Unit I - Tree

Basic Terminology

A tree is a finite set of one or more nodes such that.

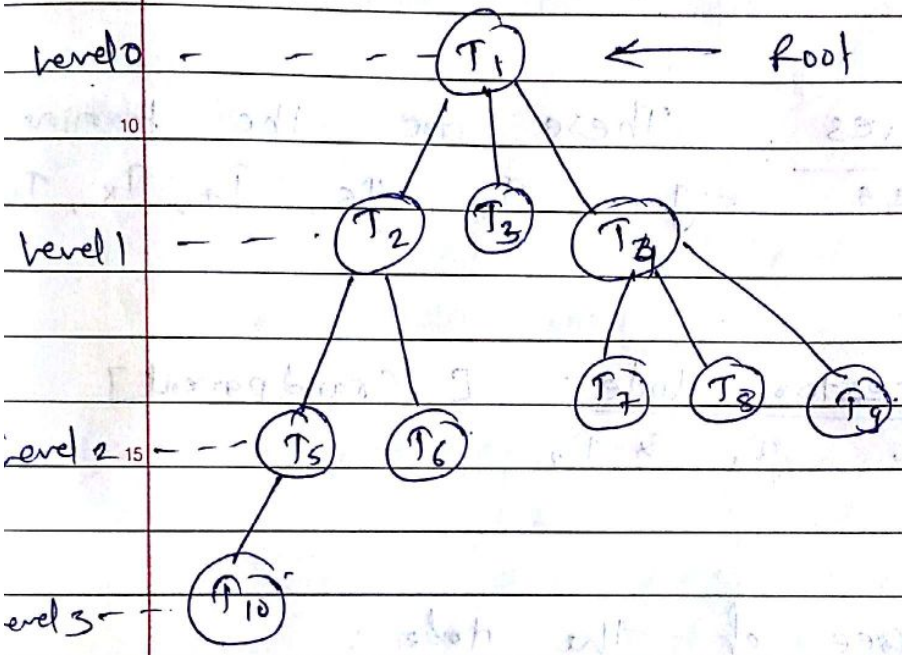


Fig: Tree

1) **Root**: Root is a unique node in the tree to which further subtrees are attached.
e.g. T_1 is Root Node

2) **Parent Node**:

The node having further sub-branches is called parent

Date / /

node. e.g. T_2 & T_4 & T_5

3. Child nodes :

The child nodes are T_2, T_4 T_5, T_6 etc.

4. Leaves nodes : These are the terminal nodes e.g. T_{10}, T_6, T_7, T_8,

5. Ancestor Node : [Grandparent]
e.g. T_1 & T_2.

6. Degree of the Node :

The total number of sub-trees attached to that node is called the degree of a node. or total no. of childrens is degree of that node.
e.g. T_1 degree is 3.

7. Degree of Tree :

The maximum degree in the

tree is degree of tree.
e.g. 3.

8. Level of the tree :

The root node is always considered at level zero.

9. Hight of the tree :

The maximum level is the hight of tree. e.g. 3.

10. predecessor :

While displaying the tree, if some particular node occurs previous to some other node then that node is called predecessor of the other node.

e.g. T_1 is predecessor of T_2

11. Successor :

Successor is a node which occurs next to some node.
eg. T_2 is successor of T_1.

12) Internal & external Nodes.

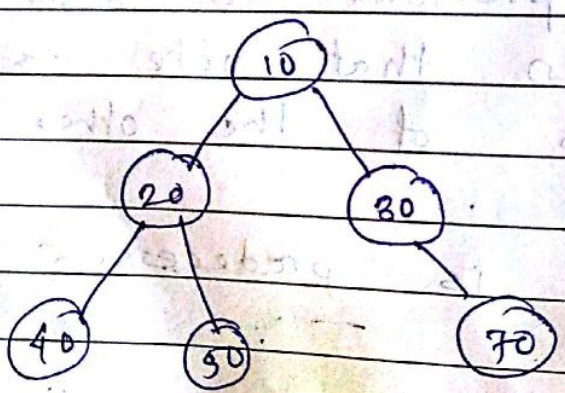
leaf nodes are known as external nodes & non-leaf Nodes are called internal nodes

13) Sibling:

The nodes with common parent are called binary trees.
eg. T2, T3, T4.

* Types of Trees:

1) Binary Tree:



A binary tree is a finite set of nodes which is either empty or consists of a root & two disjoint binary trees called its

left sub-tree & right sub-tree.

e) Complete Binary tree :

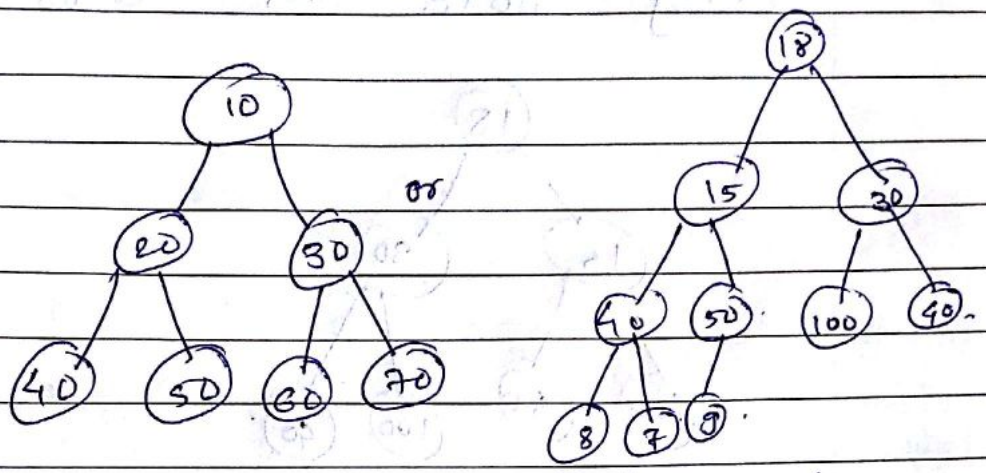


fig : Complete binary Tree.

[If all levels are completely filled except possibly the last level & last level has all keys as left as possible]

- level i has 2^i nodes.
- In a tree of height h
  - leaves are at level h.
  - No. of leaves is 2^h.
  - No. of internal nodes = $2^h - 1$
  - No. of internal Nodes = No. of leaves $- 1$
  - Total no. of nodes $2^{h+1} - 1 = n$

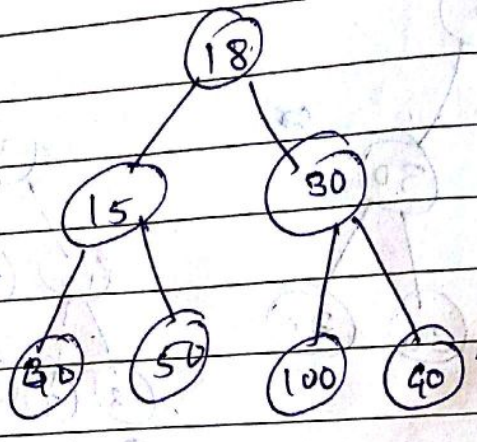
$\Rightarrow$ Tot. internal Nodes + No. of leaves

$2^h - 1 + 2^h$

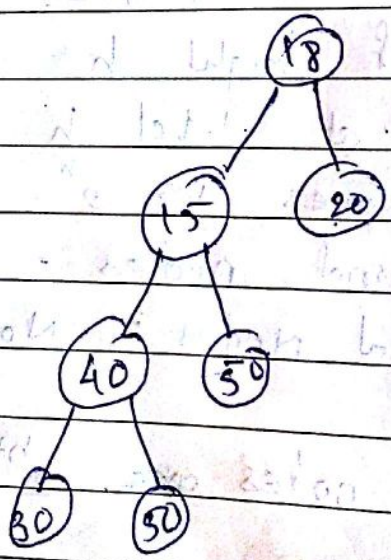
$= 2^{h+1} - 1$

3) Full Binary Tree :

A Binary tree is full if every node has 0 or 2 d



or



In a full binary tree, No. of leaf nodes is number of internal nodes plus 1

$L = I + 1$

L = No. of leaf Nodes

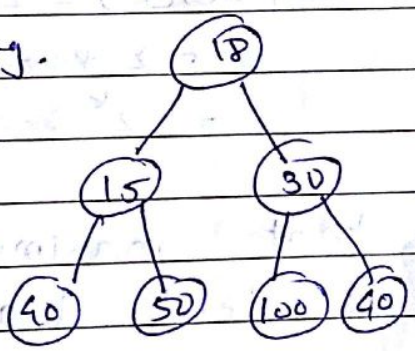
I = No. of

So a full Binary tree can be a complete binary tree but a complete binary tree not necessary to be full binary tree

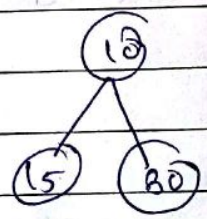
4) Perfect Binary Tree :

A Binary tree is perfect binary tree in which all internal nodes have 2 children & all leaves are at the same level.

eg.

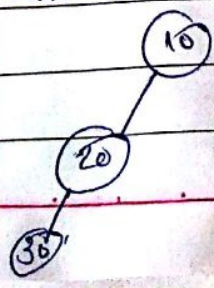


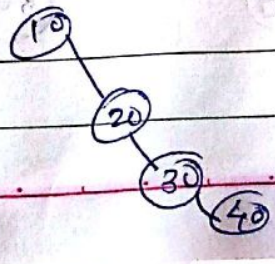
Or



5) Left & Right Skewed Trees.

The tree in which each node is attached as a left child of parent node then it is left skewed tree





Properties of Binary Tree

i) For a binary tree maximum number of nodes at level are 2^L.

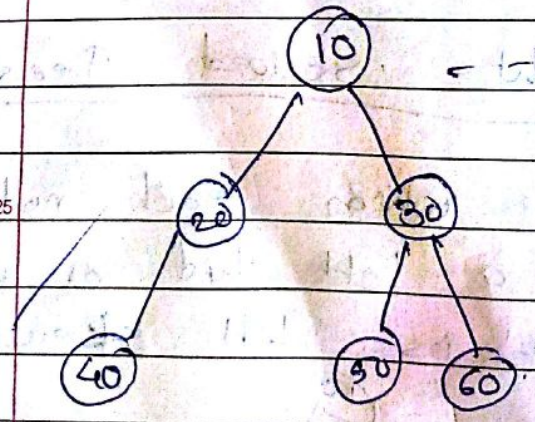
→ at level 0, --- max. nodes(0) = 1

at 1, --- max.nodes(1) = 2 * max.no
= 2 * 1
= 2.

at 2, --- max.nodes(2) = 2 * max no
= 2 * 2
= 4 = 2^2

at k --- max.nodes(k) = 2 * max.no
= 2 * --- (2*
= 2^k.

This proves that maximum no of nodes at level L are 2^L



level 0

(ii) A full Binary tree of height h has $(2^{h+1}-1)$ nodes.

Proof :

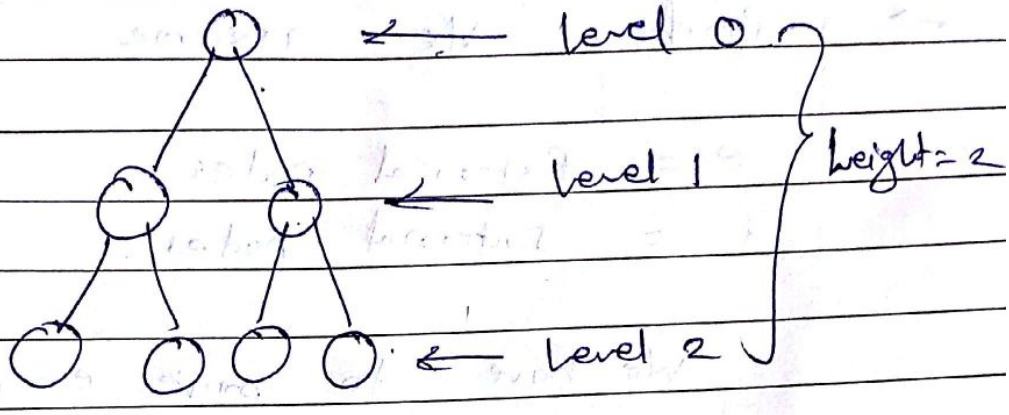


Fig. Full Binary tree.

- At level i has 2^i Nodes.
- Leaves are at level h = 2^h.
- Total No. of internal Nodes
  = $2^h - 1$.

∴ Total No. of Nodes = Total No. of internal Nodes + leaf Nodes

= $2^h - 1 + 2^h$

= $2^{h+1} - 1$ proved.

(iii) Total no. of external nodes in a binary tree are internal nodes +1 i.e. $e = i+1$

→ Proof : We assume

e = External nodes
i = Internal nodes.

∴ We have to prove $e = i+1$

If there is only one node ie. root node

(10) $i = 0$,

$\therefore e = 0+1$

$\therefore e = 1$

∴ only one external node is pres
Thus $e = i+1$ is true

Whenever we add the node to a binary tree we add external nodes & then previous external node becomes an int node.

Again, $i = 1$, $e = 2$

Hence, $e = i+1$

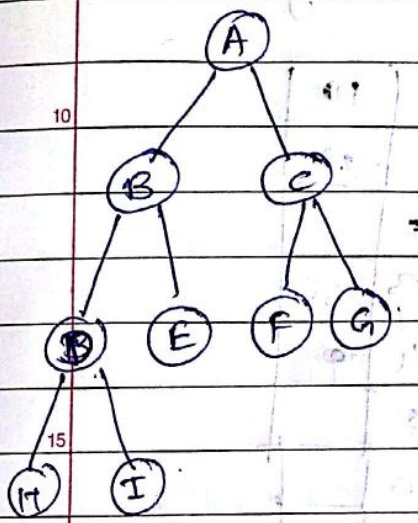
i.e. $2 = 2$ is true.

Date / /

* Representation of Binary Tree:

1. Sequential representation
2. Linked representation

1) Sequential Representation:



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

Root = A = index 0
left child of A i.e. n = 0

Array size >= Total no. of nodes

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

$= 2^4 - 1$

$= 15$

>= 15 ⇐

When n = 0, the root node will placed at 0th location.

Parent (n) = floor[(n-1)/2]

left-(n) = (2n+1)

right (n) = (2n+2)

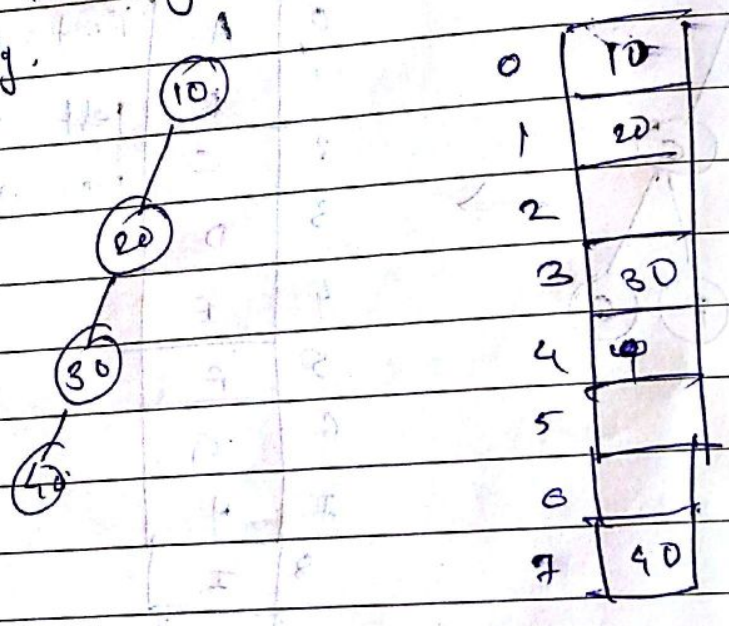
Advantage

1) Direct access to any node can possible & finding the parent, left right child of any particular is fast.

Disadv.

1. Wastage of memory.

e.g.



2. Memory Shortage.

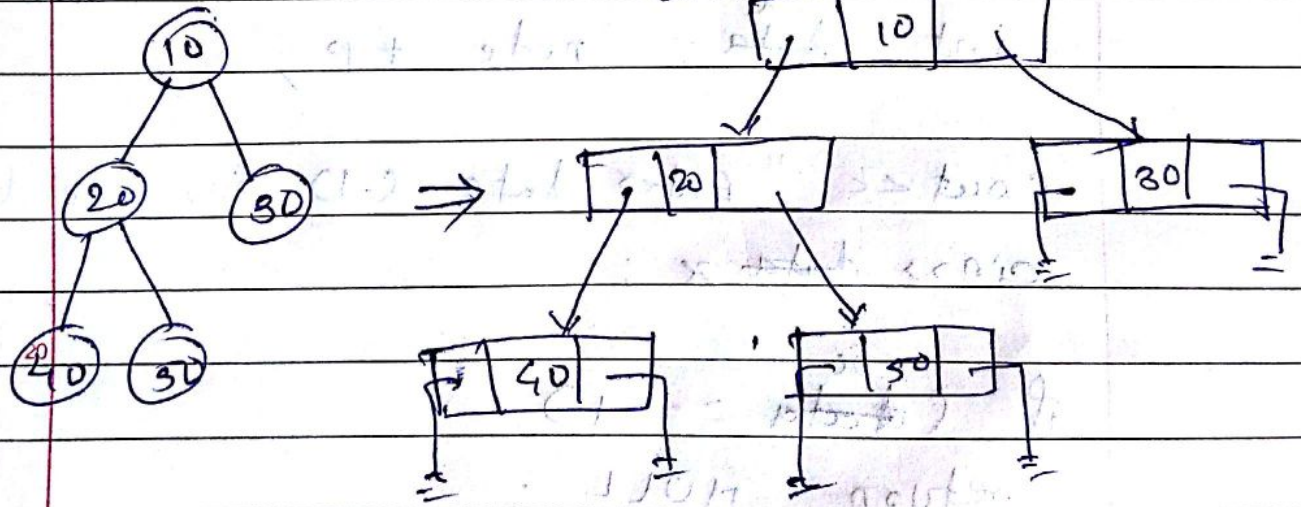
3. Insertions & deletion of any n in a tree will be costlier.

Page
Date / /

Linked representation :

ADT :

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



Operations on Binary Tree:

1) Create()
2) Insert()
3) Delete()
4) Display() → traversing.

1) Create():

```
root = NULL;

node * create()
{
  int x; node *p;

  cout << " Enter data (-1) for no data
  cin >> x;

  if (x == -1)
    return NULL;

  else
  {
    p = new node;
    p->data = x;
    p->left = NULL;
    p->right = NULL;
    if (root == NULL)
      root = p;
```

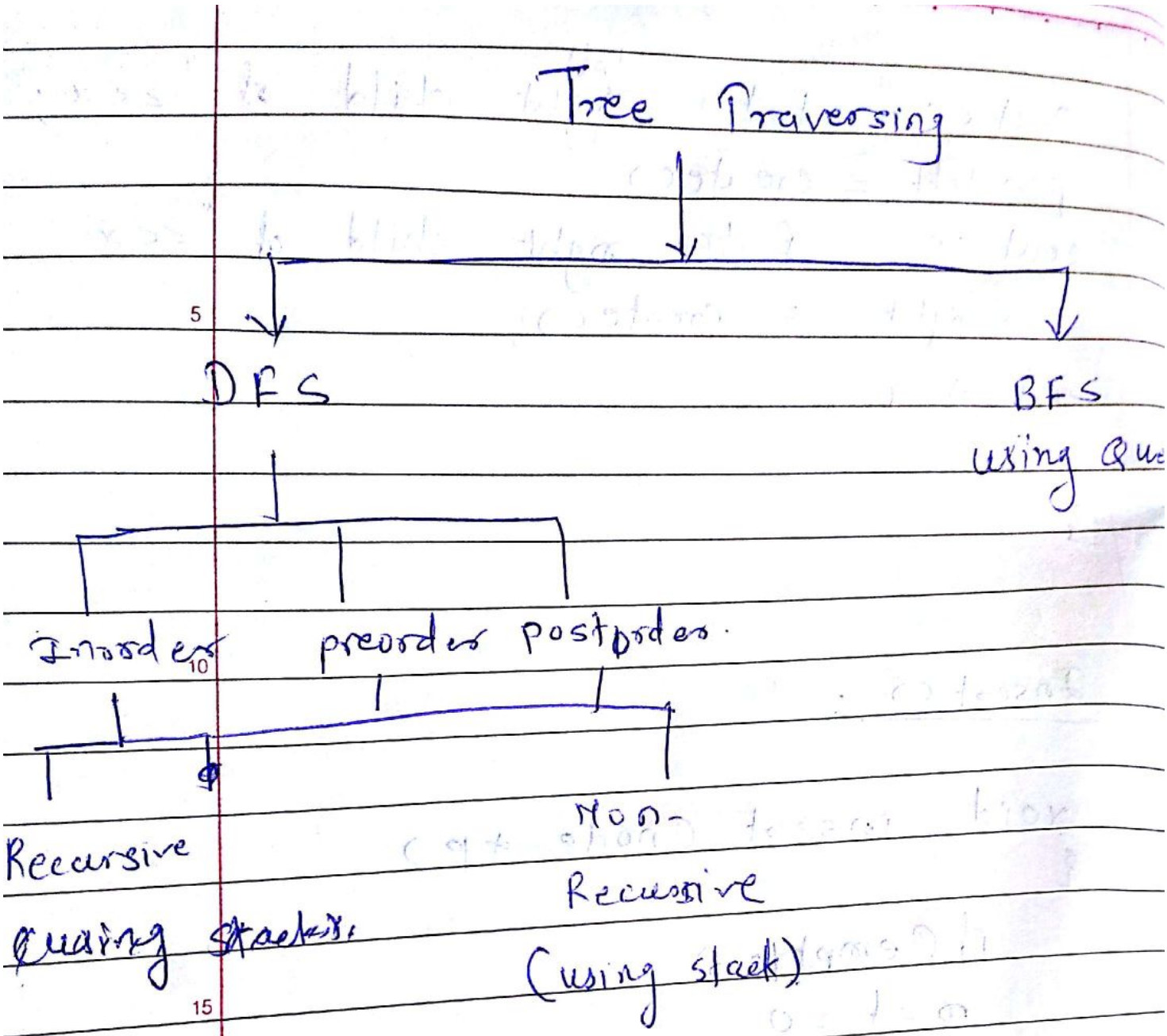
```
cout << " Enter left child of " << x;
p -> left = create();
cout << " Enter right child of " << x;
p -> right = create();
return p;
}
}
```

4) Insert() :

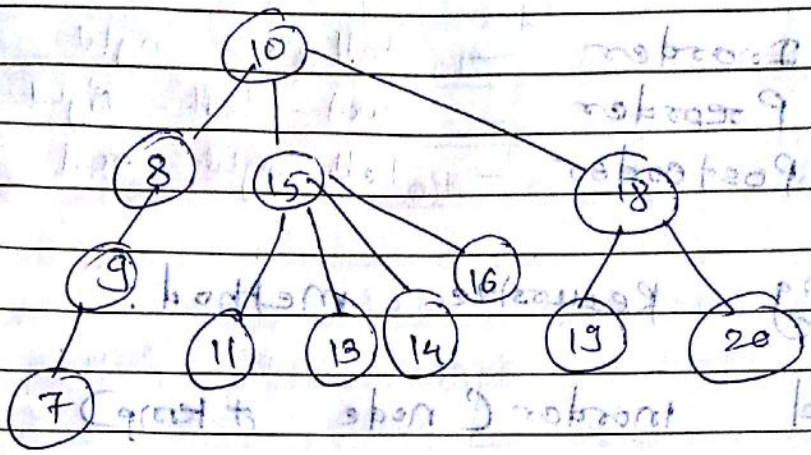
```
void insert (node *p)
{
    if (empty())
    r = f = 0;
    else
      r = r + 1;
    data[r] = p;
}
```

5) Node * delete()

```
{
    node * p;
    p = data[f];
    if (r == f)
      r = f = -1;
    else
      f = f + 1;
} return p;
```



* General Tree and its Representation

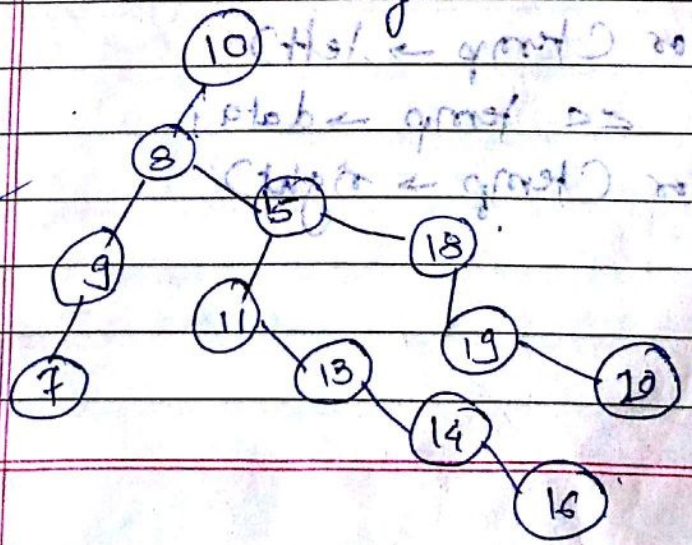


Converting a Tree into a binary Tree

i) The root of a general tree becomes the root of the binary tree.

ii) Find the 1st child of node, attach it as a left child to the current node in a binary tree.

iii) The right sibling can be attached as a right child of that node.



Binary Tree Traversals:

DFS: Depth First Search

i) Inorder — left - root - right
ii) Preorder — root - left - right
iii) Postorder — left - right - root.

Using Recursive method.

```
void inorder (node *temp)
{
    if (temp != NULL)
    {
        inorder (temp->left);
        cout << temp->data;
        inorder (temp->right);
    }
}
```

```
void preorder (node *temp)
{
    if (temp != NULL)
    {
        preorder (temp->left);
        cout << temp->data;
        preorder (temp->right);
    }
}
```

* Recursive postorder traversal:

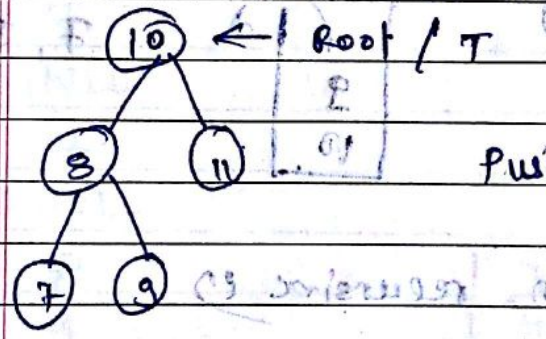
Algorithm:

q/p If tree is not empty then

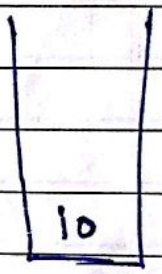
{

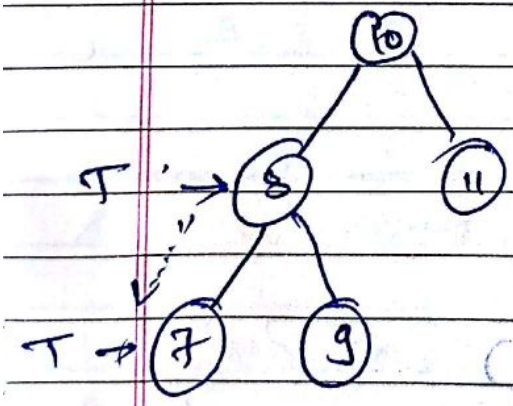
a. traverse the left subtree in postorder.
b. traverse the right subtree in postorder.
c. visit the root node.

* Non - Recursive Traversal

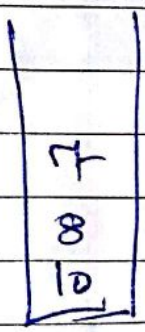


Push 10 move to left branch.



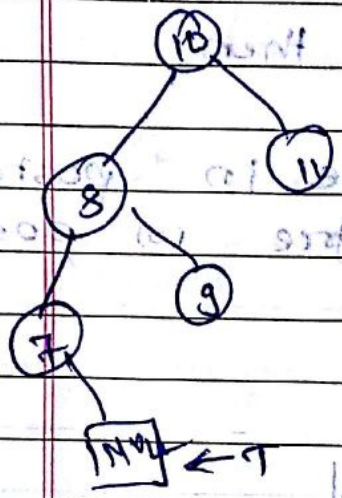


Push 8 to move to left branch

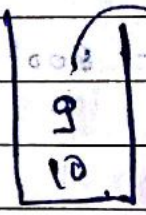


T = NULL

as T = NULL pop topmost elem 7, & then print it & mov right child of 7.



Pop 8 & print 8 & move to right of 8




```
void inorder_non_recursive()
{
    stack s;
    while(T != NULL)
    {
        s.push(T);
        T = T->left;
    }
    while(!s.empty())
    {
        T = s.pop();
        cout << T->data;
        T = T->right;
    }
```

```
    T = T->right;
}
while (T != NULL)
{
    s.push(T);
    T = T->left;
}
}return;
}
```

T	Stack	o/p
10	7	7, 8
8	8	
7	10	
NULL		
7		
NULL		
8	8	
9	10	
NULL		7, 8, 9
9	10	

Non-Recursive postorder traversal Algorithm

1. Push root to first stack
2. Loop while first

* Tree Traversal Examples:

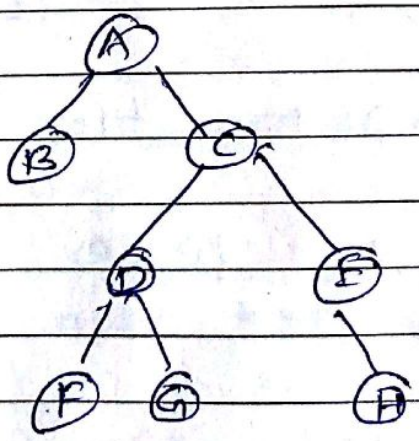
* Binary Tree Traversal (BFS)

- It is a level by level traversal of a binary tree. Within levels, nodes are listed left to right. It is known as breadth first search (BFS) or breadth first traversal (BFT).

- Level by Level traversal on a binary tree is implemented through a queue

- Level by level traversal of the binary tree using a queue is shown below

Tree

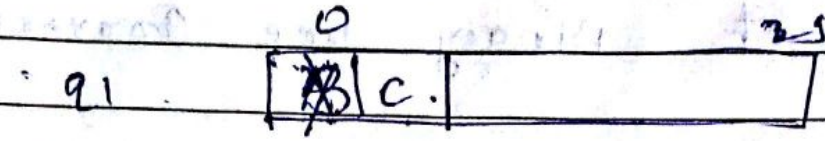


Level	Elements of the Q
0	A
1	BC
2	DE
3	FGH

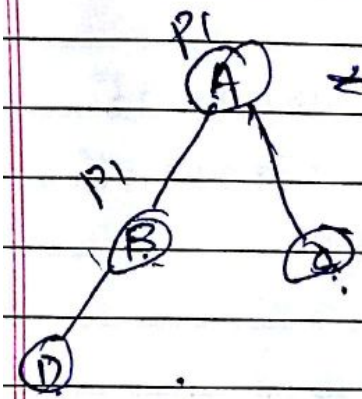
```
// Program for BFS

struct node
{
  int data;
  node *left, *right;
};
class Q
{
  node *data[80];
  int R, F;
public:
  Q()
  {
    R=F=-1;
  }

void init()
{
  R=F=-1;
}

int empty()
{
  if (R==-1)
  return 1;
  else
  return 0;
}
```

```
void insert (node *p)
{
  if (empty())
    R = F = 0;
  else
    R = R + 1;
  data[R] = p;
}

node *delete()
{
  node *p = data[F];

  if (R == F)
    R = F = -1;          } no need
  else
    F = F + 1;
  return (p);
}
};

Void BFT (node *T)
{
  Q q1, q2;
  node *p1, *p2;

  if (T == NULL)
  return;
```

```
q1.insert(T);
cout << "\n" << T->data;
while (!q1.empty())
{
// Replace all nodes of the queue with the nodes of next level store of next level in q2. */

cout << "\n";
q2.init();
while (!q1.empty())
{
p1 = q1.delete();
if (p1->left != NULL)
{
q2.insert(p1->left);
cout << " " << p1->left->data;
}

if (p1->right != NULL)
{
q2.insert(p1->right);
cout << " " << p1->right->data;
}
}

q1 = q2;
}
}
```



o/p A.



o/p ⇒

A B C D

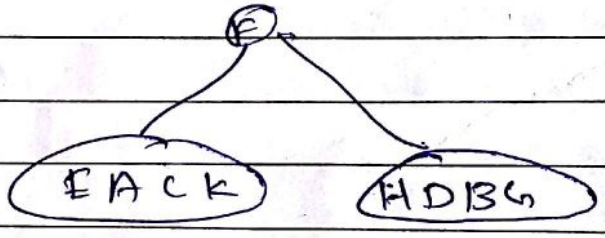
* Creation of Binary tree Traversal Sequence

→ Inorder E A C K F H D B G
Preorder F A E K C D H G B

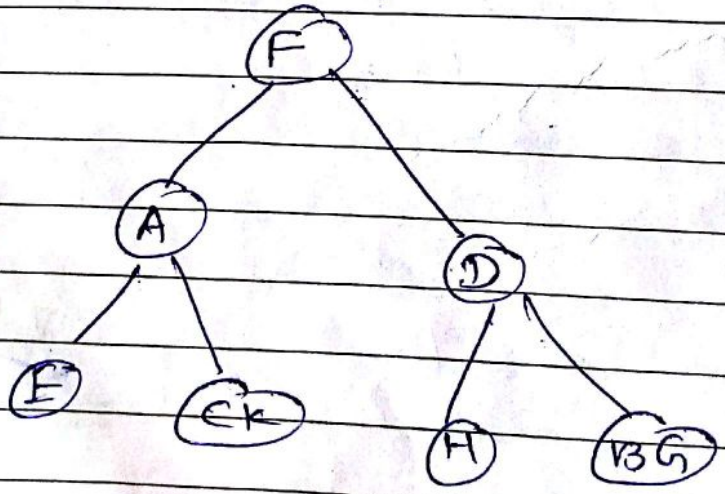
1. In preorder traversal root is the first node. Hence F is the root of the binary tree.

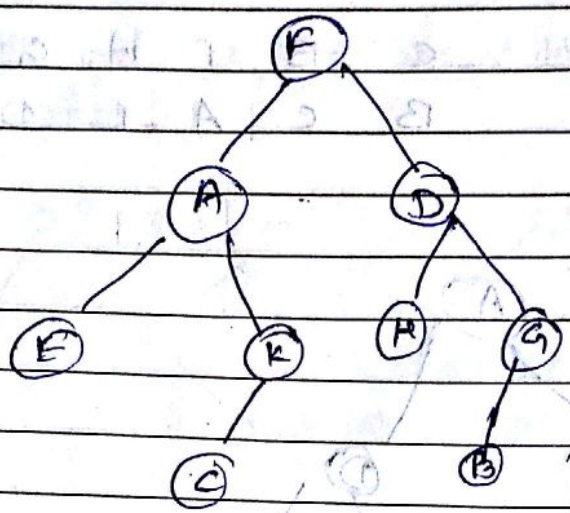
(F) root.

2. From Inorder.

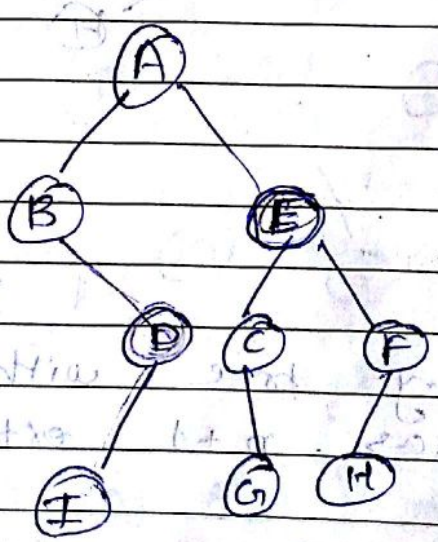


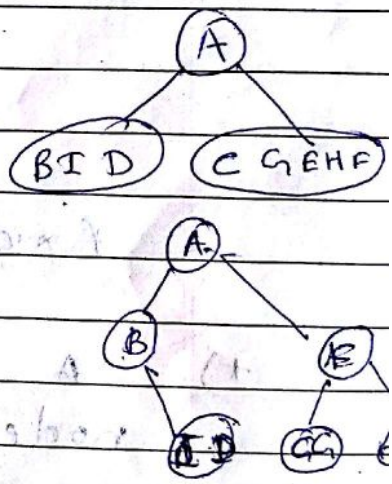
3.

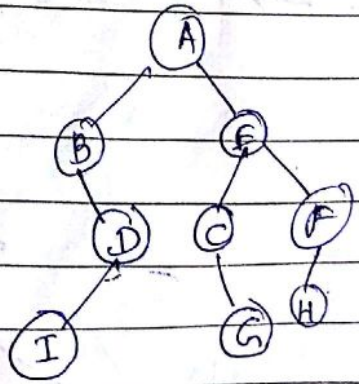




2) Inorder B I D A C G E H F
Postorder I D B G C H F E A



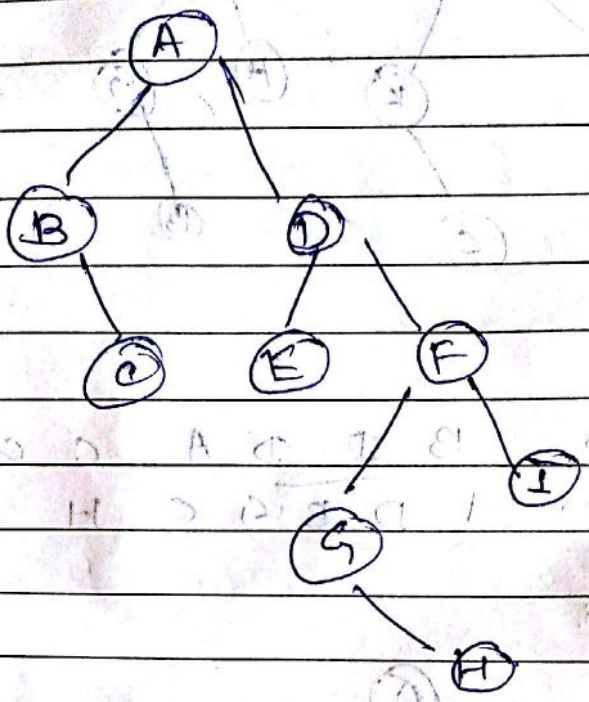




* Note: 1) Use Inorder seq. for checking left & right child
2) And use preorder and postorder for checking root or parent value.

Postorder : C B E H G I F D A

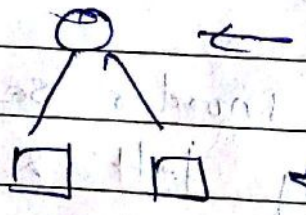
Inorder : B C A E D G H F I

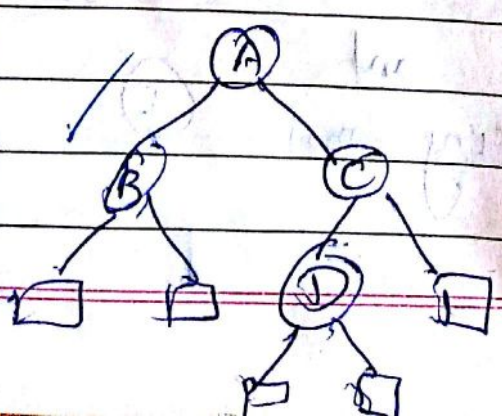


Example :

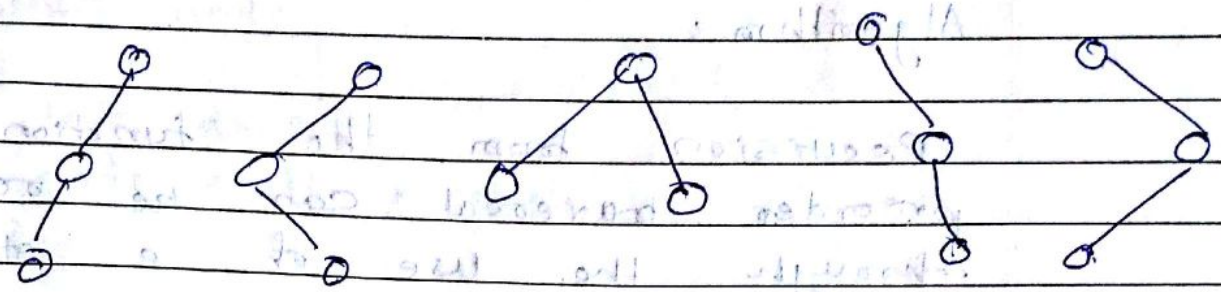
*) A binary tree with n internal nodes has n+1 external nodes

→

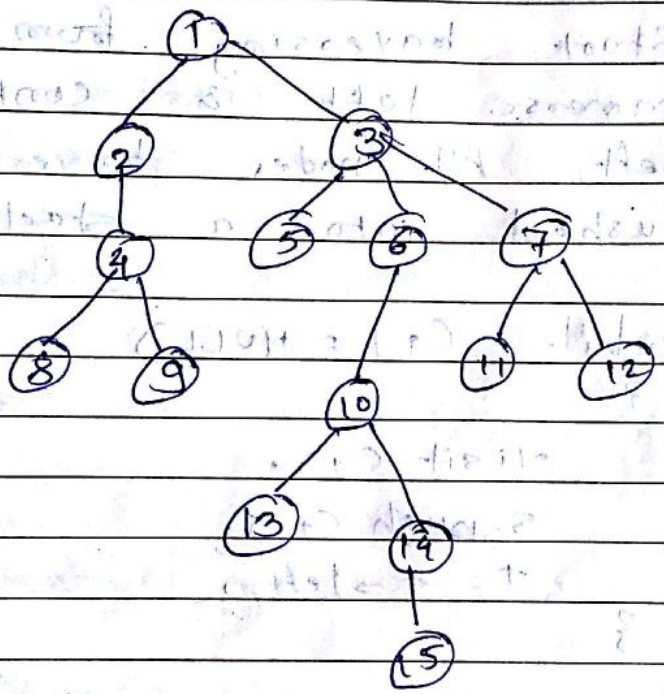




2) Draw all possible binary tree with 3 nodes:



3)



i) The sibiling of node 6 are 5, 7

ii) The height of node 3 : 4

iii) The longest path in tree : 1 to 15

iv) The subtree with root 5 : NULL

v) The depth of Node 13 : 4

* Non-Recursive Preorder Traversal:

Algorithm:

Recursion from the function for preorder traversal can be removed through the use of a stack.

Step 1:

Start traversing from the root traverse left & continue traversing left. All nodes traversed are pushed into a stack (S).

```
while (T != NULL)
{
    visit (T);
    S.push (T);
    T = T->left;
}
```

(nodes are saved in the stack the purpose of backtracking traversal to right subtrees of all nodes visited so far)

Step 2:

// program

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

class stack
{
 node *data[30];
 int top;
 public:
   stack()
   {
    top = -1;
   }

int empty()
{
  if (top == -1)
return 1;
else
return 0;
}

void push (node *p)
{
   top++;
   data[top] = p;
}
```

```
node * pop
{
  node *p;
  p = data[top];
  top--;
  return p;
}

void preorder_non_recursive (node *T)
{
  stack S;
  while (T != NULL)
  {
    cout << "\n" << T->data;
    S.push (T);
    T = T->left;
  }

  while (!S.empty())
  {
    T = S.pop();
    T = T->right;
    while (T != NULL)
    {
      cout << T->data;
      S.push(T);
      T = T->left;
    }
  }
}
```

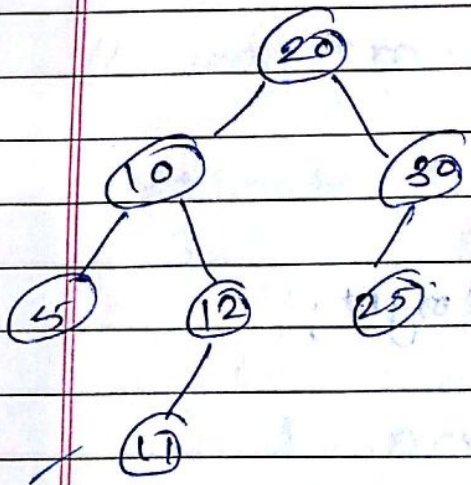
* Binary Search Tree (BST)

→ Defⁿ : A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies the following conditions:

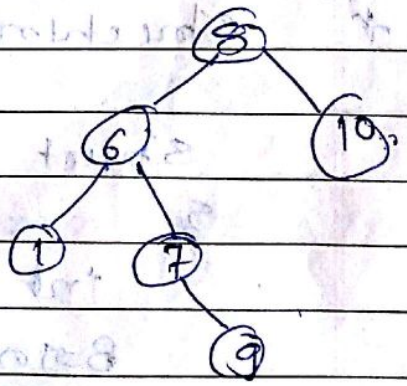
1) All key are distinct.

2) For every node, x in the tree, the values of all the keys in its left subtree are smaller than the key value in x.

3) For every node, x, in the tree, the values of all keys in its right subtree are larger than the key value in x.



A BST.



Not a BST

* Operations on a Binary Search

1) Initialise
2) Find
3) make empty
4) Insert
5) Delete
6) Create
7) Findmin
8) Findmax

1) Initialise

```
BSTnode * initialize()
{
    return NULL;
}
```

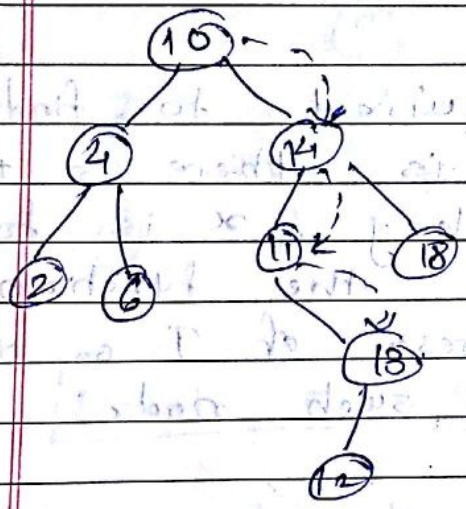
* Structure

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

* Insert operation

The function insert (T, x) adds elements x to an existing BST. T is tested for NULL. If so, a new node is created to hold x. T is made to point to the new node. If the tree is not empty we search for x as in find() operation.

To insert new key 12



// program:

```
BSTnode *insert (BSTnode *T, int x)
{
  if (T == NULL)
  {
    T = new BSTnode;
    T -> data = x;
    T -> left = T -> right = NULL
  }
  return T;
```

```
if (x > (T->data))
{
    T->right = insert (T->right, x);
    return T;
}
T->left = insert (T->left, x);
return T;
}
```

* Find operation :

It is often required to find whether a key is there in tree. If the key x is found in the node T, the function returns the address of T or if there is no such node.

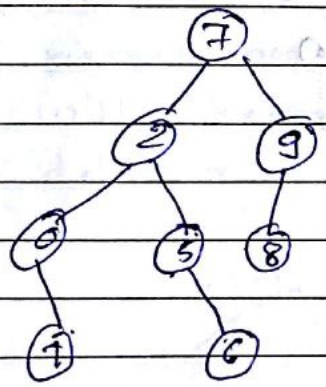
// program

```
BSTnode *find (BSTnode * root, int x)
{
    if (root == NULL)
        return NULL;
    if (root->data == x)
        return root;
```

if (x > root->data)
return find(root->right, x));
else
return find((root->left), x));

* Insert the log integers 7, 2, 9, 0, 5, 6, 8, 1 into a binary search tree by repeated application of insert operation

→

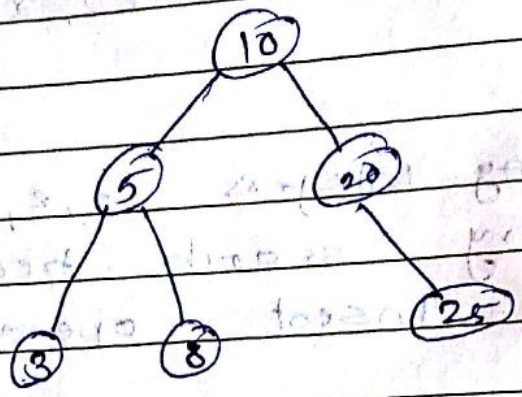


* **Delete operation** :-

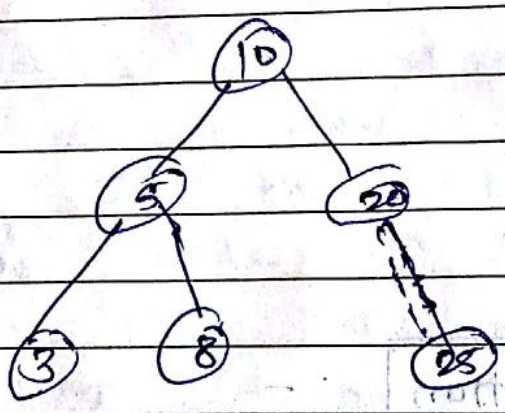
In order to delete a node, we must find the node to be deleted. The node to be deleted may be :
a) A leaf node
b) A node with one child.
c) A node with two children.

a) If the node is a leaf node, it can be deleted immediately by setting

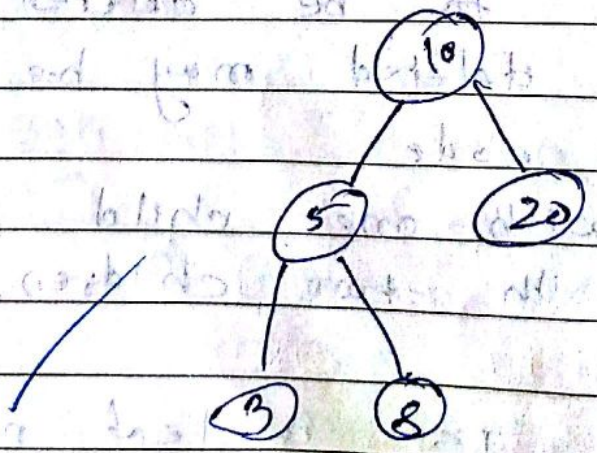
the corresponding parent pointers to NULL. For example, consider the tree shown below.



Before deletion



deletion

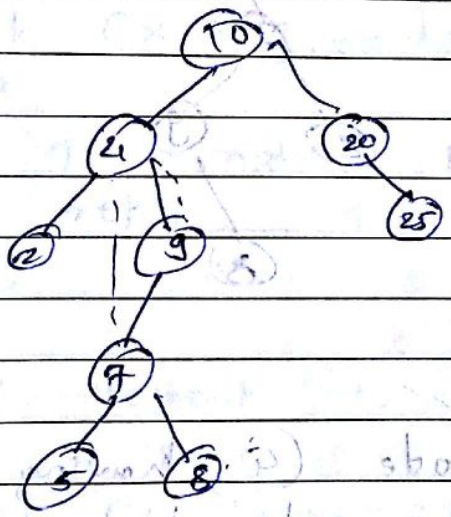


After deletion

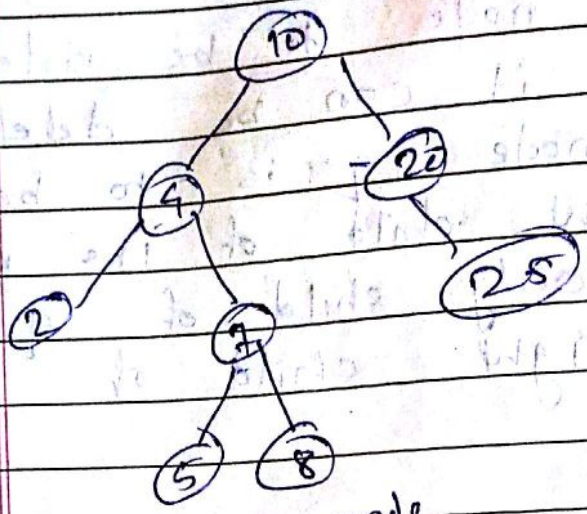
- Even when the node to be deleted has one child, it can be deleted easily. If a node q is to be deleted & it is right child of its parent node p. The only child of q will become the right child of p after deletion of q.

- Similarly, with a node q is to be deleted & it is left child of its parent node p. The only child of q will become the left child of p after deletion.

b]

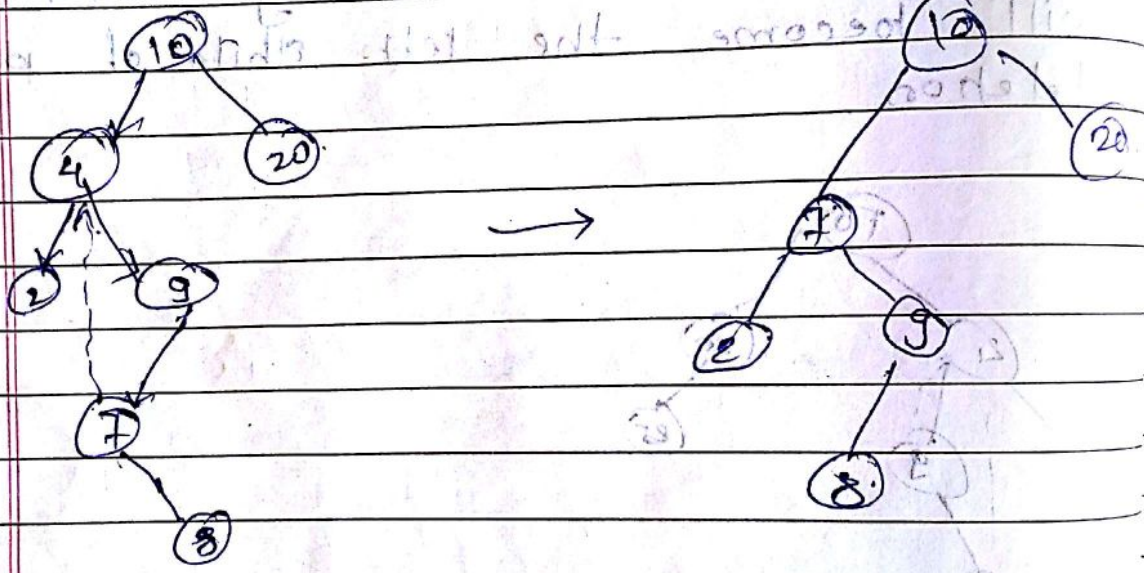


Here suppose node q is to be deleted then 7 will become right child of 4.



[if node is having child

c)



[deletion of node (4) having the case in which the node to be del two children is a bit complicated. the strategy is to replace the data of node with the smallest data of the subtree & then delete the smallest the right subtree. The smallest right subtree will either be leaf node of degree 1.

// program.

```
BSTnode * Delete (BSTnode *T, int x)
{
  if (T == NULL)
  {
    cout << "\n Element not found";
    return T;
  }
  if (x < T->data)
  {
    T->left = Delete (T->left, x);
    return T;
  }
  if (x > T->data)
  {
    T->right = Delete (T->right, x)
    return T;
  }

// element is not found.
  if (T->left == NULL && T->right == NULL)
  {
    Temp = T
    free (Temp);
    return NULL;
  }
```

node

Scanned by CamScanner

```
if (T->left == NULL)
{
    temp = T;
    T = T->right;
    delete temp;
    return T;
}
if (T->right == NULL)
{
    temp = T;
    T = T->left;
    delete temp;
    return T;
}
// node with 2 children
tem = find_min (T->right);
T->data = temp->data;
T->right = Delete (T->right, temp->d
return T;
}
```

* Create.

A binary search tree can be created by making repeated calls to insert operation.

```
BSTnode * create()
{
    int n, x, i;
    BSTnode *root;
    root = NULL;
    cout << "\n Enter no. of nodes :";
    cin >> n;
    cout << "\n Enter tree elements:";
    for(i=0; i<n; i++)
    {
        cin >> x;
        root = insert(root, x);
    }
    return(root);
}
```

* **Findmin** :

This function returns the address of the node with smallest value in the tree.

```
BstNode * findmin (BSTnode *T)
{
    while (T -> left != NULL)
        T = T -> left;
    return T;
}
```

* **Findmax** : this fun^n returns the address of the node with largest value in the tree.

```
BstNode * findmax (BstNode *T)
{
    while (T -> right != NULL)
        T = T -> right;
    return T;
}
```

* Threaded Binary Trees (TBT)

* Binary Tree and BST

We have studied both binary tree & BST. A BST is a special case of the binary tree. The comparison.

1. Both of them are trees with degree two, that is, each node has atmost 2 children.

2. The BST is a binary tree with the property that value in a node is greater than any value in a node's left subtree & less than any value in a node's right subtree.

3. The BST guarantees fast search time provided the tree is relatively balanced, whereas for a binary tree the search is relatively slow.

* Threaded Binary Tree (TBT):

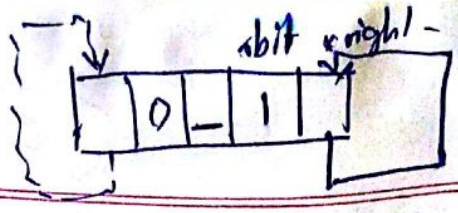
During Diff^n operations like delete, insert, some observations —

1) first is that for all leaf nodes and those with 1 child the Lchild &/or Rchild fields are set to NULL.

2) 2nd observation is in traversal proc^n. The traversal fun^n use stack to store information about those nodes whose processing has not been finished. In case of non-recursive traversals, user-defined stack is used, & in case of recursive traversals internal stack is used. there is addition time for processing, but additional space for storing the stack is required.

To solve this problem, the sol^n is TBT.

In a linked representation of binary tree, there are more null than actual pointers. These null li can be placed by pointers called

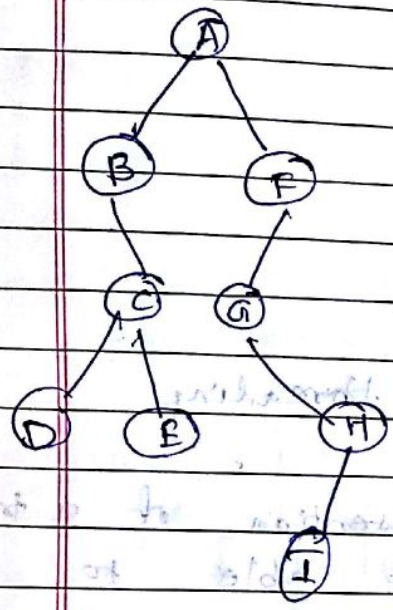


Structure of dummy node

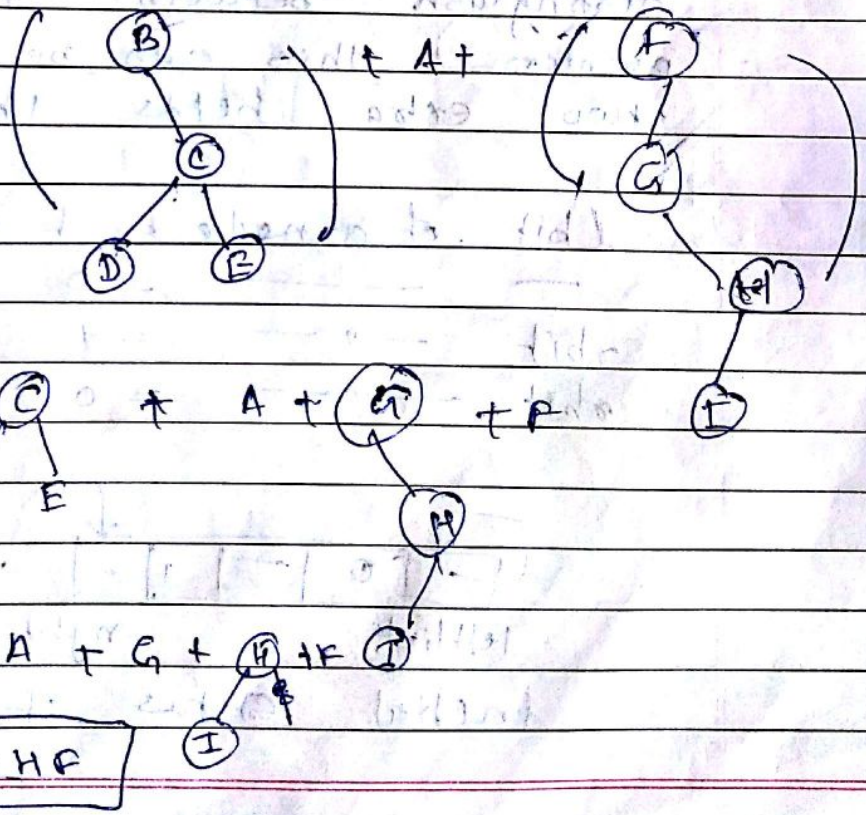
Date
Page

A left null link of a node is replaced with the address of it's inorders ~~successor~~ predecessor.

e.g. the give tree

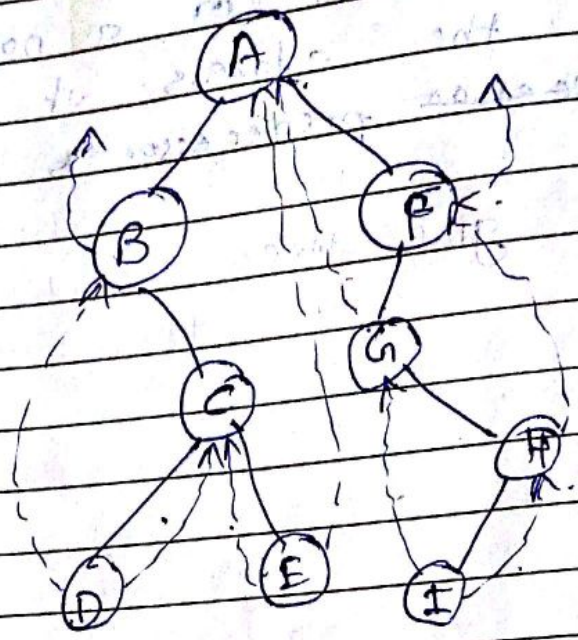


inorder on



B D C E A G I H F

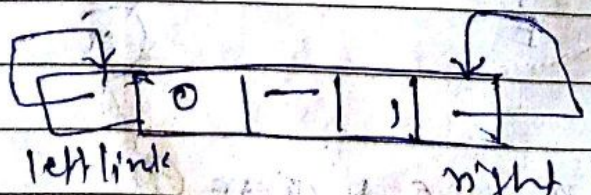
BDCEAGIHF



e. Tree After threading

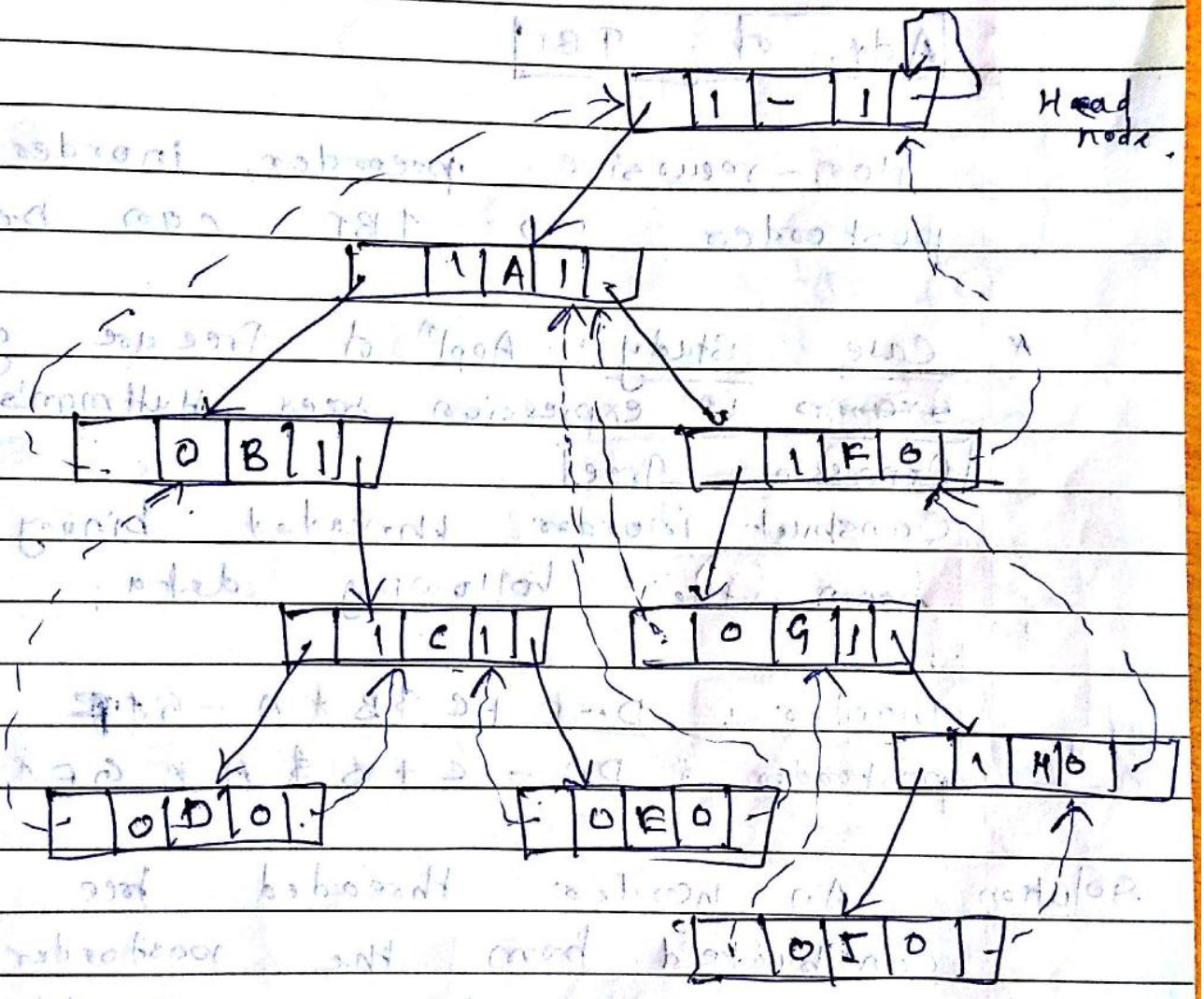
- In the memory representation of a node we must be able to distinguish between threads & n: pointers. This can be done by two extra fields 1 bit & rbit

lbit of a node = 1 - left child is
" " = 0 " "
rbit " " = 1 right "
rbit " " = 0 "



Initial Status of head no

In fig (2) 2 threads have been left dangling. Node B has no inorder predecessor & the node F has no inorder successor. this problem can be solved by taking a head node.



memory representation of TBT

```
struct TBTnode
{
    char data;
    TBTnode *left;
    TBTnode *right;
    int lbit, rbit;
}
```

Adv. of TBT

Non-recursive preorder, inorder & postorder on TBT can be

* Case Study: Appl^n of Tree are game Example: expression trees, Huffman's

Expression Tree

Construct inorder threaded binary tree from the following data:

inorder : D - E + C $ B * A - G * F

postorder → D E - C + B $ A * G F * -

Solution An inorder threaded tree can constructed from the postorder sequence as it is a postfix of an expression.

Input

1) D E - C + B $ A * G F * -

2) E - C + B $ A + G F * -

3) - C + B

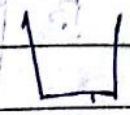
4) C + B

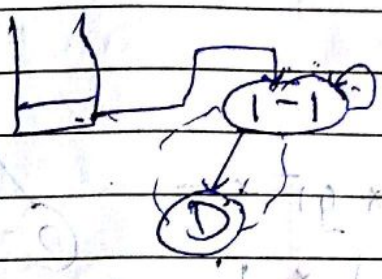
5) + B $

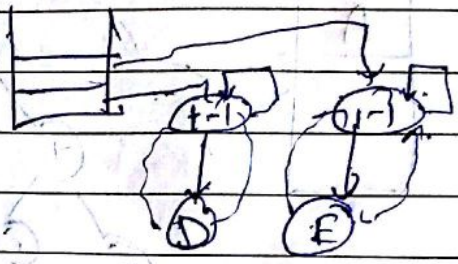
6) B $ A +

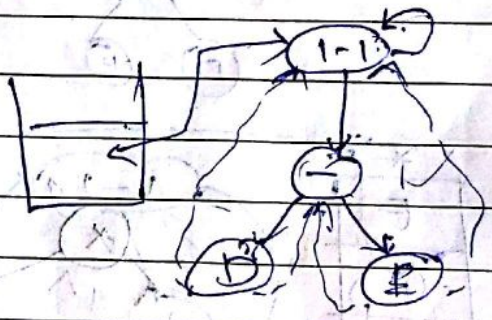
Stack

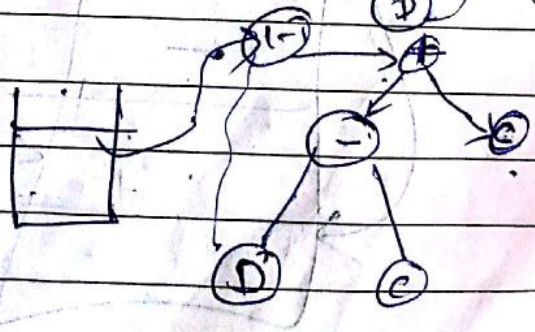
Initially.

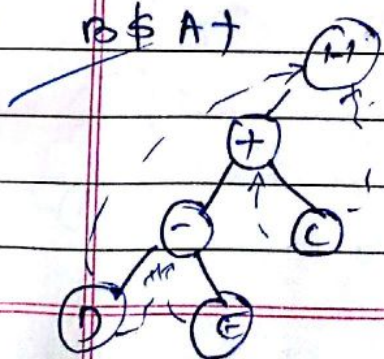


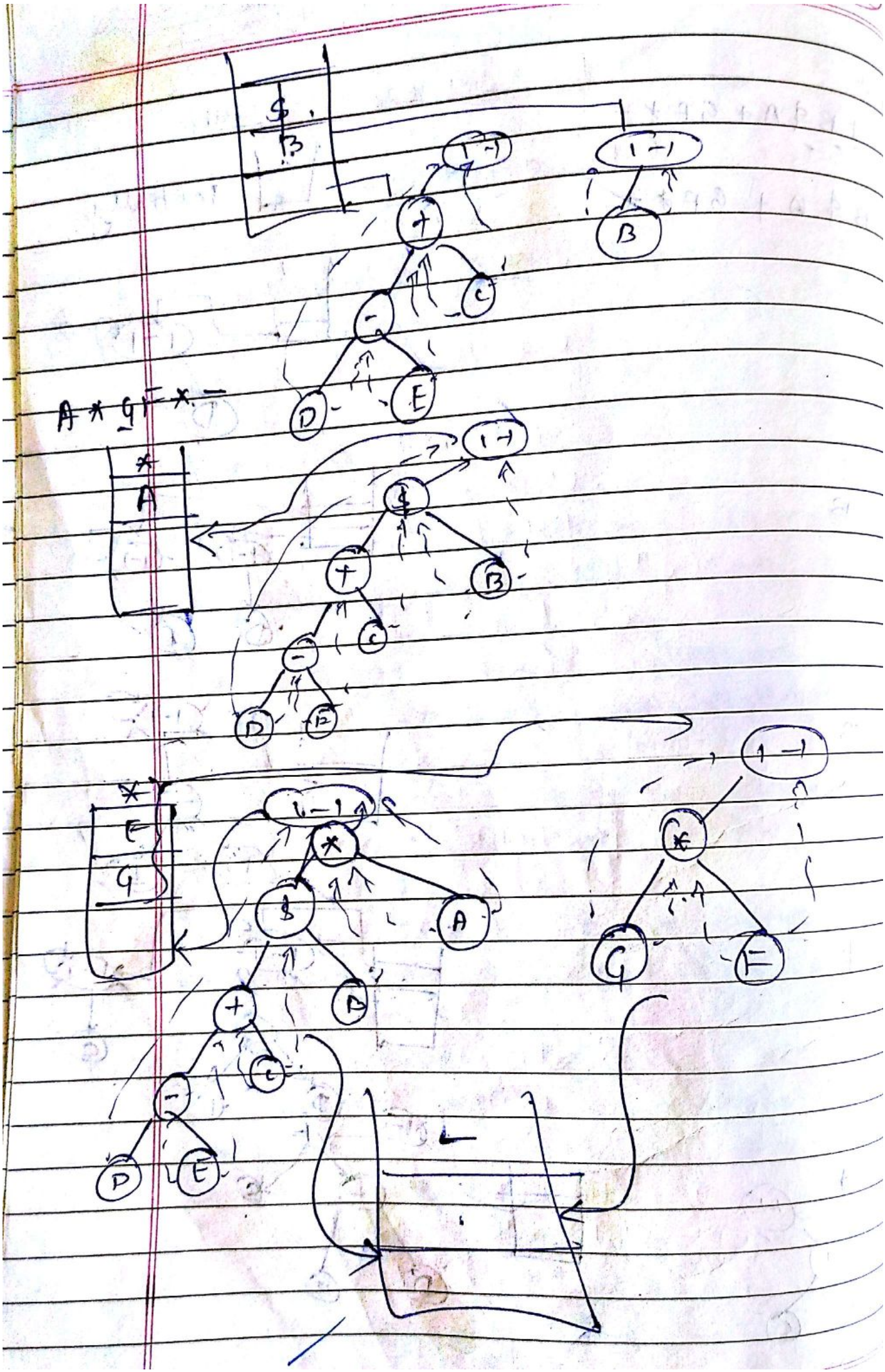


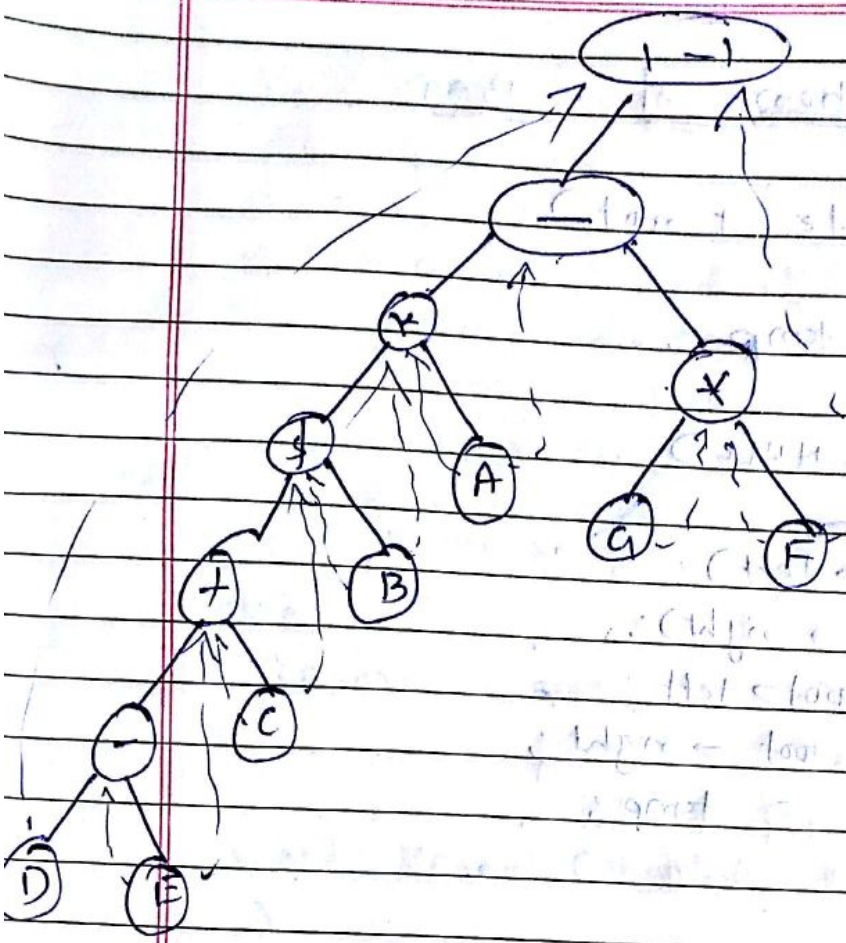




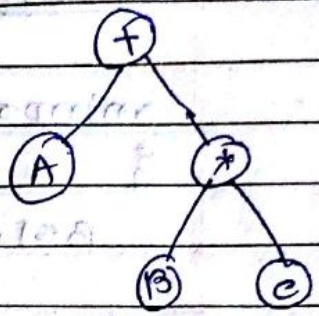








For e.g.



Inorder Traversal : A+B*C

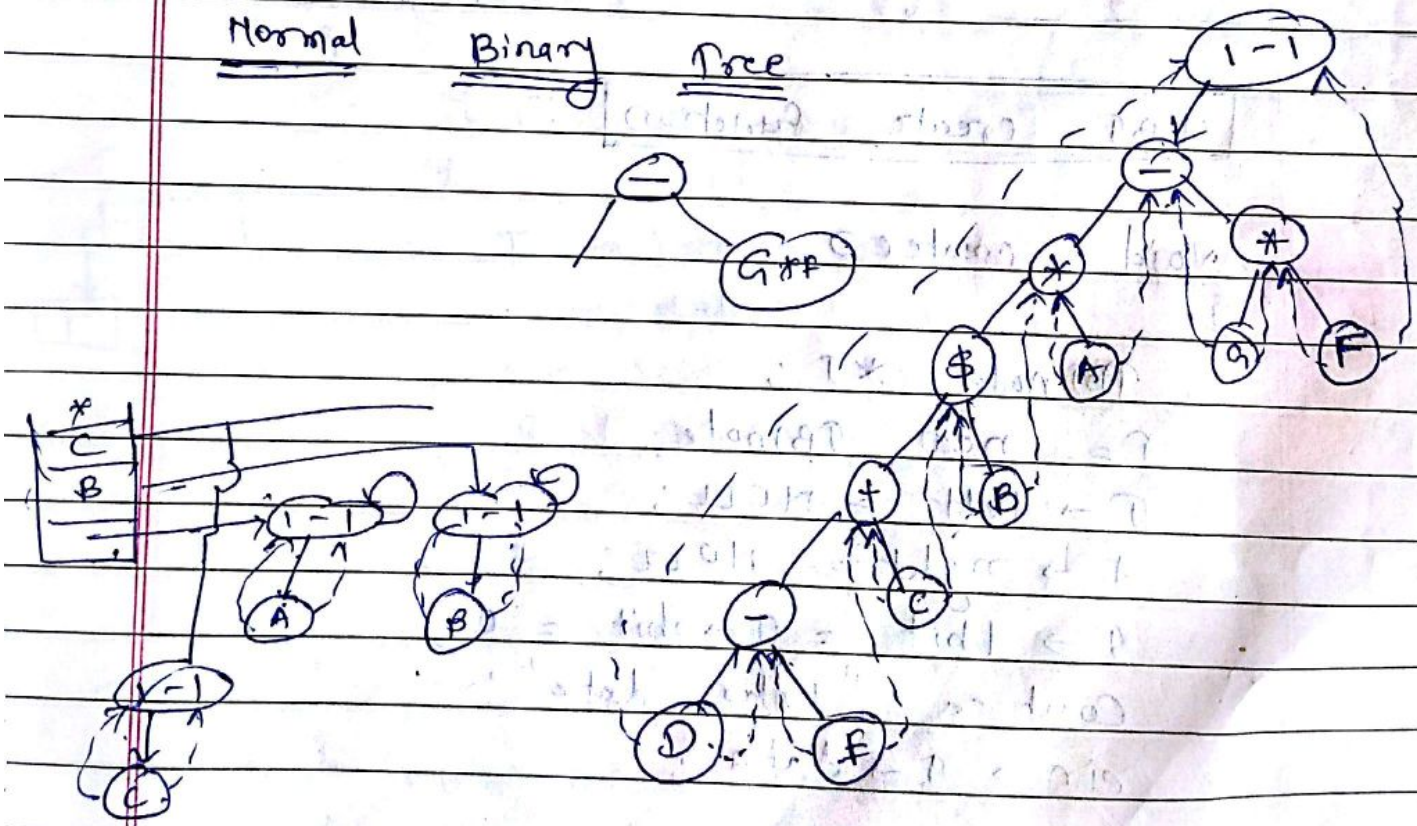
Preorder : +A*BC

Postorder : ABC*+

Inorder gives infix exps

preorder gives postfix expres

Normal Binary Tree



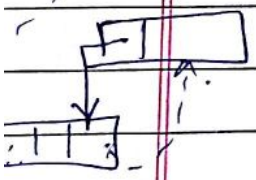
Mirror Operation of BST

```
mirror( BSTnode * root)
{
  BSTnode *temp;
  if (root != NULL)
  {
  mirror (root -> left);
  mirror (root -> right);
  temp = root -> left;
  root -> left = root -> right;
  root -> right = temp;
  }
}
```

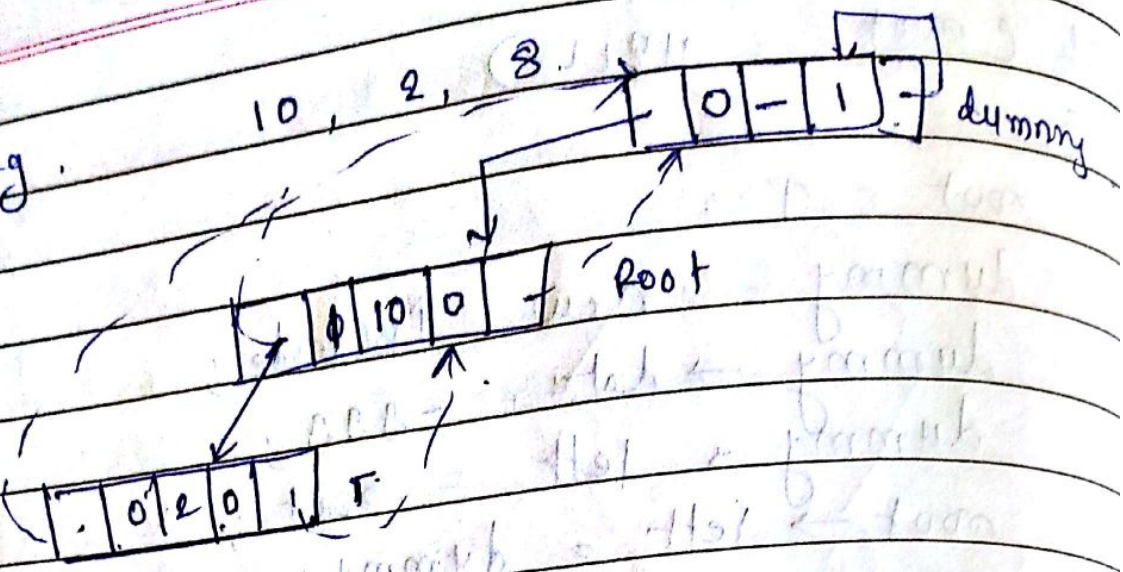
TBT create function :

```
Void create()
{
  TBTnode *T;
  T = new TBTnode;
  T -> left = NULL;
  T -> right = NULL;
  T -> lbit = T -> rbit = 0;
  cout << "Enter data";
  cin >> T -> data;
```

```
if (root == NULL)
{
  root = T;
  dummy = new TBTnode;
  dummy -> data = -999;
  dummy -> left = root;
  root -> left = dummy;
  root -> right = dummy;
  dummy -> right = dummy;
}
else
  insert (root, T);
}

Void Insert (TBTnode *R, TBTnode *T)
{
  if (T -> data < R -> data)
  {
    if (R -> lbit == 0)
    {
      T -> left = R -> left;   (Thread)
      T -> right = R;
      R -> left = T;
      R -> lbit = 1;
    }
    else
      insert (R -> left, T);
  }
  {
    same for if (T -> data > R -> data)
  }
```



e.g. 10, 2, 8



Display fun^n

```
void inorder (Tnode * Root, node * dy
{   T = Root ;
  while ( T != dummy )
  {
    T = T -> left ;
    cout << T -> data ;
  }
  while ( T -> rbit == 0 )
  {
    T = T -> right ;
    if ( T == dummy )
      return ;
    cout << T -> data ;
  }
  temp  T = T -> right ;
}
```

left child display

Parent ki display

move to right side for display.