* BINARY TREE

Raj
Page:
Date: / /

Data is stored in hierarcy order (unlike till now - linear)

TERMS

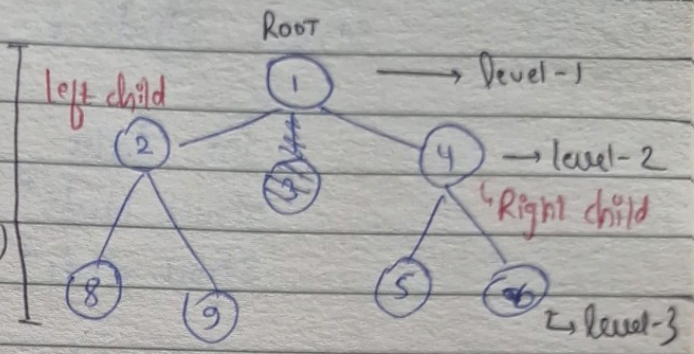
In Binary Tree, each node can have only 2 childs (as suggest in name)

Root Start point of BT

Branch Branch of each node

child node associated to primary node through branch

Leaves last/ ending node is leaves



PROPERTIES

(1) Max. nodes at level-L = 2^L

(2) Max nodes for height - H = $2^H - 1$

(3) For N nodes, minimum no of levels or min height possible = $\log_2(N+1)$ — log

(4) A BT with L leaves has atleast $\log_2(N+1)+1$ no. of level

MAKE A BT

Raj
Page:
Date: / /

#CODE

① Make a Structure (class) Node → with left / Right child

```
Struct Node {

    int data                    value
    struct Node* left ;         left pointer
    Struct Node* Right ;        Right "
```

constructor to initialise left right as NULL

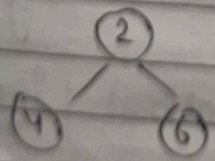
```
    Node (int val) {

        data = val
        left = NULL ; Right = NULL ;
    }
};
```

② Make a tree

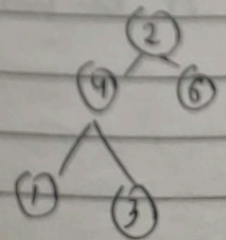
```
int main () {
```

make a root node → with value = 2

```
Struct Node* root = new Node (2)
root → left = new Node (4) ;
root → right = new Node (6)
```

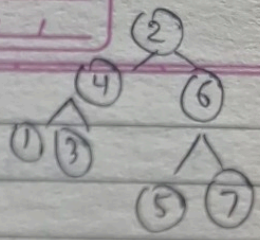



```
root → left → left = new Node (1)
root → left → right = new node (3)
```



Raj
Page:
Date: / /

root → right → left = new Node (5)
root → right → right = new Node (7)



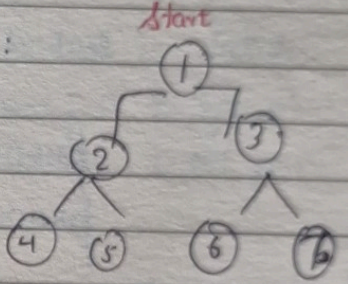
BT traversal

① Preorder traversal.

order → 1 2 4 5 3 6 Pre

Root
Left subtree
right subtree

eg:



② Inorder trav

4 2 5 1 6 3 7 In

3 NULL NULL 3 Right NULL NULL

left subtree (all)
Root
Right subtree

③ Post order traval

4 5 2 6 7 3 1 Post

Left true
Right true
ROOT

Recursive functions for traversal

Raj
Page:
Date: / /

① void preorder (struct Node* root) {

~~cout << root~~ if (root == NULL)

Base condition { return ; }

cout << root → data ;

preorder (root → left)
preorder (root → right)

}

② Inorder (Struct Node* root)

~~if (root == NULL) { return ; }~~

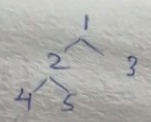
~~cout << root → o~~

Inorder (root → left)
cout << root → data << " ";
Inorder (root → right)

③ Postorder (struct Node* root)

if (root == NULL) { return ; }

Postorder (root → left)
Postorder (root → right)
cout << root → data



Raj
Page:
Date: / /

BUILD TREE FROM Inorder & Preorder

e.g: Pre | 1 | 2 | 4 | 5 | 3 | ; Inorder | 4 | 2 | 5 | 1 | 3 |

Algo:-

1) Pick element from preorder and create node
2) Increment Preorder Index
3) Search for picked element ~~in~~ position in inorder
4) Call to build left subtree from i 0 to (posit^n - 1)
5) " " " right " " inorder 0 to (posit^n - 1)
6) Return Node

WORKING

PREORDER

1 2 4 3 5
i

(node = 1)

for left of 1

1 2 4 3 5
i

(node = 2)

left of 2

1 2 4 3 5
i

node = 4

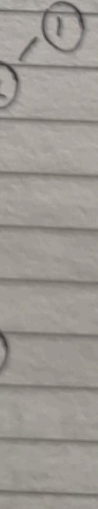
INORDER

4 2 1 5 3
s curr e

4 2
curr

4
curr

return to prev node



Raj
Page:
Date: / /

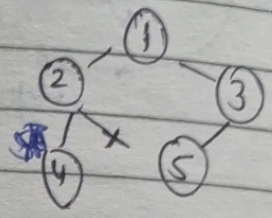
- It will return to '2' → Nothing in right → will return to 1 → right Bana.

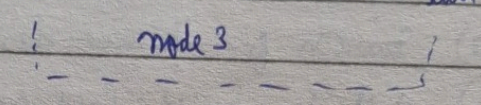
Pre

1 2 4 3 5

In

5 3

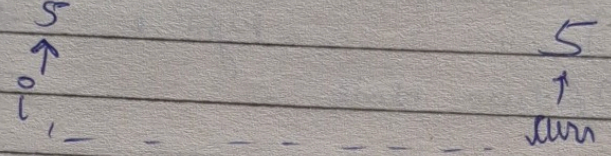




∴

1 2 4 3 5

5



CODE

Raj
Page:
Date: / /

* Pre & In order

```
Node* buildtree (int pre[], int In[], int start, int end)
{
    Static int indx = 0;

    int curr = pre[ind]
    indx++;
    Node* node = new Node (curr);

    int pos = search (in, start, end, curr);
    node → left = buildtree (preorder, in, start, pos - 1)
    node → right = buildtree (pre, in, pos + 1, end)

}
```

* Post & In order

```
Node* buildtree (int post[], int In[], int start, int end)
{
    Static int ind = n-1;
    int curr = pre (ind).
     ind--;
    Node* node = new node (curr)
    int pos = search (In, start, end, curr)
        if (start == end) {return curr};
    node → right = build tree (post in, start(pos+1), end)
    node → left = buildtree (post, In, start, pos-1);
```

Raj
Page:
Date: / /

BUILD TREE FROM INORDER & POSTORDER

1) Start from post_order and pick element to create a node

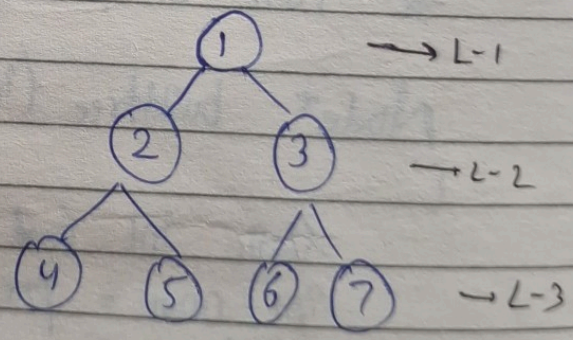
2) Decrement postorder Index

3) Search for picked element's pos in order

4) Call to build right subtree from Inorder (pos+1) to (n)
   " " left " " " (0) to (pos-1)

LEVEL ORDER TRANSVERSAL, Sum at K^{th} level in BT

1) Insert all nodes at queue and at end of each node insert NULL and once NULL comes then take out earlier element(node) and examine it



Raj
Page:
Date: / /

F QUEVE B

[1] [NULL]

- As null comes, remove Phili wala node

[1] → add in sequence or print it

[2] [3]
[NULL]

- As we have taken left/Right of node [1]

- If NULL comes out then a level has ended

[2] [3] [NULL]

[2] now add its left/Right child

[3] [NULL] [4] [5]

[NULL] [4] [5] [6] [7]
[3]

[4] [5] [6] [7] → chl
[NULL]

can't add any of their child

[NULL]

NULL is removed and Queue is empty

∴ we can exist

Raj
Page:
Date: / /

Shortcut — Array representation of BT

10	7	4	3	11	14	6

Parent index = i
Left son = 2i + 1
Right son = 2i + 2

```
Struct Node {
    int data;
    struct Node* left;
    struct Node* Right;

    Node (int val)
    {
        data = val
        left = NULL ; Right = NULL
    }
};
```

➤ levelOrder (Node* root)

```
int main () {

    struct Node* root = new Node(1)
    root → left = new Node (2)
    root → right = new Node (3)

    levelOrder ( root);
return 0;
}
```

```
levelOrder (Node* root)
{ if (root == NULL) return;

    queue <Node*> q;

    q.push ( root) ; q.push (NULL)

    Node* fnode = q.front();

    while ( ! q.empty() )
    {
        q.pop();
        if (fnode != NULL)
        { cout << fnode → data ;
            if (fnode → left != NULL)
                { q.push (fnode → left) }
            if (fnode → right != NU
                { q.push (fnode → right }
        }
        else if (!q.empty())
            { q.push (NULL);
    }
```

7

Raj
Page:
Date: / /

Height and Diameter in BT

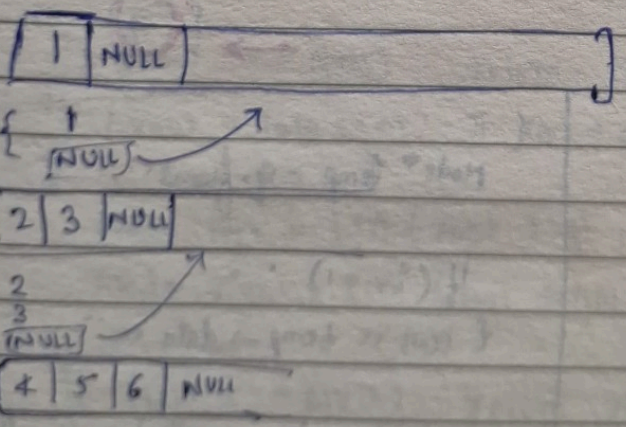
- Height

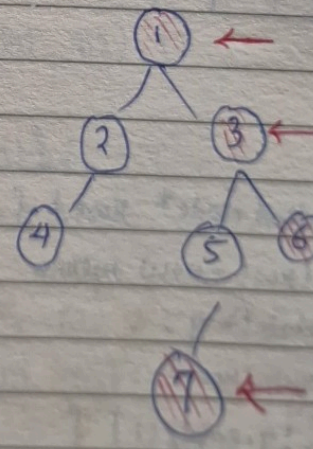
WRONG

```
int height (Node * root)
{
    if (root == NULL) return 0;

    int leftheight = height (root->left);
    int rightheight = height (root->right);

    return max (leftheight, rightheight) + 1;
}
```

~~You can add 1 individually also.~~

RIGHT VIEW





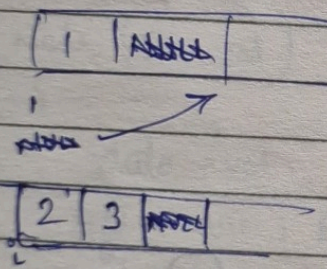
Page: Raj
Date: / /

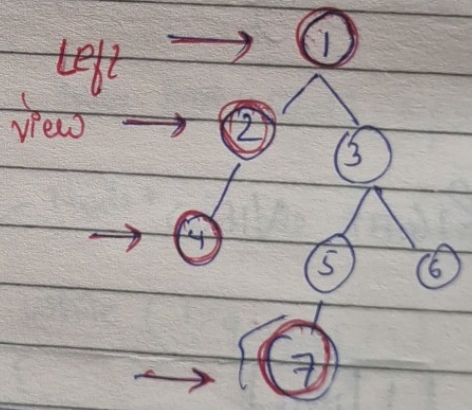
```
void rightview(Node* root)
{ if(root==NULL) return;

queue<Node*> q;
q.push(root);
q.push(NULL);

while(!q.empty())
{
```

```
Node* node = q.front();
q.pop();
if(node != NULL)
{
    if(q.front()==NULL){ cout<<node->data; }
}
Insert left & right Node { if(node->left!=NULL) q.push(node->left);
                           if(node->right!=NULL) q.push(node->right); }
else if(!q.empty()){ q.push(NULL); }
```

* NULL ke just pehle wala node is rightmost Node

LEFT VIEW






```
void leftview(Node* root) {
    if(root=NULL) return
    q<Node*> q;
    q.push(root)
while(!q.empty()) {

    int n=q.size();
for(int i=1; i<=n; i++)
```

```
{
Node* temp = q.front();

if(i==1)
{ cout<<temp->data<<" "; }

if(Node->left!=NULL) q.push(temp->left)
if(Node->right!=NULL) q.push(temp->right)
```

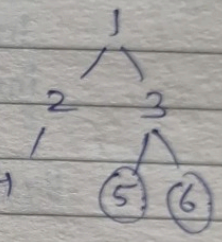
Page: Raj
Date: / /

Minimum dist b/t 2 Nodes

↳ no. of nodes (edges) to be traversed to reach from one node to another

• Steps:

1) Find lowest common ancestor (LCA)
2) Find distance of n_1, n_2 from LCA
3) dist b/t n_1 to n_2
4) d_1 = dist of n_1 from LCA
   d_2 = " " n_2 " LCA



$n_1 = 5$, $n_2 = 6$
dist = 2

1) Lowest Common Ancestor (LCA)

CASES
① CURR = a || curr = b
② a = left subtree
   b = right subtree
③ Both a, b in any subtree
④ None of a, b in any subtree

```
Node* LCA (Node root, int n1, int n2) {

    if (root == NULL) return NULL;
    if (root → data == n1 || root → data == n2) return root;

    Node left = LCA (root → left, n1, n2);
    Node right = LCA (root → right, n1, n2);

    if (left == NULL) return right;
    if (right == NULL) return left;

return root;
```

mil gaya

Better Approach O(n)

Page: Raj
Date: / /

Min dist blt Nodes

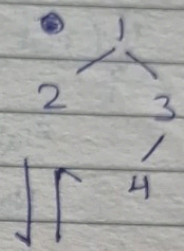
```
int findmindist (Node* root, int n1, int n2)
{
    if (!root)  return 0;

    int left = findmindist (root->left, n1, n2);
    int right = findmindist (root->right, n1, n2);

    // conditions

    if (left && right)  // LCA of n1, n2
    {
        mindist = left + right;
    }

    else if (root->data == n1 || root->data == n2)  // Current node is either n1, n2
    {
        if (left || right)
        { mindist = max(left, right)
          return 0;
        }
        else {
          return 1;
        }

    else if (left || right)  // Current node neither n1 nor n2 but coming in blt n1, n2
                return max(left, right) + 1;
    return 0;  // Current node != n1|n2 not coming in blt n1|n2 path
```

Page: Raj
Date: / /

Flatten the Binary Tree.

Flatten tree into linked list
(no other DS to be used)

BN



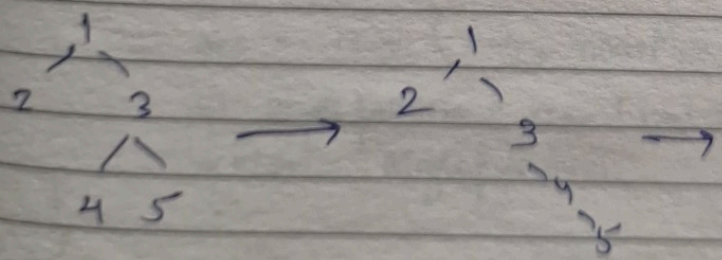
1 → 2 → 3 → 4

Flatten tree

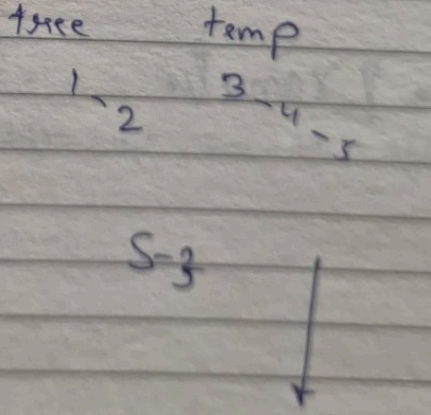
STEPS

1) Recursively, flatten left, right subtree
2) Store left and right tail (extremes)
3) Store right subtree in 'temp' and make left subtree as right subtree
4) join right subtree with left tail

VISUALIZE



S-1



S-3

1-2-3-4-5

S-4

Raj
Page:
Date: / /

Code :

```
void flat (Node* root)
{
    if (root != NULL)
    {
        flat (root → left) ;
        Node* temp = root → right
        root → right = root → left
        root → left = NULL ;

        Node* t = root → right
        while (t → right != NULL)
        {
            t = t → right
        }
        t → right = temp ;
    }

    flat (root → right) ;
}
```

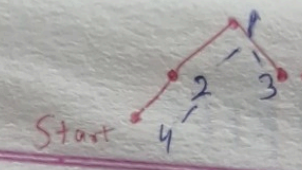
Base case → `if (root == NULL || root → left == NULL && root → right = NULL) { return ; }`

Storing right tree in temp and left ko NULL

Calculate tail of right tree

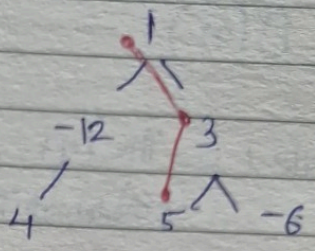
Now t = tail of right tree

- Diameter



Raj
Page:
Date: / /

Max Sum Path



- Strategy :

For each Node:

1) Node val
2) Max path through left child + node val
3) Max path through right child + node val.
4) Max path through: left + right + node val.

CODE

```
int maxSumUtil ( Node* root , int &ans) {

    if ( root == NULL)      return 0;

    int left = maxSumUtil ( root→ left, ans )
    int right = maxSumutil ( root → right , ans)

    int nodeMax = max ( root→ data , max (root→data + left,
                                          root data + right)

    ans = max ( ans , nodeMax );
    int singlePathSum = max ( root → data + left , root → data + right);
    return singlePathSum;
}

int maxSum ( Node* root ) {
        int ans = INT_MIN;
        maxSumUtil (root, ans)
        return 0;
}
```

BINARY SEARCH TREES

Page: Raj
Date: / /

BT → BST

$O(n) \longrightarrow O(\log n)$ Time comp. of Search

- Rule-1 : The left subtree of Node contains only nodes with keys lesser than node's key
- Rule-2 : The right subtree of Node contains only nodes with key greater than Node's key
- Rule-3 : Left & right subtree must be a binary search tree. There must be No duplicate nodes

CODE

```
Struct Node {
    int data;
    Node *left, *right;

    Node (int val) {
        data = val;
        left = right = NULL;
    }
};
```

For LCA of BST, whichever node have both the no. on either side is LCA

Raj
Page:
Date: / /

// Insert in BST

```
                                        int val
Node* insertAtBST (Node* root)
{
    if (root == NULL)  return new Node (val);

    if ( val < root → data)
    {
        root → left = insertAtBST ( root → left , val);
    }
    else {
        root → right = insertAtBST ( root → right , val);
    }

    return root;
}
```

// Search in BST

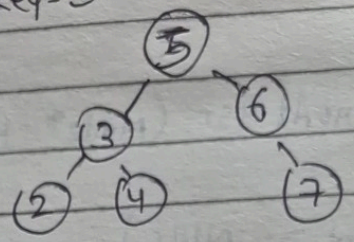
```
Node * SearchBST (Node* root , int key) {

    if ( root == NULL)      return NULL ;
    if ( root → data == key)    return root ;
else if (root → data < key)     return SearchBST (root → right , key);
else                        return SearchBST ( root → left , key) ;

}
```

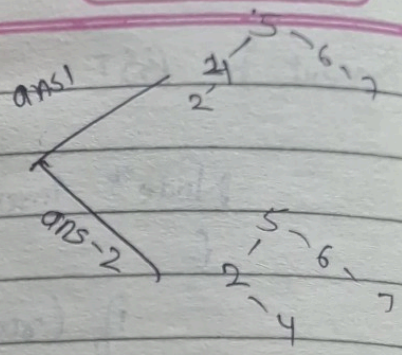
Raj
Page:
Date: / /

(node from)

Delete a BST.

Key = 3





Case 1 : Only one child

Case 2 : Both left, Right ✓

↳ delete the leftmost element on right subtree after replacing with root value

pdf

```
TreeNode* deleteBST(root, key)
{
    if (!root) return NULL;
    // Search the key node.
```

Search

```
if (key < root->val)   root->left = deleteBST(root->left, key)
if (key > root->val)   root->right = deleteBST(root->right, key);
```

Found

```
else {

// No left child
if (root->left == NULL)
{
    TreeNode* temp = root->right;
    delete root;
    return temp;
}
// No Right child.
if (!root->right)
{   temp = root->left;
    delete root
    return temp;
```

```
// Have Both Child (left + Right)
Find leftmost node in right subtree
TreeNode* leftmost = root->right
while (leftmost->left)
  { leftmost = leftmost->left;
  }

root->val = leftmost.
```

BST can be build by preorde

Raj
Page:
Date: / /

BUILD

BST from Preorder

left (NULL, data) | 7 | 5 | 4 | 6 | 8 | 9 | Right (data, NULL)

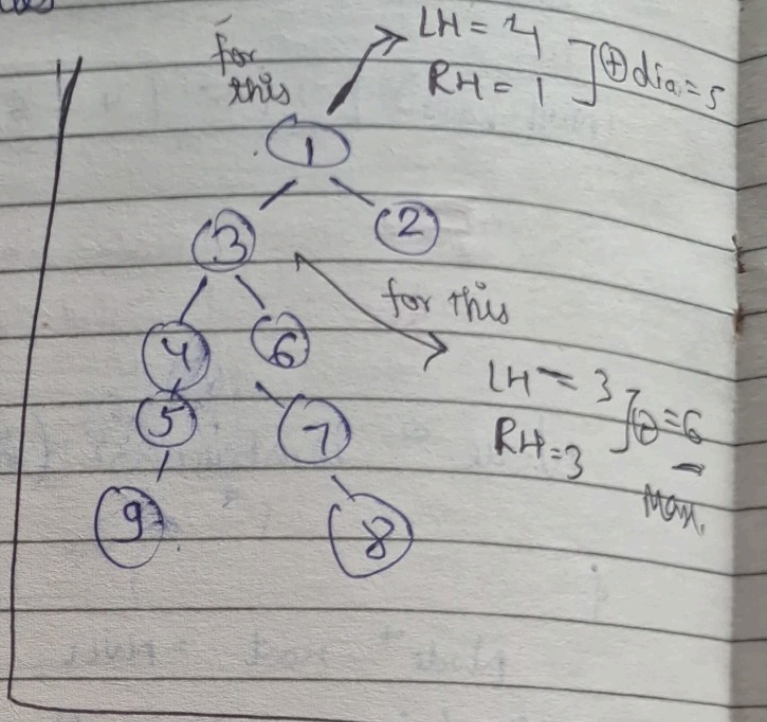
For every Node there is (min, max) pair for left / right subtree

```
Node * constructBST (int preorder[], int* preorderIdx,
                     int key, int min, int max)
{
    Node* root = NULL
    if (key > min && key < max) {
    {
        root = new Node(key)
        *preorderIdx = *preorderIdx + 1;

        if (*preorderIdx < n)
        {
            root->left = constructBST (preorder, preorderIdx, preorder[*preorderIdx]
                                       , min, key, n)
        }
    if (*preorderIdx < n)
    {
        root->right = constructBST ("", "", preorder[*preorderIdx]
                                    min, key, n)
    }
}
```

Raj
Page:
Date: / /

- Diameter of BT
  Longest path bet any two nodes

- find height of (left, right) for each node
  +
  max of (left + Right)
  is diameter



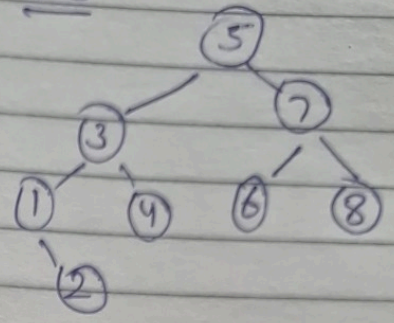
K^{th} Smallest element

O(N)
O(1)

Raj
Page:
Date: / /

- Do inOrder traversal which is sorted in case of BST
- Use counter until "k"

k=3




```
void inorder (root, &k, &ans)
{
  if (!root) return
  inorder (root->left, k, ans)
  k--;
  if (k==0) { ans = root->val
  } return;
  inorder (root->right);
}
```

```
int Kth smallest (root, k)
{
  int ans = -1;
  inorder (root, k, ans);
  return ans.
}
```

CHECK for VALID BST

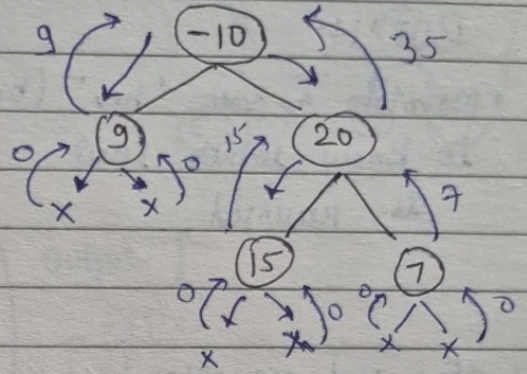
INT_MAX → ∞ , INT_MIN → first call value

```
bool check (root, maxi, mini)
{
  if (!root) return true;
  if (root->val >= maxi ||
      root->val <= mini)
    return false
  return {check(root->left, root->val, mini) &&
          check(root->right, maxi, root->val)};
}
```

BINARY TREE (PT-2)

Raj
DATE / /
PAGE

Max. Path Sum




```
int mPS (TreeNode* root)
{
    int maxi = INT_MIN;
    maxPathSum(root, maxi)
    return maxi;
```

CODE

```
int maxPathSum (TreeNode* root, int &maxi)
{
    if (root == NULL) return 0;
```

Better take '0' than a -ve value.

DP concept for tree backtrack

```
    int left = max (0, maxPathSum(root->left, maxi));
    int right = max (0, maxPathSum(root->right, maxi));
```

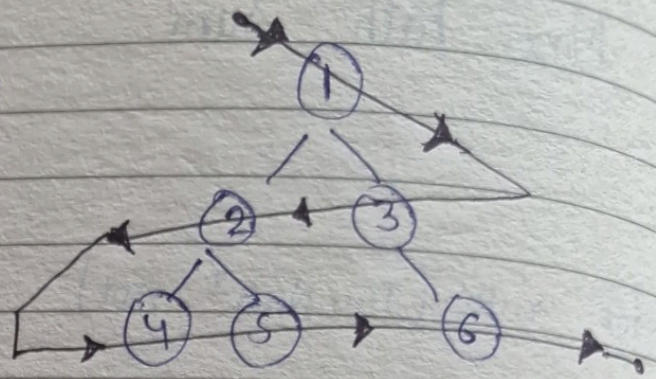
Contain the maximum sum ever stored.

```
    maxi = max(maxi, (right + left + root->val));

    return max(left, right) + root->val;
}
```

we can only return one path from a Node
(eg: (20) we will only take left + (20))

ZIG - ZAG traversal

- Same as level order traversal BUT!!!
- Maintain a var 'rev' (bool) to know which level is reversed.



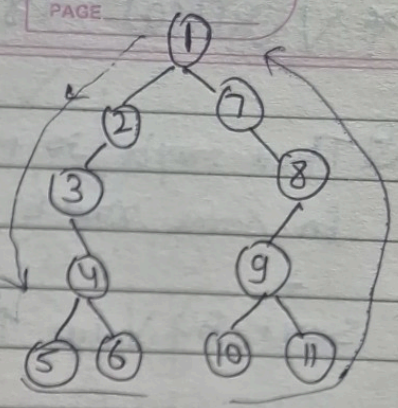
output [(1), (3, 2), (4, 5, 6)]

CODE

```
bool rev = 1;
while (!q.empty())
{
    int size = q.size();      vector<int> row;
    for (i=0 to size)
    {
        TreeNode* node = q.front();      q.pop();
        int index;
        if (rev == 0)  index = size-1-i;   } insert from back
        else           index = i;          "    "  front
        row[index] = node->val;
        if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
    }
    ans.push_back(row);
    rev = !rev;
}
```

Boundary Traversal. (AntiCKW)

Raj
DATE / /
PAGE



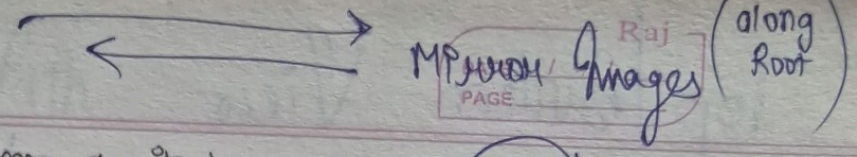

```
bool isLeaf (TreeNode* root)
    return (node->left == NULL && node->right == NULL);

void addleftbound (root, ans)
{ TreeNode* curr = root->left;
    while (curr)
    { if (!isleaf(curr)) ans.push_back(curr->val)
      if (curr->left) curr = curr->left;
      else            curr = curr->Right;
    }

Void addrightbound (root, ans)
{ TreeNode* curr = root->right ; vector temp
    while (curr)
    {
    }
  reverse temp
    ans.push_back (temp);
}

void addleaf (root, temp)
{
    if (!root) return;
    if (isleaf (root))
      ans.push_back (root->val)
      return
    }
    addleaf(root->left, ans)
    addleaf (root->right, ans)
}
```

Output: [

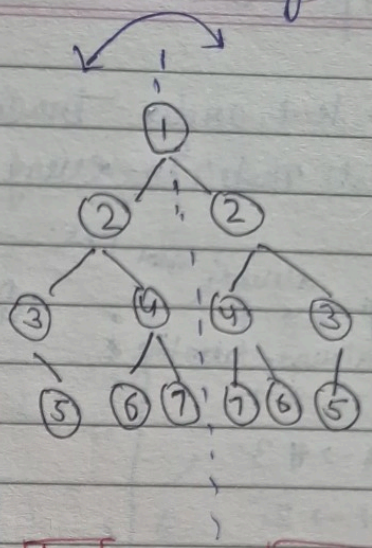
App:
- Left Boundry (exclude Leaf) By Preorder
- Leaf Nodes
- Right Boundary (exclude leaf)

```
Vector<int> Main funct" (root)
{
  Vector <int> ans;
  if (!root) return ans;

  if ( ! isleaf (root)
  { ans.push_back (root);
    return ans;
  }
    addleftbound ( )
    addleaf ( )
    addrightbound ( )
}

return ans.
```

SYMMETRIC TREE



check if it is mirror image of itself




```
bool util (leftRoot, RightRoot)
{
        // Both NULL
  if (!leftRoot && !RightRoot)
      return true

  // one of them is NULL
  if (!leftRoot || !RightRoot) return false
  // value not equal
  if (value not equal)  return false
  // Iterate through true
if (util (LR->left, RR->right) &&
    util (LR->right, RR->left) )
  {
    return true
  }

else return false
}

bool SymmTree (root)
```

Left (L) Right (R)

L→left should equal to R→right

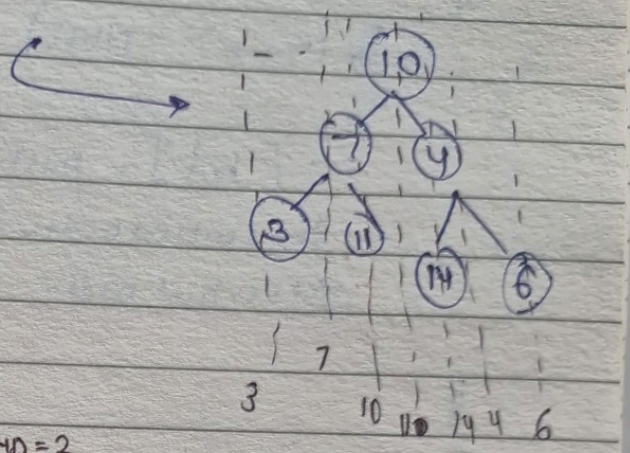
R→left should equal to L→right

Vertical order of Binary Tree

Page: Raj
Date: / /

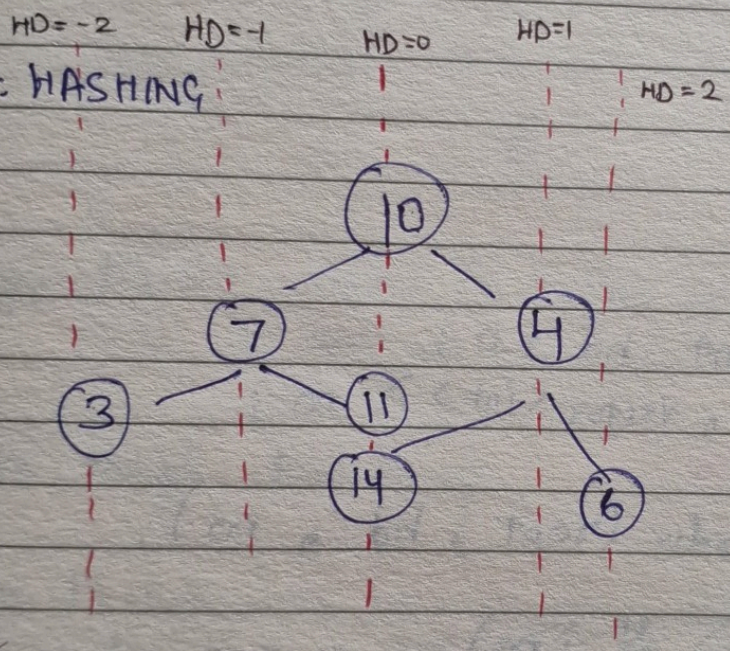
Q Print Vertical order of Binary tree in its array form

10	7	4	3	11	14	6



Parent index = i
left child " = 2i + 1
right child = 2i + 2

#Approach : HASHING



Key (HD)
0 : 10, 11, 14
-1 : 7
-2 : 3
1 : 4
2 : 6

STEPS

Base case : currNode == NULL {return}

1) Start from Root node
2) Recursively call left & right with (HD-1) and (HD+1) as arguments
3) Push value into vector corresponding to horizontal distance (HD)
4) Output Map

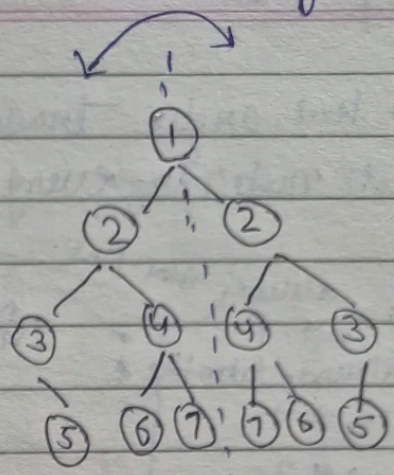
CODE

```
void getverticalorder (Node* root , int hd , map<int,vi> &m)
{
    if (root == NULL)  Return;

    m[hd].pushback (root -> key);
    getverticalorder (root -> left , hd-1 , m);
    getvertical order (root -> right , hd+1 , m);

}

main ()
{
    Tree
        int hd = 0;
    map < int , vector <int> > m;

    get vertical order (root , hd , m);

    for (auto it : m)
    {
        for (auto i : it.second)
        {
            cout << i << " ";
        }
    } cout << endl;
```

SYMMETRIC TREE

Mirror Images (along Root)

check if it is mirror image of itself



Left (L) Right (R)

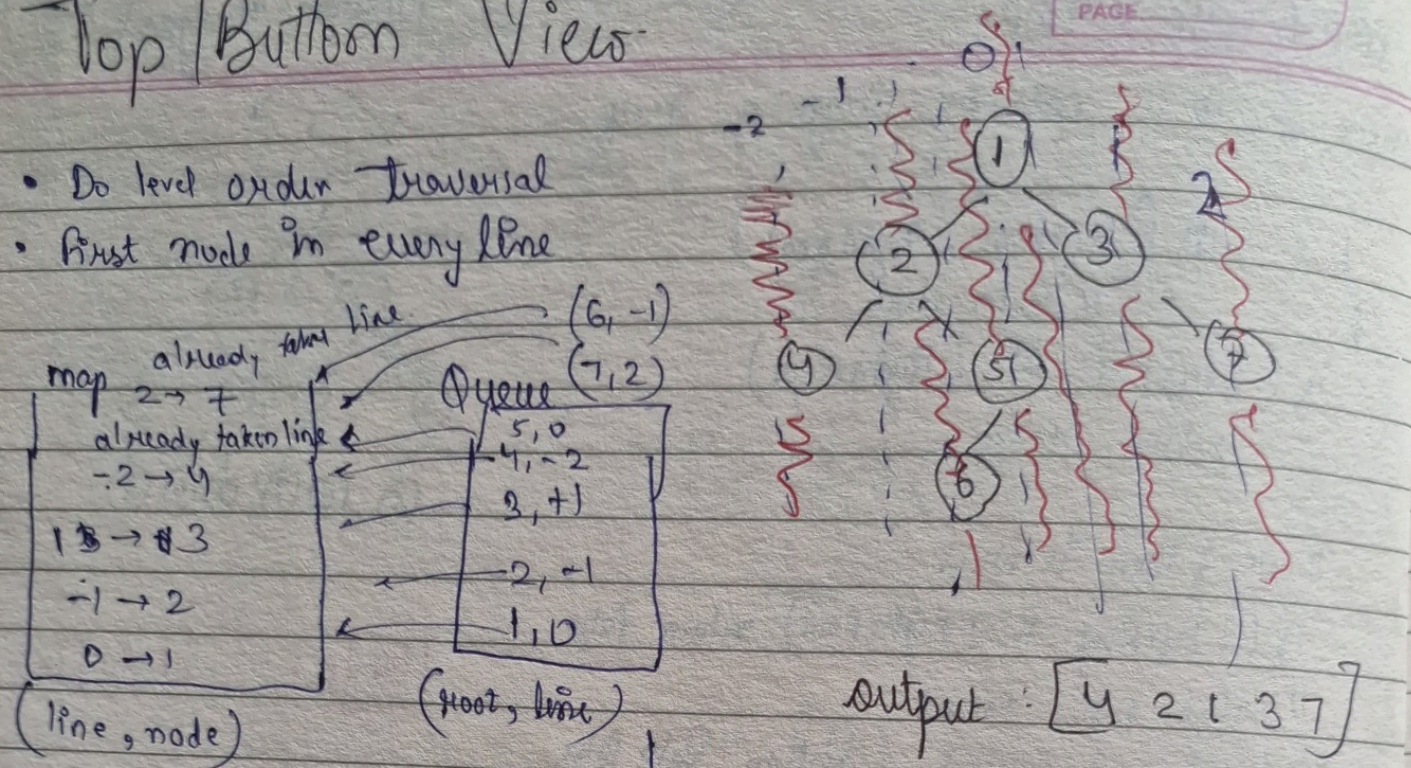
L → left should equal to R → right

R → left should equal to L → right

```
bool util ( leftRoot , RightRoot)
{
        // Both NULL
  if ( ! leftRoot && ! RightRoot)
      return true

  // one of them is NULL
  if ( ! leftRoot || ! RightRoot) return false
  // value not equal
  if ( Value not equal)   return ~~valu~~ false
  // Iterate trough true
if (util (LR→ left , RR→ right) &&
   util ( LR→ right, RR→ left) )
  {
    return true
  }
  else return false
}

bool SymmTree ( root)
```

Top / Bottom View

- Do level order traversal
- First node in every line



↓

If that line (key) already in map
Then don't consider it.

```
map<int, int> m
queue<pair<TreeNode*, int>> q
q.push(root);
q.push({root, 0});

while(!q.empty())
{
auto p = q.front(); q.pop();
TreeNode* node = p.first
int line = p.second

if(m.find(line) == m.end())
{
  m[line] = node->val;
}

if(node->left) q.push(node->left);
if(node->right) q.push(node->right);

}
```

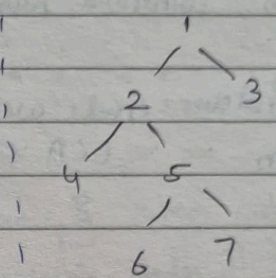
Root to Node Path

Raj
DATE / /
PAGE

Q Determine a path from root to given Node

- Use Inorder traversal

eg: Node = 7



output : [1, 2, 5, 7]

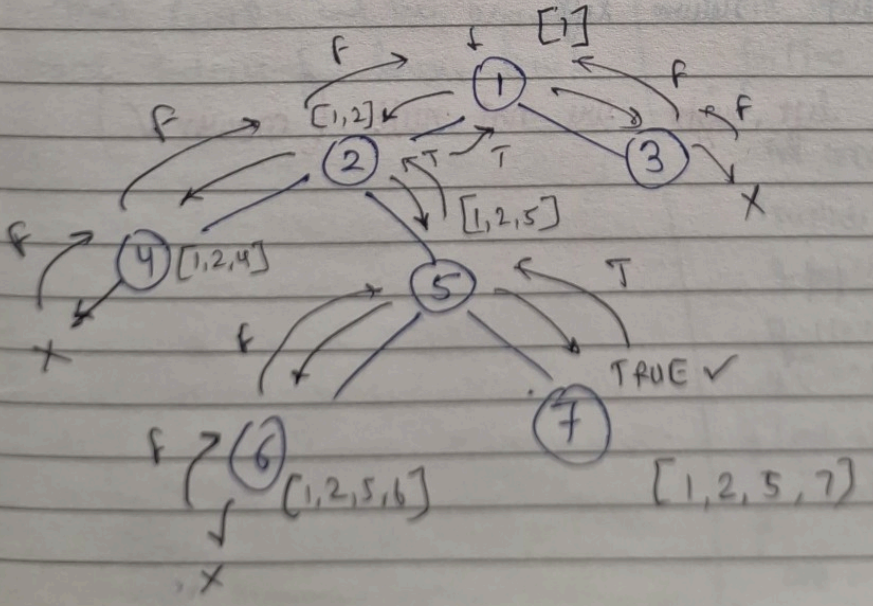
```
bool getPath(root, &arr, x)
{
    if(!root) return false;
    arr.push_back(root->val); Insert element
    if(root->val == x) return true;
```

Now check left and right subtree

```
    if(getPath(root->left, arr, x) || getPath(root->right, arr, x))
        return true;
```

Otherwise just pop this element & return false

```
    arr.pop_back();
    return false
```



LCA

Lowest common ancestor of 2 given Nodes (P, Q)

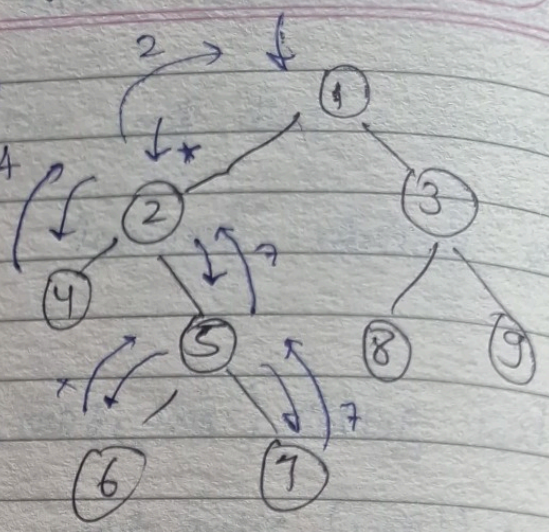
Raj

DATE / /

PAGE

We will traverse Tree and when we find node p/q we will return them (otherwise return NULL)

One whatever node we get p/q Both == LCA ✓



Input: (4, 7) output: [2]
" (2, 6) " [2]

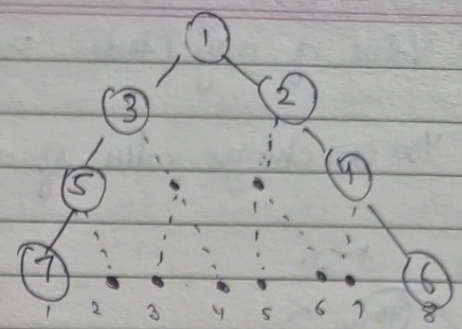
```
TreeNode* LCA(root, p, q)
{
  if(!root || root==p || root==q) return root

  TreeNode* left  = LCA(root->left , p, q);
  TreeNode* right = LCA(root->right, p, q);

  if(left==NULL)  return right
  else if(right==NULL) return left

  else // Both left, right are not null (answer ✓)
  return root;
}
```

MAX. WIDTH

Raj
DATE / /
PAGE

Find max width of BT ,?

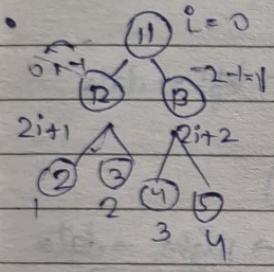
- We will do indexing of each level like 0 1 2 3 - - - -
  first last



Output : 8 = Max width
(max 8 possible nodes ⇑)

$i = (i - \text{MinIndex})$

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



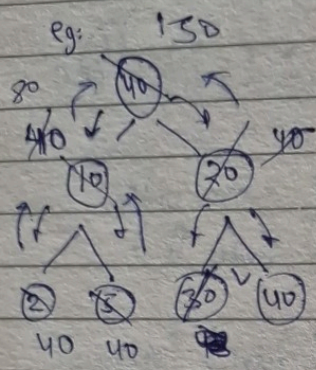
from original Index we Subtract minimum index of that level and then generate new indexes for lower level

```
int width (root)
{   int first, last
    if (!root) == 0;   int ans=0
    queue< pair< TreeNode*, int >> q
    q.push ({root, 0});
    while (!q.empty())
{
    int size = q.size();
    int imin = q.front().second;
    for(i=0 to size)
    {
    int currInd = q.front().second - imin
    TreeNode* node = q.front().first;
    q.pop();
    if (i==0) first = currInd
    if (i==size-1) last = currInd
    if (node->left) q.push(node->left, currInd*2+1)
    " " "right ..... right, currInd*2+2)
    }
    ans = max(ans, last - first+1
}
```

CHILDREN SUM PROPERTY

↳ Value of any Node = left child + right

You can change value of any node by 1 any no of times



1) while going down from root, we will inc lower value until we reach leaf.
(one only when child < Parent)

2) while going up from leaf nods we will make Parent = sum of childs so that tree will follow Property

NOTE: Top-down approach ensures that we never fall short while coming up

```
void CSP (root)
{
  if (!root) return
  int child = 0
  if (root → left)
    child += root → left → data

  if (root → right)
    child += root → right → data

  if (child >= root → data)
      root → data = child
  if child < root then make Both
    left / right = Root
  else {
  if (root → left) root → left → data = root → data
  if (root → right) root → right → data = root → data;
  }
  now left, right
  CSP (root → left)
  CSP (root → right)
  Now assign left right to root   int tot = 0
  if (root → left)   tot += root → left → data
  " " right   " " " " " right "
  if (root → left || root → right) root → data = tot
```

Print all

• Use BFS

Burn Tree (Minimum Time) from a Node

Raj
DATE / /
PAGE

Node = 2

Return min time to burn a BT

- Use BFS traversal until you can't

APP

1) We need track of all parents to traverse upwards

① Make Parents pointers.

Map: 7 → 4, 6 → 3, 5 → 3, 4 → 2, 3 → 1, 2 → 1 (node → Parent)

Queue: 7, 6, 5, 4, 3, 2, 1

2) From given Node (eg: 2)

Visited map: mark visited = =1

4 → 1, 1 → 1, 2 → 1

t=1

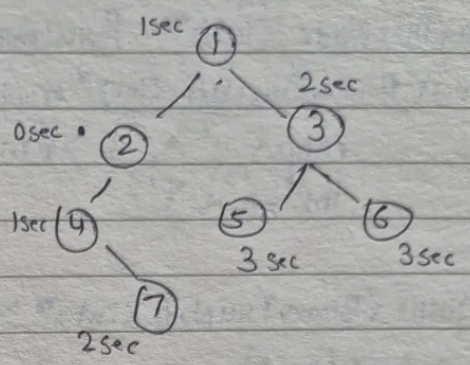
t=0

Queue: 1, 4, 2 (child (L+e) Parent)

each node can burn: up, Left, right

And make sure these elements are not repeated - By Map

~~Do Same~~



Output = 3 seconds

- Do same for all the nodes in Queue and pop them after

CODE explain

1) Parent Pointer give you map and (starting node) pointer to given node (target)

2) It will map by LVL order traversal and whenever we give start node it will store it in 'ans'

```
BinaryTreeNode<int>* ParentPointer
funct(BinaryTreeNode<int>* root,
      map<"+", "">& mp
      int start)
{
 queue<BinaryTreeNode<int>*> q
  q.push(root)
 BinaryTreeNode<int>* ans

while(!q.empty()){
 BinaryTreeNode<int>* node;
   node = q.front();
if(node->data == = start) ans = node;
   q.pop();
  if(node->left)                    Same with right
{   mp[node->left] = node
    q.push(node->left);
  }

}
  return ans;
}
int Burn(BTN<int>* root, int start)
  {
   map<BTN, BTN> mp
   BTN target = ParentPointer
                (row, mp, start)
int maxi = findDist(map, target);
 return maxi
```

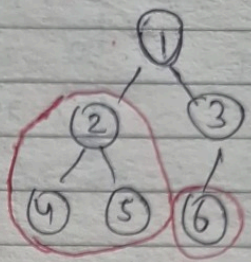
```
int findDist(mp, target)
{
  queue<BTN<int>*> q;
  q.push(target);                    Time.
map
  map<BTN, int> vis;   int maxi=0;
    vis[target] = 1; // mark visited as 1
  while(!q.empty())
{
  int size = q.size();
  int flag = 0;
  check for left, right, top for full queue
  for(i=0 to size)
{
  if(node->left && !vis[node->left])
  { flag=1; vis[node->left] = 1;
  } q.push(node->left);

  if(node->right && !vis[node->right])
{ flag=1; vis[node->righ]=1; q.push(node->right)
}
//TOP  TOP. check (Parent for mp MP)
if(mp[node] && !vis[mp[node]])
  { flag=1; q.push(mp[node]);
    vis[mp[node]] = 1;
  }
} even if you burn 1 node, inc the time
   if(flag==1) maxi++;
}
 return maxi
```

Count all node in < O(n)

Raj
DATE / /
PAGE

we will count nodes by formula $(2^{H}-1)$

∵ total possible node for height 'H' $= (2^{H}-1)$




```
int CountNodes (root)
{
  if (root == NULL) return 0;

  int lh = leftheigh (root)
  int rh = rightheight (root)

  if (lh == rh) return (2^lh - 1);
  else{
    return countNodes (root → left) + countNodes (root → right) + 1 ;
  }
}
```

Plus 1 so that you include the parent

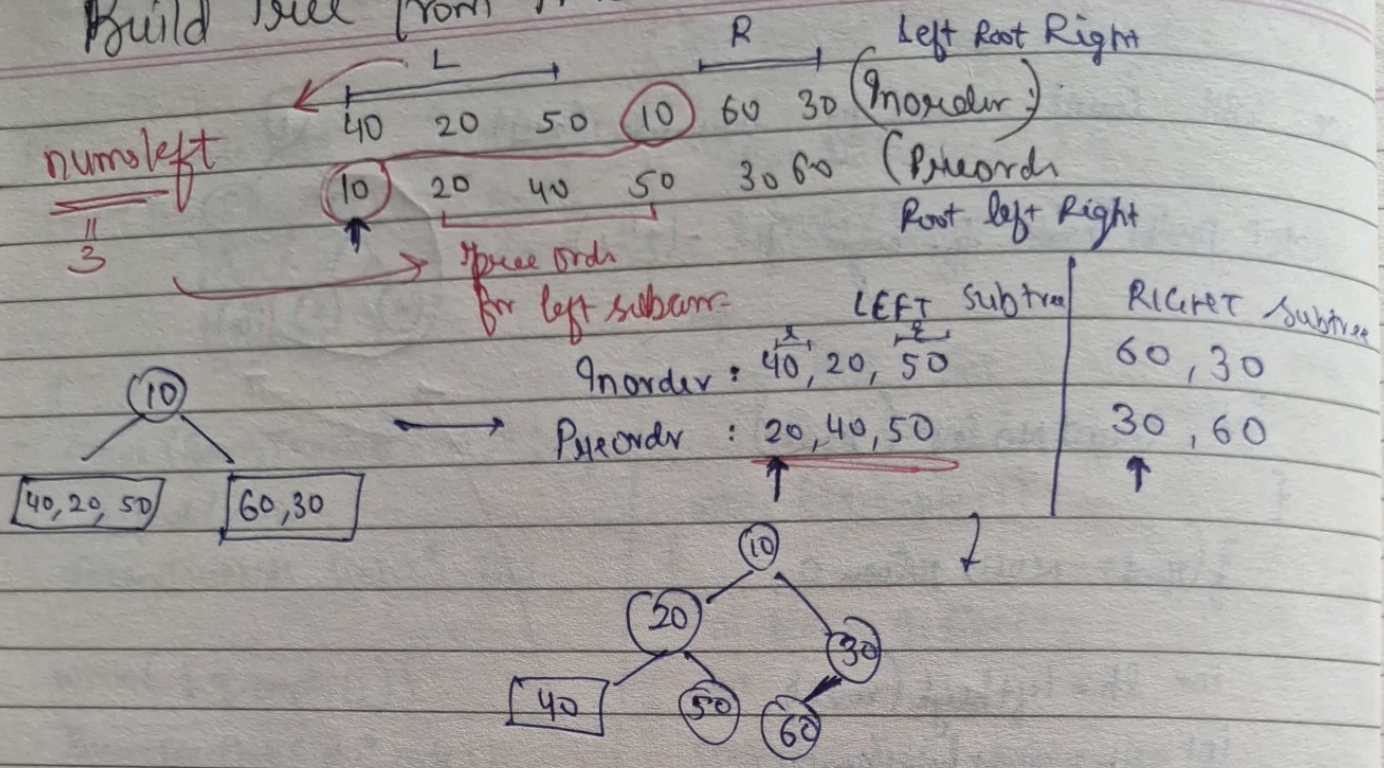
```
int leftheight (root1)
{ int height = 0;
  while (root1)
  { height ++;
    root1 = root1 → left ;
  }
return height
```

{ } same for right

Build Tree from PRE & IN order

Raj

DATE / /
PAGE



1) Map all the Inorder element to their indices (key)

2) Base case preStart > preEnd || inStart > inEnd

3) Determine first preorder element - Make a node, find its position in Inorder array. = x
left of x = Inorder of left subtree = numsleft
right of x = Inorder of right subtree

preStart + numsleft = preorder for left subtree

Raj
DATE / /
PAGE

Code

```
TreeNode* buildTree(vector pre, vector in)
{
    KEY  Value  Index of Inorder vector
    map<int,int> inmap; // Map Inorder vector with its Index
    for(i=0 to in.size())
    { inmap[in[i]] = i;
    }
    TreeNode* root = Util(pre,0,pre.size()-1, in,0,in.size-1, inmap)
    return root;
}

TreeNode* Util(pre[], prestart, preend, in[], instart, inend, inmap)
{
    // Base Case
    if(prestart > preend || instart > inend) return null

    TreeNode* root = new TreeNode(pre[prestart])

    int inroot = inmap[root->val]
    int numsleft = inroot - instart;   *   use numsright for Post order

    root->left = util(pre[], prestart+1, prestart+numsleft,
                      in[], instart, inroot-1, inmap)

    root->right = util(pre[], prestart+numsleft+1, preend,
                       in[], inroot+1, inend, inmap)
                  (+1)

    return root;
```

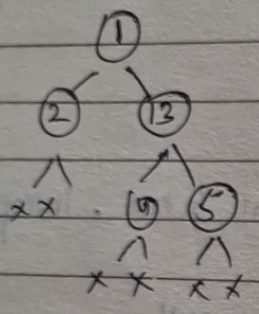
Serialize and Deserialize BT

Make String by BT and BT by String

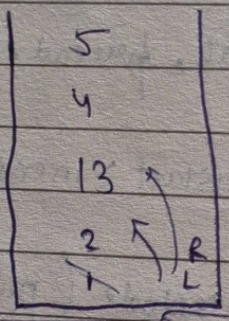
String Serialize (root)

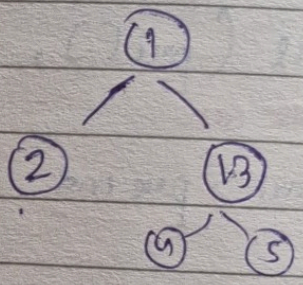
↳ By level order traversal, when null comes put "#,"

↳ Put nulls ~~and~~ in queue. (don't condition them)



String = [1,2,13,#,#,4,5,#,#,#,#]





NEW CONCEPT

Raj

DATE / /
PAGE

CODE

```
TreeNode Deserialise. (string data)
{
if (data.size() == 0) return NULL;
```

New concept

```
stringstream s(data);
string str;
getline (s, str, ',')
```

→ This will avoid traversal on ','

So eg: S = 1, 2, 3, null, 14

str = 1 2 3 null 14

∵ seperator = ','

```
TreeNode* root = new TreeNode
                    (stoi(str));
queue <TreeNode*> q ; q.push(root)
while (!q.empty()),
{
TreeNode* node = q.front(); q.pop();
getline (s, str, ',');  → give you next char (after skipping ',')
  if( str == '#' ) node->left = NULL;
  else {
    TreeNode* leftNode = new TreeNode (stoi(str));
    node->left = leftNode;
    q.push (leftnode)
  }
```

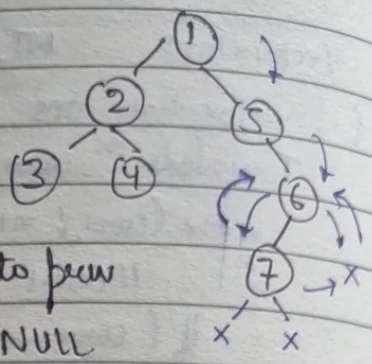
{Same for right}

```
}
return root.
```

FLATTEN Tree to linked list (By rearranging)

DATE / /
PAGE

Let go the right extreme like

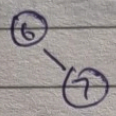


flat (node → right)
flat (node → left)

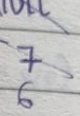
- change right to prev

Then we will shift left side to right and left to NULL

node → right = prev
node → left = NULL
prev = node

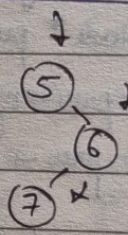


prev = NULL



prev = stores the
- last flatten Node

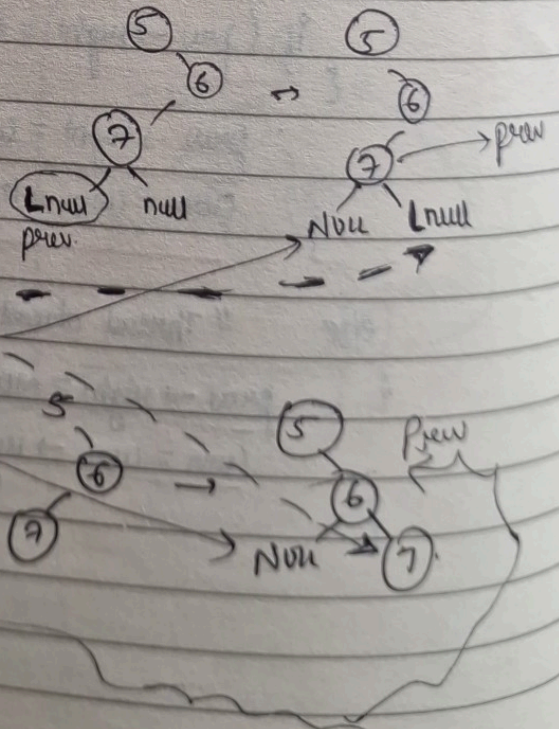
prev = NULL



prev = null

{ null is null But represent

```
Void flat (TreeNode* node)
{
    if (node == NULL) return ;

    [ flat (node → right)
      flat (node → left) ]

    { node → right = prev ;
      node → left = NULL
      prev = node ; }

} return ;
```



Construct

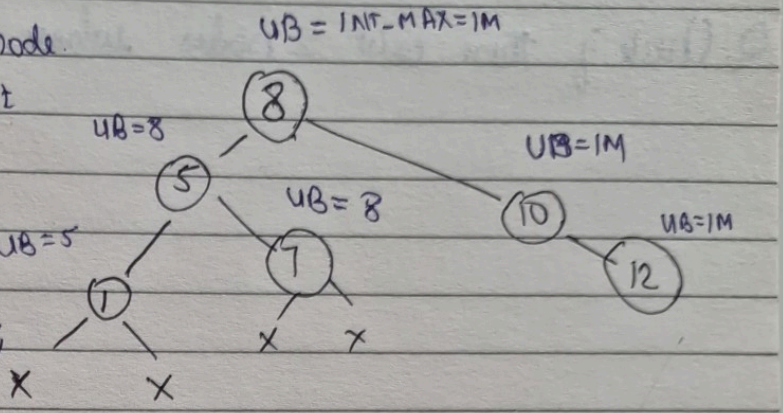
- We main a and value

```
TreeNode*
{ int i=0
  retur
}

TreeNode*
{
  if (i=
  { TreeN
    i++
  root →
  root →
  return
}
```

Construct BST from PREORDER

Raj
DATE / /
PAGE

- We main a upper bound every node and value should be less than it



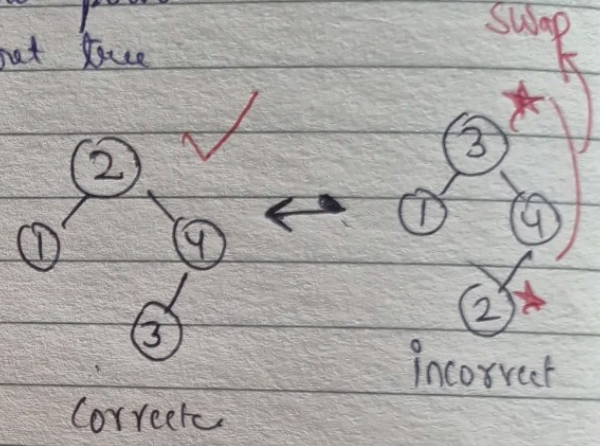

```
TreeNode* main(Vector A)
{ int i=0
  return build(A, i, INT_MAX);
}

TreeNode* build (vector<int> &A, int &index, int bound)
{
  if (i == A.size() || A[i] >= bound) return NULL;

  { TreeNode* root = new TreeNode(A[i]);
    i++;

  root->left  = build(A, i, root->val);   // UB for left subtree is its parent
  root->right = build(A, i, bound);

  return root.
}
```

Recover BST

There are 2 swapped nodes in BST, determine them and correct the BST

One violate node will be prev and other be root in traversal

- We do inorder traversal (sorted for BST)
- Use fact that previous can't be more than curr node
- Whenever (in traversal) we found out that prev > curr then we store those nodes in a node pointer
- After that we swap it and get a correct tree




```
{ TreeNode * prev ;
  TreeNode * first ;
  TreeNode* second ; }
void inorder(TreeNode* root)
{
  if (!root) return

  Inoreder (root->left)

  if( prev && prev->val > root->val)   // Violation
  {
    if (first == NULL)   // only update once in first violation
    {
      first = prev;
    }
    Second = root;   // updated in each violation

    prev = root; // update the prev for each traversal
  }
  inorder (root->right);
```

```
void recover ( TreeNode* root)
{
    inorder (root);

  // Swap the pointed nodes
  if(first && second)
  {
    Swap (first->val, second->val);
  }
```

Always check

Largest BST in Binary tree

Brute (validate each node) $O(N^2)$

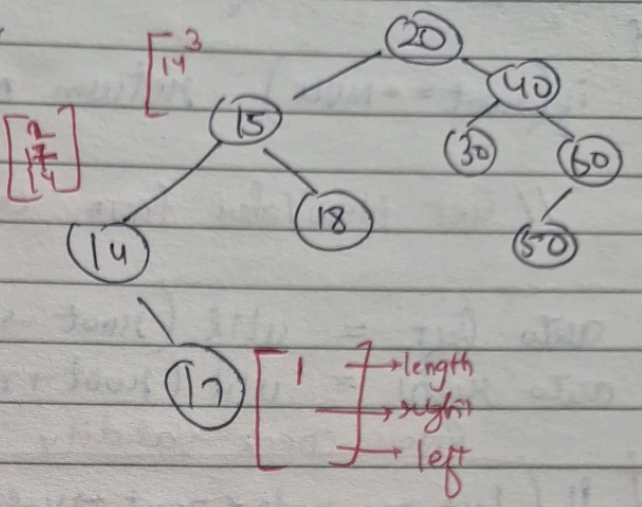
Raj

DATE / /

PAGE

Better approach :- $O(n)$

- Do POST order traversal
- Compare Left and right for each node and determine length.
- Start from last and determine the longest len which satisfy the condition
- Initialize Left, Right bounds with INT_MIN, INT_MAX.



let define new data structure

CONCEP

for a Node to Validte

Max node of leftSubtree < root

min node of rightSubtree > root

```
class NewValue {
public:
int maxnode, minnode, maxSize;
NewValue (int minnode, int maxnode, int maxSize)
{
    this → maxnode = maxnode;
    this → minnode = minode;
    this → maxSize = maxSize;
}

class Solution {
```

NewValue Util (TreeNode* root)
{
if(root == NULL) return NULL;

// Get NewValue from Left, right

auto left = util (root → left)
auto right = util (root → right)

Now check validity

If (left.maxnode < root → val && root → val < Right.minnode)

return NodeValue (min (root → val, left.minnode),
max (root → val, right.maxnode),
left.maxsize + right.maxsize + 1)

// Otherwise return maxnode = INTMax, minnode = INTMin

return (INT_MIN, INT_MAX, max (left.maxsize, right.maxsize))

max of either size

}

int largestBSTinBT (root)
{
return Util (root) .maxsize;
}

New Value formed