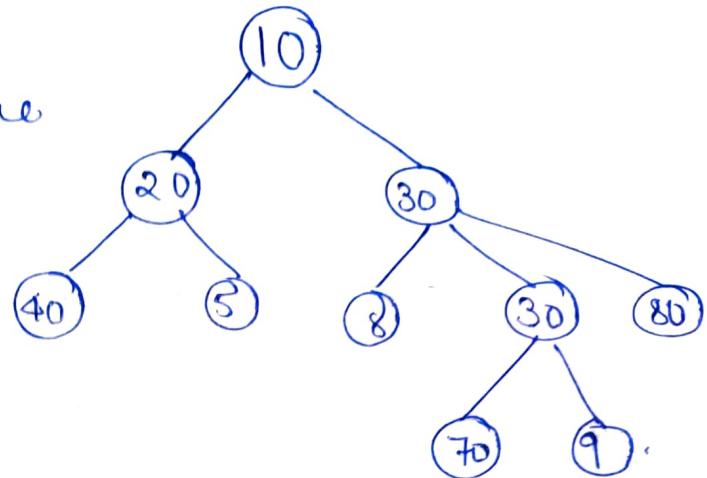


Tree Data Structure

- Hierarchical 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 nodes that lies in the subtree having the node as root

Ex:- descendants of 10 - everything

descendants of 30 - 8, 30, 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

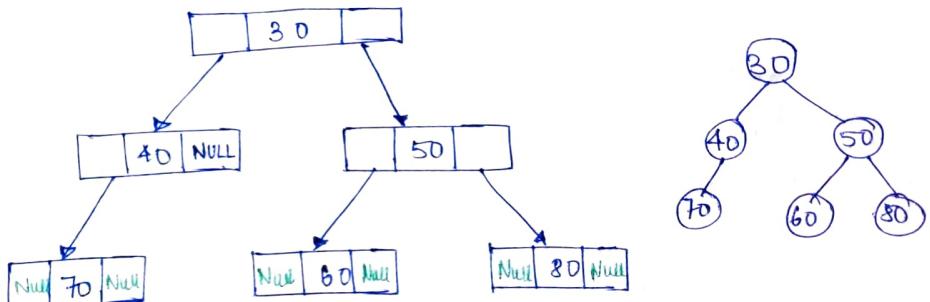
- organisation str. / folder str. / → hierarchical structures
- Nesting is involved → Tree is used
- Inheritance
- HTML DOM elements.

Variations of Trees

- ↳ Trie
- ↳ suffix tree
- ↳ Binary Index tree
- ↳ segment tree

Binary Tree

- degree of a node can be almost 2
i.e. each node can have 10, 11, 12 nodes.

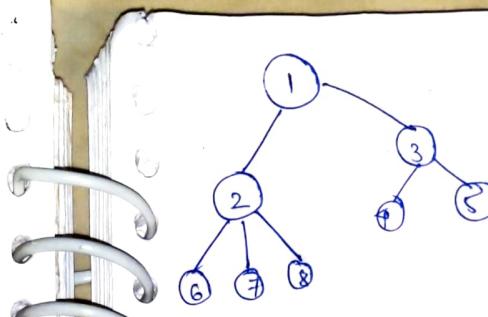


Implementation of Trees

- 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 nodes.



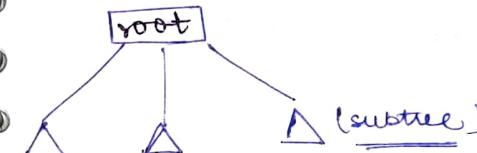
root → 1

children of root → 2, 3

children of 2 → 6, 7, 8

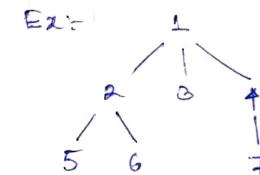
children of 3 → 4, 5

We have to imagine tree as



Take Input

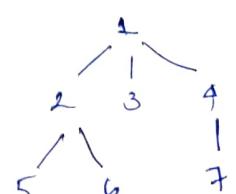
- ① Recursive (Depth first input)



user will input
 $\rightarrow 1 \ 2 \ 5 \ 6 \ 3 \ 4 \ 7$

(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;
}
```

3.

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 enters the no of child

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);
    }
}
```

takeInput levelwise

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



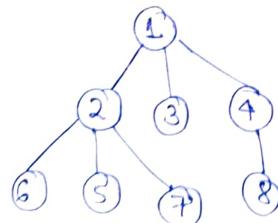
popping 1
 inserting 2,3,4



popping 2 & inserting 6,5,7



popping 3 & inserting nothing



TreeNode * takeInput()

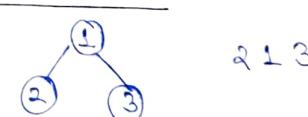
```
TreeNode * takeInput() {
    cout << "Enter data";
    cin >> data;
    TreeNode * root = new TreeNode(data);
    queue<TreeNode *> q;
    q.push(root);
    while (!q.empty()) {
        cout << "number of children of " << q.front()->data
        cin >> children;
        while (children--) {
    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

Traversal

- depth first traversal
- Breadth first traversal (level order)

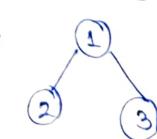
Depth First Traversal

→ inorder



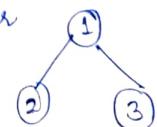
2 1 3

→ preorder



1 2 3

→ postorder



2 3 1

```

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

```

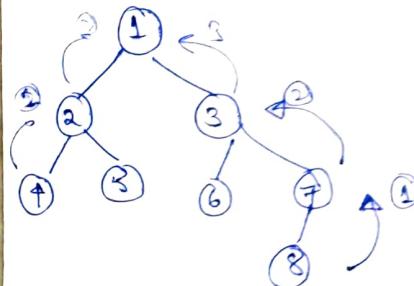
```

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

```

Height of Binary Tree

$$\text{height} = \max_{\substack{\text{left sub tree} \\ \text{right sub tree}}} + 1.$$



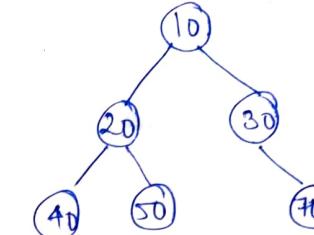
```

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

```

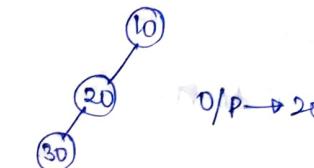
Print Nodes at Distance 'k'

I/P K = 2



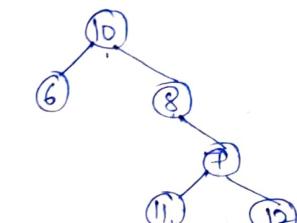
O/P → 40, 50, 70

I/P → K = 1



O/P → 20

I/P → K = 3



O/P → 11, 12

K = 0



O/P → 1

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);

}

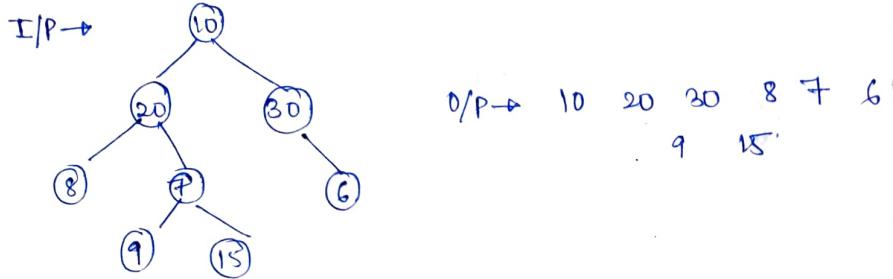
else {

count++;

printkm (root->left, k, count);

printkm (root->right, k, count);

Level Order Traversal



Naive Approach

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

```

print(0)
print(1)
⋮
print(n)
  
```

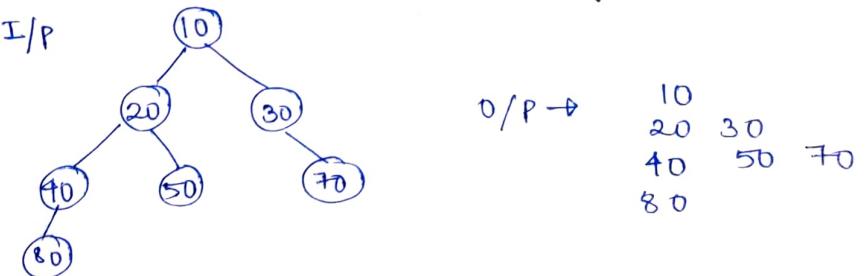
Effective Approach

```

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
 $\Theta(w)$
 ↴ width

Level Order Traversal line by line



- idea is to insert a marker in the end of every level
- As the queue is of `TreeNode*` type, the best marker would be `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.

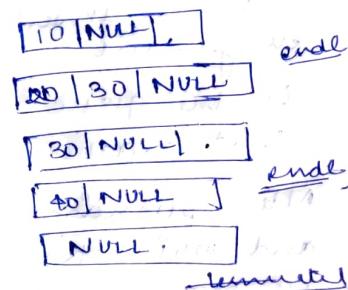
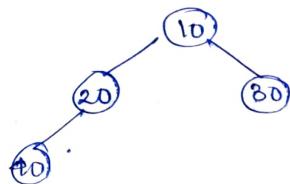
```

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

if ($\text{curr} \rightarrow \text{left}$) q.push ($\text{curr} \rightarrow \text{left}$);
if ($\text{curr} \rightarrow \text{right}$) q.push ($\text{curr} \rightarrow \text{right}$);

3

- ④ we will terminate loop when q.size == 1 because at that point there will be only NULL pointers left.



Second Approach

- ① we basically have two loops, inner loop prints the level and outer loop puts 'in' 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);
        }
    }
}

```

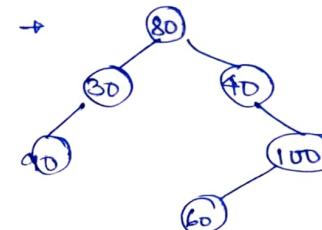
4

```
cout << "m";
```

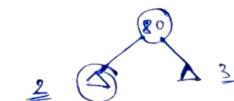
time complexity
 $\rightarrow \Theta(n)$.

Size of Binary Tree

五



O/P \rightarrow L

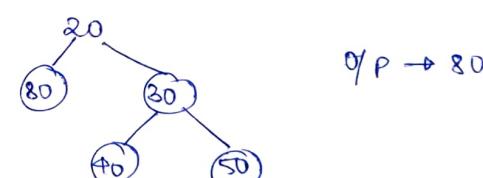


```
int getSize (TreeNode *root) {  
    if (root == NULL)  
        return 0;  
}
```

return 1 + getSize($\text{root} \rightarrow \text{left}$) + getSize($\text{root} \rightarrow \text{right}$)

Time complexity $\rightarrow O(n)$.
Aux space $\rightarrow O(n)$

Maximum in Binary Tree



```
int getMax (TreeNode *root) {
```

```

if (root == NULL) return INT_MIN;
else
    return max (root->data, max (
        getmax (root->left), getmax (root->right)));

```

1

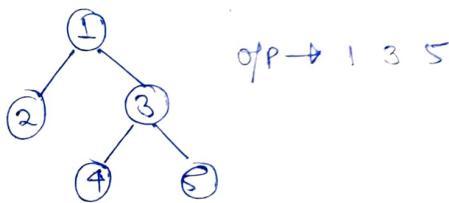
Print left view of Binary Tree

```

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

```

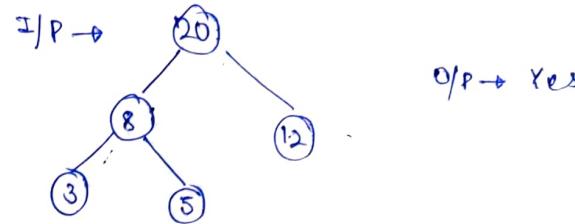
Right view of Tree



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

(Leetcode - 119)

children sum property



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

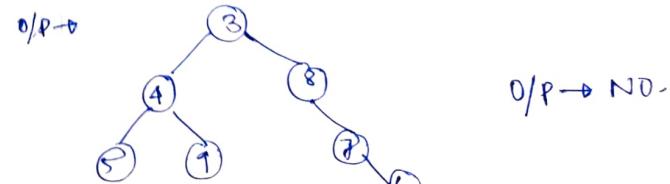
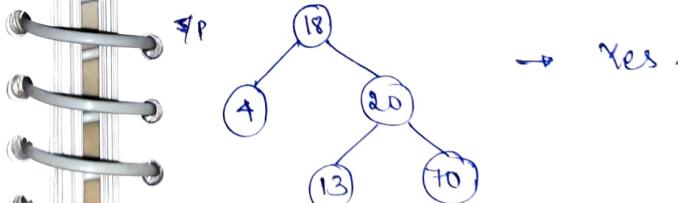
```

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



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

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));
}

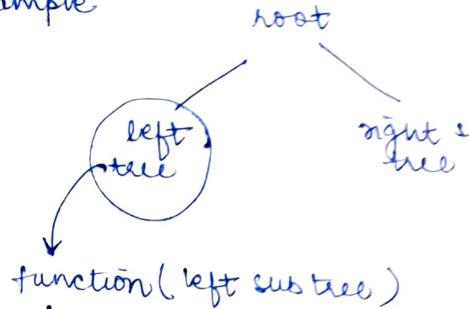
```

time complexity $\rightarrow O(n^2)$

Effective approach

- we need to 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 (lh, rh) + 1;
}

```

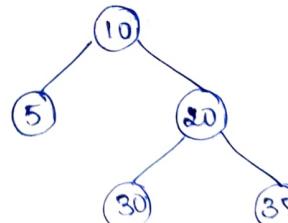
Maximum ^{height} width of Binary Tree

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

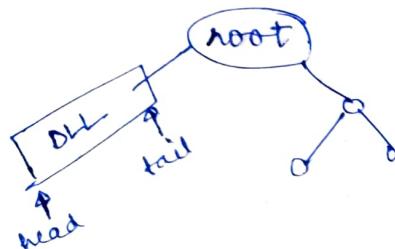
inorder →

```

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

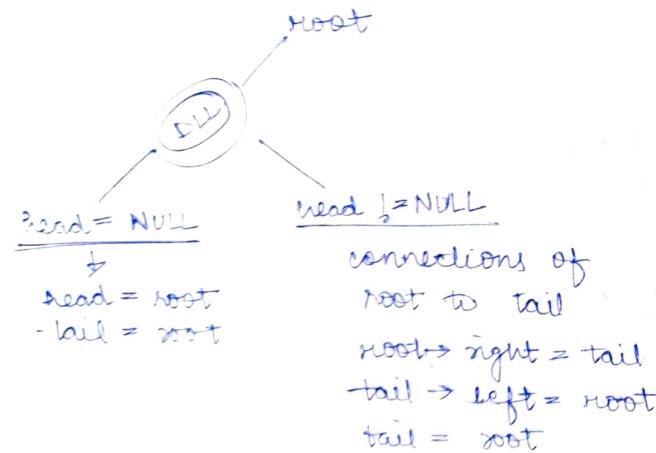
```

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



In this we have to pass pointers $\&$ by reference because we're changing the node where they the pointers are pointing.

To reflect these changes outside the function we can either use

$\text{Node}^* \& \text{root}$

or

$\text{Node}^{**} \& \text{root}$

```

void BTtoDLL ( Node *root, Node **&head = NULL,
                Node **&tail = NULL );

```

```

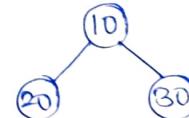
if (root == NULL) return NULL;
BTtoDLL (root→left, head, tail);
if (!head) {
    head = root;
} else {
    tail→left = root;
    root→left = tail;
    tail→right = root;
}
tail = root;
BTtoDLL (root→right, head, tail);

```

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

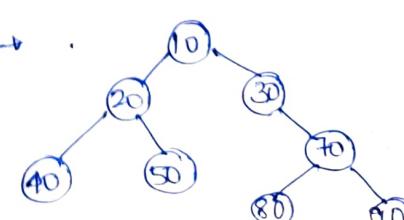
I/P : inp [] = {20, 10, 30}
pre [] = {10, 20, 30}

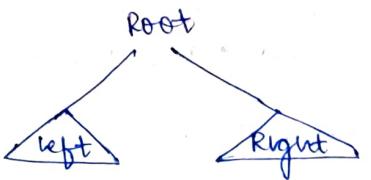
O/P :



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

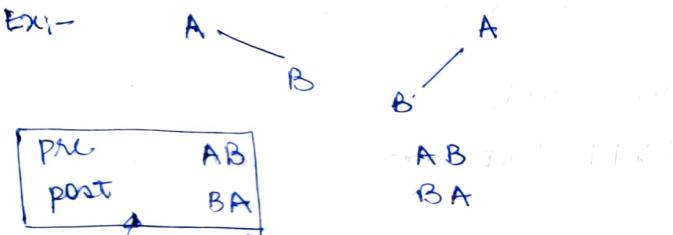
O/P →



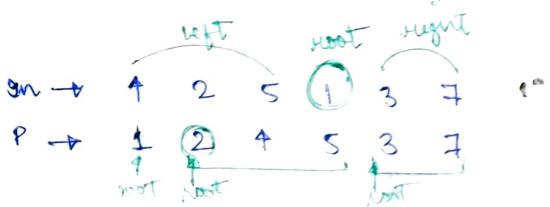


inorder \rightarrow left root right
pre \rightarrow root left right

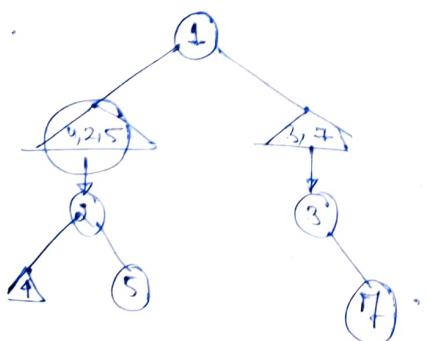
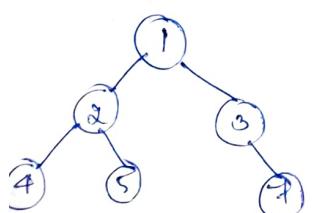
- we need inorder traversal to form a tree, we can't form a tree if we are given only preorder and postorder traversals
- Because inorder uniquely divides left and right subtree.



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



root will be first element of preorder traversal.



We will ~~see~~ 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 $\underbrace{9}_{\textcircled{3}}$ right $\underbrace{15 \ 20 \ 7}_{\textcircled{3}}$

(connect 3 with $\text{fun}(\text{root}, \text{left window})$)

at $i=1$

search 9 in inorder
 $\underbrace{9}_{\textcircled{9}} \ \underbrace{3}_{\textcircled{3}} \ 15 \ 20 \ 7$
only node (ie no child)

at $i=2$

search 20 in inorder
 $\underbrace{3}_{\textcircled{3}} \ \underbrace{3}_{\textcircled{3}} \ \underbrace{20}_{\textcircled{20}} \ 15 \ 7$

4 6 7 9 10 8 6
9 7 10 6 8 9 6
6 7 9 10 8 6 7
7 9 10 6 8 6 7



4 \rightarrow left = $\text{fun}(6, 9, 8)$
6 \rightarrow left = $\text{fun}(7, 9, 6)$
7 \rightarrow left = $\text{fun}(1, 7, 6)$
9 \rightarrow left = $\text{fun}(2, 6, 7)$
as required. 7 \rightarrow left = 7
6 \rightarrow left = 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~~ LBN

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

- ② We'll call ~~tree~~ → ~~root~~ → right first because as we are traversing from last the orders will be

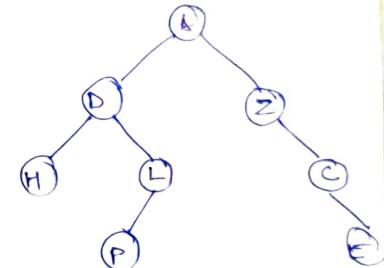
NRL
P ↑
node
node → right

Build Tree from Inorder and Level Order Traversal

Inorder - H D P L A Z C E
LevelOrder - A D Z H L C P E

Identifying root

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



HDPL → A → AZCE
Among this
the root is
whichever comes
first in LOT
Z is root

D
H D PL
H is root
D is left of H
P is left of L

Z CE
Z is left of
C is root
E is right of C

We will

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

Algorithm

```

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

```

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

We can use map to store level order traversal elements and their indices

→ Reduce the search complexity to $O(n^2)$

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

```
if (start > end)
    return NULL;
```

```
int mIndex = start;
```

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

```
    mIndex = j;
}
```

```
TreeNode * root = new TreeNode (inorder[mIndex]);
```

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

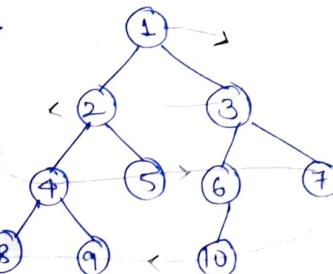
```
root->right = construct (inorder, m,
                           mIndex + 1, end);
```

```
return root;
```

3.

Tree Traversal: spiral form

I/P →



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

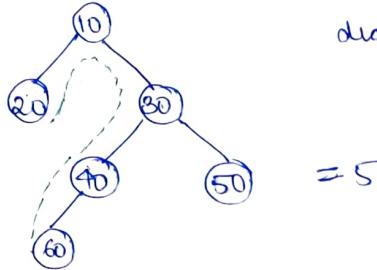
we will use two stacks here to store alternate
levels.



- 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 & point
 - (b) push children of the taken out node in reverse order.

Diameter of Binary Tree

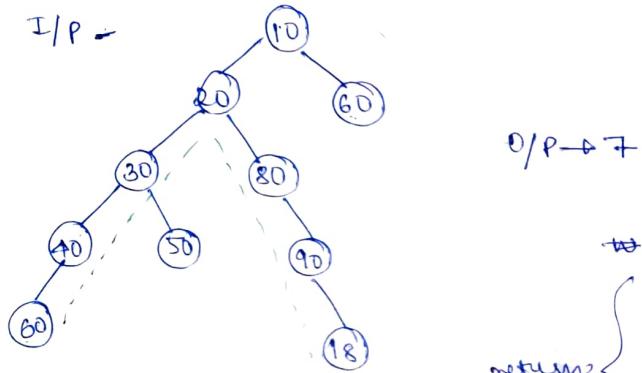
I/P



diameter = longest path b/w
two leaf nodes

= 5

I/P -



O/P → 7

For any node we need to calc.
 $1 + \text{lh} + \text{rh}$

return max from all nodes

left height right height

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));
}
  
```

5.

$O(n^2)$

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.

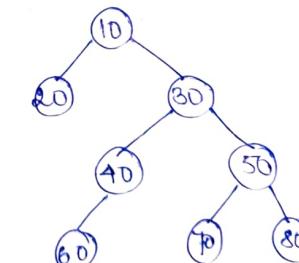
```

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

6.

Highest Common Ancestor (LCA)

I/P



Ancestor of 40 → 30, 10 &
ancestor of 70 → 30, 50, 10

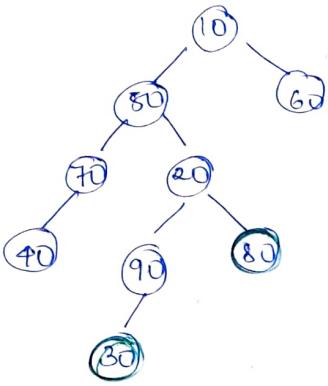
↳ lowest among them → 30

LCA (40, 80) → 10
 LCA (80, 30) → 30
 LCA (70, 20) → 50

• LCA is also helpful
in finding the
minimum distance
between two nodes.
→ we can find the
LCA of two nodes &
then calculate
distance of both
from LCA

Naive solution

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



path1 → {10, 50, 20, 90, 30}
path2 → {10, 50, 20, 80}

The common among them are

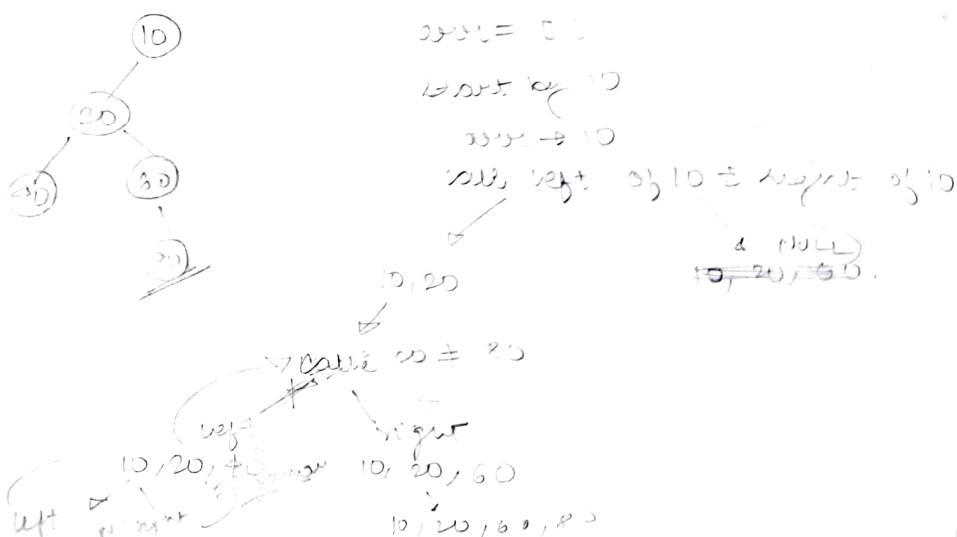
10, 50, 20
└── f
└── ↗ p → 20
└── closest

• 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 fundPath (Node *root, vector<Node *>&p, int n);
```

if (root == NULL) return false;
p.push_back (root);
if (root->key == n) return true;
if (fundPath (root->left, p, n))
 fundPath (root->left, p, n)) return true;
p.pop_back ();
return false;

5.

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

if (path1[i] != path2[i])
 return path1[i];
return NULL;

O(n) &
(but requires
2 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

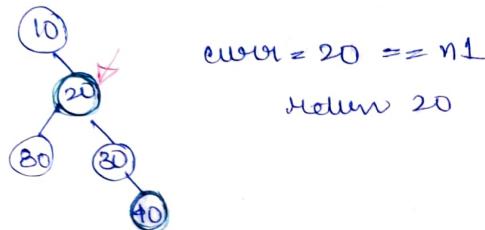
Ex. let curr is the node which is in process

- (a) $\text{curr} == n_1 \text{ or } \text{curr} == n_2$

(return curr)

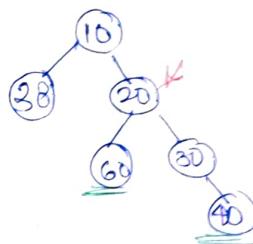
In this case curr is going to be the LCA as c2 is in lower level.

Ex



- (b) if of subtrees contain n_1 and other contain n_2

→ curr becomes LCA



for 10 → both are on left
for 28 → none is present
for 20 → one is on left
other is 40

Hence 20 is LCA

- (c) one of subtrees contain both n_1 and n_2
(we'll return whatever that subtree returns)

- (d) if none of subtree contain n_1 or n_2
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 {
 As both
 of them
 are not null.
 return root;
 }

 case-3 {
 if (lca1 == NULL)
 return lca1;
 else
 return lca2;
 }

}

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



Ex: ↙
 $n_1 = 50$
 $n_2 = 30$

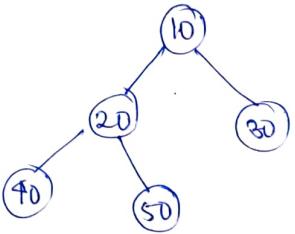
LCA → 20 ← return

~~O(n)~~
One traversal
of tree

(Hence it works only
when both are present)

Count Nodes in complete Binary Tree

I/P



O/P $\rightarrow 5$

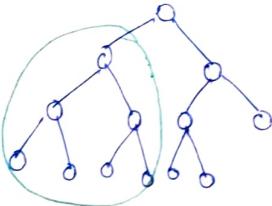
Naive Solution

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

return $1 + \text{fun}(\text{root} \rightarrow \text{left}) + \text{fun}(\text{root} \rightarrow \text{right})$

Effective Approach

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



If a tree is complete then the number of

$$\text{nodes are } = 2^d - 1$$

where $d \rightarrow \text{depth}$

- * To check whether a tree is complete or not we can use $\text{root} \rightarrow \text{left} \rightarrow \text{leaf} \dots \rightarrow$ length of left most branch

$\text{root} \rightarrow \text{right} \rightarrow \text{leaf} \dots \rightarrow$ length of right most branch

- if the subtree is not complete switch to naive method $\text{1} + (\text{root} \rightarrow \text{left}) + (\text{root} \rightarrow \text{right})$

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;
```

```
}
```

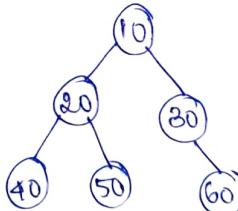
```
    return 1 + countNode (root->left) + countNode (root->right); } }
```

$\Theta(\log n \cdot \log n)$

directly returning 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

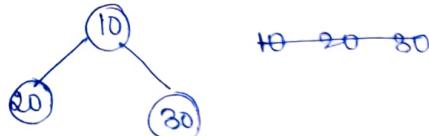
the Solution

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

* It requires two traversal of binary tree

Effective approach

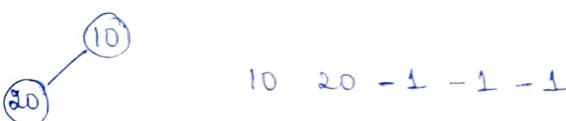
①



→ 10 20 -1 -1 30 -1 -1

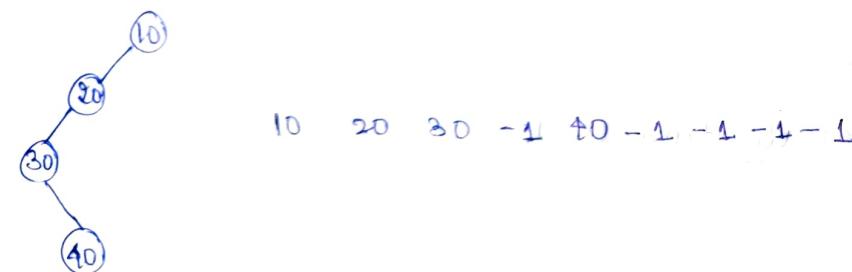
we use -1 for
NULL
assuming -1 is
not present

②



10 20 -1 -1 -1

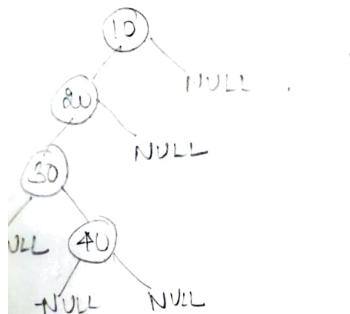
③



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

Deserialize

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



when we see a
-1 we put NULL
null

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);
}
  
```

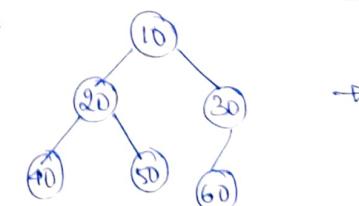
Deserialization

```

Node * deserialize (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



→ 10 20 50 10 60 80