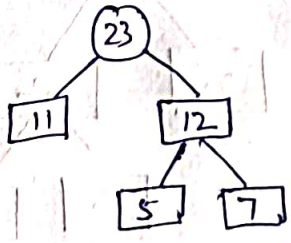
Nodes:	A	B	C	D	E	F	G
weights:	16	11	7	20	25	5	16

First method

Sol: 1. Take two nodes of minimum weights as 7 and 5.

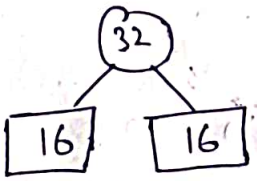
now list is 12, 16, 11, 20, 25, 16

2. Take two minimum weights as 11, 12 create subtree



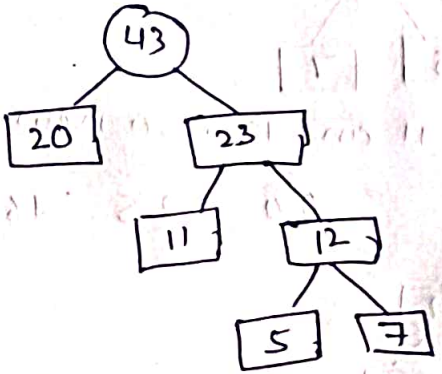
now list is 23, 16, 20, 25, 16

3. take two min weights 16, 16 & create subtree.



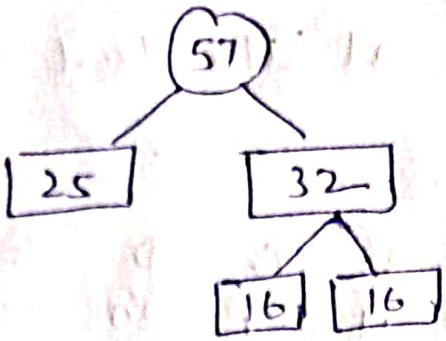
now list is 32, 23, 20, 25

4. Take two min. weights 20, 23, create subtree



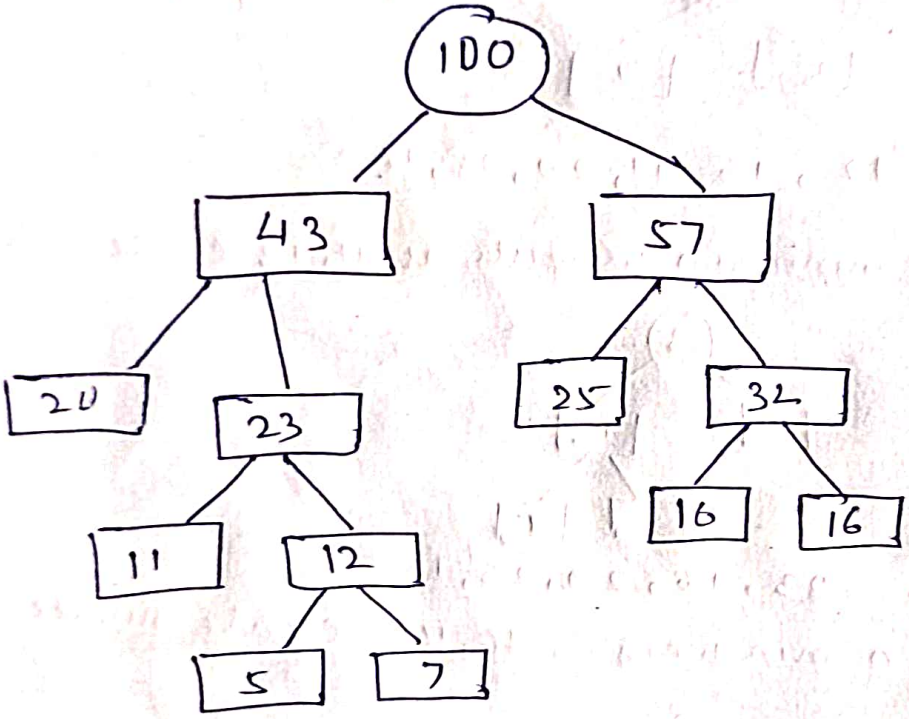
List is 43, 32, 25

5. Take two min weights 25, 32 create subtree



List is 43, 57

6. take two min weights 43, 57 create subtree

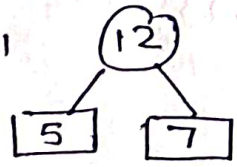


Second method

16 11 7 20 25 5 16

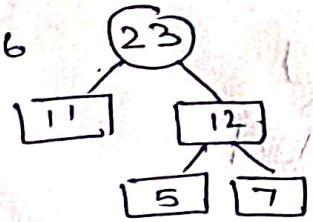
1. min weights 5 and 7 so new list is

16 11 12 20 25 16

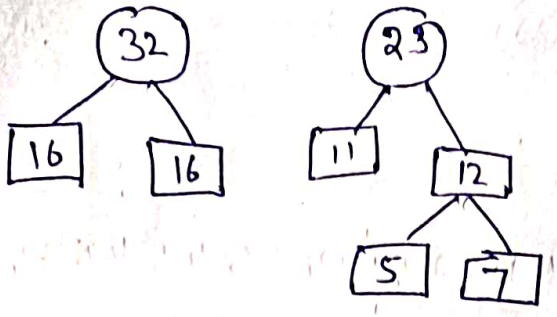


2. min weights are 11 and 12. So new list is

16 23 20 25 16

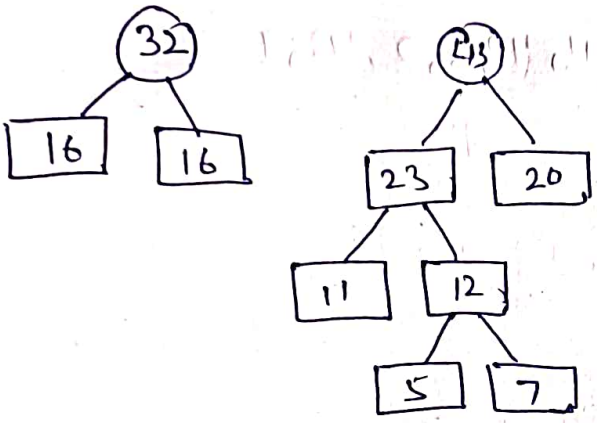


3. min weights are 16 and 16. new list is



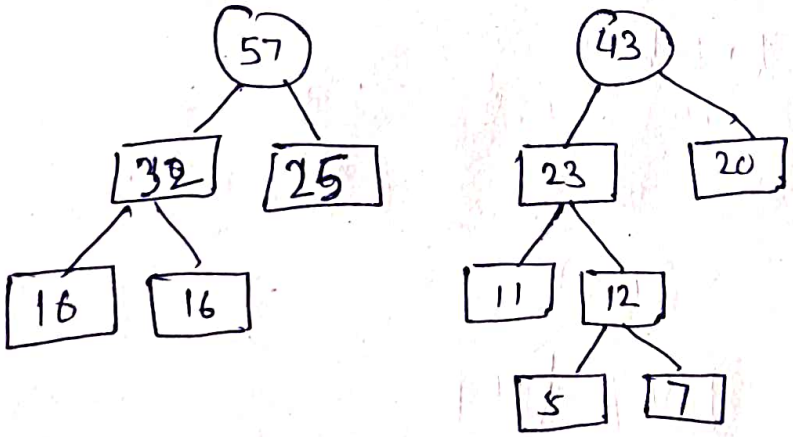
20 25

4. min weights are 20 and 23. new list is

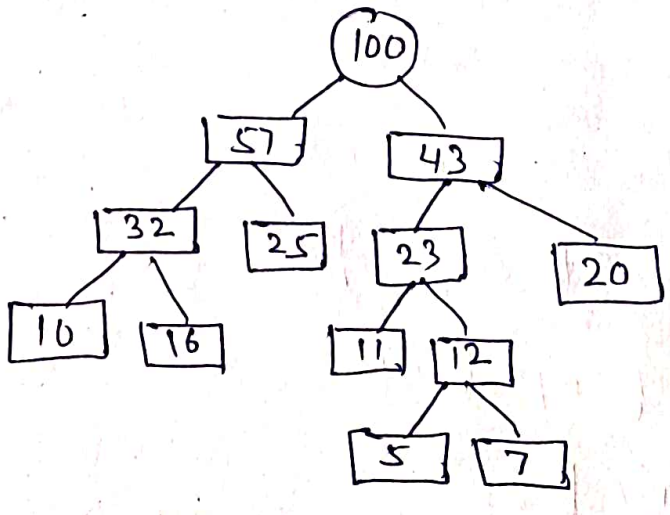


2, 5

5. min. weights are 25 and 32 new list is



6. min weights are 57 and 43. new list is



Threaded binary tree :-

4 1 1 2 3

5, 8, 16, 17, 24, 29, 30, 34, 35, 37, 43, 46, 47, 49, 57, 54, 55, 56, 57, 58.

5 1 1 2 3 (ASM)

57, 30, 27, 34, 35, 49, 11, 48, 6, 2, 43, 9

Unit-5 (Tree) Set-2

Threaded Binary Tree- In linked list representation of any binary tree, half of the entries in the pointer field left and right will contain NULL entries. This space is may be more efficiently used by replacing the NULL entries by special pointers called threads, which points to the node higher in tree, such trees are called threaded tree.

There are many ways to thread a binary tree.

(1) **Right Threaded tree:-** The right NULL pointer of each node can be replaced by a thread to the successor of that node under Inorder traversal called a right thread and tree will called a right threded tree.

ex→

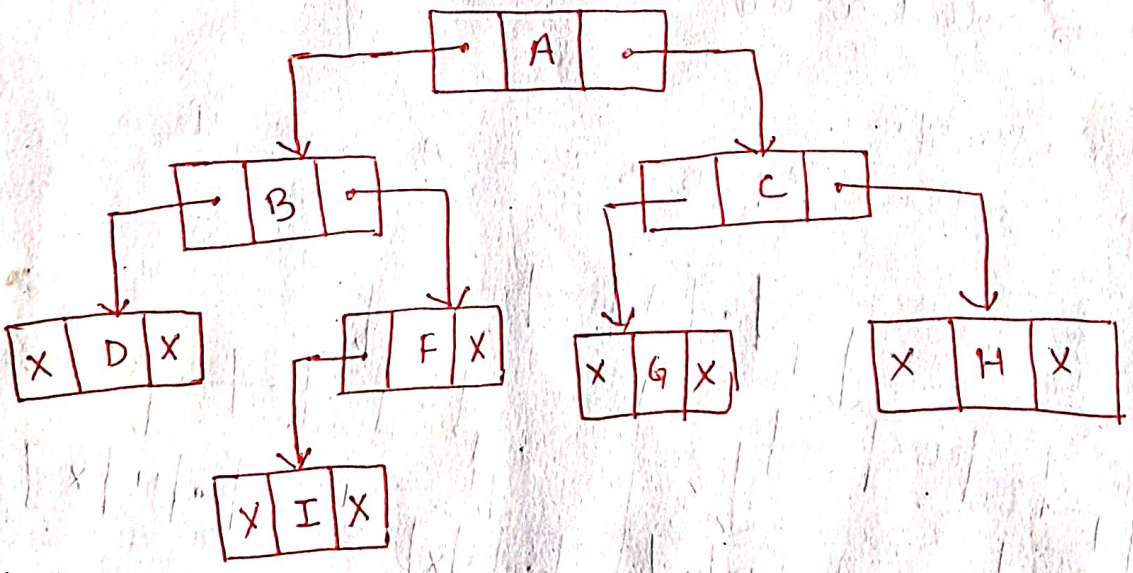
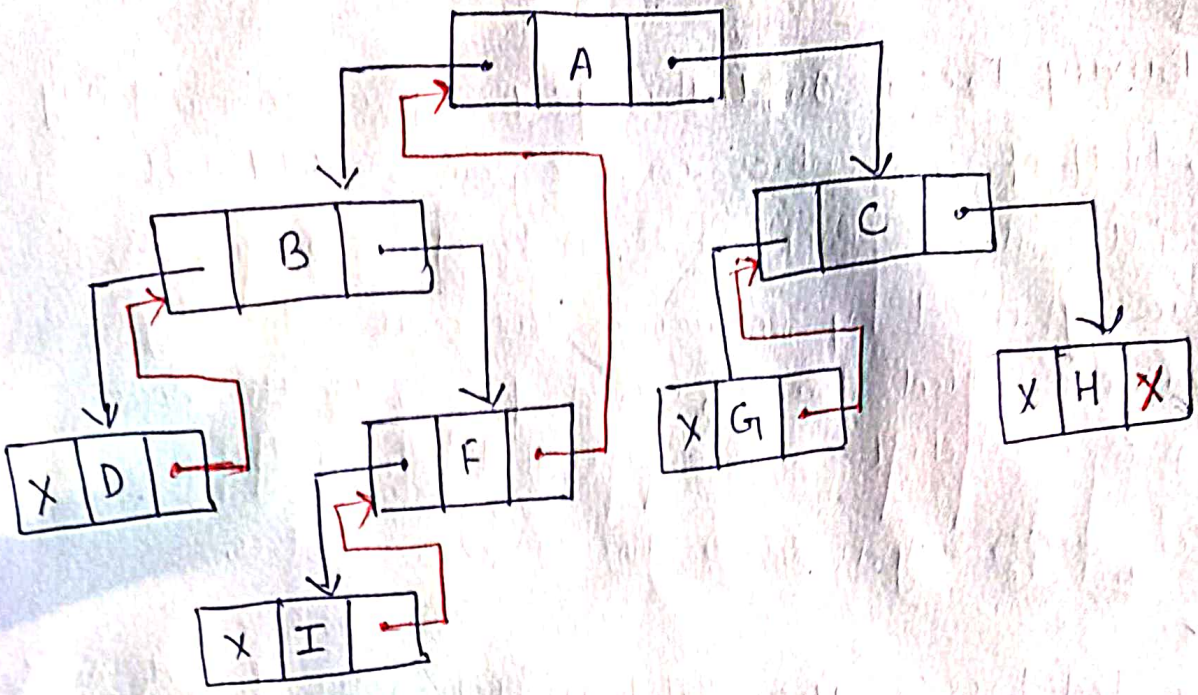


fig 1

In oreder D B I F A G C H

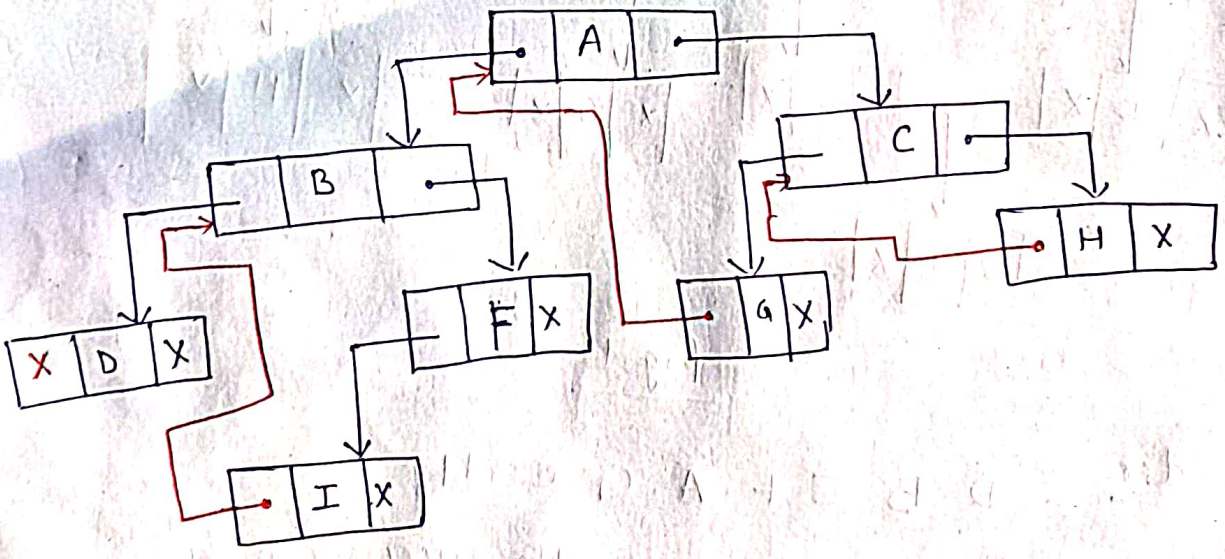
Replace right NULL pointer with successor node.



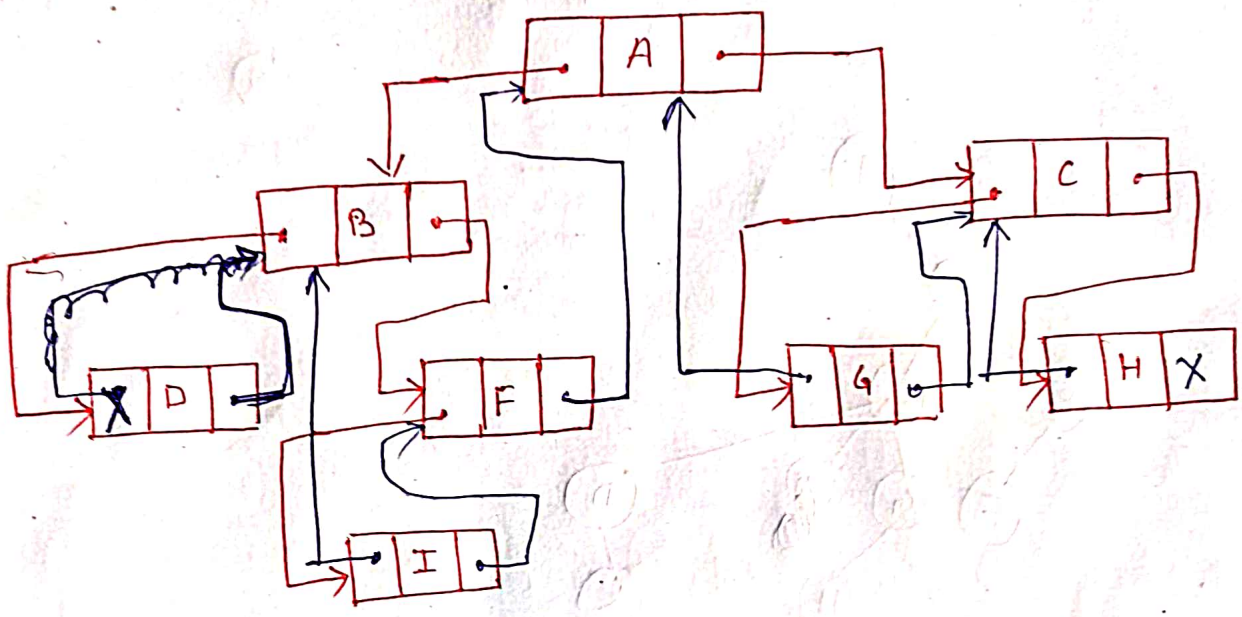
2) Left Threaded tree - The left NULL pointer of each node can be replaced by a thread to the predecessor of that node under Inorder traversal called a left thread and ~~tree~~ tree is called a left thread tree.

Ex→ Consider fig 1.

Inorder traversal : D B I F A G C H



(3) Fully Threaded Tree:- Both left and right NULL pointers can be used to point to predecessor and successor of that node, respectively under Inorder traversal. such a tree called a fully threaded tree.



Construct General tree to binary tree:-

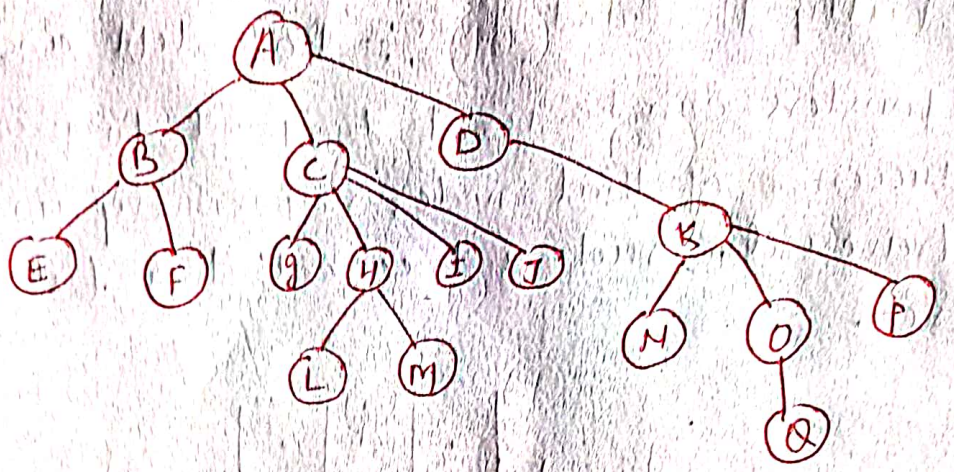
The method used for converting a general tree to a binary tree as follows:

1) Make the root of the binary tree, the same as the root of general tree.

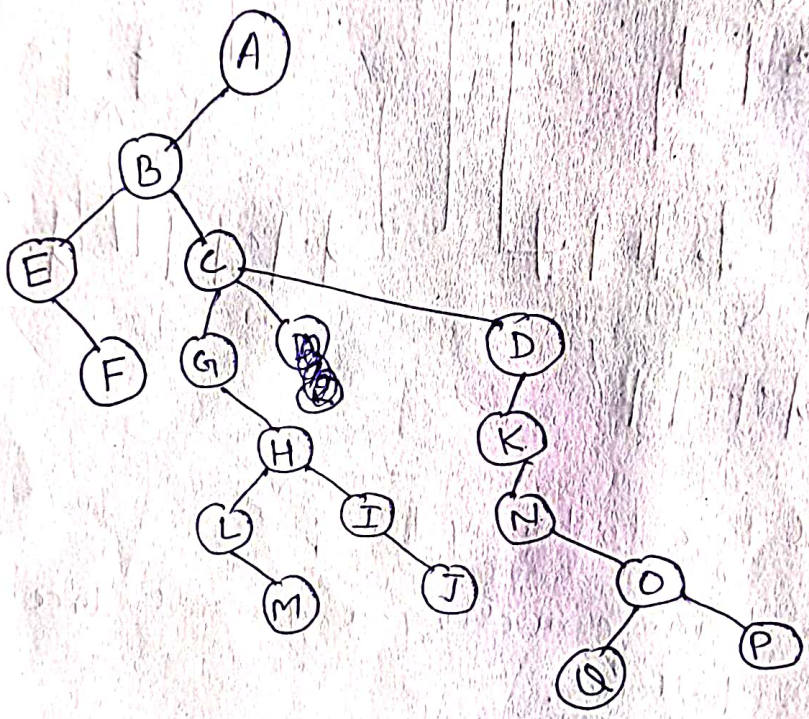
2) If $X1, X2, \ldots, X_N$ are child of X in general tree then make X1 is left child of binary tree and X_2 is the right child of X1, X3 is right child of X_2, X_N is the right child of X_{N-1} in binary tree.

P.T.O.

Example-



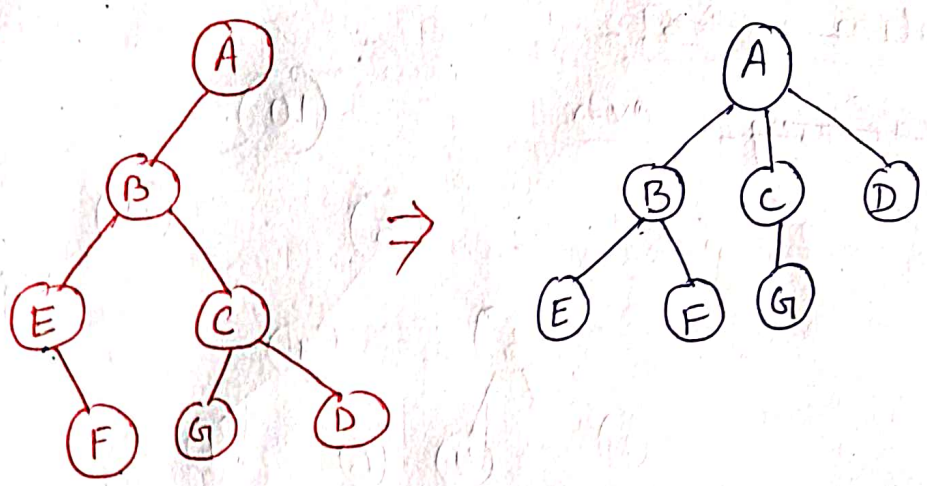
Sol'r



Conversion of binary tree to general tree :-

1) Make the root of the general tree, the same root of binary tree.
2) If X_1 is the left child of node X in a binary tree and X_2 be the right child of X1 ... and X_N is the right child of X_{N-1} then make $X_1, X_2, X_3, \ldots, X_N$. The child of X in equivalent general tree.

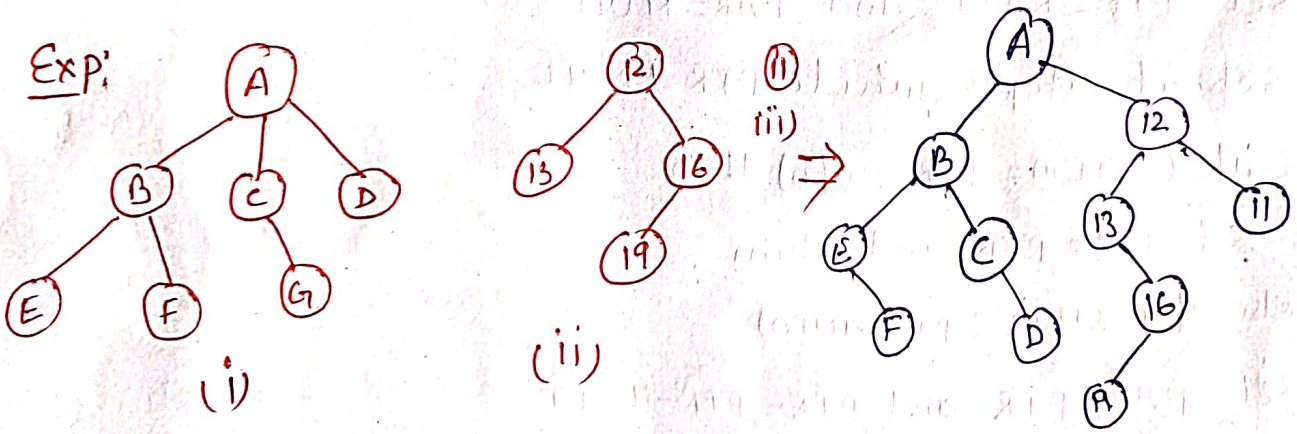
Example-



Forest to binary tree conversion:

Forest is a set of several trees that are not linked to each other. Initially left most tree is represented as binary B1, after that construct second tree to binary tree B2 and B2 is the right child of B1 & so on...

Exp:



Binary Search Tree- A binary search tree is a binary tree which satisfies the following rules:

1) The value of left child or left subtree of node is less than the value of the node.
2) The value of a right child or right subtree of a node is greater than or equal to the value of the node.
3) All the children either left or right should follow the above two rules.

Operation in BST -

1) Searching - (y→)

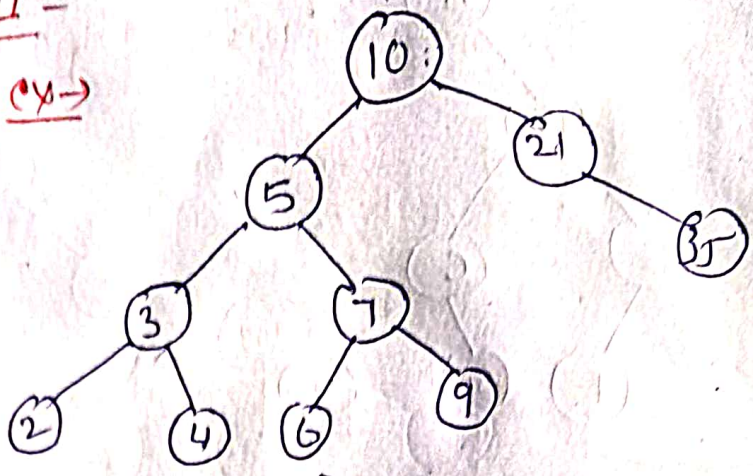


Fig. 2

Operation in binary search tree -

1) Searching → Algo:

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

1. Set PTR = ROOT and PAR = NULL
2. repeat step3 while PTR ≠ NULL
3. if (ITEM = PTR → info) then
   Set LOC = PTR and return
   else if (ITEM < PTR → INFO)
   Set PAR = PTR and PTR = PTR → LEFT
   else
   PAR = PTR and PTR = PTR → RIGHT
4. set LOC = NULL
5. Exit.

Ex. - In fig 2. Search the following point elements:

a) 7

b) 45

(i) Initial PTR = 10 (add) → address of 10 PAR = NULL item = 7

7 < 10 (item < ptr → info)

then PAR = PTR = 10 (add) PTR = PTR → left = 5 (add)

now 7 > 5 (item > PTR → info)

PAR = PTR = 5 (add) PTR = PTR → RIGHT = 7 (add)

7 = 7 (item = PTR)

LOC = PTR = 7 (address of 7)

item found

(ii) item = 45 PTR = ROOT = 10 (add) PAR = NULL

45 > 10 ⇒ T ⇒ PAR = PTR = 10 (add) PTR = PTR → right = 21 (add)

now 45 > 21 (ITE > PTR → info)

PAR = PTR = 21 (add) PTR = PTR → RIGHT ⇒ 35 (add)

45 > 35 ⇒ T ⇒ PAR = 35 (add) PTR = PTR → RIGHT = NULL

so LOC = NULL

item not found.

2) Insert → Algorithm

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

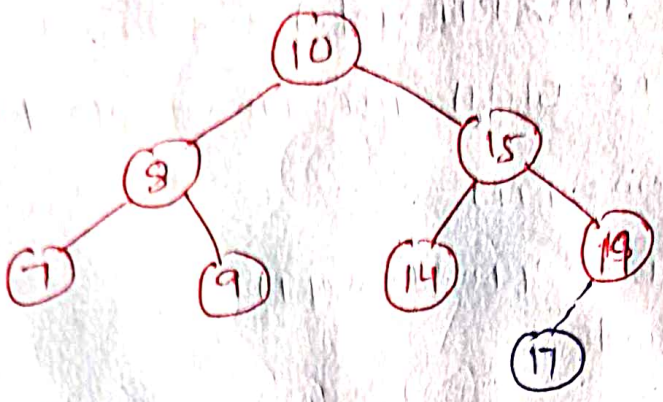
1. call search (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. if (LOC ≠ NULL) then exit
3. set new → info = item
   new → right = NULL
   new → left = NULL
4. if (PAR = NULL) then
   Set ROOT = new
   else if (item < PAR → INFO)
   PAR → Left = new

else

PAR→right = new

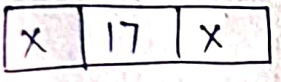
5. Exit.

Example→



Insert 17 in a BST.

Sol:- first call search function and find loc. In this 17 is not present so LOC = NULL & PAR = 19 (add)



new

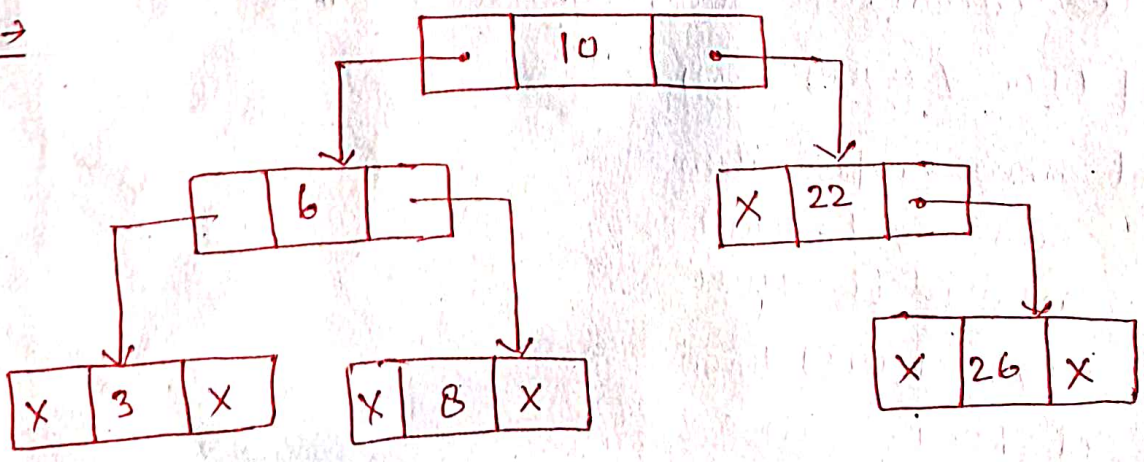
17 < 19 ⇒ T ⇒ PAR→left = new ⇒ 19→left = 17

3) Deletion→

Case 1: Node N has no children. (N is deleted node)

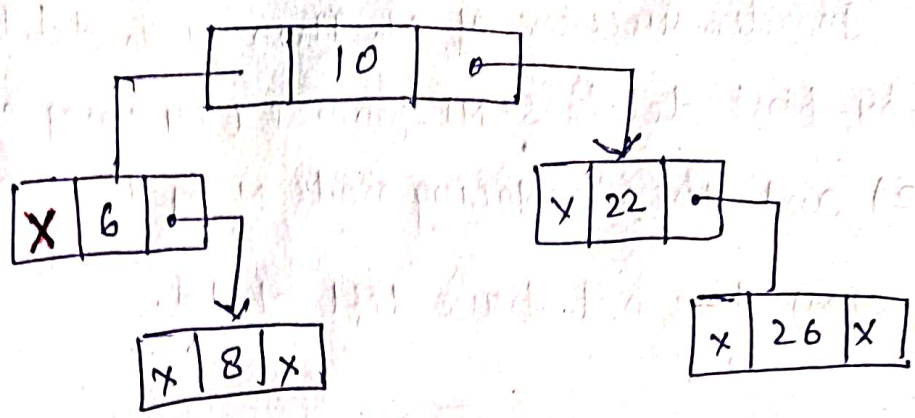
N is deleted from tree by replacing the location of N in Parent node P(N) by the NULL pointer.

Ex→



Delete 3 from Tree A.

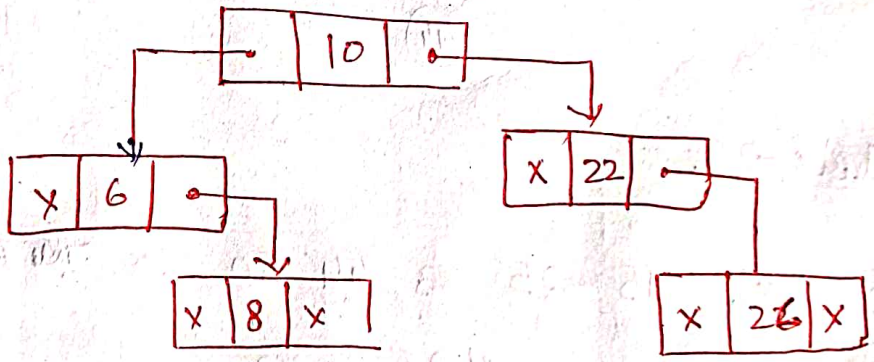
Sol: child of 3 = NULL



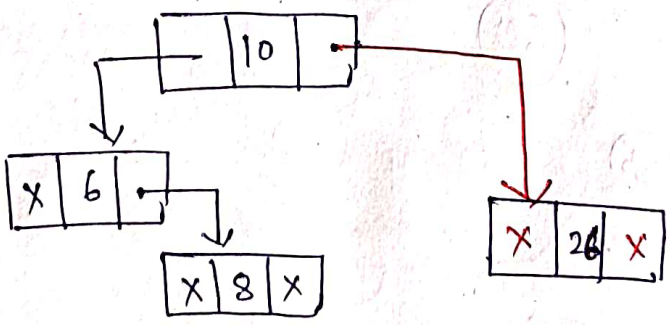
Case 2: N has exactly one child.

N is deleted by replacing the location of N in P(N) by the location of only child of N.

Exp:- Delete 22 from Tree



Sol:



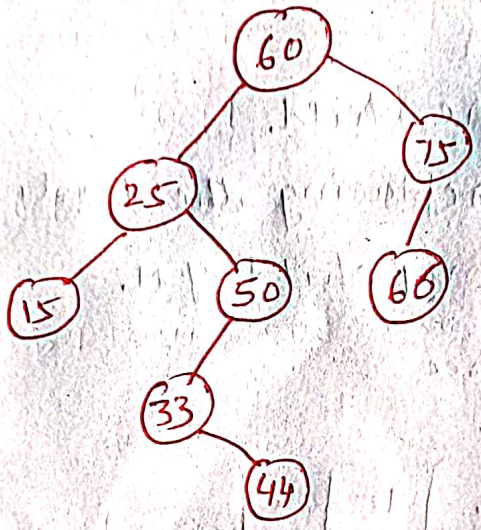
Child of 22(N) is 26. So PAR → right = child(N)

i.e., 10 → right = 26

Case 3; N has two children. Let S(N) denote the inorder successor of N. Then N is deleted from tree by first delete S(N) from T (by using case 1 or case 2) and then replacing node N by the node S(N).

Note:- S(N) does not have left child.

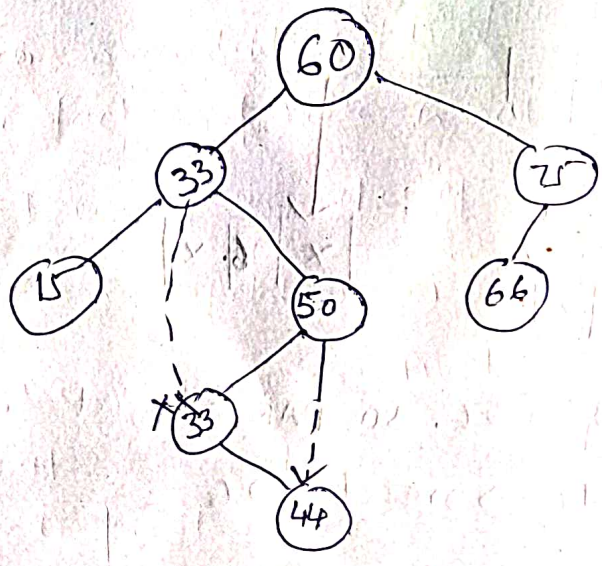
Ex→



Delete 25.

Sol:- Inorder : 15 25 33 44 50 60 66 75

Inorder succesor of 25 is 33.



Algorithm

Delete BST (INFO, LEFT, RIGHT, ROOT, ITEM)

1. call search(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. if LOC = NULL then write ITEM not in tree and Exit
3. if (loc→right ≠ NULL and loc→left ≠ NULL) then

   call CaseB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

   else

   call CaseA (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
4. Exit.

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

1. if (LOC→left = NULL and LOC→right = NULL) then

   Child = NULL

   else if (LOC→left ≠ NULL)

   Child = LOC→Left

   else

   child = LOC→right

2. if (PAR ≠ NULL)

   {

   if(LOC = PAR→left) then

   Set PAR→left = child

   else

   PAR→right = child

   }

   else

   ROOT = Child

3. return

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

1. Find successor and Parent of successor
   a) SUC = LOC → right and PARSUC = LOC
   b) Repeat while (suc → left ≠ NULL)
      Set PARSUC = SUC and SUC = SUC → left
2. LOC → info = SUC → info
3. if (PARSUC → left = SUC)
   PARSUC → left = SUC → right
   else
   PARSUC → RIGHT = SUC → right
4. return

AVL Tree-

- AVL is height balanced tree
- AVL tree should be binary search tree.
- Balance factor of each node of AVL tree should be -1, 0, 1.

Balance Factor (bf) = max. left subtree height − max. right subtree height

If balance factor of node is not 0, -1 and 1 then tree is unbalanced and use four types of rotations to balance tree.

1) LL rotation 2) RR Rotation 3) RL rotation 4) LR rotation

Rotations are depends on Pivot node P.

pivot node (P): It is nearest subbalanced node to the new inserted node.

Node C: It is child of P at the side of inserted node.

1) Left-Left Rotations (LL) :- If new node is inserted at left of P and left of C. we use only right rotation on P.

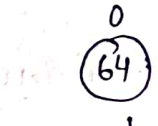
2) Right-Right Rotation :- If new node is inserted at right of P and right of C. we use only left rotation on P.

3) Left right rotation :- if new node is inserted at left of P and right of C. We use two rotations:
(i) first we use right-right rotation on C.
ii) the we use left-left rotation on P.

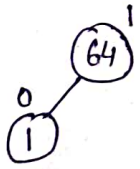
4) Right left rotation - If new node is inserted at right of P and left of C. we use two rotations.
(i) First we use left-left (LL) rotation on C.
ii) then we use right-right rotation on P.

Example :- 64, 1, 14, 26, 13, 110, 98, 85. create AVL tree.

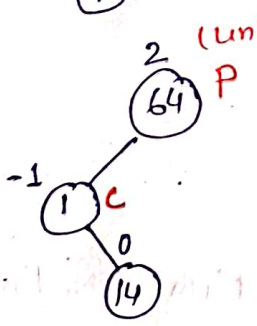
64



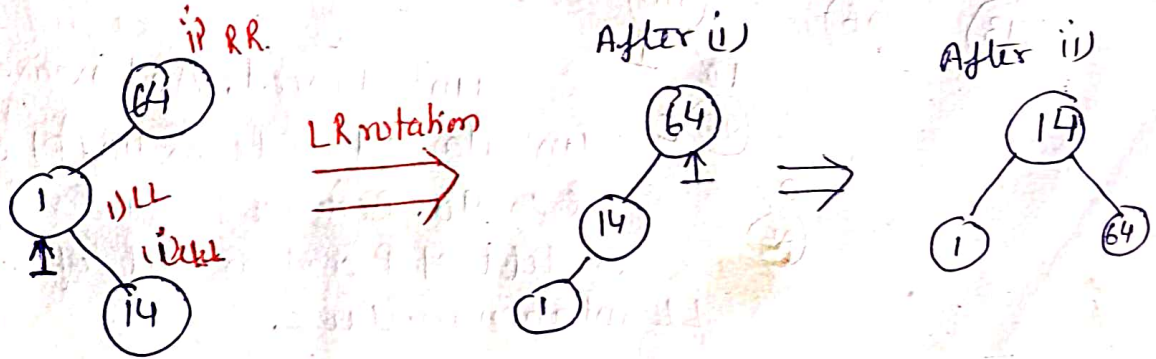
1



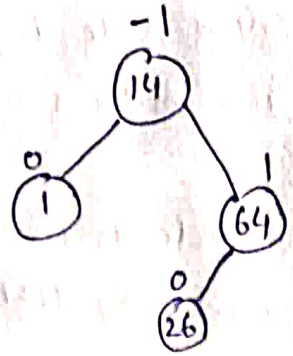
14



Inserted node is left on P and right on C. so we use left-right (LR) rotation.

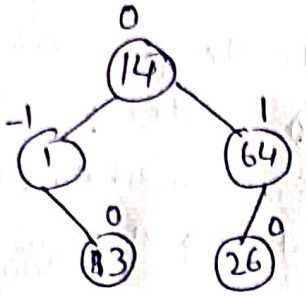


26



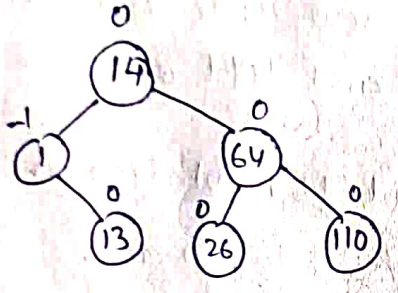
balanced

13



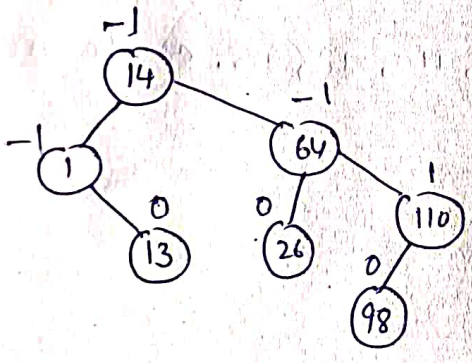
balanced

110



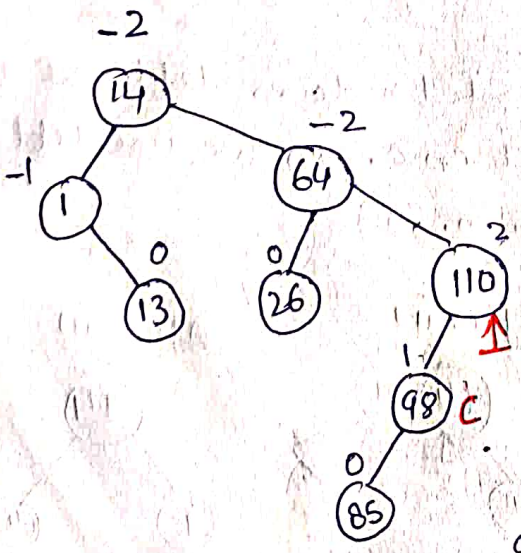
balanced

98



balanced

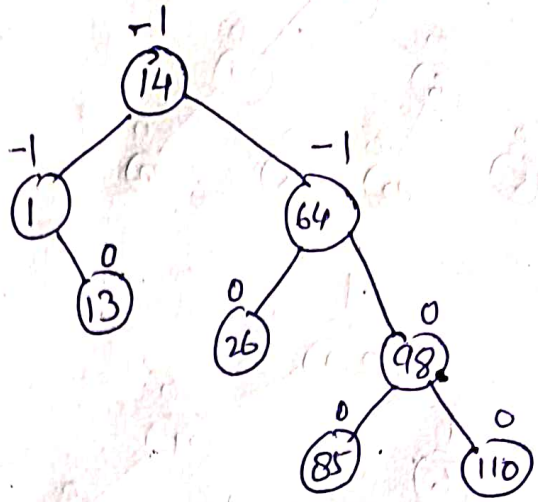
85



P(nearest unbalanced node)

unbalanced. And nearest unbalanced node is 110 (P) and C is 98. Because inserted node is on left of P and on left of C. So LL rotation will use.

So afte LL rotation



balanced

Example 2:-

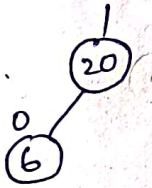
20, 6, 29, 5, 12, 25, 32, 10, 15, 27, 11. create AVL tree.

Sol:-

20

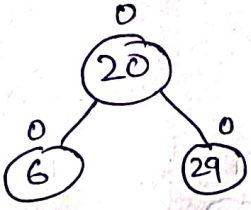
balanced

6



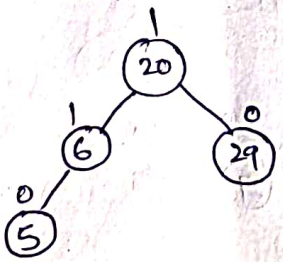
balanced

29



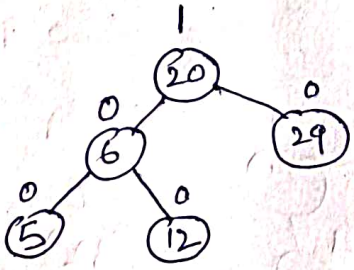
balanced

5



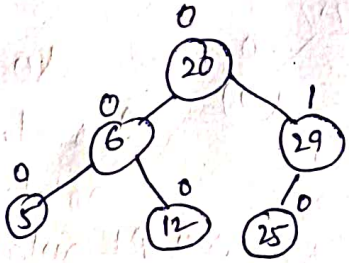
balanced

12



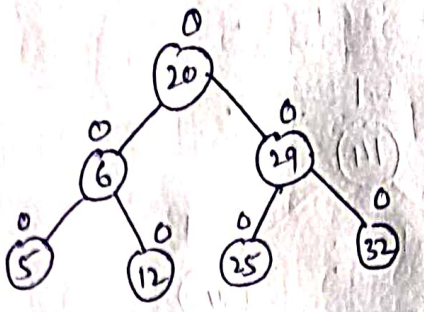
balanced

25



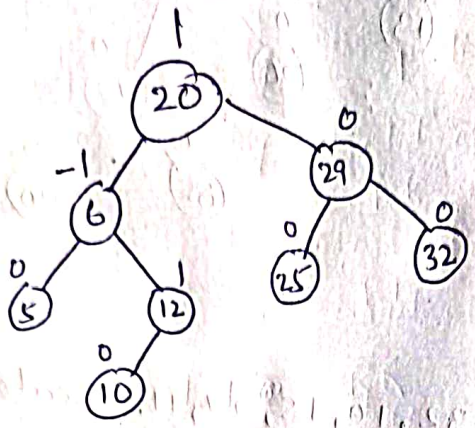
balanced.

32



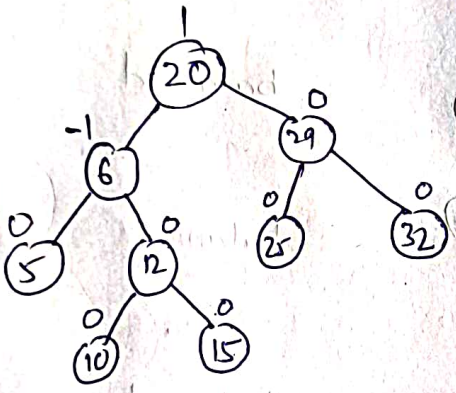
balanced

10



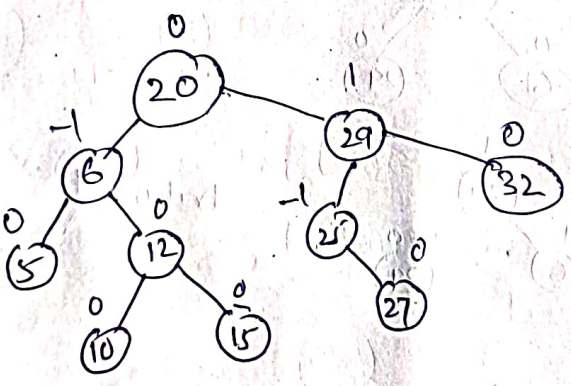
balanced.

15



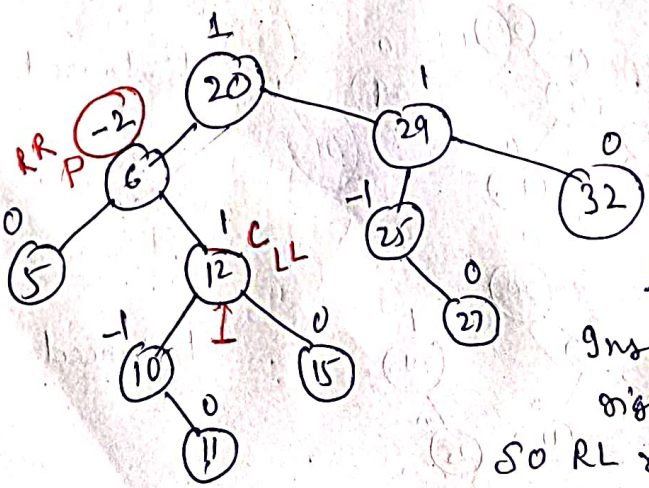
balanced

27



Balanced

11



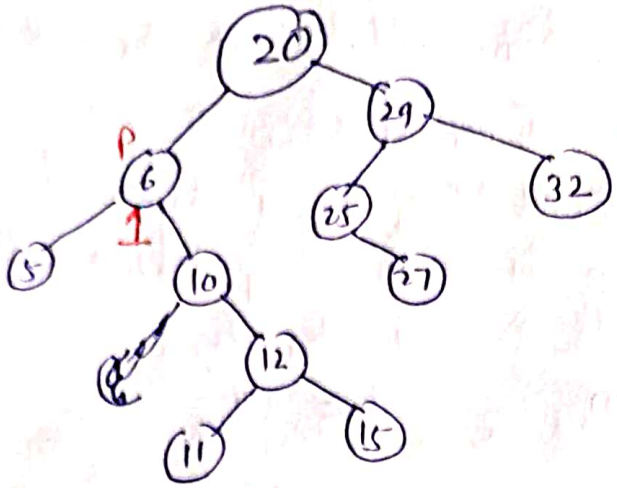
unbalanced at weight of 6 is -2.

So P=6 and C=12

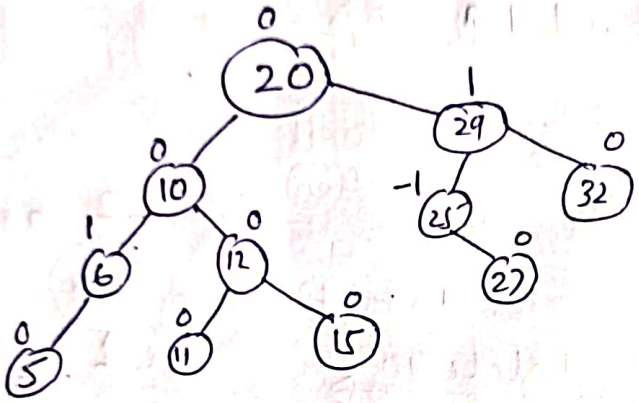
Inserted element 11 is right on P and left of C.

So RL rotation is used.

(i) LL rotation on C (12) gives



ii) RR rotation on P(6)



balanced

Deletion in AVL Tree:-

1) Initially, the AVL tree is searched to find the node to be deleted.
2. To delete in AVL tree is same as in Binary ~~tree~~ search tree.
3. After Deletion of node, check the balance factor of each node.
4. Rebalance the AVL tree if tree is unbalanced. for this we use R_0, R_1, R_{-1}, L_0, L_1, L_{-1}, rotations are used.

Suppose X will be deleted. Let P be the closest ancestor node on the path from X to the root node with a balance factor of +2 or -2.

C is the child of P in opposite side of X.

If X is right side of P and BF of C=0, then R0

If X is right side of P and BF of C=1 then R1

If X is right side of P and BF of C=-1 then R-1

if X is left side of P and BF of C=0 then L0 used

if X is left side of P and BF of C=1 then L1 used

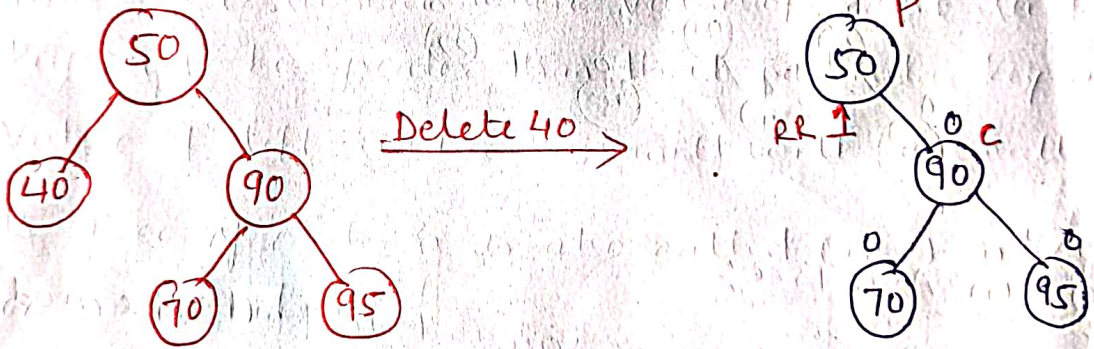
if X is left side of P and BF of C=-1 then L-1 used

R0 = L-L rotation L0 = RR

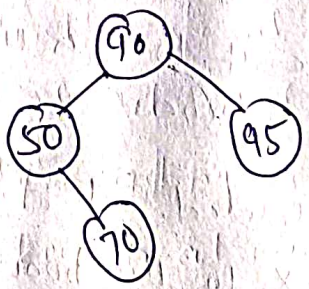
R1 = LL L1 = RL

R-1 = LR L-1 = RR

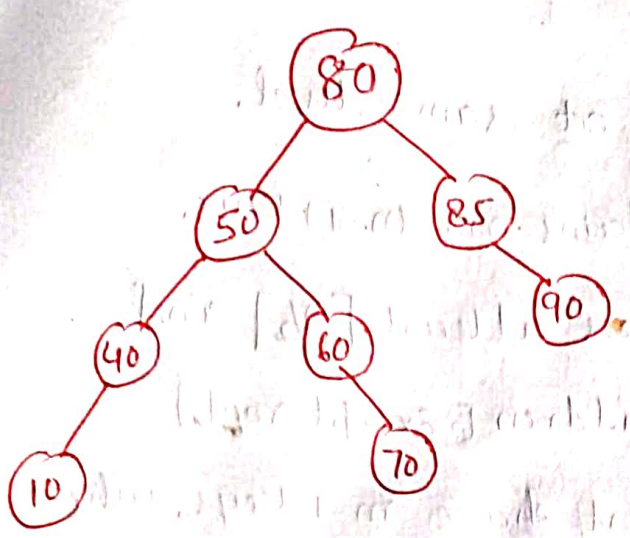
Example 1:-



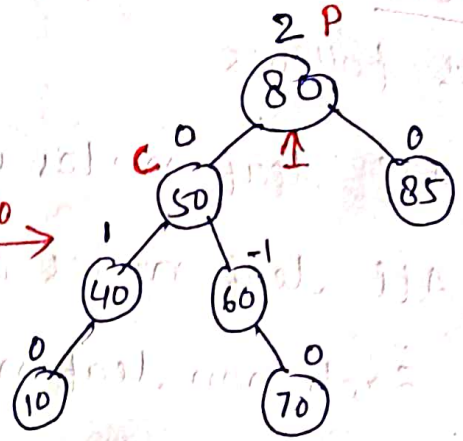
delete node X (40) is left of P(50). and balance factor of C=0, so L0 is used. L0(RR).



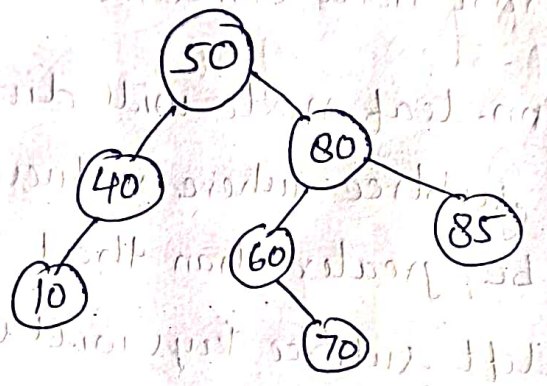
Example 2 :-



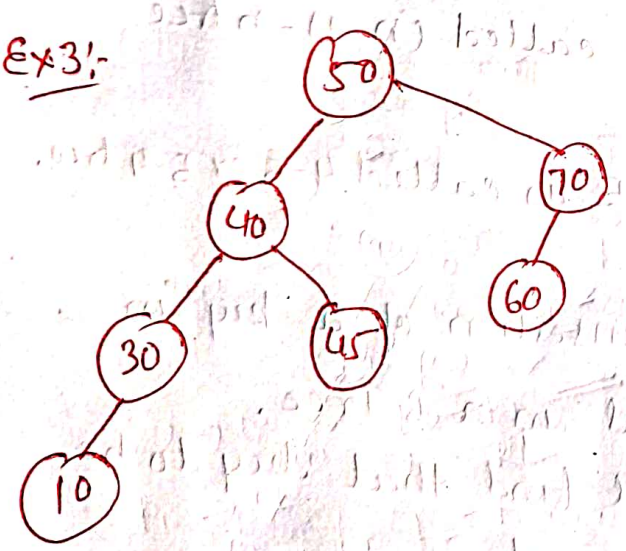
delete 90



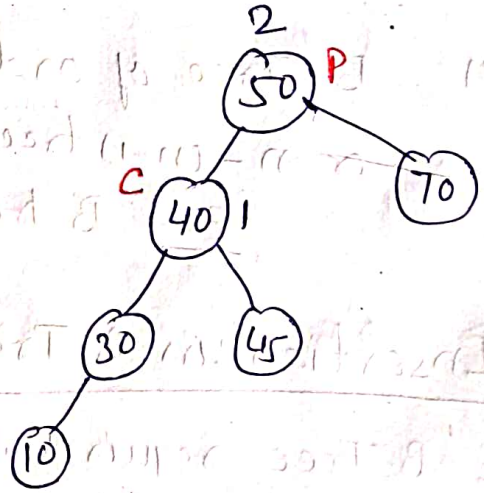
so P(80) and C(50). X(90) is right on P(80) and BF of C(50) is 0. so RO is used. RO means LL rotation.



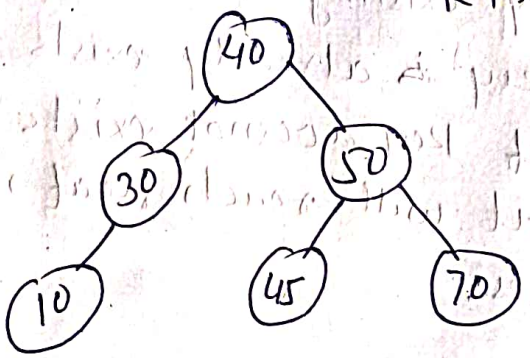
Ex3 :-



Delete 60



R1 is used



B-Tree- B tree of order-n can be defined as follows

1) All leaf nodes will be at same level.
2) All leaf nodes can contains max (n-1) keys.
3) Each non leaf nodes has atleast $\lceil n/2 \rceil$ and max. n node empty children [Except root]
4) All non leaf node must have m-1 keys, where m is the number of children for that node.
5) Each non-root node contains at least $\lceil (n-1)/2 \rceil$ keys.
6) Keys in non-leaf node will divide the left and right subtree where values of right subtree keys will be greater than that particular key and value of left subtree keys will will be smaller.
7) B-Tree of order n is also called (n-1)-n tree or n-(n-1) tree.

Ex→ B tree of order 5 is called 4-5 or 5-4 tree.

Insertion in B-Tree→ The insertion of a key in a B-Tree requires first traversal in a B-Tree. Through traversal it will find that key to be inserted is already existing or not.

Suppose key is already exists than no insertion required. if key doesnot exists in tree then through traversal it will reach leaf node. Now we have two cases:

(i) Node is not full

(ii) Node is full.

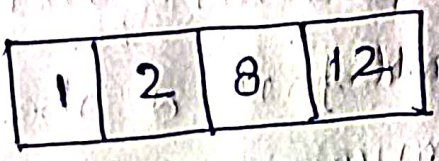
- if the leaf node in which key is to be Inserted is not full, then insertion is done in the node.
- if the node is full, then insert the key in the order into the existing set of keys in the node.
- Then split the node at its median into two nodes at the same level
- then push the median element up by one level.
- Accomadate the median element in the parent node if it is not full, otherwise repeat the same procedure
- This may even call for rearrangement of the keys in the root node or the formation of a new node itself.

Q:→ Create B tree of order 5. The given keys are

1, 12, 8, 2, 25, 6, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 53, 55, 45.

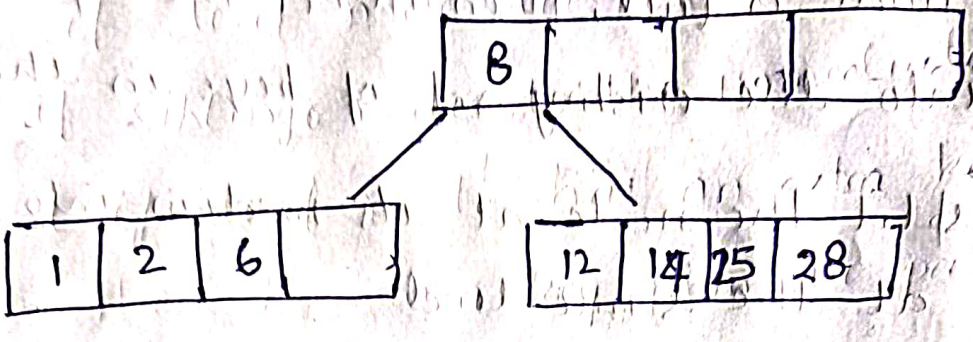
P.T.O.

Sol:



25 split → 1, 2, 8, 12, 25

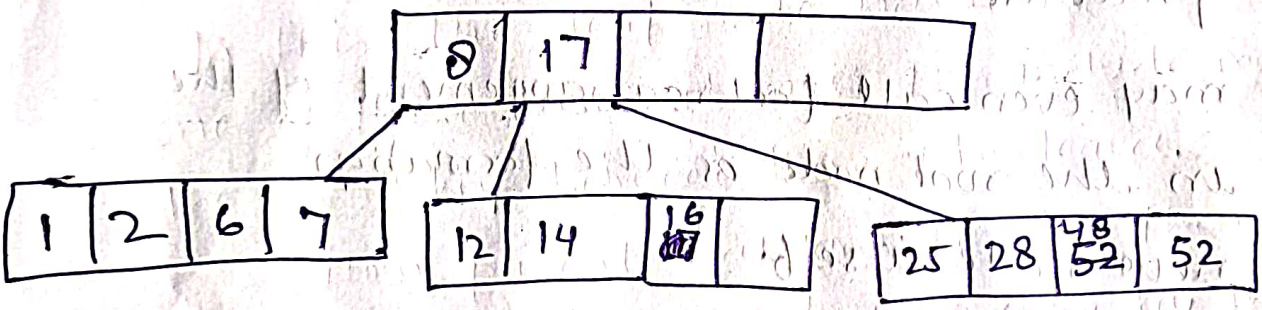
middle



6, 14, 28

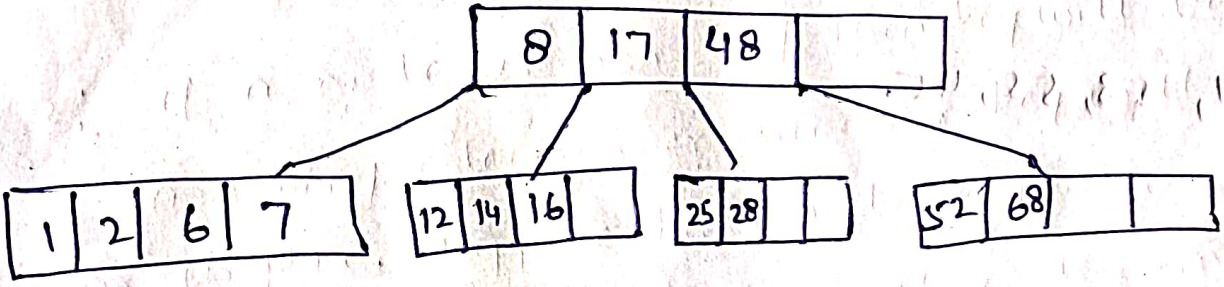
17 → 12, 14, 17, 25, 28 split

mid

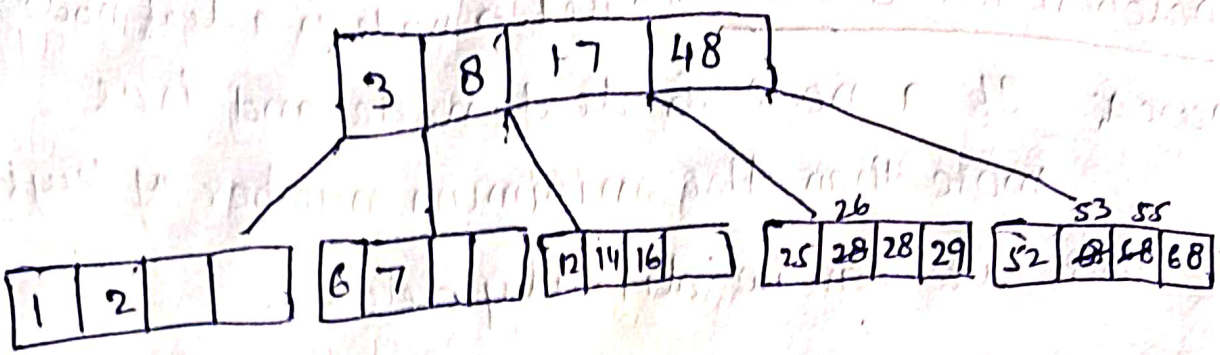


17, 7, 52, 16, 48

68 → 25, 28, 48, 52, 68 → split



3 → 1, 2, 3, 6, 7 split

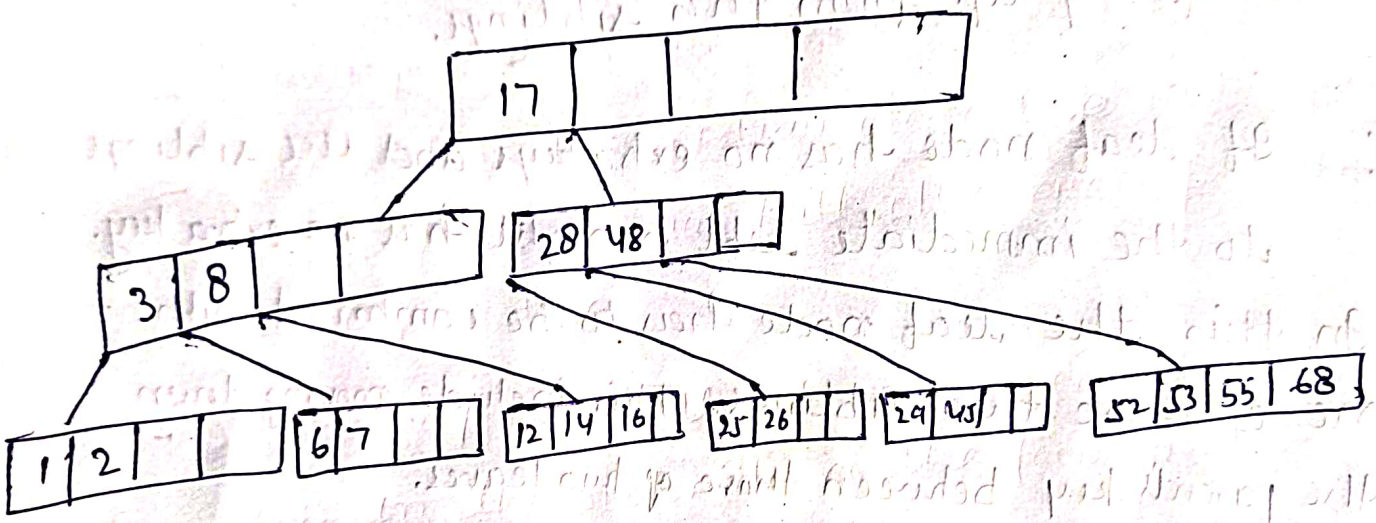


26, 29, 53, 55

45 → 25, 26, 28, 29, 45 → split

when 28 moves upward it becomes 3, 8, 17, 28, 48

so again split.



Practice → 1)

Deletion in B-Tree → case 1. → node is a leaf node

case a: If a node is leaf node and has more than the minimum number of keys then it can be directly deleted.

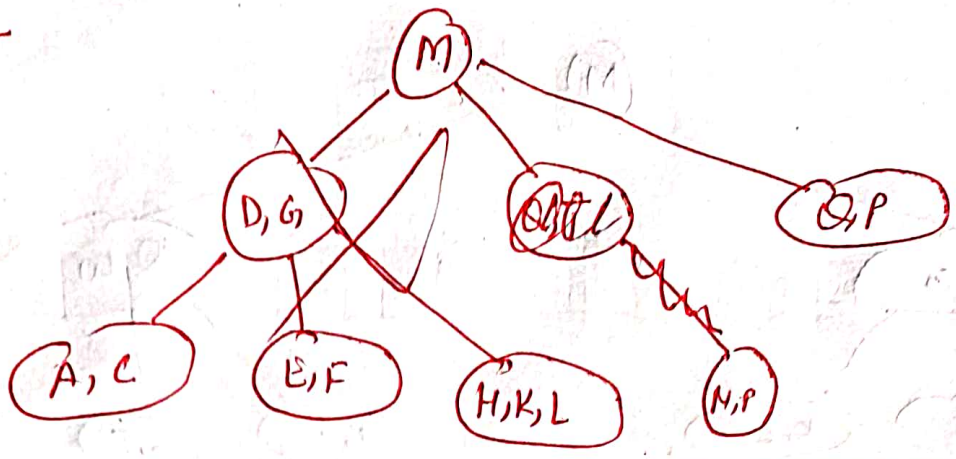
case b: if leaf node doesnot have extra key. If siblings nodes to the immediate left or right of leaf node has an extra key, we can then borrow a key from the parent and move a key up from this siblings.

case c: If leaf node has no extra keys and the sibling to the immediate left or right has no extra key. In this the leaf node has to be combined with one of these two siblings. This include moving down the parent's key between those of two leaves.

Now parent has one less key then it borrows from siblings, if sibling has no extra key then merge with sibling & move down a key from

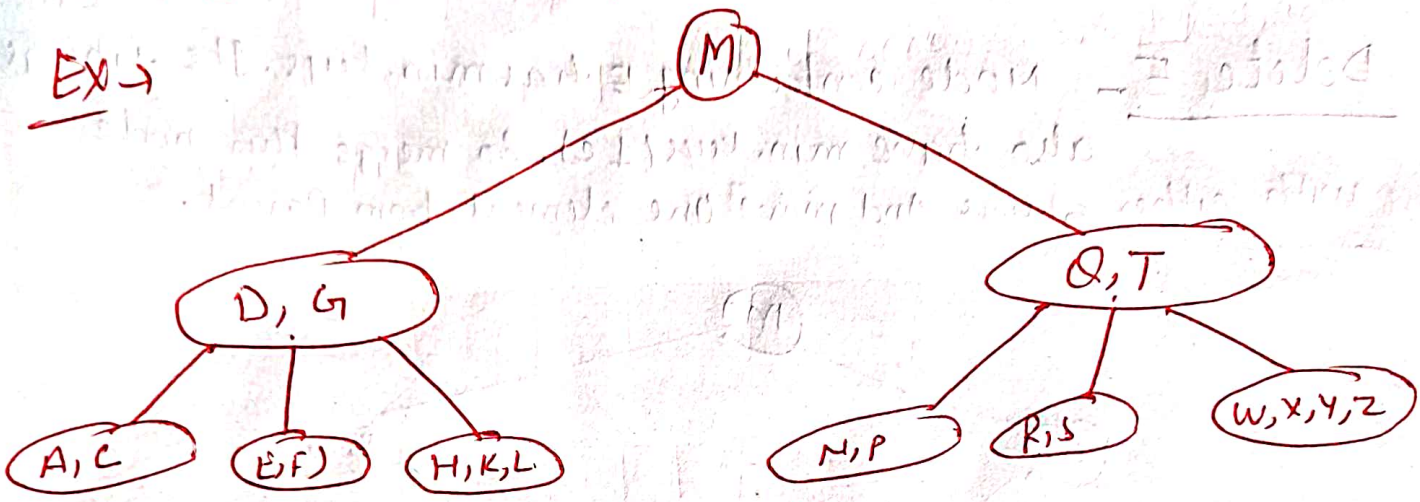
case 2: if a node is non-leaf then key will be deleted and its predecessor or successor key will come on its place. suppose both nodes of predecessor and successor have minimum no. of keys, then nodes of predecessor & successor keys will be combined.

Exp -



Consider B-Tree of order 5.

Ex →

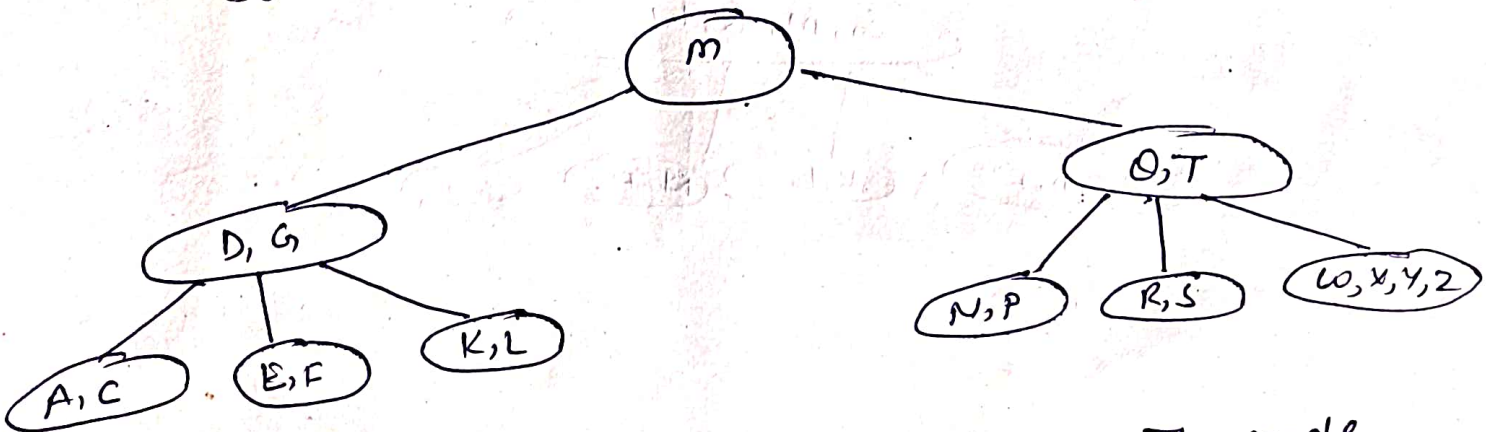


Delete H, R, E

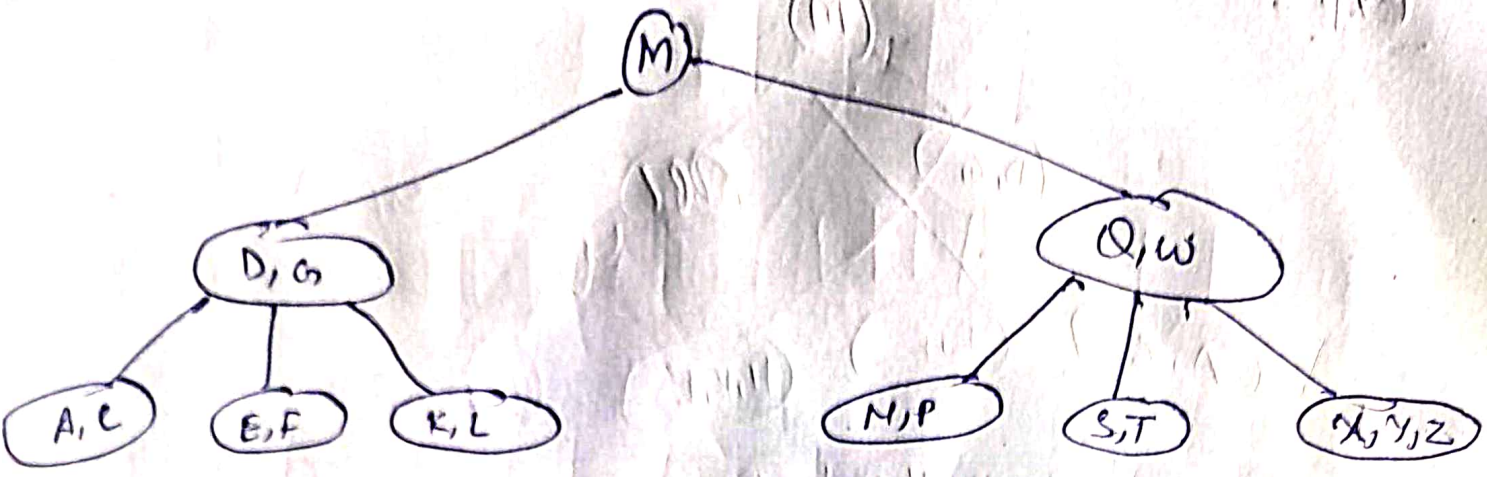
Sol:- Delete H. Every (except root) Node contains min $\left(\frac{n-1}{2}\right)$ keys $= \left(\frac{5-1}{2}\right) = 2$ keys.

Here if we delete H. Then node contains 2 ~~nodes~~ keys (1 B).

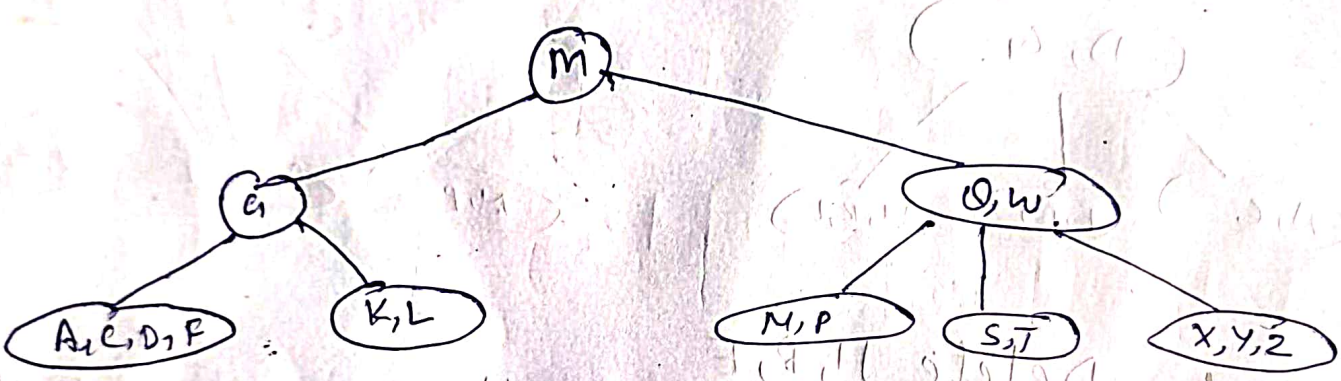
So tree becomes



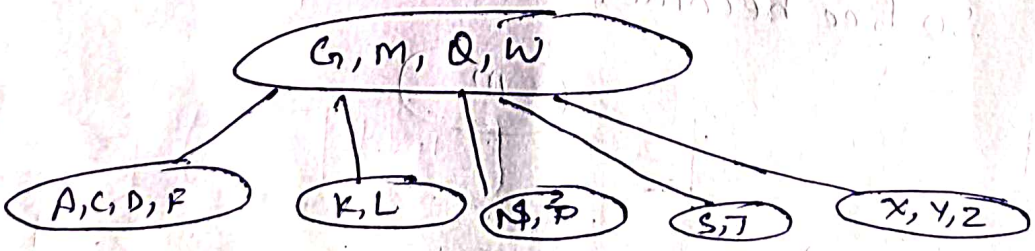
Delete R. ~~If we delete R, then that node~~ The node containing R contains min ~~nodes~~. keys. so check its immediate left and right node. Right will contain 4 keys (1 B). So borrow a key (T) from parent and move key from right sibling (W) to parent.



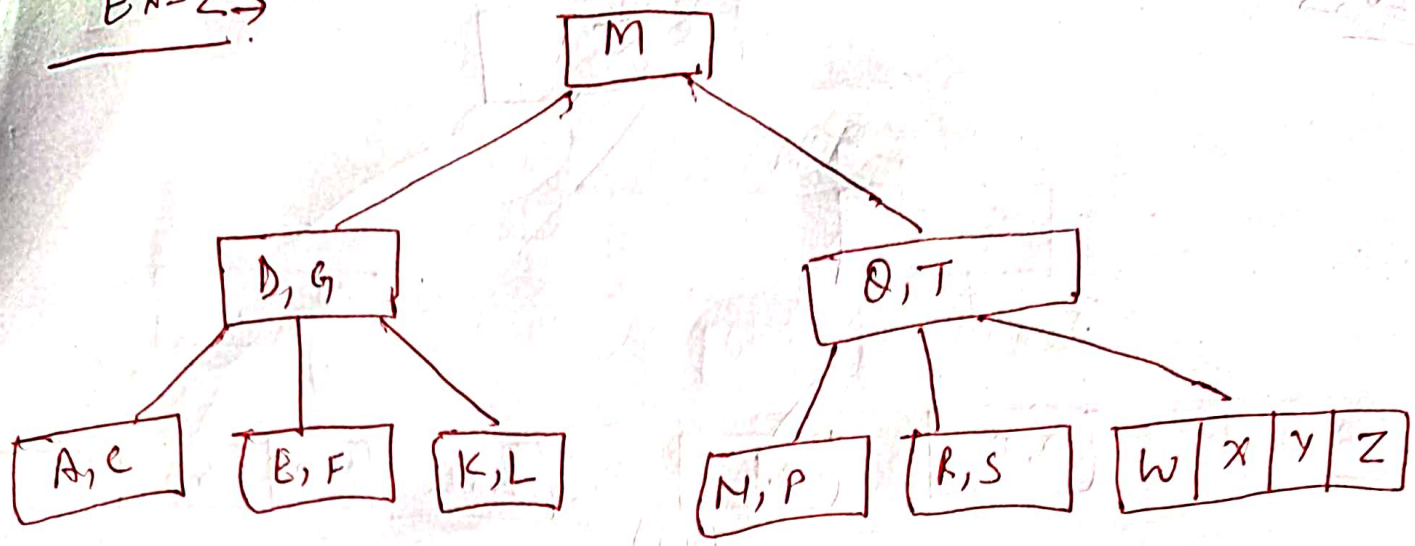
Delete E- Node containing E has min. keys, its siblings also have min. keys (1 C). So merge this node with either siblings And move one element from Parent.



Now node containg G has this node. node merge with node containg Q and W and node M is moved

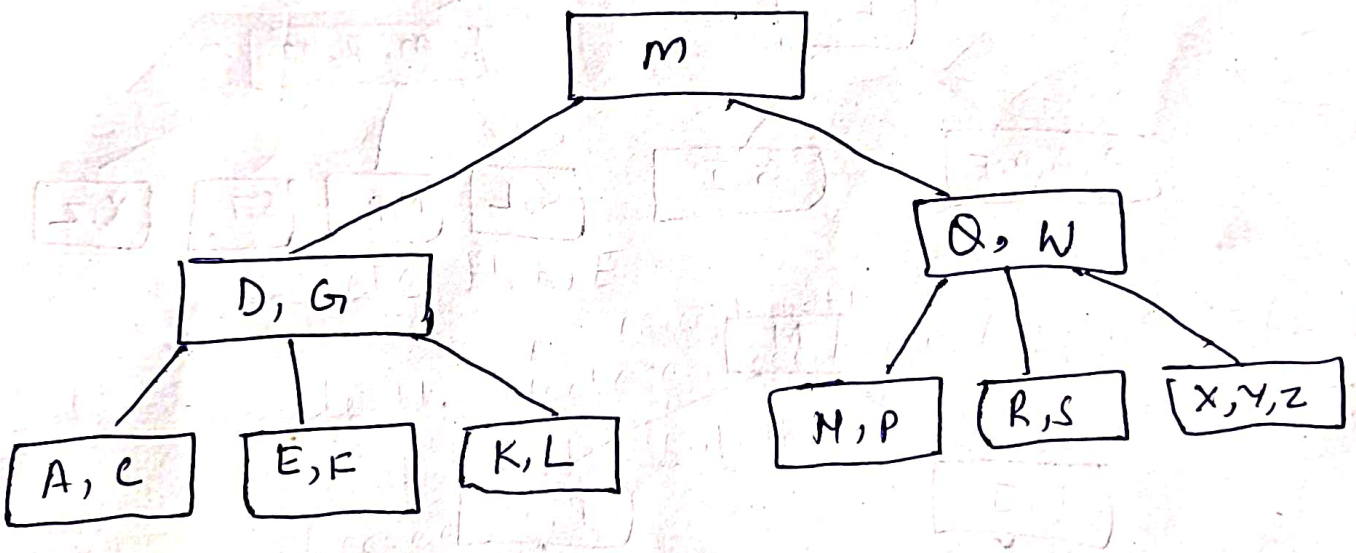


Ex-2→

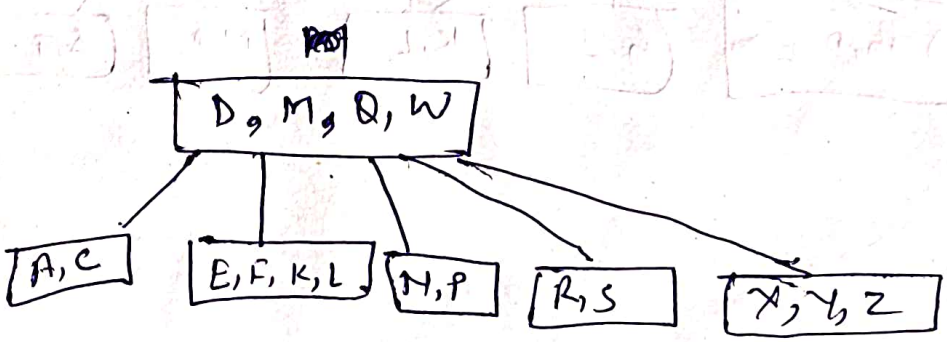


Delete T, G.

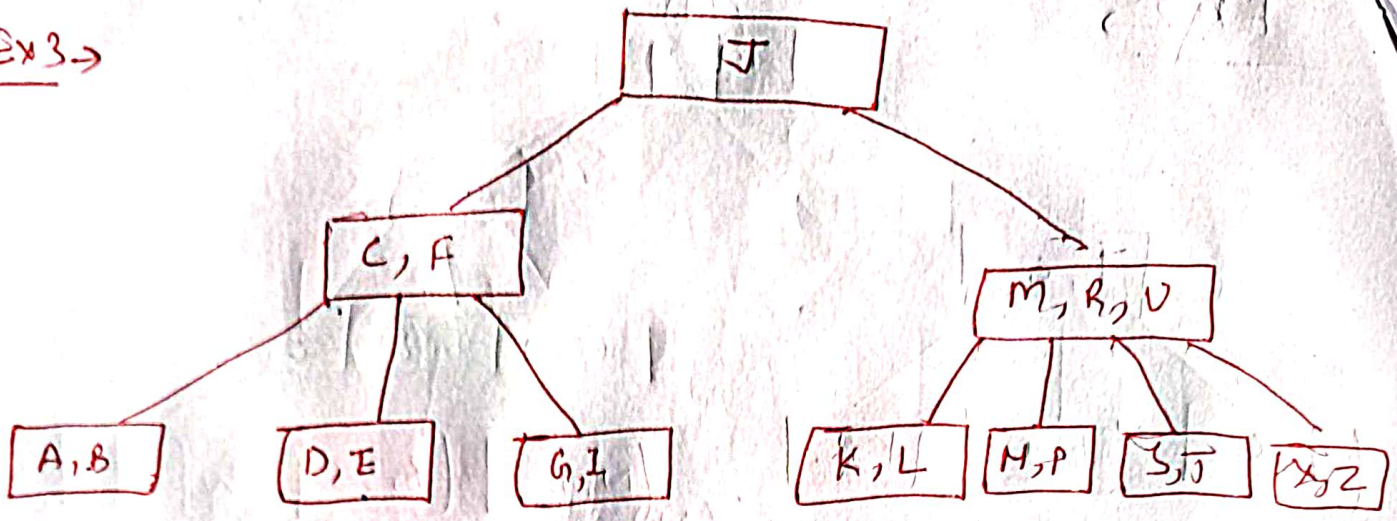
Sol- Delete T



Delete G.

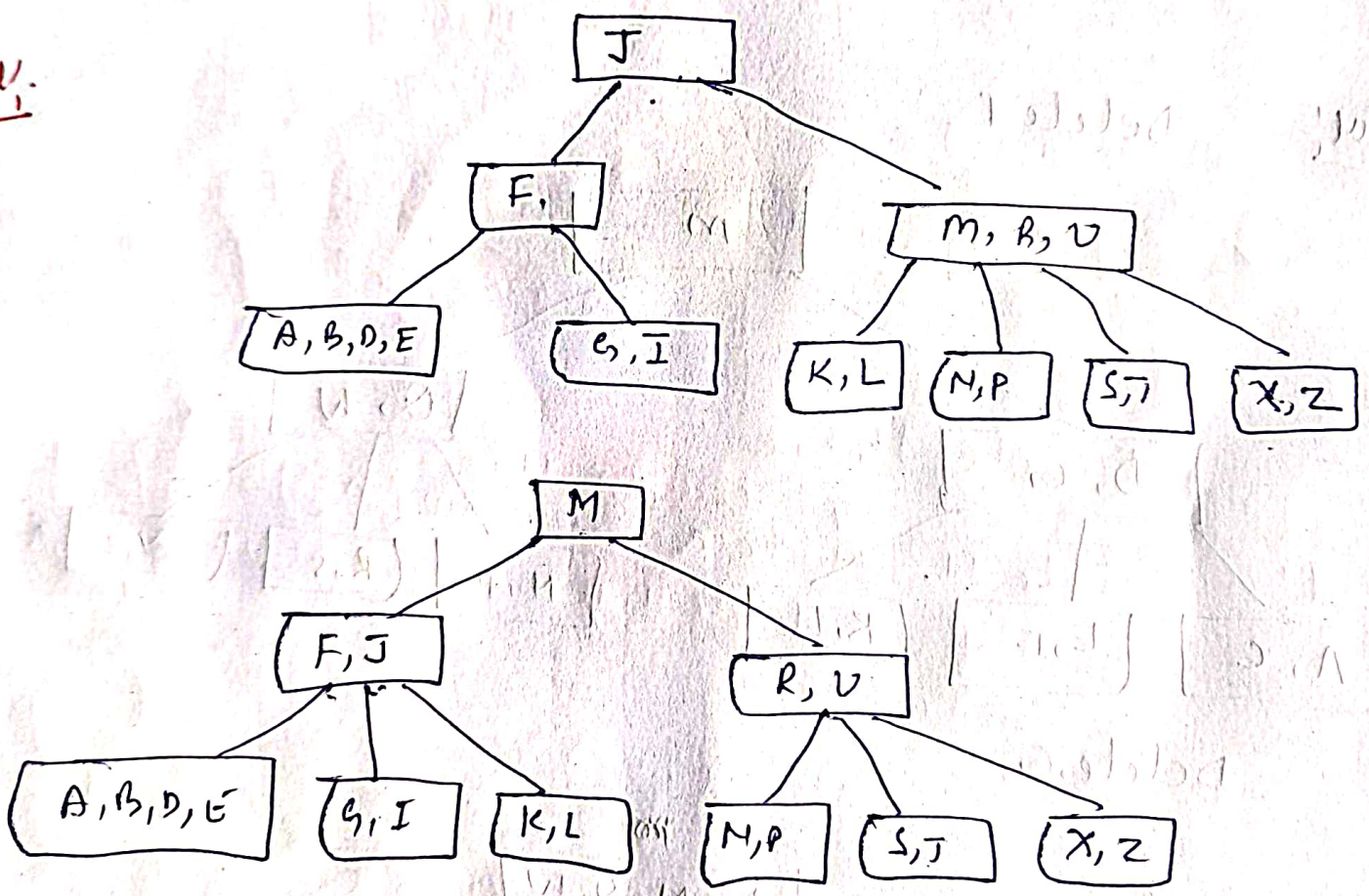


Ex 3 →

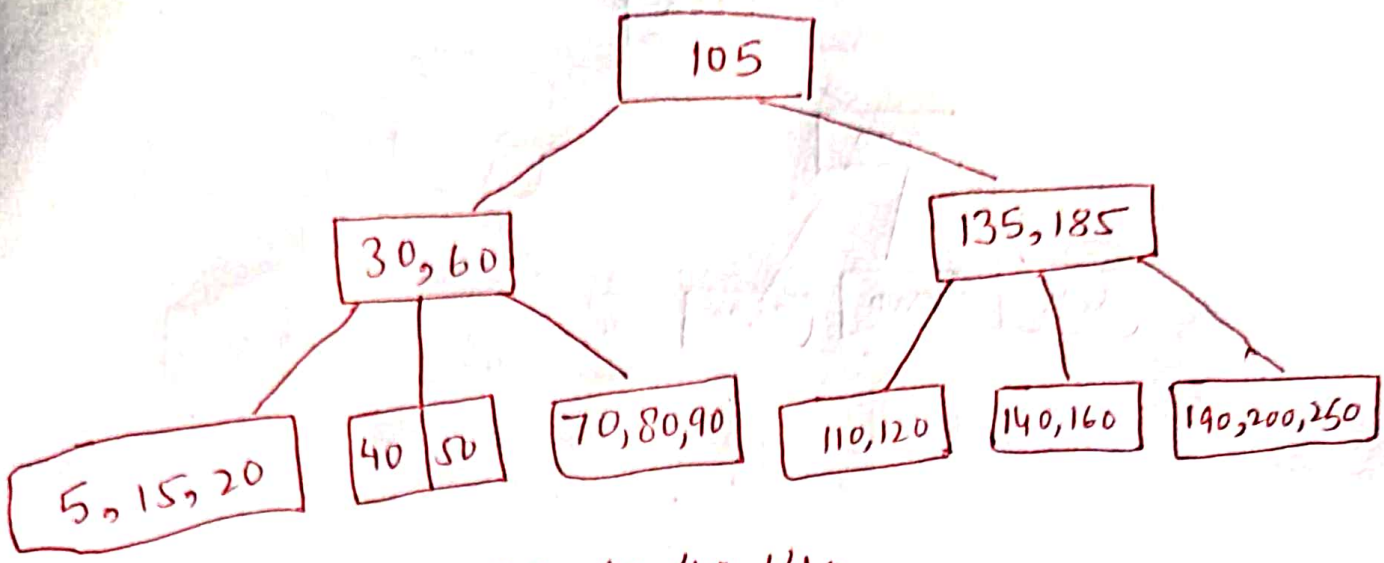


Delete C

Sol:

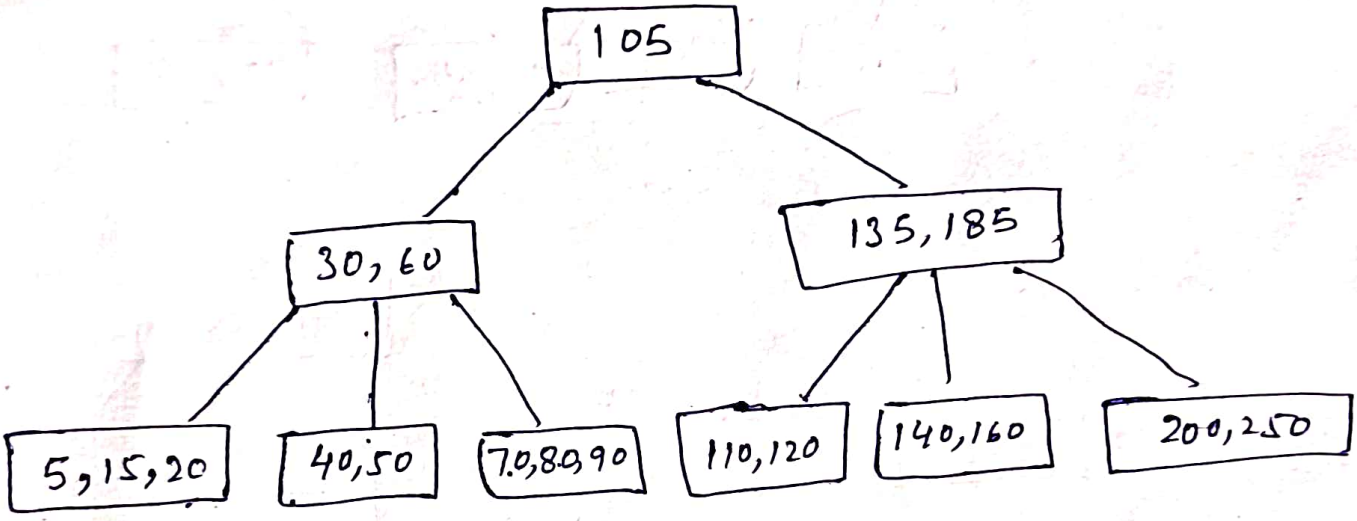


Ex4- B-Tree of order 5

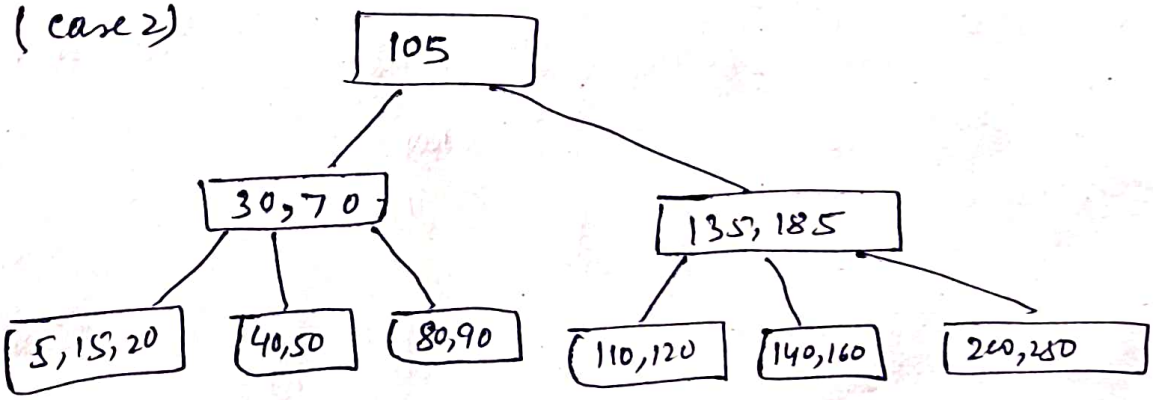


Delete 190, 60, 40, 140

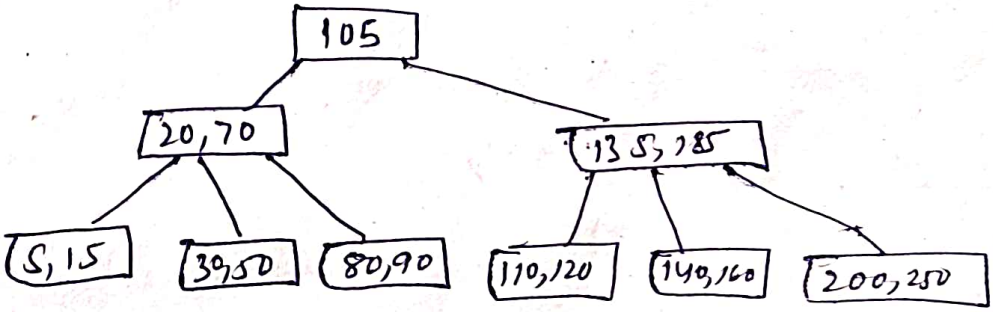
Sol)- Delete 190 (1A)



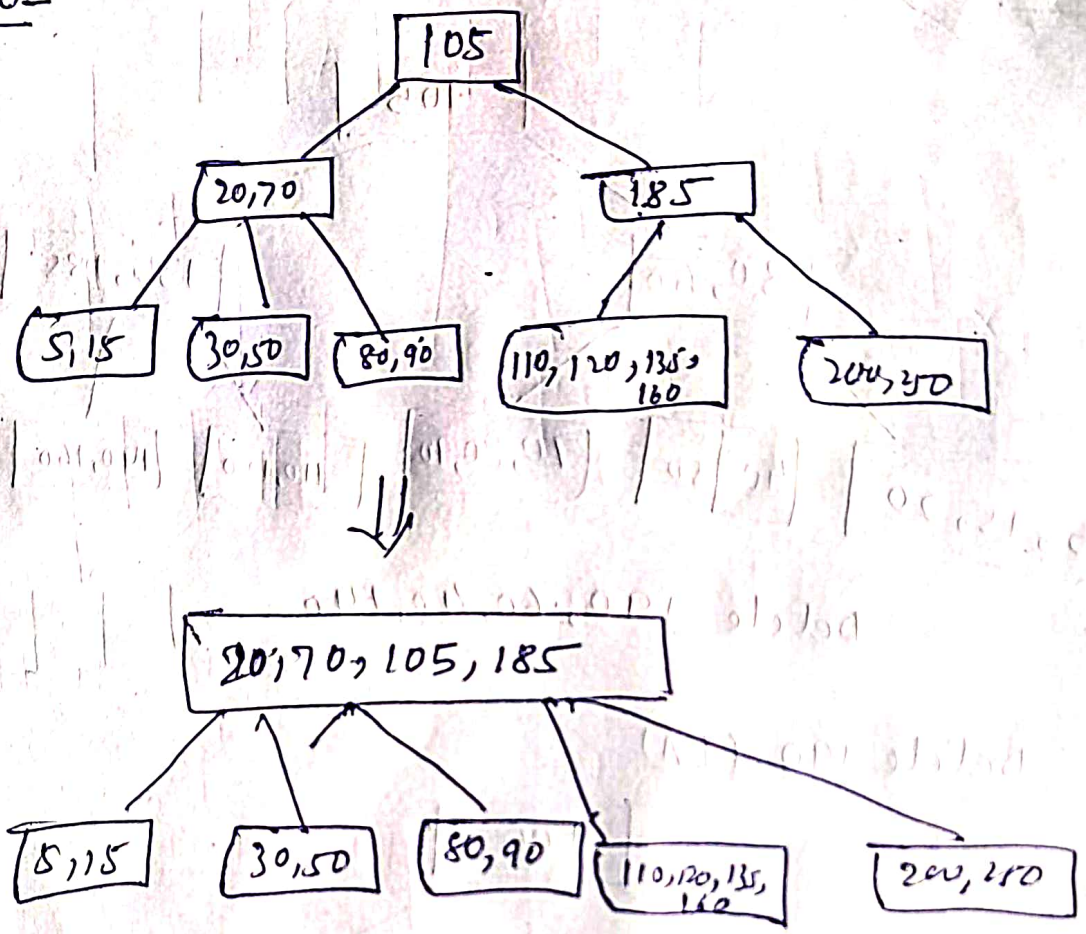
Delete 60 (case 2)



Delete 40 (1B)



Delete 140-


105
20,70
185
5,15
30,50
80,90
110,120,135,160
200,250
20,70,105,185
5,15
30,50
80,90
110,120,135,160
200,250

Unit-5 (Set-3)

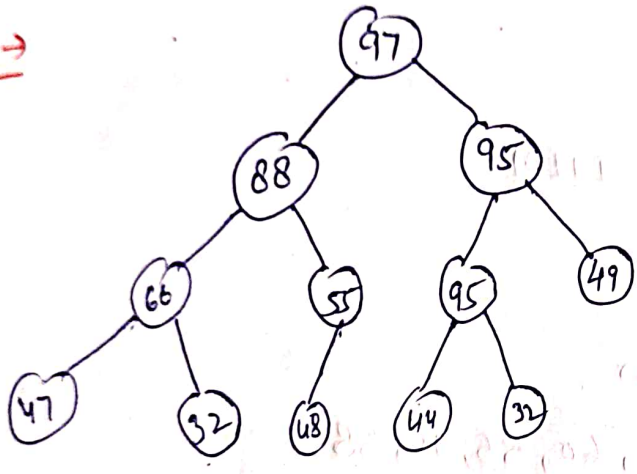
Heap Tree- Suppose H is a complete tree with n elements. Then H is called a heap. or max heap.

H is called a max-heap, if each node N of H has the following property:

The value of N is greater than or equal to the value of each of the children of N.

H is called min heap if value at N is less than or equal to the value of any of children of N.

Ex→



max. heap

Inserting into a heap → Suppose H is a heap with N elements. and an ITEM of information is given. We insert ITEM into the heap as follows:

1) First adjoint item at the end of H so that H is still a complete tree, but not necessarily a heap.
2) Then let ITEM rise to its appropriate place in H so that H is finally a heap.

Algorithm →

Insert_Heap(TREE, N, ITEM, PTR, PAR)

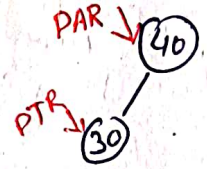
1. Set N = N+1 and PTR = N
2. repeat step 3 to 6 while (ptr > 1)
3. PAR = ⌊ PTR/2 ⌋
4. if (item ≤ TREE [PAR]) then
   set Tree[PTR] = ITEM and return.
5. TREE[PTR] = TREE[PAR]
6. PTR = PAR
7. TREE[1] = ITEM
8. Exit

Q:- Build a heap from
40, 30, 50, 22, 60, 55, 75, 58

Sol:- 40

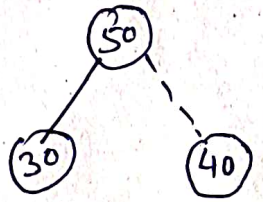
→ N = 1 PTR = 1 ⇒ PTR > 1 ⇒ False
so Tree[1] = 10

30



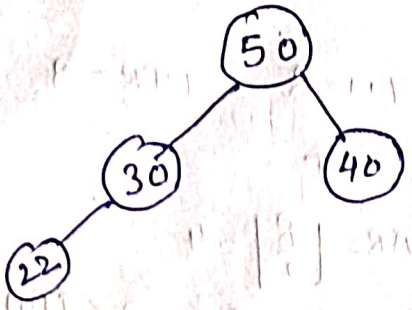
→ item = 30 N = 1+1 = 2 PTR = 2
PTR > 1 ⇒ 2 > 1 ⇒ True
PAR = ⌊ PTR/2 ⌋ = 2/2 = 1
30 ≤ T[1] ⇒ 30 < 40 - T
T[PTR] = T[2] = 30 and return

50



→ item = 50 N = 2+1 = 3 PTR = 3
PTR > 1 ⇒ 3 > 1 ⇒ T
PAR = ⌊ 3/2 ⌋ = 1 50 < T[1] ⇒ 50 < 30 ⇒ F
T[PTR] = T[PAR] ⇒ T[3] = T[1] = 40
PTR = 1 ⇒ 1 > 1 ⇒ F T[1] = item = 50

22



item = 22 N = 3+1 = 4

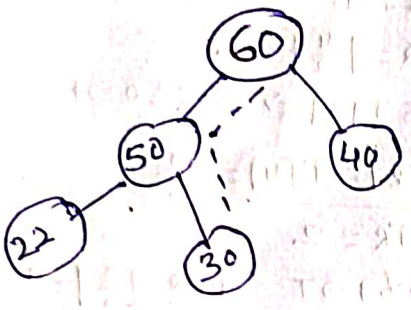
PTR = 4

$4 > 1 \Rightarrow PAR = \lfloor \frac{4}{2} \rfloor = 2$

$22 < T[2] = 22 < 30 \Rightarrow T$

$T[4] = item = T[4] = 22$

60



item = 60 N = 4+1 = 5 PTR = 5

$5 > 1 \Rightarrow T$ $PAR = \lfloor \frac{5}{2} \rfloor = 2$

$60 < T[2] = 60 < 30 \Rightarrow F$

$T[PTR] = T[PAR] \Rightarrow T[5] = T[2] = 30$

PTR = 2

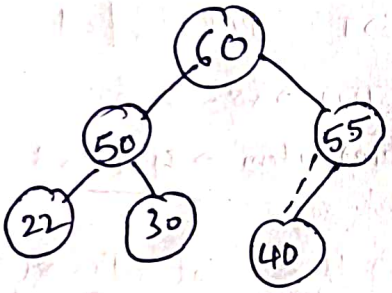
$2 > 1 \Rightarrow T$ $PAR = \lfloor \frac{2}{2} \rfloor = 1$

$60 < T[1] \Rightarrow 60 < 50 \Rightarrow F$

$T[2] = T[1] = 50$

$T[1] = 60$

55



item = 55 N = 6 PTR = N = 6

$PTR > 1 \Rightarrow 6 > 1 \Rightarrow T$

$PAR = \frac{6}{2} = 3$ $55 \leq T[3] = 55 \leq 40 \Rightarrow F$

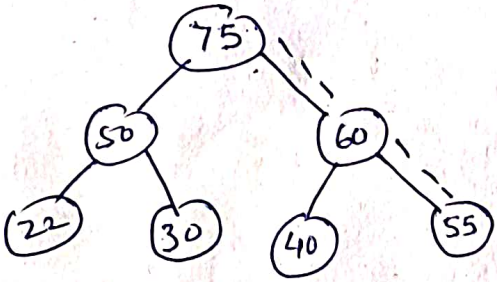
$T[6] = T[3] \Rightarrow T[6] = 40$

$PTR = PAR = 3$, $3 > 1 \Rightarrow T$

$PAR = \frac{PTR}{2} = \frac{3}{2} = 1$, $55 < T[1] \Rightarrow 55 < 60 \Rightarrow T$

$T[3] = 55$

75



item = 75 N = 7 PTR = 7

$7 > 1 \Rightarrow T$ $PAR = \lfloor \frac{7}{2} \rfloor = 3$

$75 \leq T[3] \Rightarrow 75 \leq 55 \Rightarrow F$

$T[7] = T[3] = 55$ PTR = 3

$3 > 1 \Rightarrow T$ $PAR = \lfloor \frac{3}{2} \rfloor = 1$

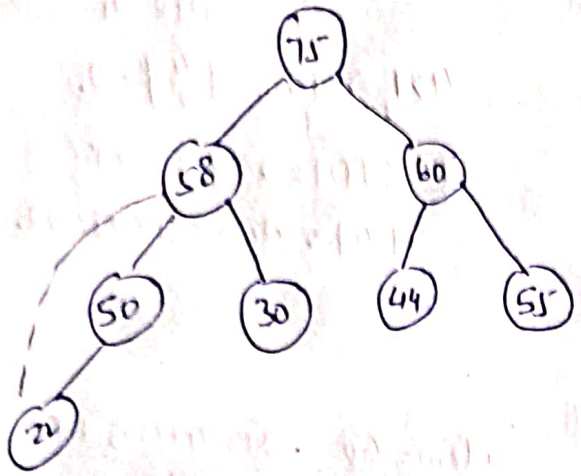
$75 \leq 60 \Rightarrow F$

$T[3] = T[1] = 60$ PTR = 1

$1 > 1 \Rightarrow F$

$T[1] = item = 75$

58



N=7+1=8 PTR=8

8>1 ⇒ T

PAR = $\lfloor \frac{8}{2} \rfloor$ = 4

item ≤ T[PAR] ⇒ 58 ≤ T[4] ⇒ 58 ≤ 22 ⇒ F

T[4] = T[8] = 58

PTR = 4 4>1 ⇒ T

PAR = $\lfloor \frac{4}{2} \rfloor$ = 2

item ≤ T[2] ⇒ 58 ≤ 50 ⇒ F

T[2] = T[4] = 58

PTR = 2

2>1 ⇒ T PAR = $\lfloor \frac{2}{2} \rfloor$ = 1

58 ≤ T[1] = 58 ≤ 75 ⇒ T

T[2] = item = 58

2>1 ⇒ T PAR = $\lfloor \frac{2}{2} \rfloor$ = 1

58 ≤ T[1] ⇒ 58 ≤ 75 ⇒ T

T[PTR] = item ⇒ T[2] = 58

Delete Heap– This procedures assign the root Tree[1] to the variable ITEM and then reheaps the remaining elements. The variable LAST save the value of the original last node of H. The pointers PTR, LEFT, RIGHT gives the location of last and its left and right children as LAST in the tree.

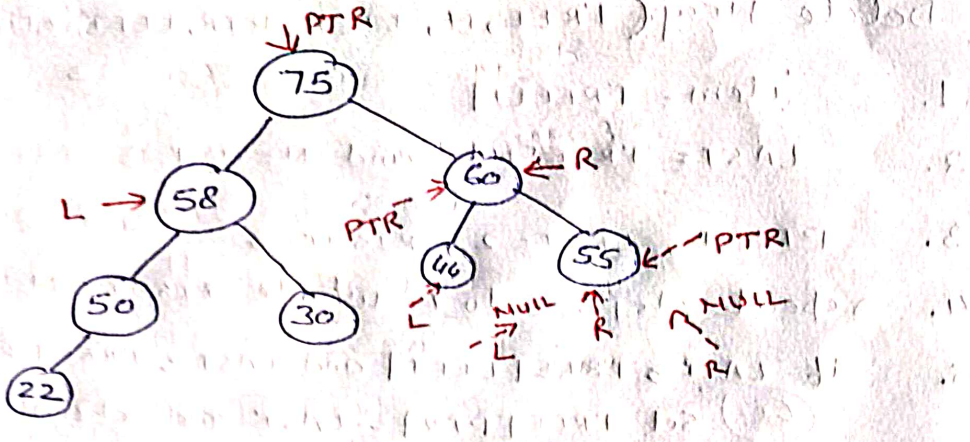
Delete_Heap(TREE, N, ITEM, PTR, LEFT, RIGHT)

1. item = TREE[1]
2. LAST = TREE[N] and N = N-1
3. PTR = 1, LEFT = 2, RIGHT = 3
4. repeat steps 5 to 7 while RIGHT ≤ N
5. if LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT] then.
   Set TREE[PTR] = LAST and return
6. if TREE[RIGHT] ≤ TREE[LEFT] then
   Set TREE[PTR] = TREE[LEFT], and PTR = LEFT
   else
   Set TREE[PTR] = TREE[RIGHT] and PTR = RIGHT
7. Set LEFT = 2*PTR and RIGHT = LEFT+1
8. if LEFT = N and if LAST < TREE[LEFT] then.
   Set TREE[PTR] = TREE[LEFT] & PTR = LEFT
9. Set TREE[PTR] = LAST
10. Return.

Heap Sort(A, N)

1. Repeat for i = 0 to N-1
   call insertHeap(A, i, item, PTR, PAR)
2. repeat while N >= 1
   a) call DeleteHeap(A, N, ITEM, PTR, LEFT, RIGHT)
   b) A[N+1] = ITEM
3. Exit.

Q: Heap Sort



Delete Root 75.

N = 8

Sol: item = Tree[1] = 75

LAST = TREE[8] = 22 — N = 8 − 1 = 7

PTR = 1 LEFT = 2 RIGHT = 3

RIGHT ≤ N ⇒ 3 ≤ 7 ⇒ T

LAST ≥ Tree[Left] and LAST ≥ TREE[RIGHT] ⇒ 22 ≥ 58 and 22 ≥ 60 ⇒ F

TREE[RIGH] ≤ TREE[LEFT] ⇒ 60 ≤ 58 ⇒ False

TREE[PTR] = TREE[RIGHT] ⇒ TREE[1] = 60 PTR = 3

LEFT = 6 RIGHT = 7

RIGHT ≤ N ⇒ 7 ≤ 7 ⇒ T

22 ≥ 44 and 22 >= 55 ⇒ F

55 ≤ 44 ⇒ F

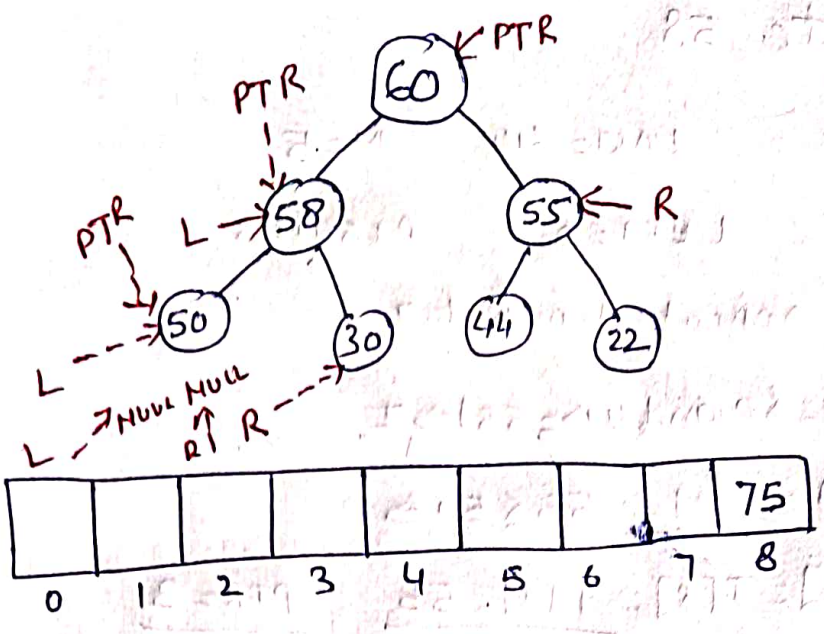
TREE[3] = Tree[7] = 55 PTR = 7

LEFT = 14 RIGHT = 15

15 <= 7 ⇒ F

LEFT = N ⇒ 14 == 7 ⇒ F

TREE[7] = 22



Now Delete 60

ITEM = 60 LAST = 22 N = 6

PTR = 1 LEFT = 2 RIGHT = 3

$3 \leq 6 \Rightarrow T$ repeat step 5 to 7

(22 >= 58 and 22 > 55) $\Rightarrow$ False

$T[3] \leq T[2] \Rightarrow 55 \leq 58$

$T[PTR] = T[LEFT] \Rightarrow T[1] = T[2] \Rightarrow T[1] = 58$ PTR = 2

LEFT = 4 RIGHT = 5

$5 <= 6 \Rightarrow T$ repeat step 5 to 7

(22 >= 50 and 22 >= 30) $\Rightarrow$ F

$T[5] <= T[4] \Rightarrow T$

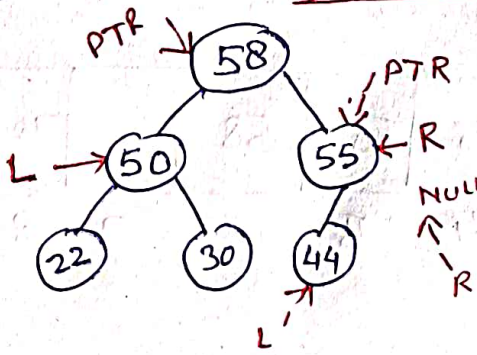
$T[2] = T[4] = 50$ PTR = 4

LEFT = 8 RIGHT = 9

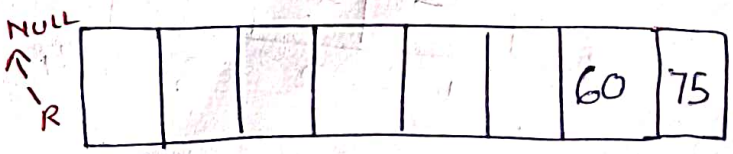
$9 \leq 6 \Rightarrow F$

LEFT == N $\Rightarrow 8 == 6 \Rightarrow F$

T[PTR] = LAST $\Rightarrow T[4] = 22$



Array becomes



Now delete 58

ITEM = 58 LAST = 44 N = 5

PTR = 1 LEFT = 2 RIGHT = 3

3 <= 5 repeat steps 5 to 7

(44 >= 50 and 44 >= 55) ⇒ F

T(3) ≤ T[2] ⇒ 55 ≤ 50 ⇒ F

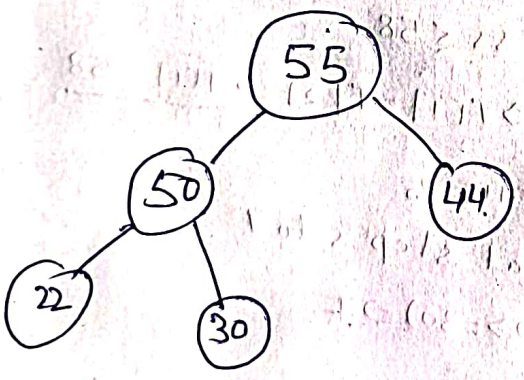
T[1] = T[R] ⇒ T[1] = 55 PTR = 3

LEFT = 6 RIGHT = 7

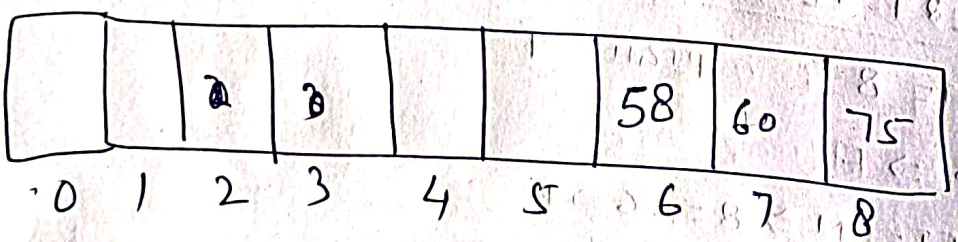
7 <= 5 ⇒ F

LEFT == N ⇒ 6 == 5 ⇒ F

T[3] = 44.



Array becomes



Similarly deletes All roots and array becomes