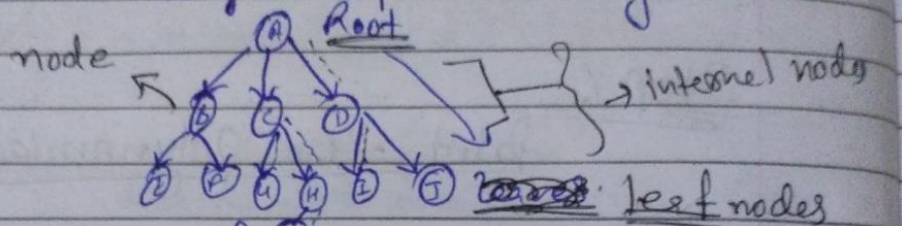
Date:
Page No.:

Unit-4

Tree

→ Non-linear data structure (multiple layer or levels).

→ Data in the form of hierarchy.



→ Tree grows from top to bottom so we can traverse from top to bottom.

Definition

Tree can be defined as a collection of entities (nodes) linked together to simulate a hierarchy.

→ Roots = A

→ nodes = A - B elements of tree.

→ Parent node = A is parent of B, C, D.
∴ immediate predecessor of any node
note → Root doesn't have any parent node.

→ Child node = E and F is child of B.
∴ immediate successors of any node.
note → leaf does not have any child.
E, F, G, H, I, J → leaf nodes or external nodes.

→ non-leaf / internal nodes :- at least have one child.

→ Path → It is a sequence of consecutive edges from source to destination node.

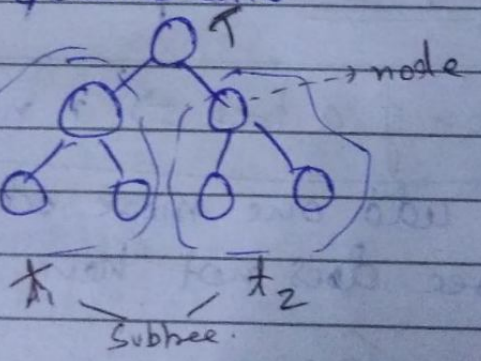
→ Ancestor → Any predecessor node on the path from root to that node.
Ancestor of I = A, D

→ Decendent → Any successor node on the path from that node to leaf node.
decendent of C = H, K, G

Q. Common decendent of C and D.
→ None.

Q. Common Ancestor of H and I.
Ancestor of H = A, C. } Common = (A)
" " I = A, D }

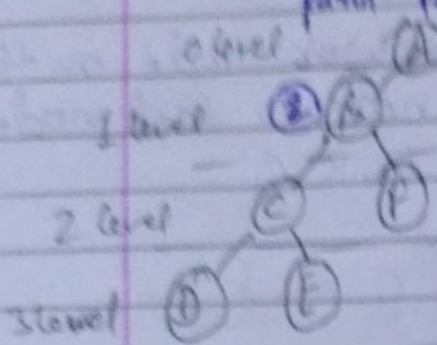
→ Subtree → Subtree of any tree T is a tree containing a node of Tree T and all its decendents.



→ Sibling → All the children of same parent.
→ degree → degree of a node is the no. of children of that node. eg degree of A = 3.
" " J = 0 (leaf node)
⊛ degree of tree → Max. degree of any node in the tree is the degree of tree.
→ depth of a node → The length of the path from root to that node.
depth of H = 1+1 = 2.

Q: depth of root node = ? (always)

→ height of a node: no. of edges in the longest path from that node to leaf.



height of B = B→D = 2
B→E = 2
B→F = 1
∴ B = 2

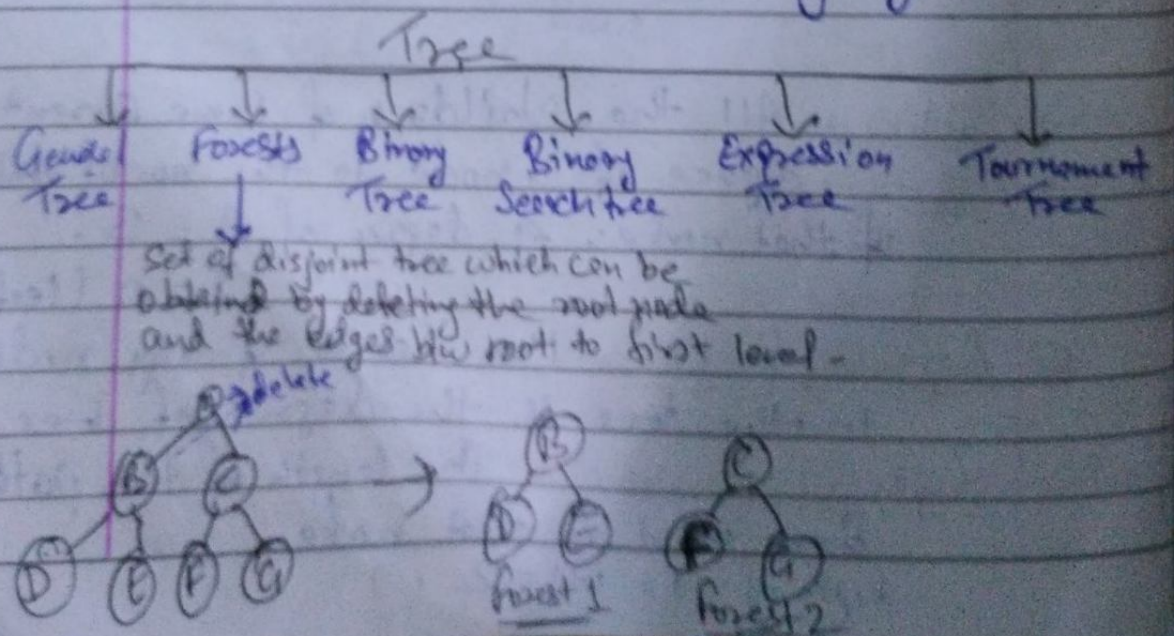
depth B = 1.

⇒ height of a tree = height of root node = 3
Depth of a tree = no. of edges from leaf to root in max path. = 3 (in given case)
(equivalent to level of tree)

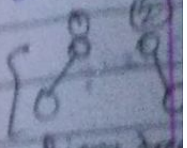
⇒ Levels of node = Depth of node.

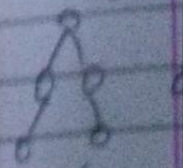
Condition of a tree ⇒ n nodes = (n-1) edges

note: If we add one more edge it forms a cycle. and a tree does not have any cycle



Q: 18
Ans





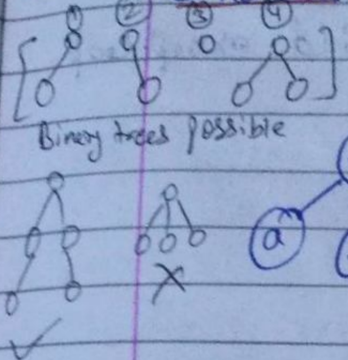
Tree application → Routing Protocol, file s/s, quick search etc.

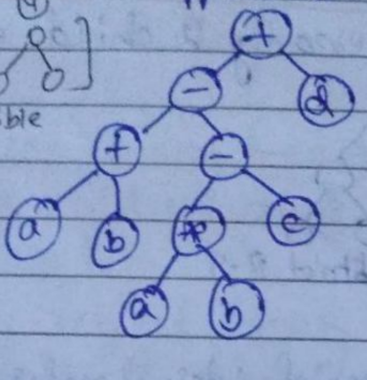
Date :

Page No. :

Q:18 Binary Tree

Ans data structure where node can have at most 2 children. application → expression evaluation.





$\Rightarrow (a+b)-((a*b)-c)+d$

Binary Tree node → | Pointer to left child | Data | Pointer to right child |

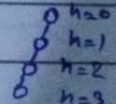
Properties of binary tree

① max. no. of nodes at level $l = 2^l$ ($L=3 \Rightarrow 2^3=8$)

② max. no. of nodes of B.T of height $h = (2^{h+1}-1)$

$2^0+2^1+2^2+\dots+2^h$ (GP series) ↗sum

③ Min. no. of nodes of B.T of height $h = (h+1)$



④ B.T. with n (max. no. of nodes) nodes, min possible height or min. no. of levels is ⇒ (from ②) $n = 2^{h+1}-1$ (from here find h)

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

$\log_2(n+1) = \log_2 2^{h+1}$ ($\because \log_2 2$)

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

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

$h = \lceil \log_2(n+1) - 1 \rceil \Rightarrow$ min. height

[ ] → ceiling function

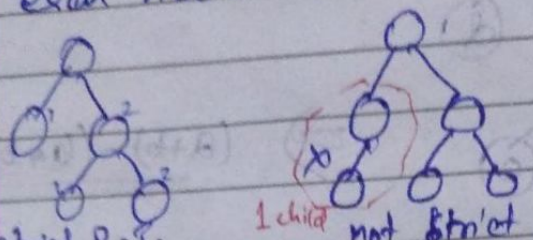
⑤ max height with n as min no. of nodes.

$n = h+1$ (min. no. of nodes at h)

$h = n-1$

Types of Binary tree

① Strict binary Tree / full binary tree → each node contains either 0 or 2 children. or each node have exacly 2 child except leaf.

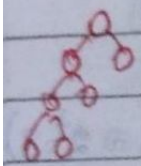


strict B-T. not strict B-T

Property

no. of leaf node = no. of internal nodes + 1

max. no. of nodes = $2^{h+1} - 1$ (Same as binary tree).



min. no. of nodes = $2h + 1$

Min height = $\lceil \log_2(n+1) \rceil - 1$ (Same as binary tree).

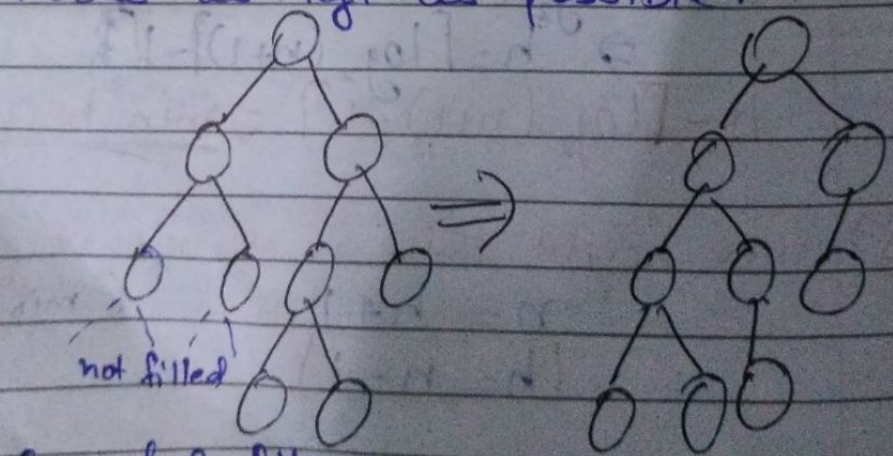
Max height ⇒ $n = 2h+1$

$$h = \frac{n-1}{2}$$

$$= (n-1)/2$$

② Complete Binary tree

All the levels are completely filled (except possibly the last level) and last level has nodes as left as possible.



Second Condition not met

X

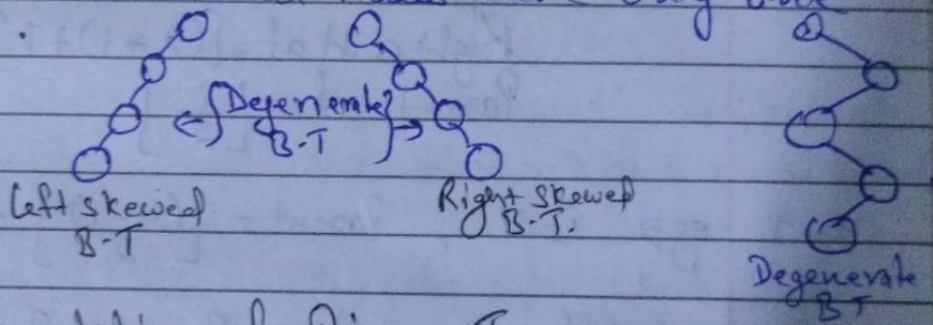
Complete binary Tree (CBT)

last node ⇒ left → Right.

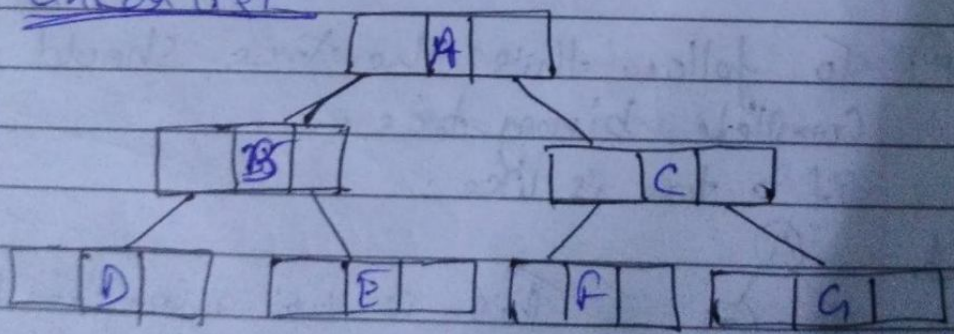
Max. node = $2^{h+1} - 1$ (Same as Binary Tree)
Min node = 2^h (Find out)
Min height = $\lceil \log_2(n+1) \rceil - 1$
Max height = $\log n$.

(3) Perfect binary tree → All internal nodes have 2 children and all leaves are at same level.
Every Perfect binary tree can be a Complete BT as well as Strict binary tree. but vice versa is not possible.

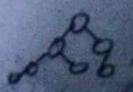
(4) Degenerate Binary tree →
All the internal nodes have only one child.



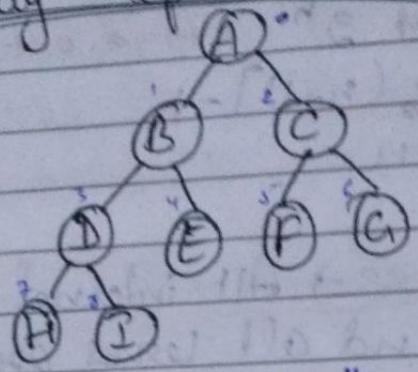
(2) Implementation of Binary Tree
(1) Linked list



Almost Complete Binary Tree → (1) Left child must be there then only right child will be there.
(2) Previous level must be completed before going to next level.



② Array Representation



Case I

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

Case II

A	B	C	D	G	F	G	H	I
1	2	3	4	5	6	7	8	9

Case I if a node is at i^{th} index :- Left child = $[(2*i)+1]$

Right child = $[(2*i)+2]$

Parent = $\left\lfloor \frac{(i-1)}{2} \right\rfloor$ floor value

Case II

If a node is at i^{th} index :

Left child at = $(2*i)$

Right child at = $[(2*i)+1]$

Parent at = $\left[\frac{i}{2}\right]$

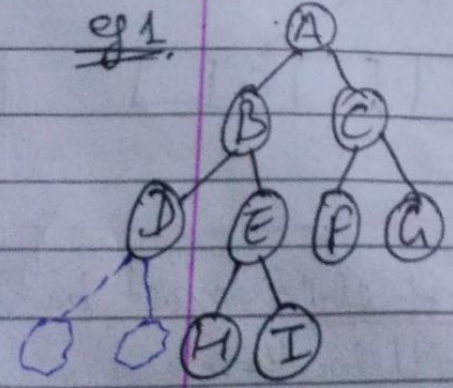
Case I eg ⇒ $i=4$, Parent = $\left\lfloor \frac{(4-1)}{2} \right\rfloor = \lfloor 1.5 \rfloor = 1$

at index 1 we have B. ∴ B is parent of E.

(#) To follow this the tree should be a Complete binary tree.

If a tree is like :

eg 1.



This is not a complete binary tree as all the leaf nodes are not present is left most side. D doesn't have any child but E have two child.

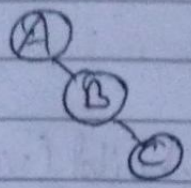
To represent tree like this in array we have to

Leave blank spaces for the child of D to make it a correct representation for holding the formula.

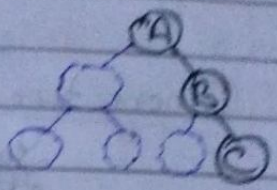
A	B	C	D	E	F	G	_	_	H	I
0	1	2	3	4	5	6	7	8	9	10

A	B	C	D	E	F	G	_	_	H	I
1	2	3	4	5	6	7	8	9	10	11

eg2.



not complete.



Complete

Array representation:

A	_	B	_	_	_	C
0	1	2	3	4	5	6

(A) In Python

→ Binary tree library helps to directly implement binary tree. also support heap and binary search tree (BST).

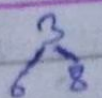
→ Not preinstalled with python but can be, from python's standard utility module. Command for the same is:

pip install binarytree.

Syntax for node:- binarytree.Node(value, left=None, Right=None).
(value → number (data))

⇒ Creating Node:

```
From binarytree import Node
root = Node(3)
root.left = Node(6)
root.right = Node(8)
Print('Binary tree :', root)
Print('List of nodes:', list(root))
Print('Size of tree :', root.size)
Print('Height of tree:', root.height)
```

Output: 

List of Nodes: [Node(3), Node(6), Node(8)]

Size of tree: 3
height of tree: 1

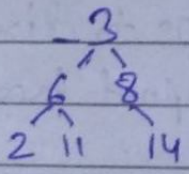
⇒ Binary tree from List

Syntax:- binarytree.build(values)
Returns:- root of binary tree.

Program:-

```
from binarytree import build
nodes = [3,6,8,2,11,None,14]
binary_tree = build(nodes)
Print('Binary tree from list:\n', binary_tree)
Print('\nList from binary tree:', binary_tree.values)
```

output:



List from binary tree: [3,6,8,2,11,None,14]

⇒ Random binary tree:

tree() → generates random Binary tree. Returns Root

Syntax:- binarytree.tree(height=(0-9) values, is_perfect=False)

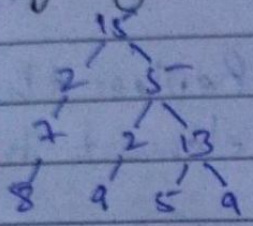
Program:-

```
from binarytree import tree
root = tree()
Print("Binary tree of any height:")
Print(root)
root2 = tree(height=2)
Print("Binary tree of given height:")
Print(root2)
```

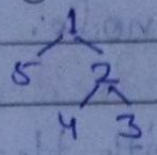
root3 = tree(height=2, is_Perfect = True)
Print('Perfect binary tree of given height:')
Print(root3)

Output

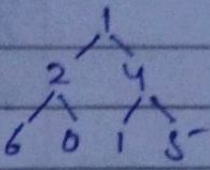
① Binary by tree of any height.



② Binary tree of given height: (2)



③ Perfect binary tree of given height:



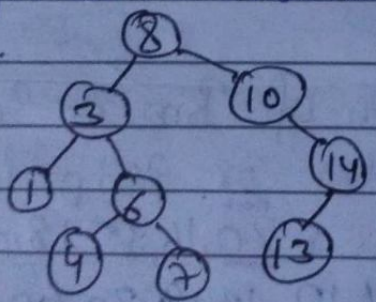
④ Binary Search tree (height = ⌊log n⌋) → balanced BST (average case)

It is an ordered or sorted binary tree. It is a node-based binary tree structure with following properties. (used for searching of data as it is sorted).

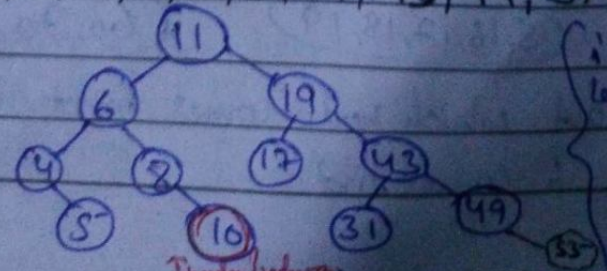
① Left subtree of a node contains only nodes with keys lesser than the node's key.

② Right subtree of a node contains only nodes with keys greater than the node's key.

③ Left and right subtree each must also be a binary search tree.



Q: Draw BST by inserting following nos. left to right.
11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



Traversal { 3 time → Post order
2 time → Inorder (sorted).

Date :
Page No. :

Now insert 55 → It'll be in right part of 49.

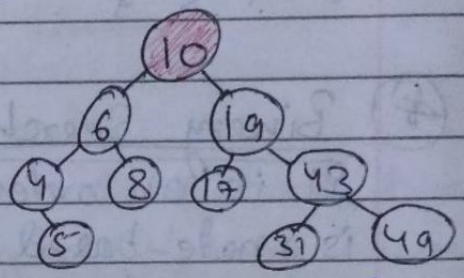
(4) Deletion in BST

(1) 0 child (directly delete the node).
(2) 1 child. (node could be replaced by its child)
(3) 2 children.

For two children we have two ways to delete the node.

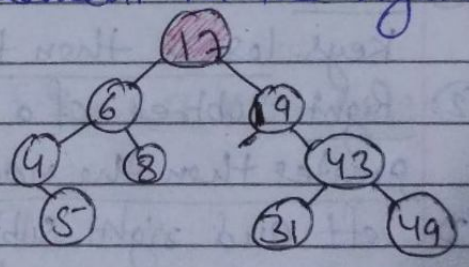
⇒ Inorder Predecessor → Replace the node with the largest element in the left subtree.

Inorder Predecessor →



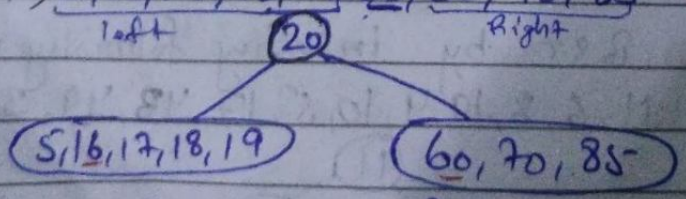
⇒ Inorder Successor → Replace the node with the smallest element in the right subtree.

Inorder Successor →



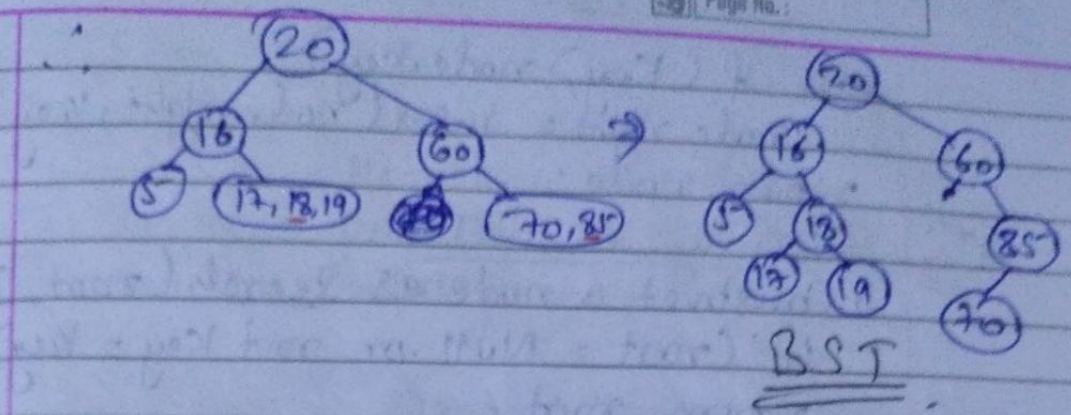
Q: Construction of Binary Search Tree when only preorder or postorder is given.

given → (Root, left, right) Preorder → 20, 16, 5, 7, 18, 17, 19, 60, 85, 70.
(Ascending Order) Inorder → 5, 16, 17, 18, 19, 20, 60, 70, 85 (left, Root, Right)



find out which node comes first in pre-order (left, right)
here 16 comes first.

BST store Unique element as it is used to store key element for searching in database.



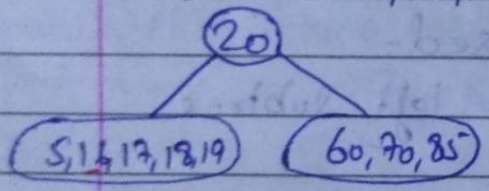
Q.19 Construct BST from Postorder.

Ans (Left, Right, Root)

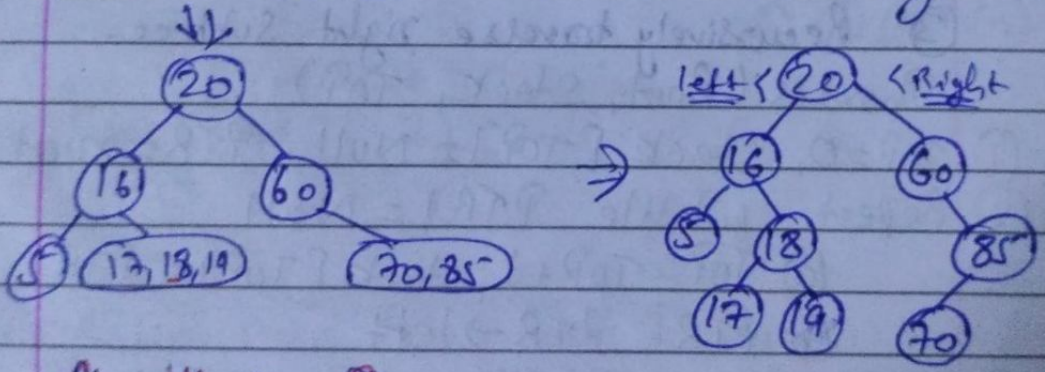
Postorder → 5, 17, 19, 18, 16, 70, 85, 60, 20. (Root)

Inorder traversal will always be in ascending order for Binary Search tree.

Inorder → 5, 16, 17, 18, 19, 20 (Root), 60, 70, 85 (Left, Root, Right)



Find which element in the Subtree comes first in Postorder (from Right → left)



Algorithms:

① Insertion in BST

→ Create node : insert (node, key)

→ If root is empty:

If (node = NULL)

return new node (key)

→ if (key < node.key)

node.left = insert(node.left, key)

```
else if (key > node.key)
    node.right = insert(node.right, key)
return node.
```

② Searching in BST

→ Construct a node as search(root, key)

```
if (root = Null or root key = key)
    return root.
else if (root.key < key)
    return search(root.right, key)
else
    return search(root.left, key)
```

Algo for Preorder.

Until all node traversed-

1. Recursively traverse left subtree
2. visit root node
3. Recursively traverse right subtree.

Preorder (Root, Stack, TOP)

1. TOP = 0, Stack [TOP] = Null, PTR = root.
2. Repeat while PTR != Null
   a. TOP = TOP + 1, Stack [TOP] = PTR
   b. PTR = PTR → left
3. PTR = Stack [TOP], TOP = TOP - 1
4. Repeat Step 5 → 7 until PTR != Null
5. Do
   write PTR → info
6. If PTR → right != Null
   a. Ptr = Ptr → Right
   Goto step 2
7. PTR = Stack [TOP], TOP = TOP - 1.
8. Exit.

Heap Tree (Insertion)

Insert key one by one in given order O(nlogn)

Heapify Method. O(n)

(3) Deletion in BST

(*) Heap Tree

Heap tree follow two property.

① Structural property (ACBT condition)

② ordering property (Max, Min heap)

Heap tree → Condition → Almost complete binary tree.

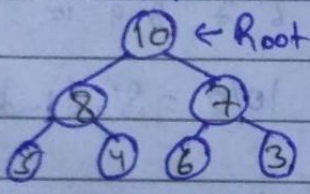
① All nodes of previous level must be complete before moving to next level.

② Left child should complete first.

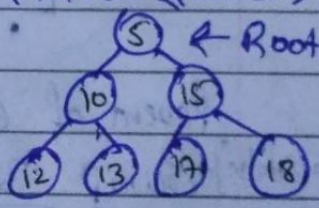
Heap Tree

Max Heap (Parent ≥ child)

A[Parent(i)] ≥ A[i] } condition.



Min Heap (Parent ≤ child)



Array.

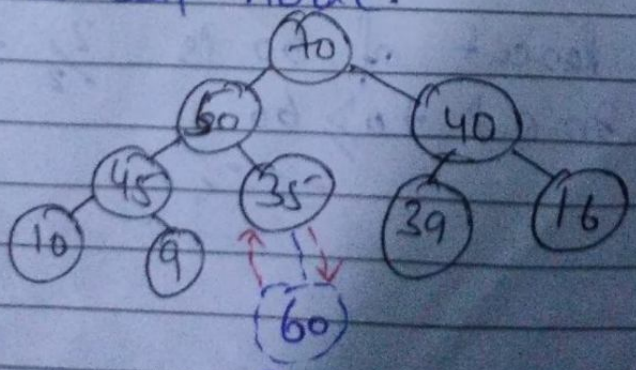
10	8	7	5	4	6	3
1	2	3	4	5	6	7

① Insertion in Max heap

(*) Always insert from leaf to follow the ACBT.

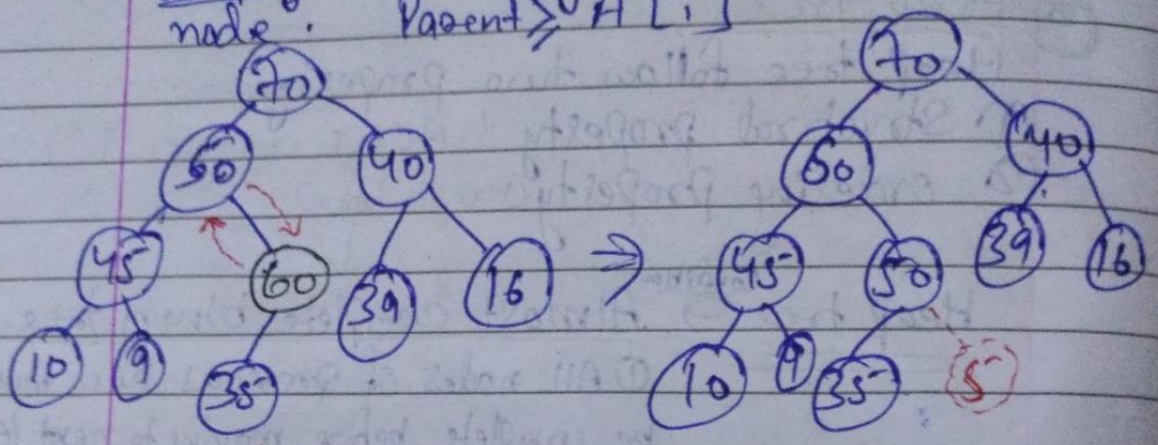
→ we insert the node at first leftmost vacant place at leaf node.

Insert 60



Insert

Heapify for balancing the ~~root~~ parent node. Parent ≥ A[i]



Ans

⇒ Now insert 5

Array representation:

1	2	3	4	5	6	7	8	9	10
70	60	40	45	35	39	16	10	9	60

Parent of 60 = i/2 = 10/2 = 5 (index).

[35 ≥ 60 ⇒ false] ⇒ Swap 35 ⇄ 60

1	2	3	4	5	6	7	8	9	10
70	50	40	45	60	39	16	10	9	35

Parent of 60 = i/2 = ⌊5/2⌋ = 2 (index).

[50 ≥ 60 ⇒ false] ⇒ Swap 50 ⇄ 60

1	2	3	4	5	6	7	8	9	10
70	60	40	45	50	39	16	10	9	35

Now Parent of 60 is 2/2 = 1 index = 70 and 70 ≥ 60.

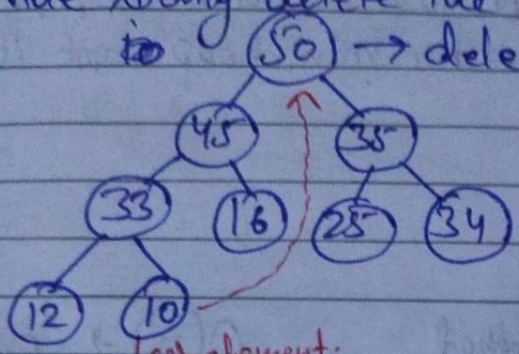
⇒

$n(\log n)$ → time complexity to build a heap using this method.

Date:
Page No.:

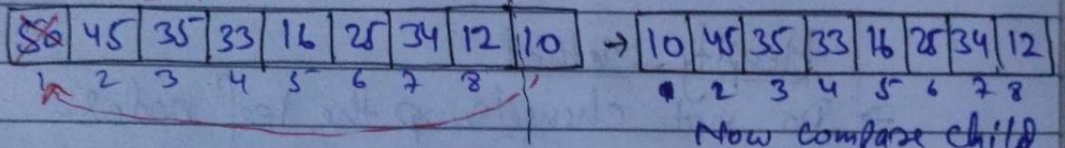
(2) Deletion in Max heap — delete — $O(1)$ → time, $O(n)$ — arrange

Rule → only delete the root node or (lowest level last node) ← best case



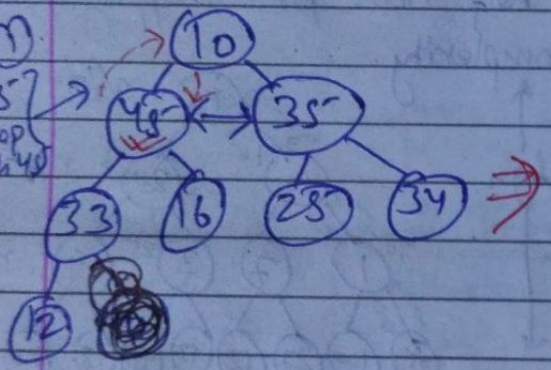
→ delete.

last element shifted to root.

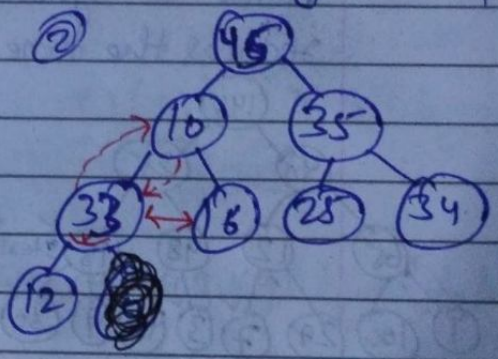


Now compare child and parent to satisfy Max heap.

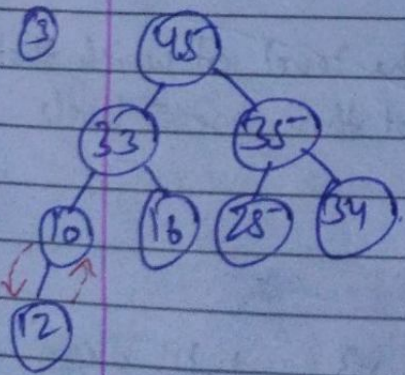
(1) 45 > 35 ∴ swap 10 with 45

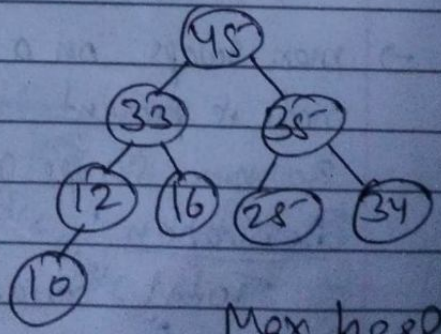


(2)



(3)





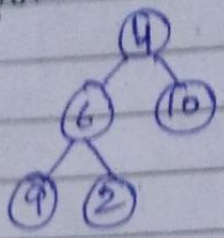
Max heap

For array find left child = $2 \times i$
Right child = $(2 \times i) + 1$ } complex
and swap with parent index.

Heap Sort

input → 4, 6, 10, 9, 2 output → 2, 4, 6, 9, 10

→ For min heap root is smaller.



Heap

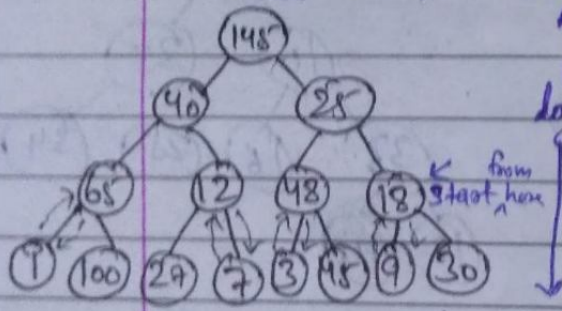
(1) Build heap method $O(n)$ → Complexity

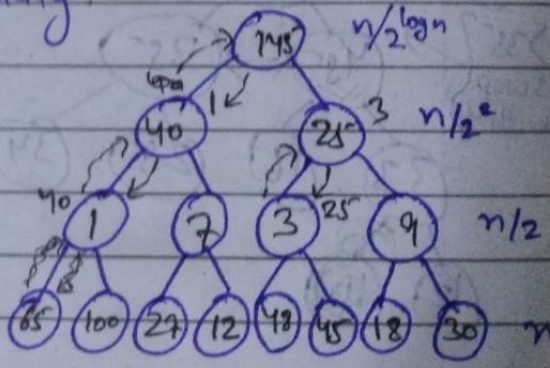
Heapify Method

→ Do not change/swap the leaf nodes.

→ If total n elements, then $n/2$ = leaf nodes.

→ Hence it ignores half node for swapping and reduces the time complexity.





→ max swaps on a node is the level at which that node is present. eg → below root there are 3 levels so max swaps possible = 3.

Derivation:

Total swaps = S

$$S = \frac{n}{2^0}\times 0 + \frac{n}{2^1}\times 1 + \frac{n}{2^2}\times 2 + \frac{n}{2^3}\times 3 \cdots + \frac{n}{2^{\log n}}\times \log n$$

(AP series)

$$\Rightarrow S = n\left[\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots + \frac{\log n}{2^{\log n}}\right] \quad (1)\times\frac{1}{2}$$

Multiply eq (1) by 1/2

$$\Rightarrow \frac{S}{2} = n\left[\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \frac{4}{2^5} + \cdots + \frac{\log n - 1}{2^{\log n}} + \frac{\log n}{2^{\log n+1}}\right] \quad (2)$$

Subtract (2) from (1)

$$\frac{S}{2} = n\left[\left[\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}+\cdots+\frac{1}{2^{\log n}}\right]-\frac{\log n}{2^{\log n+1}}\right]$$

Use GP formula $r<1$ → $\left[S=\frac{a\cdot(1-r^n)}{1-r}\right]$

$$\frac{S}{2} = n\left[\left(\frac{\frac{1}{2}\cdot\left(1-\frac{1}{2^{\log n}}\right)}{1/2}\right)-\frac{\log n}{2^{\log n+1}}\right]$$

$$\Rightarrow \frac{S}{2} = n\left[\left(\frac{2^{\log n}-1}{2^{\log n}}\right)-\frac{\log n}{2^{\log n+1}}\right]$$

$2^{\log n} \Rightarrow n^{\log 2} \Rightarrow n^1 = 1$

$$\frac{S}{2} = n\left(\left[\frac{n-1}{n}\right]-\frac{\log n}{n\cdot 2}\right)$$

$$= \frac{n(n-1)}{n} - \frac{n(\log n)}{n\cdot 2}$$

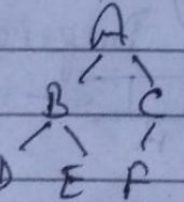
$$\frac{S}{2} = (n-1) - \frac{\log n}{2} \Rightarrow S = (2n-2) - \frac{n\log n}{n}$$

$$\Rightarrow S = 2n-2-\log n \Rightarrow \underline{O(n)} \quad [\because n > \log n]$$

(4) (20) Aug **Threaded Binary Tree**

Special pointer → thread, points to nodes higher in tree.

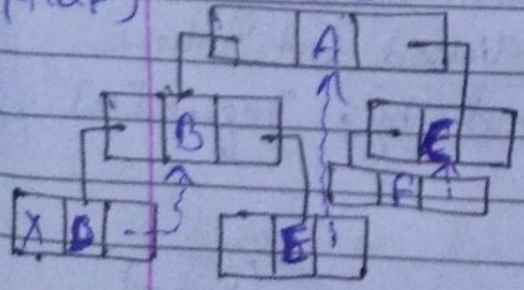
Inorder:



⇒ DBEAFC

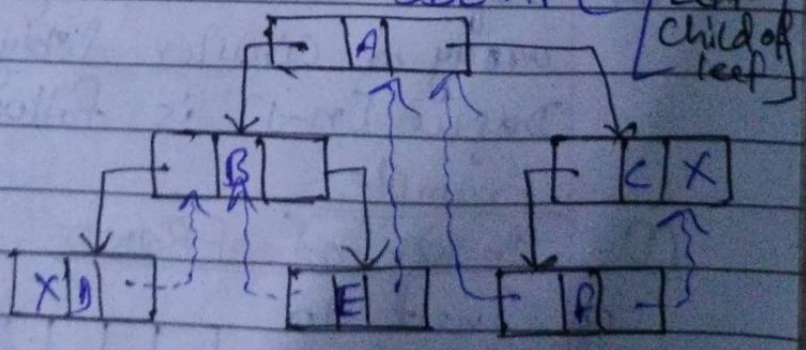
Two types of threaded binary trees:

(1) one way threading (Right child of leaf) → DBEAFC



(2) Two way threading.

DBEAFC [Left child of leaf]



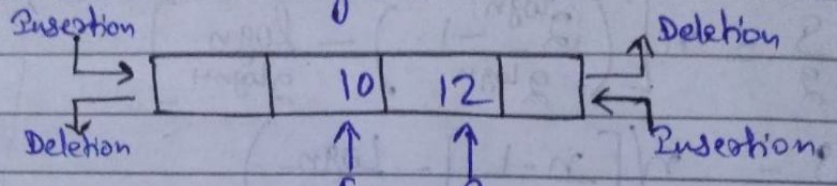
(+) Used to Reduce Null pointers in leaf nodes.

✱ Unit 2 (Left part)

Date :
Page No. :

Deque (Double Ended Queue)

Deque or double ended queue is a type of queue in which insertion and deletion can be perform from both the end. i.e. FRONT or REAR. It does not follow FIFO Rule.



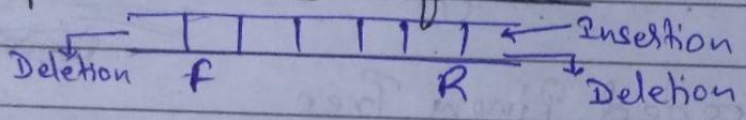
Also known as "Head Tail Linked List"

Operations

1. Insert at front
2. Insert at Rear
3. Delete from Front
4. Delete from Rear.

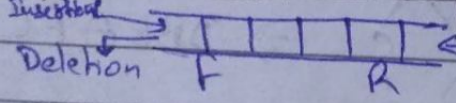
Types of Deque

1. Input Restricted Deque



Front Side → Only Deletion, Rear Side → Insertion Deletion.

2. Output Restricted Deque



Front Side → Insertion & Deletion. Rear side → Insertion only.

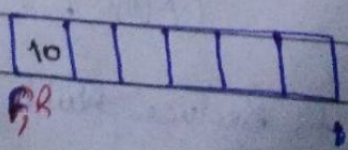
⇒ Deque can be implemented using circular array or circular doubly linked list whose Deque [n-1] is followed by Deque [0]

Example

1. Insert at FRONT

(a) Insert 10

f=0
R=0

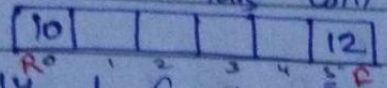


(F--)

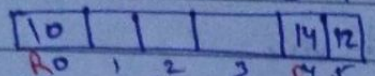
when F>R → Empty Queue.
then we have to reset F = R = -1

* if we use a queue with F>R, it will be wastage of space.

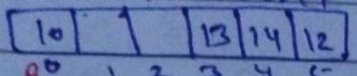
(b) Insert 12. this will be inserted at (n-1) location.
n=6 ⇒ n-1=5



(c) Insert 14 at front.
location = 5-1 = 4.

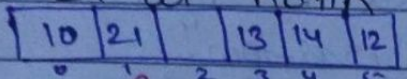


(d) Insert 13 at front. location = 4-1 = 3

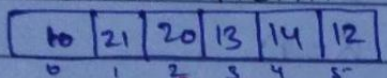


(2) Insert from REAR.

(e) Insert 21 at REAR (R++) location = R+1 ⇒ 0+1=1

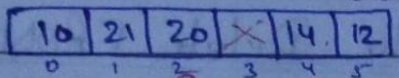


(f) Insert 20 at REAR R = R+1 = 1+1 = 2

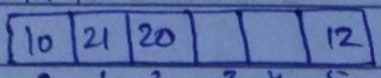


(3) Delete from FRONT. (F++)

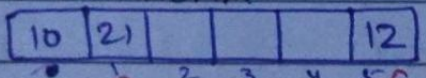
(g) Delete FRONT location ⇒ F = F+1 ⇒ 3+1 = 4



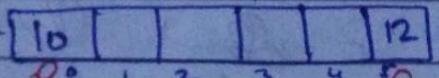
(h) Delete FRONT



(4) Delete from REAR. (R--)



(i) Delete from REAR



Algorithm for Insert from FRONT

(1) Procedure - Begin.
(2) IF F=0 and R=N-1
write "overflow"

③ If f = -1 set f = R = 0
Else if f = 0
Set f = N-1
Else, f = f-1
④ Q[f] = item.
⑤ Procedure END

Algorithm for Insert REAR.
① If f = 0 and R = N-1
write "overflow"
② If f = -1 set f = R = 0.
else, if R = N-1
Set R = 0
Else, R = R+1
③ Q[R] = item.

Algorithm for Delete FRONT
① If f = -1
write "undesflow"
② If f = R
Set f = -1, R = -1
Else, If f = N-1
Set f = 0
Else, f = f+1
③ If f = 0
Set f = -1

Algorithm for Delete REAR
① If f = -1
write "undesflow"
② If f = R set f = -1, R = -1

else. if R=0 set R=N-1
else.
R=R-1

END

Infix to Postfix $K+L-M*N+(O\wedge P)*W/U/V*T+Q$

^ → R→L
/ * → L→R
+ - → L→R
Associativity.

Ans $KL+MN*-OP\wedge W*U/V/T*+Q+$

Infix to Prefix (Reverse))

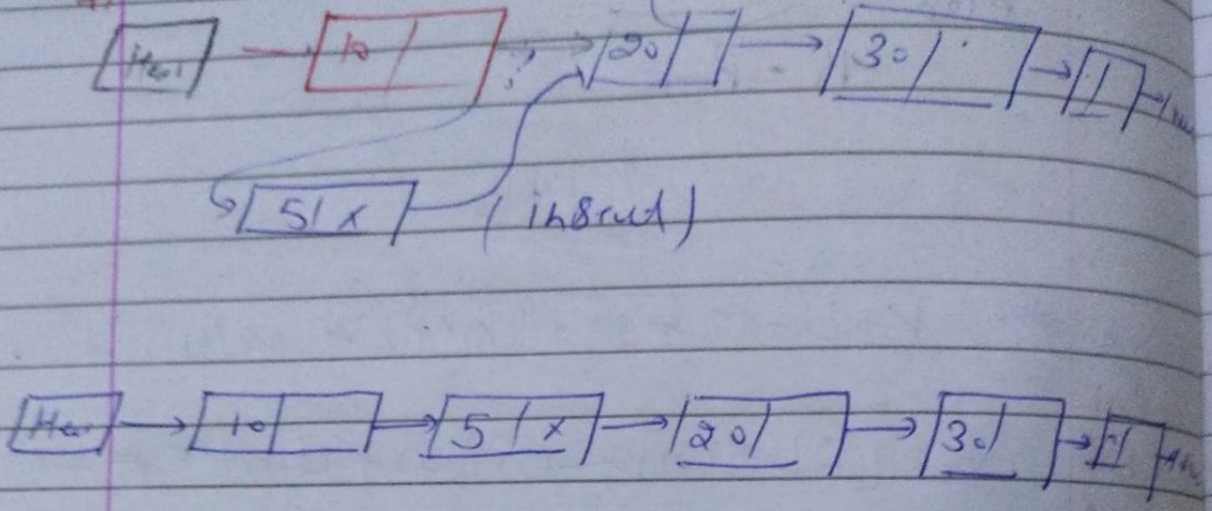
$Q+T*V/U/W*(P\wedge O)+N*M-L+K$

in L→R Associativity Some Precedence then Push like */ +* Confusion

```
+
+
+*
+*/
+*//
+*//*(^)
      POP
+     +*//*
      Pop for +
*     ++*  → Pop
*     ++-
+     ++-+
```

Ans ⇒ QTVUWPO^*//*NM*LK+-++

Reverse → ++-+KL*MN*//*^OPWUVTQ



linearray.py

```
lst = []
n = int(input("Enter how many ele. you want:"))
print("Enter no. in array:")
for i in range(0,n):
    ele = int(input("num:"))
    lst.append(ele)
print("Array:",lst)
```

A*B/C

L→R

/*ABC

A+B/C*(D+E)-F

A+B/C*DE-F

A+/BC*DE-F

A+*/BCDE-F

-+A*/BCDEF

A*(B+D)/E-F*(G+H/K)

A* BD+/E-F*GHK/+

A* BD+E/-F*GHK/+

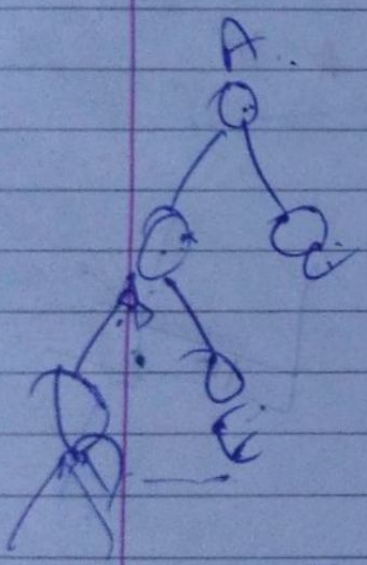
ABD+E/*-FGHK/+*

ABD+E/*FGHK/+*-

-/*A+BDE*F+G/HK

ABC

ABDEC



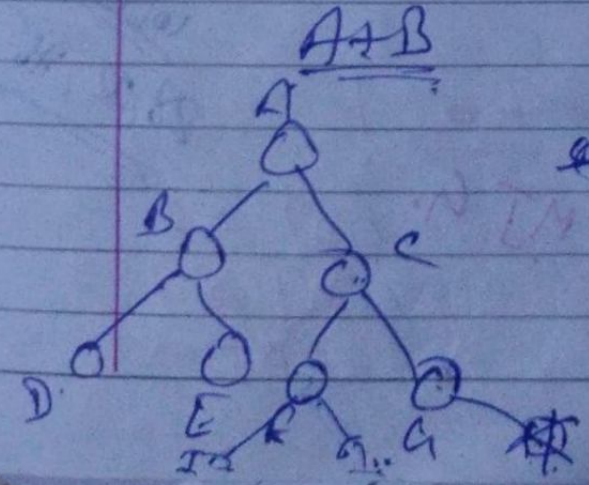
⇒ BST (insert)

⇒ AVL (insertion).

⇒ Construct tree ~~from~~ ~~postfix~~ ~~set~~ tree using In-Pre.

code for beginning of list (sign)

BD E



A+B

DBEAFCG