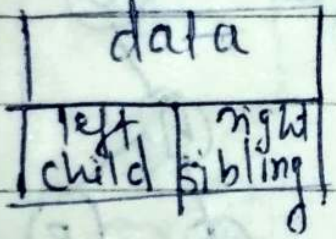
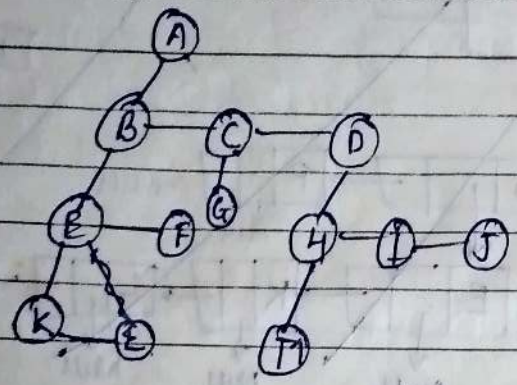
(2) Left child - Right Sibling Representation :-



Left child - right sibling node structure.

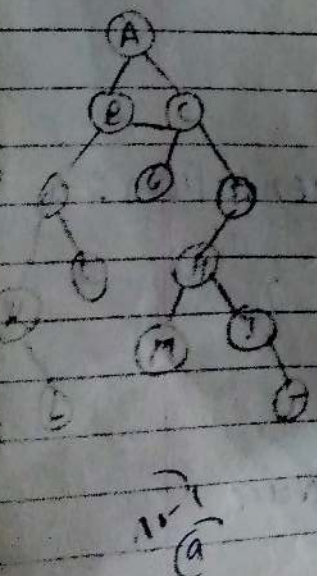
left child - right sibling representation :-

- To convert the tree into this representation, we have every node has at most one leftmost child & at most one closest right sibling.

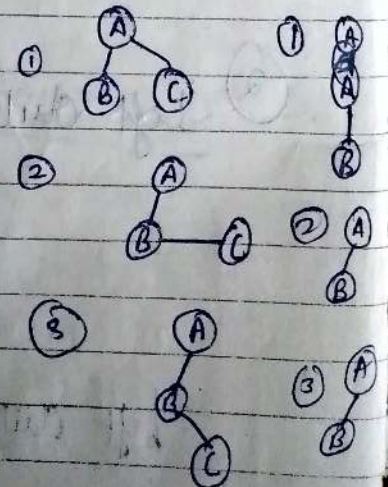


(3) Representation as a Degree-Two tree :-

- we simply rotate the right sibling pointers clockwise by 45°.



example :-



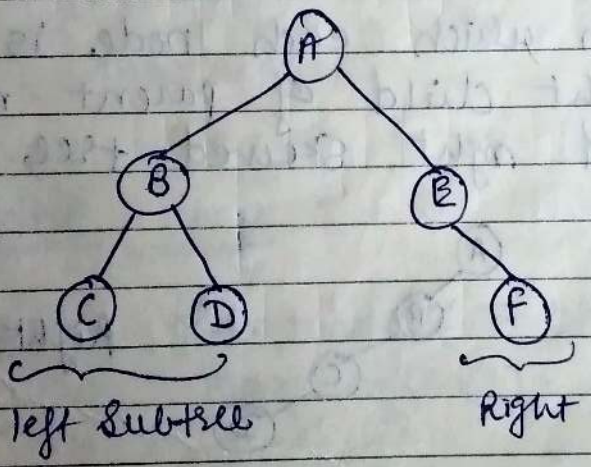
* Binary Trees :-

Definition :-

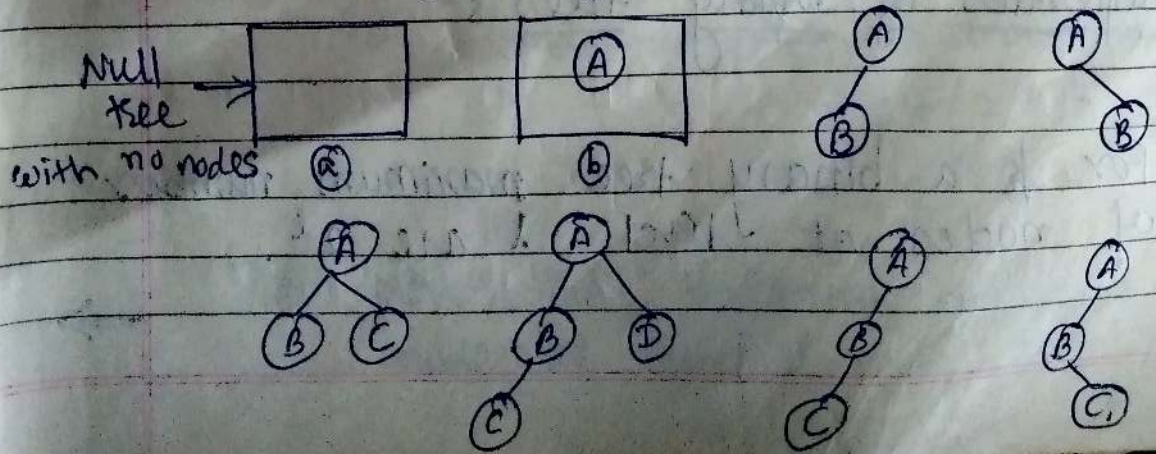
A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

- A binary tree is a tree in which no node can have more than two subtrees.

or a node can have zero, one or two subtrees.



Collection of Binary trees :-

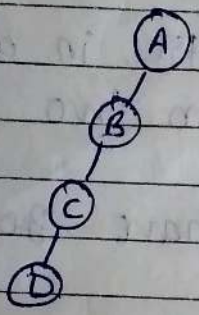


* Properties of Binary trees :-

⊛ Left and Right skewed trees :-

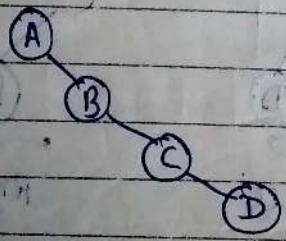
- The tree in which each node is attached as a left child of parent node then it is left skewed tree.

= ~~The tree~~



left skewed tree.

→ The tree in which each node is attached as a right child of parent node then it is called right skewed tree.



Right skewed tree.

properties of Binary trees :-

(1) For a binary tree maximum number of nodes at level 'l' are 2^l.

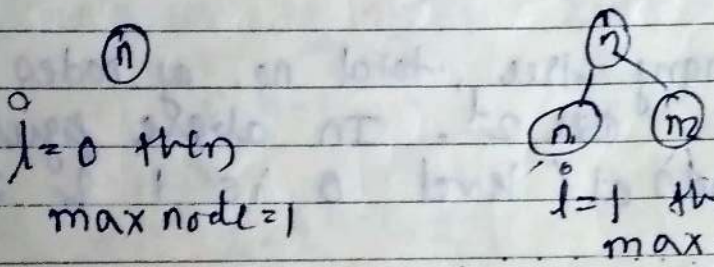
⟹

proof:- Induction Basc :

As we know, root node is a unique node present at 0^{th} level.

That means, if $l = 0$, maximum no. of nodes are 2^l i.e $2^0 = 1$

The binary tree is a tree in which each node has maximum two nodes.
Hence if level $m = 1$ then maximum nodes are, $2^1 = 2^1 = 2$.



Inductive step :

As, Max_nodes(0) = 1.
Max_nodes(1) = 2 * max_nodes(0)
= 2 * 1 = 2
" (2) = 2 * max_nodes(1)
= 2 * 2 = 4.
⋮
So max_nodes(K) = 2 * max_nodes(K-1)
= 2 * (2 * (2 * max_node(0)))
= 2^K.

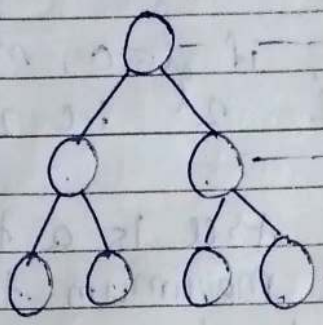
This proves that maximum number of nodes at level l are 2^l.

full binary tree = is a tree in which every node other than the leaves has 2 children.
(strictly binary tree)

② A full binary tree of a given height h has exactly $(2^{h+1}-1)$ nodes.

⇒ Induction Basis :-

$(2^{3}-1) = (8-1) = 7$



Height of a given tree ie $h = 3$.

for binary tree, total no. of nodes at level 1 are 2^1. In above figure, total nods at level 0 is 1 & 2 are 4

Thus the total no. of nodes in the given binary tree are,

$$\text{total\_nodes} = 2^0 + 2^1 + 2^2 + \cdots + 2^1.$$

$$= \sum_{i=0}^{1} 2^i$$

If height $h = 1$;

then $2^{h+1} = 2^0 = 1$. ie root node.

height $h = 2$

then $2^{h+1} = 2^{1+1} = 2^1$ ie 2 nodes.

Induction hypothesis :

Assume total nodes $\leq 2^h - 1$ for a full binary tree.

Inductive step :

Now a tree T has 2 subtrees T_L & T_R.
& height of subtrees is $h-1$.

Hence. $T = T_L + T_R$
total node = total node of (T_L) + total node(T_R)
+ 1.

$$T = (2^{h-1} - 1) + (2^{h-1} - 1) + 1.$$

$$T = 2 \cdot 2^{h-1} - 1$$

$$T = 2^h - 1$$

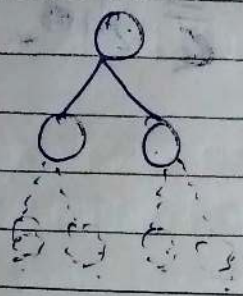
Thus. a full binary tree of a given height h has $2^h - 1$ nodes is proved.

(3) In a binary tree the no. of external nodes is one more than the no. of internal nods. i.e $e = i + 1$.

Proof :- It is true for one node:

$i = 1$
$e = 1 + 1 = 2$

It is also true for 3 nodes :-



$i = 3$

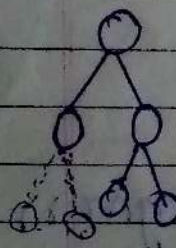
$e = 3 + 1 = 4$.

Induction hypothesis :-

~~Assume it is also true with upto n nodes.~~

Assume true for a tree with n nodes ($e = i + 1$). Now we want to add new external nodes.

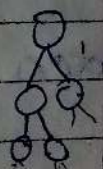
Inductive step :-



when we add two external nodes then old external becomes internal. So we have,

$$e_{new} = (e-1) + 2$$

$$= e + 1$$



$$= (i+1) + 1 = i + 2$$

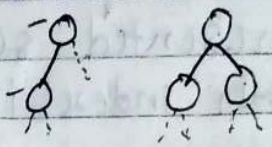
$$i_{new} = i + 1$$

Hence

$$e_{new} = i_{new} + 1.$$

(4) The number of external nodes e satisfies.

$(h+1) \leq e \leq 2^h$.

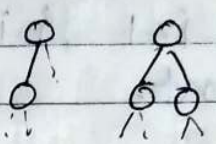


h=2

$3 \leq e \leq 4$

(5) The number of internal nodes i satisfies.

$h \leq i \leq 2^h - 1$.



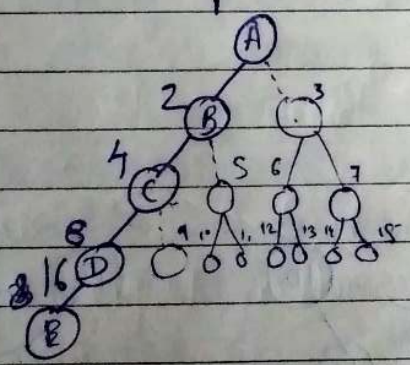
h=2

$2 \leq i \leq 3$

* Binary tree Representation :-

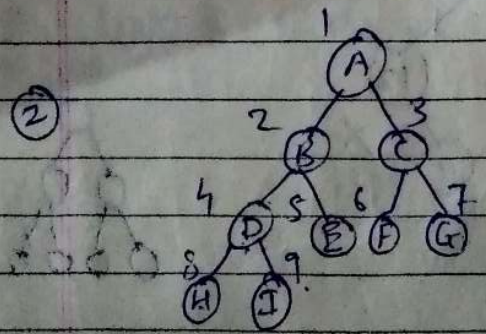
(1) ~~ASCII~~ Array Representation.
(2) Linked representation.

(1) Array Representation :-



0	–
1	A
2	B
3	–
4	C
5	–
6	–
7	–
8	–
16	E

(2)



1	–
2	A
3	B
4	C
:	:
9	I

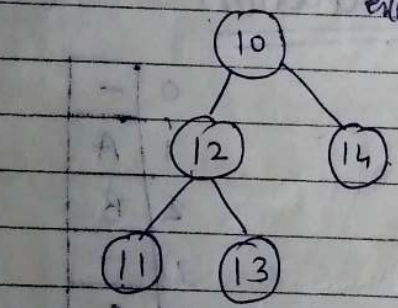
① If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have.

1) parent (i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
If $i = 1$, i is at the root and has no parent.

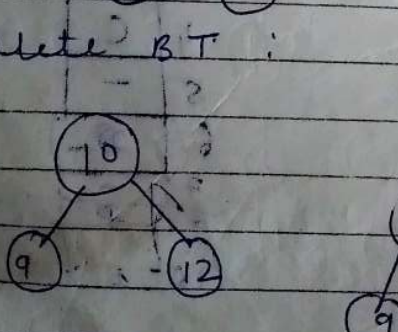
2) left child (i) is at $2i \le n$. If $2i > n$ then i has no left child.

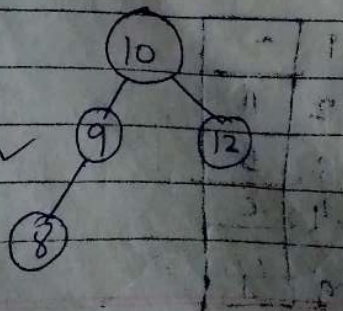
3) right child (i) is at $2i+1$ if $2i+1 \le n$,
If $2i+1 > n$, then i has no right child.

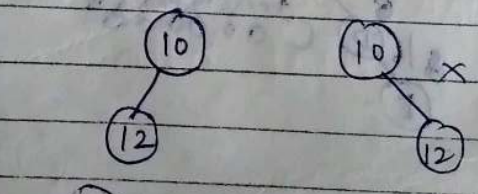
Full Binary tree : Each node has exactly 0 or 2 children

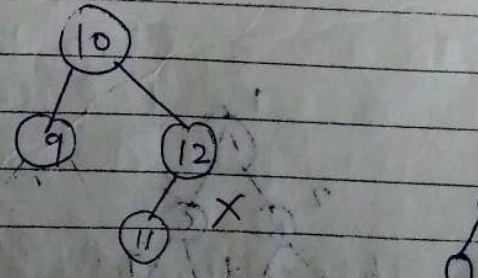


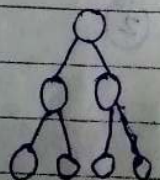
Complete BT :





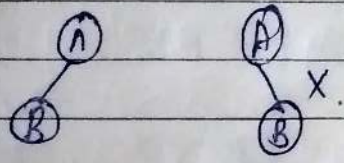






* Complete Binary tree :-

A binary tree in which every level, except possibly the deepest is completely filled. and
~~At depth d, the height of the tree,~~ all nodes must be as far left as possible.



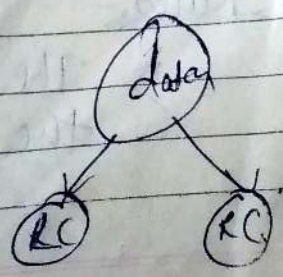
* Linked Representation :-

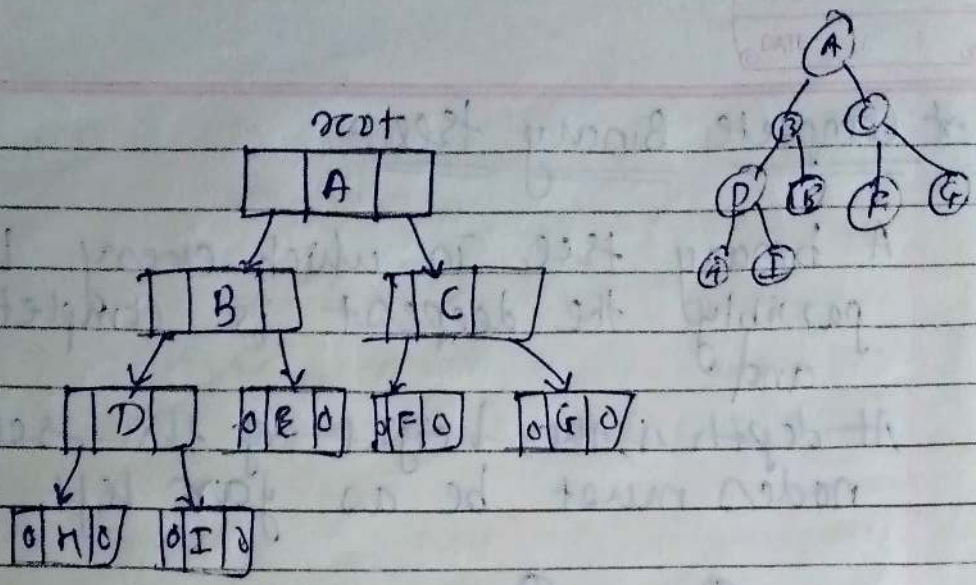
→ Array representation is good for complete binary trees, it is wasteful for many other binary trees.

→ Insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes to reflect the change in level no. of these nodes.

- So these problems are overcome by using linked representation.

left child	data	right child

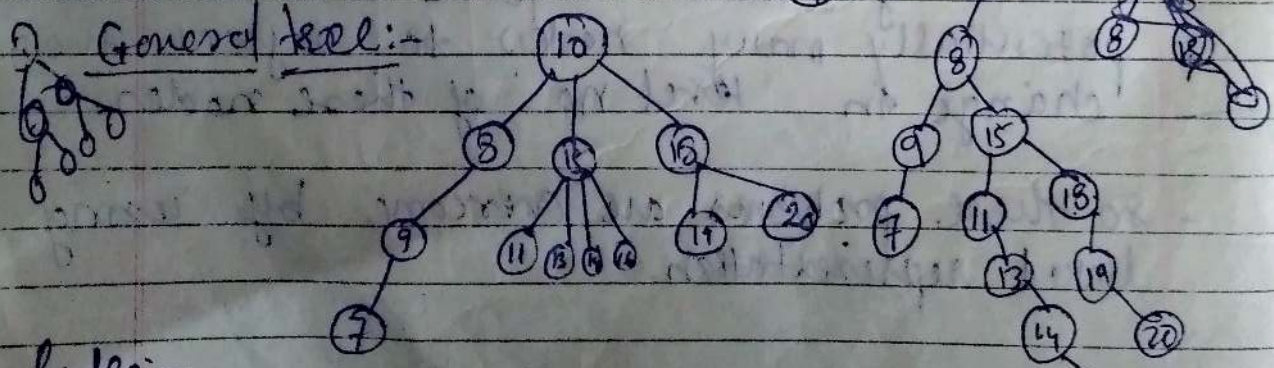





```
struct node
{
    int data;
    node *left;
    node *right;
};
```

* Converting tree to Binary tree :-

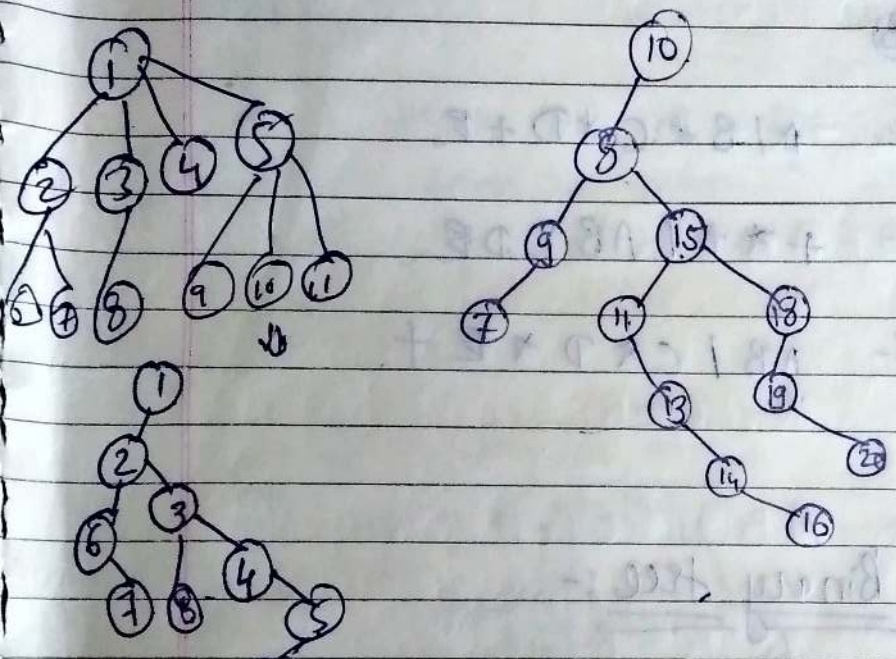
1) General tree :-



Rules :-

1) The root of the general tree is the root of the binary tree.

2) find the first child node of the node & attach it as a left child to the current node in binary tree.
3) The right siblings can be attached as a right child of that node.

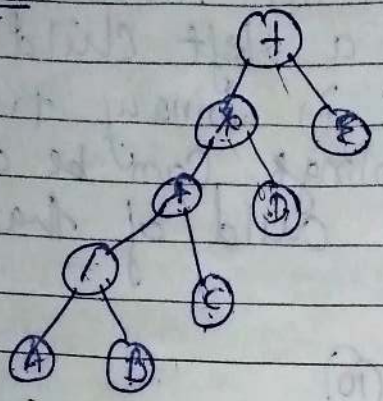


* Binary tree traversals :-

→ Three important traversal :- ~~LRV, LVR, ALV~~
Left - root - Right.
Left - right - root.
root - left - right.
} traverse left before right.

Inorder - Left - Root - right.
preorder - root - left - right.
post order - left right root.

example:-



① Inorder :- A/B*C*D+E

② preorder :- +**/ABCDE

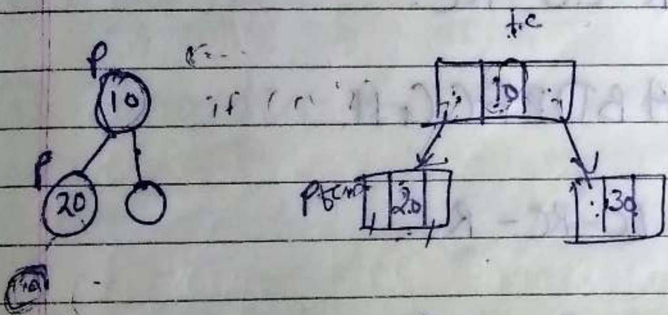
③ postorder :- AB/C*D*E+

* Creation of Binary tree :-

```
class tree
{
    int data;
    tree *lptr;
    tree *rptr;
public:
        tree *create(int);
        tree* insert(int, tree *);
};
tree* tree::create(int x)
{
    tree *p;
    p = new node;
    if(p == NULL)
```

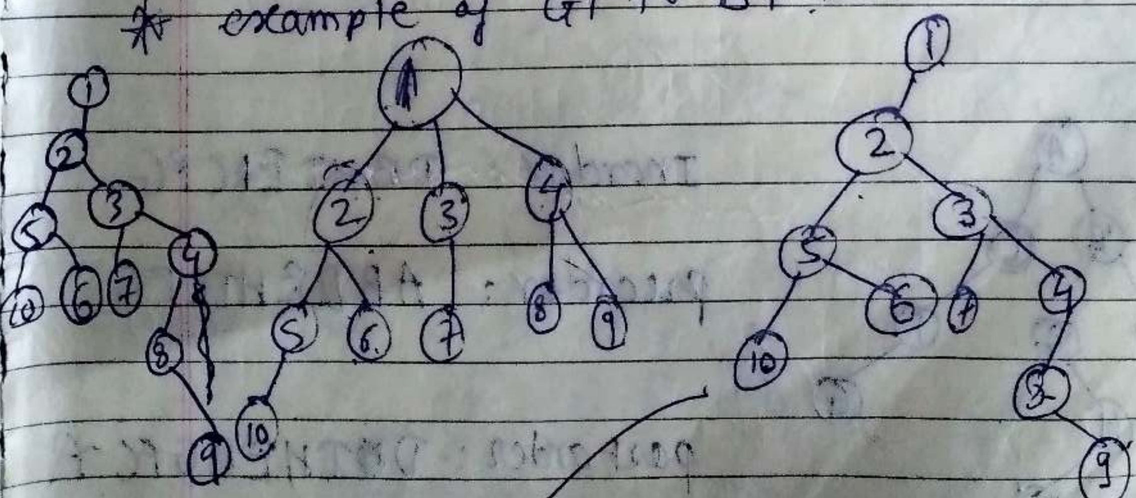
```
p -> data = x;
p -> lptr = NULL;
p -> rptr = Null;
return p;
}
```

```
tree* tree :: insert(int x, tree *root)
{
    tree *p, *temp;
    temp = root;
    p = new tree;
```



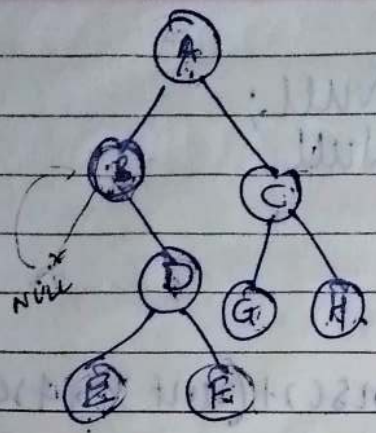
---

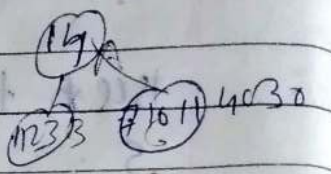
* example of GT to BT :-




```
p = new tree;
p -> data = x;
p -> lptr = Null
```

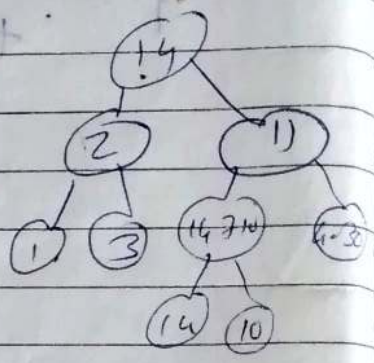
Example:-





(1) Inorder:- LC-R-RC.

→ BEDFAGCH.



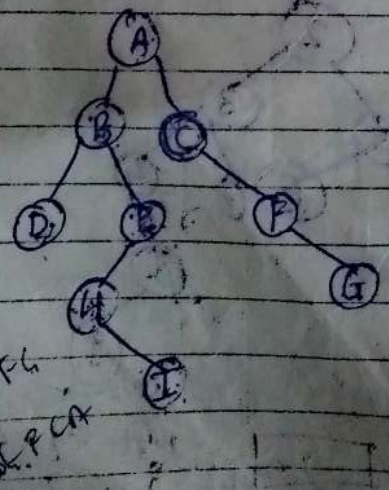
(2) preorder:- R-LC-RC.

~~AB~~ ABDEFCGH.

(3) postorder:- LC-RC-R.

EFDBGHCA.

(2)



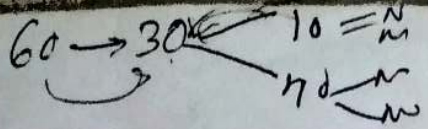
DBHIEACFG

DIHEBGFCA

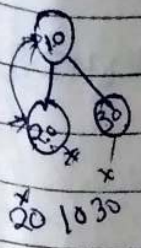
Inorder :- DBHIEACFG.

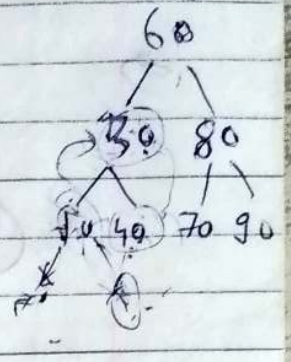
preorder :- ABDEHICFG.

postorder :- DIHEBGFCA



```
Void inorder (tree *root).   ← address of the root node is passed.
{
  if (root != NULL)
  {
    inorder (root -> left);
    cout << "in" << T -> data;
    inorder (root -> right);
  }
}
```





60 30 10

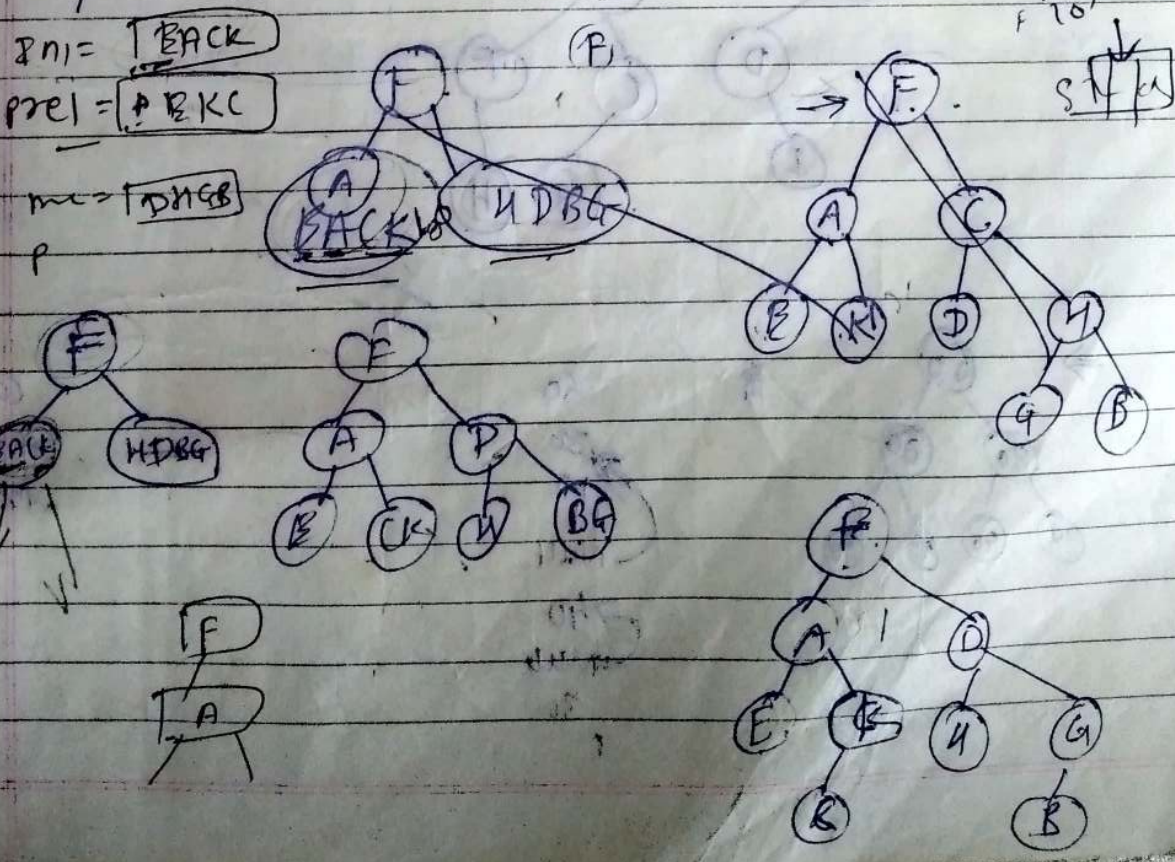
60 → 30 - 10

F'10'

Create Binary tree from following traversal:-

(1) Inorder :- E A C K F H D B G

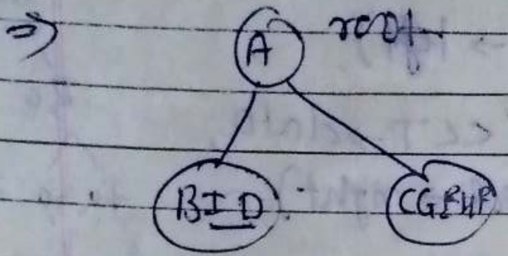
(1) preorder :- F A E K C D H G B



In: 0 1 2 3 4 5 6 7 8

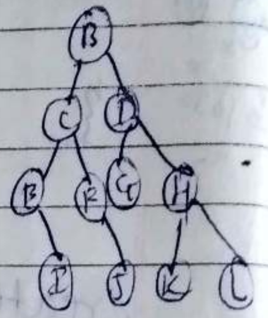
(2) B I D A C G E H F.   C = 9

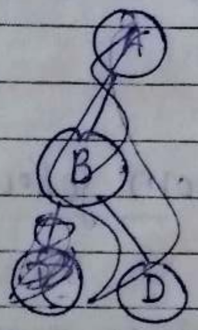
post :- I D B G C H F E A.

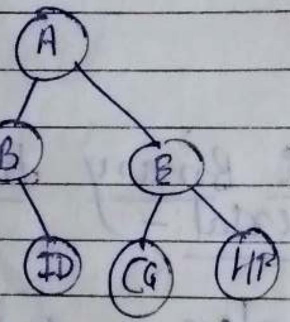


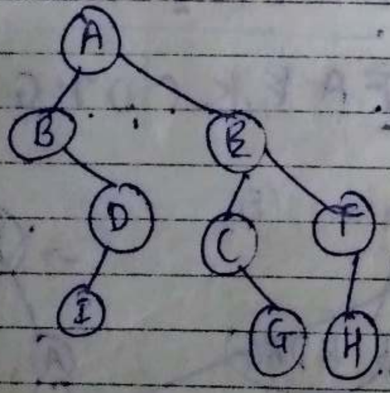
(3) Inorder: E I C F J B G D K H L

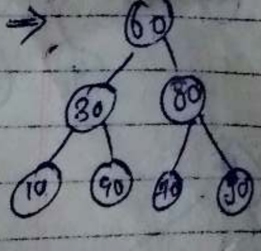
post :- I E J F C G K L H D B







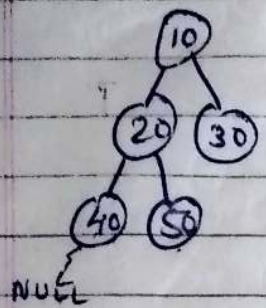


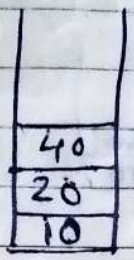


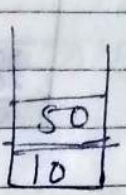
60
30
10
15
Null.
40
Null
30.

cout<< data
→ Pre(tree → lpt)
→ Pre(tree → rpt)

* Binary traversal using non-Recursion method:



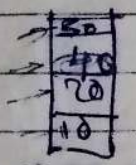




① preorder :- 10 20 40 50 30

10 20 40 50 30

② Inorder :- 40 20 50 10 30



40 40 20 50

root = root → rptr.

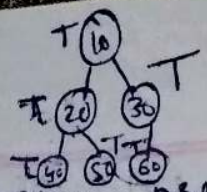
10: 40 20 50 10 30

20
40
NULL

```
void inorder (tree * root)
{ if (root != NULL)
  inorder(root → lptr);
  Cout << "\n" << root→data;
  inorder(root → rptr);
}
```

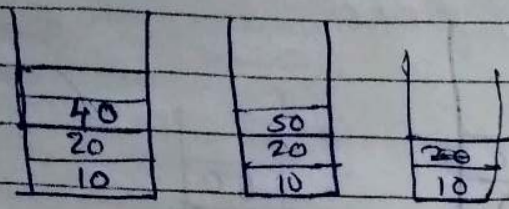
```
char in[30];
cin >> in;
```

8999099937

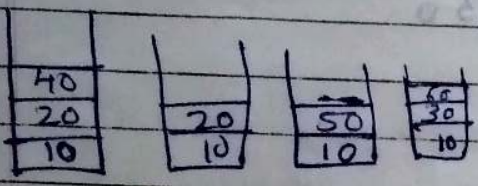


* Non-Recursive ~~preorder~~ preorder :-

→

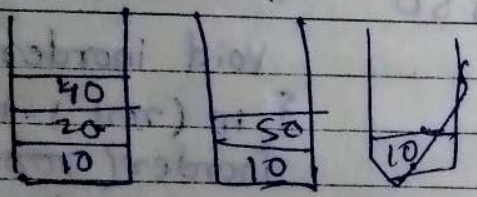


10 20 40 50



10 20 40 50 30 60

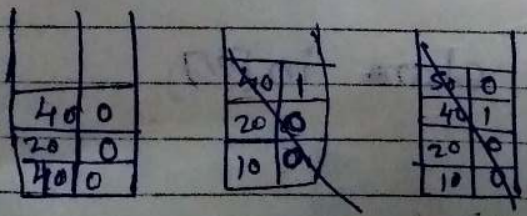
* Inorder :-



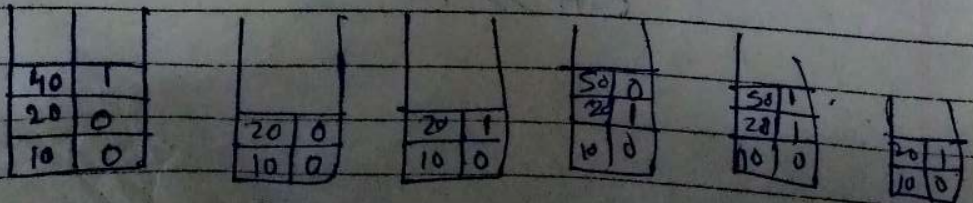
40 20 50 10 60 30

* post order :-

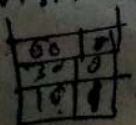
①



②

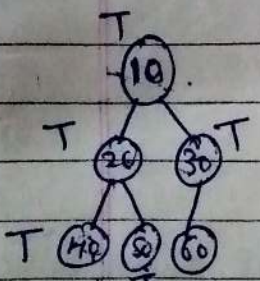


Visit 40: visit 50 visit 20

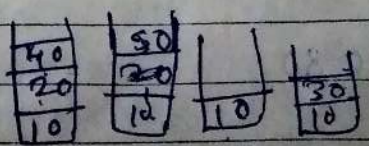


* Non-recursive preorder :-

```
Void preorder_nonrec (node* tree *T)
{
    stack s;
    while(T!=NULL)
    {
      cout << 'in' << T->data;
       s.push(T);
       T=T->lptr;
    }

   while( !s.empty())
   {
    T=s.pop();
    T=T->rptr;
      while(T!=NULL)
       {  cout << "in" << T->data;
          s.push(T);
          T=T->lptr;
       }
   }
}
```



10 20 40 50 30 60.

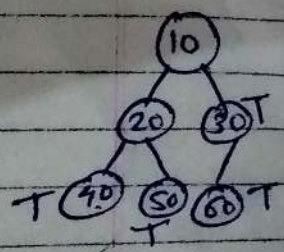


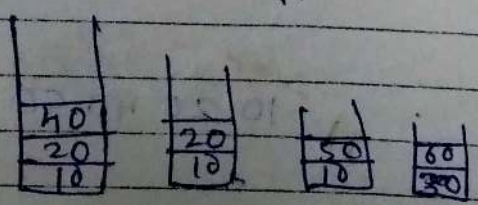
10 20 40 50 30 60

T=40 T=20 T=10

* Non-recursive inorder :-

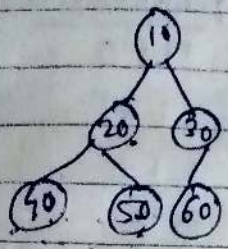
```
Void inorder_nonrec (tree *T)
{   stack S;
    while (T != NULL)
    {   S.push(T);
        T = T->lptr;
    }
    while (!S.empty())
    {   T = S.pop();
        cout << "in" << T->data;
        T = T->rptr;
        while (T != NULL)
        {   S.push()
            T = T->lptr;
        }
    }
}
```

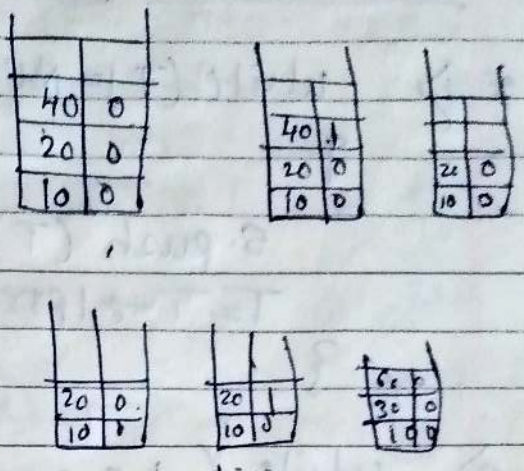




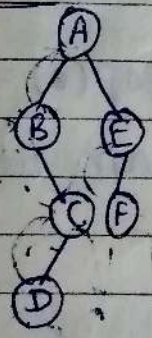
= 40 20 50 10 60 30

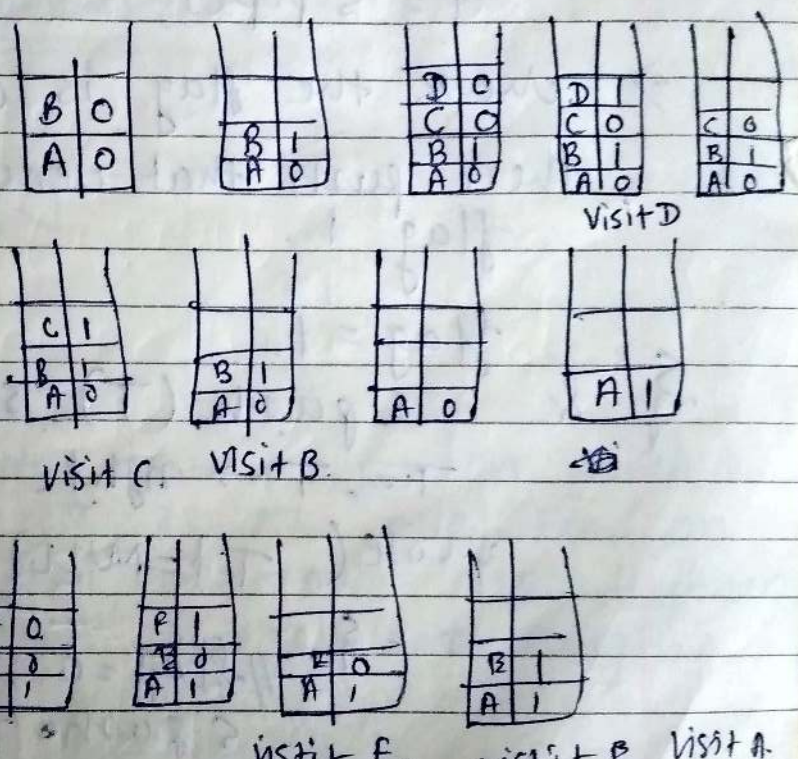
Non-recursive postorder :- 40 50 20





Example :





⇒ D C B F E A.

1) while going to left, addrs. of the treenode along with flag=0 is pushed onto the stack
2) During backtracking to traverse the right subtree an element is popped & if flag is 0 then it is pushed back with flag = 1.
3) when we return from right subtree & flag = 1 then we print that node

Non-recursive postorder traversal :-

* 1) while (T != NULL)
{
s.push (T); s1.push (NULL)
T = T → lptr;
}

2) while (! s.empty())
{
T = s.pop();

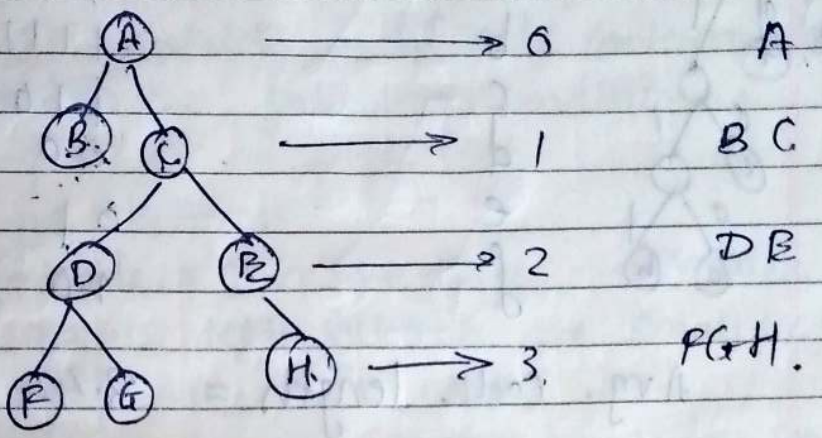
3) check the flag is 0 if (s1.pop() = NULL) then push that element again with flag 1.
flag = 1.

{ & s.push (T); s1.push ((node *)1);
T = T → right.
while (T != NULL)
{ //flag = 0; s1.push (NULL);
s.push (T);
T = T → lptr;
}

4) else if then flag = 1
p.p & print the data

5) end

* Binary tree traversal (BFS) :-
Breadth first traversal.



* Huffman algorithm :

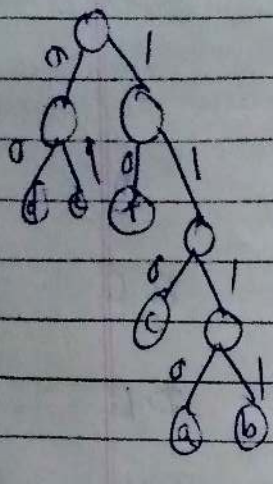
Huffman coding :-

example :-

· Suppose character a, b, c, d, e, f have probabilities .07, .09, .12, .22, .23, .27 resp. Find an optimal Huffman code and draw Huffman tree. What is the average code length?

⇒ (1)

(a) (b) (c) (d) (e) (f)
.07 .09 .12 .22 .23 .27.



Symbols	Huffman Code.
a	1110
b	1111
c	110
d	00
e	01
f	10

Avg. code length = 3.44 bits.

Q.2) a 0.4 b = 0.2 c 0.35 d = 0.05
Find huffman code. & draw huffman tree what is the code length?

→

a	0
b	10
c	110
d	111

Algorithm:-

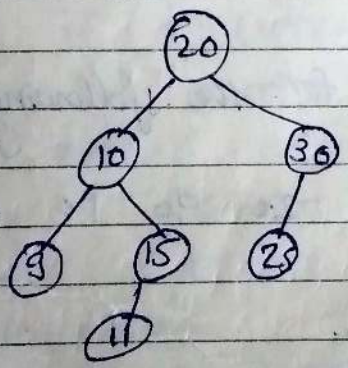
1. Each character is converted into a single node tree. Each node is labled by its frequency.
2. The weight of the tree is equal to sum of the frequencies of its leaves.
3. Select the two trees in the forest that have smallest weights. Combine these two trees into one. weight of the combined tree = sum of the weights of the two trees.

* Binary Search Tree (BST) :-

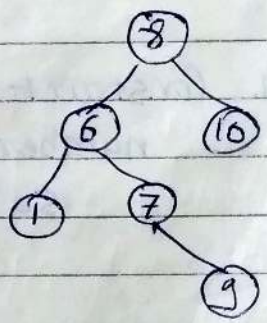
A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies the following conditions:

1) All keys 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 the root.
3) for every node x, in the tree, the values of all the keys in its right subtree are larger than the key value in the root
4) The left & right subtrees are also binary search tree.

Binary Search Tree finds its application in Searching.



It is a BST.



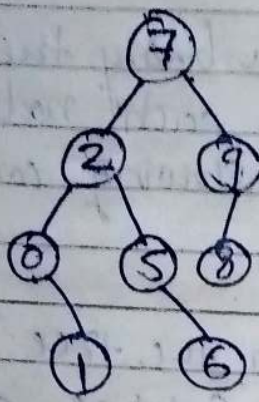
Not a BST.

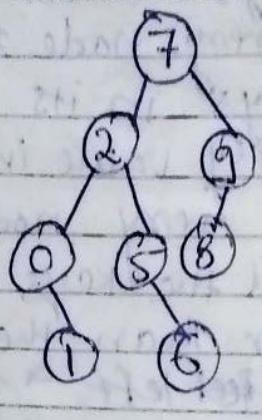
* Operations of Binary Search Tree:-

① Insert operation:-

⇒

Insert 7, 2, 9, 0, 5, 6, 8, 1. into BST.

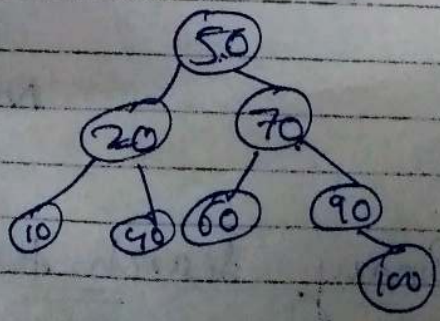




BST Complexity :- O(nlogn)

Q.1 Construct a BST for the following sequence of numbers :-

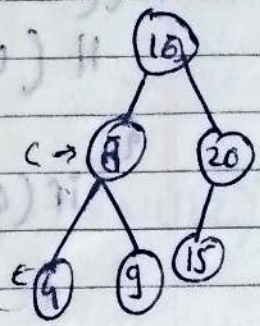
50 70 60 20 90 10 40 100



Q.2. 10, 08, 15, 12, 13, 07, 09, 17, 20, 18, 04, 05

(2) Delete operation :-

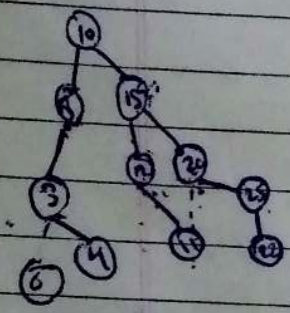
```
→ void BST :: delete (int d)
{
  int flag = 0;
  tree * curr, * parent;
  curr = root;
  while (curr != NULL)
  {
    if (curr -> data == d)
    {
      flag = 1;
      break;
    }
    else
    {
      parent = current;
      if (d > curr -> data)
        curr = curr -> rptr;
      else
        curr = curr -> lptr;
    }
  }
  if (flag == 0)
  { cout << "Data not found" <<
    return; }

// leaf node deletion.

if ((curr -> lptr == NULL) && (curr -> rptr == NULL))
  {
    if (parent -> lptr == NULL)
      parent -> lptr = NULL;
    else
      parent -> rptr = NULL;
    delete curr;
    return;
  }
```

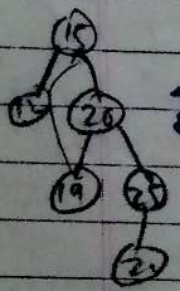


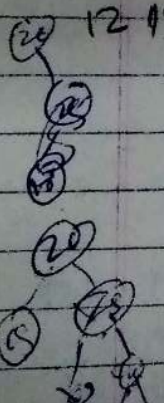
P = 10

// Node with single child.

```
if((curr->lptr == NULL && curr->rptr != NULL)
   || (curr->lptr != NULL && curr->rptr == NULL))
{
  if (curr->lptr == NULL && (curr->rptr != NULL)
  {
    if ( Parent->left == curr)
    {
      parent->left = current->rptr;
      delete curr;
    }
    else
    {
      Parent->rptr = curr->rptr;
      delete curr;
    }
  }
  else  // left child present, no right child.
  {
    if ( parent->lptr == curr)
    {
      Parent->lptr = curr->lptr;
      delete curr;
    }
    else
    {
      parent->rptr = curr->lptr;
      delete curr;
    }
  }
  return;
}
```



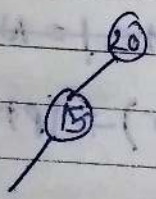


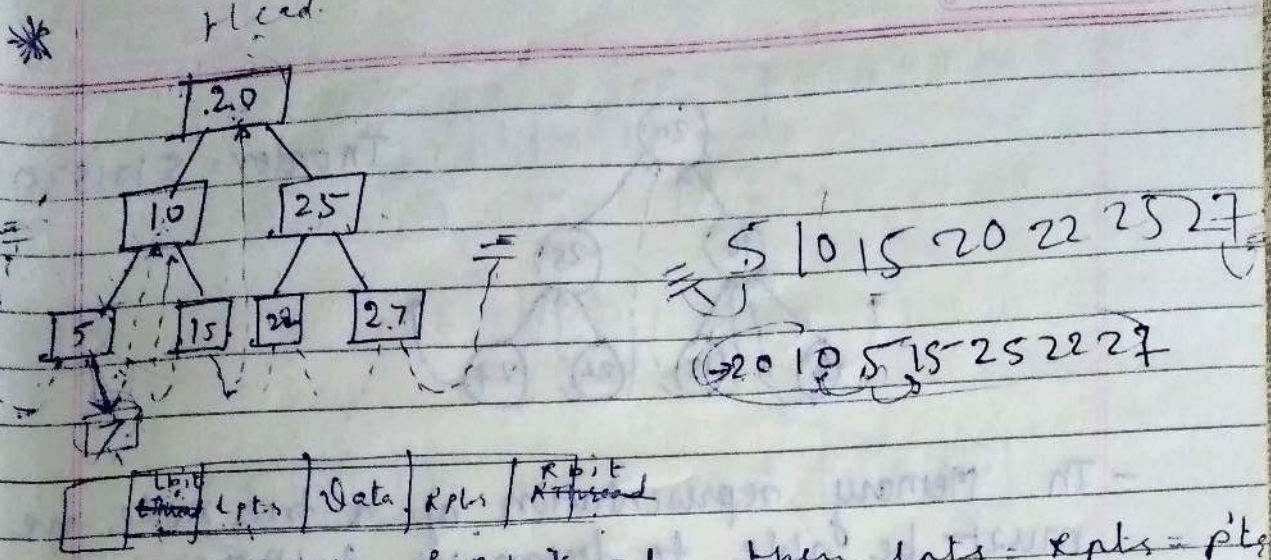


// Node with 2 children

```
if(curr->lptr != NULL && curr->rptr != NULL)
{
    tree * ch;
    ch = curr->right;
    if( (ch->lptr == NULL && ch->rptr = NULL))
    {
        curr = ch;
        delete ch;
        curr->rptr = NULL;
    }
    else. // right child has children
    {
      if((curr->rptr != NULL
      if((curr->rptr)->lptr != NULL )
      {
        tree * lcur, *lcurp;
        lcurp = curr->rptr;
        lcur = (curr->rptr)->lptr;
        while(lcurr->left != NULL)
            {
              lcurp = lcur;
              lcurr = lcurr->left;
            }
        curr->data = lcurr->data;
        delete lcurr;
        lcurp->left = NULL;
    }
```

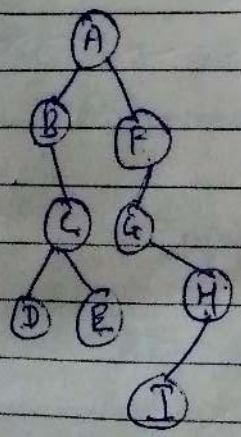
```
}
else
{   tree_node *temp;
    temp = curr->~~rptr~~; rptr;
    curr->data = temp->data;
    ~~delete temp~~
    curr->rptr = temp->rptr;
    delete temp;
}   }
return;
}
```





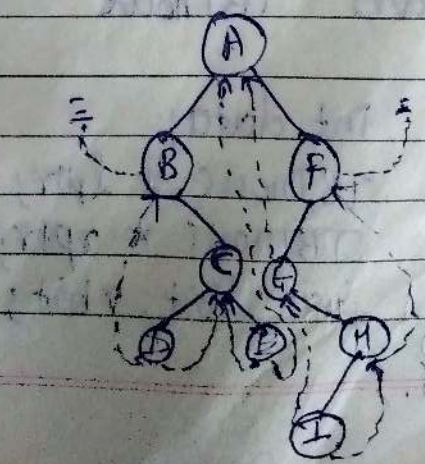
* Threaded Binary Trees (TBT) :-

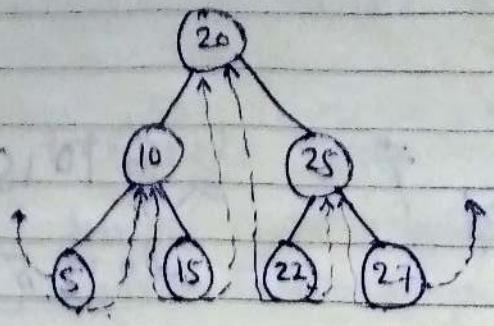
- In a linked representation of a binary tree, there are more null links than actual pointers.
- These null links can be replaced by pointers, called threads to other nodes.
- A left null link of a node is replaced with the address of its inorder predecessor.
- Similarly a right null link of a node is replaced with the address of its inorder successor.



~~It is Inorder :- BDCEAIHGF~~

Inorder :- B D C E A G I H F.



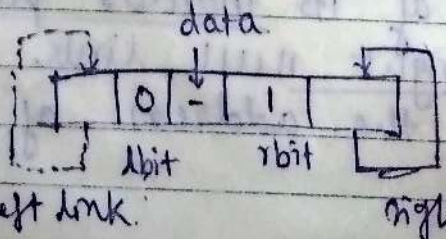


Inorder = 5 10 15 20 22 25 27

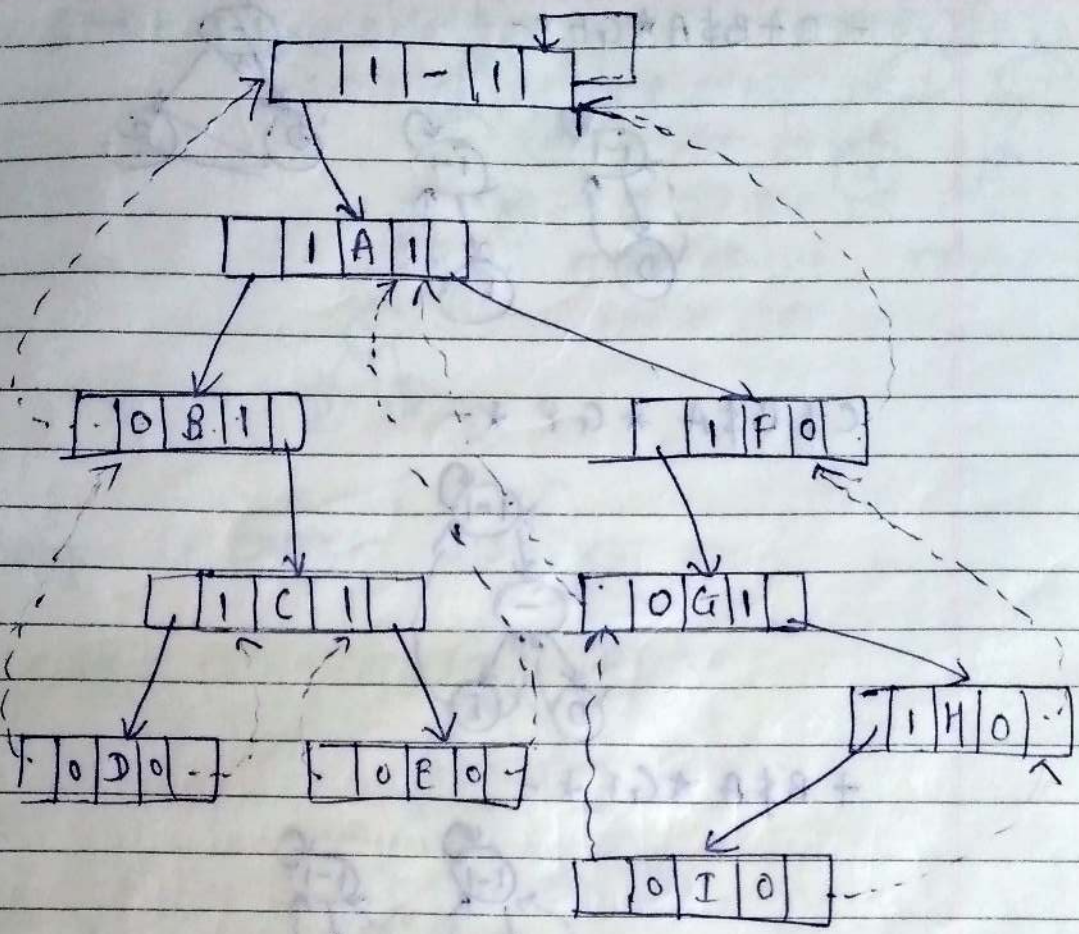
- In Memory representation of a tree node we must be able to distinguish betn threads & normal pointers.
  i.e. Using extra bits rbit & lbit.

lbit & rbit = 1 → left & right child is normal.

lbit & rbit = 0 → left & right links are replaced with a thread.




```
class TBTnode
{
    int data;
    TBTnode * lptr;
    TBTnode * rptr;
    int lbit, rbit;
};
```



Q: Construct inorder threaded binary tree for the following data:

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

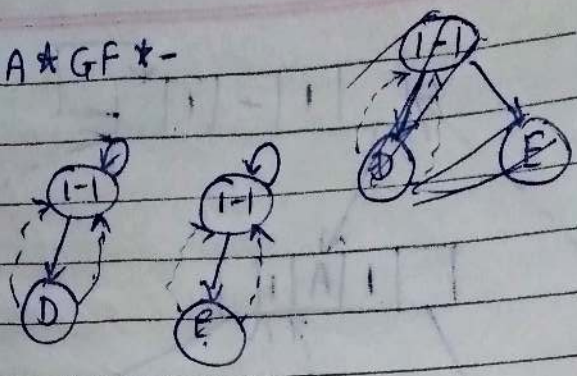
postorder: DF-C+B$A*GF*-

write it's pre-order traversal. Represent stepwise construction.

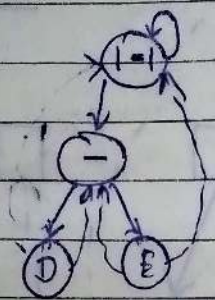
⇒ Inputs.

DE-C+B$A*GF*-

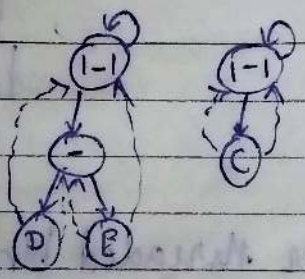
- C + B $ A * G F * -



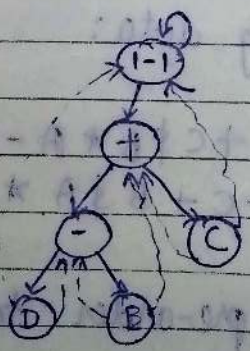
C + B $ A * G F * -



+ B $ A * G F * -

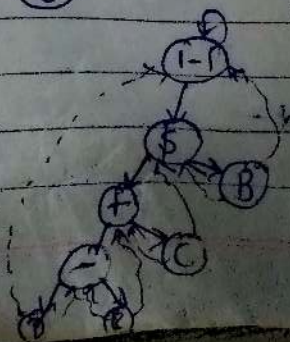


B $ A * G F * -

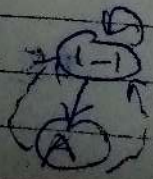


$ A + G F * -

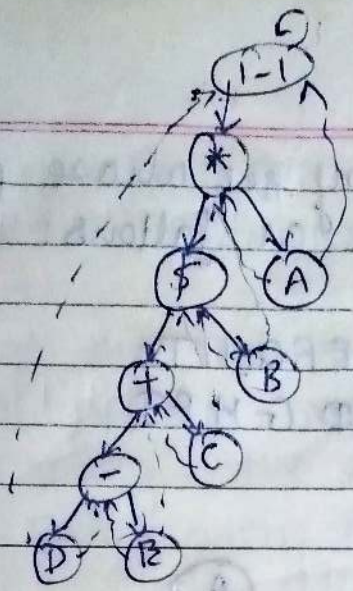
A + G F * -



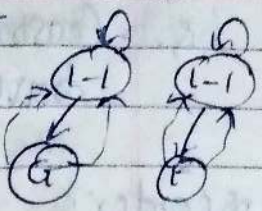
A G F * -



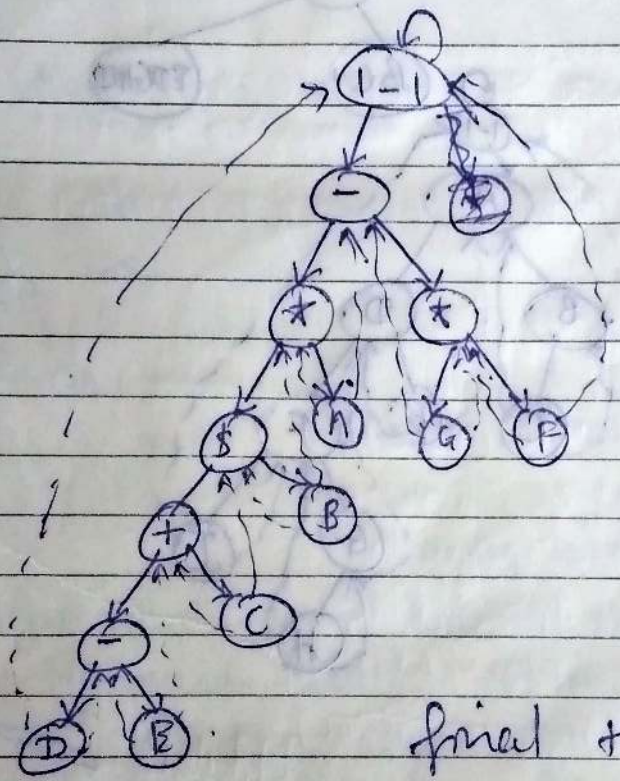
GF*-



*-



empty.

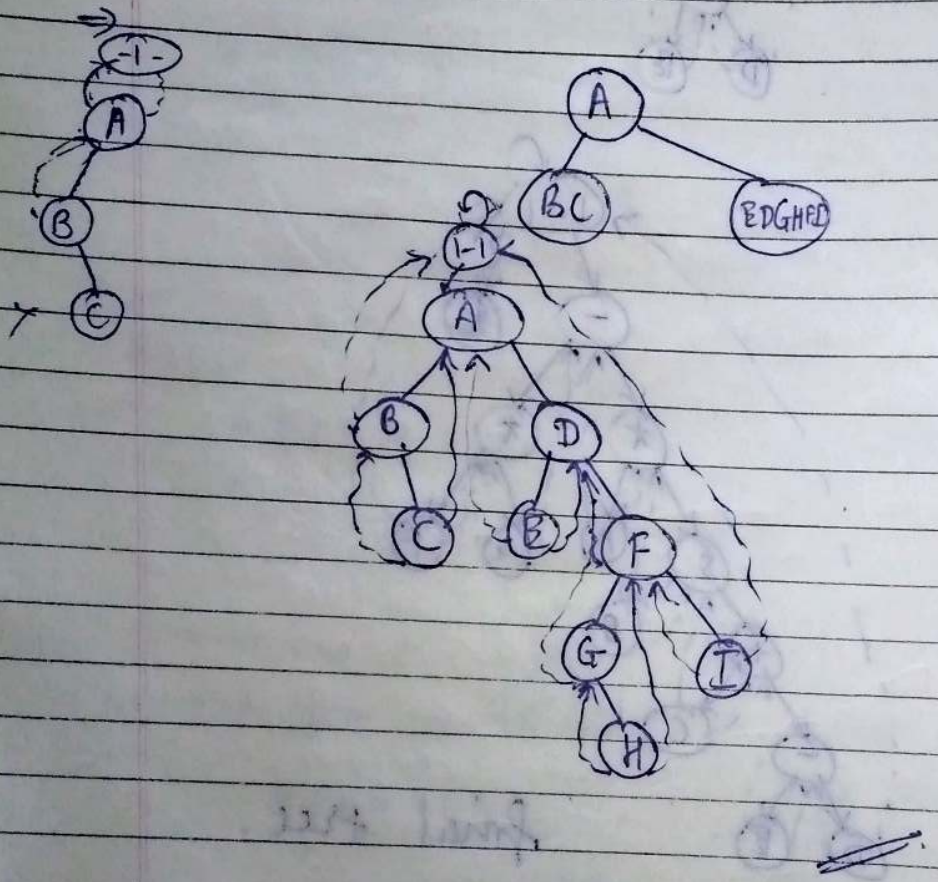


final tree.

preorder :- -*$+-DECBA*GF

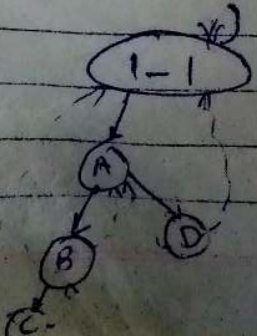
Q. Construct a binary tree whose preorder & inorder traversals are as follows:

preorder: A B C D E F G H I
Inorder: B C A E D G H F I



* Preorder traversal (TBT) :-

→ A non-recursive preorder traversal on TBT can be implemented without a stack.

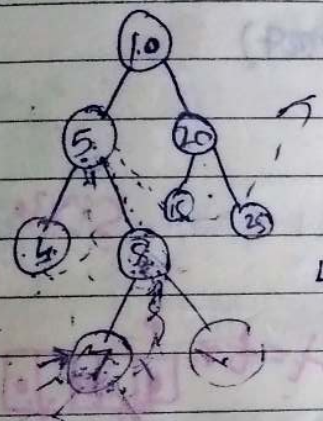


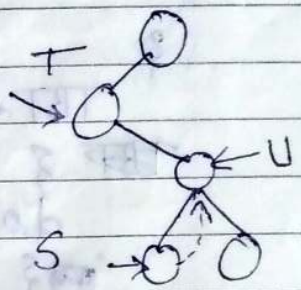
preorder = A B C D.

1) If the left child of a node is a normal child then left child is preorder successor.

2) If left child of the node is a thread. we can locate the preorder successor by climbing up the tree using the right thread till we reach a node with normal right child.

* Inorder traversal of a TBT :-





4 5 7 8 10 15 20 25

postorder = 4 7 8 5 15 25 20 10

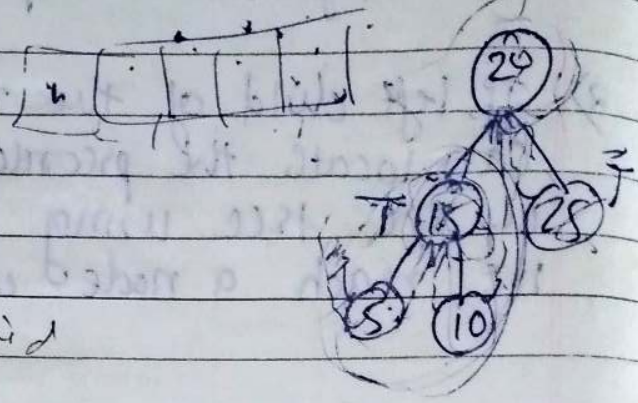
→ If the right child of a node is not a thread then the leftmost child in the right subtree will be the inorder successor.

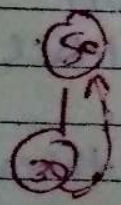
→ If the right link is a thread then the thread itself points to the successor.

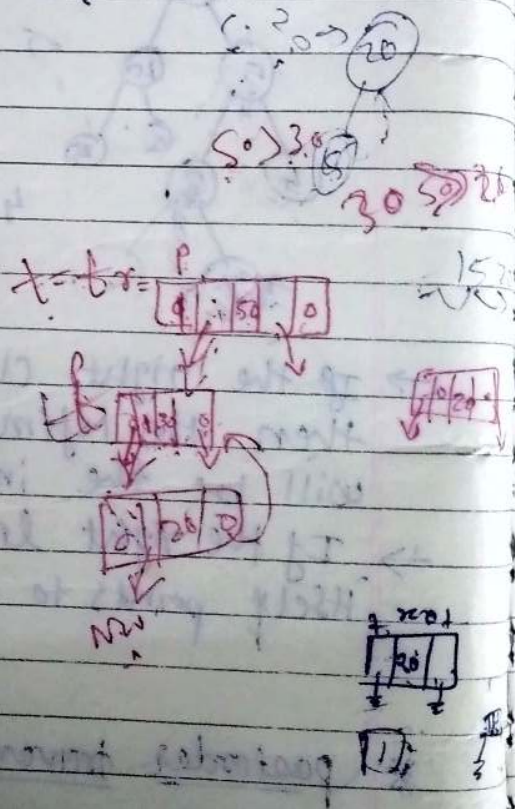
* postorder traversal of a TBT :-

* Threaded tree Creation & insertion method :-

```
class TBT
{
    int data;
    TBT *lptr;
    TBT *rptr;
    int lbit, rbit;
public:
    TBT *Create(TBT *root);
};
TBT * TBT :: Create(TBT *root)
{
    do
    {
        p = new(tree);
        cout << "Enter data";
        cin >> p->data;
        p->lptr = NULL;
        p->rptr = NULL;
        if (root == NULL)
        {
            root = p;
            root->lbit = 0
            root->rbit = 0;
        }
        else
        {
            temp = root;
            f = 0;
            while ((temp && f == 0))
            {
                if (*tmp->data > p->data)
```



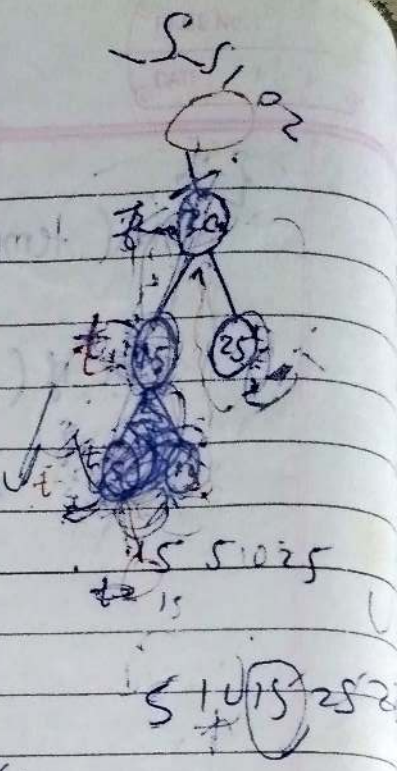




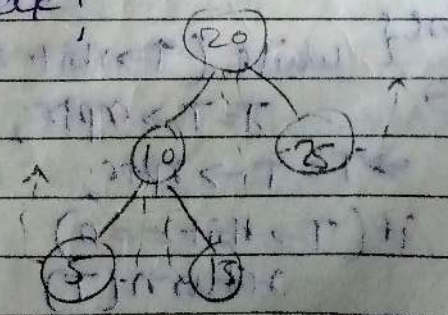
```
{
    if ( temp -> lbit == 1)
            temp = temp -> lptr;
    else if ( temp -> lbit == 0)
        {
            temp -> lbit = 1;
            p -> lbit = 0;
            p -> rbit = 0;
            p -> lptr = temp -> lptr;
            p -> rptr = temp;
            temp -> lptr = p;
            f = 1;
            p -> child = 0
        }
}
```


```
if ( temp -> data < p -> data)
    {
        if ( temp -> rbit == 1 )
                temp = temp -> rptr;
        else if ( temp -> rbit == 0)
            {
                temp -> rbit = 1;
                p -> lbit = 0;
                p -> rbit = 0;
                p -> rptr = temp -> rptr;
                p -> ~~lchild~~ lptr = temp;
                temp -> rptr = p;
                f = 1;
                p -> display();
            }
    }
}
cout << " do you want to add more nodes: ";
    c = getche();
} while ( c == 'y' || c == 'Y');
return (root);
}
```

```
void TBT :: pre(TBT * root)
{
   TBT *temp;
   temp = root;
   root = root -> lptr;
   int flag = 1
   while(1)
   {
   cout << temp -> data;
     while (temp -> lbit == 1)
       { temp = temp -> lptr;
         cout << temp -> data;
       }
     flag = 1;
     while ( temp -> rchild != NULL)
     {
       if ( temp -> rbit == 0)
             temp = temp -> rptr;
       else
         {
             temp = temp -> rptr;
             flag = 0;
             break;
         }
     }
     if (flag == 1)
           break;
   }
}
```



```
void TBT :: In (TBT *root)
{
  temp = root;
  while(1)
  {while ( temp -> lbit == 1)
        temp = temp -> lptr;

   cout << temp -> data;
   flag = 1;
   while ( temp -> rptr != NULL)
   {
        if ( temp -> rbit == 0)
        {
            temp = temp -> rptr;
            cout << temp -> data;
        }
        else
          { temp = temp -> rptr;
              flag = 0;
              break;
          }
   }
   if ( flag == 1)
      break;
  }
}
```



pre :- 20 10 5 15 25

post 5 10 15 25 20

```
void TBT :: post(TBT *root)
{
  TBT *temp, *T;
  T = root;
  while (T->lbit==1 || T->rbit==1)
  {
    if (T->lbit==1)
      T = T->lptr;
    else
      T = T->rptr;
  }
  while (T != root)
  {
    cout << "\n" << T->data;
    T = post_scc(T);
  }
  cout << "\n" << root->data;
}
```

// function for post order successor of a node.

```
TBT *post_scc(TBT *T)
{
  if (T->child == 1)
  {
    while (T->lbit == 1)
      T = T->lptr;
    return (T->lptr);
  }
  else
  {
    while (T->rbit == 1)
      T = T->rptr;
    T = T->rptr;
    if (T->rbit == 0)
      return (T);
    else
    {
      T = T->rptr;
      while (T->lbit == 1 || T->rbit == 1)
      {
        if (T->lbit == 1)
          T = T->lptr;
        else
          T = T->rptr;
      }
      return (T);
    }
  }
}
```

child=1 right child.
0 = left child.

// locate parent of T when it is left child

* Applications of tree :-

1) Game tree :-

→ one of the most important application of trees is Game tree:

→ Games like tic-tac-toe, chess, 8-puzzle checkers, 16-puzzle checkers etc. are solved by constructing the game tree.

for example 8-puzzle problem.

2	8	3
1	6	4
7	_	5

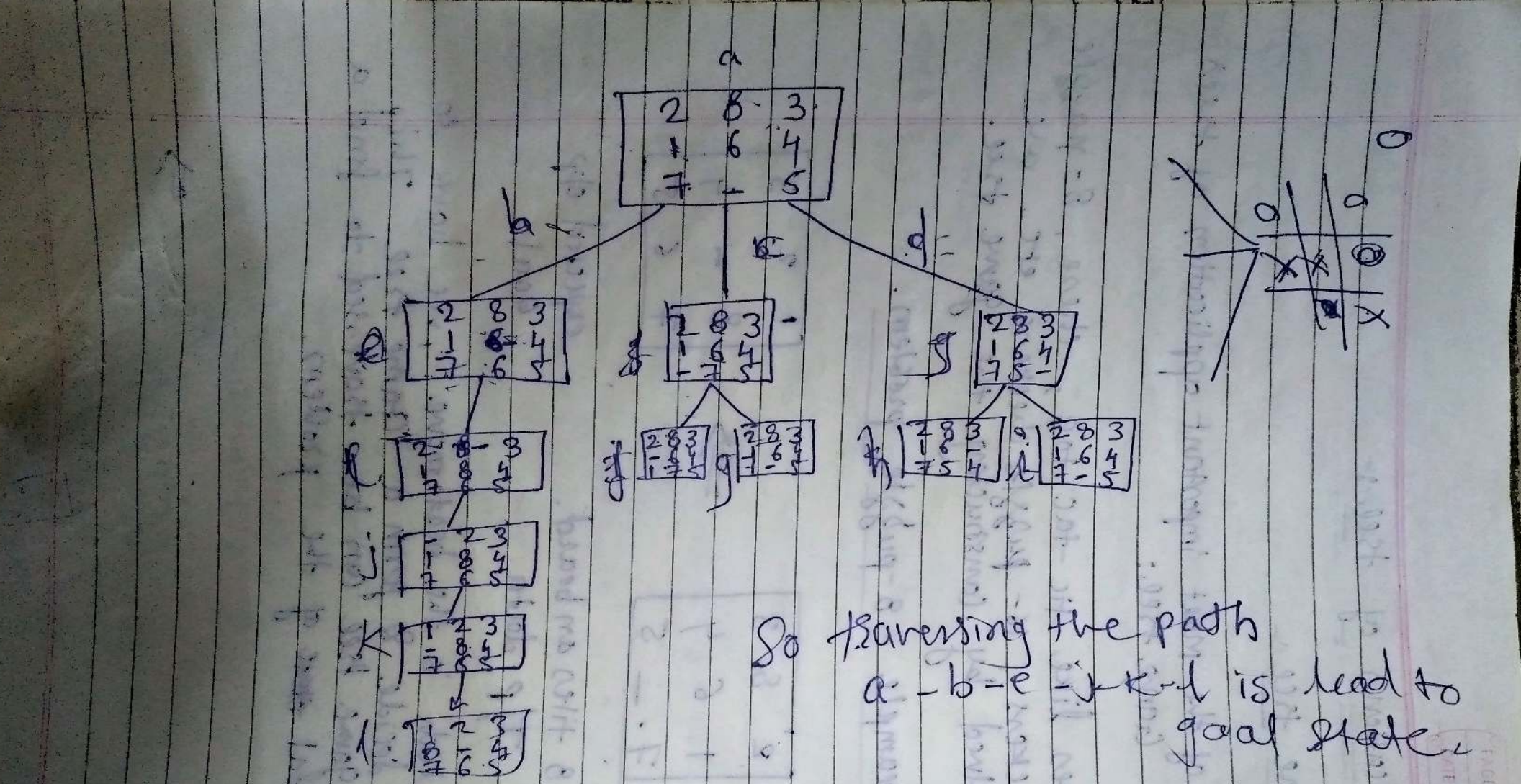
⟹

1	2	3
8	_	4
7	6	5

8 tiles on board. . expected o/p.
I/p data. Goal.

which is the best move, we have to decide & form a Game tree. This game tree can be traversed to find a stated goal ~~state~~ of the problem.

⇒


a
b
c
d
e
f
g
h
i
j
k
l
So traversing the path
a - b - e - j - k - l is lead to
goal state.

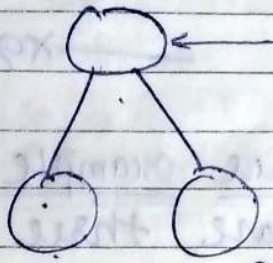
② Decision tree :-

→ Imp. application is decision making.

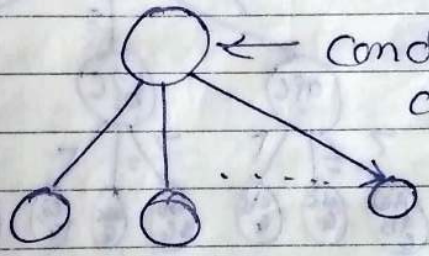
→ Most of the program involve conditional stmt rather ~~that~~ than just i/p & o/p.

For example :-

① If clause :-



② Switch case :-



adv :-

1) It provide a mechanism for visualizing the code execution.

2) It also provides the framework to analyse how general ~~&~~ algorithm work.

③ Expression trees :-

→ Traversals in expression trees will give you the result of postfix, infix & prefix.

→ E tree consist of opsands & operations. where operands are leaves of a tree.

—— xox ——

* Decision tree example :-

Compare three nos a, b, c