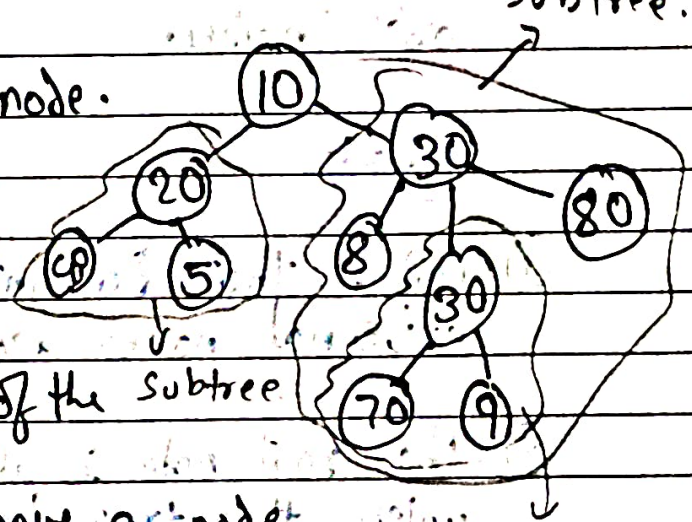
# TREE

⇒ Tree Data Structure :- (Non-linear DS)

Consider a problem where you need to represent ~~Her~~ Hierarchy for example organisation structure You want to represent organisation structure using ~~a memory~~ data structure in memory.

Another example is folder inside a folder & ~~may~~ many more folders inside that folders.

Previous all data structure use linear data structure tree uses hierarchical data structure.
Tree data structure is recursive in nature.




- Node :- All individuals are node.
- root :- 10 is root node.
- leaf :- 40, 5, 8, 70, 9, 80
  nothing below them.
- child :- node below a node is called children of the node.
- Parent :- node which is above a node is called parent node.
- SubTree :- There are many subtree inside a tree.
- descendants :- descendants of a node are all the nodes that layes in the subtree with this particular node as root.

example :- 40 & 5 are descendats of 20.

- ancestors:- The ancestors of a node include that node, the parent of that node, the grand parent of that node & so on up to the root.

- degree:- Number of node just below to that node

example:-

Degree of 10 is 2
Degree of 20 is 2
Degree of 30 is 3
Degree of 30 is 2
All the leaf node have degree zero.

- internal nodes:- An internal node in a tree is a node that has at least one child node. They are also know as inner nodes, branch nodes, or inodes.

Note:-

(1) Nodes that do not have child nodes are called external nodes leaf nodes, or terminal nodes.

(2) The root node is also considered an internal node, unless the tree has only one node.

(3) The nodes in a tree can be organized into levels based on how many edges away they are from the root.

④ The height of a node is the length of the longest path from that node to a leaf.

⑤ The connections b/w two nodes are called as edge.

⇒ Types of Tree :-

① Binary Tree :- Each node has at most 2 children
② Binary search Tree :- A binary tree where the left child contains smaller values, & the right child contains larger values.
③ Full Binary Tree :- Every node has 0 or 2 children.
④ Complete Binary Tree.
⑤ Perfect Binary Tree.
⑥ Balanced Tree.
⑦ AVL Tree.
⑧ B-Tree.

⇒ Application of Tree :-

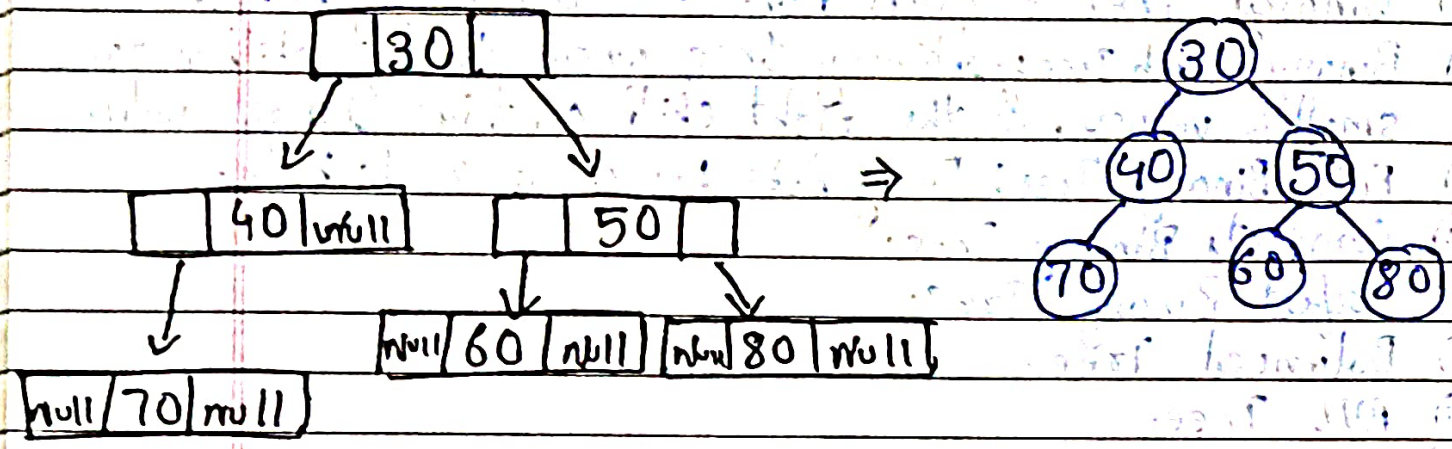
⇒ To represent hierarchical data
- → Organization Structure
- → folder Structure
- → xml / HTML content (JSON object)
- → In OOP (Inheritance)

⇒ Binary search Tree.
⇒ Binary Heap.
⇒ B & B+ Tree in DBMS
⇒ Spanning & shortest path tree in computer networks
⇒ Parse Tree, Expression Tree in Compilers.

⇒ Trie:
⇒ Suffix Tree.
⇒ Binary Index Tree.
⇒ Segment Tree.

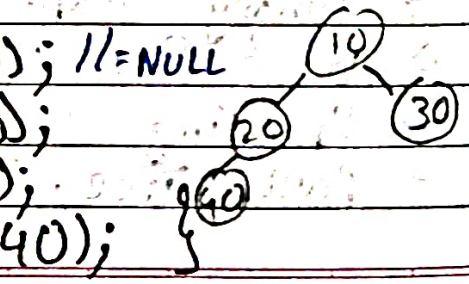
## ⊛ Binary Tree:-

Every node has at-most two children.




So, most of the praticully used & most popular data structure are variations of Binary tree.

⇒
```
struct Node {
  int Key; Node *left; Node *right;
  Node (int K) {
      Key = K; left = right = NULL; }
};

int main() {
  Node *root = new Node(10); //=NULL
  root -> left = new Node (20);
  root -> right = new Node (30);
  root -> left -> left = new Node(40); }
```



## ⊛ Tree Traversal :-

Breadth First (or level Order)

Depth first
① → Inorder. (left - Root - Right)
② → Preorder. (Root - left - Right)
③ → Postorder. (left - Right - Root)

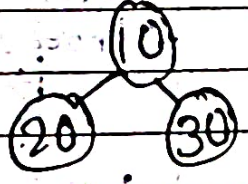
In Depth first traversal we go to one side we finish it completely or traverse it completly. then we go to other side. It can be understood by Recursion.

Recursive { ① Traverse Root. ② Traverse left subtree. ③ Traverse Right Subtree. } steps.

left → left Sub-tree
Right → Right Sub-tree.

We have to do this steps to get the all node to traverse so there are many way actually there are (3!) solution so there are 6 solution but we choose the 3 solution. In which left traverse is first. they are: Preorder, Inorder, Post order.

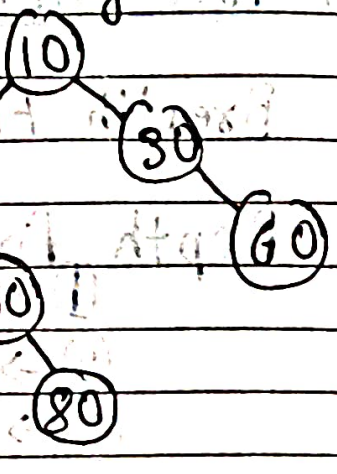
example :-




Inorder :- 20 10 30
Preorder :- 10 20 30
Postorder :- 20 30 10

- Inorder:- (Left-~~Right-Root~~) - Root - Right

40 20 70 50 80 10 30 60




- Preorder:- (Root - Left - Right)

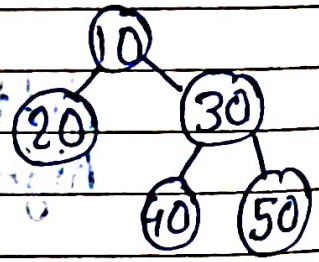
10 20 40 50 70 80 30 60

- Post order:- (Left - Right - Root)

40 70 80 50 20 60 30 10

## (*) Implementation of Inorder traversal:-

Inorder:- 20 10 40 30 50




```
void inorder (Node *root) {
if (root != NULL)
{ inorder (root->left);
   cout << (root->key) << " ";
   inorder (root->right);
}
}
```

inorder(10)
→ inorder(20)
→ inorder (NULL)
→ Print (20)
→ inorder (NULL)
→ Print (10)
→ inorder(40)
→ inorder (NULL)
→ Print (40)
→ inorder (NULL)
→ Print(30)

Time:- O(n)

Aux Space:- O(h+1) = O(h)

h = height of tree.

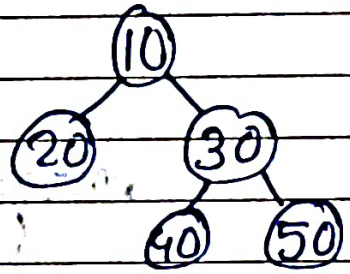
↳ inorder(50)
  ↳ inorder(NULL)
  ↳ Print (50)
  ↳ inorder(NULL)

## ⊛ Implementation of Preorder Traversal :-

```
void preorder(Node *root) {
if (root != NULL) {
    cout << (root->key) << " ";
    preorder(root->left);
    preorder(root->right); }
}
```

O(n)
Time :- O(n)
Space :- O(h)
h = height.

preorder :- 10 20 30 40 50



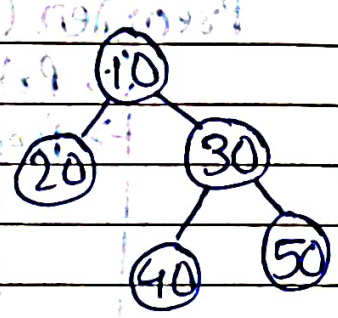

Preorder (10)
↳ Print (10)
↳ Preorder (20)
  ↳ Print (20)
  ↳ Preorder (NULL)
  ↳ Preorder (NULL)
↳ Preorder (30)
  ↳ Print (30)
  ↳ Preorder (40)
    ↳ Print (40)
    ↳ Preorder (NULL)
    ↳ Preorder (NULL)
  ↳ Preorder (50)
    ↳ Print (50)
    ↳ Preorder (NULL)
    ↳ Preorder (NULL)

## ⊛ Implementation of postorder Traversal :-

⇒
```
void postorder (Node *root) {
    if (root != NULL)
    { postorder (root->left);
      postorder (root->right);
      cout << (root->key) << " "; }
}
```

Time :- O(n) / θ(n)
Space :- O(n) / θ(h)

Postorder :- 20 40 50 30 10

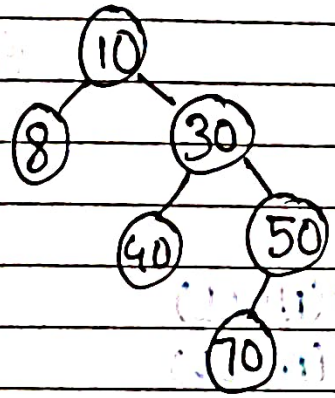



```
postorder (10)
  -> postorder (20)
       -> Postorder (NULL)
       -> Postorder (NULL)
       -> Print (20)
  -> Postorder (30)
       -> postorder (40)
            -> Postorder (NULL)
            -> postorder (NULL)
            -> Print (40)
       -> Postorder (50)
            -> Postorder (NULL)
            -> Postorder (NULL)
            -> Print (50)
       -> Print (30)
  -> print (10)
```

## (*) Height of a Binary Tree

The number of edges from the root node to the furthest leaf node. It's also know as the maximum depth of the tree.

1) I/P:-




O/P:- 4 or 3

↳ node ↳ edge

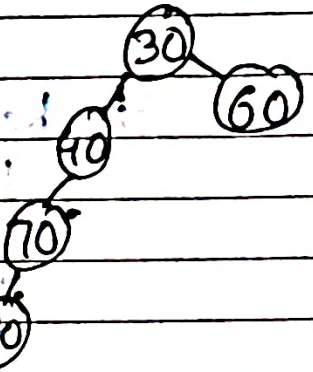
2) I/P:- (10)

O/P:- 1 or 0

3) I/P:- NULL (or null)

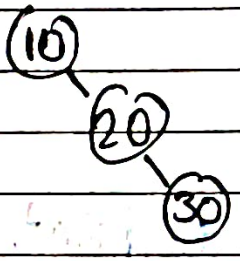
O/P:- 0 or -1

4) I/P:-




O/P:- 4 or 3

5) I/P:-




O/P:- 3 or 2

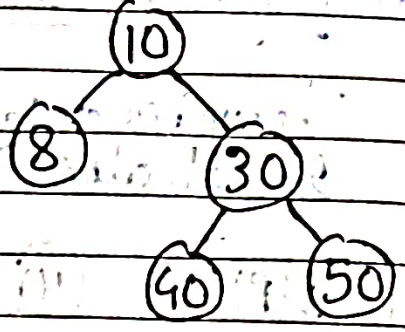
⇒

```
int height(Node *root) {
if (root == NULL)
   {return 0; }
else
  return max(height(root->left), height(root->right))
                +1 ;
   }
```

Time:- O(n) or θ(n)

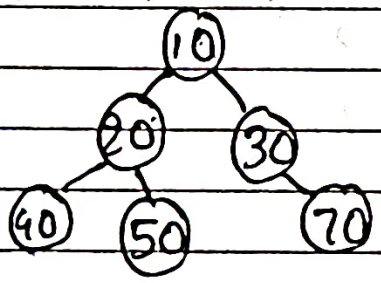
Space:- O(h) or (θ(h)

height (10)
↳ height (8)
↳ height (NULL)
↳ height (NULL)
↳ height (30)
↳ height (40)
↳ height (NULL)
↳ height (NULL)
↳ height (50)
↳ height (NULL)
↳ height (NULL)



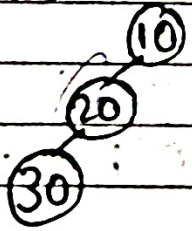

(*) Print Nodes at distance K.

1) I/P:- K = 2



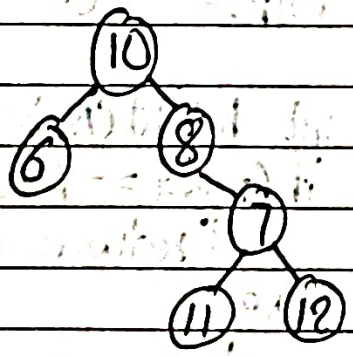

O/P:- 40 50 70

2) I/P:- K=1




O/P:- 20

3) I/P:- K = 3

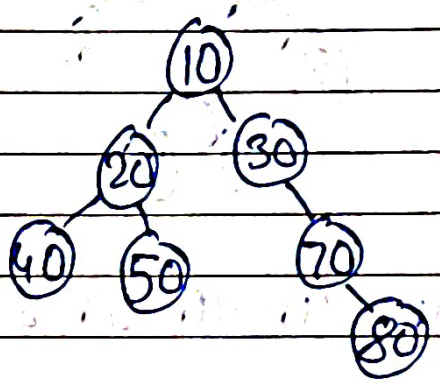



O/P:- 11 12

⇒ void printDist (Node *root, int k) {
if(root==NULL) { return; }
if(k==0) { cout << (root->key) << " "; }
else { printDist (root->left, k-1);
printDist (root->right, k-1); }
}

Time :- O(n)
Space :- O(h)
or θ(h)

I/P :- k = 2



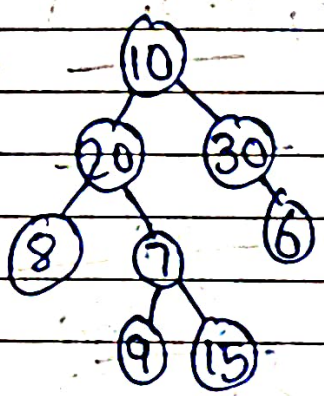

printDist(10, 2);
↳ printDist (20,1)
↳ print Dist(

O/P :- 40 50 70

printDist (10,2)
↳ Print Dist (20,1)
↳ printDist (40, 0)
↳ printDist (50, 0)
↳ print Dist (30,1)
↳ Print Dist (NULL, 0)
↳ print Dist (70, 0)

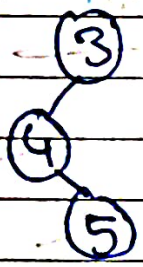
## ⊛ Breadth first Search
or
## level order Traversal

1) I/P:-




O/P:- 10 20 30 8 7 6 9 15

2) I/P:-




O/P:- 3 4 5

4) I/P:-


O/P:- 8 6 2 4 3

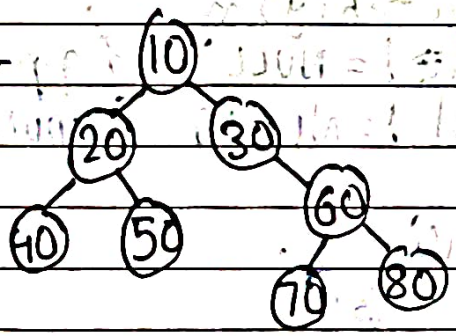
We are going to use queue data structure.

⇒
```
void printlevel (Node *root)
{ if (root == NULL) { return ;}
  queue <Node *> q ;
   q.push (root);
  while (q.empty == false) {
  Node *curr = q.front();
   q.pop();
  cout << (curr -> key) << " ";
  if (curr -> left != NULL)
      { q.push (curr -> left ); }
  if (curr -> right != NULL)
      { q.push (curr -> right); }
  }
}
```

Time :- $\theta(n)$
Space :- $\theta(n)$
or
$\theta(w)$
w = width.

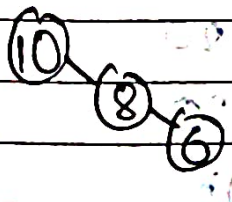
## ⊛ level order Traversal line by line :- (Method :- 01)

1) I/P :-



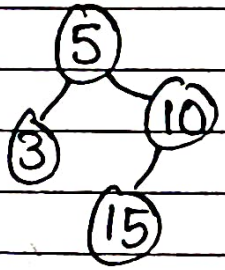

O/P :- 10
20 30
40 50 60
70 80

2)




O/P :- 10
8
6

3) I/P :-

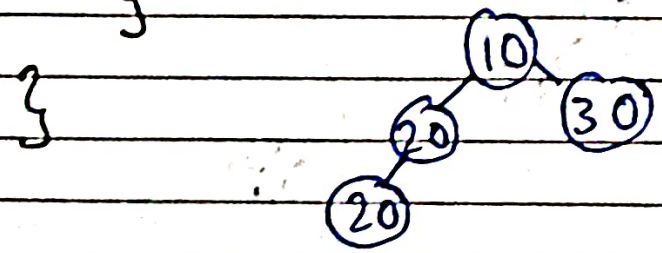



O/P :- 5
3 10
15

=> The soln is based on fact that when you traverse the last node of the level your next level is completely there into the queue.

=>

```
void printLevelOrderLine(Node(*root)
{ if(root==NULL) {return;}
  queue<Node*> q;
  q.push(root);          Time:- O(n+h)
  q.push(NULL);          Aux Space:- Θ(W)
  while(q.Size()>1)                ↳ width
  { Node *curr= q.front();
    q.pop();
    if(curr==NULL)
    { cout<<"\n";
      q.push(NULL);
      continue; }
    cout<<(curr->Key)<<" ";
    if(curr->left!=NULL) {q.push(curr->left)
    if(curr->right!=NULL) {q.push(curr->right)
  }
}
```

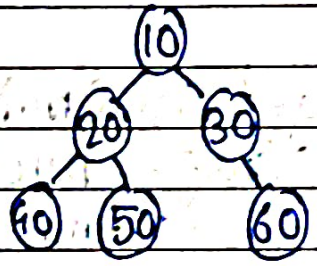


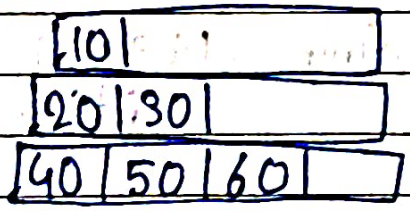

| 10 | NULL | .... | 10 |
|---|---|---|---|
| NULL | 20 | 30 | \n |
| 20 | 30 | NULL | 20 |
| 30 | NULL | 40 | 30 |
| NULL | 40 | .... | \n |
| 40 | NULL | .... | 40 |
| NULL | .... | | |

## ⊛ ~~for~~ level order traversal line by line :- (Method:-2)

⇒

```
void printLevelOrder(Node *root) {
if (root == NULL)  {return;}
queue<Node*> q;
q.push(root);
while (q.empty() == false) {
 int count = q.size;
 for (int i=0; i<count; i++)
{ Node *curr = q.front();
  q.pop();
  cout << (curr->key) << " ";
  if (curr -> left != NULL)  {q.push(curr->left);}
  if (curr-> right != NULL)  {q.push(curr->right);}
}
 cout << "\n";
}
}
```

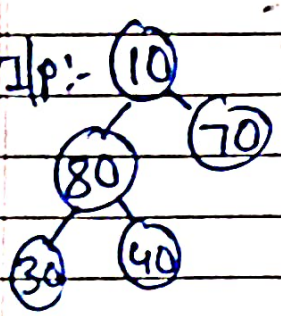



10 \n
20 30 \n.
40 50 60 \n

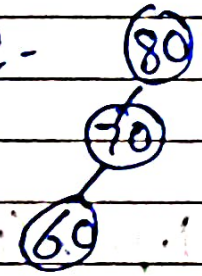
## ⊛ Size of a Binary Tree :-

1) I/P :-



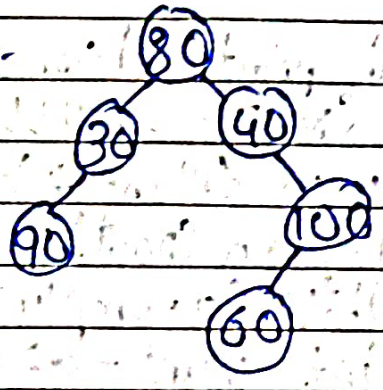

O/P :- 5

2) I/P :-




O/P :- 3

3) I/P :-



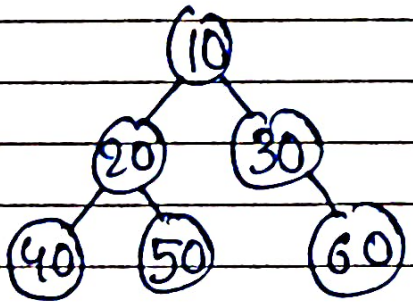

O/P :- 6

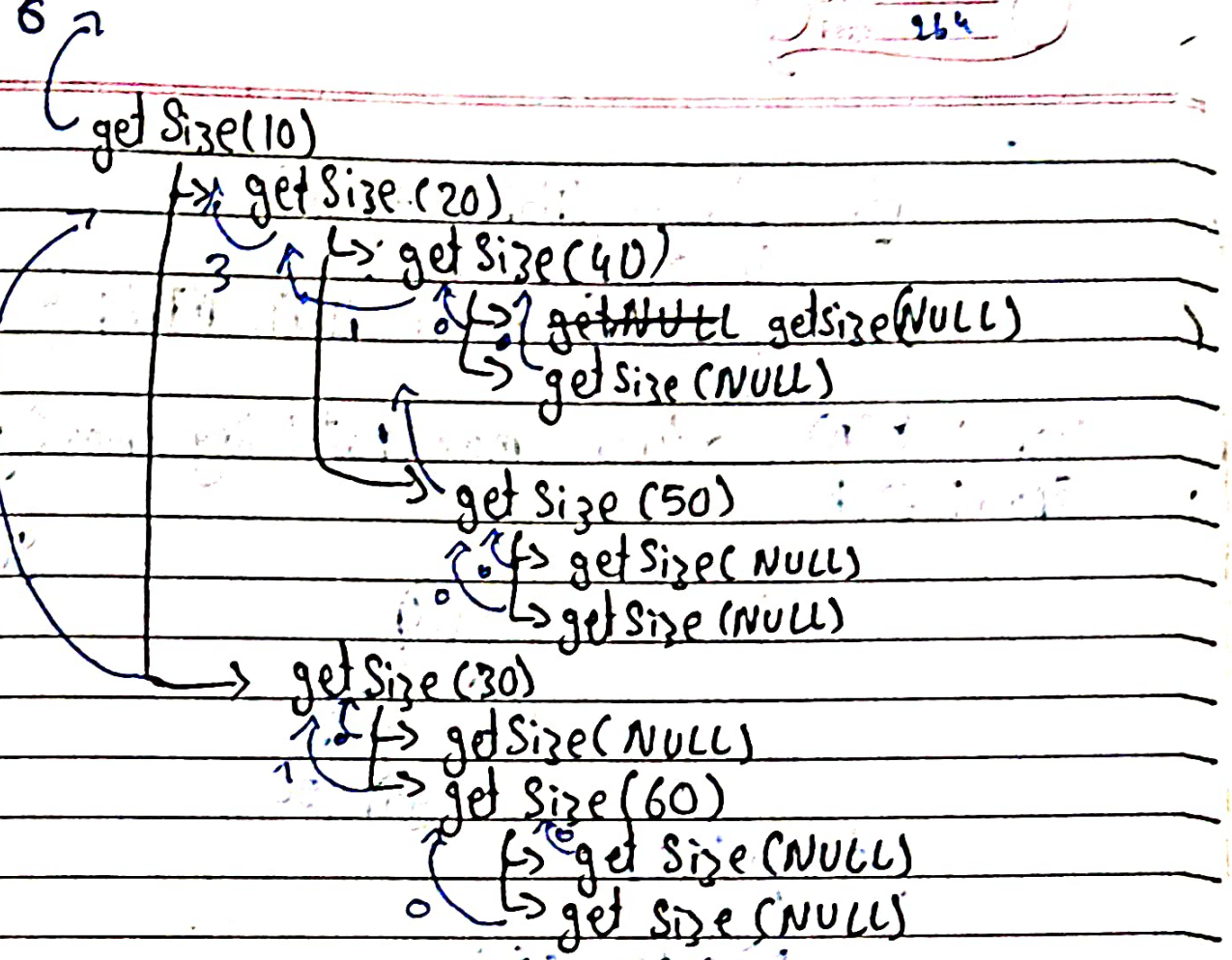
4) I/P :- NULL

O/P :- 0

⇒
```
int getSize(Node *root)
{ if (root == NULL)
        {return 0;}
else
    return ((1+getSize(root->left)+
             getSize(root->right));
}
```

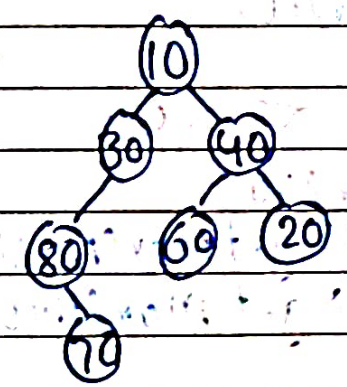
Time :- O(n)
Space :- O(h)






## ⊛ Maximum in Binary Tree:-

1) I/P:-

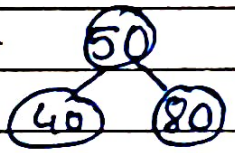



O/P:- 80

I/P:- NULL

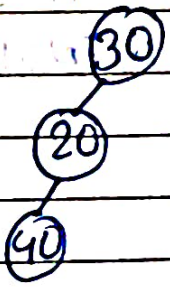
O/P:- -∞

I/P:-




O/P:- 80

2) I/P:-

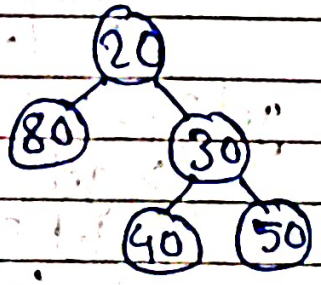



O/P:- 40

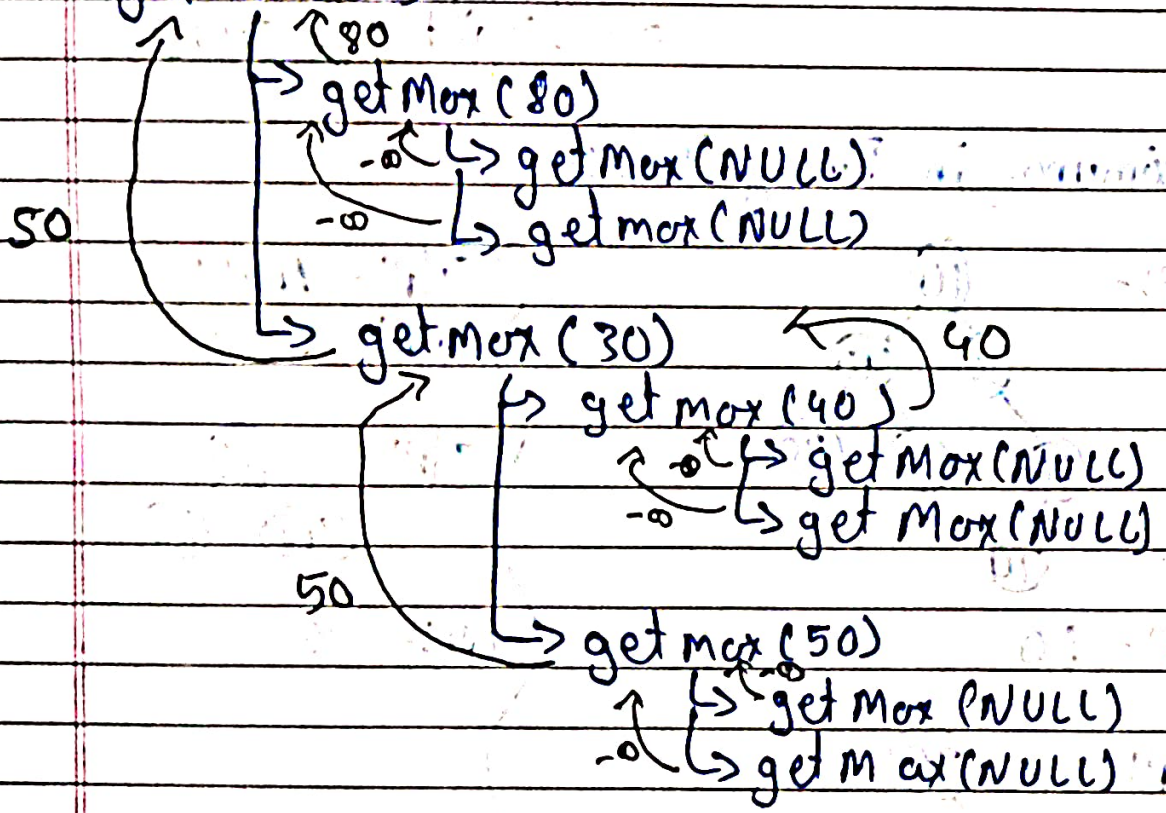
→

```
int getMax (Node *root)
{  if (root == NULL)
       {return INT_MIN; }
   else
   return max (root->key, max (getMax(root->left),
                               getMax(root->right)));
}
```

Time :- $\theta(n)$
Space :- $O(h)$




getMax(20)




max (20, 80, 50)

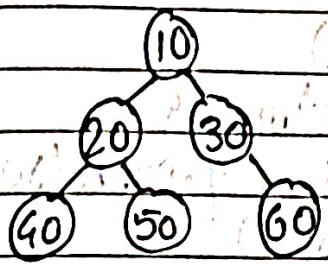
80

→ Therefore 80 as output.

*Write a iterative soln :- * use queue.

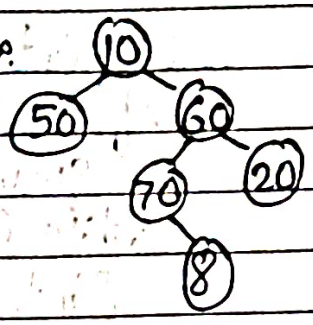
## ⊛ Print left view of Binary

1) I/P:-



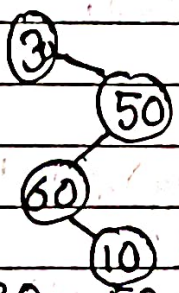

O/P:- 10 20 30 40

3) I/P:-




O/P:- 10 50 70 8

2) I/P:-




O/P:- 30 50 60 10

⇒ Method-I (Recursive)

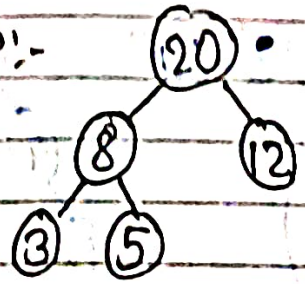
```
int maxlevel=0;
void printleft(Node *root, int level) {
if(root == NULL) {return;}
if (maxlevel < level) { cout << (root->Key) << " ";
                        maxlevel = level; }
printleft(root->left, level+1);
printleft (root->Right, level+1); }

void printleftview(Node *root)
    {
      Printleft (root, 1);
    }
```

⇒ Method-II (Iterative)

```
void printLeft(Node *root)
{ if(root == NULL) {return;}
    queue<Node *> q; q.push(root);
  while(q.empty() == false)
  {int count = q.size();
   for(int i=0; i<count; i++)
    { Node *curr = q.top();
     q.pop();
     if i=0 (i==0)
        { cout<<(curr->key)<<" "; }
    if(curr->left != NULL)
        {q.push(curr->left); }
    if(curr->Right != NULL)
        {q.push(curr->right);}
    }
  }
}
```

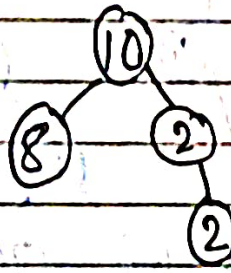
## ⊛ Children Sum property :-

1) I/P :-




O/P :- Yes

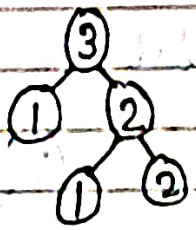
2) I/P :-




3) I/P :- 5 O/P :- Yes

4) I/P :- NULL O/P :- Yes

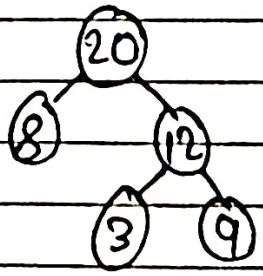
5)




```
=> bool isSum(Node *root) {
if (root == NULL) { return true; }
if (root->left == NULL && root->right == NULL)
        { return true; }
int sum = 0;
if (root->left != NULL) { sum += root->left->key; }
if (root->right != NULL) { sum += root->right->key; }

return (root->key == sum &&
        isSum(root->left) &&
        isSum(root->right));
}
```

Tim :- O(n)

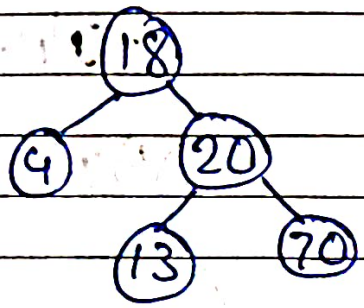
Spa :- O(h)




```
isSum(20)
   ↳ isSum(8)
   ↳ isSum(12)
        ↳ isSum(3)
        ↳ isSum(9)
```

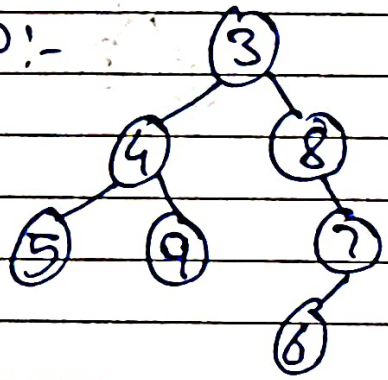
## ⊛ Check for Balanced Tree :-

1) I/P :-



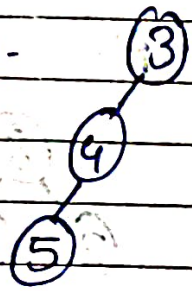

O/P :- Yes

2) I/P :-

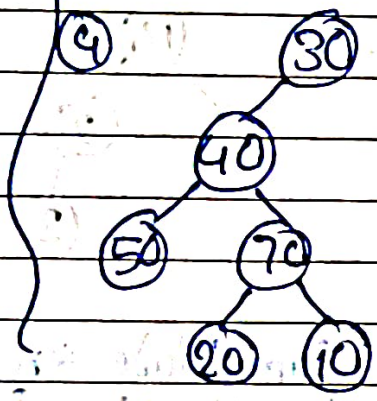



O/P :- NO.

3) I/P :-




O/P :- NO




⇒ Meaning of Height Balanced :-
The difference b/w right subtree & left subtree should not be more than One.

⇒ Naive Sol$^n$:- O(n²)

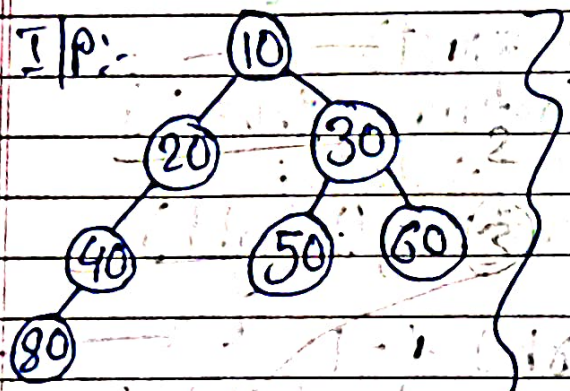
```
bool isBalanced (Node *root) {
if (root == NULL) {return true;}
  int lh = height (root -> left);
  int rh = height (root -> right);

  return (abs(lh - rh) <= 1 &&
          isBalanced (root -> left) &&
          isBalanced (root -> right));
}
```

⇒ Efficient Sol$^n$:- O(n)

```
int isBalanced (Node *root) {
if (root == NULL) return 0;
int lh = isBalanced (root -> left);   } check for left subtree &
if (lh == -1) {return -1;}            } Also get the left height

int rh = isBalanced (root -> right);  } check for right subtree
if (rh == -1) {return -1;}            } Also get the right height

if (abs(lh - rh) > 1) {return -1;}
else
   {return max(lh, rh) + 1;}
}
```

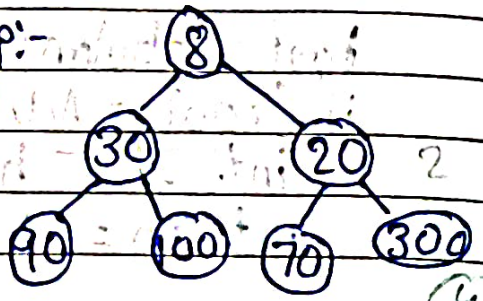
## ⊛ Maximum Width of Binary Tree:-

1) I/P:-



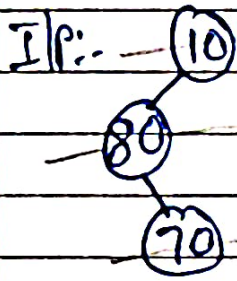

O/P:- 3

2) I/P:-




O/P:- 4

3) I/P:-




O/P:- 1

4) I/P:- NULL

O/P:- 0.

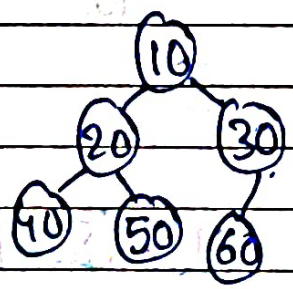
Count the number of nodes at each level & find at which ~~is the maxim~~ level number of node is more.

Hint:- Use level order traversal line by line logic

```
int maxWidth(Node *root) {
 if(root == NULL) { return 0; }
  queue<Node*> q;
   q.push(root);
   int res=0;
   while(q.empty()==false) {
    int count = q.size();
    res = max(res, count);
    for(int i=0; i<count; i++) {
     Node *curr = q.top();
     q.pop();
     if(curr->left != NULL)
          { q.push(curr->left); }
     if(curr->right != NULL)
          { q.push(curr->right); }
    }
   }
   return res;
}
```

Time :- O(n)

Space :- O(w) (w = width)




count=1 | 10 | ....

count=2 | 20 | 30 | ....

count=3 | 40 | 50 | 60 | ....

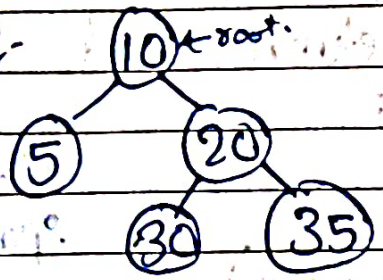
## ⊛ Binary Tree to Doubly linked list :-

Inplace

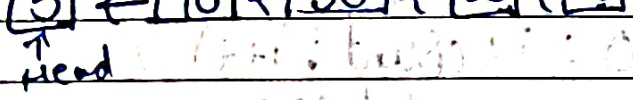
Inorder

Left - Root - Right

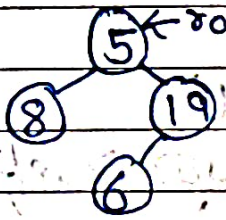
1) I/P :-



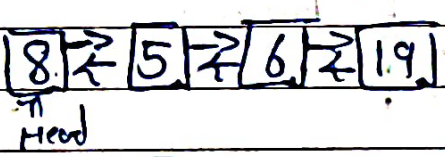

O/P :-



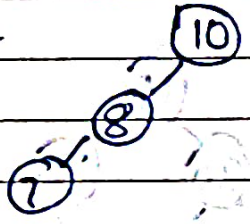

2) I/P :-



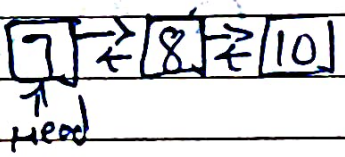

O/P :-




3) I/P :-




O/P :-




⇒ In binary tree, we have right & left. By in Doubly linked list we has next & previous.

So we will use

left as previous & right as next.

⇒

```
Node *prev = NULL;

Node *BTtoDLL(Node *root) {
if(root == NULL) {return root;};
Node *head = BTtoDLL(root->left); } traverse the left subtree
if(prev == NULL) {head = root;};
else
{ root->left = prev;
  prev->Right = root; }
    prev = root;
```

} Process the current node

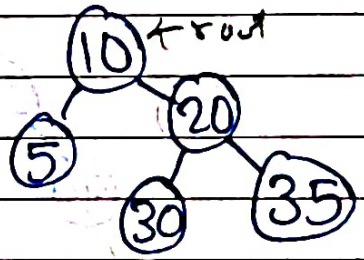
```
BTtoDLL(root->right); } then we traverse the Right subtree.

return head; };
```

Time :- O(n)

Space :- O(h)

↓

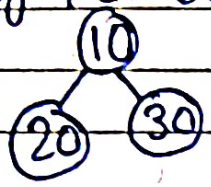
height



## ⊛ Construct Binary Tree from Inorder & Preorder

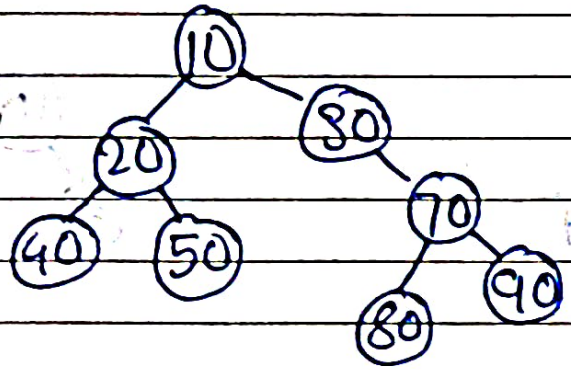
1) I/P:- in[] = {20, 10, 30}
pre[] = {10, 20, 30}

O/P:- Root of the below tree.



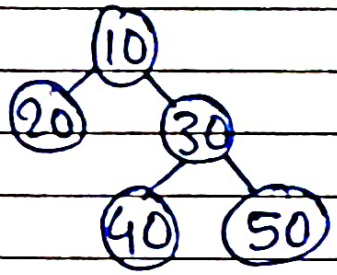

2) I/P:- in[] = {40, 20, 50, 10, 30, 80, 70, 90}
pre[] = {10, 20, 40, 50, 30, 70, 80, 90}

O/P:- Root of the below tree.




3) I/P:- in[] = {20, 10, 40, 30, 50}
pre[] = {10, 20, 30, 40, 50}

O/P:-

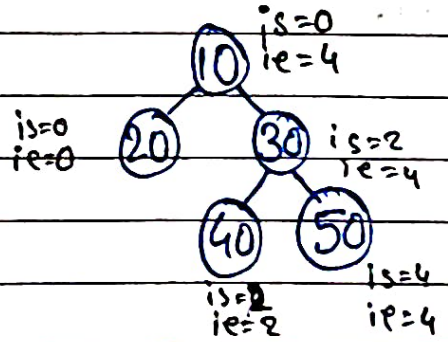


=>

Time :- $O(n^2)$

```
int preIndex=0;

Node *cTree(int in[], int pre[], int is, int ie)
{
  if(is > ie) {return NULL;}
  Node *root = new Node(pre[preIndex++]);

  int inIndex;
  for(int i=is; i<=ie; i++) {
     if(in[i] == root->key) {
        inIndex=i;
        break; }
  }
  root->left = cTree(in, pre, is, inIndex-1);
  root->right = cTree(in, pre, inIndex+1, ie);

  return root;
}
```

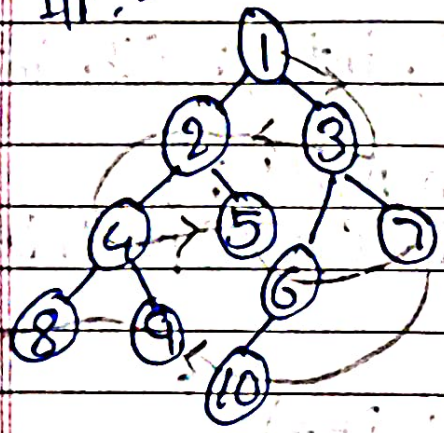
in[] = {20, 10, 40, 30, 50}
pre[] = {10, 20, 30, 40, 50}




We can use Hash function to reduce the time complexity.

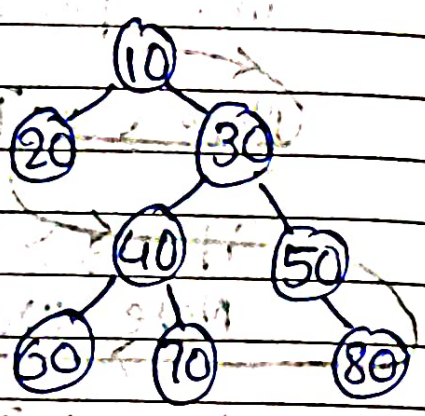
## ⊛ Tree Traversal in Spiral form:-

1) I/p:-




o/p:- 1 3 2 4 5 6 7
10 9 8

2) I/p:-




o/p:- 10 30 20 40 50 80 70
60

⇒ Method 1:-

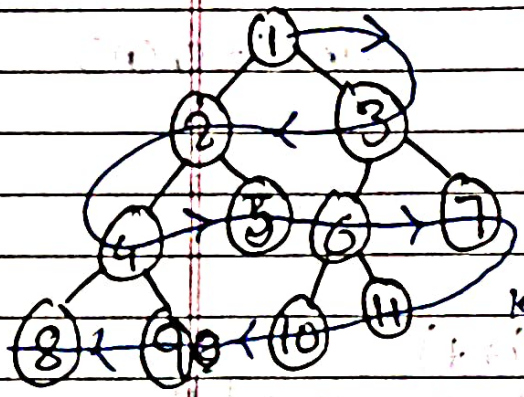
In this method we are using a queue that is useful for normal order traversal we also use stak to reverse the alternate levels.

This solution is based on line by line traversal.

```
void printSpiral(Node *root) {
if (root == NULL) { return; }
queue <Node *> q;
stack <int> S;                    Time :- O(n)
bool reverse = false;
q.push(root);
while (q.empty() == false) {
    int count = q.size();
    for(int i = 0; i < count; i++)
    { Node *curr = q.front(); q.pop();
      if (reverse)
          { S.push(curr->key); }
      else
          {cout << curr->key << " "; }
      if(curr->left != NULL)
          { q.push(curr->left); }
      if (curr->right != NULL)
          { q.push(curr->right);
    }
     if(reverse) {
      while (S.empty() == false) {
          cout << S.top() << " ";
          S.pop(); }
      }
      reverse = !reverse;
      // cout << endl;
    }
}
```

- Method :- 2

Idea is to use two stack.




(1) Push root to the stack S1.
(2) While any of the two stacks is not empty :
  while S1 is not empty
  { a) Take out a node, print it
    b) Push children of the taken out node into S2.
  while S2 is not empty.
  { a) Take out a node print it.
    b) push children of the taken out node into S1 in reverse order.

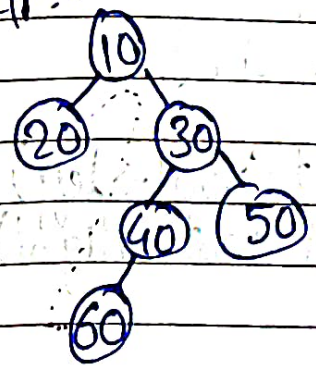
left, right

right, left

⇒
```
void printSpiral (Node *root) {
    if (root == NULL)
        {return;}
    stack <Node *> S1;
    stack <Node *> S2;
    S1.push(root);
while ( ! S1.empty() || ! S2.empty()) {
  while (! S1.empty()) { Node *temp = S1.top();
  S1.pop(); cout << temp->key;
  if (temp->right) { S2.push (temp->right); }
  if (temp->left) { S2.push (temp->left); } }

  while (! S2.empty()) { Node *temp = S2.top();
    S2.pop(); cout << temp->key;
    if (temp->left) { S1.push (temp->left);
    if (temp->Right) { S1.push (temp->right);
  }
  } }
```

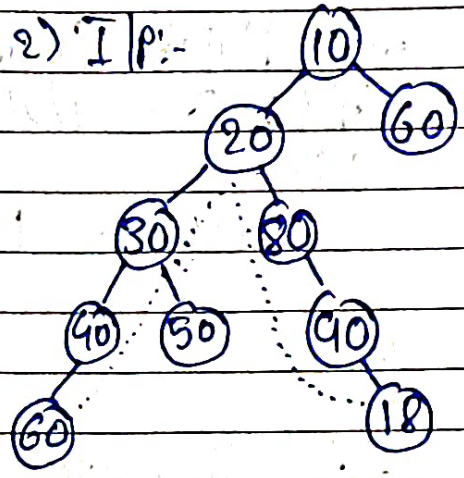
## ⊛ Diameter of Binary tree:-

1) I/P:-




O/P:- 5

2) I/P:-




3) I/P:- NULL
O/P:- 0

4) I/P:- 10
O/P:- 1

The distance b/w two leaves node is know as diameter longest path

Idea:- find the following value for every node & return the maximum:

max (1 + lh + rh)

lh → left Height

rh → Right Height.

⇒ Naive :- $O(n^2)$

```
int Height (Node *root) {
  if (root == NULL) {return 0;}
  else
    {return 1+ max (Height (root->left), root
                    Height (root->Right));
}

int diameter (Node *root)
{ if (root == NULL) {return 0;}
  int d1 = 1+ Height (root->left) +
           Height (root->right);

  int d2 = diameter (root->left);
  int d3 = diameter (root->right);

  return max (d1, max (d2, d3));
}
```

⇒ Efficient Sol^n :- O(n)

Idea :- Precompute height of every node & store it in a map.
No overhead of Map.

This function returns height, but sets the "res" variable to have diameter.

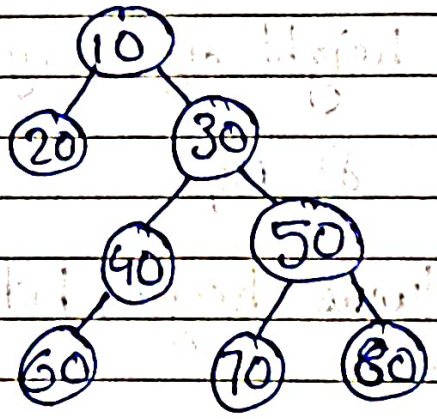
⇒

```
int res = 0;
int height (Node *root) {
 if (root == NULL) { return 0; }
 int lh = height (root->left);
 int rh = height (root->right);
 res = max (res, lh+rh+1);

 return 1 + max(lh, rh);
}
```

Space :- O(h)
↓
height

## ⊛ Lowest Common Ancestor (LCA)

1) I/P:-




n1 = 60
n2 = 70

O/P:- 30

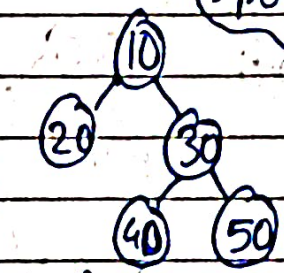
More example in some Tree:-

LCA(20, 80) = 10
LCA(80, 30) = 30
LCA(70, 80) = 50

⇒ Naive Sol^n:- Idea is to create two array & insert every node from root till n1 & n2 then see which are the common nodes & take at lower level in the binary tree.

```
bool findPath (Node *root, vector <*Node*> &p, int n)
{
 if (root == NULL)  {return false;}
  p.push_back(root);
  if (root->key == n) { return true;}
 if ( findPath(root->left, p, n) ||
     findPath(root->Right, p, n))
  { return true;}
   p.pop_back();
   return false; }

Node *LCA (Node *root, int n1, int n2) {
 vector <Node*> path1, path2;
 if (findPath(root, path1, n1) == false ||
     findPath(root, Path2, n2) == false)
       {return NULL;}
 for (int i=0; i< path1.size()-1 && i<path2.size()-1;
                 i++)
  {if (path1[i+1] != path2[i+1])
          {return path[i]; }
  }
   return NULL;
}
```

Time:- $\theta(n)$
Three Traversal
Space:- $O(n)$




n1 = 20, n2 = 50
path1[] = {10, 20}
path2[] = {10, 30, 40}

⇒ Efficient Sol^:-

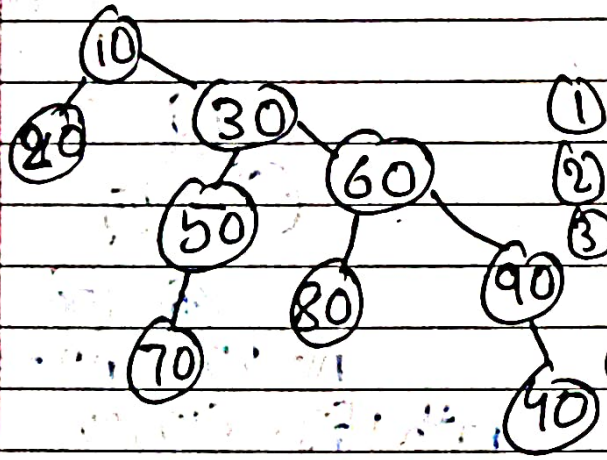
We assumes that n1 & n2 exists in the binary tree.

(1) Requires one Traversal & O(h) extra space for the recursive traversal.

(2) Assumes that both n1 and n2 exist in the tree. Does not give correct result when only one (n1 or n2) exists.

Idea:- We do normal recursive traversal. We have the following cases for every nodes.
(1) If it is same as n1 or n2.
(2) If one of its subtrees contains n1 & other contains n2.
(3) If one of its subtrees contains both n1 & n2.
(4) If none of its subtrees contains any of n1 & n2.

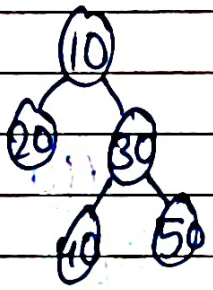
Example of above case




(1) n1 = 10, n2 = 40, curr = 10
(2) n1 = 80, n2 = 40, curr = 60
(3) n1 = 80, n2 = 40, curr = 30 or 10
(4) n1 = 80, n2 = 40, curr = 20

```
Node *lca(Node *root, int n1, int n2)
{
  if(root == NULL) { return NULL; }
  if(root->key == n1 || root->key == n2)
    { return root; }

  Node *lca1 = lca(root->left, n1, n2);
  Node *lca2 = lca(root->right, n1, n2);

  if(lca1 != NULL && lca2 != NULL)
      { return root; }
  if(lca1 != NULL)
      return lca1;
  else
      return lca2;
}
```

Time :- O(n)
One Traversal.
Space :- Less space in worst case.

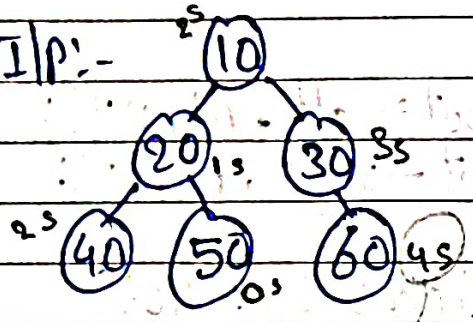



lca(10)
- lca(20)
  - lca(NULL)
  - lca(NULL)
- lca(30)
  - lca(40)
  - lca(50)

## Burn a Binary Tree from a leaf :-

You have a leaf node you have to tell the time to completely burn the binary tree.

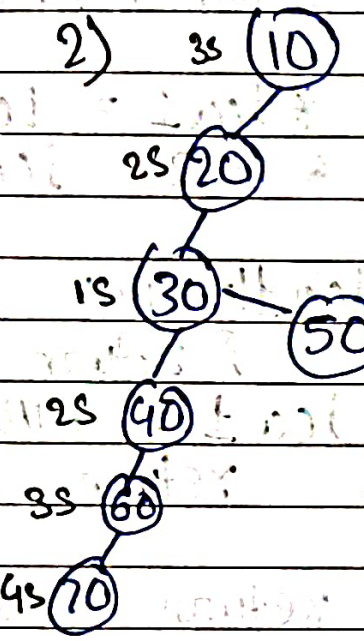
1) I/P :-




leaf = 50

O/P :- 4 sec to burn complete

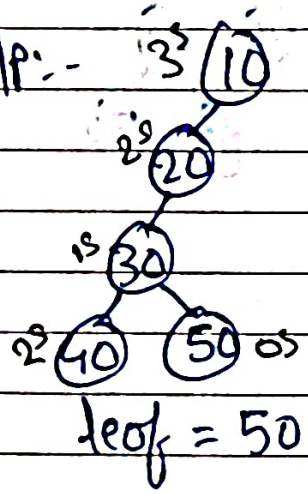
2)




leaf = 50

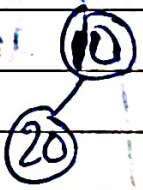
O/P :- 4

3) I/P :-




leaf = 50

4)




leaf = 30

O/P :- -1

We just have to find the farthest node from the given leaf.

Based on diameter of the Binary tree.

⇒ res is shared by all function call.
Initially: dist = -1.

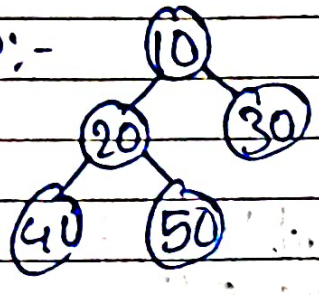
```
int res = 0;
int burnTime(Node *root, int leaf, int &dist) {
  if (root == NULL) { return 0; }
  if (root->data == leaf) { dist = 0; return 1; }
  int ldist = -1, rdist = -1;
  int lh = burnTime(root->left, leaf, ldist);
  int rh = burnTime(root->right, leaf, rdist);
  if (ldist != -1) {
    dist = ldist + 1;
    res = max(res, dist + rh); }
  else if (rdist != -1)
  { dist = rdist + 1;
    res = max(res, dist + lh); }

  return max(lh, rh) + 1; }
```

Tings Things Done

① Return Height
② Set dist if given leaf is a descendant.

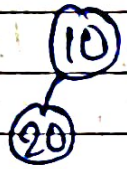
## (*) Count Nodes in a Complete Binary Tree :-

1) I/P :-




O/P :- 5

2) I/P :-

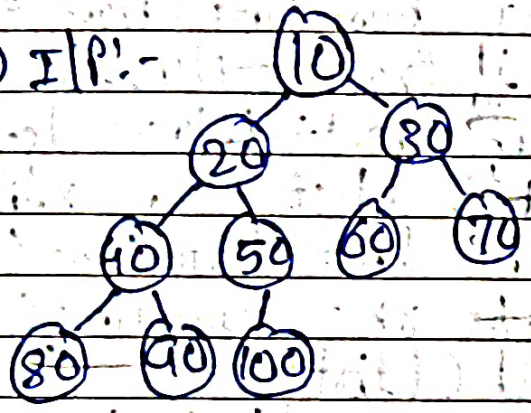



O/P :- 2

3) I/P :- NULL

O/P :- 0

4) I/P :-



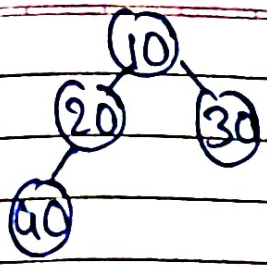

O/P :- 10

Complete Binary tree :-

A complete binary tree is a type of binary tree where all ~~tree~~ levels are filled, except possibly the last level. The last level is filled from left to right.

⇒ Naive :- O(n)

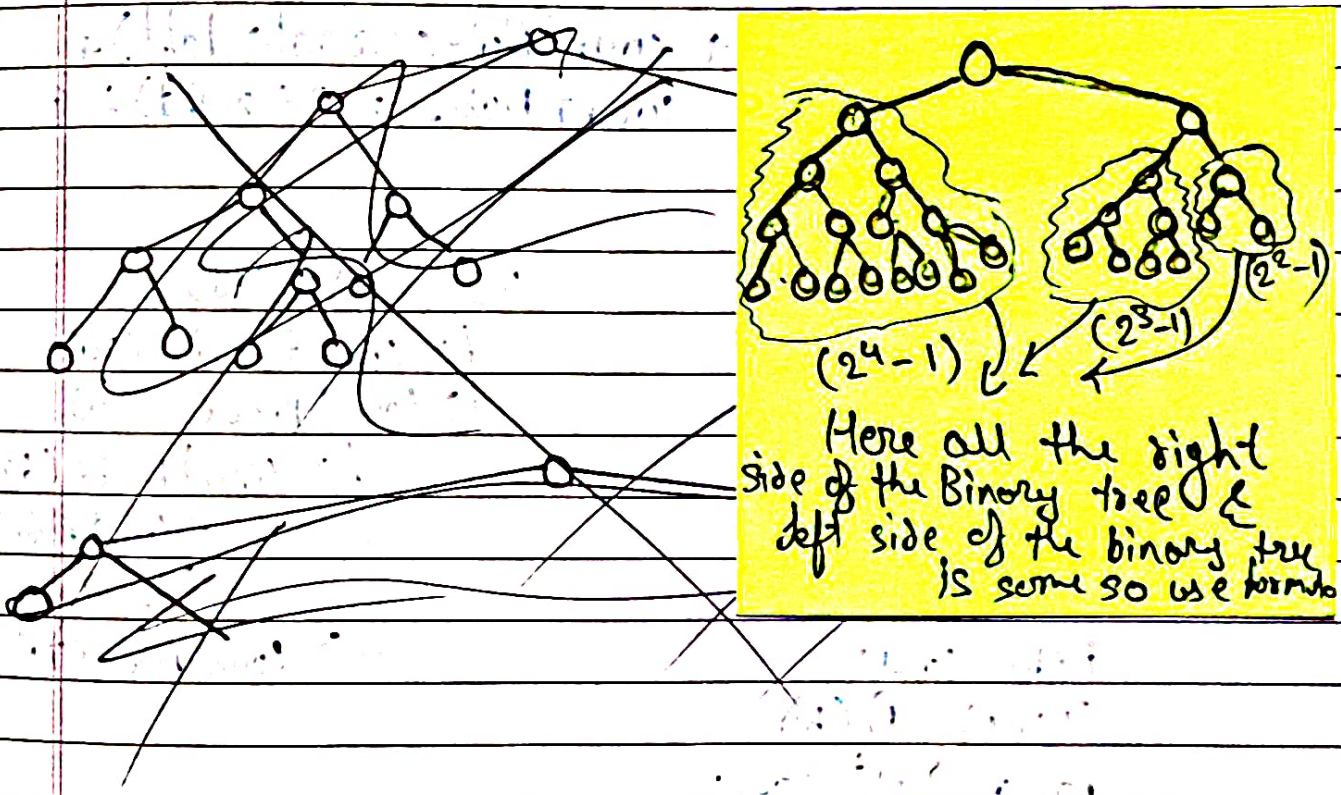
```
int countNode(Node *root)
{ if (root == NULL)
    {return 0;}
  else
    {return (1+countNode(root->left)+
             countNode(root->right);
    }
```




countNode(10)
↳ countNode(20)
↳ countNode(40)
↳ countNode(NULL)
↳ countNode(NULL)
↳ count. Node(NULL)
↳ count Node(30)
↳ count Node(NULL)
↳ count Node(NULL)

⇒ Efficient Soln:- $O(\log n * \log n)$

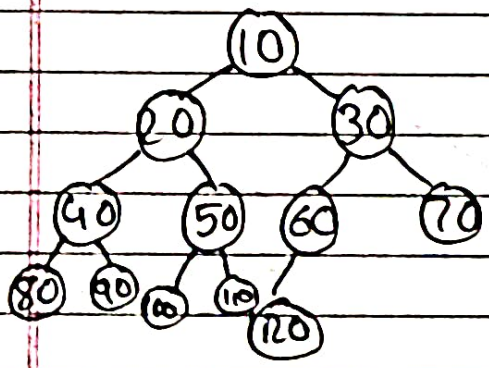
We will use the fact that we know it's a complete Binary tree.




We can check the left side of the tree & right side of the tree if it's equal then we can just use the $2^{\lambda}-1$
$\lambda$ = level.

⇒

```
int count (Node * root) {
int lh=0, rh=0;
Node *curr = root;
while(curr != NULL) {
lh++;
curr = curr -> left }; }
curr = root;
while (curr != NULL)
{ rh++;
  curr = curr -> right; }
if (rh == lh)
  { return pow(2, lh) - 1; }
else
{ return 1+ countNodes(root->left)+
          count Nodes(root->right); }
}
```



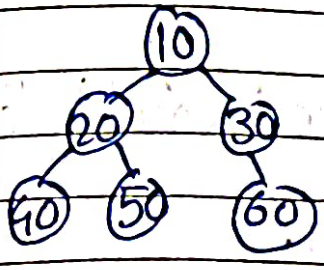

countNode(10)
↳ countNode(20).
↳ countNode(30)
↳ countNode(60)
↳ countNode(120)
↳ countNode(NULL)
↳ countNode(70).

Best case:-
Time:- $\theta(h)$
not a Best case:-
Time:- $T(n) < T(2n/3) + \theta(h)$

↳ After solving

$$\theta(\log_{3/2} n * \log_2 n)$$

## (*) Serialize & Deserialize a Binary tree:-




Serialize ⟶

⟵ String or Array.

Deserialize.

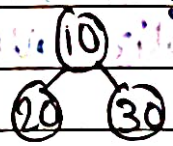
**What we have to do:-**

```
void serialize (Node *root, vector<int> &arr) {...}
Node *deSerialize (vector<int> &arr) {.....}
```

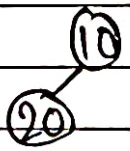
⇒ Pretraversal Approach.

We use -1 for NULL

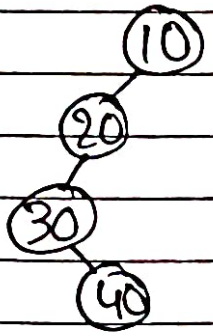
Assumption:- -1 is not present in the tree data.

1) 


serialize ⇒ 10 20 -1 -1 30 -1 -1

2) 

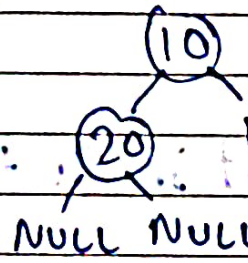

serialize ⇒ 10 20 -1 -1 -1

3) 


serialize ⇒ 10 20 30 -1 40 -1 -1 -1 -1

→ Serialize

```
const int Empty = -1;

Void Serialize (Node *root, vector<int> &arr)
{ if (root == = NULL)
    { arr.push_back (Empty);
        return; }
    arr.push_back (root -> data);
    Serialize (root -> left, arr);
    Serialize (root -> right, arr);
}
```




Serialize (10)
↳ Serialize (20)
↳ Serialize (NULL)
↳ Serialize (NULL)
↳ Serialize (NULL)

arr [] = {10}
arr [] = {10, 20}
arr [] = {10, 20, -1}
arr [] = {10, 20, -1, -1}
arr [] = {10, 20, -1, -1, -1}

Time :- $\theta(n)$
Space :- $\theta(n)$

⇒ (Deserialize)

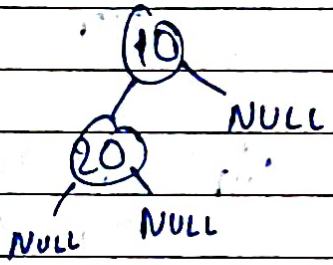
```
const int Empty = -1;
int index = 0;
Node *deSerialize(vector<int> &arr) {
if(index == arr.size()) {return NULL;}
int val = arr[index];
 index++;
if (val == Empty) {return NULL;}

Node *root = new Node(val);
root->left = deSerialize(arr);
root->right = deSerialize(arr);
 return root; }
```

deSerialize() : index=0

→ deSerialize() :- index=1

→ deSerialize() :- index=2

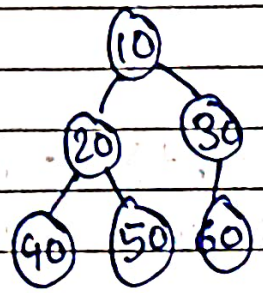
arr[] = {10, 20, -1, -1, -1}




Time :- $\theta(n)$

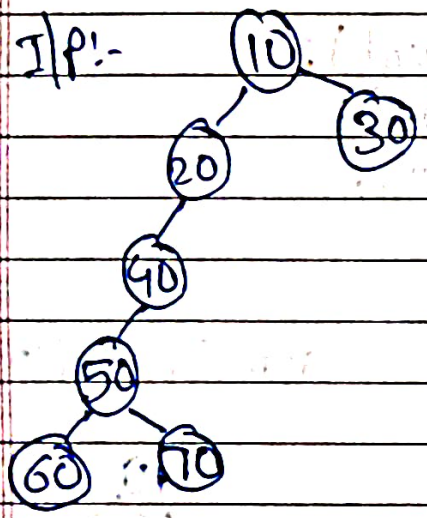
## ⊛ Iterative Inorder Traversal :-

1) I/P :-




O/P :- 40 20 50
10 60 30

2) I/P :-

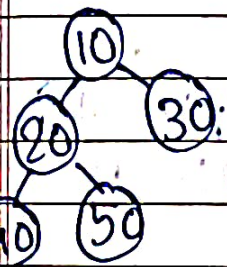



O/P :- 60 50 70 40
20 10 30

3) I/P :- 10
O/P :- 10

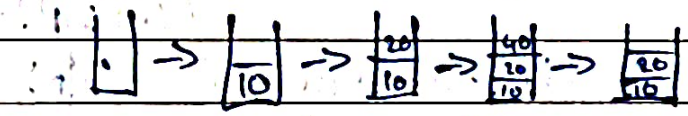
4) I/P :- NULL
O/P :-

```
void iterativeInorder(Node *root) {
  stack<Node*> st;
  Node *curr = root;
  while (curr != NULL) {
  while (curr != NULL || S.empty() == false)
  { while (curr != NULL) {
      S.push(curr)
      curr = curr -> left; }
    curr = S.top(); S.pop();
    cout << (curr -> key) << " "; Space :- O(h)
    curr = curr -> right;
  }
  }
```




Initially: curr = Ref(10);
St = [ ]

Iteration ① of the outer loop.




O/P:- 40

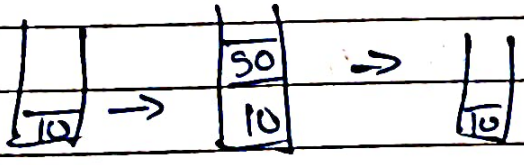
curr = NULL

Iteration ②:

O/P:- 40 20


curr = ref(50)

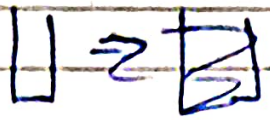
Iteration ③:-




O/P:- 40 20 50, curr = NULL

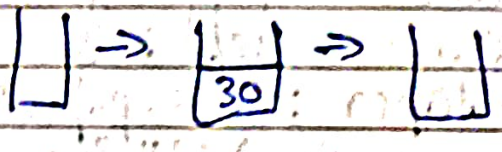
Iteration (4) :-

O/P :- 40 20 50



curr = Ref(30)

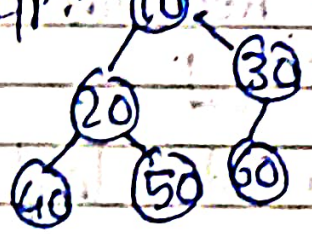
Iteration (5) :-




O/P :- 40 20 50 10 30 , curr = NULL.

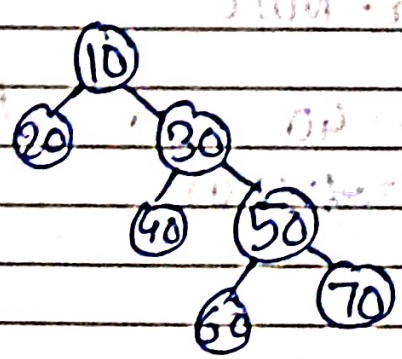
(*) Iterative preorder Traversal :-

1) I/P :-




O/P :- 10 20 40 50 30 60

2) I/P :-




O/P :- 10 20 30 40 50 60 70

3) I/P :- 10

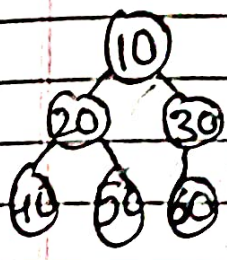
O/P :- 10

4) I/P :- NULL

O/P :-

⇒

```
void iterativePreorder (Node *root) {
if(root==NULL) {return;}
 Stack <Node *> st;
 st.push(root);
 while (st.empty()==false) {
  Node *curr = st.top();
  cout<<(curr->key)<<" ";
  st.pop();
  if(curr->right!=NULL) {st.push(curr->right);}
  if(curr->left!=NULL) {st.push(curr->left);}
 }
}
```




Initially:- st [10]

Iteration 1:- o/p : 10

st [20, 30]

Iteration 2:- o/p:- 10 20

[40, 50, 30]

Iteration 3:- o/p:- 10 20 40

[50, 30]

Iteration 4:-

o/p:- 10 20 40 50

st [30]

Iteration 5:-

o/p:- 10 20 40 50 30

st [60]

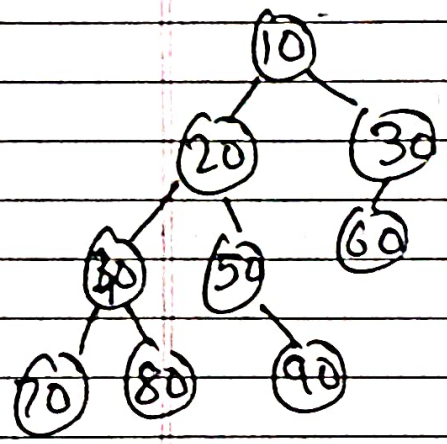
Iteration 6:-

o/p:- 10 20 40 50 30 60.

## ⊛ Iterative preorder Traversal (Space optimized soln)

Time :- O(n)
Space :- O(h)

① Create an empty stack, st
② st.push(root)
③ curr = root.
④ while (st is not empty)
{ while (curr is not NULL)
{ print (curr's key)
if (curr's right is not null)
{st.push(curr's right)}
curr = curr's left.
}
curr = st.pop();




Initially [10]

Iteration :- ① of outer loop.
O/P :- 10 20 40 70
st [80, 50, 30]

Iteration ② :- O/P :- 10 20 40 70 80
st [50, 30]

Iteration (3) O/P:- 10 20 40 70 80 50

St | 90 | 30 |

Iteration (4) :- O/P:- 10 20 40 70 80 50 90.

St | 30 |

⇒

```
void iterativePreorder (Node *root) {
    if (root==NULL) {return;}
     stack <Node *> st;
     Node *curr = root;
     while (curr!=NULL || st.empty() == false)
     { while (curr !=NULL)
        {cout<<(curr->key)<<" ";
         if(curr -> right != NULL)
         {st.push (curr->right);}
           curr = curr->left;
     }
       if(st.empty() ==false)
       { curr = st.top() ;
           st.pop();}
     }
   }
```