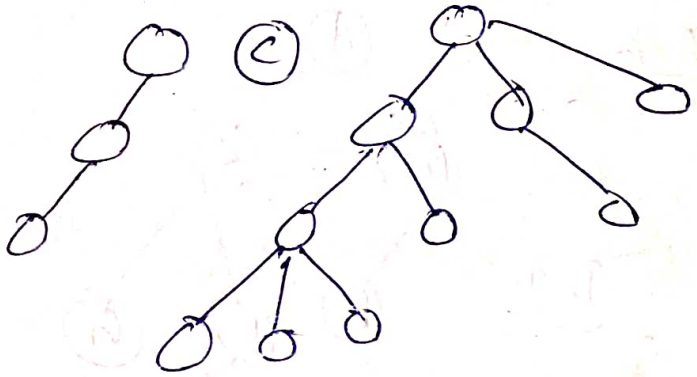
Tree Unit-5

Tree is a non linear data structure which allows a parent child relationship bet^n various pieces of data and thus allows us to arrange our records, data and files in hierarchical fashion.

Unlike natural trees, a tree data structure is depicted upside down with the root at the top and the leaves at the bottom.

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

Basic:— (a) ○ (b)

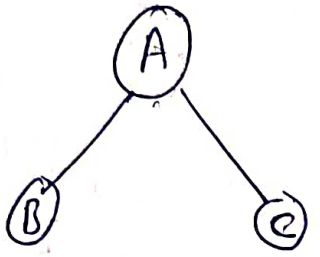


Basic Terminology:

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.

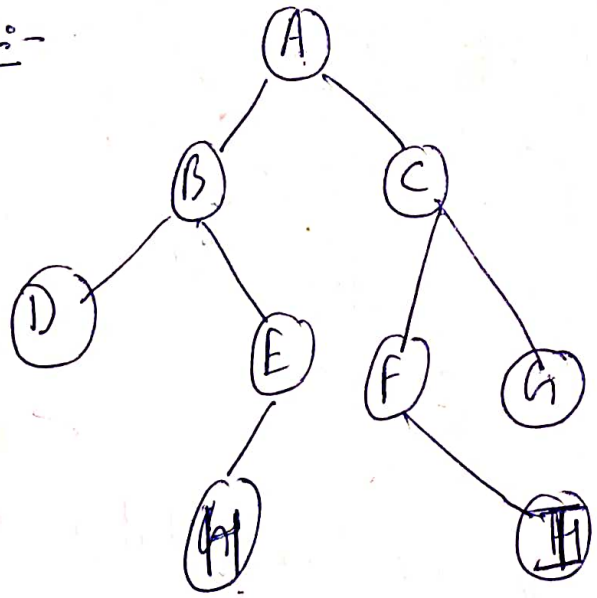
③ Parent: The parent of a node is the immediate predecessor of that node. In figure, A is the parent of B and C.



④ child: The immediate successors of a node are called childs of that node. In above figure, B and C are childs of A.

⑤ Leaf node (External node) or terminal node: A leaf node which does not have any child node.

Exp:-



leaf nodes are D, G, H, I

⑥ Siblings: The child nodes of a given parent node are called siblings. (same father for all nodes)

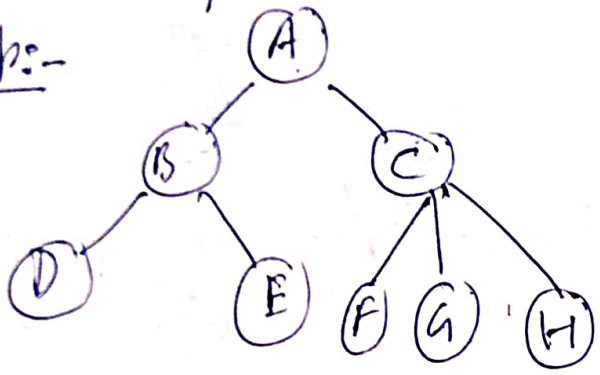
Exp:- (B, C), (D, E), (F, G) are siblings.

(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.

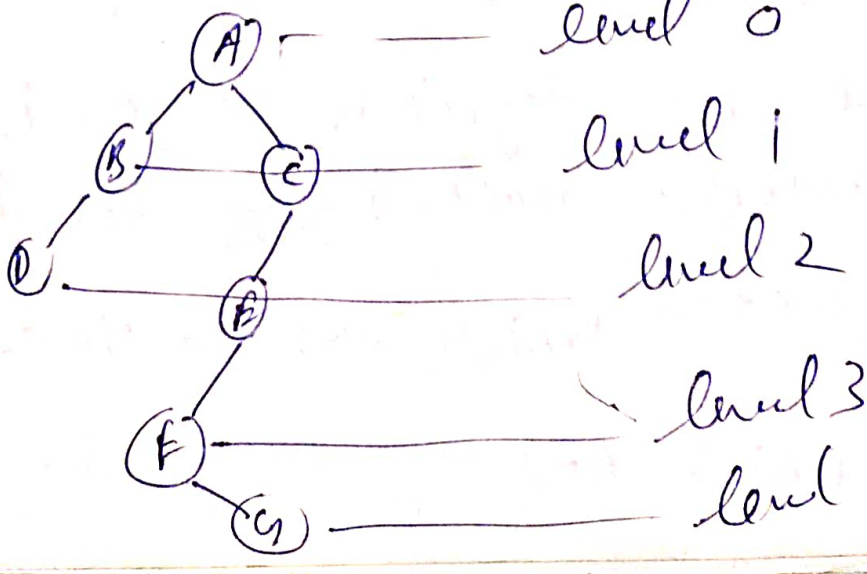
Exp:-



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

(10) Degree of a tree: It is the maximum degree of nodes in a given tree. Degree of above tree is 3.

(11) Level: The entire tree structure is leveled in such a way that the rood node is at level 0. and it's immediate children are at level 1 and so on.



Ancestor :- X is say to Ancestor of Y if Y lies any of the subtree of node X.

Descendant : X is say to descendant of Y if X lies any of the subtree of the node Y.

Internal node or non-terminal node :

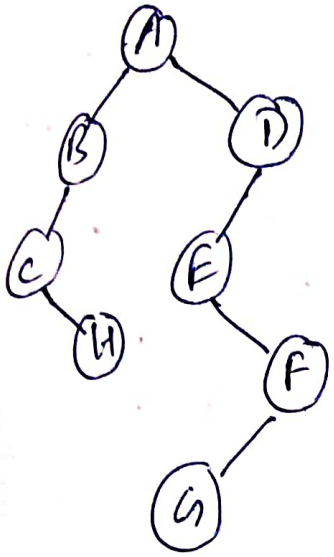
Any node that whose degree is not zero is called the internal node. Except The Root.

Empty or Null Tree :- which has no node.

Rooted Tree :- which has only one node.

Height of node: Number of edges in longest path from that node to leaf node.

Exp:

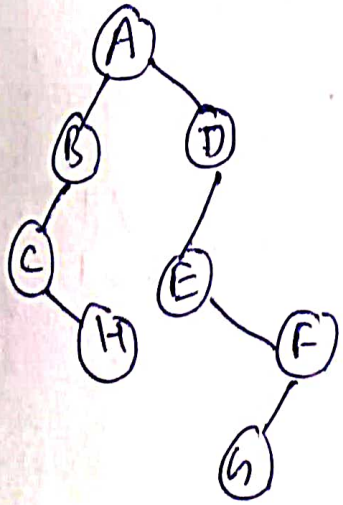


Height of A = 4
B = 2
C = 1
D = 3
E = 2
F = 1
G = 0
H = 0

Height of tree: No of edges in longest path from root to leaf nodes.

Exp: Above tree height = 4

Depth of node: Number of edges from that node to root node.



Depth of A = 0
B = 1
C = 2
D = 1
E = 2
F = 3
G = 4
H = 3

Depth of tree: Number of edges in longest path from leaf node to root node.

Exp: Depth of above tree = 4

	h given		n given	
	Max. Nodes	Min. Nodes	Max. Height	Min. Height
Binary tree	$2^{h+1}-1$	$h+1$	$n-1$	$\lceil \log_2(n+1) \rceil - 1$
Full tree	$2^{h+1}-1$	$2 \cdot h+1$	$\frac{n-1}{2}$	$\lceil \log_2(n+1) \rceil - 1$
complete binary tree	$2^{h+1}-1$	2^h	$\log_2 n$	$\lceil \log_2(n+1) \rceil - 1$

Note:

Max nodes means min height

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

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

$h+1 = \log_2(n+1)$

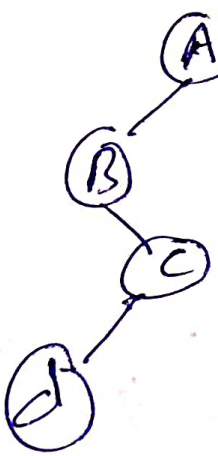
$h = \lceil \log_2(n+1) \rceil - 1$

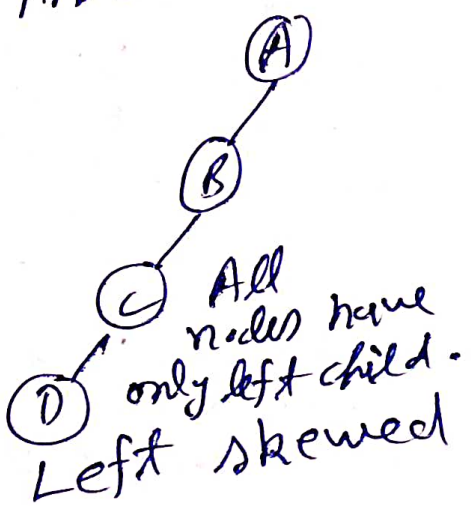
Min. nodes means max. height.

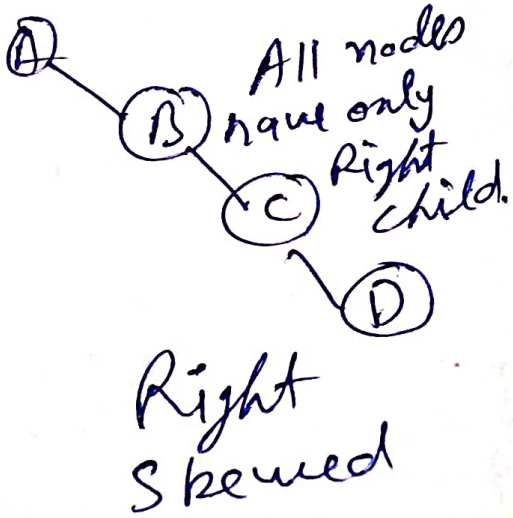
So $h+1 = n$ So $h = n-1$.

Same for full or complete tree.

Degenerate tree (Pathological tree): A tree where every internal nodes has one child.



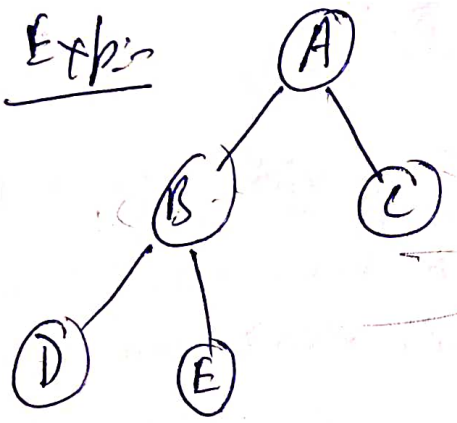




The nodes with same level number are said to belong to the same generation.

~~Similar~~ 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 a binary tree is at most two.

Exp:-

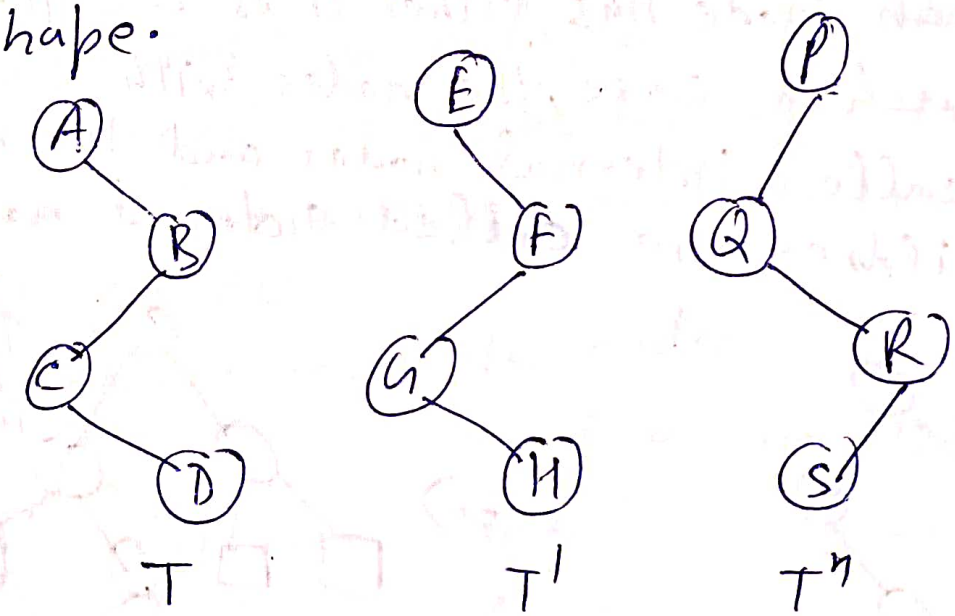


Max. possible nodes at any level = 2^l

Max. no of nodes in binary tree with height $h = 2^{h+1} - 1$

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

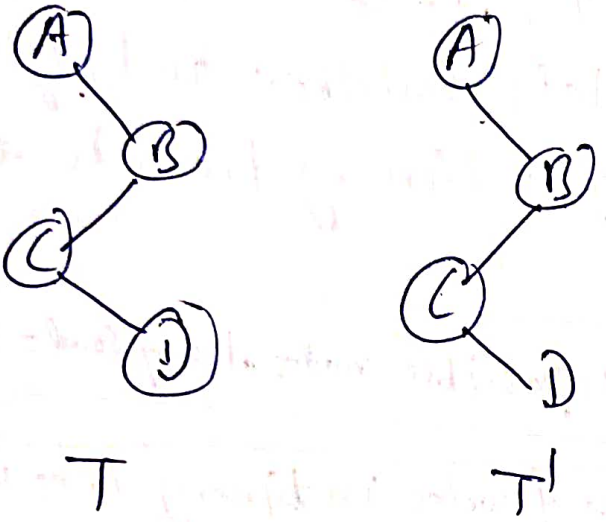
Exp:-



T & T' are similar but T & T'' are not similar

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

Eg:-



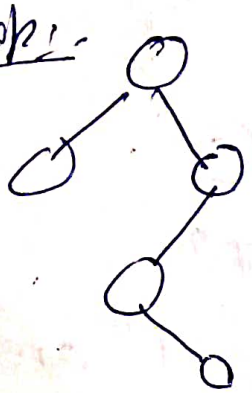
T T'

T & T' are copies.

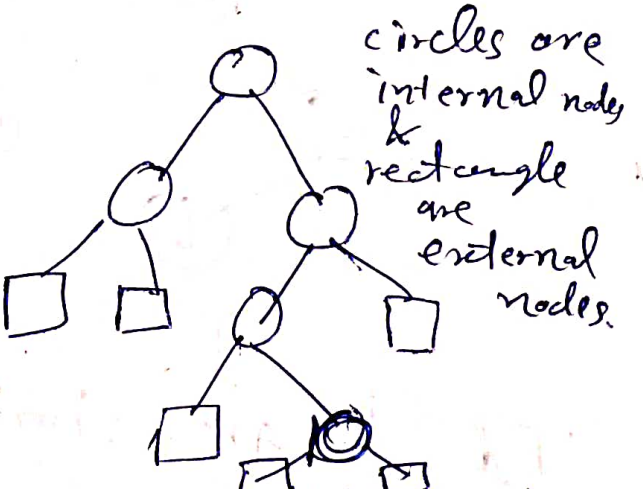
Type of binary tree:-

① Extended binary tree: A binary tree T is said to be an extended binary tree or 2-tree if each node 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.

Eg:-



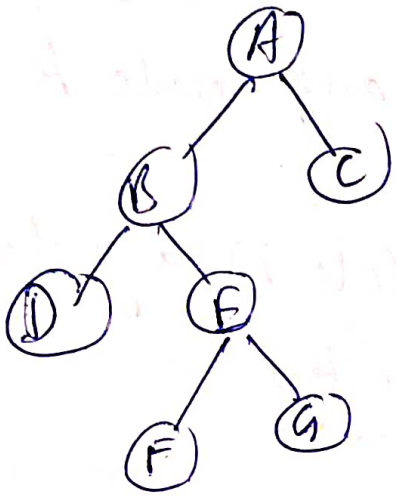
binary tree
Not-extended tree



Extended binary tree

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

Exp:-

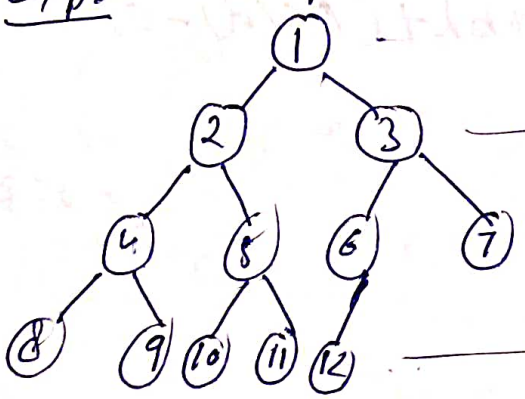


This is also called 2-Tree or full tree. or proper tree.

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 Tn with exactly n nodes.

Exp:- complete tree of 12 nodes.



level 0 max. possible nodes = 1

level 1 _____ = 2

level 2 _____ = 4

level 3 (last level) _____ = 8

In a complete binary tree for any node k, left child will be 2k and right child will be 2k+1. [Assume that root element is stored in A[1].

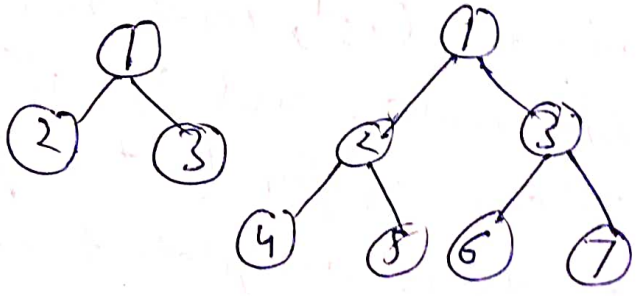
In a complete binary tree any node k has its parent at $\lfloor \frac{k}{2} \rfloor$ place.

The depth (Height) of complete binary tree T_n with n nodes is given by

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

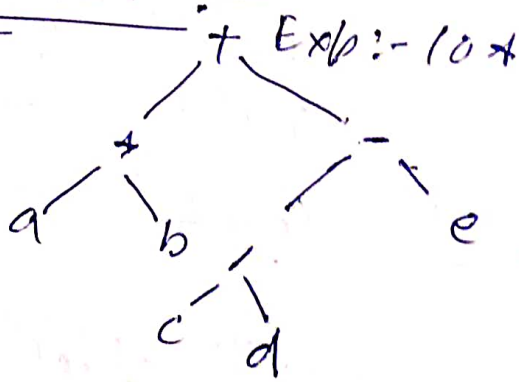
Perfect Full Tree:- The tree T is said to be full if all it's level have maximum number of possible nodes.

Exp:-



Representation of algebraic expression in the form of tree :-

Exp:- (a*b)+(c-d)-e)



Representation of binary tree:-

(1) Sequential representation (Static or Array):

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

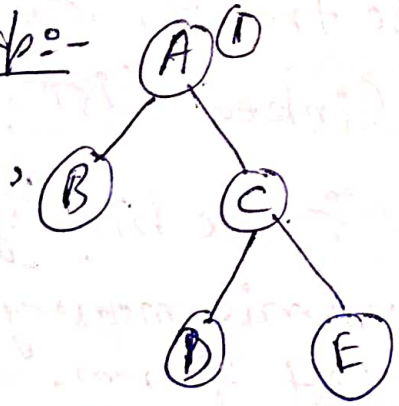
(1) The root of T is stored in Tree(1).

(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[1] = NULL then tree is empty.

If height of tree is h then size of array to represent binary tree will be $2^{h+1}-1$.

Exp:-



height of tree = 3

So size of array = $2^{3+1}-1 = 15$

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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	-	-	D	E					-	-	-	-

Tree * - is used for NULL.

Advantage:-

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

Disadvantage:-

① Not suitable for normal binary tree but it is ideal for complete binary tree.
② Here most of the array entries are empty i.e. wastage of memory.
③ Addition and deletions of nodes are inefficient because of the data movements in the array.
④ There is no way to enhance the tree structure.

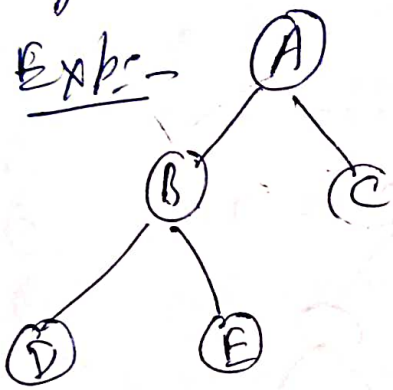
② Representation of Tree using linked list (Dynamic representation) :- The binary tree can be represented using dynamic memory allocation of a node in a linked list form. In linked list allocation technique a node in a tree contains Three field:

Left	Info	Right

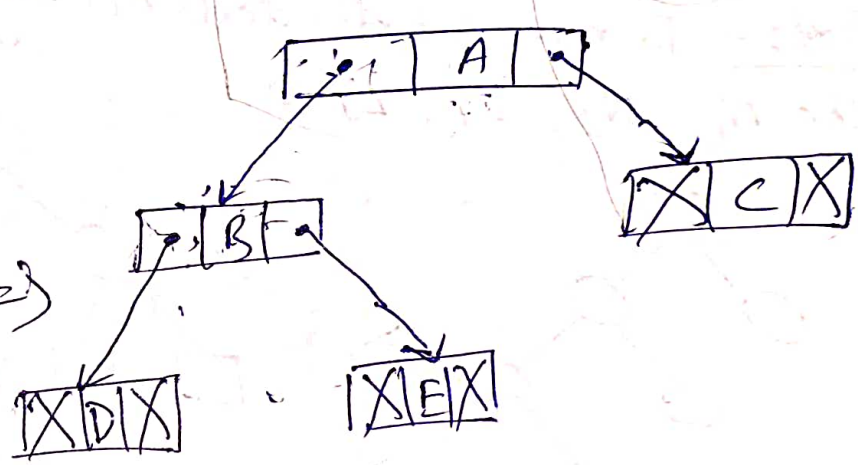
① Left ② Info ③ Right

Here Info is for information field of node.
~~Left is for left child of node which points the~~
second member Left contains the address of left child. if node has no left child then Left should be NULL.
Third member Right contains the address of right child if node has no right child then Right should be NULL.

Exp:-



=>




```
struct node
{
    int Info;
    struct node *Left;
    struct node *Right;
};
```

Advantage	Disadvantage
① No wastage of memory.	① In this pointer fields are involved which take more space then just data field
② Enhancement of tree is possible	② Programming language do not support dynamic allocation have difficulty to implement tree
③ Insertion & Deletion involve no data movement	

Path length in Extended binary tree:

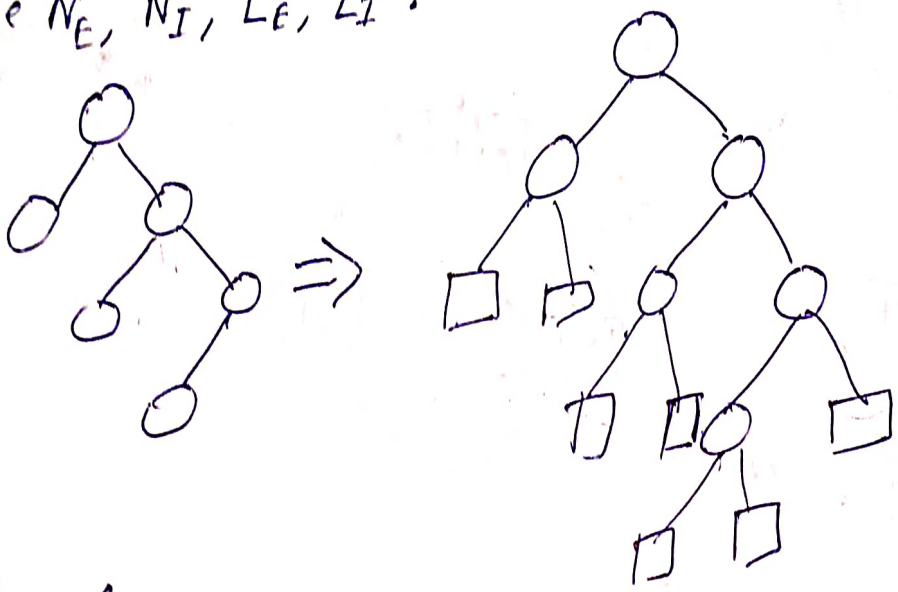
External Path length (L_E) = Sum of all path lengths from each external node to root.

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

* $N_E = N_I + 1$ where N_E is Number of External nodes.
  — N_I — Internal nodes.

$E = L_I + 2N_I$

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



$N_I = 6$

$N_E = 7$

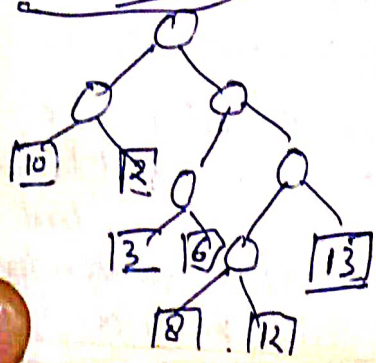
$P_E = 2+2+3+3+4+4+3$

$P_E = 21$

$P_I = 0+1+1+2+2+3$

$= 9$

Weighted external Path length:

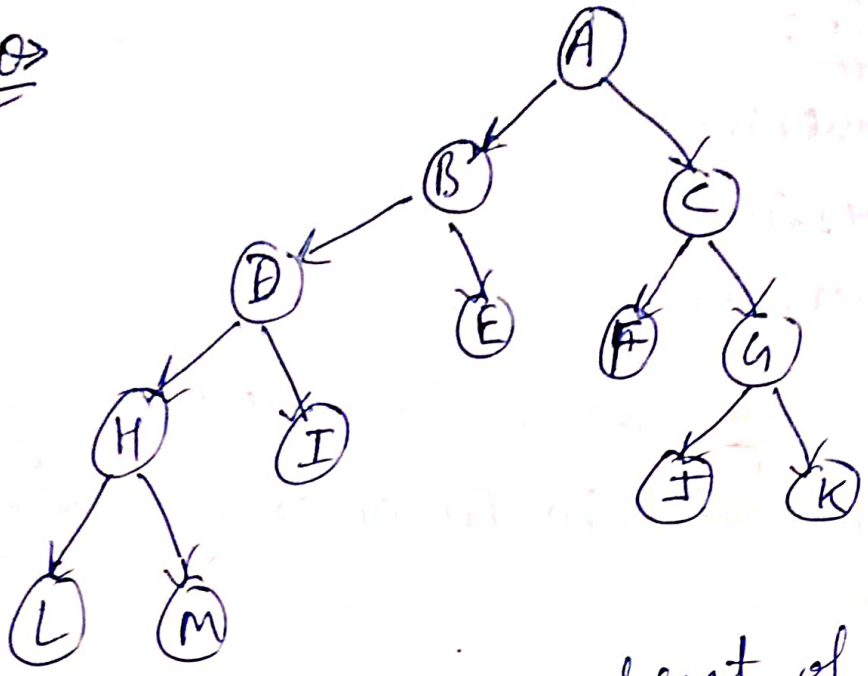


$P_{WE} = 10\times2 + 2\times2 + 3\times3 + 6\times3 + 8\times4 + 12\times4 + 13\times3$

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

$= 170$

Q⇒



(1) Right & left descendent of node B.
(2) write all siblings.
(3) H & F are same generation, D & G same Generation
(4) Leaf nodes of tree.
(5) write internal nodes of Tree.
(6) which is root
(7) Depth of Tree
(8) ~~locate~~ C is 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:

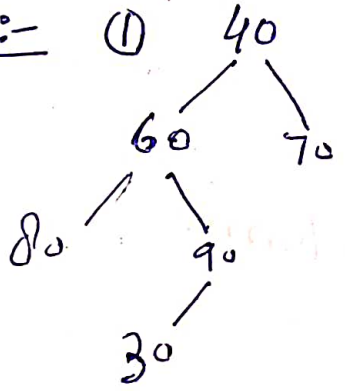
1. Pre-order traversing
2. In order traversing
3. Post order traversing

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

1. Process the Root R
2. Traverse the left subtree of R in Pre-order.
3. Traverse the right subtree of R in Pre-order.

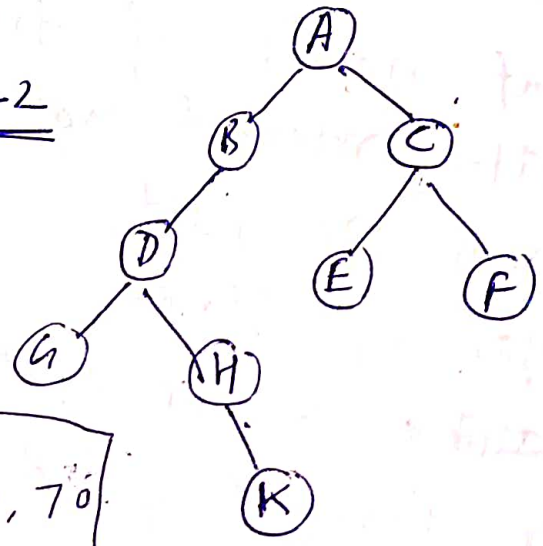
Root, Left, Right

Exp:- 1



Pre order 40, 60, 80, 90, 30, 70

Exp:-2



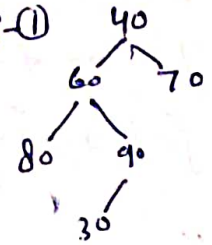
Pre order: A, B, D, G, H, K, C, E, F

In order traversing:- We can traverse any binary tree T with root R in In-order by following rules:

1. Traverse the left subtree of R in In-order
2. Process the Root R.
3. Traverse the right subtree of R in In order

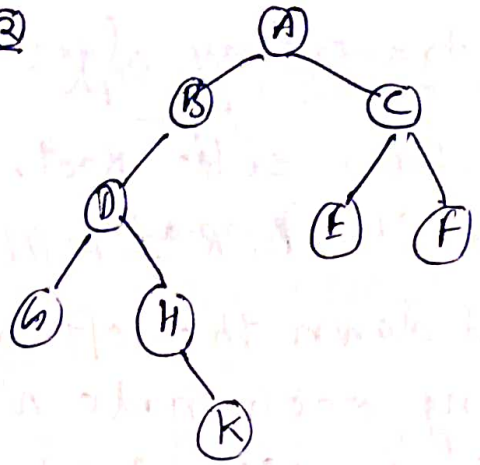
Left, Root, Right

Exp:-①



In-order: 80, 60, 30, 90, 40, 70

Exp②



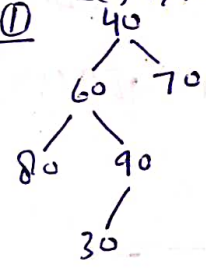
In-order: G, D, H, K, B, A, E, C, F

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

① Traverse the left subtree of R in Post order.
② Traverse the right subtree of R in Post order
③ Process the root R.

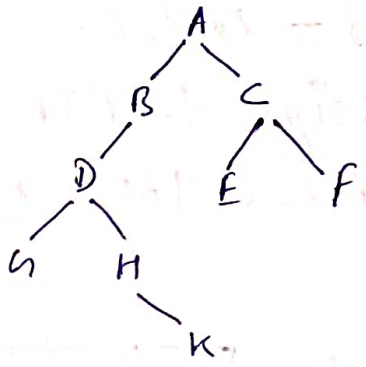
Left T, Right, Root

Exp①



Post order: 80, 30, 90, 60, 70, 40

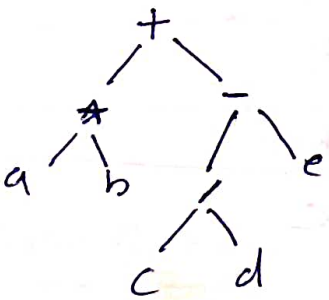
Exp②



Post order: G, K, H, D, B, E, F, C, A

Exp:- Represent following expression in tree form & also write Pre-order, Post-order, In-order.

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



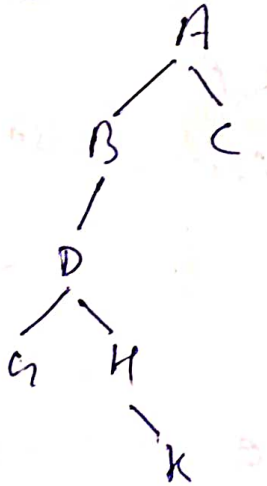
Pre order: + * ab - / c d e
Post order: a b * c d / e - +
In order: a * b + c / d - e

Pre order traversing algo: Initially push NULL on to STACK then set PTR := Root. Then repeat the following steps while PTR ≠ NULL.

(a) 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 to STACK. The traversing ends after a node N with no left child L(N) is processed. Thus PTR is updated using the assignment ~~PTR = PTR~~. PTR = LEFT[PTR] and traversing steps when LEFT[PTR] = NULL.

(b) Pop & assign to PTR the top element on STACK. If PTR ≠ NULL then return to step(a) otherwise exit.

Exp:-



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

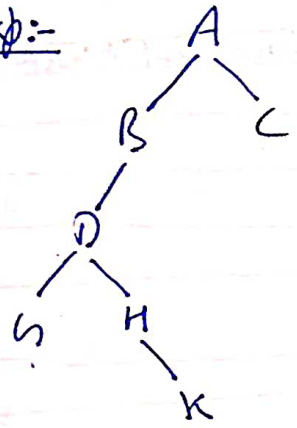
Pre Order : A B D G H K C

Traversing algo for In-order: Initially push NULL on to STACK and then set PTR := ROOT. Then repeat the following steps untill NULL is popped from STACK.

(a) Proceed the left most path rooted on PTR, pushing each node N onto STACK and stopping when a node N with no left child is pushed onto STACK.

(b) Pop & 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 (a).

Ex:-



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

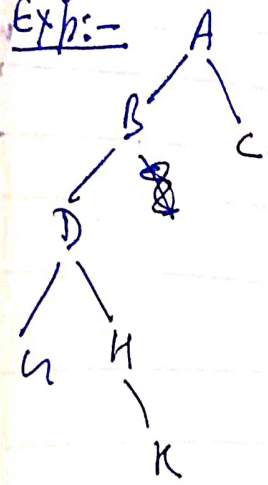
In-order : G D H K B A C

Post order traversing algo: Initially push NULL onto STACK and then set PTR := ROOT and then repeat the following steps untill NULL is popped from stack.

(a) Proceed the left most path rooted at PTR, At each node N of the path, push N on to STACK and, if N has a right child R(N), push -R(N) on to stack.

(b) Pop and process positive nodes on stack, if NULL is popped then exit, If a negative node is popped i.e. if PTR = -N for some node N, set PTR=N by assigning PTR := -PTR and return to step (a).

Exp:-



Post order:

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

STACK	Processed Node
ϕ	
ϕA.	
ϕA-C	
ϕA-CB	
ϕA-CBD.	
ϕA-CBD-H	
ϕA-CBD-HG	
ϕA-CBD-H	(G)
ϕA-CBD	-H → H
ϕA-CBDH	
ϕA-CBDH-K	
ϕA-CBDH	-K → K
ϕA-CBDHK	
ϕA-CBDH	(K)
ϕA-CBD	(H)
ϕA-CB	(D)
ϕA-C	(B)
ϕA	-C → C
ϕAC	
ϕA	(C)
ϕ	(A)

Pre-order algo:-

PREORDER(INFO, RIGHT, LEFT, ROOT):-

① Set TOP := 1, STACK[1] = NULL, PTR = ROOT.
② Repeat steps ③ to ⑤ while PTR ≠ NULL.
③ Apply PROCESS to INFO[PTR].
④ If RIGHT[PTR] ≠ NULL then
Set TOP := TOP+1 and STACK[TOP] = RIGHT[PTR]
⑤ If LEFT[PTR] ≠ NULL then
Set PTR := LEFT[PTR]
Else
Set PTR := STACK[TOP] and TOP = TOP-1.
⑥ Exit.

INORDER(INFO, RIGHT, LEFT, ROOT) :-

① Set TOP = 1, STACK[1] = NULL, PTR = ROOT.
② Repeat while PTR ≠ NULL
(a) set TOP = TOP+1 and STACK[TOP] = PTR
(b) PTR = LEFT[PTR]
③ Set PTR = STACK[TOP] and TOP = TOP-1
④ Repeat steps 5 to 7 while PTR ≠ NULL
⑤ Apply PROCESS to INFO[PTR]
⑥ If RIGHT[PTR] ≠ NULL then
(a) set PTR = RIGHT[PTR]
(b) go to step 2
⑦ Set PTR = STACK[TOP] and TOP = TOP-1.
⑧ Exit

POSTORDER (INFO, RIGHT, LEFT, ROOT) –

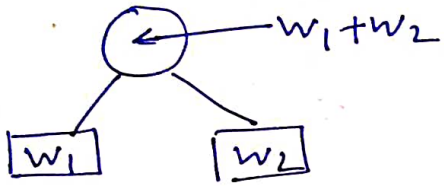
① Set TOP = 1, STACK[1] = NULL and PTR = ROOT.
② Repeat steps 3 to 5 while PTR ≠ NULL.
③ Set TOP = TOP+1 and STACK[TOP] = PTR
④ If RIGHT[PTR] ≠ NULL then
  Set TOP = TOP+1, STACK[TOP] = – RIGHT[PTR].
⑤ Set PTR = LEFT[PTR].
⑥ Set PTR = STACK[TOP] and TOP = TOP–1
⑦ Repeat while PTR > 0
  (a) Apply PROCESS to INFO[PTR].
  (b) Set PTR = STACK[TOP] and TOP = TOP–1.
⑧ If PTR < 0 then
  (a) Set PTR = – PTR
  (b) go to step 2
⑨ exit.

Huffman algorithm:

Huffman algo is used to find minimum weighted path length for a tree.

(1) Let there are n weights $w_1, w_2, w_3 \ldots w_n$.

(2) Take two minimum weights and create a subtree. Suppose w_1 and w_2 are two minimum weights then the subtree will be



(3) Now the remaining n-1 weights will be

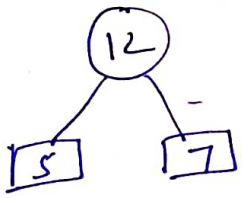
$w_1+w_2, w_3, w_4, \ldots w_n$.

(4) Create all subtrees at the last weight. ~~Now 2 tree is the desired solution.~~

Exp:-

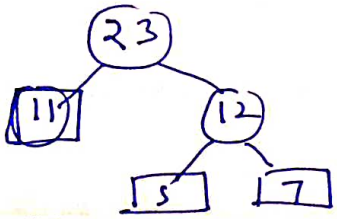
Nodes	A	B	C	D	E	F	G
weights	16	11	7	20	25	5	16

Step (1) - Take Two nodes of minimum weights as 7 & 5. & create subtree



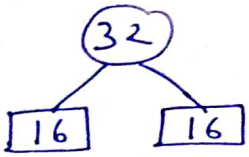
Now list is 12, 16, 11, 20, 25, 16

Step-2:- Take Two minimum weights as 11, 12 & create subtree



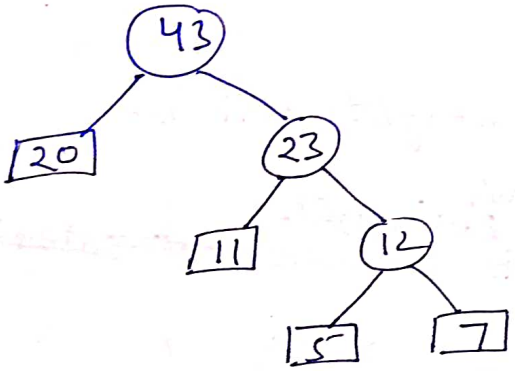
Now list is 23, 16, 20, 25, 16

Step 3 - Take two minimum weights as 16, 16 & create subtree



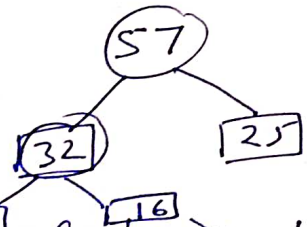
Now list is 32, 23, 20, 25.

Step 4 - Take Two minimum weights as 20, 23 & create subtree



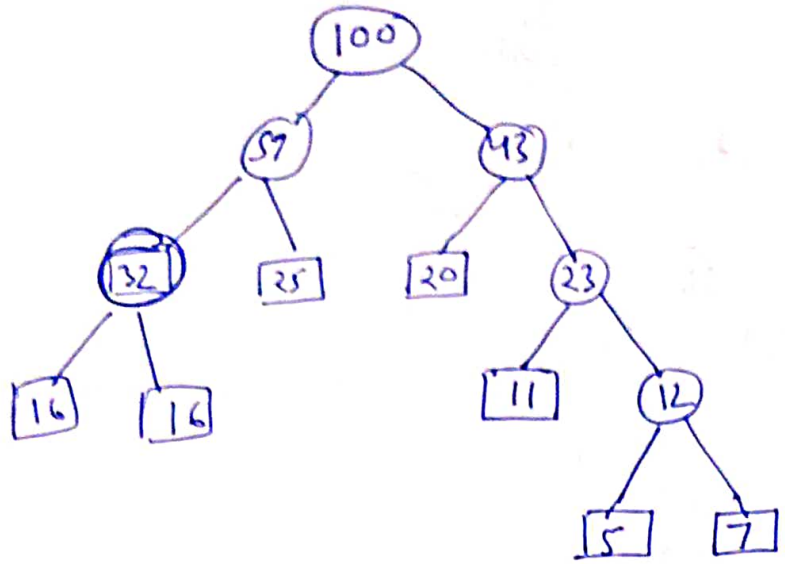
Now list is 43, 32, 25.

Step 5:- Take two minimum weights as 32, 25 & create subtree



Now list is 43, 57.

Step 6:- Take two minimum weights as 43, 57 & create subtree.



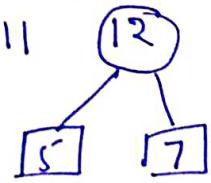
ans:-

OR

Node :	A	B	C	D	E	F	G
weight	16	11	7	20	25	5	16

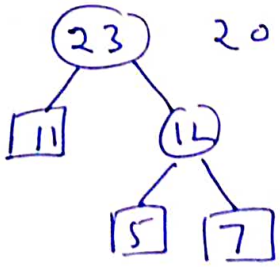
Step-1 minimum weights are 5,7 so now list is

16 11 12 20 25 16

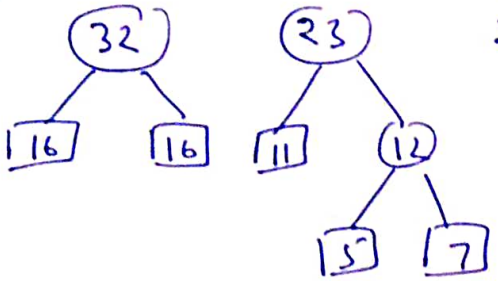


Step2- minimum weight are 12,11 so now list is

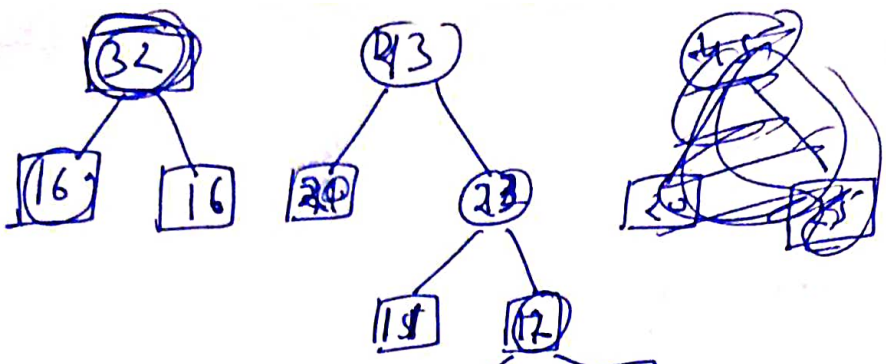
16 23 20 25 16



Step 3- minimum weights are 16,16 so now list is

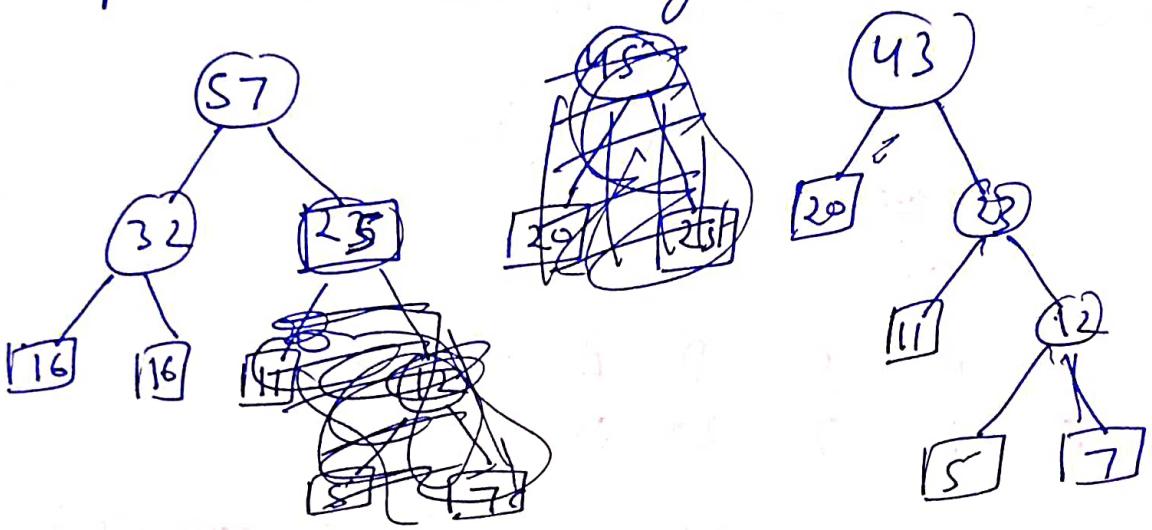


Step 4- minimum weights are 20, 23 now list is

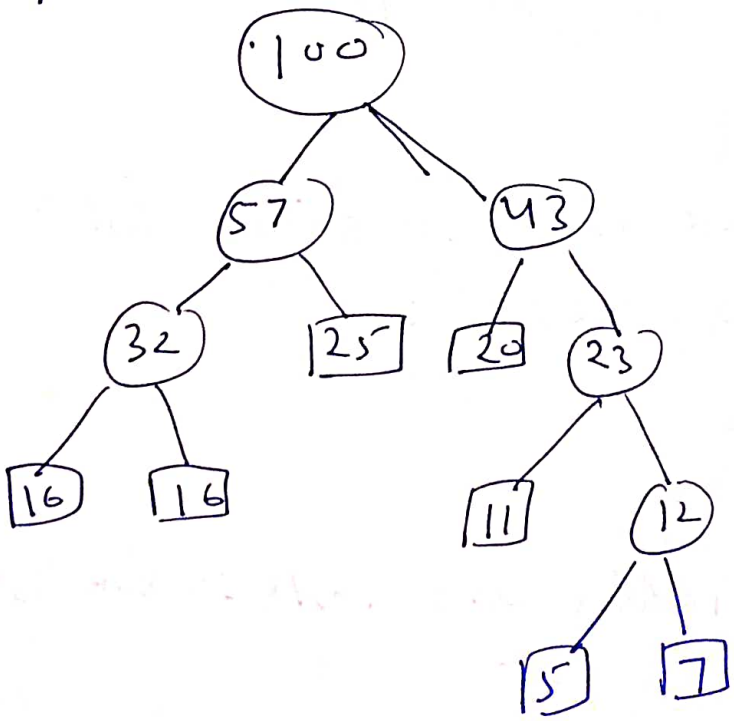


25

Steps - minimum weights are 32, 25 & now list is



Step 6 - now Two minimum weights are 43, 57



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 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.

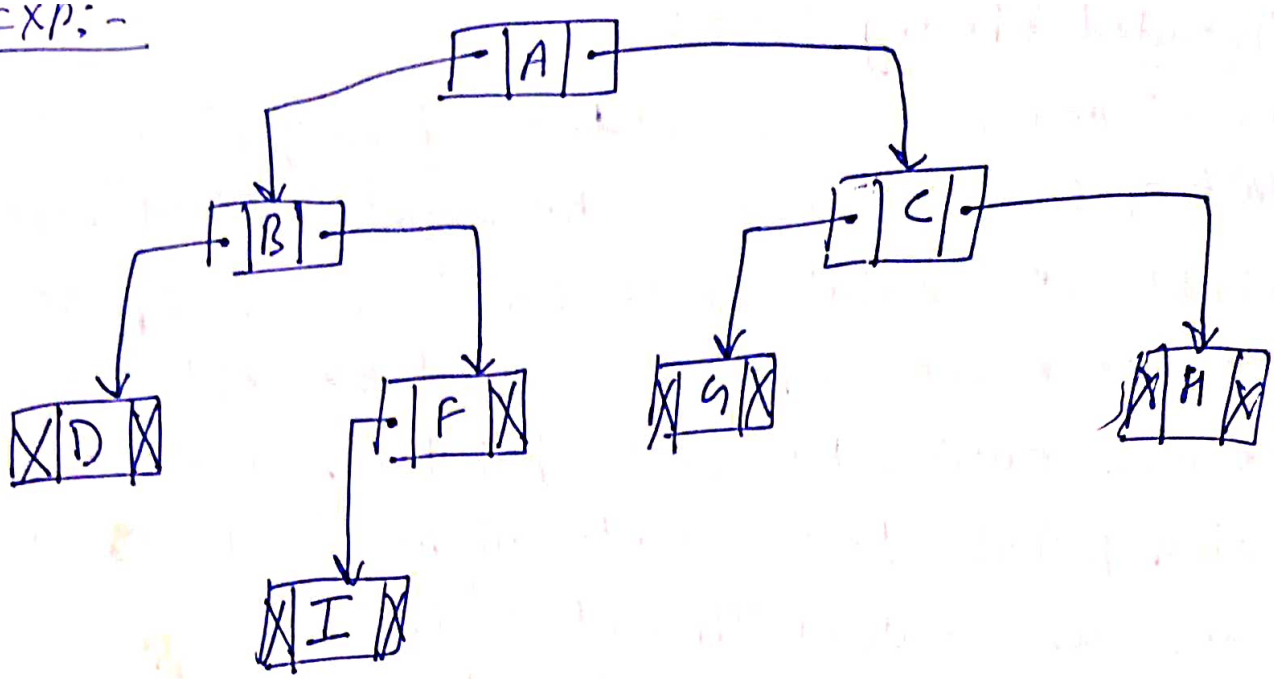
There are many ways to thread a binary tree. These are:

① The right NULL pointer of each node can be replaced by a thread to the successor of that node under In-order traversal called a right thread, and the tree will called a right threaded tree.

② The left NULL pointer of each node can be replaced by a thread to the predecessor of that node under In-order traversal called a left thread & the tree will called a left threaded tree.

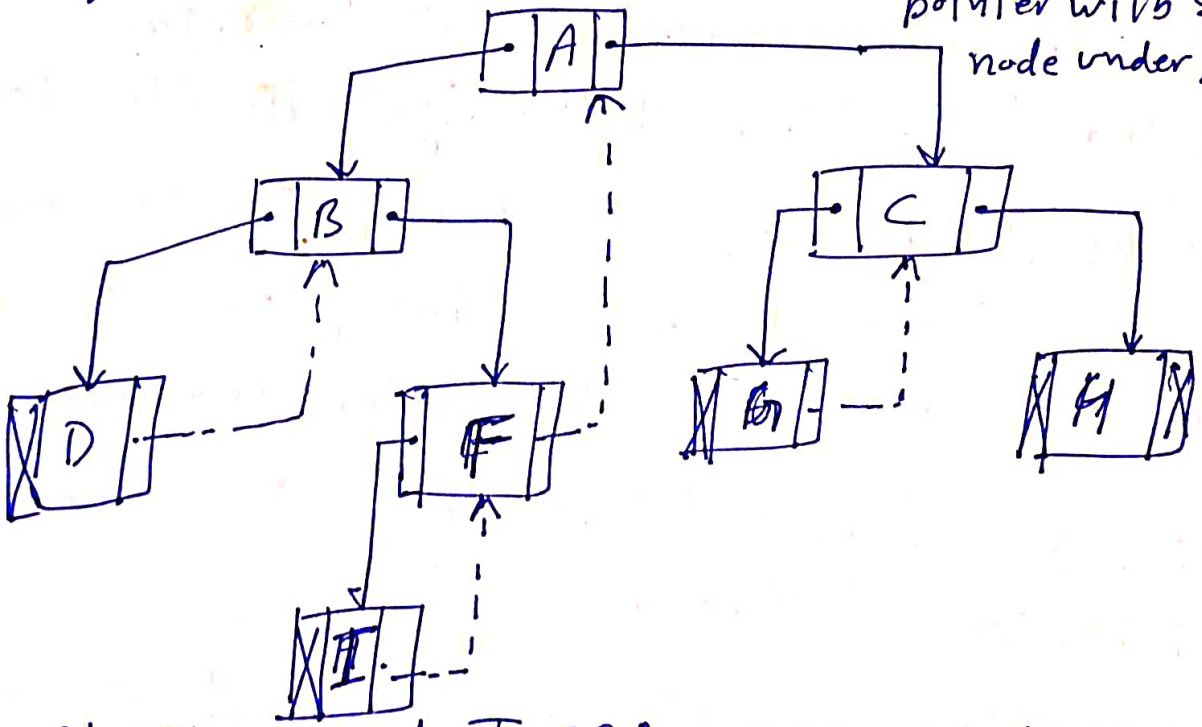
③ Both left and right NULL pointers can be used to point to predecessor and successor of that node, respectively, under In-order traversal. Such a tree called a fully threaded tree.

Exp:-

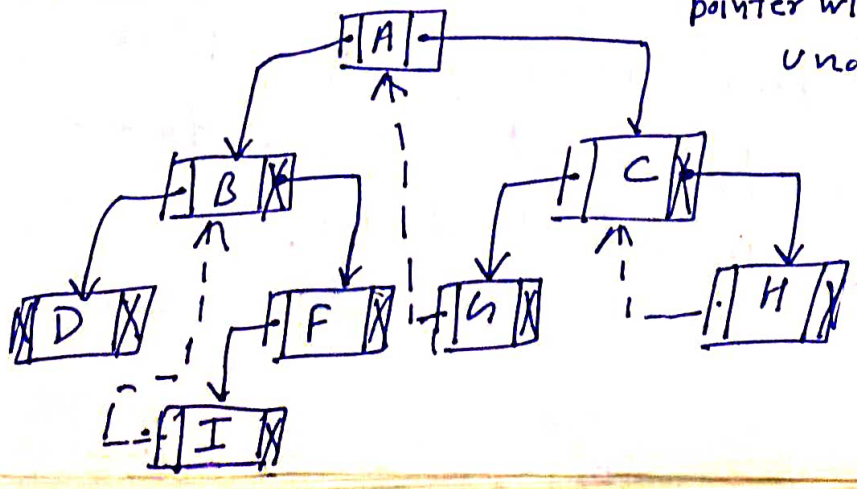


In order of tree: D, B, I, F, A, G, C, H

① Right Threaded Tree:- Replace Right NULL pointer with successor node under In order.

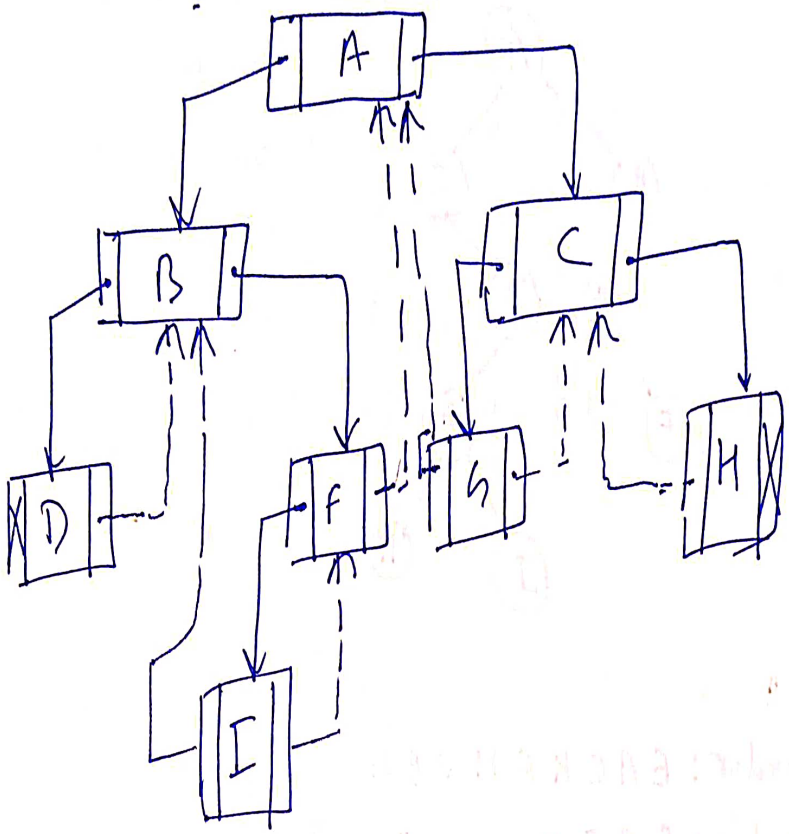


② Left Threaded Tree: Replace Left NULL pointer with Predecessor node under In-order.



(3) Fully Threaded Tree:

Replace left & right NULL Pointer with predecessor & successor node under In-order.



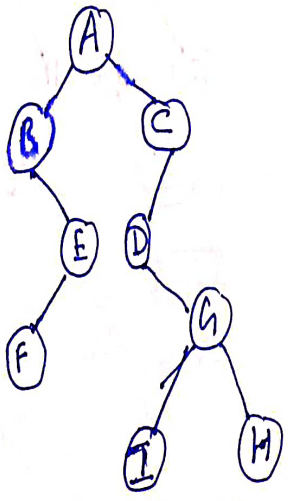
Ø In order : B F E A D I G H C

Preorder : A B E F C D G I H

create binary tree ?

ans:- In order : B F E A D I G H C.

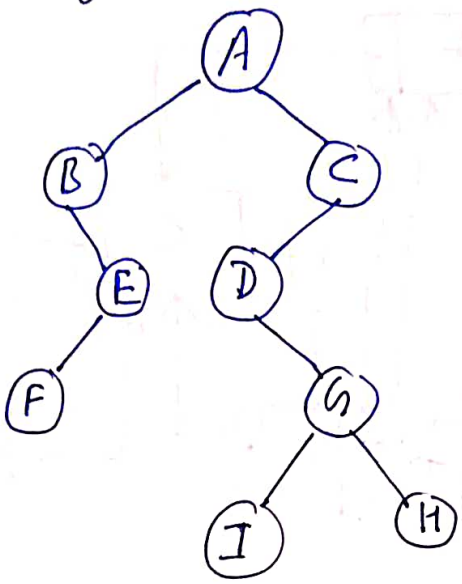
Pre order : A B E F C D G I H



Q> In order : B F E A D I G H C

Post order : F E B I H G D C A

Create binary tree.



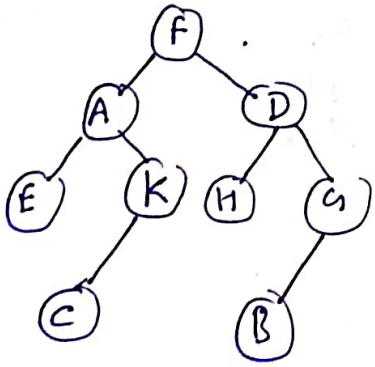
Q> In order : E A C K F H D B G

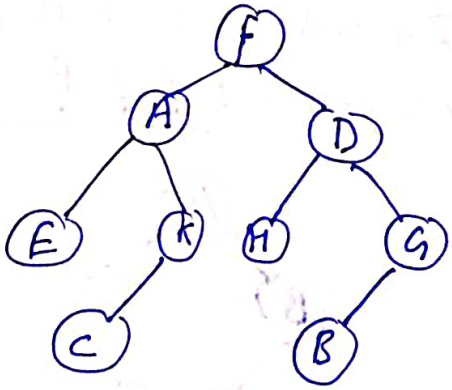
Pre order : F A E K C D H G B

Create binary tree?

In order : E A C K F H D B G

Pre order : F A E K C D H G B





Q> In order: E A C K F H D B G

Post order: E C K A H B G D F

Create binary tree?

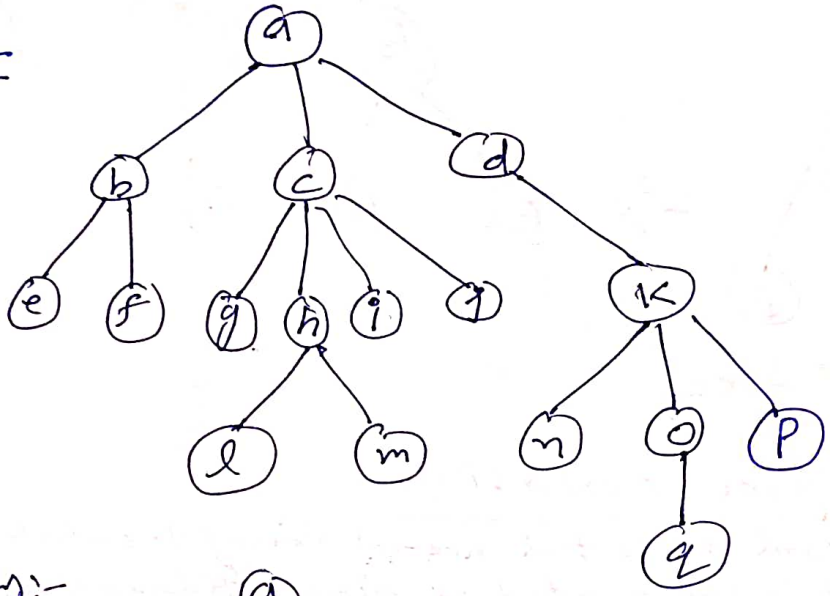
Convert General tree to binary tree:

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

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

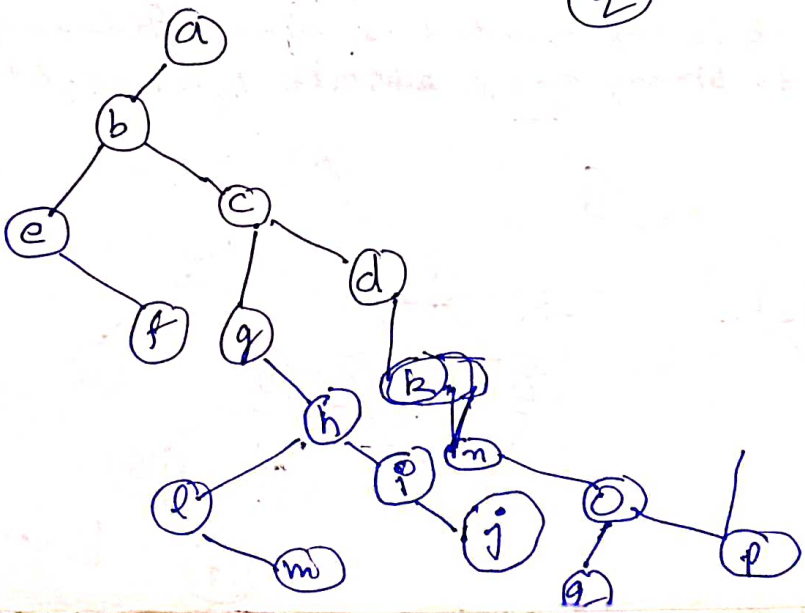
(2) If $X_1, X_2, \ldots X_N$ are child of X in General tree then make X_1 is left child of binary tree and X_2 is right child of X_1, X_3 is right child of X_2, X_N is right child of X_{N-1} in binary tree

Exp:-



convert into binary

ans:-



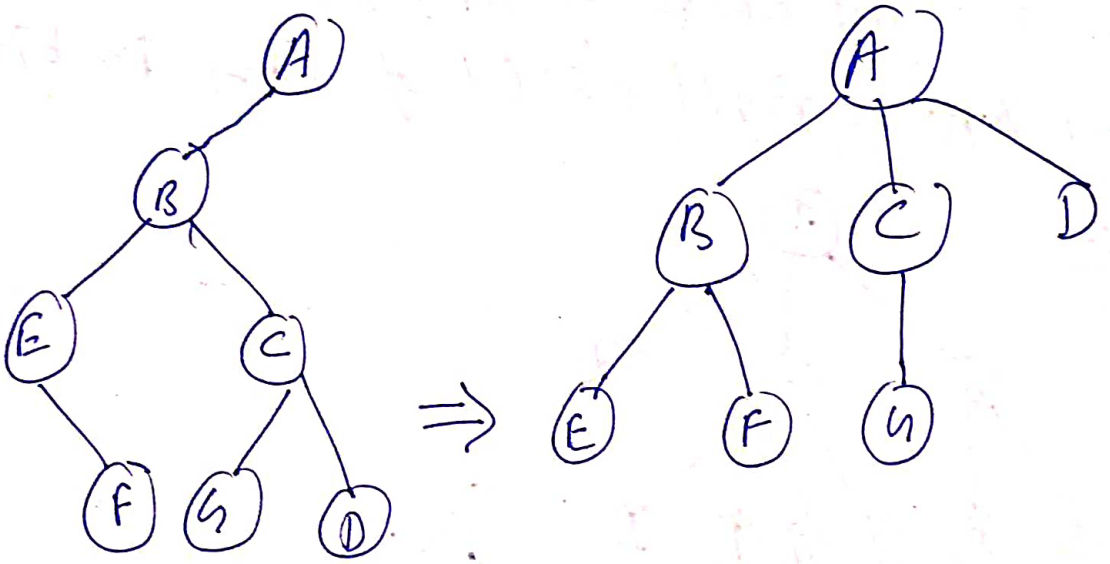
Conversion of binary tree to general tree:

Method:-

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

② If X_1 is the left child of node X in binary tree and X_2 be the right child of X_1 .--- and X_N is right child of X_{n-1} then make $X_1, X_2, X_3, \rightarrow X_N$ the child of X in equivalent general tree.

Exp:-

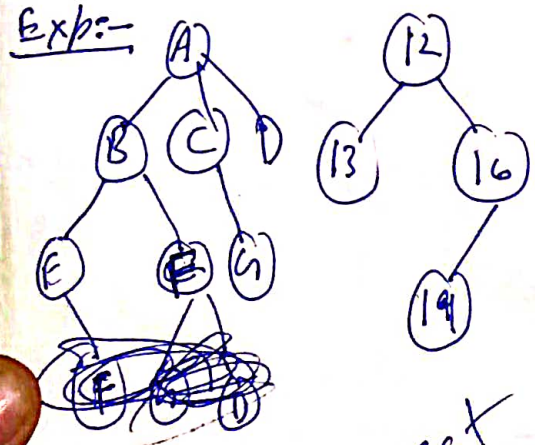


binary tree

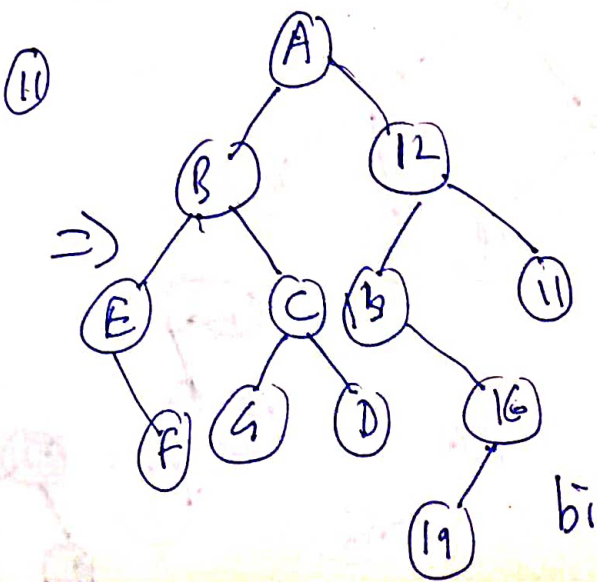
Forest to binary tree conversion:-

Forest is a set of several trees that are not linked to each other. Initially left most tree is represented as binary tree B_1 after that ~~make~~ second tree to binary tree & ~~add~~ make B_2 the right child of B_1. Convert B_2 & so on---.

Exp:-



forest



binary

Binary Search tree (BST) :-

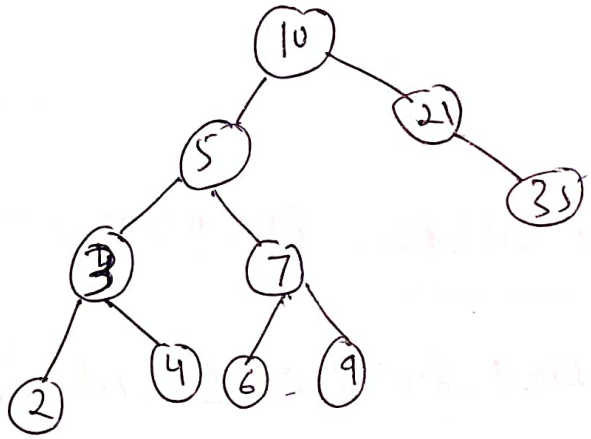
A binary search tree is a binary tree which satisfies the following rules:

① The value of left child or left subtree of node is less than the value of the ~~root~~ node.

② The value of right child or right subtree of node is greater than or equal to the value of the ~~root~~ node

③ All the children either left or right should follow the above two rules.

Exp:-



Operation in binary search tree:

① Searching in binary search tree

② Insertion in binary search tree

③ Deletion in binary search tree

Algo for searching in BST:—

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

① Set PTR = ROOT and PAR = NULL.

② Repeat steps ③ while PTR ≠ NULL.

③ If ITEM = INFO[PTR] then PTR->INFO then
Set LOC = PTR and Return.

Else If ITEM < INFO[PTR] then PTR->INFO then
Set PAR = PTR and PTR = LEFT[PTR]. PTR->LEFT

Else
set PAR = PTR and PTR = RIGHT[PTR]. PTR->RIGHT

④ Set LOC = NULL.

⑤ Exit

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

① Call Search (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).

② If LOC ≠ NULL then exit.

③ Set NEW->INFO = ITEM
INFO[NEW] = ITEM and NEW->RIGHT = NULL
RIGHT[NEW] = NULL
and LEFT[NEW] = NULL NEW->LEFT = NULL.

④ If PAR = NULL then set ROOT = NEW
Else If ITEM < INFO[PAR] then PAR->INFO then
set LEFT[PAR] = NEW PAR->LEFT = NEW

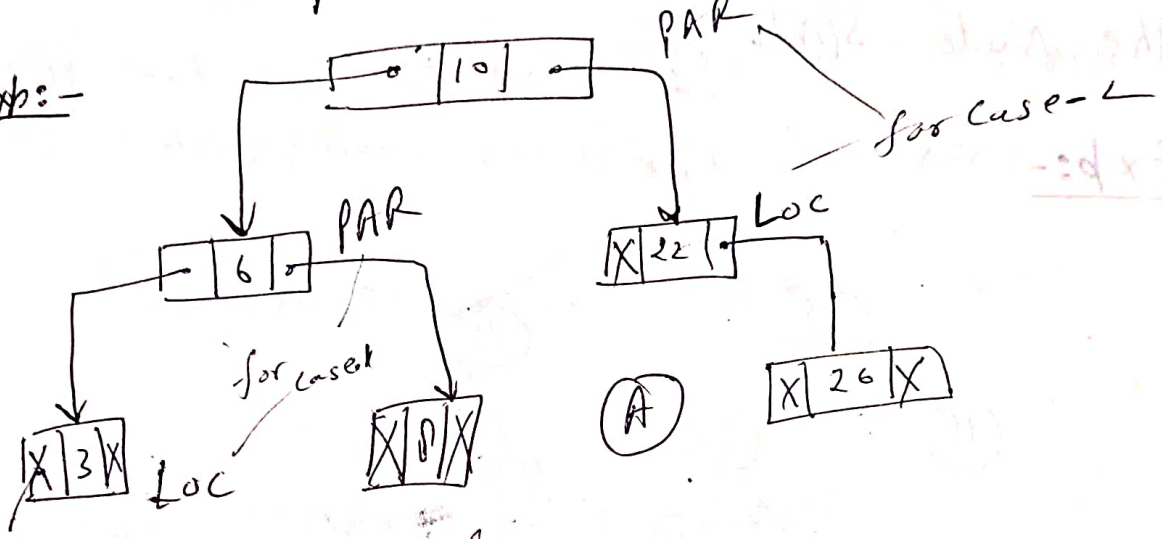
Else
set RIGHT[PAR] = NEW PRR->RIGHT = NEW

⑥ Exit

Deletion in binary search tree:

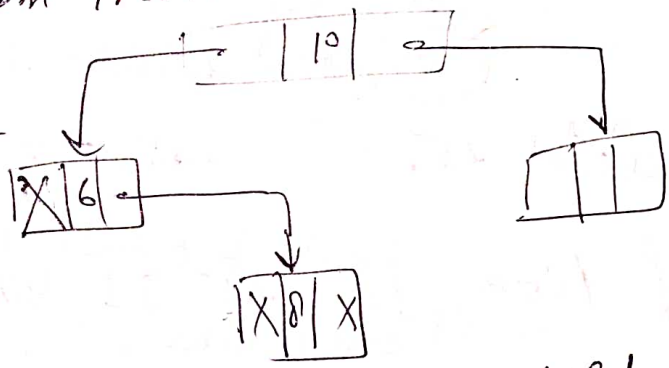
case: 1 N has no childrem. Then N is deleted from tree of m by replacing the location of N in the Parent node P(N) by the NULL pointer.

~~case~~ **Exp:-**



(i) Delete 3 from Tree A.

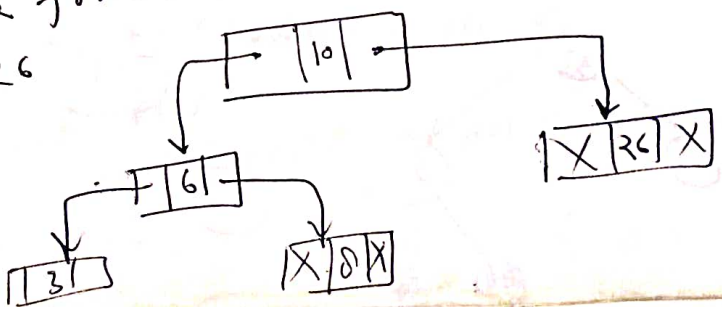
child of 3 = NULL



Case: 2 N has exactly one child. Then N is deleted by replacing the location of N in P(N) by the location of the only child of N.

Exp:- Delete 22 from Tree A.

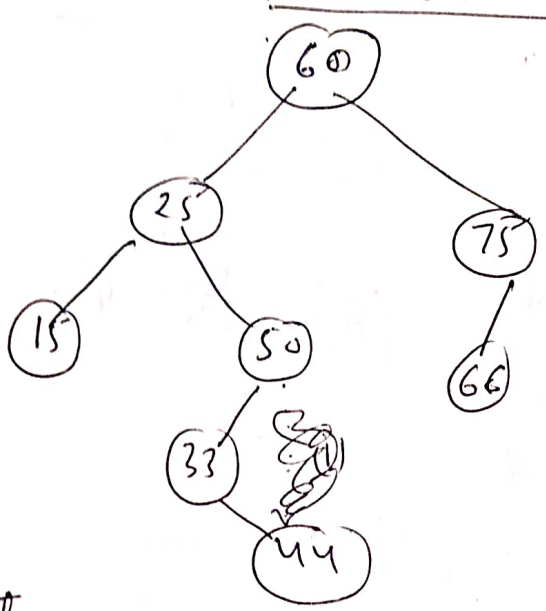
child of 22 = 26



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).

* S(N) does not have left child.

Exp:-



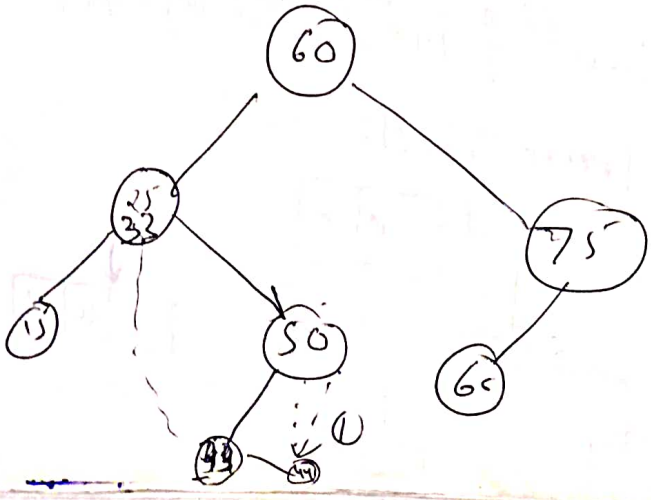
~~Delelot~~ Delete 25.

ans:-

Inorder of tree: 15 25 33 44 50 60 ~~66~~ 75

Inorder successor of 25 = 33

ans:-



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

(1) Call Search (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
(Find the location of ITEM & its parent)

(2) If LOC = NULL then write ITEM not in Tree and Exit.

(3) If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL then
(RIGHT[LOC] → LOC->RIGHT; LEFT[LOC] → LOC->LEFT)

call CASE3 (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
Else
call CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

(4) Exit.

CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

(1) If LEFT[LOC] = NULL & RIGHT[LOC] = NULL then
(LEFT[LOC] → LOC->LEFT; RIGHT[LOC] → LOC->RIGHT)

Set CHILD = NULL.
Else if LEFT[LOC] ≠ NULL (LOC->LEFT)
Set CHILD = LEFT[LOC] LOC->LEFT
Else
Set CHILD = RIGHT[LOC] LOC->RIGHT

(2) If PAR ≠ NULL then (PAR->LEFT)
If LOC = LEFT[PAR] then
Set LEFT[PAR] = CHILD (PAR->LEFT)
Else
set RIGHT[PAR] = CHILD (PAR->RIGHT)
Else
set ROOT = CHILD

(3) Return.

CASE B (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

(1) (Find SUC and PARSUC)
SUC = LOC->RIGHT and PARSUC = LOC
(a) Set PTR = RIGHT[LOC] and ~~TEMP = LOC~~.
(b) Repeat while ~~LEFT[PTR]~~ SUC->LEFT ≠ NULL
PARSUC = SUC SUC = SUC->LEFT
Set ~~TEMP = PTR~~ and ~~PTR = LEFT[PTR]~~

~~(c) Set SUC = PTR and PARSUC = TEMP~~

(2) ~~Delete Inorder Successor using CASEA.~~
Call ~~CASEA~~(INFO, LEFT, RIGHT, ROOT, ~~SUC, PARSUC~~).

(3) ~~Replace~~ Node N by its Inorder Successor.
(a) ~~If PAR ≠ NULL~~
{ If LOC = LEFT[PAR] then
Set LEFT[PAR] = SUC.
Else
Set RIGHT[PAR] = SUC.
}
~~Else~~
set Root = ~~child~~ SUC.

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

(4) ~~Return~~. (2) loc -> INFO = SUCC -> INFO
(3) If ~~PAR~~ (PARSUC -> LEFT == SUCC)
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 or 1.

Balance factor = Max. left subtree height − Max. Right subtree Height.

If Balance factor of node is not 0, -1, 1 then tree is unbalance and we use four type 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 unbalanced node to the new inserted node.

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

(1) Left - Left Rotation (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 (R-R):-

If new node is inserted at right of (P) and right of (C).

* - We use only left Rotation on (P).

③ Left-Right Rotation (L-R):

If new node is inserted at left of (P) and right of (C). we use two rotation.

* First we use Right-Right Rotation on (C).
* then ——— Left Left ——— on (P).

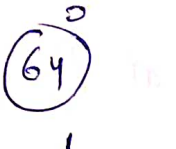
④ Right left Rotation (R-L):

If new node is inserted at Right of (P) and left of (C). we use two rotations

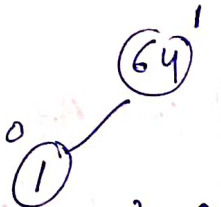
* - First we use Left-Left Rotation on (C)
* then ——— Right-Right Rotation on (P).

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

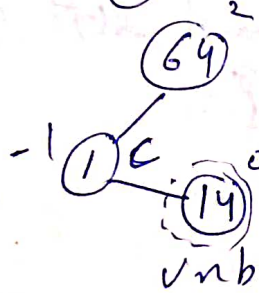
64



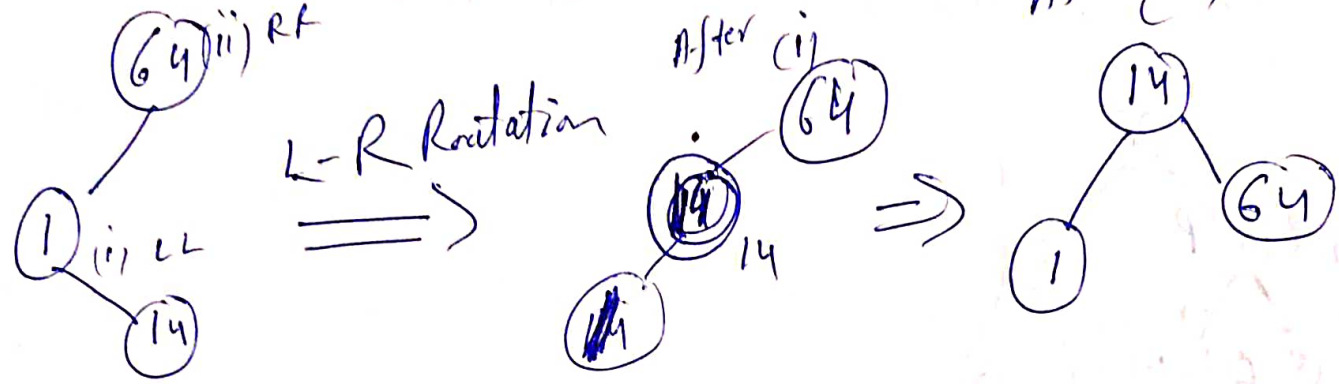
1



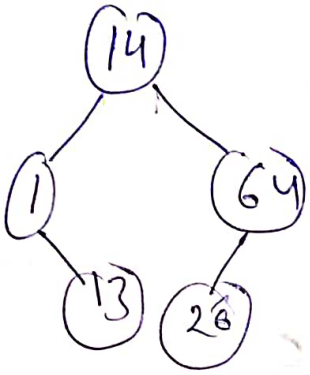
14



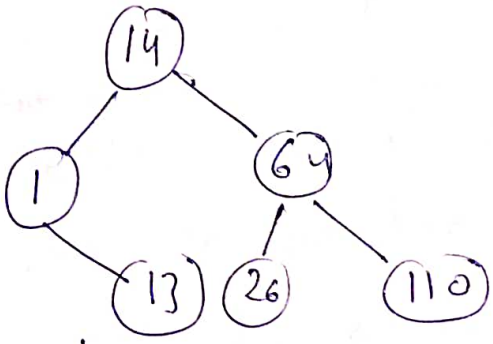
Use Left Right Rotation



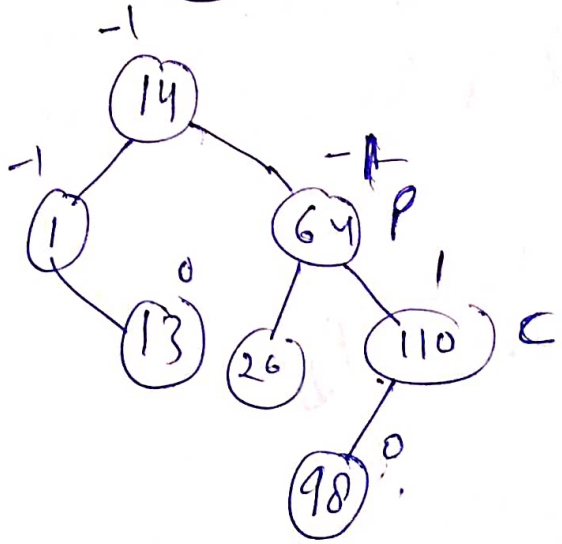
26/13



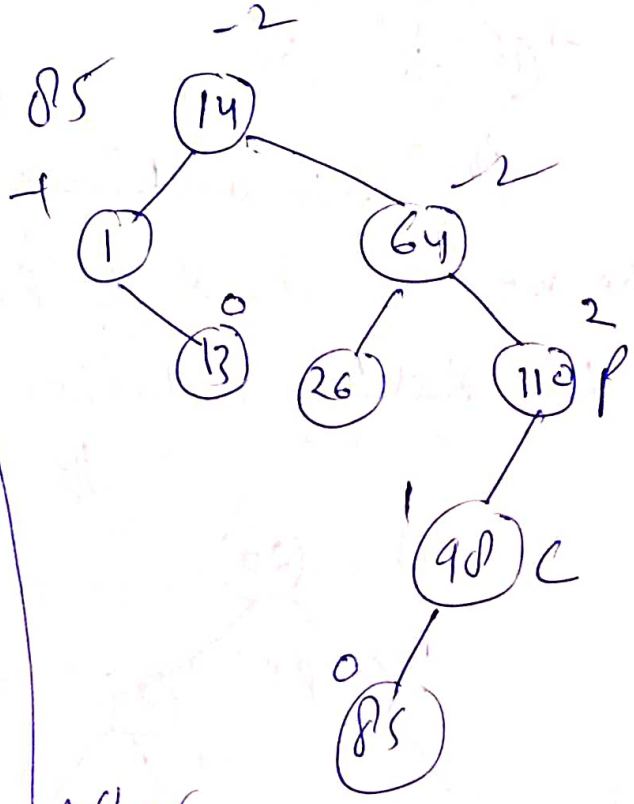
110



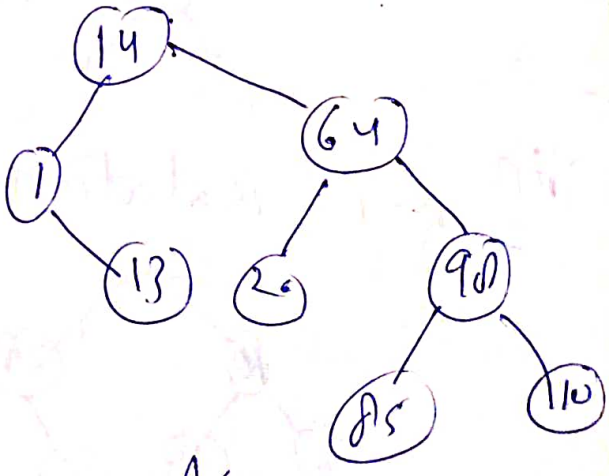
90



85



After L-L Ration



Use ~~R-L Rotation~~

Ans-

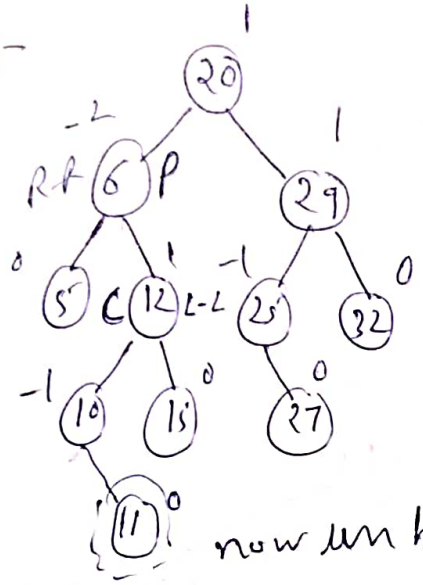
Exp:-(2) Exp3:- BCG EIFADH

u → 10, 9, 8 --- 1

(5) 1, 2, 3 --- 9

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

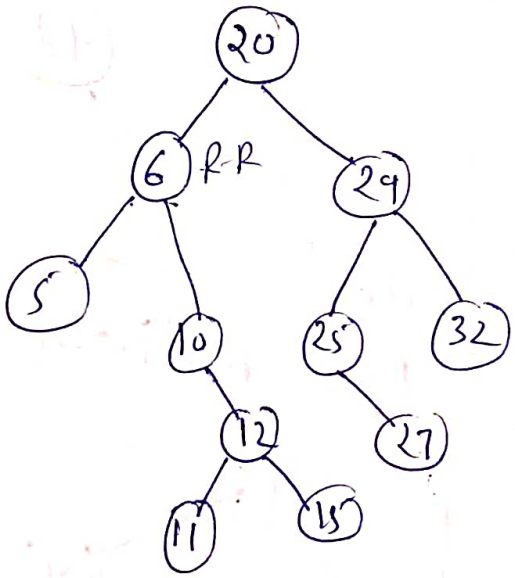
Ans:-



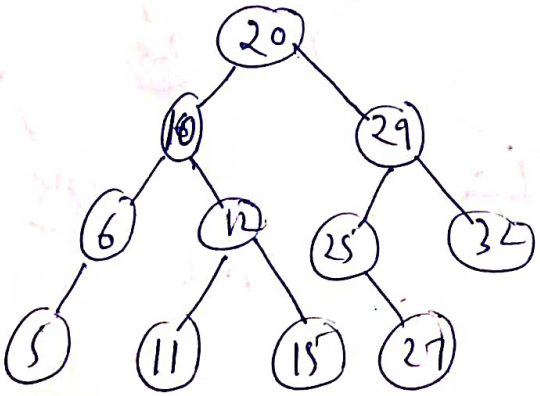
now unbalanced

So we need R-L Rotation

(i) L-L ~~Rotation~~ Rotation on (12) C



(ii) R-R Rotation on (6) P



Ans

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 search tree.

(3) After deletion of the 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.

~~x is deleted~~. 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 R_0.
C = 1 then R_1.
C = -1 then R_{-1}

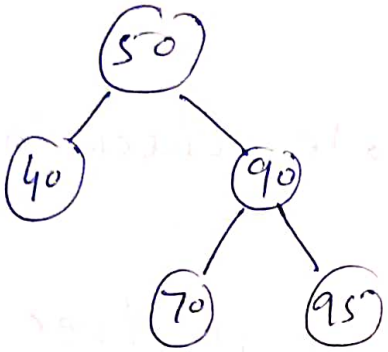
If X is Left side of P and Bf of C = 0 then L_0 used
C = 1 then L_1 used
C = -1 then L_{-1} used

R_0 = L-L Rotation , L_0 = R-R Rotation

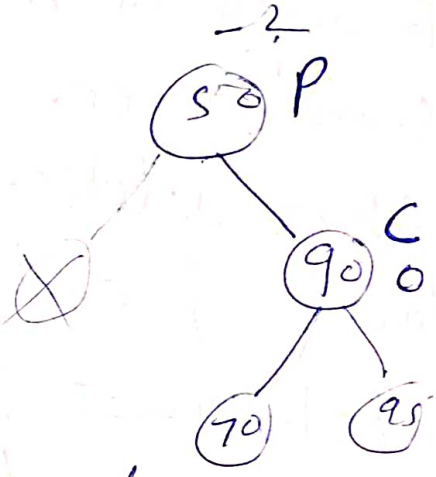
R_1 = L-L Rotation L_1 = R-R Rotation

R_{-1} = L-R Rotation L_{-1} = R-L Rotation

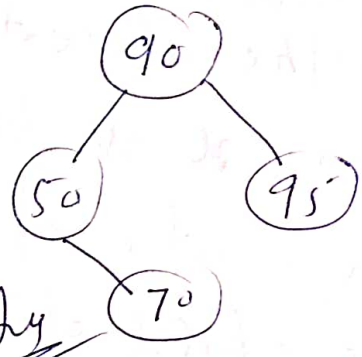
Exp:-(1)



Delete 40 →

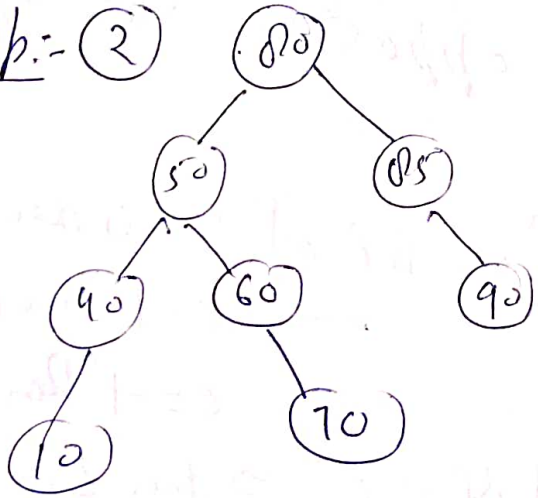


L0 used

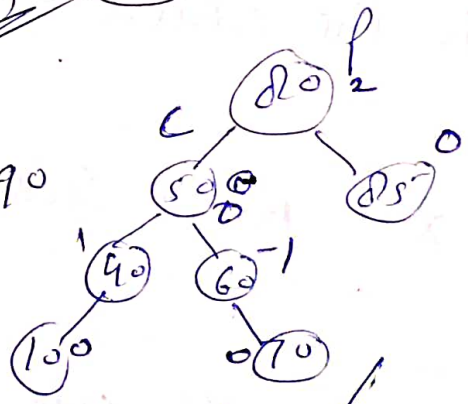


Ans

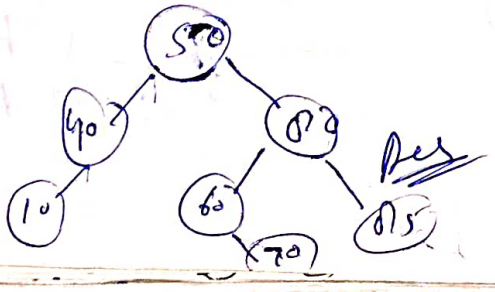
Exp:-(2)



Delete 90 →

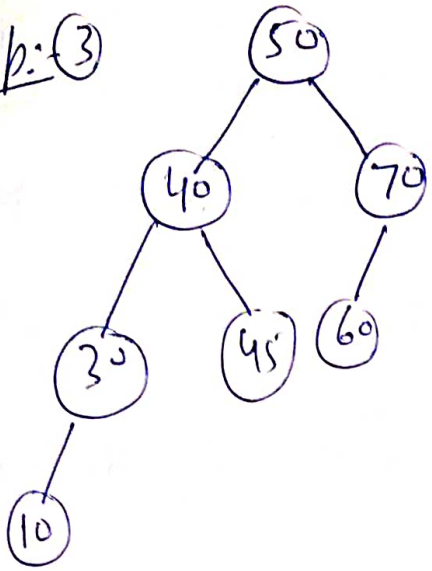


R0 used

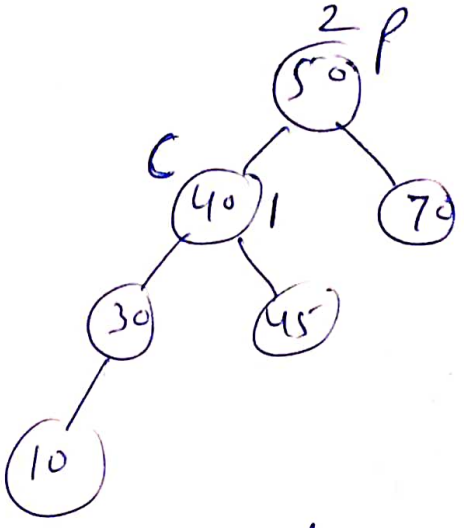


Ans

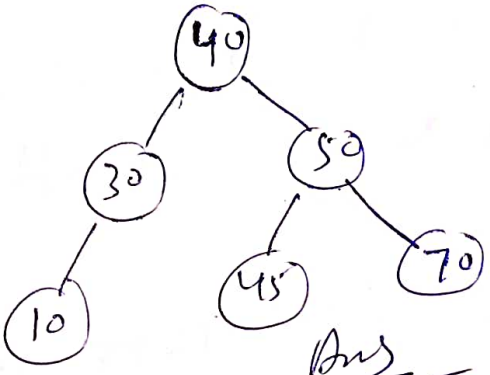
Exp:-(3)



Delete 60

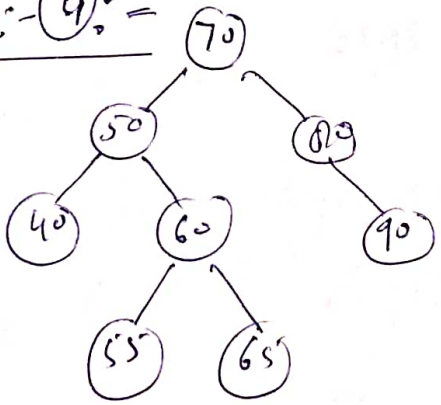


R.1 used

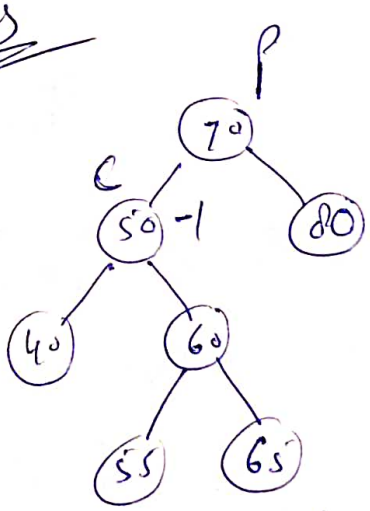


Ans

Exp:-(4):-



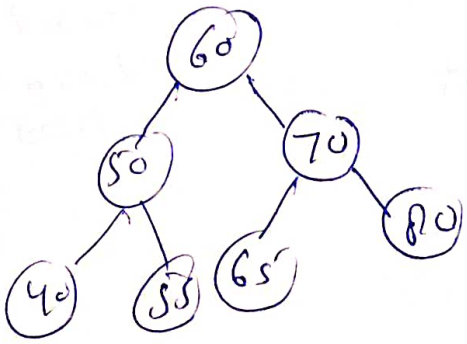
Delete 90



R-1

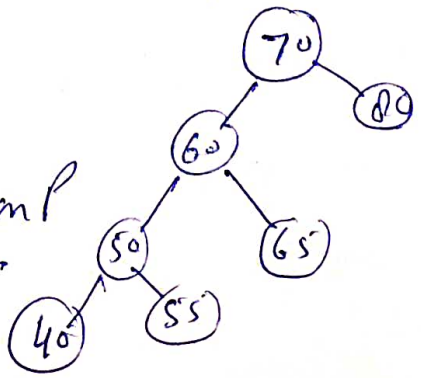
L-R used

(i) RR on C



Ans

LL on P



B-Tree:- B tree of order-n can be defined as follow

(1) Each non leaf node has atleast $\lceil \frac{n}{2} \rceil$ and max. n non empty children (Except Root)

(2) All leaf nodes will be at same level.

(3) All leaf nodes can contains max. n-1 key.

(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 $\lfloor \frac{(n-1)}{2} \rfloor$ keys.

(6) keys in non leaf node will devide the left & right subtree where value of left subtree keys will be less and values of right subtree keys will be greater than that particular key.

(7) B-Tree of order n is also called (n-1)-n tree or n-(n-1) tree.

- Exb:- B-Tree of order 5 is called 4-5 tree or 5-4 tree.

Insertion in B-Tree:- The insertion of a key in a B-Tree requires first traversal in B-Tree. Through Traversal it will find that key to be inserted is already existing or not. Suppose key already exist then no insertion required. Suppose key does not exist in tree then through traversal it will reach leaf node. Now we have two cases -

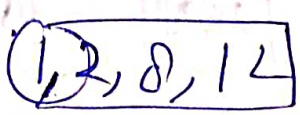
(i) Node is not full.
(ii) Node is already full

If the leaf node in which key is to be inserted is not full, then insertion is done in the node.

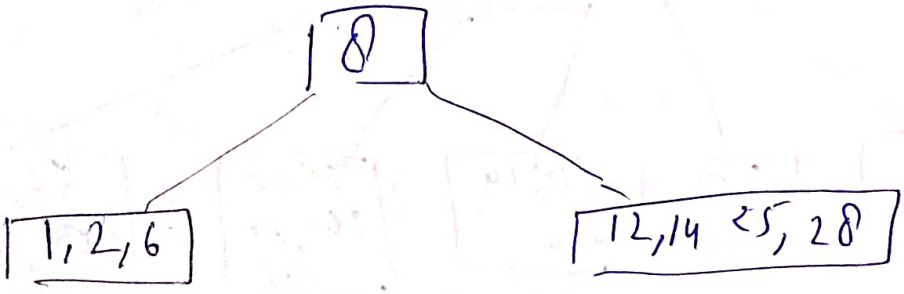
If the node were to be full, then insert the key in order into the existing set of keys in the node, split the node at it's median into two nodes at the same level, pushing the median element up by one level. Note that split nodes are only half full. Accomodate the median element in the parent node if it is not full. Otherwise repeat the same procedure and this may even call for rearrangement of the keys in the root node or the formation of a new node it self.

Exp:- Create B tree of order - 5 given keys 1, 12, 8, 2, 25, 6, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 53, 55, 45.

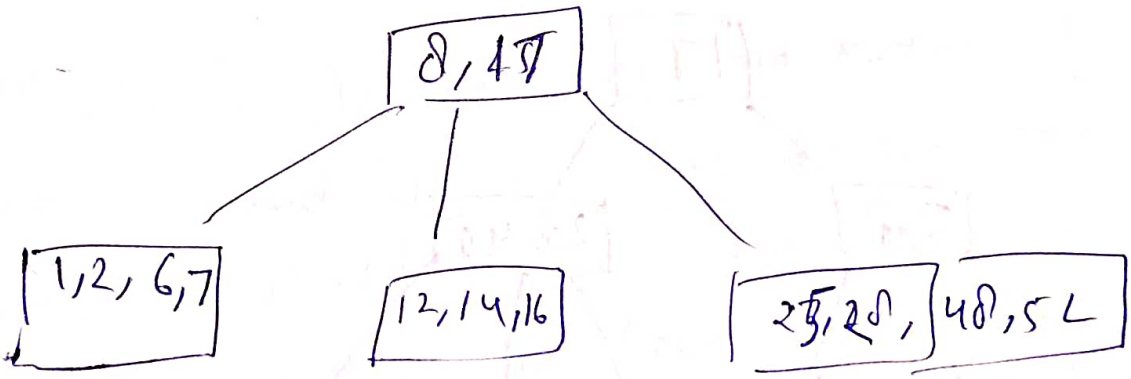
ans:-



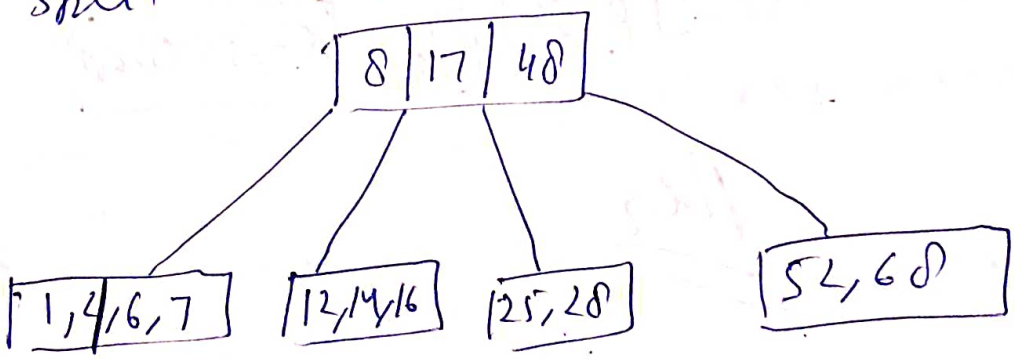
25 split



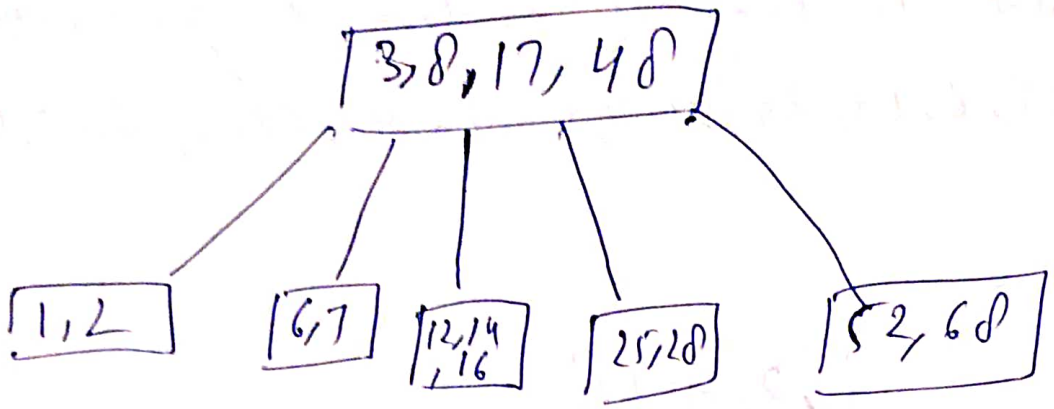
17 split

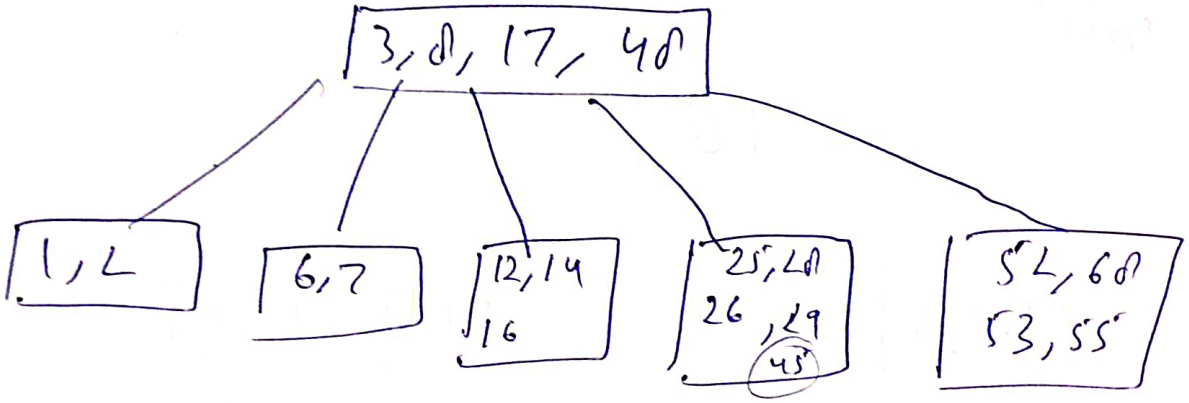


68 split

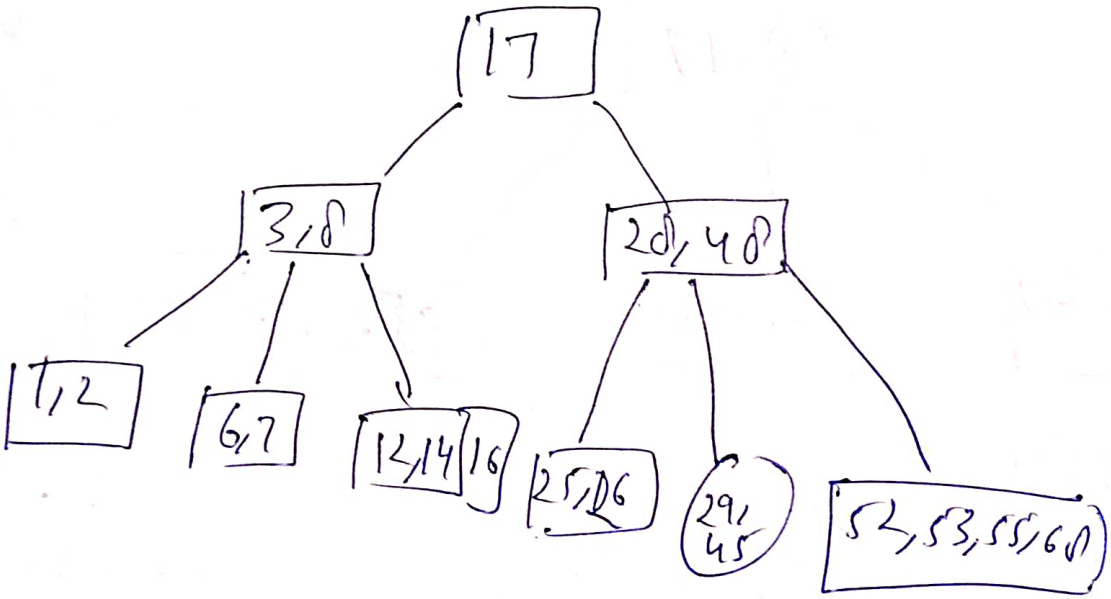


3- split


3, 8, 17, 48
1, 2
6, 7
12, 14, 16
25, 28
52, 60


3, 8, 17, 48
1, 2
6, 7
12, 14
16
25, 28
26, 29
45
52, 60
53, 55

45 split


17
3, 8
28, 48
1, 2
6, 7
12, 14, 16
25, 26
29, 45
52, 53, 55, 60

Deletion in B- Tree:

Case-1(A): If node is leaf and has more than the minimum number of keys then it can be directly deleted.

Case-1(B): If leaf node does not have extra key. If sibling node 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-1(C):- If leaf has no extra keys, ~~not~~ ~~the second~~ and the siblings ~~to~~ the immediate left or right of leaf has no extra key. In such a case the leaf has to be combined with one of these two siblings. This include moving down the parent's key ~~and~~ that way between those of two leaves.

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

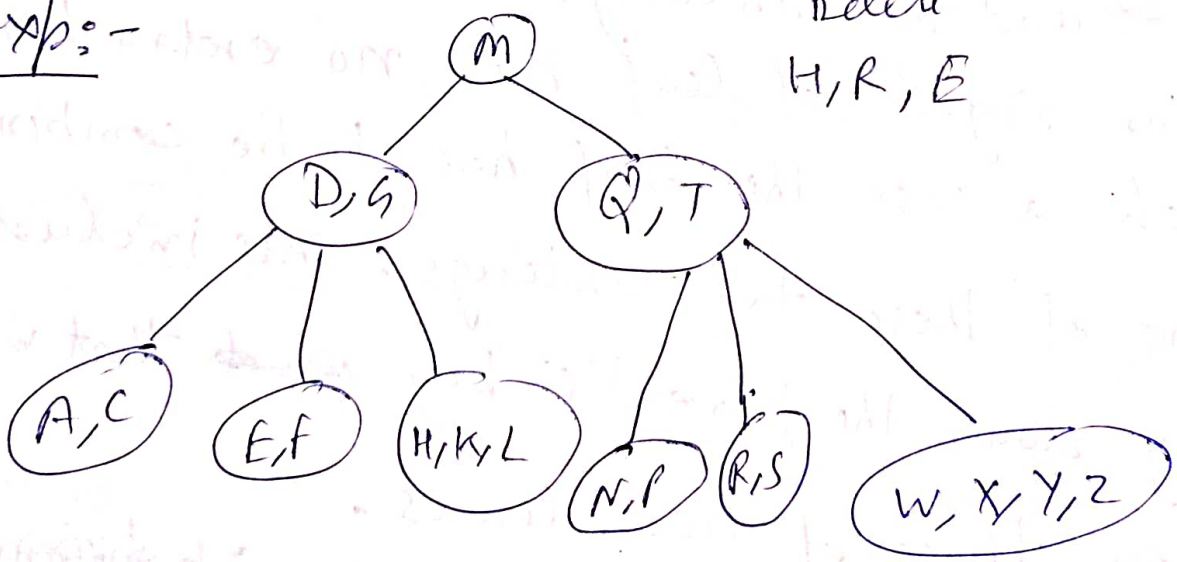
Case - 2 - If deleted key in non leaf node.

Case (2A) :- If node is not leaf then find its successor

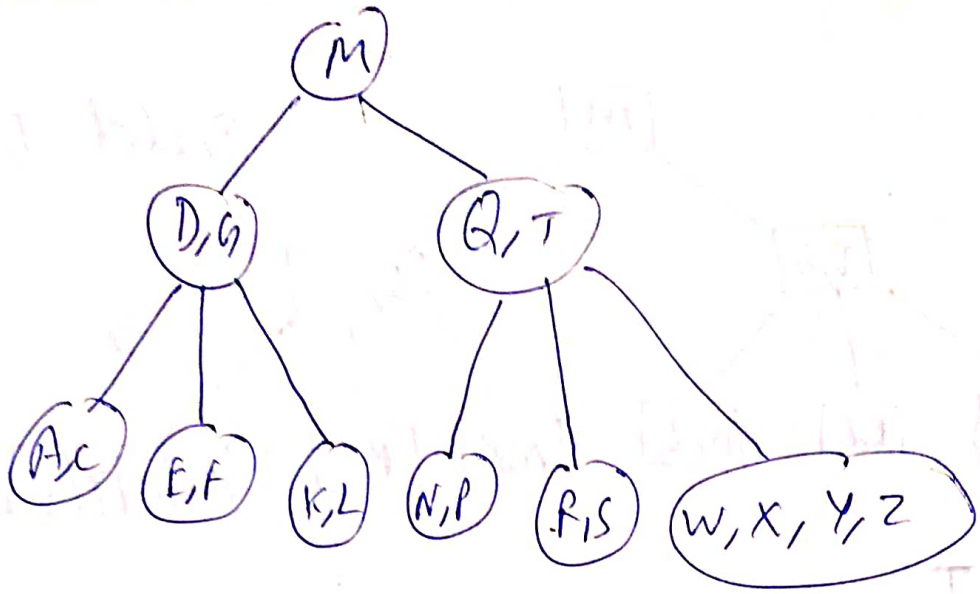
Case 2 :- If node is non leaf then key will be deleted & its predecessor or successor key will come on its place. Suppose both nodes of predecessor & successor have minimum number of keys then nodes of predecessor & successor keys will be combined.

Exp :-

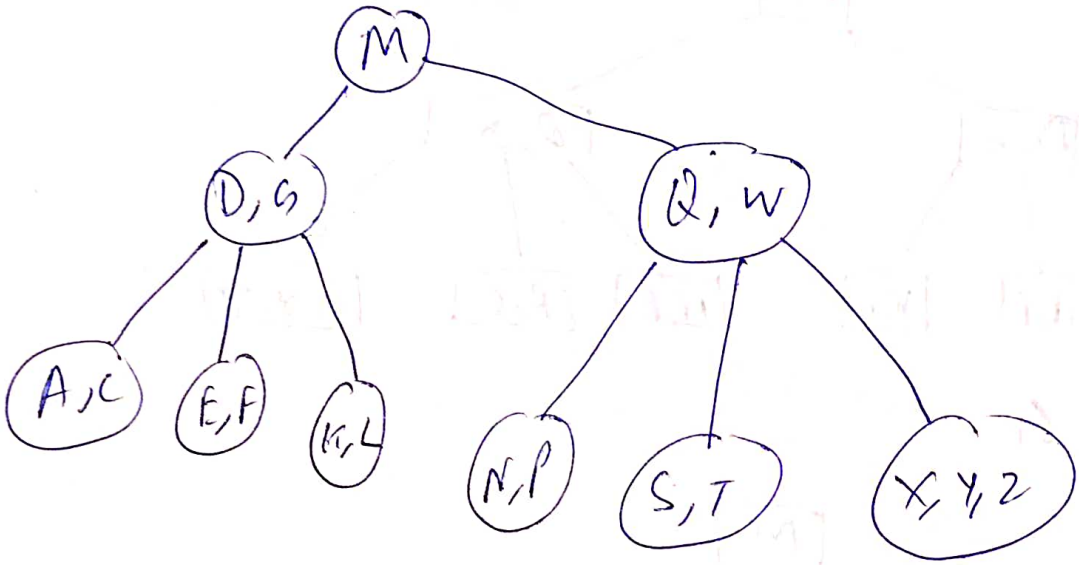
Delete
H, R, E



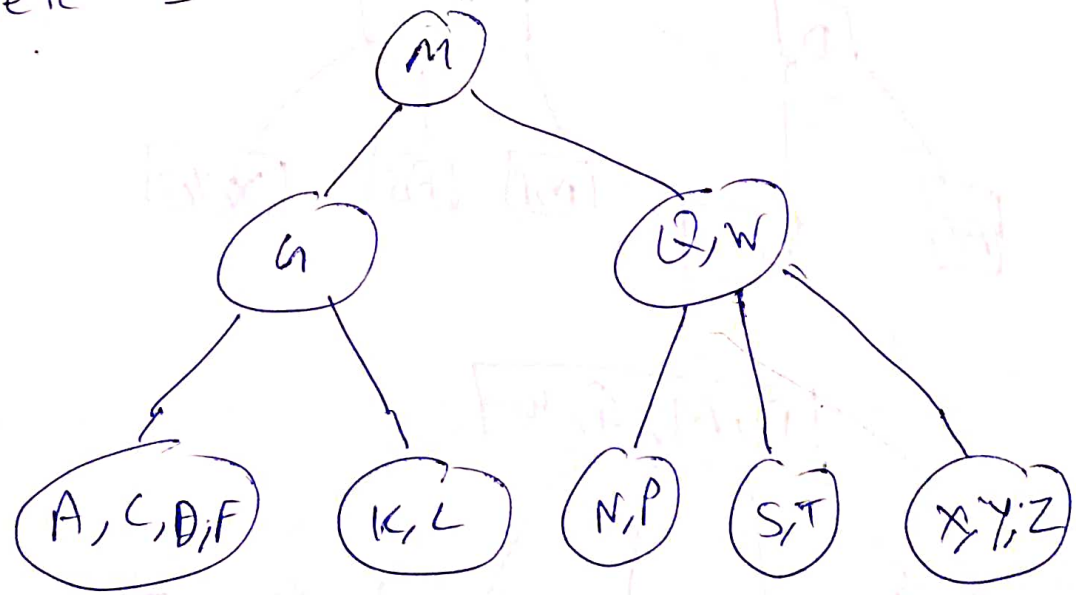
(1) Delete H (case - 1A)



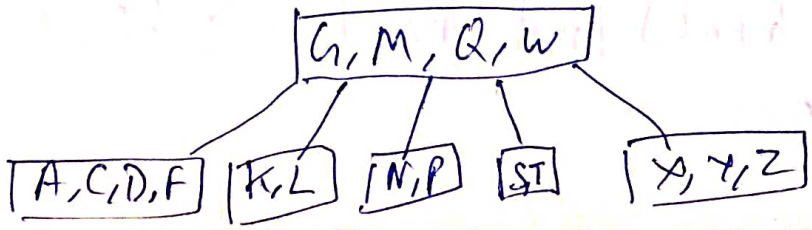
Delete R (Case-1B)



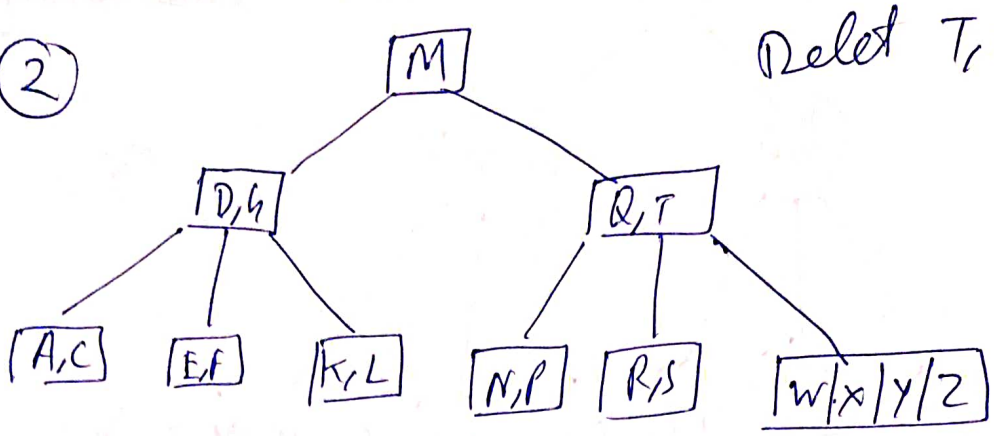
Delete E (Case-1C)



=>

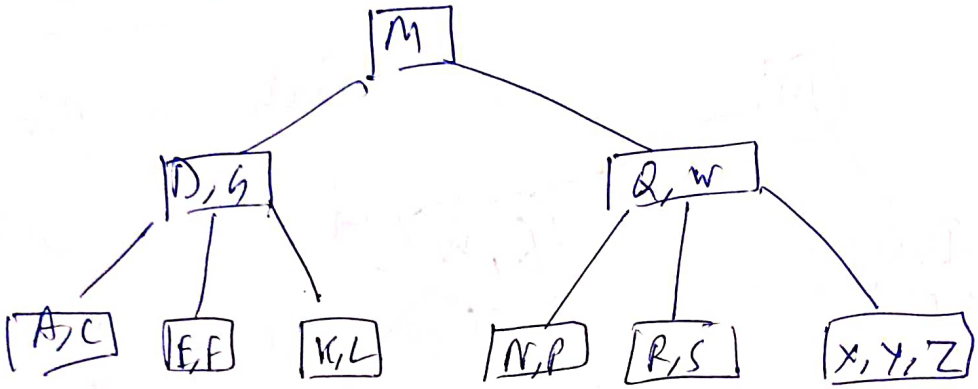


Exp:- (2)

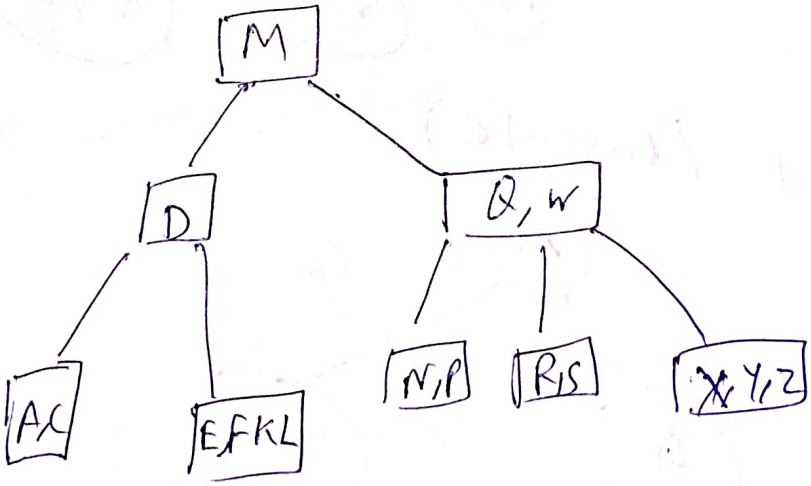


Delet T, G

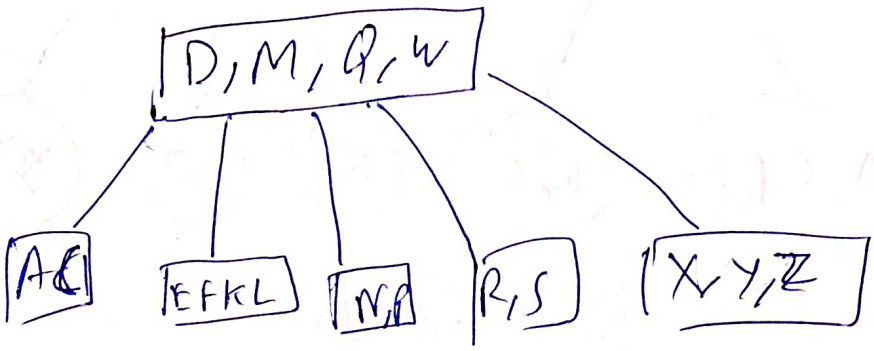
Delete T



Delete G

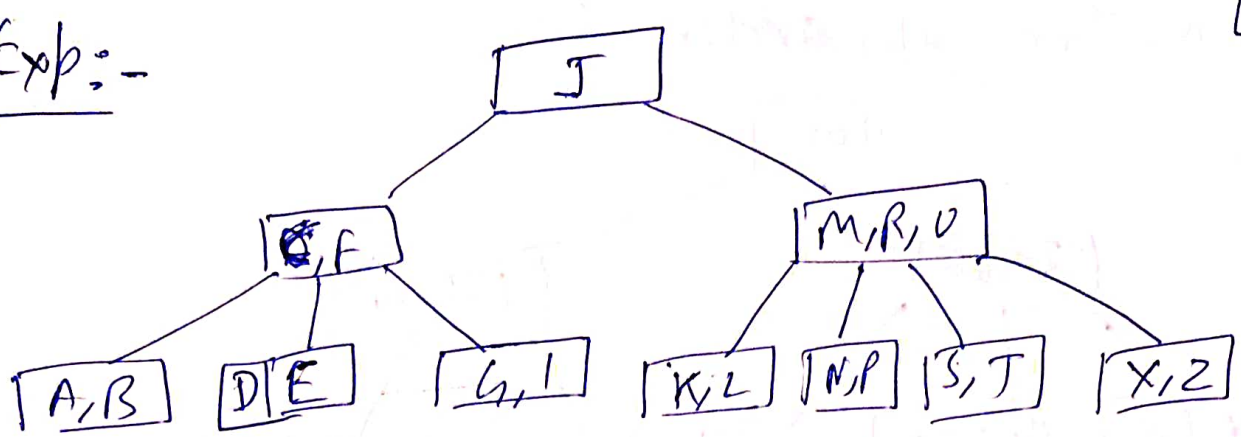


=>

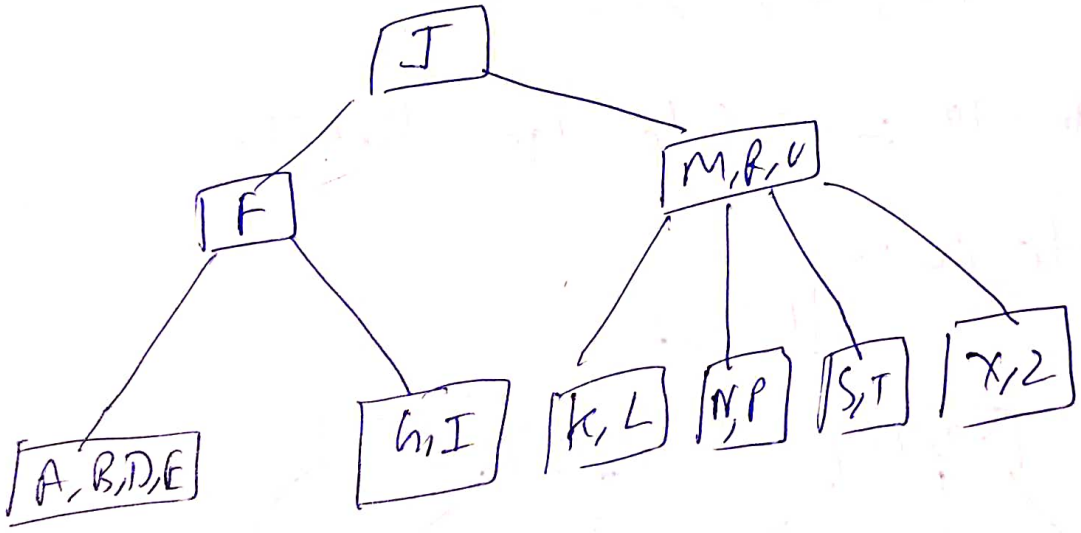


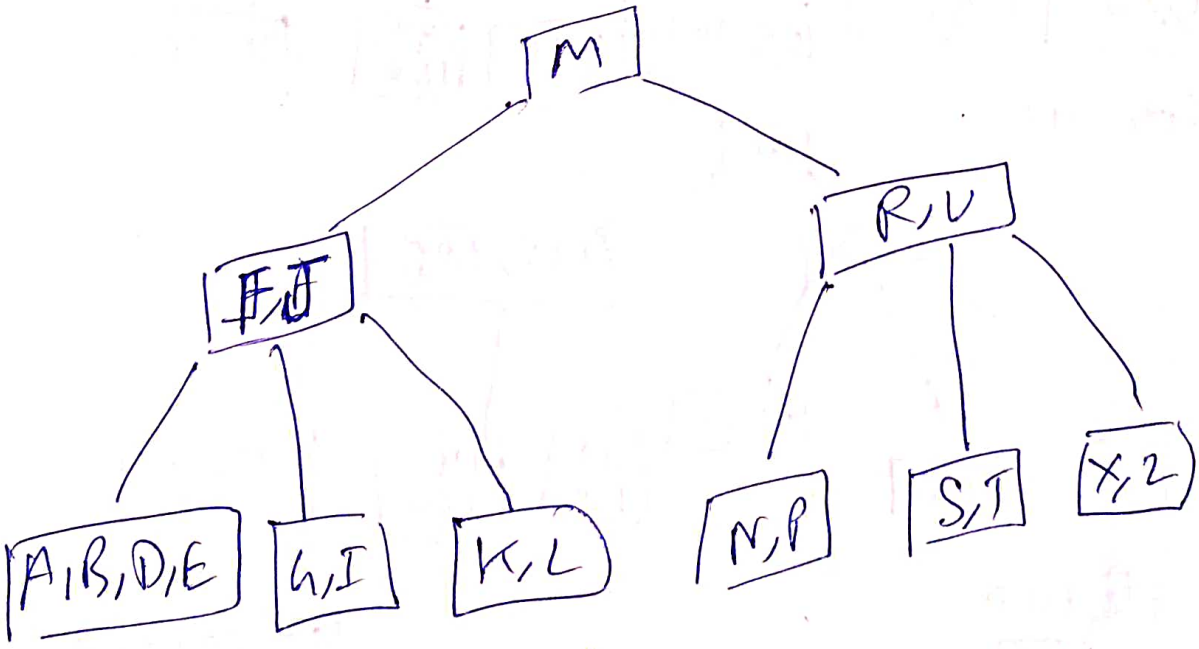
Ans

Exp:-

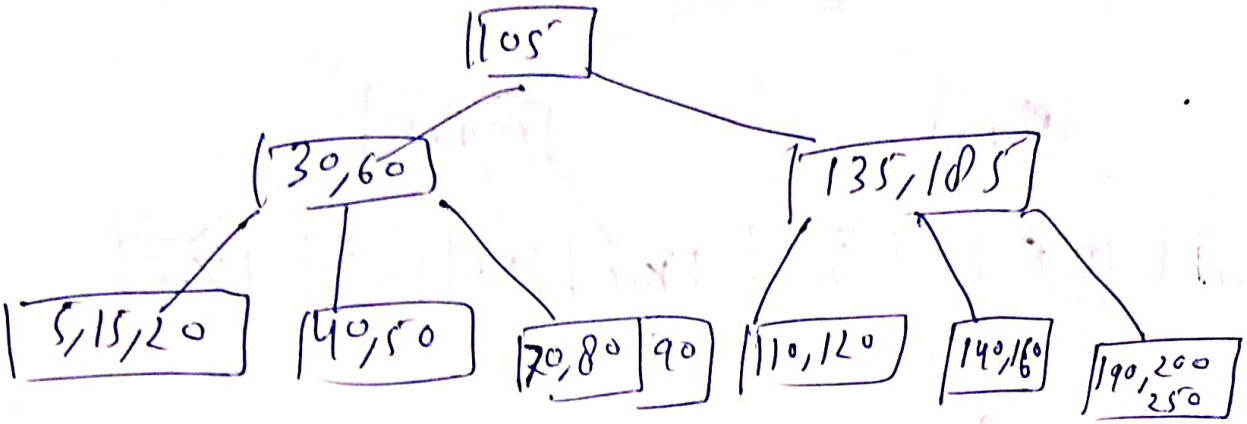

J
C,F
M,R,U
A,B
D,E
G,I
K,L
N,P
S,T
X,Z

Delete C


J
F
M,R,U
A,B,D,E
G,I
K,L
N,P
S,T
X,Z


M
F,J
R,U
A,B,D,E
G,I
K,L
N,P
S,T
X,Z

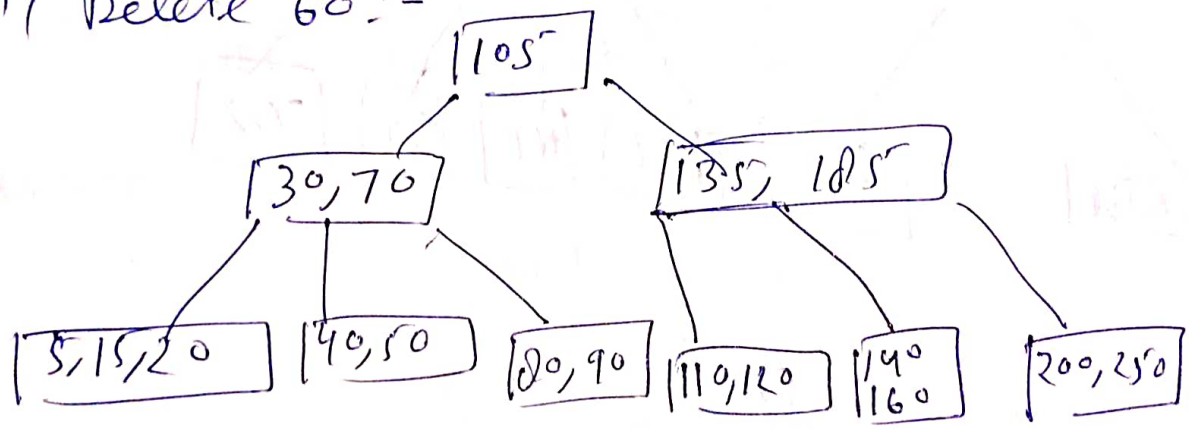
Exp:- B-Tree of order 5.



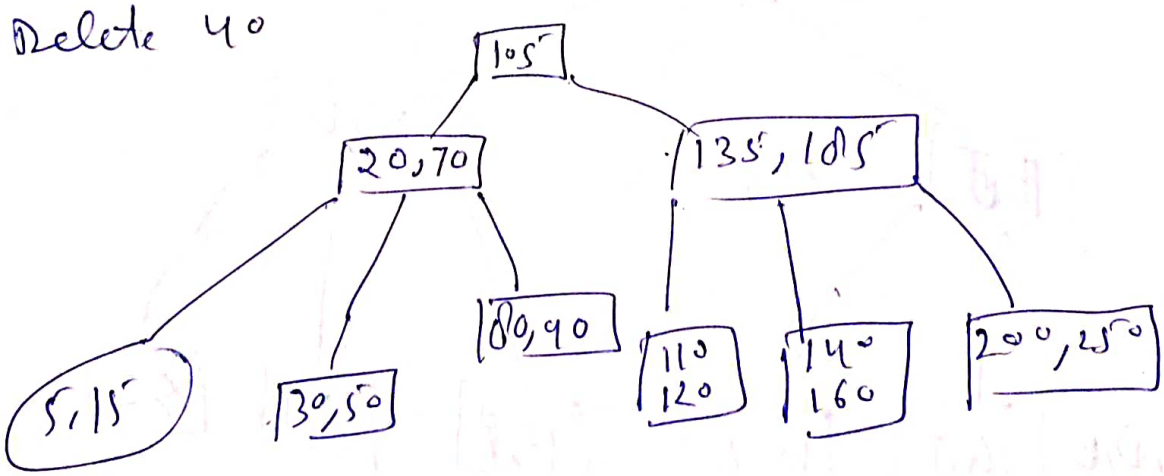
Delete 190, 60, 40, 140

(i) Delete 190 :- Delete 190 direct.

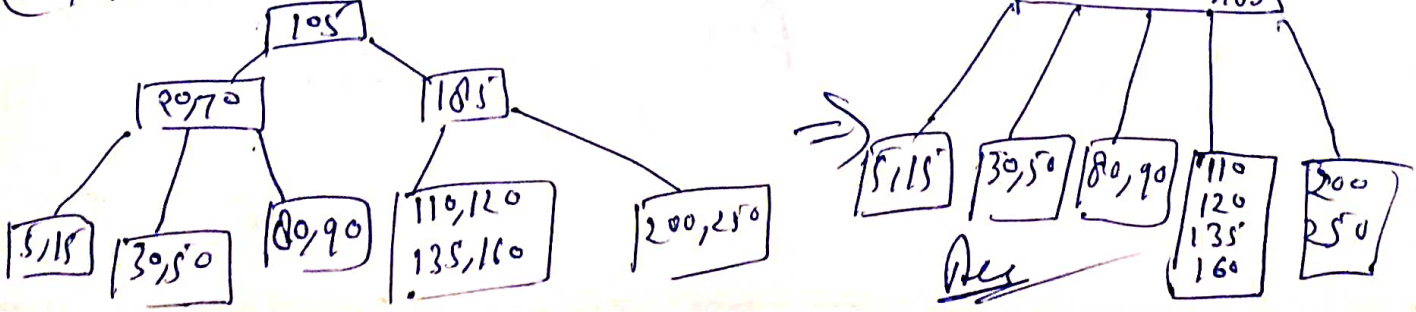
(ii) Delete 60:-



(iii) Delete 40



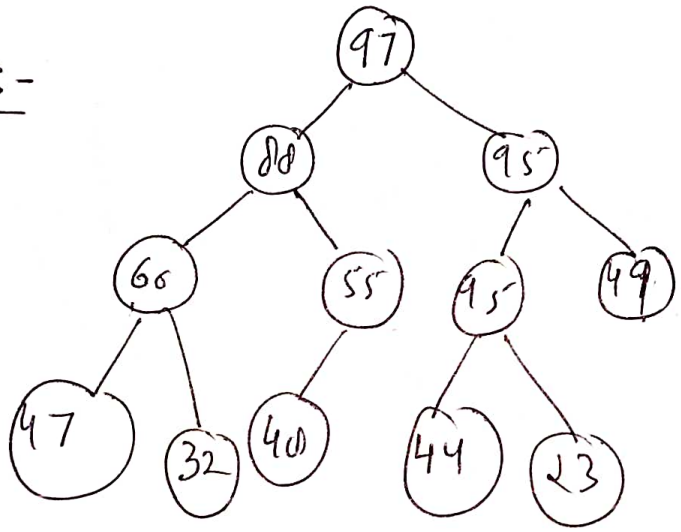
(iv) Delete 140



Heap tree: Suppose H is a complete tree with n elements. Then H is called a heap, or maxheap, 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 minheap if value at N is less than or equal to the value at any of children of N.

Exp:-



Inserting into a Heap:

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

① First adjoint ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.

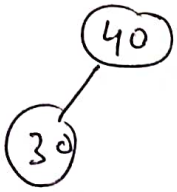
② Then let ITEM rise to its "appropriate place" in H so that H is finally a Heap.

Exp:- build a Heap from

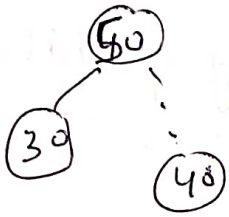
40, 30, 50, 22, 60, 55, 75, 55

40

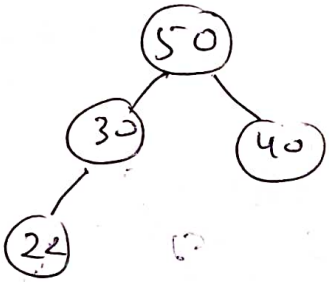
30



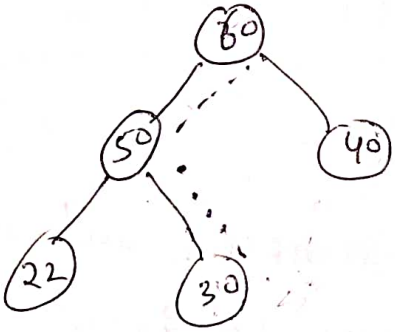
50



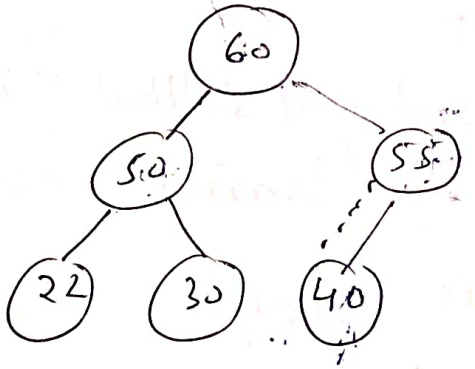
22



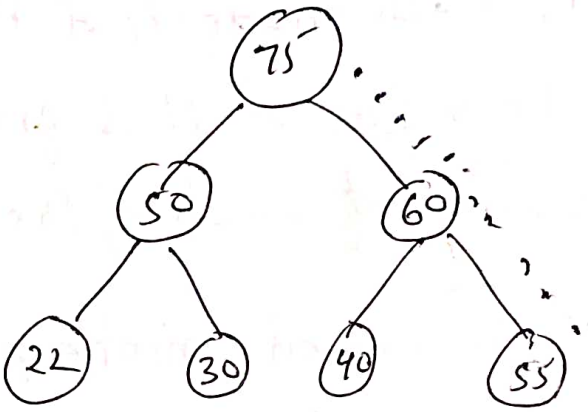
60



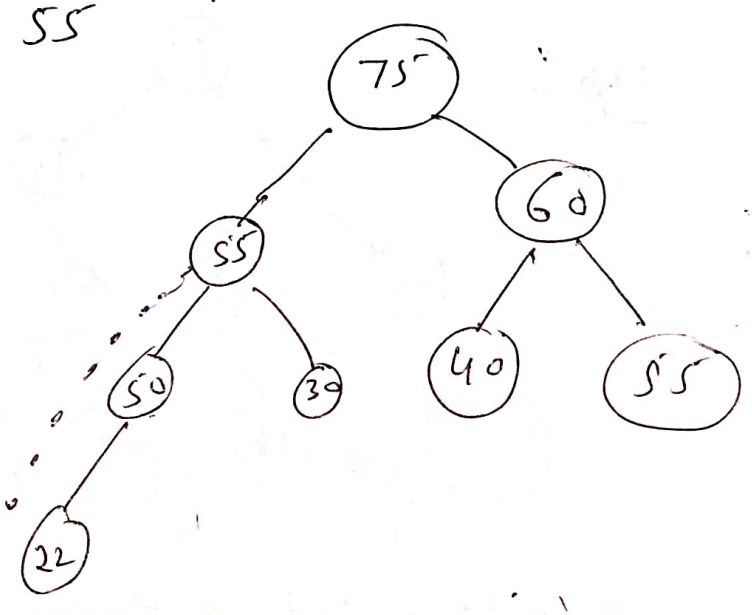
55



75



55



Array — number of node in tree — location where item will insert — Parent of ITEM

Insert Heap (TREE, N, ITEM, PTR, PAR) 28

(1) Set N := N+1 and PTR = N.
(2) Repeat step 3 to 6 while PTR > 1.
(3) set PAR = ⌊PTR/2⌋
(4) IF ITEM ≤ TREE[PAR] then
    Set TREE[PTR] = ITEM and Return.
(5) Set TREE[PTR] = TREE[PAR]
(6) PTR = PAR
(7) Set TREE[1] = ITEM
(8) Exit

Delete Heap (TREE, N, ITEM, PTR, LEFT, RIGHT):-

This procedures assigns the root TREE[1] of H to the variable ITEM and then reheaps the remaining elements. The variable LAST saves 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 sinks in the tree.

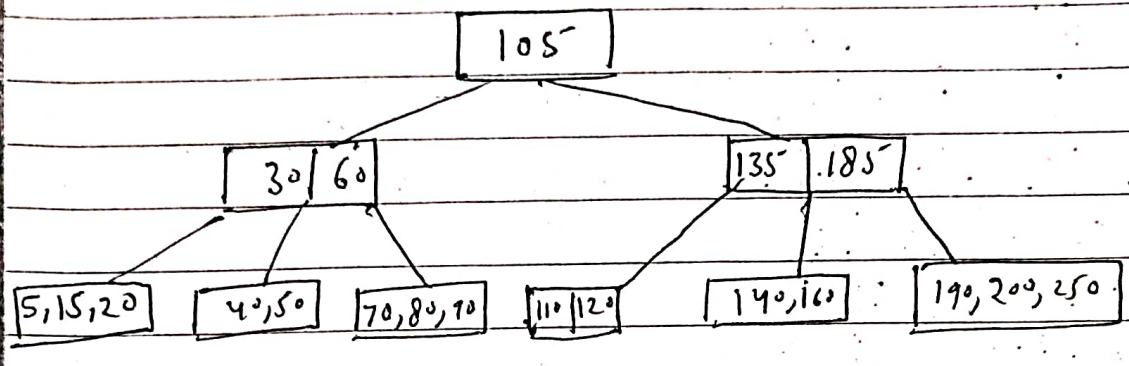
(1) Set ITEM = TREE[1].
(2) Set LAST = TREE[N] and N = N-1.
(3) Set PTR=1, LEFT=2, RIGHT=3
(4) Repeat steps 5 to 7 while RIGHT ≤ N.
(5) If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT] then
set TREE[PTR] = LAST and Return.
(6) If TREE[RIGHT] ≤ TREE[LEFT] then ~~and PTR=LEFT~~
Else set TREE[PTR] = TREE[LEFT] 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 PTR~~
then ~~set PTR=LEFT~~ set TREE(PTR)=TREE[LEFT] & PTR=LEFT.
(9) Set TREE[PTR] = LAST
(10) Return.

Heapsort (A, N) :-

(1) Repeat for ~~N=0~~ to N-1
Call InsertHeap (A, I, ~~N, DATA,~~ ITEM, PTR, PAR);
(2) Repeat while N >= 1
(a) Call DeleteHeap (A, N, ITEM, PTR, LEFT, RIGHT)
(b) Set A[N+1] = ITEM
(3) Exit.

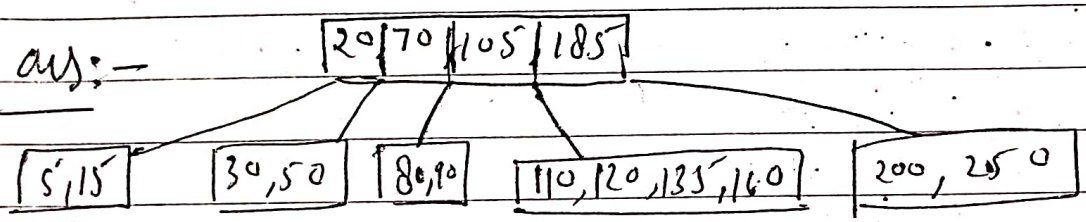
Unit-5

Q) B-tree of order 5 (Deletion)

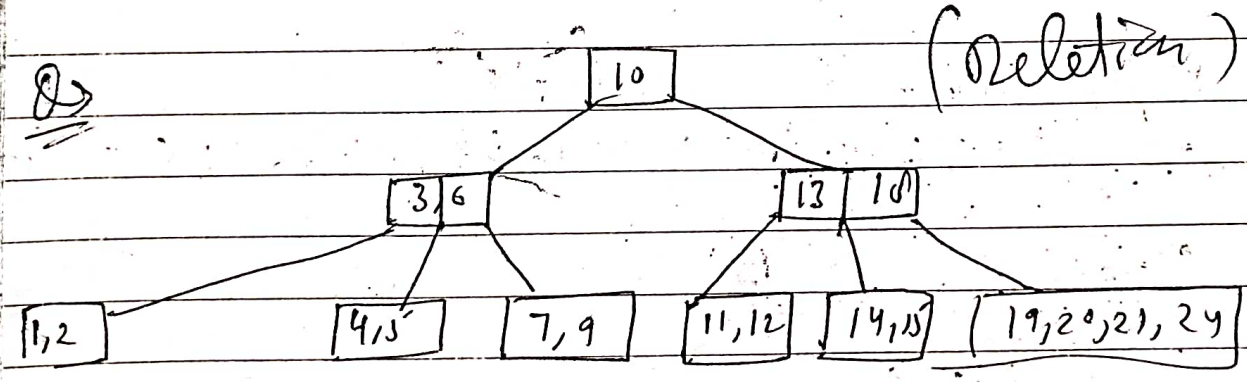


Delete :- 190, 60, 40, 140

ans :-



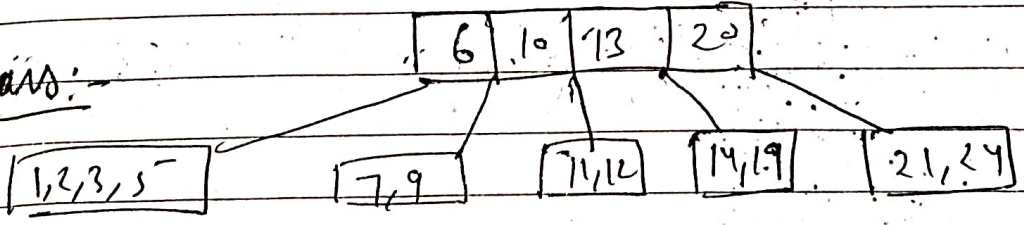
Q)



(Deletion)

Delete 8, 18, 11, 4

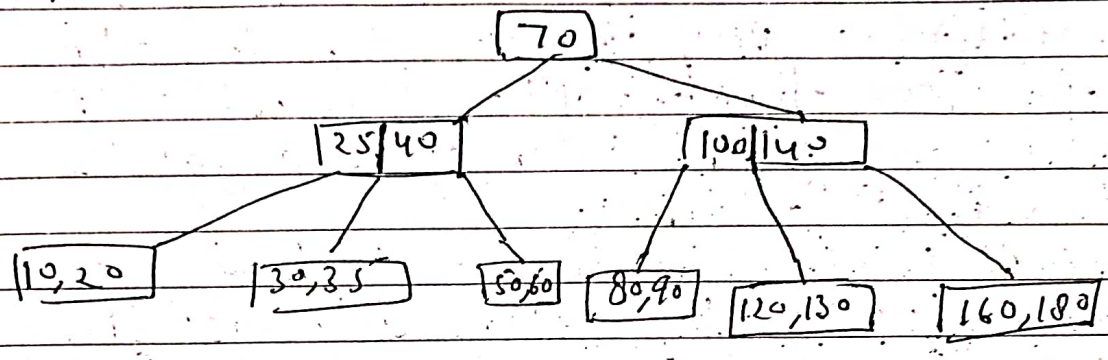
ans :-



order 5.

Q) 10, 20, 50, 60, 40, 80, 100, 70, 130, 90, 30, 120, 140, 25, 35, 160, 180 — Create B Tree = order (5).

Ans:-



Analysis of B-Tree :

The maximum number of items in a B-tree of order m and height h:

Level	no of items
root	$m-1$
level 1	$m(m-1)$
level 2	$m^2(m-1)$
level 3	$m^3(m-1)$
⋮	
level h	$m^h(m-1)$

So to number of items =

$$= (1 + m + m^2 + \dots + m^h)(m-1)$$

$$= \frac{(m^{h+1} - 1)}{m-1} \cdot (m-1)$$

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

Eg: if $m = 5$, $h = 2$

$$5^{2+1} - 1 = 124$$

(multi)

M-Way Search Tree:

Need?

> In database prog^r, the data is too large to fit in memory, so, it is stored on secondary storage (Disk)
> Disk access is very expensive, the disk I/O operation takes milliseconds while CPU processes data on the order of nanoseconds, one million time faster.
> AVL, Red Black, Binary Search tree etc have good performance if the entire data can fit in the main-memory But these trees are not optimised for external storage and require many disk accesses, this gives very poor performance for very large data.

> To reduce disk accesses
- Reduce tree hight by increasing the number of children of a node.

> To achieve Above gaal we use Multiway search tree.

Introduction:-

- M-way search tree is generalised version of binary search tree.
- An m-way search tree of height h calls for O(h) number of accesses for an insert/delete/retrieval operation.
- Number of elements in m-way search tree of height h ranges from a minimum of h to a maximum of $m^h - 1$.

to fit in
storage (Disk)
e disk
ile CPU
conds., one

e good
n fit
are not
d require
ry poor

number

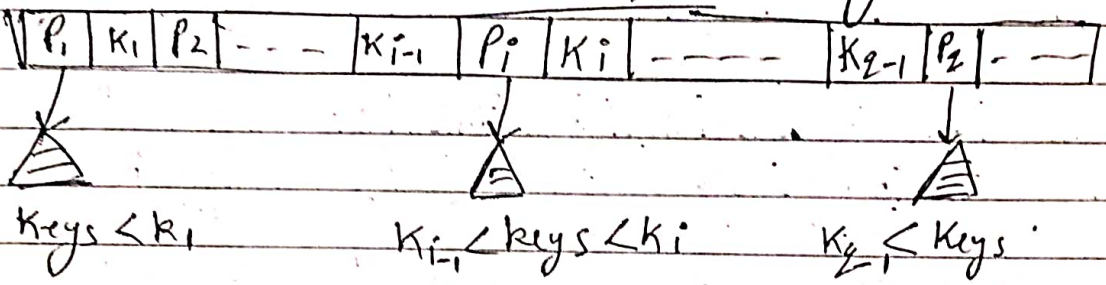
ltiway

ersion of

alls for
/delete/retrieval

h tree of
um of

- Trees having (m-1) keys and m-children are called m-way search tree.
- A binary search tree is two way search tree of order 2.
- In an m-way tree all the nodes have degree $\leq m$
- Each node has the following structure-



Ki : are the keys where $1 \leq i \leq q-1$, $q \leq m$

Pi are pointers to subtree, where $1 \leq i \leq q$, $q \leq m$

- The keys in each node in ascending order $K_1 \leq K_2 \leq$
- The key Ki is larger than keys in subtree pointed by Pi and smaller than keys in subtree pointed by P_{i+1}.
- Sub tree are the m-way tree.

- Full nodes are those nodes having m-1 keys.
- Semi-leaf: A node with atleast one empty subtree is called semi-leaf
- In top down m-way search tree, a semi-leaf must be either full or leaf

Searching a multiway tree:

For searching a key k in m-way search tree is similar to binary search tree. The procedure is as follows:

(1) If the key is less than K_1 and K_2 then the search is continued in m-way search tree pointed by pointer P_1.

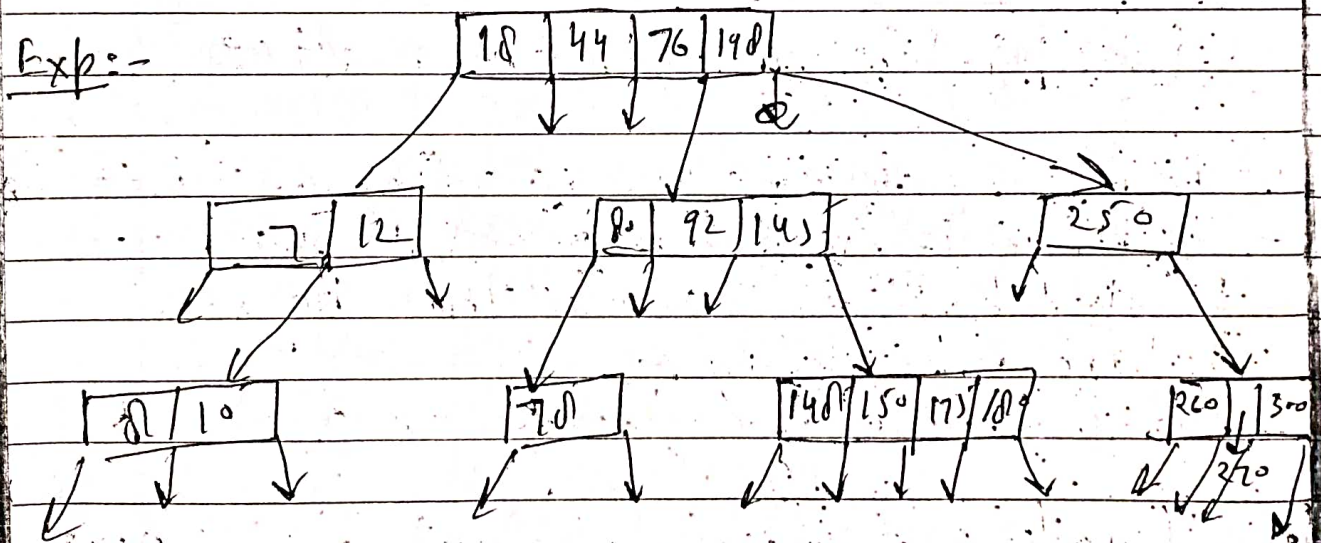
(2) If the key lies bet^n K_1 and K_2 then the search is continued in m-way search tree pointed by P_2.

(3) If the key lies bet^n K_2 and K_3 then the search is continued in m-way search tree pointed by P_3.

:

(4) If the key is greater than K_{q-1} then search is continued in m-way search tree pointed by P_q.

Exp:-

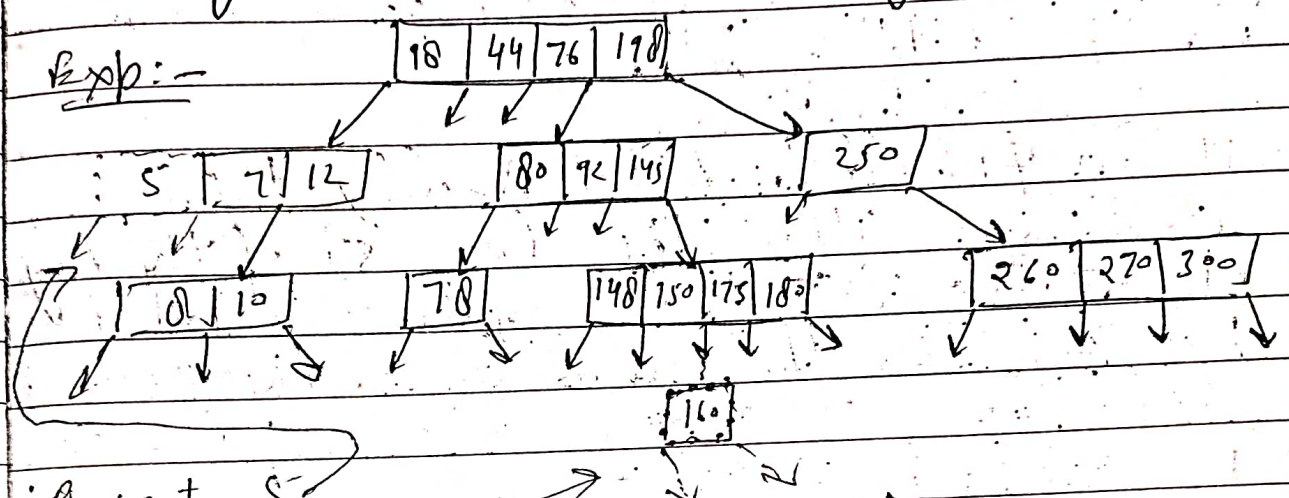


Search 78 :- we begin with root node & 78 > 76 > 44 > 18 but < 198 so we move to the 4th subtree of root node. Since 78 is available so search is successful.

Insertion in m-way search tree:

In insertion we proceed the same way as searching.

Exp:-

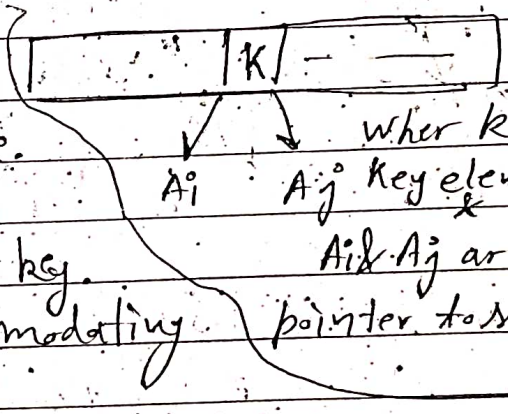


i) Insert 5

ii) Insert 160

Deletion in m-way search tree:

Let K be a key to be deleted from the m-way search tree. To delete the key we proceed as one would to search for key. Suppose the node accommodating the key as follows

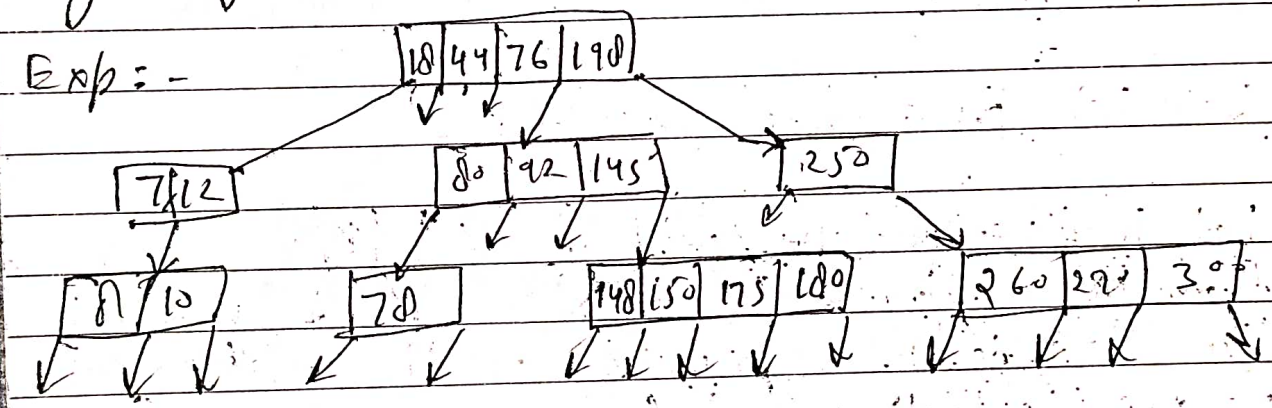


- If (Ai = Aj = Null) then delete K.
- If (Ai ≠ Null & Aj = Null) then choose the largest of the key elements K' in the child node pointed by Ai delete the key K' & replace K by K'. Obviously deletion of K' may call for subsequent replacement & therefore deletion in a similar manner, to enable the key K' move up the tree.

- if (A_i = Null & $A_j \neq$ Null) then choose the smallest of the key elements K'' from the subtree pointed by A_j. delete K'' & replace K by K''.

- if ($A_i \neq$ Null & $A_j \neq$ Null) then choose either the largest of the key elements K' in the subtree pointed by A_j Or the smallest of the key elements K'' in the subtree pointed by A_j to replace K.

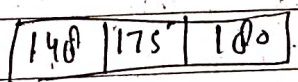
Exp:-



(i) Delete 150

ans:- A_i = Null & $A_j \neq$ Null

delete 150 Directly.

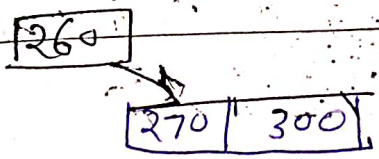


(ii) Delete 250

A_i = Null ; $A_j \neq$ Null

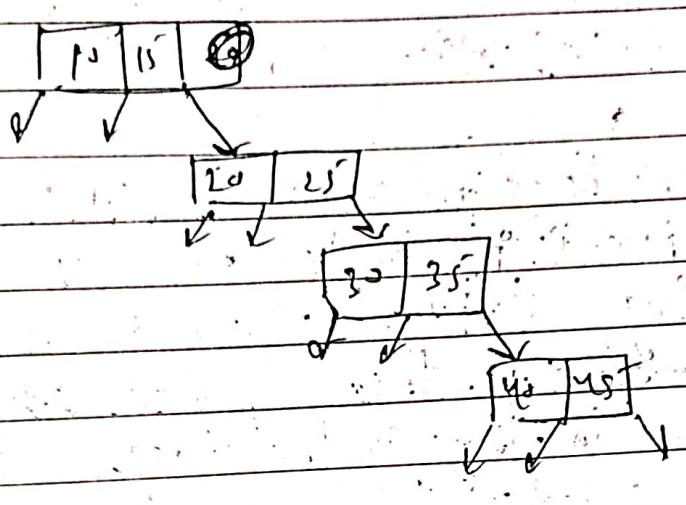
smallest of subtree pointed by A_j = 260

Delete 260 & Replace 250 with 260

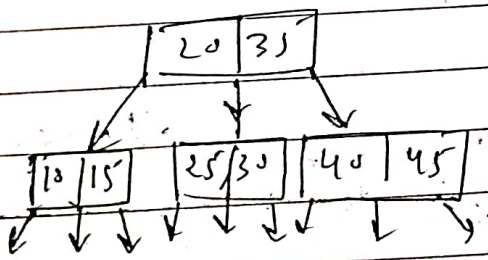


Q) Construct 3-way search tree order is given —

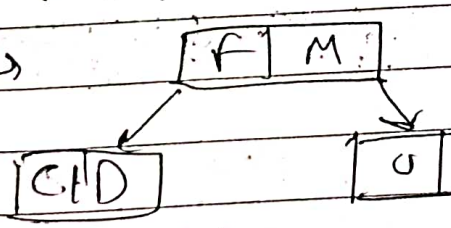
List A: 10, 15, 20, 25, 30, 35, 40, 45

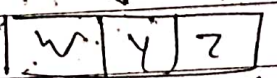


(or) List B: 20, 35, 40, 10, 15, 25, 30, 45

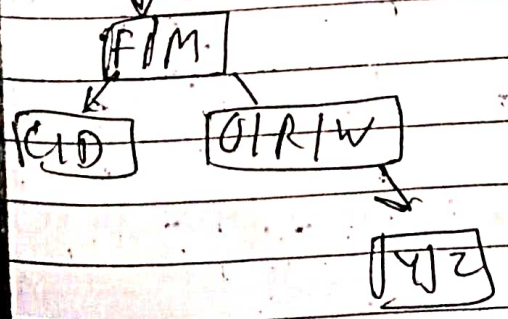


Q)

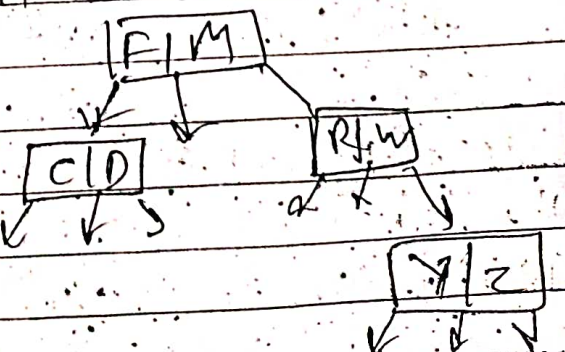




(i) Delete V



(ii) Delete O



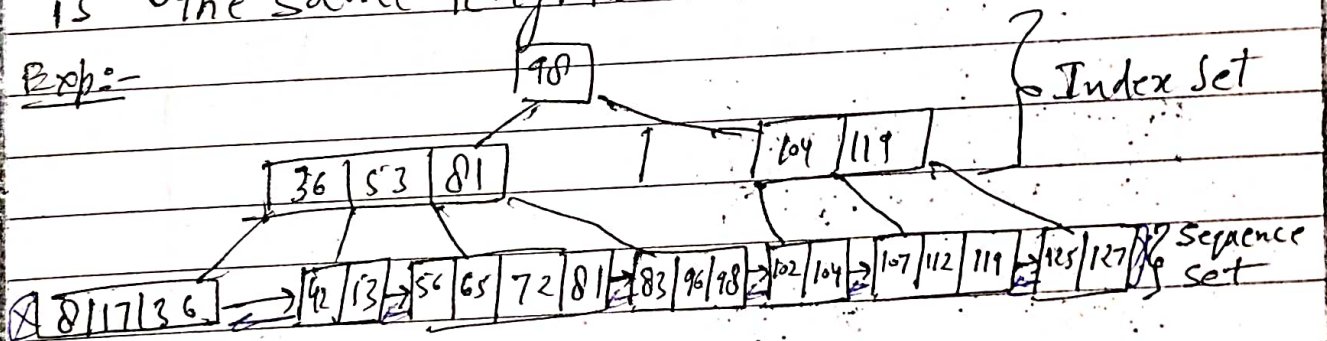
B^+ - tree:

One of the major drawbacks of the B-tree is the difficulty of traversing the keys sequentially. B^+ tree retain the rapid random access property of the B-tree, while also allowing rapid sequential access.

On the B^+ tree, all keys are maintained in leaves and keys are replicated in non-leaf nodes to define paths for locating individual records. The leaves are linked together to provide a sequential path for traversing the keys in the tree.

The B^+ tree is called a balanced tree because every path from the root node to a leaf node is the same length.

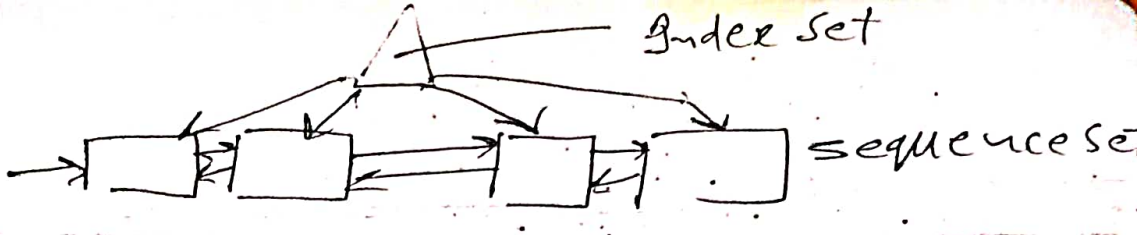
Exp:-



B^+ - tree Structure:

B^+-tree consists of two parts:

1. Index set
   - Provides indexes for fast access of data.
   - consist of internal nodes that store only key & subtree pointer
2. Sequence set
   - Provides efficient sequential access of data (using doubly linked list)
   - consists of leaf nodes that contain data pointer



B^+-tree Index Node Structure:-

The basic structure of the B^+-tree index node of m is same as that of B-tree node of order m.

- The root has atleast two subtree unless it is a leaf.
- Each non-root index node holds $q-1$ key and q subtree pointers, where $\lceil m/2 \rceil \leq q \leq m$
- only difference is that index node not contain data pointer $[P_1 | K_1 | P_2 | \cdots | K_{i-1} | P_i | K_i | \cdots | K_{q-1} | P_q]$

$K_{i-1} < \text{keys} \leq K_i$ $K_{q-1} < \text{keys} \leq n-1$

$\text{keys} < K_1$

B-tree sequence Node structure:

$[K_1 | D_1 | \cdots | K_{i-1} | D_{i-1} | K_i | D_i | \cdots | K_{q-1} | D_{q-1} | \rightarrow]$

- Data Pointer
- Data Pointers
- Pointers to previous & next leaf node in tree

Searching in B^+-tree:-

- Start from the root.
- If internal node is reached.
  - Search key among the keys in that node
    - ↳ Linear search or binary search
  - If key ≤ smallest key, follow the leftmost child pointer down
  - If key > largest ——— rightmost ——— pointer down.
  - If $K_i \leq key < K_j$ follow the child pointer between K_i & K_j

- If a leaf is reached
  - search key among the keys stored in that leaf
    - Linear search or binary search
  - If found return the corresponding record otherwise record not found.

Insertion into a B+ tree:

Steps are as follows:

① The search operation is used to find the leaf node in which the key value of the node has to be inserted.
② If the key value already exists, no more insertion is needed. Else if the said key value does not exist, insert the value in the leaf node in an ordered fashion.
③ When a node is split, The middle key is retained in the left half node as well as being promoted to the father

Deletion from a B+ tree:—

Steps are as follows:

① Search the B+ tree for the key value.
② If the key value in tree, Remove it as that of B-tree
③ When a key value is deleted from a leaf there is no need to delete that key from index set of the key.

B-tree	B+-tree
Data Pointer stored in all nodes	only in leaf node.
② No redundant key	Redundant key may exist
search can end at any node	ends at leaf node
Slow sequential access	Efficient sequential access