Unit - 3

Trees

Trees :

- Basic Terminologies
- Binary trees
- Types of Binary tree.
- Binary tree representation using array and linked list.
- creation of a binary tree
- Binary search tree. (BST)
- creation of binary search tree.
- Tree traversal (recursive & non-recursive)
- finding height and counting leaf nodes of BST (with & without recurrive)
- Deletion of node from a BST.
- Finding mirror image of the BST with & without recurrion.
- printing a tree level wise.

Data structure

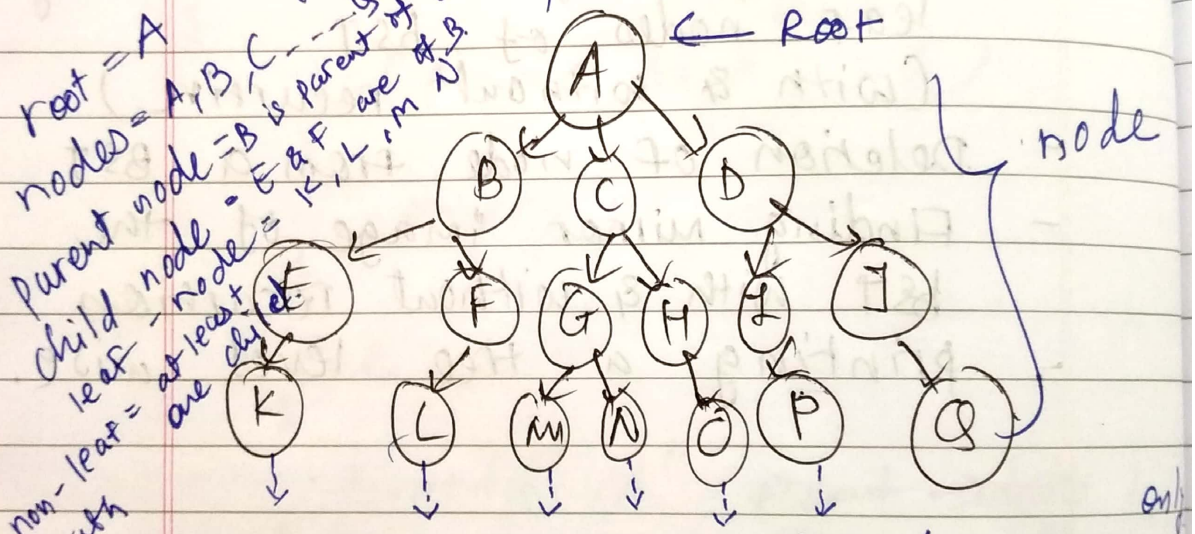
linear
- array
- Linked list
- Stack
- Queue

- single level
- seq.

non-linear
- tree
- Graph

- multiple level
- non-seq.
- hierarchy
- level wise

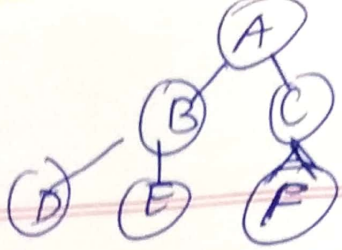
⇒ Logical & representation of tree.



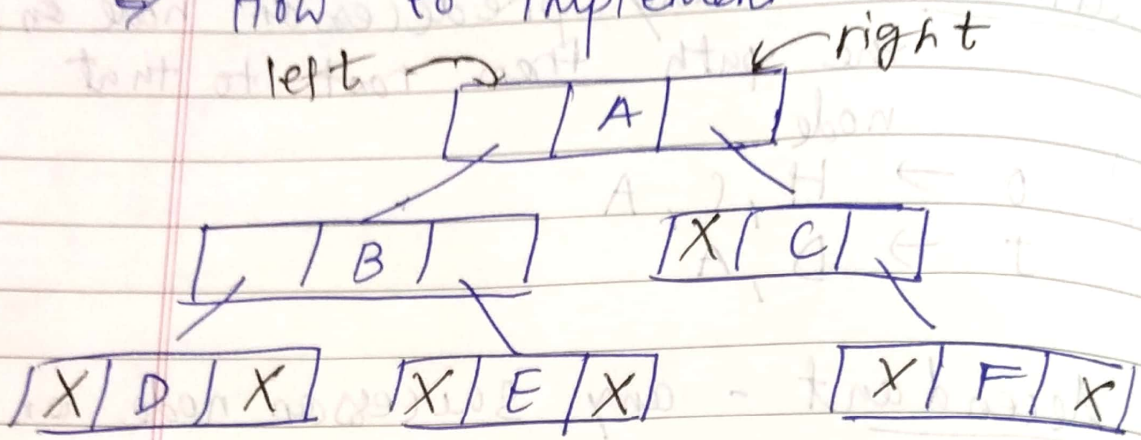
- You can go from Top to Bottom only

Root → Branches → leaves.

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



⇒ How to implement.




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

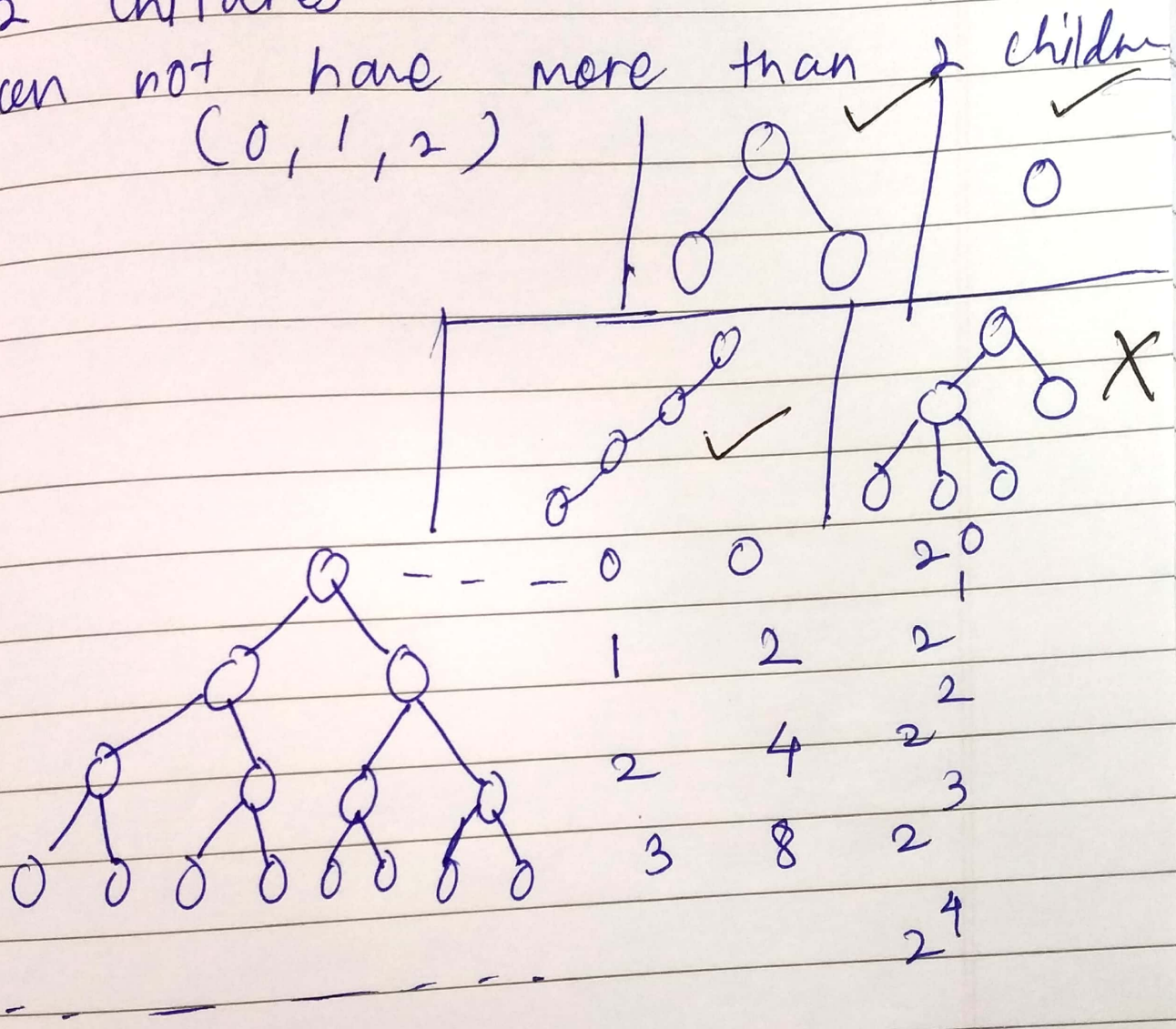
Application of tree:

- File system (hierchical)
- Routing protocal.

organize data for Quick search (insert & delete)

* Binary tree & its types.

- each node can have <u>at most</u> 2 childres.
- can not have more than 2 childr
  (0, 1, 2)



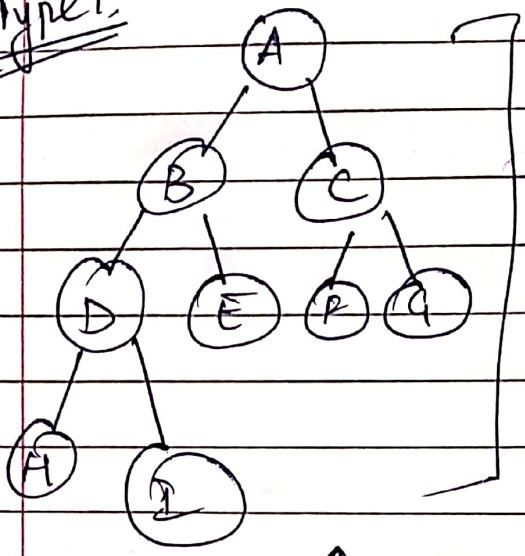
- Maximum number of possible node at any level i is 2^i.

PAGE NO.:

Creating of Binary Tree using array.

Array Representation of Binary tree

Type 1:



Pictorial representation of a tree.

Case i)

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

if a node is in i^{th} index

⇒ left child would be at

$$[(2 * i) + 1]$$

⇒ Right child would be at

$$[(2 * i) + 2]$$

Parent would be at $\left[\frac{(i-1)}{2}\right]$

Note: type 1 formula can be used for type 2 as well with index starting at zero.

Case 2:

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

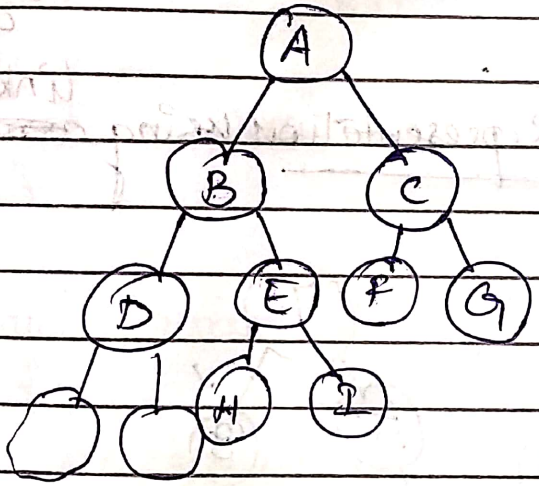
node is i^{th} index.

left child = $2 * i$

right child = $(2 * i) + 1$

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

Type 2:

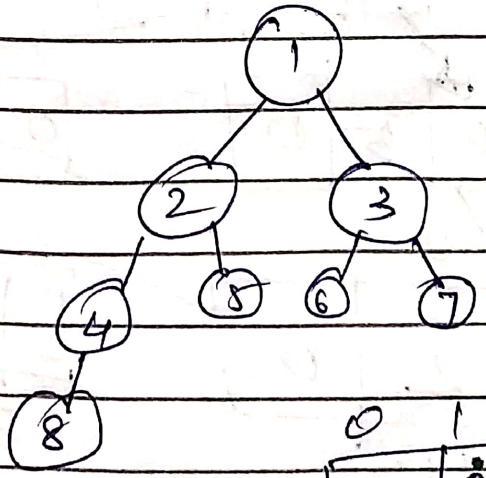


always the tree shd be binary tree.

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

~~0 1 2 3 4 5 6 7 8 9 10~~

Type 3:

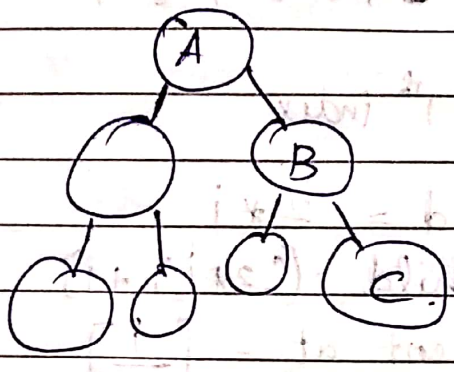


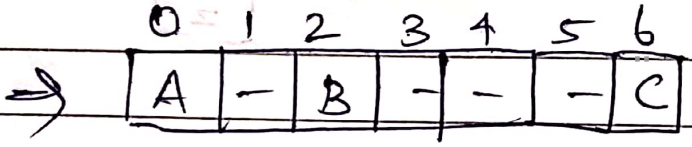
this is also CBT (complete binary tree)

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

PAGE NO.:

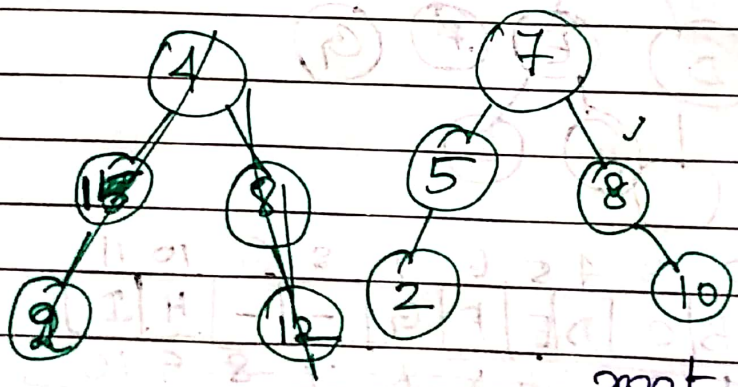
Type 4:

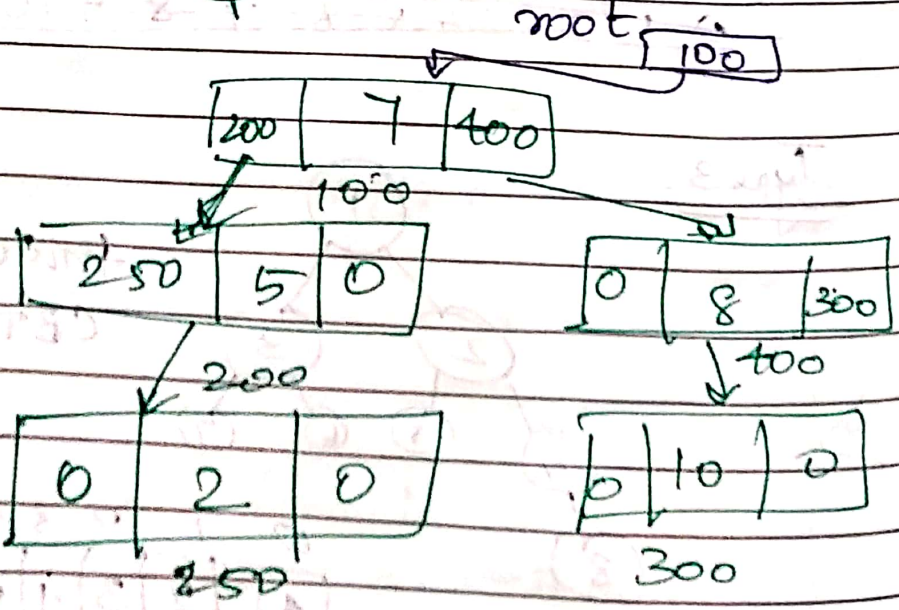




representation of tree in array

Binary Representation using linked list





Creation of Binary tree using Linked List

PAGE NO.:

```
struct node
  {
  int data;
  struct node *left,*right;
  };

void main()
 {
 struct node *root;
 root =0;
 root = create();
 }

struct node *create()
 {
 int element;
 struct node *newnode;
 newnode = malloc(struct node*)malloc
                        (size of(struct node));
 printf("Enter the data element of a node:");
 scanf("%d", &element);
 if(element == -1)
                        ↳ 0
   { return 0;//NULL
   }
 newnode->data = element;
// print Enter the left child
 newnode -> left = create()
 // get right child
```

return newnode.

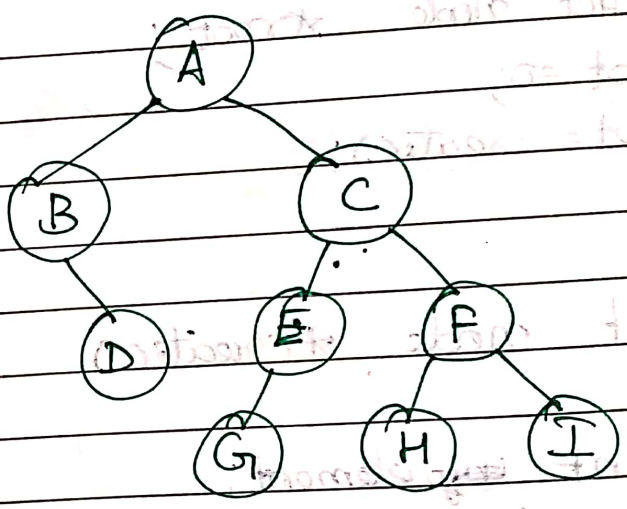
}

PAGE NO.:

Tree Traversal of Binary tree

3 types

- Inorder L R R•
- Preorder R• L R
- Postorder L R R•



Inorder: B D A G E C H F I

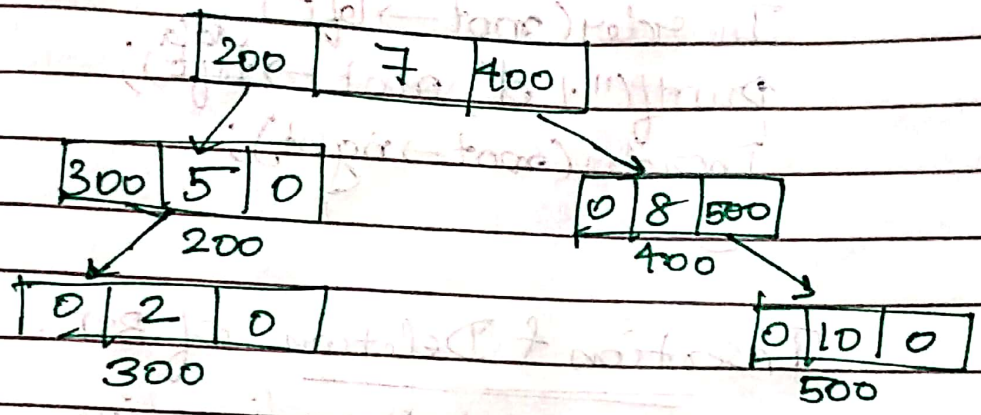
Preorder: A B D C E G F H I

Postorder: D B G E H I F C A

Implementation of tree traversal (Recursive)

Preorder:

7 5 2 8 10




```
Void main()
{
struct node *root;
// Preorder
Preorder(root);
}

void Preorder(struct node *root)
{ if(root==0)
    return;
else
{
Printf("%d", root->data);
Preorder(root->left); // left subtree
Preorder(root->right); // right subtree
}}

void Inorder(struct node *root)
{ if(root==0)
    return 0;
else
```

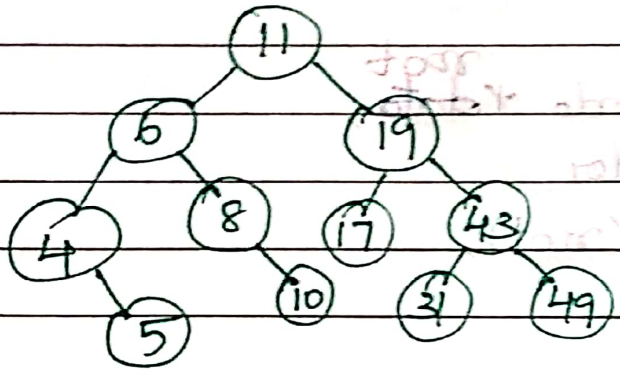
Preorder(root->

Inorder(root->left);
printf("%d", root->data);
Inorder(root->right);
}

Insertion & Deletion of BST.

L to R ⇒ direction

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



Deletion:

(A) ⇒ node that needed to be deleted has

1. 0 child
2. 1 child
3. 2 child

(1) ⇒ if I need to delete '31' just delete the node (free the node)

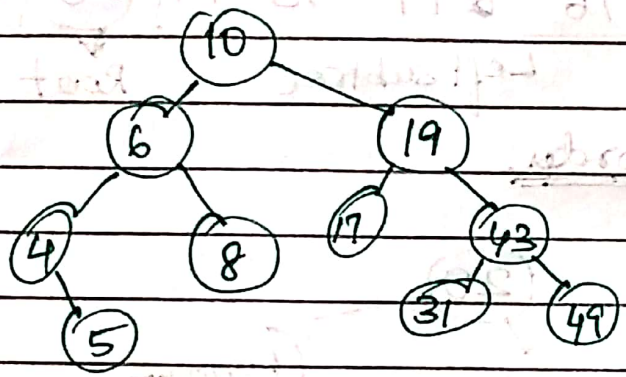
(2) '4' node to be deleted, it will replaced with its child node '5' either left or right.

Inorder: 4,5,6,8,10,11,17,19,31,43,49.

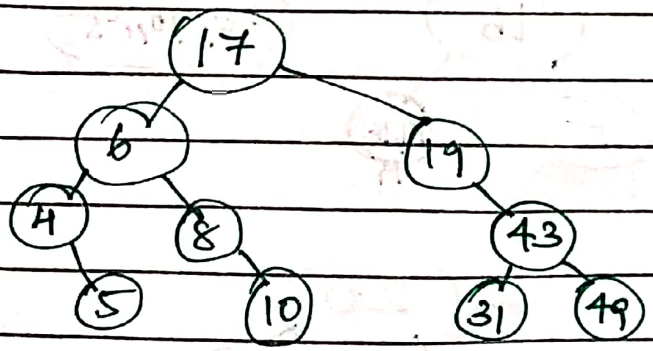
(3) (i) inorder predecessor.
(ii) inorder successor.

Inorder: Predecessor (largest element from the left subtree) → before element of the node that needs to be deleted.

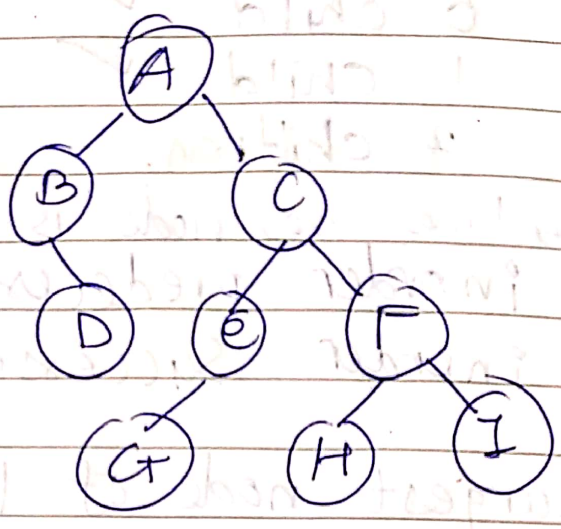
'11' will be replaced with (10).



Inorder Successor (Smallest element of the right subtree / the next element of deleted node).



* Binary tree traversal.



Inorder : Left Root Right
preorder : Root Left Right
postorder : Left Right Root

Inorder: B D A G E C H F I

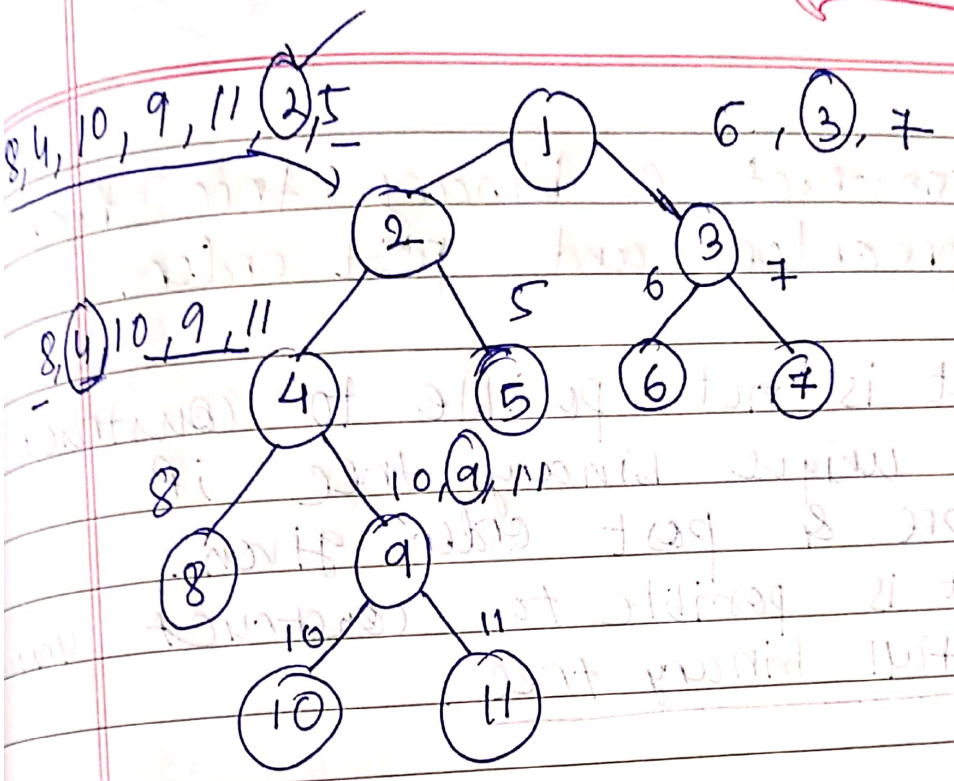
preorder: A B D C E G F H I

postorder: D B G E H I F C A

* Construct a Binary tree from preorder & Inorder

preorder: 1 2 4 8 9 10 11 5 3 6 7 (Root L, R)
Inorder: 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 (L, Root, R)

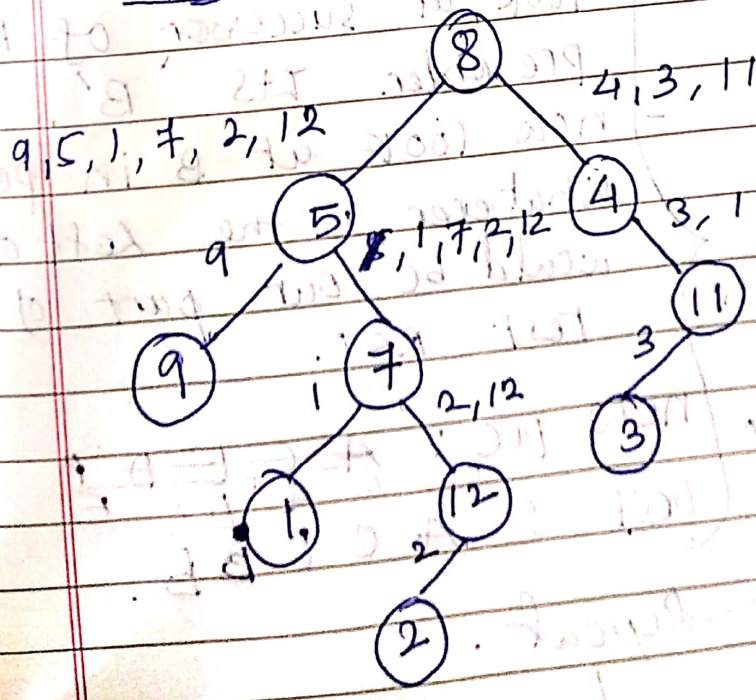
8, 4, 10, 9, 11, 2, 5 — Left sub tree
1 — Root
6, 3, 7 — Right sub tree.



* construct a Binary tree from ~~pre order~~ & postorder & Inorder

Post order: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8 (L, Right, Root)

Inorder: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11 (L, Root, Right)

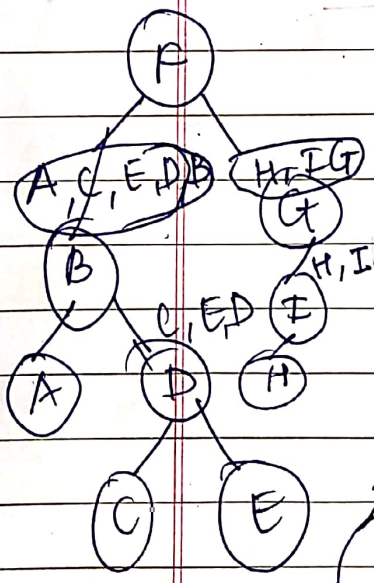


* construct a binary tree from preorder and post order.

- It is not possible to construct a unique binary tree if pre & post order given.
- it is possible to construct unique full binary tree.

pre: F B A D C E G I H (Root L R)
post: A C E D B H I G F (L R Root)

F is ROOT



but to find LST look at successor of F in pre order. Its 'B'.
- now look at B in post order whatever is the left of B. would be our part of LST. rest RST.

new pre: ~~A C E D B~~ B A D C E
post : A C E B B.

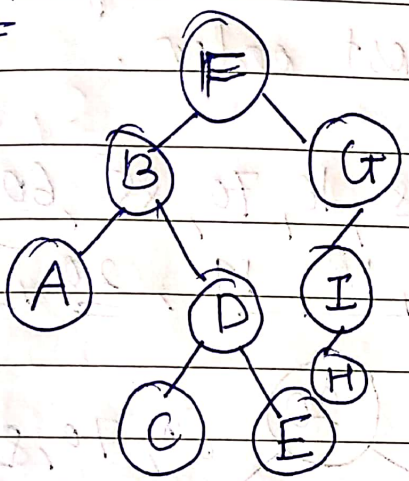
Repeat.

post : C E D.
pre : D C E.

pre : G I H
post : ~~I G H~~ H I G

~~pre~~: I H
~~post~~ H I

Method 2:

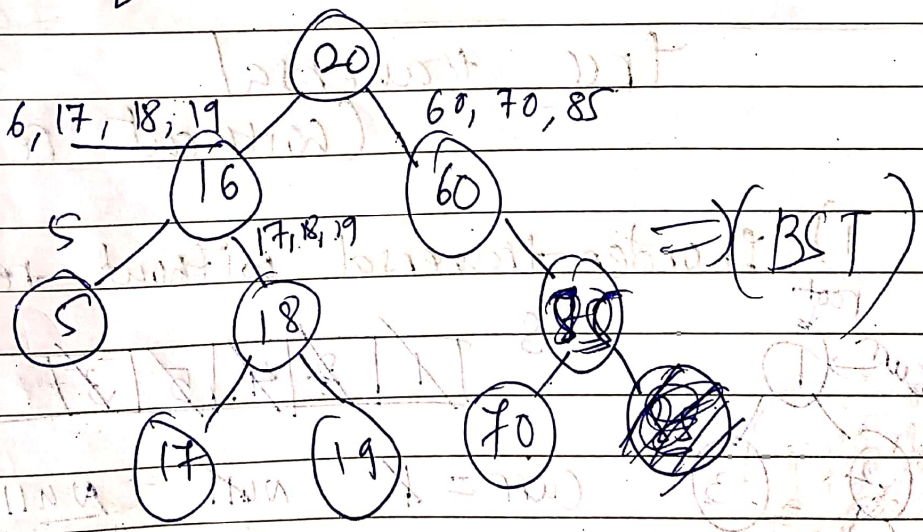


Every time look at Left of Seq. node from Preorder

* Construct a binary search tree (BST) from preorder

preorder : 20, 16, 5, 18, 17, 19, 60, 85, 70 (Root L R)

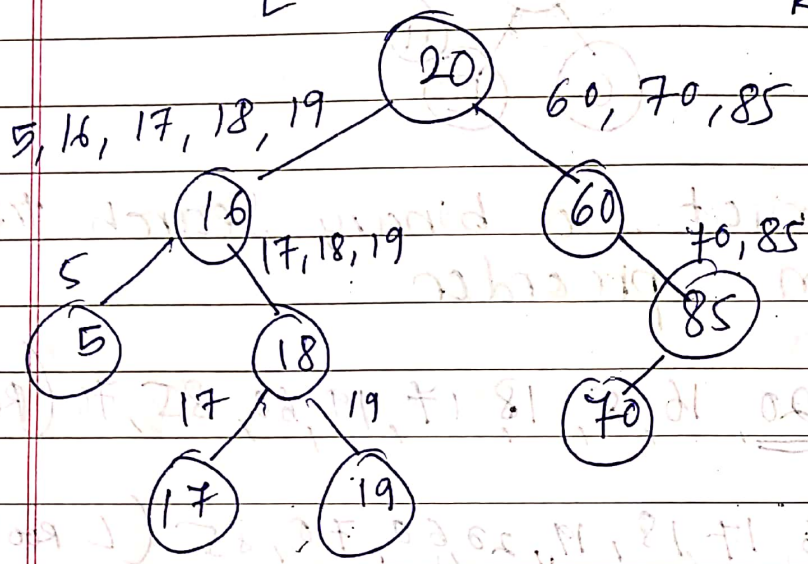
inorder : 5, 16, 17, 18, 19, 20, 60, 70, 85 (L, Root R)
L R



* construct a Binary Search tree (BST) from post order.

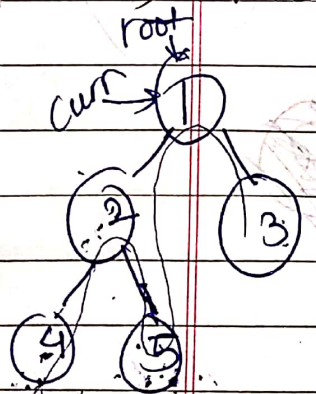
post order : 5, 17, 19, 18, 16, 70, 85, 60, 20 (L R Root)

Inorder : 5, 16, 17, 18, 19, 20, 60, 70, 85 (L Root R)



Tree traversal (without recursion)

1) Inorder traversal without recursion.



S

1	2	4	5	3

curr = 1 → null → null

2 → 5

4 → null → null

null → 3

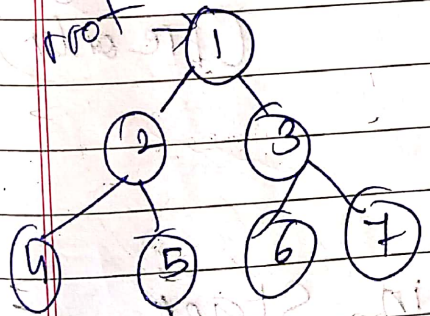
Stopping condition: stack is empty && curr == null

o/p: 4 2 5 1 3

Steps: (Algo):

1. Create an Empty stack.
2. Initialize current node to root.
3. push current node to stack. & set (curr = curr → left) until curr is NULL.
4. If curr is NULL & stack is not Empty
   i) pop top [stack] & print it.
   ii) set curr = popped_node → right
   iii) Go to Step (3).
5. If (curr == NULL && stack == Empty) then we are done.

2) post order without recursion.



S_1 | 1 | 2 | 3 | 6 | 7 | 4 | 8

S_2 | 1 | 3 | 7 | 6 | 2 | 5 | 4

while (S_1 != NULL)

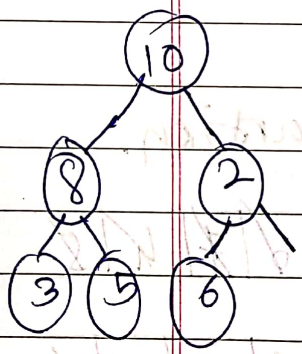
popped node = 1 2 3 5 7 4 6

o/p: 4 5 2 6 7 3 1

Algo (step):

1. push root to S_1.
2. Loop while S_1 is not empty.
   i) pop top node of S_1 & push to S_2.
   ii) push { node → left, node → right } to S_1.
3. print content of S_2. (postorder)

3) preorder traversal without recursion



S_1: 10 | 8 | 3 | 5 | 2 | 6

op: 10 8 3 5 2 6 (preorder)

Algo (step):

1. push the root in Stack.
2. while (Stack is not empty).
   i) pop & store Stack[top].
   ii) print it.
   iii) push right
   iv) push left

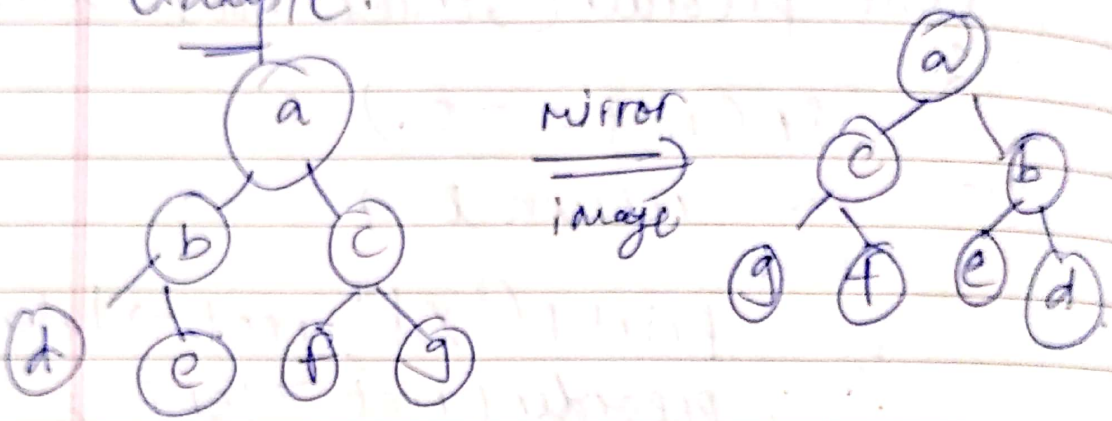
* Implementation of tree traversal (recursive).

```
void preorder(struct node *root)
{
    if(root == 0)
        return 0;
    else
        printf("%d", root->data);
        preorder(root->left);
        preorder(root->right);
}

void Inorder(struct node *root)
{
    if(root == 0)
        return 0;
    else
        Inorder(root->left);
        printf("%d", root->data);
        Inorder(root->right);
}
```

* Finding Mirror image of Binary Search tree (BST)

Example:



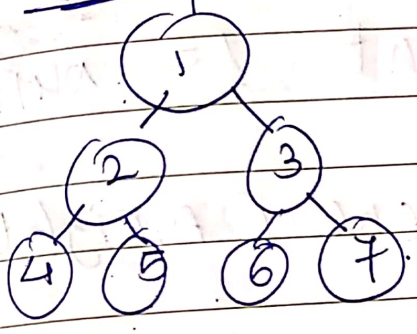

```
void mirror (node *root)
{
    if (root)
    {
        mirror (root → left);
        mirror (root → right);
        node *temp = root → left;
        root → left = root → right
        root → right = temp;
    }
    return;
}
```

with recursion

⇒ without recursion.

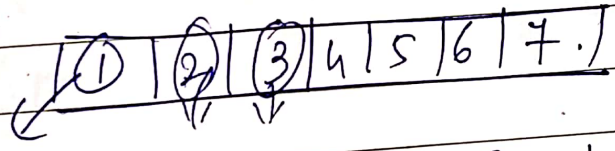
* printing tree level wise (recursive).

Example:



1 : 2, 3
2 : 4, 5
3 : 6, 7
4 :
5 :
6 :
7 :

Queue:

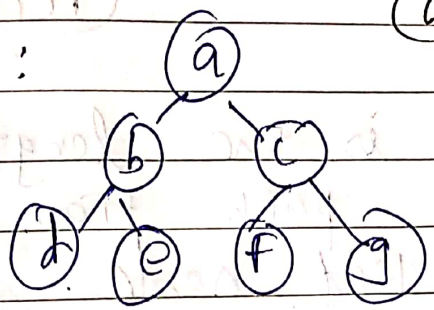



```
void printTreeLevelwise (node root)
{
    if (root == NULL)
        return ;
    Queue <node> q
    q.enqueue (root)
    while (q.size () > 0 )
    {
        node f = q.front ( );
        q.deque ( )
        print (f.data);
```

```
if(f.left != null)
{
    q.enqueue(f.left)
}
if(f.right != null)
{
    q.enqueue(f.right)
}
}
```

* Counting leaf nodes of BST (with recursion)

Example:




```
int count = 0;

void count_leaves (node *root)
{
   if(root)
   {
      if (root -> left == NULL &&
          root -> right == NULL)
      {
         count ++ ;
      }
      else
      {
         if ( root -> left)
            count_leaves (root -> left)
         if (root -> right)
            count_leaves (root -> right)
      }
   }
}
```

* finding height of BST. (recursive).

⇒ height is the length of longest path from root node to a leaf node.

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

int height(struct node *node)
{
    if (node == NULL)
        return 0;
    else
    {
        int left_height = height(node->left);
        int right_height = height(node->right);

        if (left_height > right_height)
            return (left_height + 1);
        else
            return (right_height + 1);
    }
}
```

* finding height of BST (non-recq)

```
function treeHeight(root)
    if root == NULL
        return 0;
    queue = createQueue() //emptyqueue
    height = 0;

    enqueue(queue, root)
    while true:
        node_count = size of queue
        if node_count is 0
            return height
        increment height by 1
        while node count > 0
            if node has a left child
                enqueue(queue, right child of a node)
            decrement node count by 1.
```

(1) no. of nodes present in the queue represent no. of nodes i/e present current level of the tree.

traversal is finished its moving to next level of the tree.