D.S. — 3rd SEM 24
2nd Unit

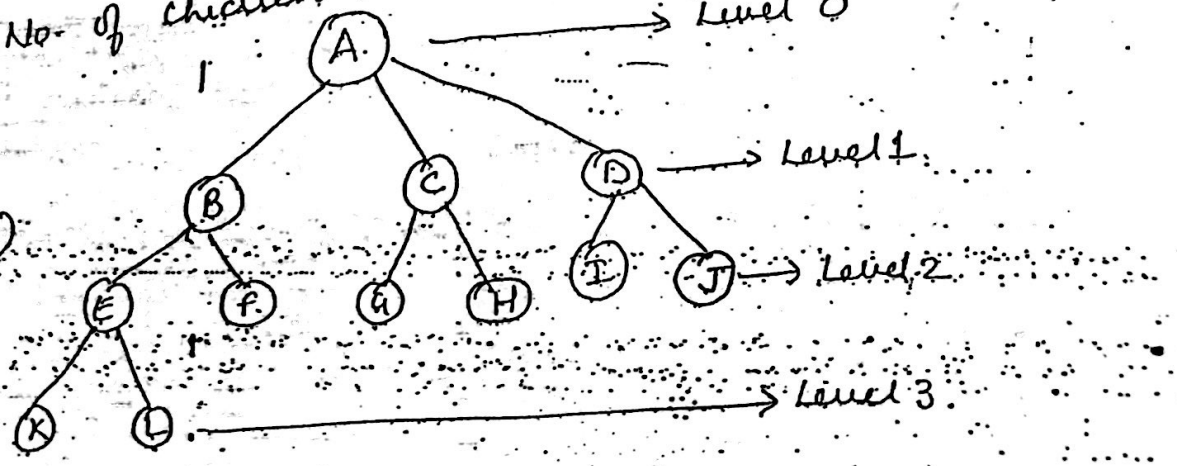
Trees

In C.S, a tree is a widely-used non-linear data structure that emulates a tree structure with a set of linked nodes. Hierarchical Representation of elements. eg: family, table of content etc.

Basic Terminology:-

There are number of terms associated with tree, are

1. Root: It is the first node in the hierarchical data structure tree.
2. Node:- Each data item in a tree is called node. It specifies the data of the node and the links to other data items.
3. Degree of a Node:- It is the number of subtrees of a node in a given tree. (The No. of children the node has)



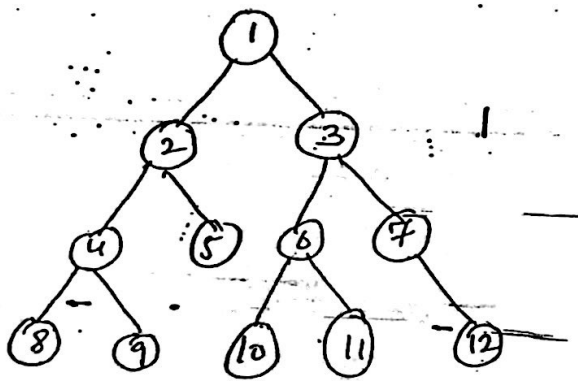
For eg
Degree of A = 3
Degree of B = 2
Degree of C = 1
Degree of D = 2
Degree of F = 0

4. Degree of Tree :- A maximum degree of nodes in a given tree. In the above tree, node A has degree = 3, which is max., so the degree of Tree = 3.

5. Siblings :- If N is a Node in Tree (T) that has left successor S_1 and right successor S_2, then N is called the parent of S_1 and S_2. Correspondingly S_1 and S_2 are called left and right child respectively of N. Also S_1 and S_2 are Siblings.

In other words, all nodes that are of same level and share the same parent are called Siblings.

For Eg:



(2)

So, Nodes 2 and 3, nodes 4 and 5, nodes 6 and 7, nodes 8 and 9, nodes 10 and 11 are Siblings.

6. Level Number :- Every node in tree is assigned a Level Number. The Root Node is defined at level 0. The left and right child of the Root Node has level Number 1.

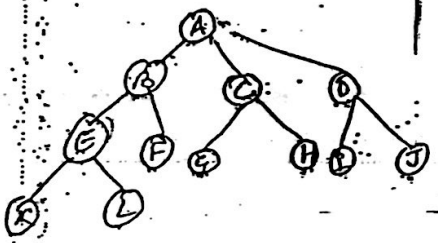
So, Child node's level Number = Parent's level No. + 1 (See Diagram 1)

7. In degree and Out degree :- Only Root Node in a tree has indegree which is equal to zero. Out degree no. of edges leaving that Node.

8. Leaf Node :- A leaf Node has no children also known as Terminal Node. For Eg. 8, 9, 10, 11 and 12.

9. Edge :- Edge is a line which connects one node to another node.

10. Path :- It is a sequence of consecutive edges from source node to the destination node for eg.

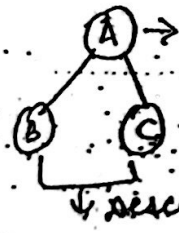


For Eg. The path b/w A to L is (A, B), (B, E) and (E, L).

11. Depth (Height) :- It is a maximum level of any node in a given tree. The length of the path from Root R to the Node (N).

12. Ancestor and Descendant Nodes :- Ancestors of a node are all the nodes along the path from the root to that Node.

Descendants of a node are all the nodes along the path from that node to the leaf Node.

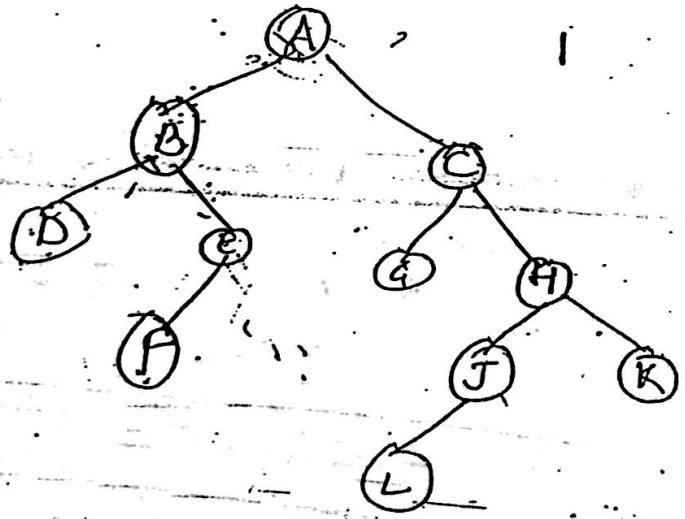


Binary Tree :-

A binary Tree T is defined as a finite set of elements, called Nodes, such that—

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

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

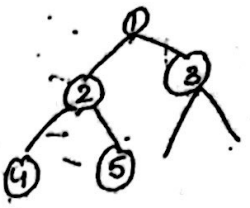


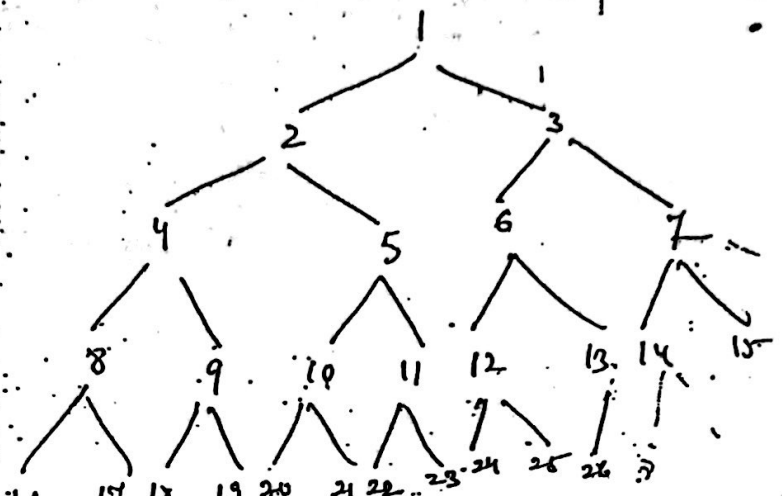
$T_n = 0$

Complete Binary Trees :- (Atmost Two children)

T_n with exactly n nodes

T_{26} with 26 nodes.





Complete Binary Tree.

The depth d_n of the complete tree T_n with n nodes is given by

$$D_n = [\log_2 n + 1]$$

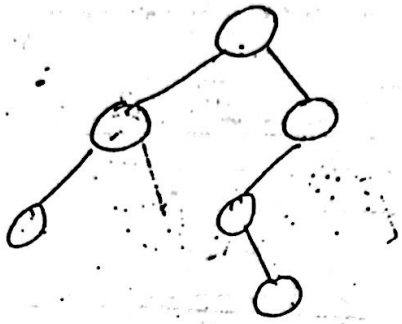
Eg :- If the complete tree T_n has $n = 1000000$ nodes, then its depth $D_n = 21$.

Extended Binary Tree : 2 Trees.
(Used to Represent Algebraic Expressions).

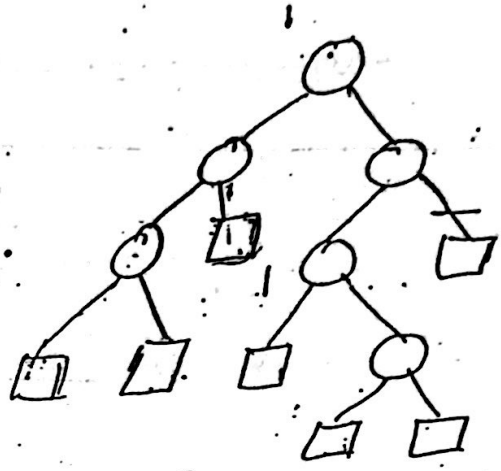
A binary tree T is said to be a 2-tree or an extended Binary Tree if each node N has either 0 or 2 children.

→ The Nodes with 2 children are called internal Nodes (denoted by circles)

→ The Nodes with 0 children are called external Nodes (denoted by squares)



Binary Tree (T).
(a)



Extended 2-Tree
(b)

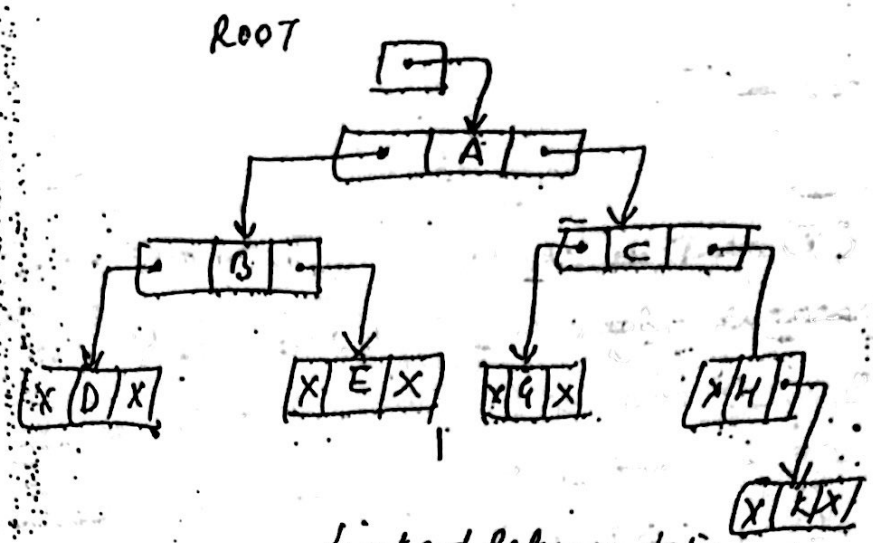
Converting Binary Tree T into 2-Tree.

Representing Binary Trees in Memory :-

→ Linked Representation of Binary Trees :-

Three parallel arrays, INFO, LEFT, RIGHT and pointer variable ROOT as follows :-

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



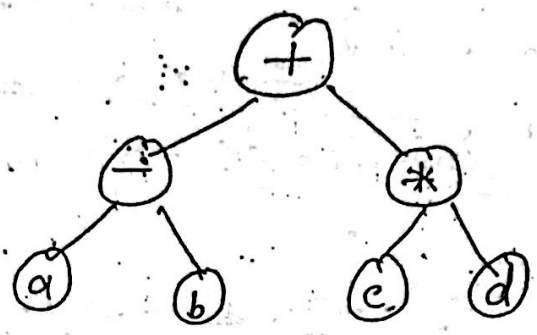
Linked Representation of Tree

Root = 5

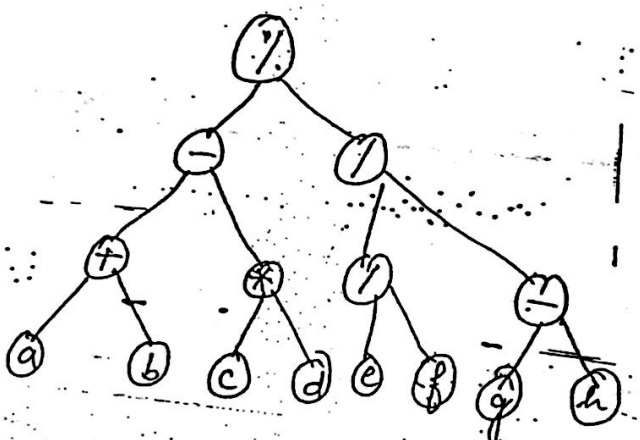
	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4			
5	A	10	2
6	H	8	1
7	D	0	0
8			
9			
10	B	7	13
11			
12			
13	E	0	0

Expression Tree :-

① $(a-b) + (c*d)$



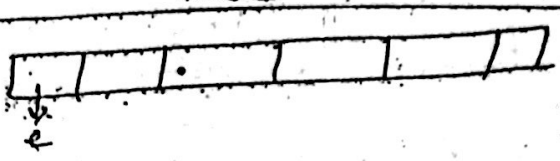
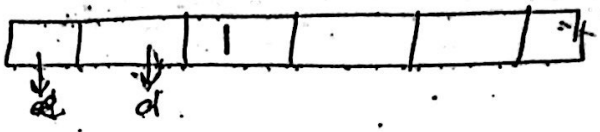
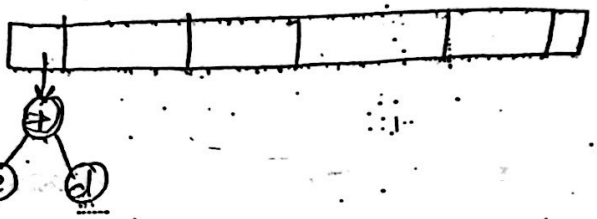
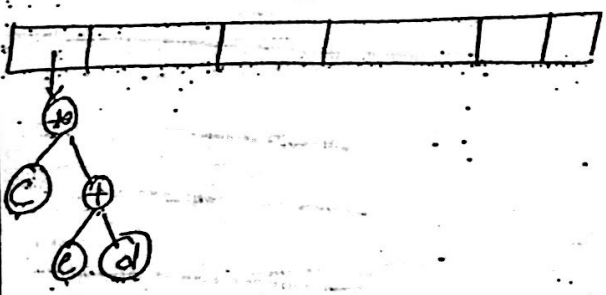
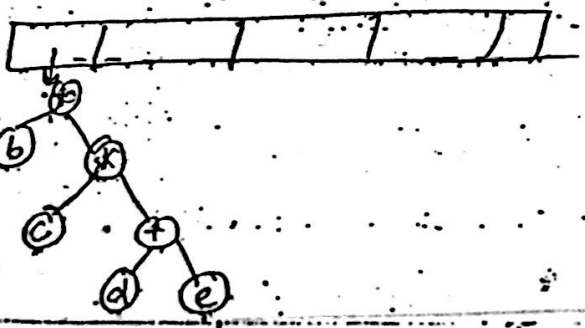
② $(a+b) - (c*d) \% (e \wedge f) / (g-h)$



Construction of an Expression Tree from a Prefix Expression

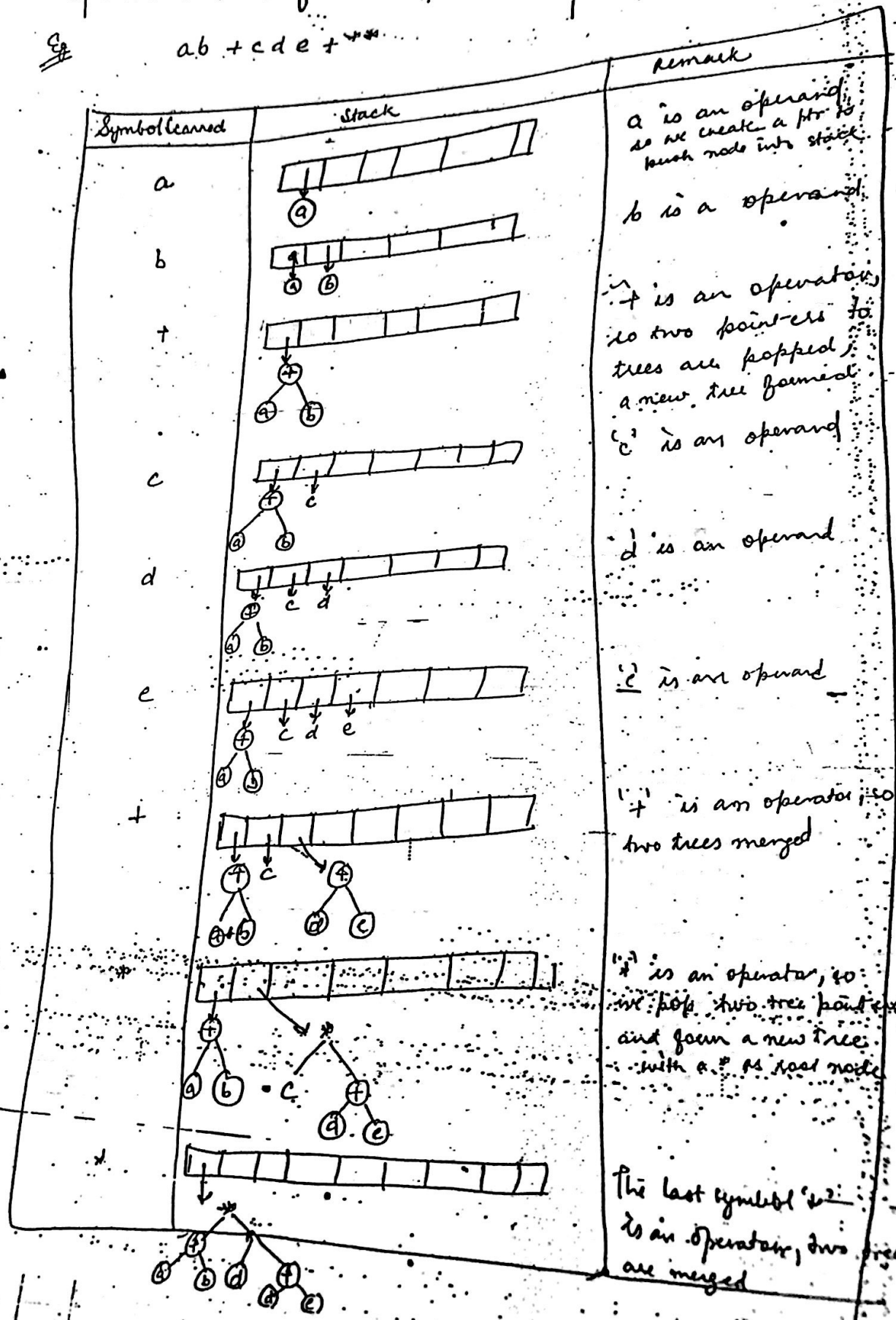
→ * b * c + d / e .

In this scan expression from right to left

Symbol Scanned	Stack	Remark.
e		'e' is an operand
d		
+		
c		
*		
b		
*		

Expression Tree from postfix expression

Eg: ab + cde + * *



Traversing Binary Trees :-

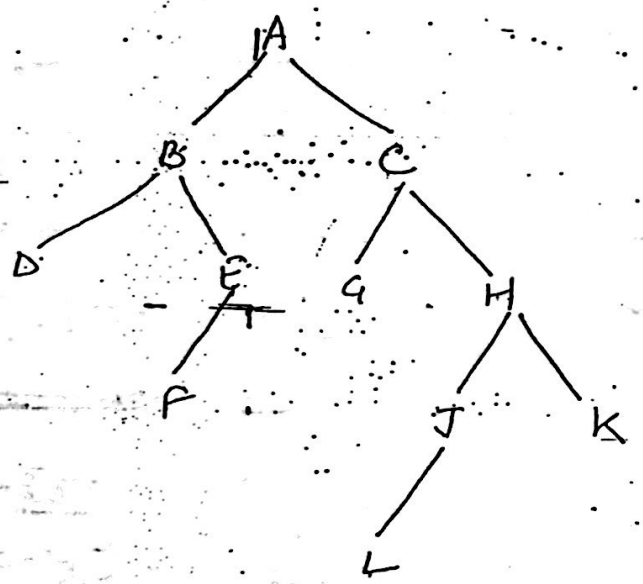
→ Preorder (Root → Left → Right)

1) Process the root R
2) Traverse the left subtree of R in preorder
3) Traverse the right subtree of R in preorder

→ Inorder (Left → Root → Right)

→ Postorder (Left → Right → Root)

Eg

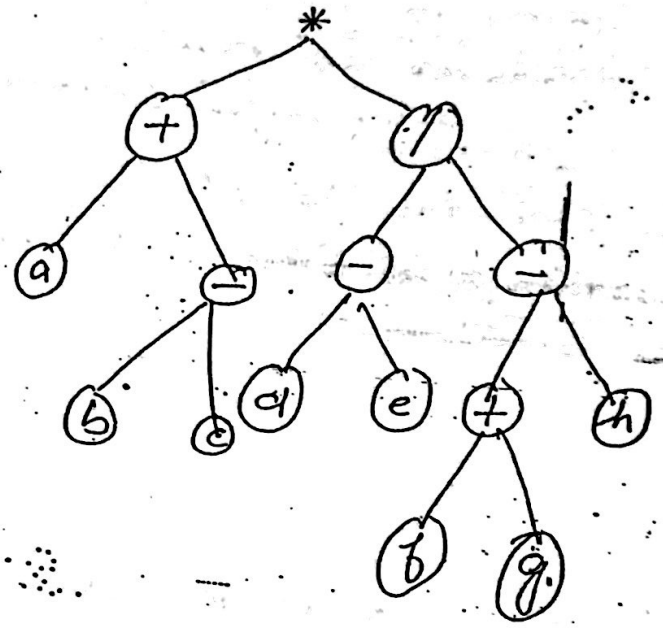


- Preorder :- A B D E F C G H J L K
- Inorder :- D B F E A G C L J H K
- Postorder :- D F E B G L J K H C A

Eg Algebraic Expression :-

$$[a+(b-c)] * [(d-e)/(f+g-h)]$$

Convert in Preorder and Postorder.



Preorder :-

* + a - b c / - d e - + f g h

Postorder :-

a b c - + d e - f g + h - / *

→ **Preorder Algorithm** (Root, left, right) :→ Recursively at each node using stack

Step 1 :- Repeat step 2 to 4 while TREE != NULL

Step 2 :- Write ^ TREE → DATA

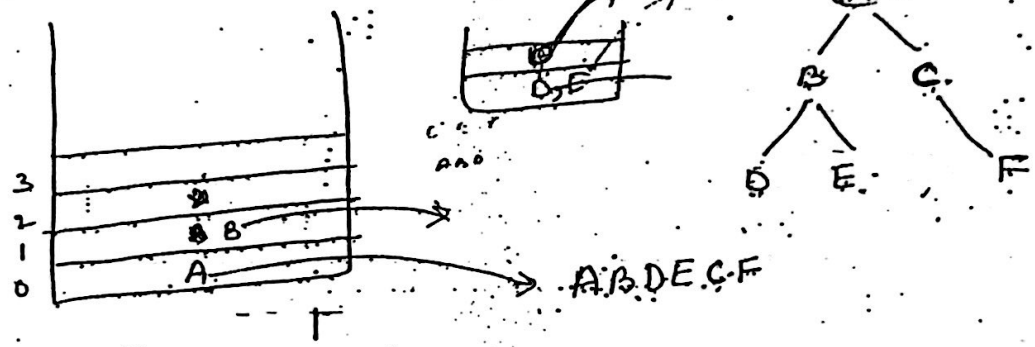
Step 3 :- PREORDER (TREE → LEFT)

Step 4 :- PREORDER (TREE → RIGHT)

[End of While].

Step 5 :- END.

ABDECF



→ In-order Algorithm :- (Left → Root → Right

Step 1 :- Repeat step 2 to 4 while TREE != NULL

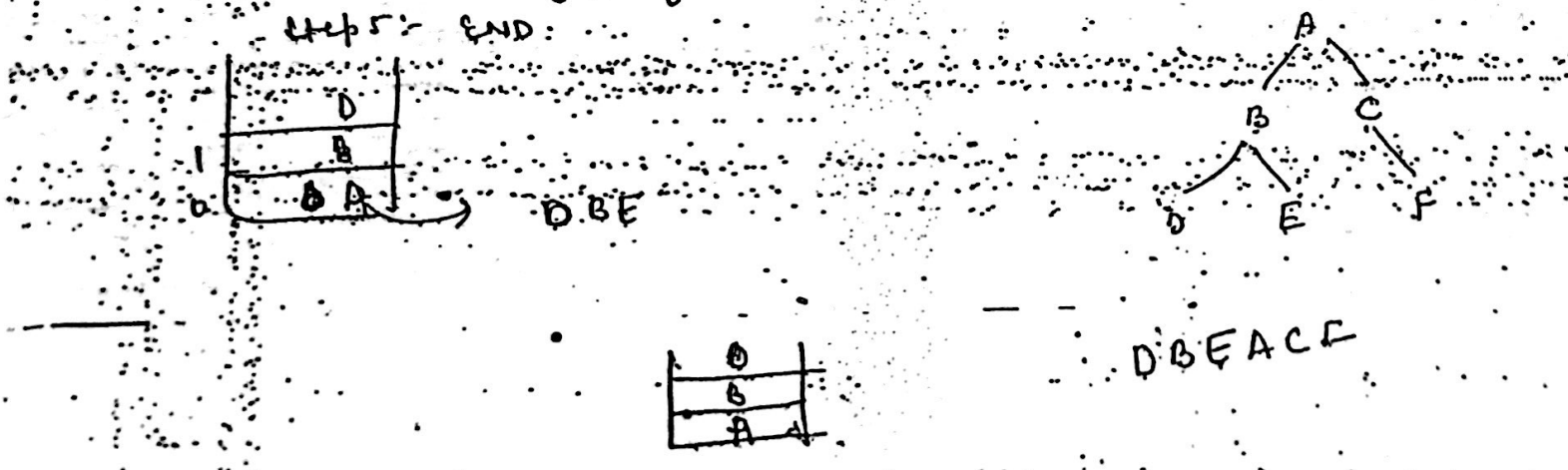
Step 2 :- INORDER (TREE → LEFT)

Step 3 :- WRITE ^ TREE → DATA

Step 4 :- INORDER (TREE → RIGHT)

(End of While)

Step 5 :- END.



Postorder Algorithm :- Left → Right → Root

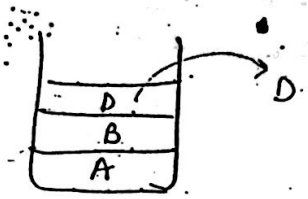
Step 1 :- Repeat step 2 to 4 while TREE != NULL

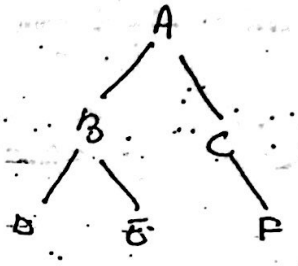
Step 2 :- POSTORDER (TREE → LEFT)

Step 3 :- POSTORDER (TREE → RIGHT)

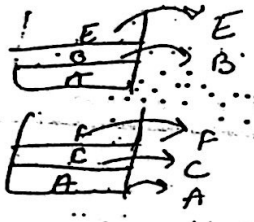
Step 4 :- Write TREE → DATA
(End of While)

Step 5 :- END.





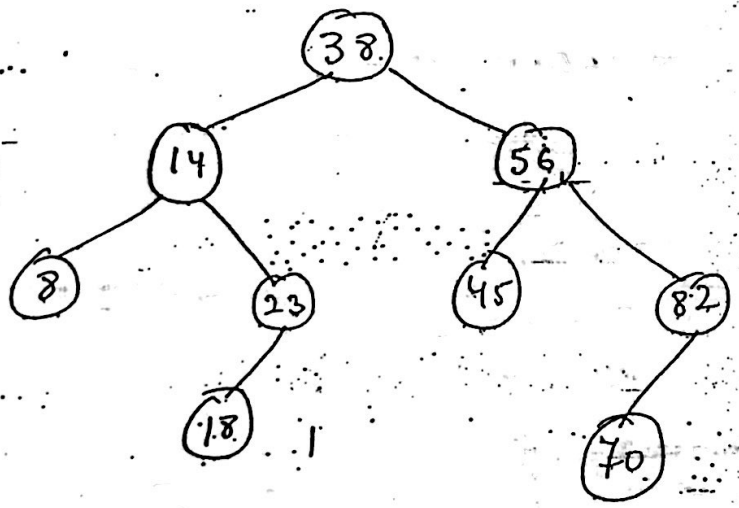
DEBFCA



Binary Search Tree :- considered to be efficient data structure especially when compared with sorted linear arrays.

Binary Search Tree also known as an ordered B.T. where nodes are arranged in an order.

In BST, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly all the nodes in the right sub-tree have a value greater to or equal to than that of root node. The same rule applicable to subtree as well.



BST

BST is a binary Tree with following properties :-

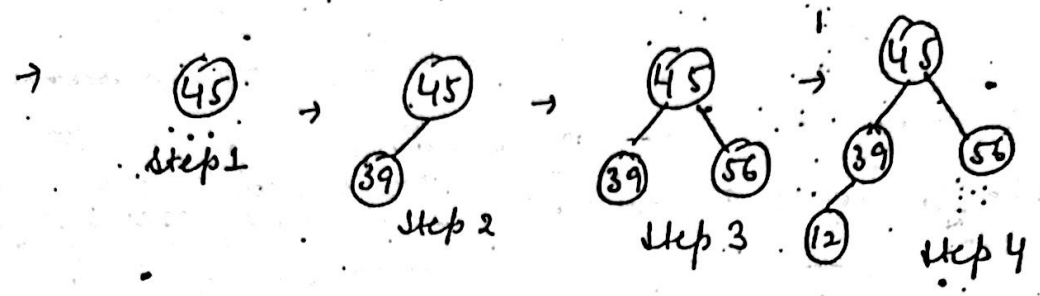
→ Left sub-Tree of a Node N contains nodes that are less than N's Value.

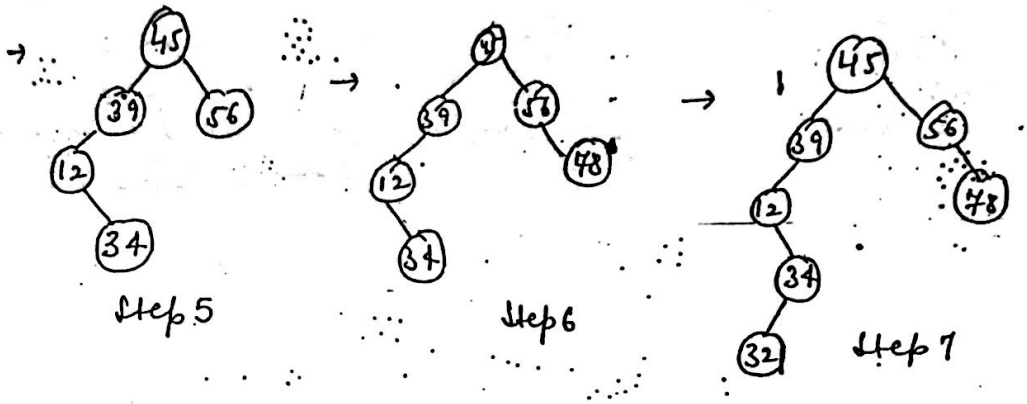
→ Right subtree of a node N contains values that are greater than N's Value.

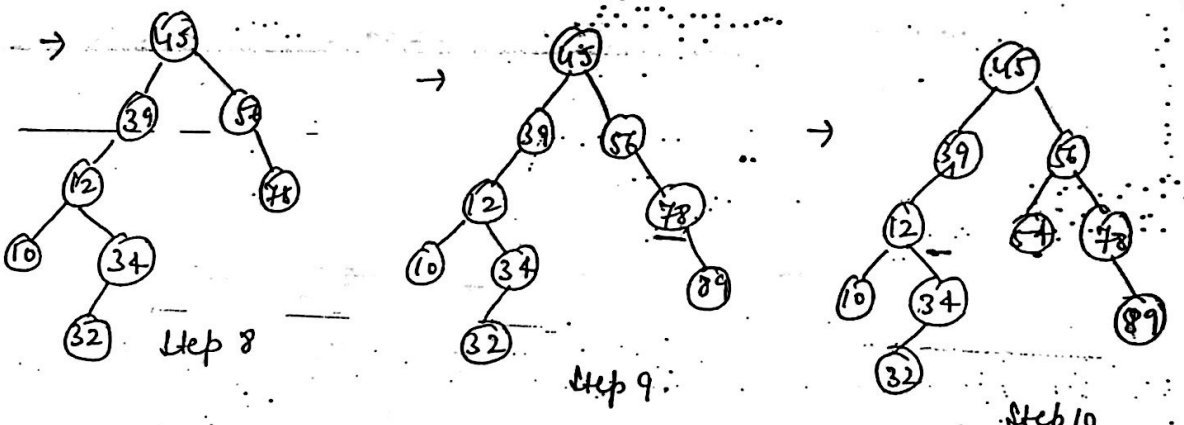
→ Both the left and right binary tree also satisfies these properties and thus are BST.

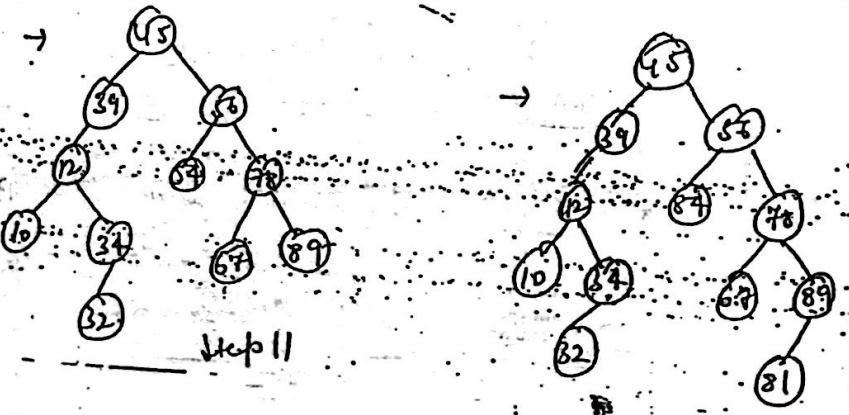
Q) Create a BST using following data elements:

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81









Operation on BST :-

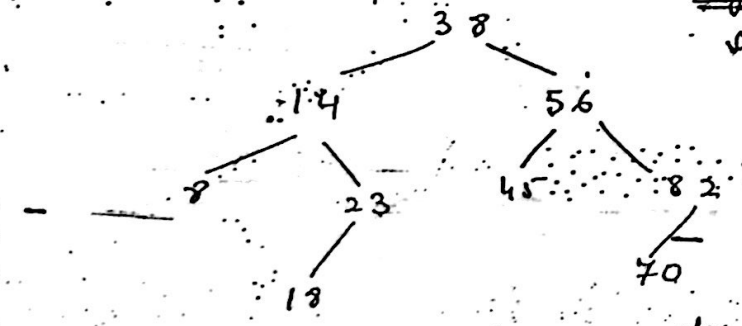
① Searching ② Inserting ③ deleting.

→ Searching ITEM = 18, we have following steps :

① Compare 18 with the Root Node = 38
since 18 < 38, proceed to the left child of 38, which is 14.

② Compare 18 with 14, since 18 > 14, proceed to the right child of 14, which is 23.

③ Compare 18 with 23, since 18 < 23, proceed to the left child of 23, which is 18.



Algo :-
Step 1 :- If TREE → DATA = VAL
or
TREE = NULL, then
Return TREE
else
if VAL < TREE → DATA
Return searchElement (TREE → left, VAL)
else
Return searchelement (TREE → RIGHT, VAL)
(end of if)
(end of if)
Step 2 :- End.

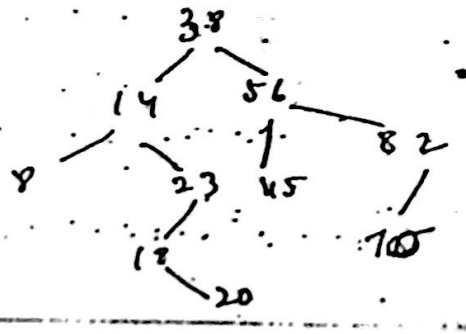
→ Insertion :- INSERT = 20.

Step 1 :- 20 < 38 ⇒ proceed left, found 14

Step 2 :- 20 > 14 ⇒ proceed to Right, found 23

Step 3 :- 20 < 23 ⇒ proceed to left

Step 4 :- 20 > 18 ⇒ insert on right

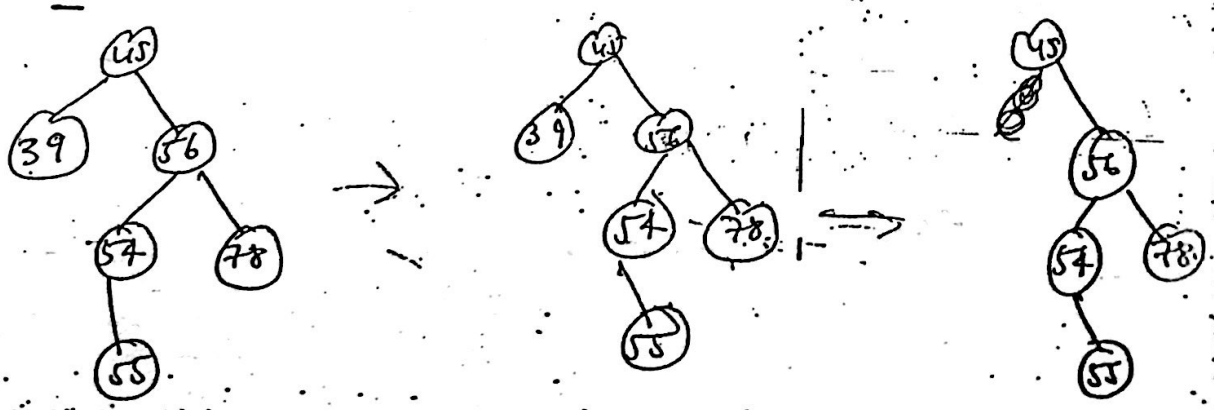


Deletion:- Insertion :- Algo.

Step 1:- If TREE = NULL, then
Allocate memory for TREE
Set TREE → DATA = VAL
Set TREE → LEFT = TREE → RIGHT = NULL
else
If val < TREE → DATA
Insert (TREE → LEFT, VAL)
else
Insert (TREE → RIGHT, VAL)
(end of if)
(end of if)
Step 2 : End.

Deletion :- C.

Case 1 :- Deleting a Node that has No Children
Suppose delete 55 78 (Simple deletion)



Step 1 → 78 → 56

Step 2 → 78 → 45 → Proceed to right

proceed to Left → Delete 78

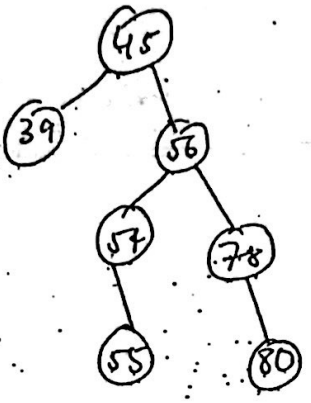
→ Delete

→ Deleting a Node with Two children :-

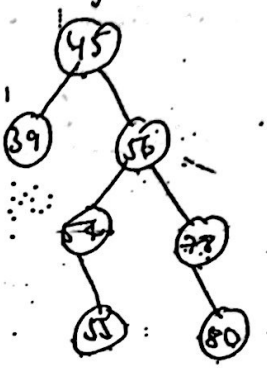
replace node's value with its inorder predecessor (right most child of the left sub tree) or inorder successor (left most child of the right sub-tree).

Deleting Node ⇒ (56)

Eg

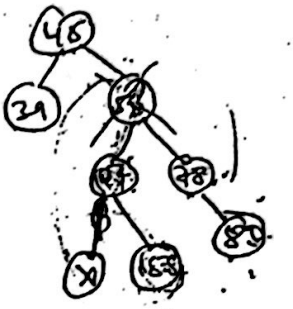


Step 1



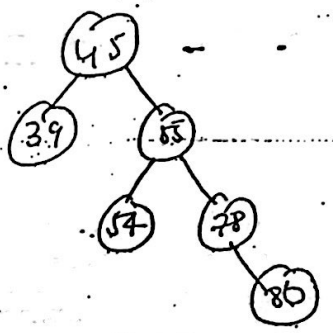
→ 56 > 45
→ Proceed to right

Step 2



Step 3

- Delete 56
- Replace 56 by 55

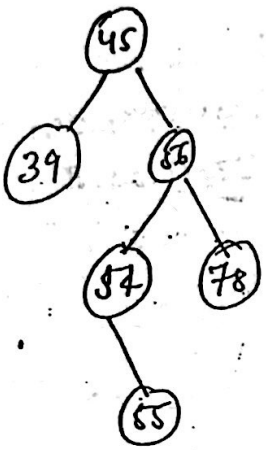


Step 4

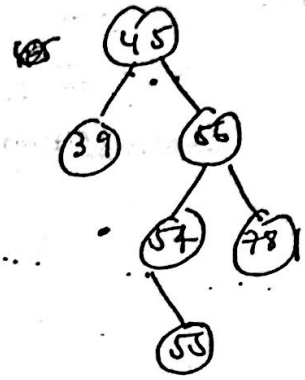
OR

→ Delete 56
→ Replace by 78

- Deleting a Node with one child :- In this case replace the Node with the child. (Delete 54)



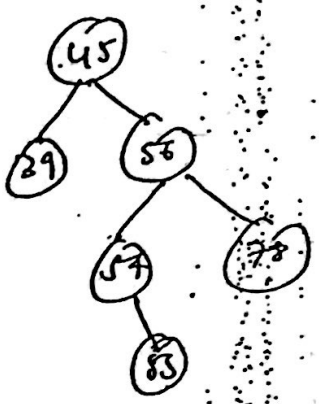
Step 1



Step 2

→ 54 > 45

→ Proceed to Right

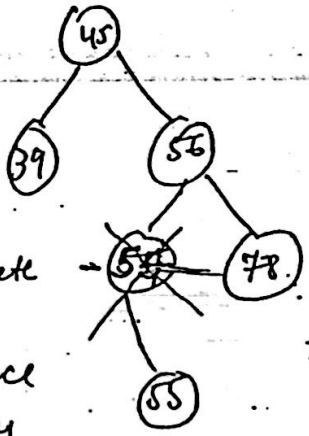


Step 3

→ 54 < 56

→ Proceed to left

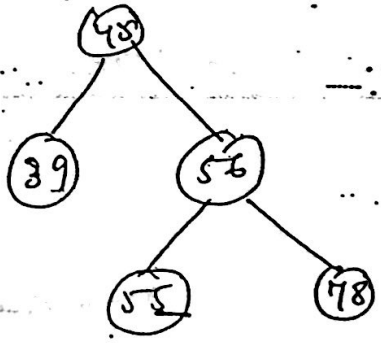
→ Found 54



→ Delete 54

→ Replace 54 by 55

Step 4



Step 5

Algorithm to delete a node from BST

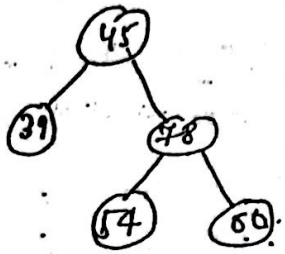
Step 1 :- If TREE = NULL, then
write "VAL not found in tree".
else if VAL < TREE → DATA
Delete (TREE → LEFT, VAL)
else if VAL > TREE → DATA
Delete (TREE → RIGHT, VAL)
else if TREE → LEFT AND TREE → RIGHT = NULL
Set TEMP = find largest Node (TREE → LEFT)
Set TREE → DATA = TEMP → DATA
Delete (TREE → LEFT, TEMP → DATA)
else
Set TEMP = TREE
If TREE → LEFT = NULL and
TREE → RIGHT = NULL
Set TREE = NULL
else if TREE LEFT ! = NULL
Set TREE = TREE → LEFT
else
Set TREE = TREE → RIGHT
FREE TEMP
(end of If)

Step 2 :- End

Determing the Height of a Tree :-

We Calculate the height of the left Sub-Tree and the Right subtree, whichever height is greater, 1 is addeed to it.

for Eg



BST with height 3

height (leftsubtree) = 1
height (rightsubtree) = 2
∴ Height of BST = height (rightsubtree) + 1
= 2 + 1 = 3.

Recursive Algo to determine the height of BST :-

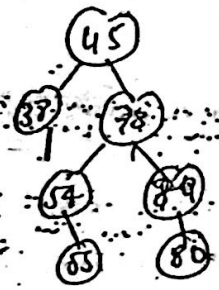
Algo :- Stp 1: If TREE = NULL, then
Return 0
else
Set LeftHeight = Height (TREE → LEFT)
Set RightHeight = Height (TREE → RIGHT)
If LeftHeight > RightHeight
Return LeftHeight + 1
else
Return RightHeight + 1
(end of if)
(end of if)

Stp 2 :- END

Determing the No. of Nodes in BST :-

Step 1 : If TREE = NULL, then
Return 0
else
Return totalNodes (TREE → LEFT) +
totalNodes (TREE → RIGHT) + 1
(end of if)

Step 2 : END



TotalNodes in left = 1
TotalNodes in right = 5

Total Nodes in TREE = (1 + 5) + 1
= 6 + 1
= 7

Removing BST :- Algo ! (Delete left Nodes and Right Nodes then delete Root.)

Step 1 :- If TREE != NULL, then

```
deleteTree (TREE → LEFT)
deleteTree (TREE → RIGHT)
Free (TREE)
(End of if)
```

Step 2 :- END.

Find the Smallest Node in BST :- Algo (Recursive)

Step 1 :- If TREE = NULL or TREE → LEFT = NULL, then

```
Return TREE
else
Return findSmallestElement (TREE → LEFT)
(End of if)
```

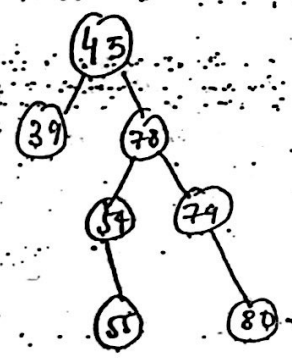
Step 2 :- END.

Find the largest Node in BST :- Algo (Recursive)

Step 1 :- If TREE = NULL or TREE → RIGHT = NULL, then

```
Return TREE
else
Return findLargestElement (TREE → RIGHT)
(End of if)
```

Step 2 :- END.



Smallest Node (left Most child of the Left Sub-Tree).

Largest Node (Right Most child of the Right Sub-Tree)

Application of BST :-

Consider a collection of n data items,

$A_1, A_2, \ldots\ldots A_n$.

Suppose we want to delete or find all duplicates in the collection. This is one of the straightforward way to do so is as follows :

Algo(1) Scan the elements from A_1 to A_n (i.e from left to Right)

(a) For each element A_K compare A_K with $A_1, A_2, \ldots A_{K-1}$, that is compare A_K with those element which precede A_K

(b) If A_K does occur among $A_1, A_2, \ldots A_{K-1}$, then Delete A_K.

→ After all elements have been scanned, there will be no duplicates.

Consider Time Complexity of Algo(1), which is determine by the no. of comparisons.
So the no. $f(n)$ of comparisons reqd by Algo(1) is approximately.

$$0+1+2+3+\ldots\ldots+(n-2)+(n-1) = \frac{(n-1)n}{2}$$

$$= O(n^2)$$

For Eg $n = 1000$ Items; Algo(1) will require approx 500,000 comparisons.

Balanced BST :-

To check if T is a balanced tree, we need to calculate its balance factor, which is difference in height b/w the left and right subtrees.

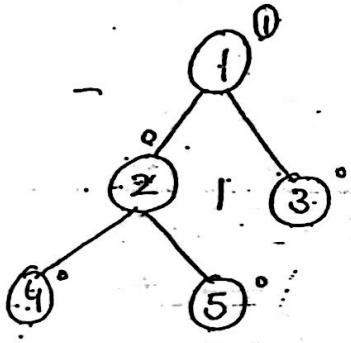
Let Height of Left SubTree = H_L

Let Height of Right Subtree = H_R

Now Balance Factor of the Tree, B can be determined by:

$$B = H_L - H_R$$

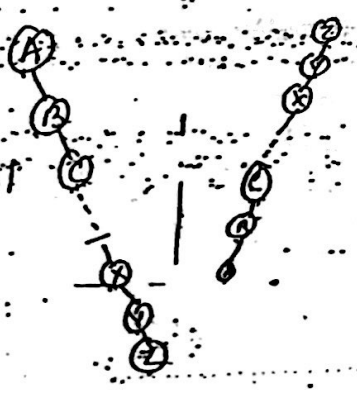
Eg



Balance of leaf Nodes = 0

Balance factor of a given tree = 1.

AVL Search Trees :- Consider an element A, B, C, ... Z to be inserted into BST, Obsiciue how the BST turns out to be right skewed & Again the insertion from Z to A in that order in the BST tree results left skewed BST.



Right Skewed BST

Left skewed BST

- The disadvantage of skewed BST, its time complexity of a search is O(n)
- So we need to balance a BST.

One of the most popular balanced trees was introduced in 1962 by Adilson - Velskii and Landis and was known as AVL trees

Definition:- An non empty binary tree is an AVL tree.
An non empty Binary Tree is an AVL tree iff given T^L and T^R to be the left and right subtrees of T and H^L and H^R to be the heights of subtrees T^L and T^R respectively, T^L and T^R are AVL trees and

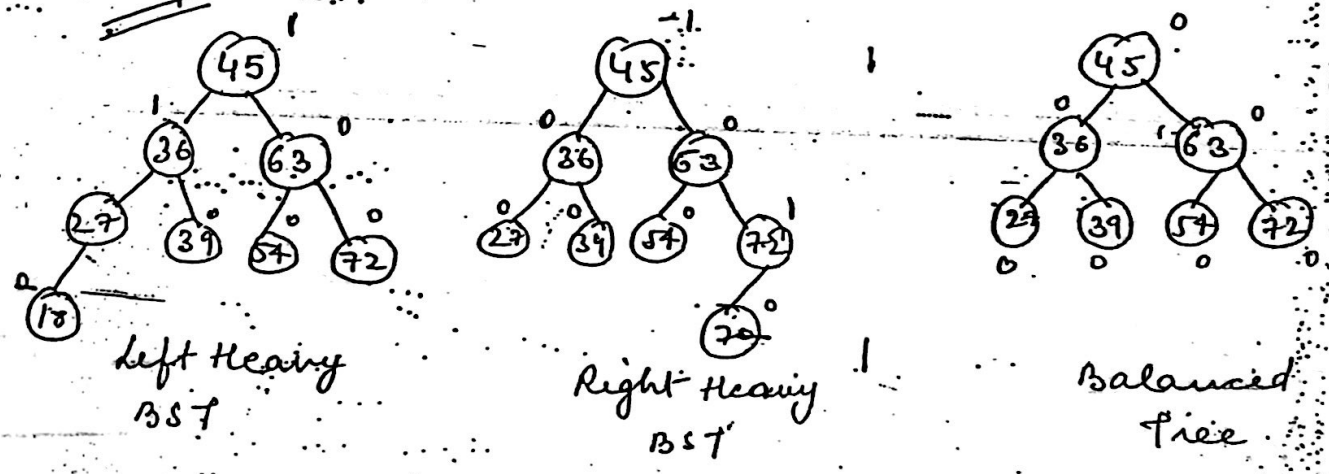
$|H^L - H^R| \leq 1$. and the Tree should be BST.

and

$H^L - H^R$ is known as the Balance factor (BF)

and for AVL Tree the BF of a Node can be either 0, 1 or -1.

For Eg



Left Heavy BST

Right Heavy BST

Balanced Tree

→ Searching an AVL Search Tree:-

Searching an AVL tree for an element is exactly similar to the method used in a BST.

Insertion in an AVL Search Tree :-

Inserting an element into an AVL search tree in its first phase is similar to that of the one used in a BST However, if after insertion of the element, the balance factor of any node in the tree is affected so as to render the BST unbalanced, we resort to techniques called Rotations to restore the balance of a search tree.

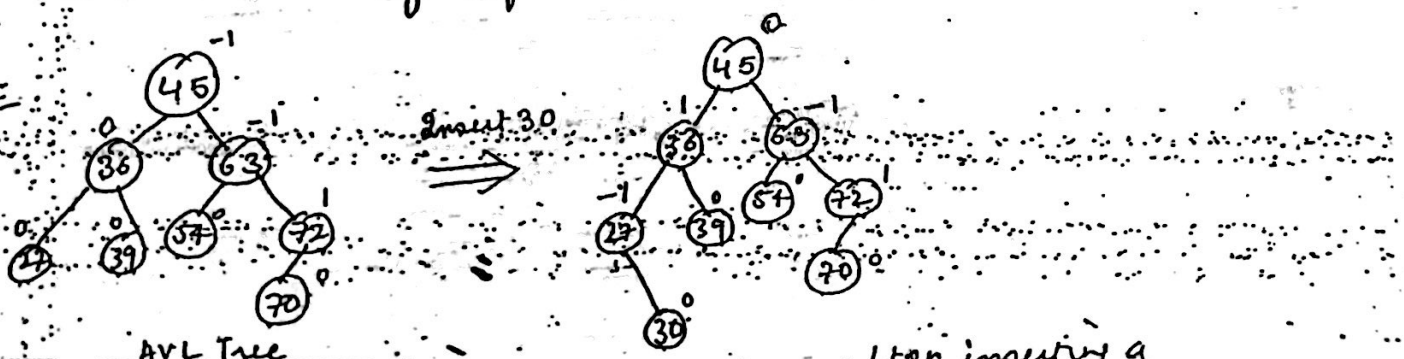
To Rebalancing rotations are classified as LL, LR, RR and RL as mentioned below:

→ LL Rotation :- Inserted Node is in the left subtree of left subtree of node A.

→ RL Rotation :- Inserted Node is in the ~~Right~~ Left subtree of Right subtree of node A.

→ RR Rotation :- Inserted Node is in the Right subtree of Right subtree of node A.

→ LR Rotation :- Inserted Node is in the Right subtree of left subtree of Node A.

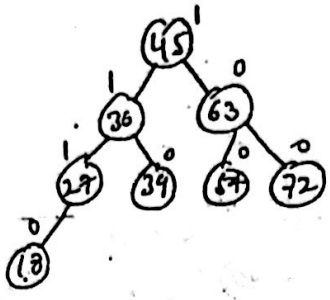


AVL Tree

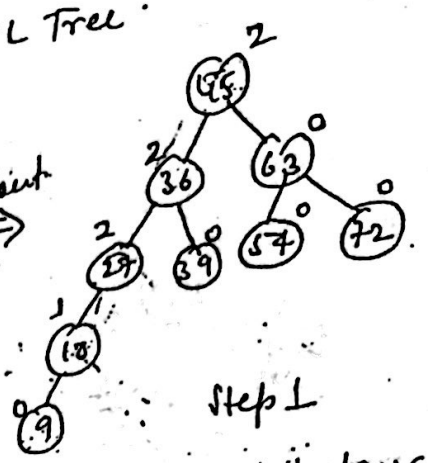
AVL Tree after inserting a node 30

∴ No need of Rotation in AVL tree as it is balanced.

Eg 1 Insert 9 in a AVL Tree



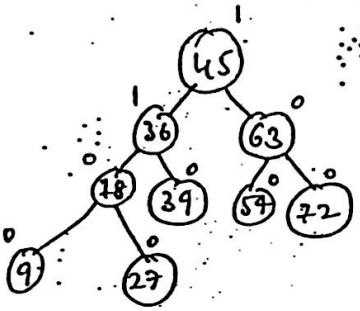
Insert 9 ⇒



Step 1

→ Unbalanced AVL Search Tree

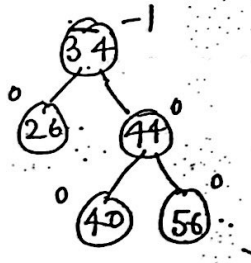
LL Rotation



Step 2

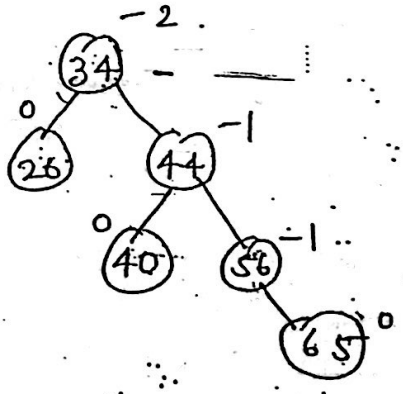
Balanced AVL Search Tree after LL Rotation.

Eg 2



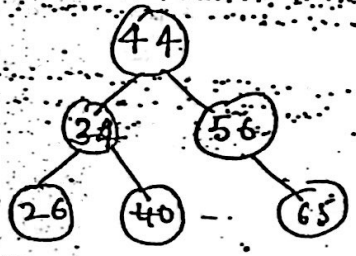
Balanced AVL Tree

Insert 65 ⇒



Unbalanced AVL Search Tree

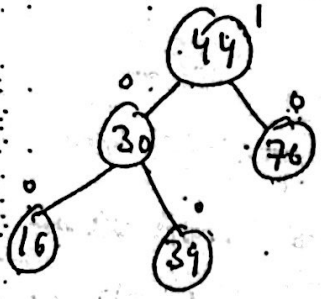
RR Rotation



Balanced AVL Search after RR Rotation

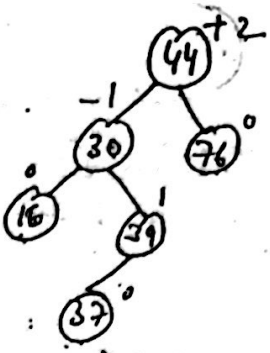
LR and RL Rotations :-

Eg:



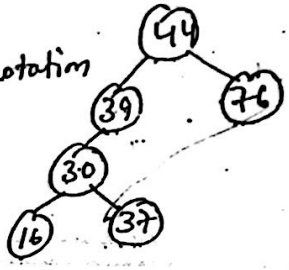
Initial AVL Search Tree

Insert 37

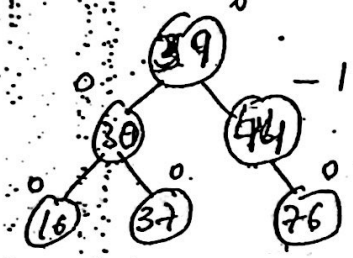


Unbalanced AVL Search Tree

1 left Rotation.

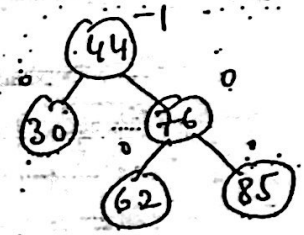


1 Right Rotation

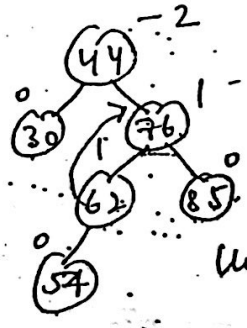


Balanced AVL Search Tree after LR Rotation.

RL Rotations :-

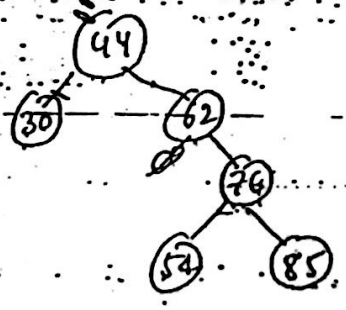


Insert 54

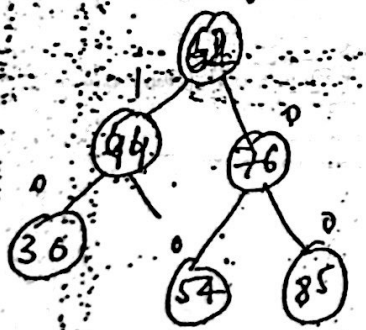


Unbalanced AVL Search Tree

1 Right Rotation



1 Left Rotation

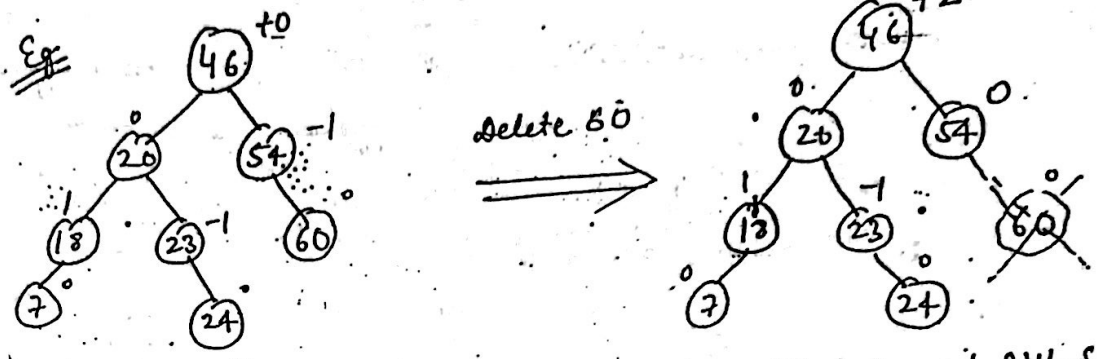


Balanced AVL Search Tree after RL Rotation.

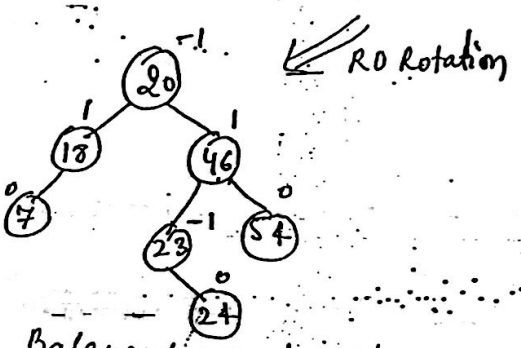
Deletion in AVL Tree :-

RO Rotation :- If Balance factor (BF)(B) = 0, the RO rotation is executed.

Eg

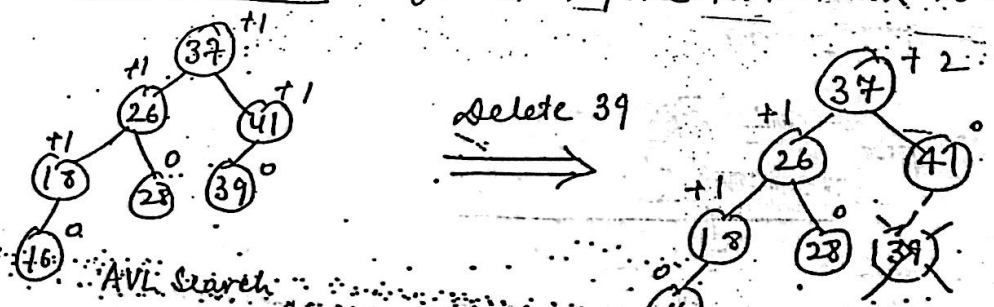


Unbalanced AVL search tree after deletion of 60



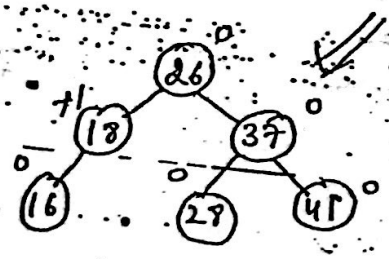
Balanced AVL Search Tree after RO Rotation

R1 Rotation :- If BF(B) = 1, the R1 rotation is executed.



AVL Search tree before deletion

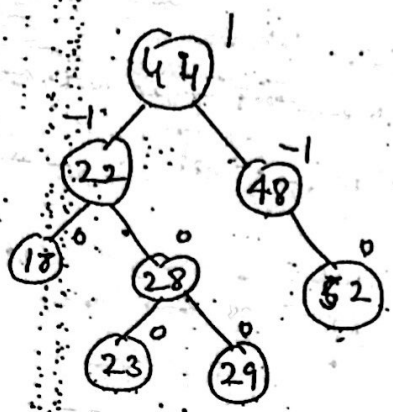
Unbalanced AVL Search tree after deletion of 39



Balanced AVL Search Tree after R1 Rotation

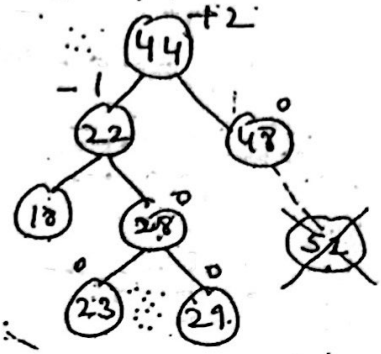
R-1 Rotation :-

If BF(B) = -1 the R-1 rotation is executed.



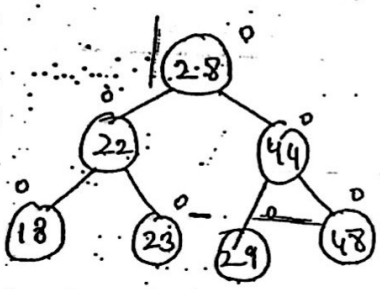
AVL Search Tree before deletion

Delete 52



Unbalanced AVL Search Tree after deletion

R-1 Rotation



Balanced AVL Search tree after R-1 Rotation.

Heaps and its implementations

→ BINARY HEAPS

→ A binary heap is defined as a complete binary tree with all its elements stored is in an sequential Array.

→ Being a complete tree, all the levels of the tree except the last are completely filled.

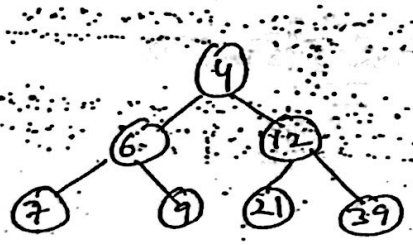
→ the height of a binary tree is given as $\log_2 n$ where n is no. of elements.

→ Heaps (also known as partially ordered trees) are very popular DS for implementing priority queues.

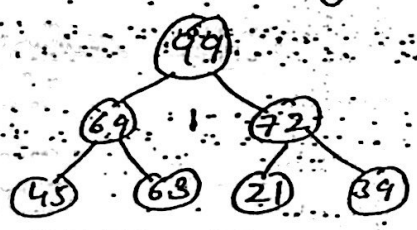
Types of Heaps :-

(1) Min Heap :- In min-heap, the elements at every node will either be less than or equal to the elements at its left and right child.

(2) Max Heap :- In max heap, the elements at every node will either be greater than or equal to the elements at its left and right child.



Min Heap



Max heap

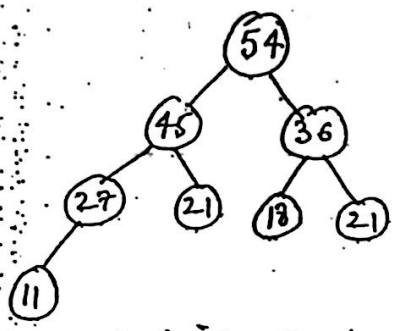
* Binary Heaps :

→ Insertion in a Binary Heap :-

Consider a max heap H with n elements. So,

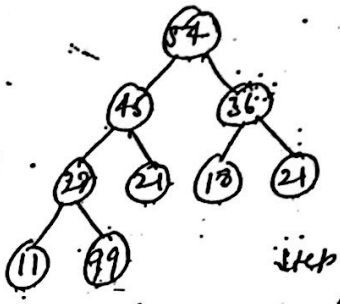
1. Add a new node at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

Eg



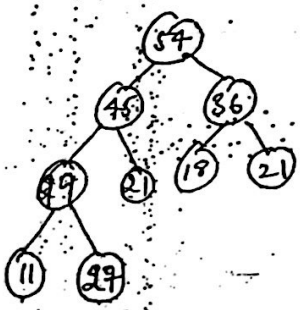
Binary Heap.

Insert 99.

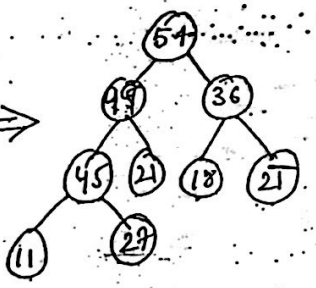


Step 1

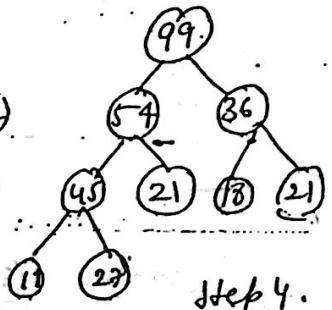
Binary heap after insertion of 99



Step 2



Step 3



Step 4.

Heapify the binary heap.

Algo

Step 1 :- [Add the new value and set its pos]
Set N = N+1, POS = N
Step 2 :- Set HEAP[N] = VAL
Step 3 :- [Find appropriate loc. of Val] Repeat steps 4 and 5 while POS > 0
Step 4 :- Set PAR = POS/2
Step 5 :- If (HEAP[POS] <= HEAP[PAR]), then Goto Step 6
else
SWAP HEAP[POS], HEAP[PAR]
POS = PAR
[END of if]
[END OF LOOP]
Step 6 :- Return

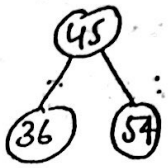
Example 1- Build a heap H from the given set of numbers 45, 36, 54, 27, 63, 72, 61 and 18. Also make memory representation of the heap.

Sol^n

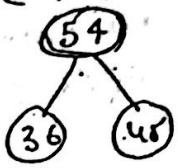
(Step 1)

(Step 2)

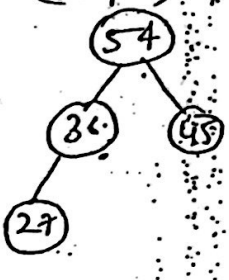
(Step 3)



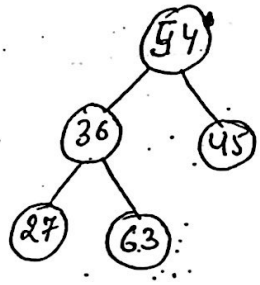
(Step 4)



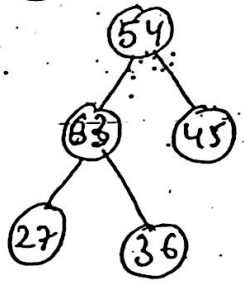
(Step 5)



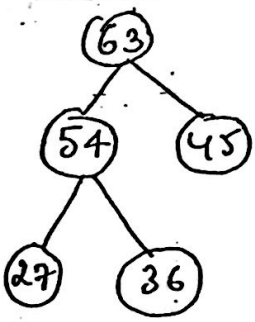
(Step 6)



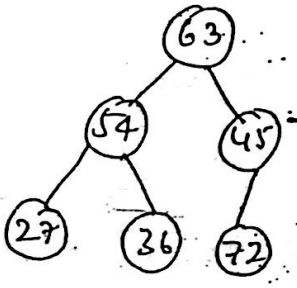
(Step 7)



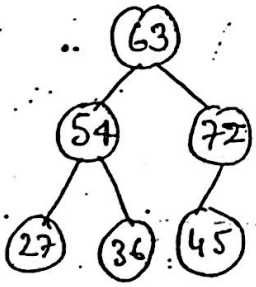
(Step 8)



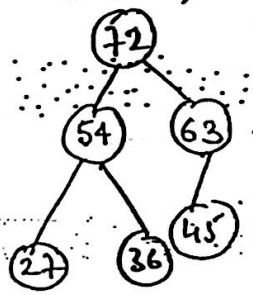
(Step 9)



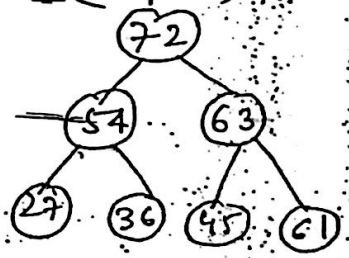
(Step 10)



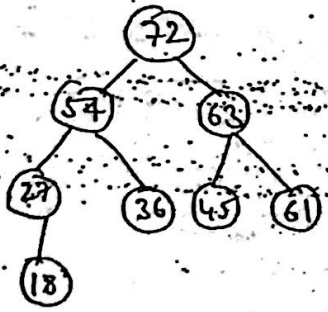
(Step 11)



(Step 12)



(Step 13)

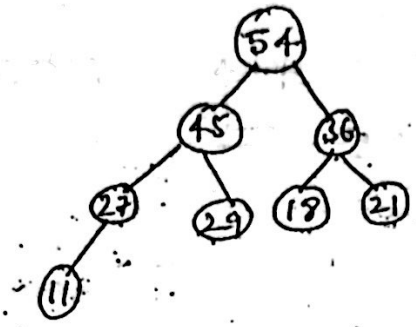


Memory Representation of H (Heap)
(Binary Heap)

HEAP[0]	HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]
72	54	63	27	36	45	61	18		
1	2	3	4	5	6	7	8	9	10

Deleting an Element from a Binary Tree :-

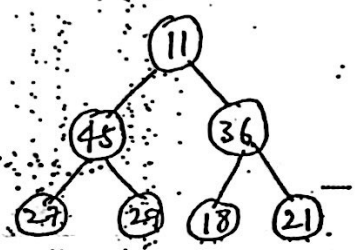
→ Deletion of Root Node :- Replace with the last Node of H (Binary Heap Tree).



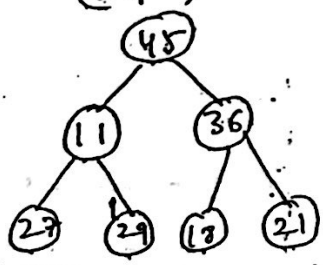
Here, the value of root node = 54 and the value of the last Node = 11.

So, Replace 54 with 11 and delete the last Node.

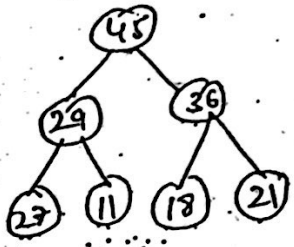
(Step 1)



(Step 2)



(Step 3)



Algo:-

Step 1: [Remove the last node from the heap]
SET LAST = HEAP[N], SET N = N-1

Step 2: [Initialization]
SET PTR = 0, LEFT = 1, RIGHT = 2

Step 3:- SET HEAP[PTR] = LAST

Step 4 :- Repeat steps 5 to 7 while LEFT <= N

Step 5 :- If HEAP[PTR] >= HEAP[LEFT] AND HEAP[PTR] >= HEAP[RIGHT], then
Go to step 8
[End of If]

Step 6 :- If HEAP[RIGHT] <= HEAP[LEFT], then
SWAP HEAP[PTR], HEAP[LEFT]
SET PTR = LEFT
else
SWAP HEAP[PTR], HEAP[RIGHT]
SET PTR = RIGHT
[End of If]

Step 7:- Set LEFT = 2 * PTR and RIGHT = LEFT + 1
(End of loop)

Step 8:- RETURN

Application of Binary Heap :-

A binary heap is mainly applied for :-

1) Sorting an Array using heapsort Algorithm.

2) Implementing priority queues.