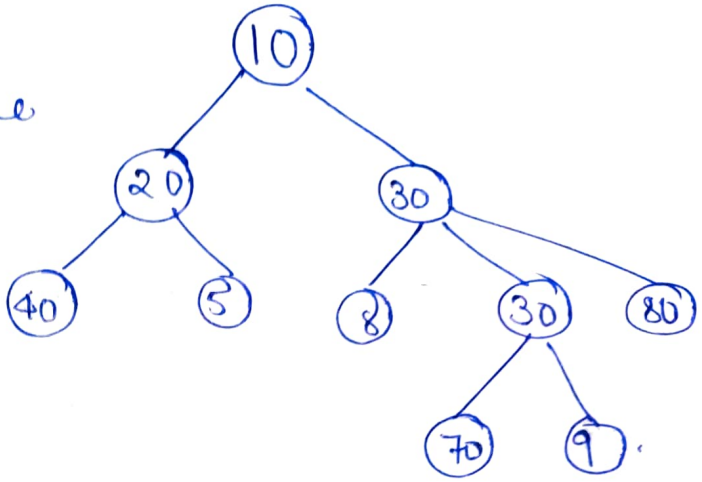
Tree Data Structure

- Heirarchical fashion
- Non linear data structure



Root

First node of the tree – 10

Edge

Link connecting any two nodes of the tree.

Siblings

children nodes of the same parent are called siblings

Descendants :- all the node that lies ~~the~~ in the subtree having the node as root

Ex:- descendants of 10 – everything
descendants of 30 – 8, 30, 80, 70, 9.
descendants of 20 – 40, 5

Ancestors :- A node that is connected to all lower levels of nodes is called ancestor

⇒ 30, 30, 10 are ancestors of 70.

Degree – No of children a node has
- leaf node has 0 degree.
- here only direct children are considered
  degree of 10 → 2.

Applications

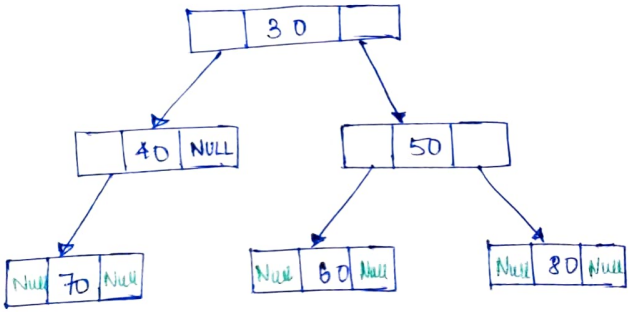
- organisation str. / folder etc. / → hierarchical structure
- Nesting is involved → Tree is used
- Inheritance
- HTML DOM elements.

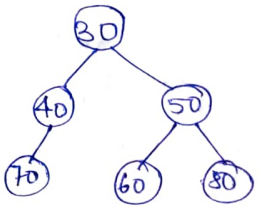
Variations of Trees

- Trie
- Suffix tree
- Binary Indexed tree
- Segment tree

Binary Tree

- degree of a node can be almost 2.
  i.e each node can have (0, 1, 2) nodes.



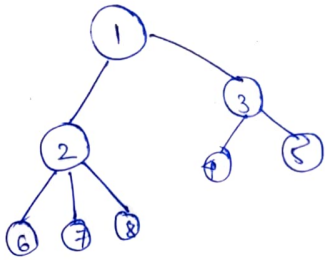


Implementation of Trees

- We have to store many children (address) for every node.
- We can't use array → size is not fixed
  we can't use LL → access time is O(n).

Best option is vector

↳ Store the address of all the child nodes in the vector of node*.



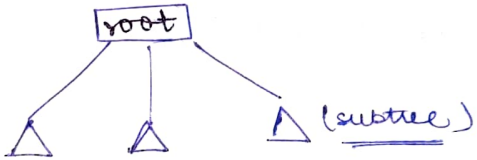
root → 1

children of root → 2, 3

children of 2 → 6, 7, 8

children of 3 → 4, 5

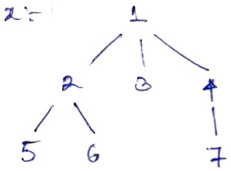
We have to imagine tree as



Takeinput

① Recursive (Depth first input)

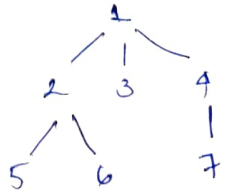
Ex:-



user will ~~type~~ input → 1 2 5 6 3 4 7

(Difficult for user to provide input)

② Level wise (Breadth)



1 2 3 4 5 6 7

(Easy)

```
class TreeNode {
    public :
    int data;
    vector<treeNode*>
        child;
};
```

making root —

```
treeNode * root =
    new treeNode (1);
```

connecting children

```
treeNode * c1 =
    new treeNode (2);
root → child.pushback
    (c1)
```

Taking input Recursive

Basic approach is to call recursion when user enter the no of child

Ex: function takeinput () {
cout << Enter data

```
TreeNode * takeInput (){
    cout<<"Enter data";
    cin >> n;
    cout<<" Enter no of children for n";
    cin >> child;
    for(int i=0; i< child; i++){
        TreeNode * c = takeinput();
        root -> children.push_back (c);
    }.
}
```

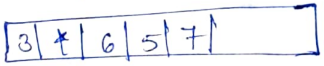
TreeNode *root = new TreeNode(n);

takeinput levelwise

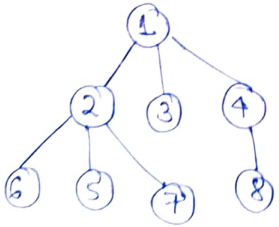
Approach is to use dequeue, we will insert the root first and while popping an element we will ask the no of children it have and push them inside the dequeue.

popping 1 & inserting 2,3,4

popping 2 & inserting 6,5,7



popping 3 & inserting nothing



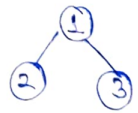

```
TreeNode * takeInput (){
    cout<<"Enter data";
    cin>> data;
    TreeNode * root = new TreeNode (data);
    queue< TreeNode *> q;
    q.push (root);
    while (!q.empty()){
        cout<<"number of children of" << q.front()->dat
        cin >> children;
        while (children--){
            cout<<"Enter the data";
            cin >> data;
            TreeNode *child = new TreeNode (data)
            TreeNode c (data); // Allocate dynamically
            q.front()-> children.push_back (c);
        }
        q.pop();
    }.
```

Traversal

(a) depth first traversal
(b) Breadth first traversal (level order)

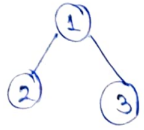
Depth First Traversal

- inorder



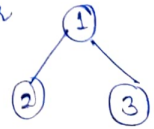
2 1 3

- preorder



1 2 3

- postorder



2 3 1.

Inorder

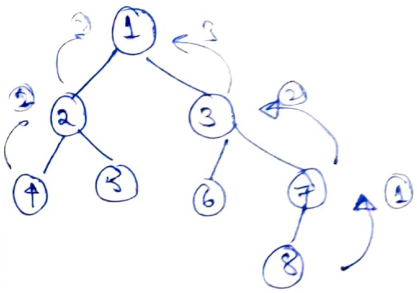
```
{ if (root == NULL)
      return
  inorder (root->left)
  cout << root->data;
  inorder (root->right);
}
```

Preorder

```
{ if (root = NULL)
      return;
  ~~preorder (root~~
  cout << root->data;
  preorder (root->left)
  preorder (root->right);
```

Height of Binary Tree

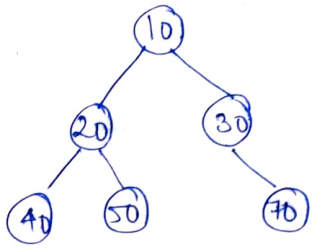
Height = max (left sub tree , right sub tree) + 1 .




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

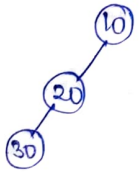
Print Nodes at Distance 'k'

I/P k = 2



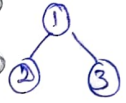
O/P → 40, 50, 70.

I/P → k = 1



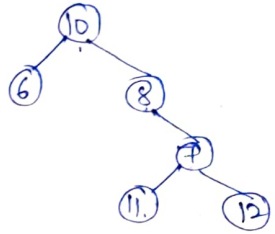
O/P → 20.

k = 0



O/P → 1

I/P → k = 3

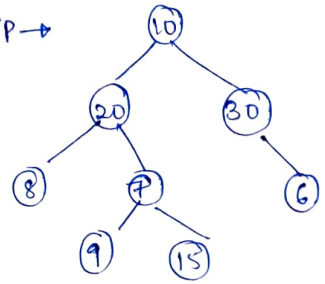


O/P → 11, 12

```
printKm (TreeNode<int> *root){, int k, int count=0){
  if (root == NULL) {
      return;
  }
  if (count == k) {
      cout << root->data;
      printKm (root->left , k, 0);
      printKm (root->right, k, 0);
  }
  else {
      count++;
      printKm (root->left, k, count);
      printKm (root->~~left~~ right , k, count);
  }
}
```

Level Order Traversal

I/P →



O/P → 10 20 30 8 7 6
9 15

Naive Approach

- calculate height of binary tree (h)
- print nodes on all levels from 0 to h

```
print(0)
print(1)
:
print(h)
```

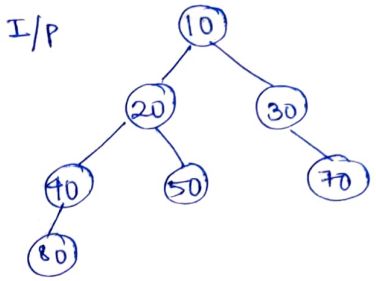
Effective Approach

```
void levelTraversal(TreeNode<int> * root){
    queue<treeNode<int>*>q;
    q.push(root);
    while(!q.empty()){
        if(q.front()->left){
            q.push(q.front()->left);
        }
        if(q.front()->right){
            q.push(q.front()->right);
        }
        cout<<q.front()->data<<" ";
        q.pop();
    }
}
```

Time comp
$\Theta(n)$
Aux. space
$\rightarrow \Theta(w)$
↳ width

Level Order Traversal line by line

I/P



O/P →
10
20 30
40 50 70
80

* idea is to insert a marker in the end of every level

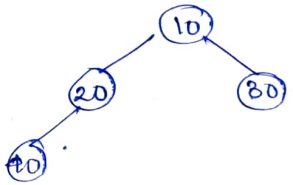
As the queue is of TreeNode * type, the best marker would by NULL.

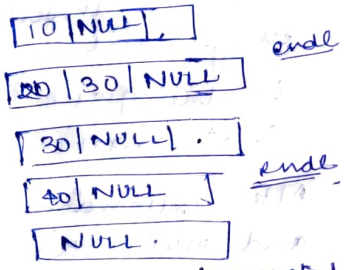
After pushing root, we push a NULL marker and while traversing the queue whenever we see a null marker, we are sure that a particular level is processed and the elements in the queue are the part of next level only. Hence we push NULL at end of queue. to denote the end of upcoming level.

```
void printLevelOrder(TreeNode * root){
    if(root==NULL) return;
    queue<TreeNode*>q;
    q.push(root); q.push(NULL);
    while(q.size()>1){
        TreeNode * curr = q.front();
        q.pop();
        if(curr==NULL){
            cout<<"\n";
            q.push(NULL);
            continue;
        }
        cout<<curr->key<<" ";
```

```
        if (curr->left) q.push(curr->left);
        if (curr->right) q.push(curr->right);
    }
}
```

- we will terminate loop when q.size == 1 because at that point there will be only NULL pointers left.





Second Approach

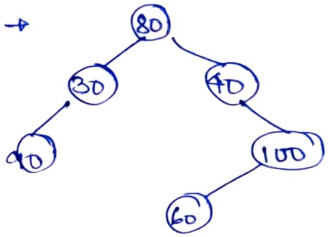
- we basically have two loops, inner loop prints the level and outer loop puts '\n' after every level.

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

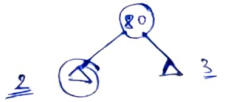
time complexity → $\theta(n)$.
Aux space → $\theta(w)$

Size of Binary Tree

I/P →



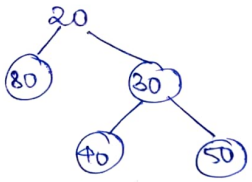
O/P → 6



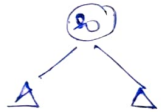

```
int getSize(TreeNode *root){
    if (root == NULL)
        return 0;
    return 1 + getSize(root->left) + getSize(root->right);
}
```

Time complexity → O(n).
Aux space → O(h) → height of binary tree

Maximum in Binary Tree



O/P → 80

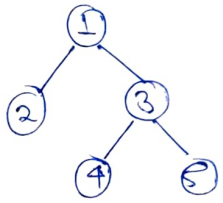



```
int getMax(TreeNode * root){
    if (root == NULL) return INT_MIN;
    else
        return max(root->data, max(
            getMax(root->left), getMax(root->right)
```

Print Left View of Binary Tree

```
void leftView (TreeNode<int> * root) {
    queue<TreeNode<int>*> q;
    if (root == NULL) return emptyVector;
    q.push(root);
    q.push(NULL);
    cout << q.front()->data << "\n";
    while (q.size() > 1) {
        TreeNode * curr = q.front();
        if (curr == NULL) {
            q.pop();
            cout << q.front()->data;
            q.push(NULL);
            continue;
        }
        if (curr->left) q.push(curr->left);
        if (curr->right) q.push(curr->right);
        q.pop();
    }
}
```

Right View of Tree

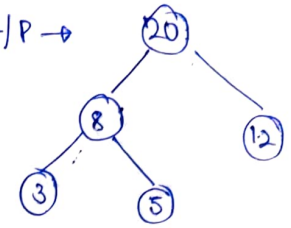


o/p → 1 3 5

* similar approach like the left view just have to maintain a prev pointer pointing to the Node before the current node

Leetcode – 199

Children Sum Property



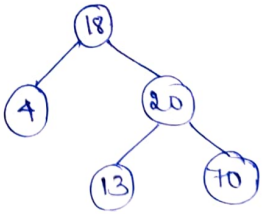
O/P → Yes

(sum of children of particular node is equal to root)

```
bool childrenSum (TreeNode<int> * root) {
    if (root == NULL)
        return true;
    if (!root->left && !root->right)
        return true;
    int sum = (root->left ? root->left->data:0)
            + (root->right ? root->right->data:0);
    return (root->data == sum) &&
           childrenSum(root->left) &&
           childrenSum(root->right);
}
```

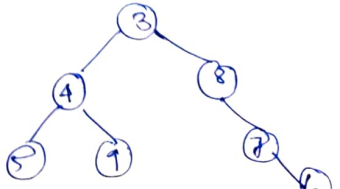
time complexity → O(n)
Aux space → O(h)

Height Balanced Binary tree



→ Yes.

difference between the heights of left and right subtree is almost one.



O/P → NO.

Naive solution

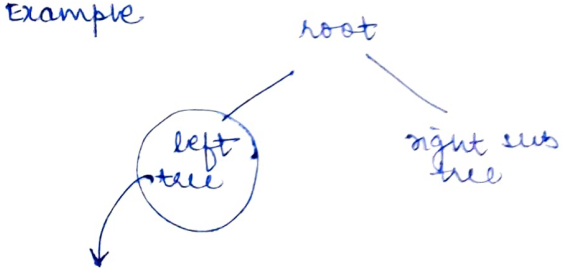
```
bool isBalanced (Node * root) {
    if (root == NULL) return true;
    int lh = height (root -> left);
    int rh = height (root -> right);
    return (abs (lh - rh) <= 1 &&
            isBalanced (root -> left) &&
            isBalanced (root -> right));
}
```

time complexity → $O(n^2)$

Effective approach

- we need of serve-two purposes by one function to reduce the complexity to $O(n)$
  1. difference in heights
  2. whether balanced or not

Example



function (left subtree)

↓

returns
1. height of left subtree
2. whether left subtree is balanced or not.

- Return -1 when tree is not balanced
- otherwise return height of tree

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

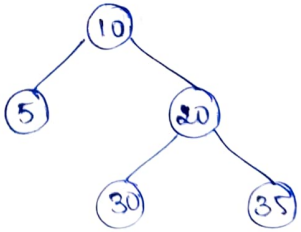
Maximum width of Binary Tree

using the concept of level order traversal line by line.

- maintain a count variable which gives the width of current level i.e between two NULL markers.

Convert a Binary tree to DLL

- inorder traversal

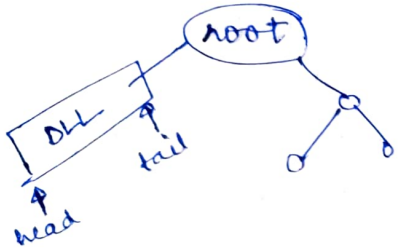


order followed is inorder traversal

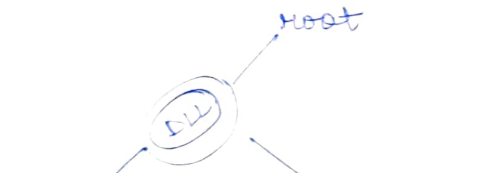
- use left → prev
  right → next

inOrder → function (root → left)
print (root → data)
function (root → right)

I will assume at a particular node we have made DLL till left of it so the scenario will be



case at a particular node we have,



head = NULL
↓
head = root
tail = root

head ≠ NULL
connections of root to tail
root → right = tail
tail → left = root
tail = root

- In this we have to pass pointers by reference because we're changing the node where may the pointers are pointing.

To reflect those changes outside the function we can either use

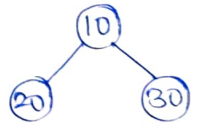
Node * & root
or
Node ** root

```
void BTtoDLL( Node *root, Node * &head = NULL
              Node * &tail = NULL);
{
    if (root == NULL) return NULL;
    BTtoDLL ( root->left , head , tail);
    if (!head) {
        head = root;
    } else {
        ~~tail->left~~
        root -> left = tail
        tail -> right = root;
    }
    tail = root;
    BTtoDLL ( root->right , head , tail);
}
```

Construct a Binary Tree when Inorder and Preorder Traversal are given -

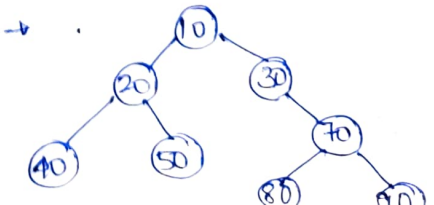
I/P : in[] = {20, 10, 30}
pre[] = {10, 20, 30}

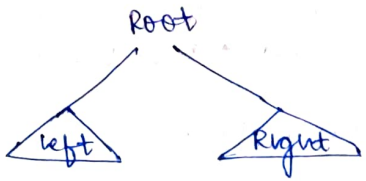
O/P : Root →



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

O/P →





inorder → left root right
pre → root left right

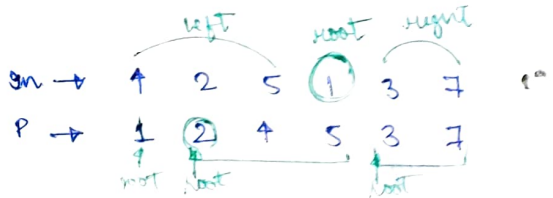
- we need inorder traversal to form a tree, we can't form a tree if we are given only preorder and postorder traversals
  - Because inorder uniquely divides left and right subtree.

Ex:-

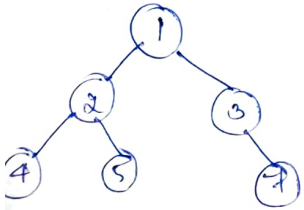
pre	AB
post	BA

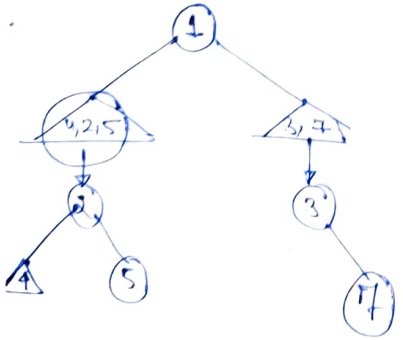
AB
BA

If we are given this info we can't construct a unique Tree.



root will be first element of preorder traversal.





we will move our pointer in preorder traversal, maintain a window in inorder traversal and search for root.

Ex:

In – 9 3 15 20 7
Pre – 3 9 20 15 7
↑

at i=0 search 3 in inorder

left right
9 (3) 15 20 7

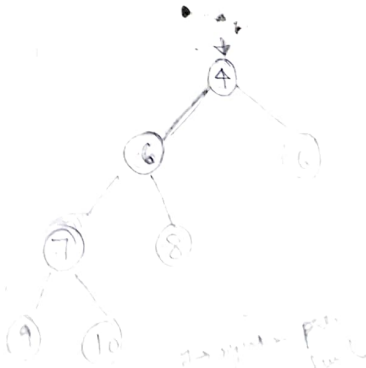
(connect 3 with fun(root, left window))

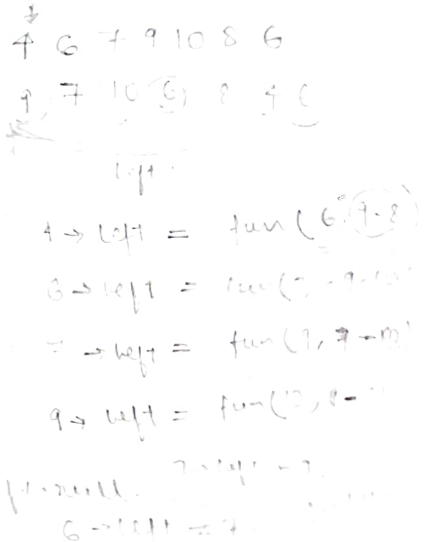
at i=1 search 9 in inorder.

right
(9) (3) 15 20 7
↑
only node (i.e. no child)

at i=2 search 20

left right
(3) (9) 15 (20) 7





```
int preIndex = 0;
Node *buildTree (int in[], int pre[], int is, int ie)
    if(is > ie) return NULL;
    Node *root = new Node(pre[preIndex]);
    preIndex++;
    for(int i= is; i<= ie; i++){
        if(in[i] == root->key){
            inIndex = i;
            break;
        }
    }
    root->left = buildTree(in, pre, is, inIndex-1);
    root->right = buildTree(in, pre, inIndex+1, ie);
    return root;
}
```

Build Tree from Postorder and Inorder

P → { 6, 2, 4, 5, 3, 1 } ~~NRL~~ LRN

I → { 6, 2, 1, 4, 3, 5 }

we'll follow the same approach but

① start from last (Because in post order root is last (left right root)

i.e ~~arr~~[post[n-1]] = root of tree

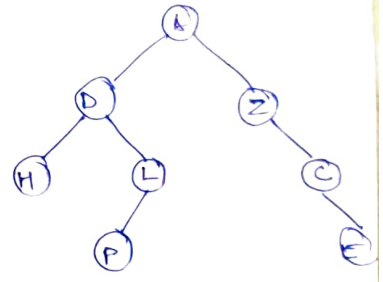
② we'll call ~~tree->rg~~ root->right first because as we are traversing from last the order will be NRL

node
node->right

Build Tree from Inorder and Level Order Traversal

Inorder – H D P L A Z C E

level order – A D Z H L C P E



Identifying root

① First element of LOT is root (segregate left & right part of BT using Inorder)

H D P L (A) ~~A~~ Z C E

Among this the root is whichever comes first in LOT

(D)

H (D) P L

H is root

L is root

P is left of L

Z is root

Z C E

left of Z is NULL

C is root

E is right of C

~~We will~~

we have to find the element inorder traversal that have minimum index in level order traversal

Algorithm

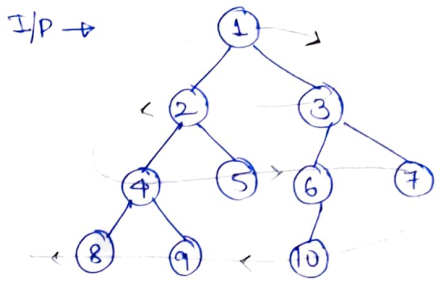
```
TreeNode * construct (start, end) {
    if (start > end) return NULL;
    inIndex = Find minimum index of node from inStart to inEnd which has minimum level order index.
    node = new TreeNode(in[inIndex]);
    node->left = construct(start, inIndex-1);
    node->right = construct(inIndex+1, end).
    return node;
}
```

- The complexity of search ~~search~~ seems to be $O(n^2)$ which makes whole algorithm $O(n^3)$.

We can use map to store level order traversal elements and their indices
↳ Reduce the search complexity → $O(n^2)$

```
TreeNode * construct_tree(vector<int>& inorder,
                          unordered_map<int,int>& m,
                          int start, int end){

    if (start > end)
        return NULL;

    int inIndex = start;
    for (j = start+1; j <= end; j++){
        if (m[inorder[j]] < m[inorder[inIndex]])
            inIndex = j;
    }

    TreeNode * root = new TreeNode(inorder[inIndex]);

    root->left = construct(inorder, m, start, inIndex-1);

    root->right = construct(inorder, m, inIndex+1, end);

    return root;
}
```

Tree Traversal Spiral Form



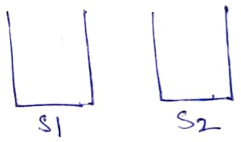
O/P → 1 3 2 4 5 6 7 10 9 8

Method-1

Using level order traversal along with stacks. (We'll store alternate levels inside stack and then reverse them).

Method-2

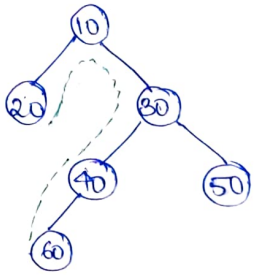
We will use two stacks here to store alternate levels.



① Push root to S1
② While any of stack is not empty
If S1 is not empty
(a) Take out a node, print it
(b) push its children in S2
If S2 is not empty
(a) Take out the node & print
(b) push children of the taken out node in reverse order.

Diameter of Binary Tree

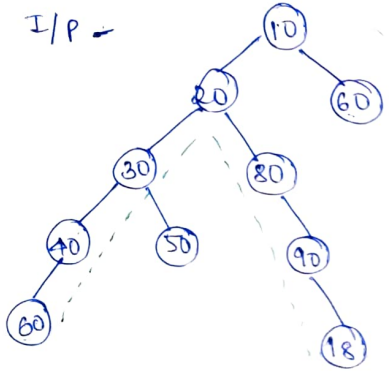
I/P



diameter → longest path b/w two leaf nodes

= 5

I/P →



O/P → 7

For any node we need to calc.

1 + lh + rh

left height, right height

return max from all nodes

Naive Approach

```
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));
}
```

$O(n^2)$.

Effective height of every Node

Approach

- Precompute height of every node.
- and store all of them in map.
  - ↳ Reduced to O(n) but overhead of map.

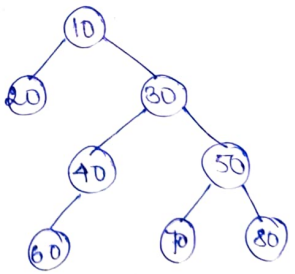
Effective Approach

We can modify the height function to compute diameter for every node.

```
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, 1 + lh + rh);   <- computing diameter
    return 1 + max (lh, rh);
}
```

Lowest Common Ancestor (LCA) AA

I/P



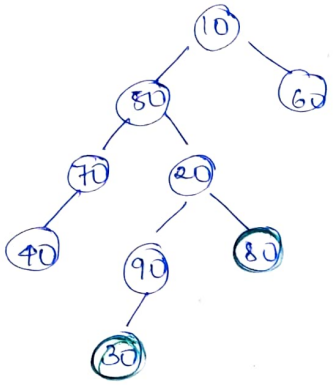
LCA (60, 80) → 10
LCA (80, 30) → 30
LCA (70, 80) → 50

Ancestors of 40 → 30, 10
Ancestor of 70 → 70, 50, 30, 10
↓
lowest among them → 30

Given is also helpful in finding the minimum distance between two nodes
→ we can find the LCA of two nodes and then similarly distance of both from LCA

Naive solution

Build two path arrays., i.e from root to the node



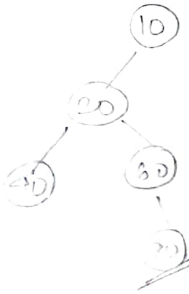
path 1 → {10, 50, 20, 90, 30}
path 2 → {10, 50, 20, 80}

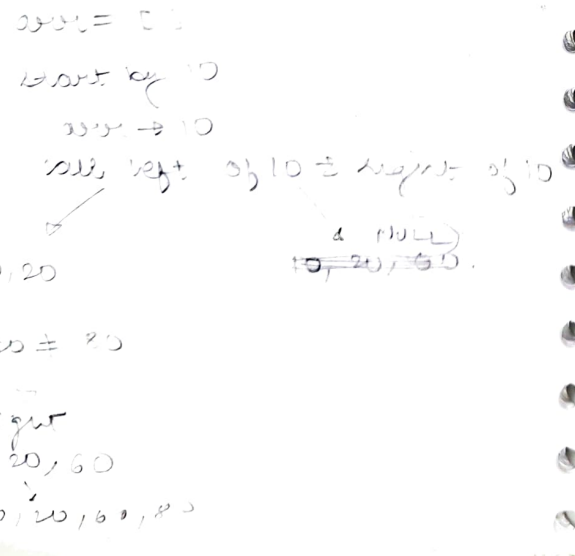
The common among them are
10, 50, 20
↑
lowest

O/P → 20

- complex part is to build the path arrays.
  - ↳ we will add a particular node
    if (node == targetnode) return true;
    else call left
    right

    If both left and right return false. remove the node.






```
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 → left, p, n)) return true;
    p.pop_back();
    return false;
}
```

A particular root is included only if
① it is target node
② it's children contain target node

```
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 path1[i];
    }
    return NULL;
}
```

O(n) but requires 3 traversals

Effective Approach

- assumes that both node is present in tree.
- If either of both node is NOT present it gives the incorrect result

Approach :

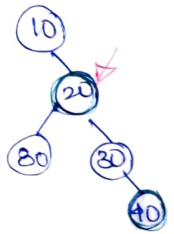
We will do normal tree traversal, possible cases are :

Let curr is the node which is in process

(a) curr == n1 or curr == n2
(return curr)

In this case curr is going to be the LCA as c2 is in lower level.

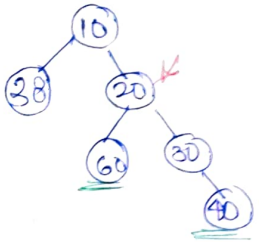
Ex



curr = 20 == n1
Return 20

(b) If of subtree contain n1 and other contain n2

→ curr becomes LCA



for 10 → both are on left
for 30 → none is present
for 20 → one is on left other is 40

Hence 20 is LCA

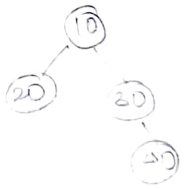
(c) one of subtree contain both n1 and n2
(we return whatever that subtree returns)

(d) If none of subtree contain n1 or n2
Return NULL

```
Node * LCA (Node * root, int n1, int n2) {
    if (root == NULL) return NULL;
case-1 { if (root -> key == n1 || root -> key == n2)
            return root;
    }
    Node * lca1 = LCA (root -> left, n1, n2);
    Node * lca2 = LCA (root -> right, n1, n2)
case-2 { if (lca1 != NULL && lca2 != NULL) {
            return root;
    }
case-3 { if (lca1 != NULL)
            return lca1;
        else
            return lca2;
    }
}
```

O(n) one traversal of tree

This code returns the node present in tree even if other is not present

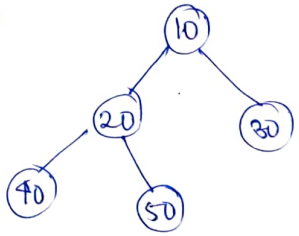


n1 = 10
n2 = 30
LCA → 30 ← returns

(Hence it works only when both are present)

Count Nodes in complete Binary Tree

I/P



O/P → 5

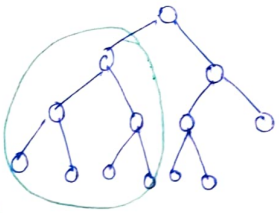
Naive Solution

using simple dfs traversal, we can compute the number of nodes

return 1 + fun(root→left) + fun(root→right)

Effective Approach

we can take advantage of the fact that the given tree is complete



If a tree is complete then the number of nodes are = $2^d - 1$

where d ⇒ depth

- If the subtree is not complete we switch to naive method i.e. 1 + (root→left) + (root→right)

* To check whether a tree is complete or not we can use root→left→left ---- = length of left most branch

root→right→right ---- = length of right most branch

If both are equal we can directly use the formula

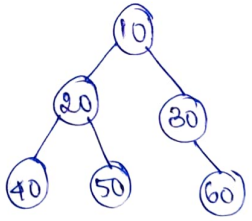
```
int countNode(Node * root) {
    int lh = 0, rh = 0;
    Node * curr = root;
    while (curr != NULL) {
        lh++;
        curr = curr→left;
    }
    while (curr != NULL) {
        rh++;
        curr = curr→right;
    }
    if (lh == rh) {
        return pow(2, lh) - 1;
    }
    return 1 + countNode(root→left)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

O(log n * log n)

→ directly returning the height

→ naive method

Serialization and Deserialization of Binary Tree



serialize →

← deserialize

String or Array

Application

- To send a tree across a network
- to test cases in programs

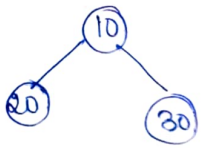
ive Solution

We can store inorder and (pre/post/level order traversal [either of them]).

→ It requires two traversal of binary tree

Effective approach

(1)

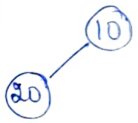


~~10 20 30~~

We use -1 for NULL assuming -1 is not present

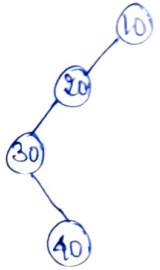
→ 10 20 -1 -1 30 -1 -1

(2)



10 20 -1 -1 -1

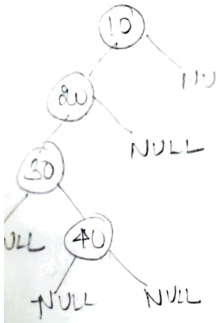
(3)



10 20 30 -1 40 -1 -1 -1 -1

Deserialize

10 20 30 -1 40 -1 -1 -1 -1



Serialization

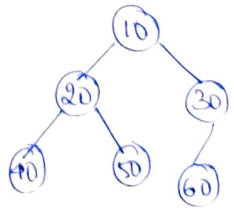
```
void serialize (Node * root, vector<int> &arr){
    if (root == NULL) {
        arr.push_back (-1);
        return;
    }
    arr.push_back (root → key);
    serialize (root → left, arr);
    serialize (root → right, arr);
}
```

Deserialization

```
Node * deserilize (vector<int> & arr, int &index){
    if (index == arr.size()) return NULL;
    if (arr[index] == -1) return NULL;
    Node * root = new Node (arr[index++]);
    root → left = deserialize (arr, index);
    root → right = deserialize (arr, index);
    return root;
}
```

Iterative inorder Traversal

I/P



→ ~~40~~ 40 20 50 10 60 30