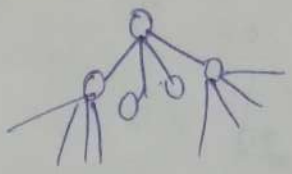
Tree

Binary Tree

```
class Node {
  int data;
  node* left
  node* right
}
```

(N-ary Tree)

```
class Node {
int data;
vector<node*>child;
}
```



Root Node parent → NULL / Root Node

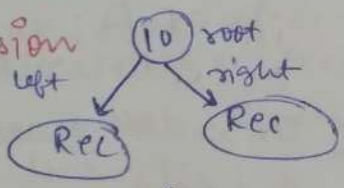
parent one, ancestor many

Traversal :- Level order, Inorder, preorder, postorder, morris traversal

creation of Tree :-

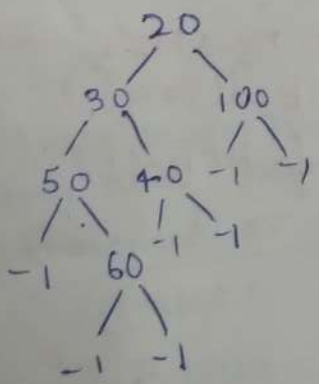
① using level order

② using recursion



1) create root
2) root → left = Rec.
3) root → right = Rec

ex :- 20 30 50 -1 60 -1 -1 40 -1 -1 100 -1 -1

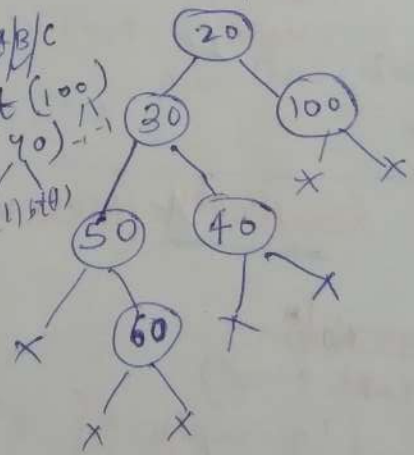


buildTree(20) → bt(100)

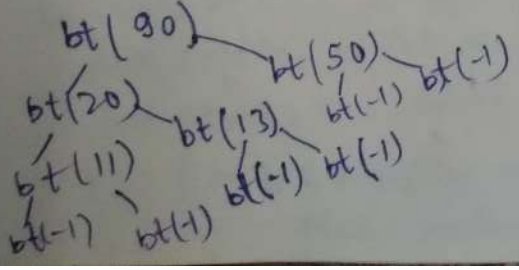
buildTree(30) → bt(40)

buildTree(50) → bt(60)

bt(-1) bt(-1) bt(-1) bt(-1)

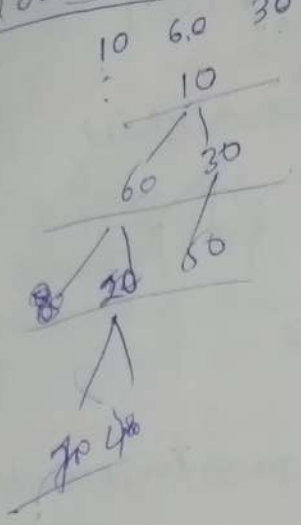


ex 90 20 11 -1 -1 13 -1 -1 50 -1 -1

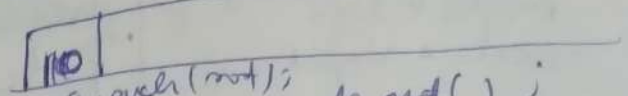


level order Traversal:-

10 60 30 80 20 50 70 40



using Queue



q.push(root);

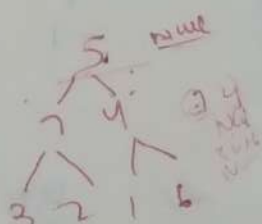
(A) temp = q.front();
(B) q.pop();
(C) Print ans;
(D) child into ans

Tree creation :-

```
class Node{
public:
    int data;
    Node * left;
    Node * right;
    Node(int data){
        this->data = data;
        left = NULL;
        right = NULL;
    }
};
```

```
void create(int data){
Node * buildTree(){
    int data;
    cout << "enter the data" << endl;
    cin >> data;
    if(data == -1)
        return NULL;
(A) Node * root = new Node(data);
(B) root->left = buildTree();
(C) root->right = buildTree();
    return root;
}
```

```
void levelOrder(Node * root)
{
    Queue<Node*> q;
    q.push(root);
while(!q.empty()){
    Node * temp = q.front();
    q.pop();
if(temp->left)
    q.push(temp->left)
if(temp->right)
    q.push(temp->right);
}}
```



print different levels :-

1 2 3 (3)

```
if (temp == NULL)
    cout<<endl;
    if (!q.empty()){
        q.push(NULL));
    }
```

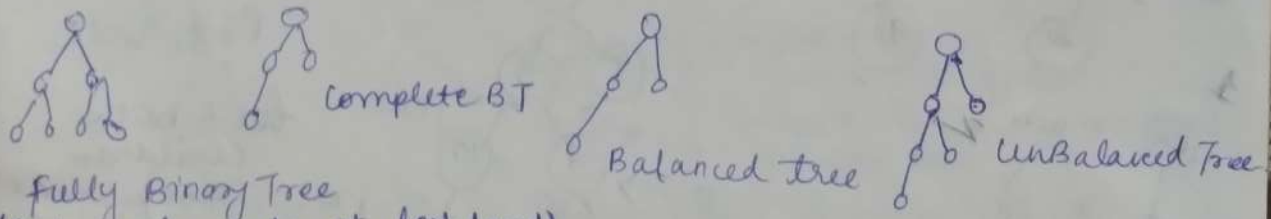
** Striver ✓
height, diameter, preorder, postorder, inorder (Recursive & Iterative)

Q- level order Traversal ☆ through create Tree
Q Iterative Preorder, Inorder, Postorder

Diameter of the tree :-

```
int op1 = dia(root->left)
int op2 = dia(root->right)
int op3 = height(root->left) + height(root->right)+1
int ans = max(op1, max(op2, op3));
```

Q Checked whether identical Tree?
Q- Check whether mirror tree?
Q Check whether a Binary tree is Balanced or not.



Fully Binary Tree
(Leaf node only at last level)

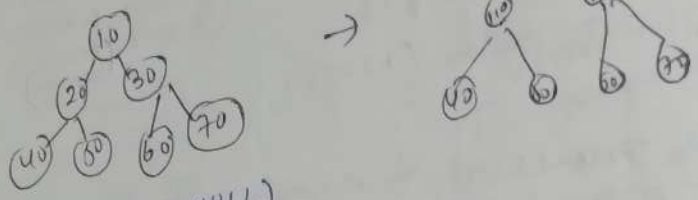
Balanced or not :-

```
if(root == NULL) return true;
int leftH = height(root->left)
int rightH = height(root->right)
int diff = abs(leftH - rightH)
bool ans1 = diff <= 1
bool leftAns = isBalanc(root->left)
bool rightAns = isBalanc(root->right)
if (ans1 && rightAns && leftAns) return true;
    else return false;
```

4

Check whether BT is sum tree or not

convert or BT to sum tree



left subtree sum
+ right subtree sum
+ current node.

```
if (root == NULL)
    return 0;
int leftA = fun(root -> left);
int rightA = fun(root -> right);
root -> data = root -> data + leftA + rightA
return root -> data;
```

lowest common Ancestor :-

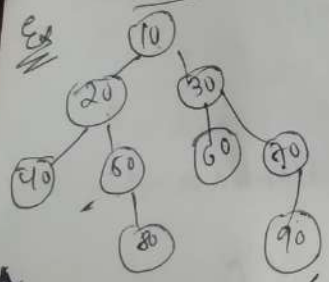
50 → 20 → 10

90 → 70 → 30 → 10

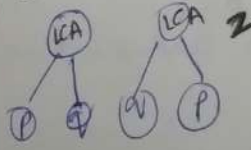
P = 50

Q = 90

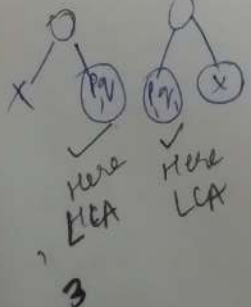
Ex



1



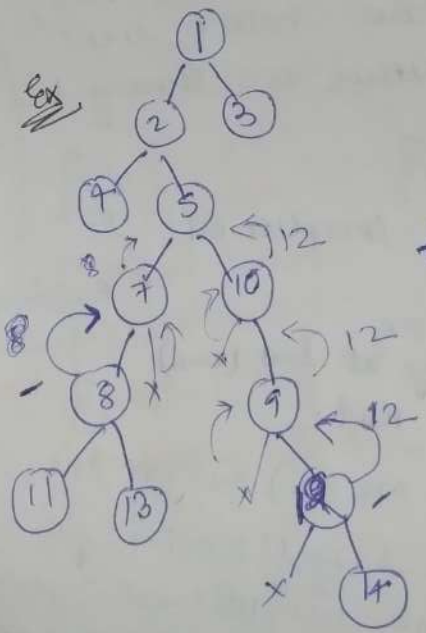
2



3

4

Ex



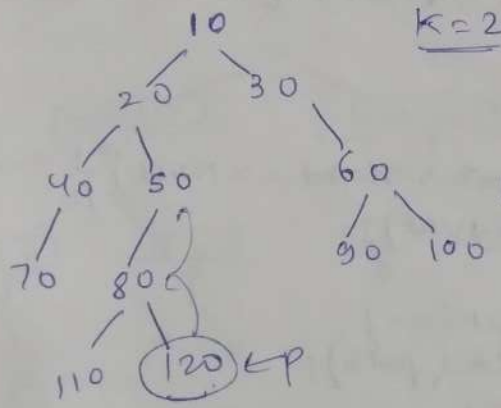
P = 8, Q = 12

for 5 both children are present

∴ LCA = 5.

```
if (root == NULL)
    return NULL;
if (root->val == p->val)
    return p;
if (root->val == q->val)
    return q;
Node* LeftAns = fun(root->left, P, Q)
Node* RightAns = fun(root->right, P, Q)

if (leftAns == NULL && RightAns == NULL)
    return NULL;
else if (leftAns != NULL && RightAns == NULL)
    return leftAns;
else if (rightAns != NULL && LeftAns == NUL)
    return rightAns.
else
    return root;
```

K^{th} Ancestor :-



K=2, Ans=50, do K-- when find P. if K==0 return ans.

B.C

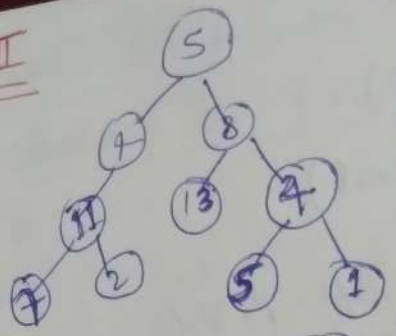
```
if (root == NULL)
    return false;
if (root->data == p->data)
    return true;
bool left = fun(root->left, K, P);
bool right = fun(root->right, K, P);
if (left == true || right == true)
    K--;
if (K == 0) return p->data
if (K == 0) { K = -1; return p->data; }
return left || Ans;
```

OR

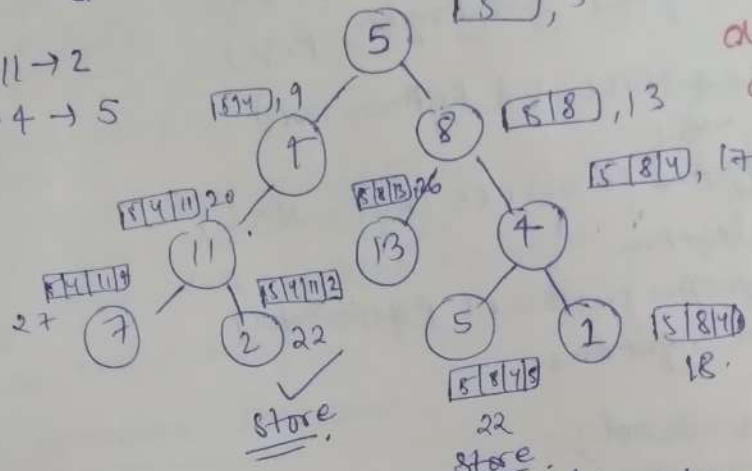
Start traversing from root and store in vector if p not found then pop_back() (when returning) else push_back. After get p pop K elements & last one will be ans.

path sum - II

Target sum = 22
root to leaf only



5 → 4 → 11 → 2
5 → 8 → 4 → 5

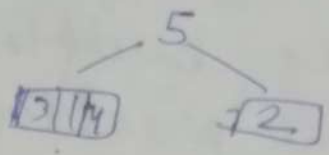


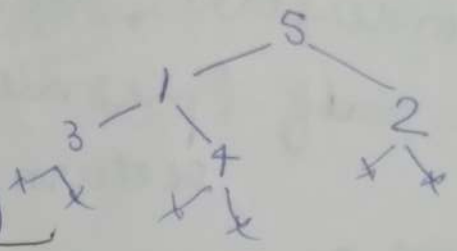
Follow :-
start from anywhere and go anywhere & find target sum.

```
void solve(Node* root, int targetSum, int &currSum, vector<int> &path
           vector<vector<int>> &ans) {
    if (root == NULL)
        return;
    // leaf node
    if (root->left == NULL && root->right == NULL) {
        path.push_back(root->val);
        currSum += root->val;
        if (currSum == targetSum)
            ans.push_back(path);
    }
    path.pop_back(root->val);
        currSum -= root->val;
        return;
    }
    path.push_back(root->val);
    currSum += root->val;
    solve(root->left, targetSum, currSum, path, ans);
    solve(root->right, " , " , " , " , " );
    path.pop_back();
    currSum -= root->val;
```

Q Given :- inorder (LNR) 3 1 4 5 2

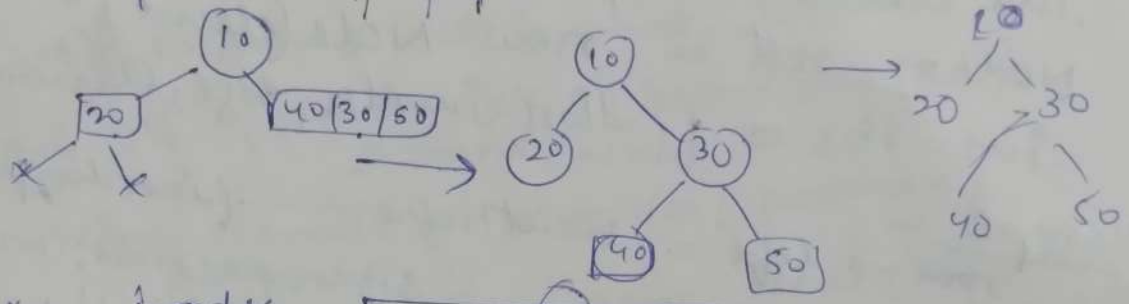
preorder (NLR) 5 1 3 4 2





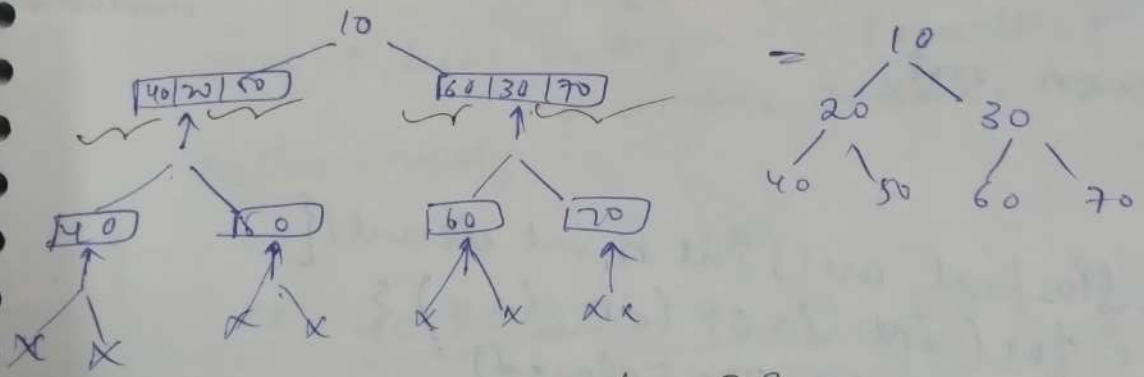
eg :- inorder 20 10 40 30 50

preorder 10 20 30 40 50



eg :- inorder 40 20 50 10 60 30 70

preorder 10 20 40 50 30 60 70



Why not single preorder ??

~~Node * buildTree(int preorder[], int inorder[], int size);~~

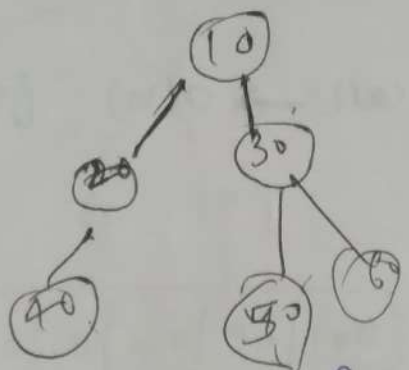
~~int findIn(int arr[], int k, int size);~~

findInInorder (int arr[], int k, int size) {

```
Node* buildTreefromPreOrderInorder(int inorder[]
, int preorder[], int size, int &preIndex, int
inorderstart, int inorderEnd) {
    B.C.  if (preIndex >= size || inorderstart >
                                    inorderEnd)
            return NULL;

    // Step A:-
    int element = preorder[preIndex++];
    Node* root = new Node(elemt);
    int Pos = findPos(inorder, size, element);

    // Step B:-
    root -> left = buildTreefrom---(inorder, preorder
                    size, preIndex, inorderstart
                    pos-1);
    root -> right = buildTreefrom---(inorder, preorder,
                    size, preIndex, pos+1,
                                    inorderEnd

    return root;
    }

int findPos(int arr[], int n, int elemt){
    for(int i=0; i<n; i++){
        if (arr[i] == elemed)
            return i
    } return -1
```

H.W :- use hashmap for searching the elemt in inorder array.

Create a tree using Inorder & Postorder.



inorder :- 40 20 10 50 30 60

Postorder :- 40 20 50 60 30 10

L R N

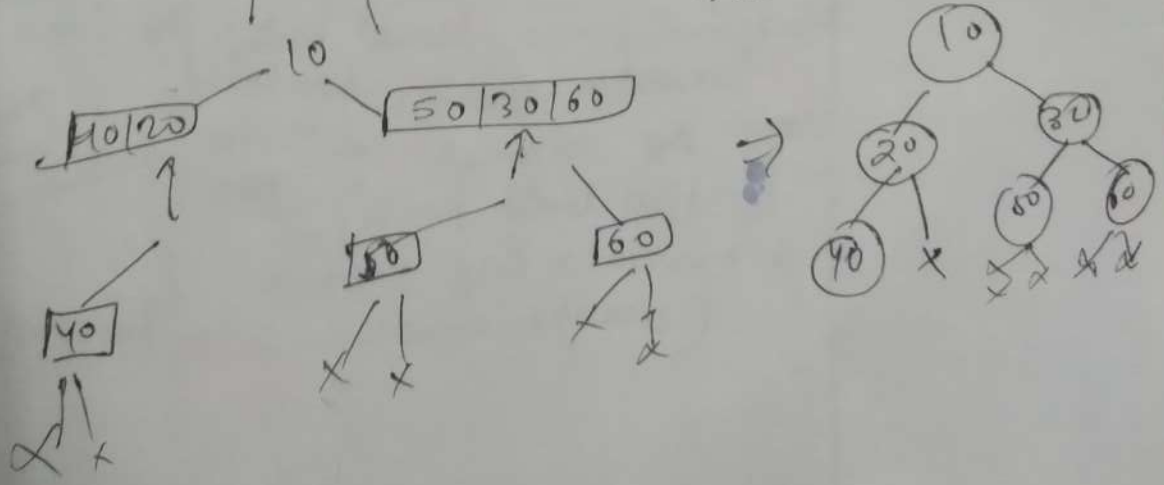
10 ↑ root

```
Node * buildtreefromPostorder (int inorder, int postorder,
int &postIndex, int instart, int inEnd, int size) {
B.C  if (postIndex < 0 || instart > inEnd)
        return NULL.
     int element = postorder[postIndex];
     postIndex--;
     Node * root = new Node (elemt);
     int pos = findPos( inorder, size, elemt)
     root → right = buildtreefromPostes ( inorder, postorder,
                    postIndex, pos+1, inEnd, size)
     root → left = buildfro - - - ( inorder, postorder,
                    postIndex, Intindex, pos-1, size)
     return root;
}
```

L R N

ex ≡ 40 20 10 50 30 60 → inorder

40 20 50 60 30 10 → Postorder L R N



Q- PreOrder & PostOrder → Creation of Tree

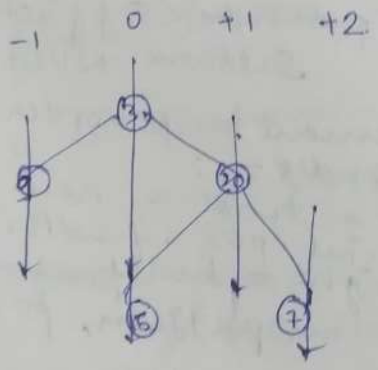
10

Through Map ; instead of function call → O(n) for pop.

use map → O(1)

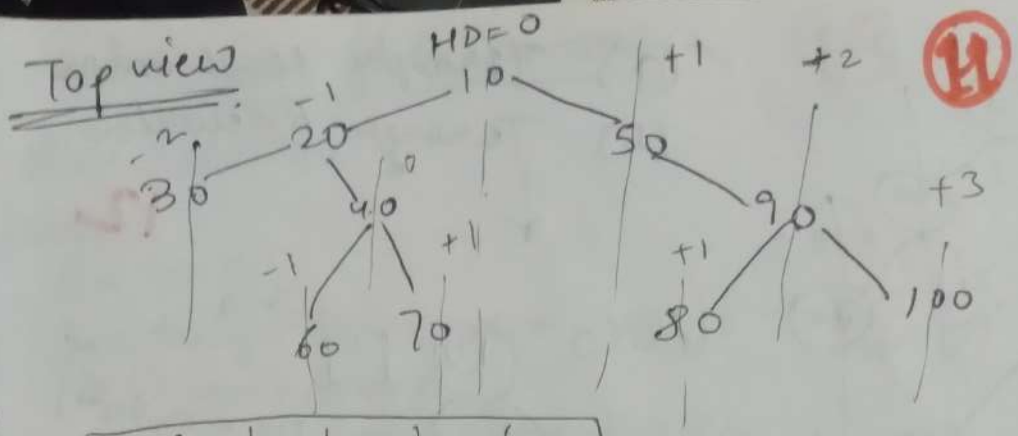
store map [~~struct~~ Inorder[i]] = i

Vertical order :-



9 3 15 20 7

Top view



30	20	10	50	90	100

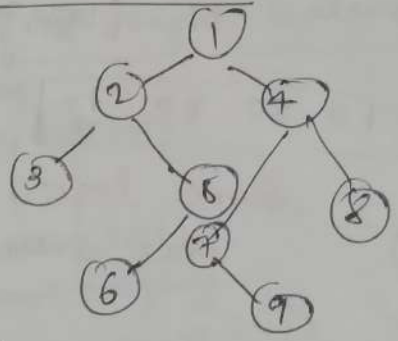
Horizontal Distance
= 0
left → HD-1
right → HD+1

level order traversal & store key-value pair only for newly ones.

* map <int, int> store horizontal distance with values.
NO updation in map.

T.C of all containers in stl & about them like map stores in sorted order.

Bottom view



⇒ 3 6 7 9 8

Similar as Top views but updation in map is allowed.

Some level, same horizontal distance, store element acc. to question or bigger one. (Store level wise & take first element & leave others)

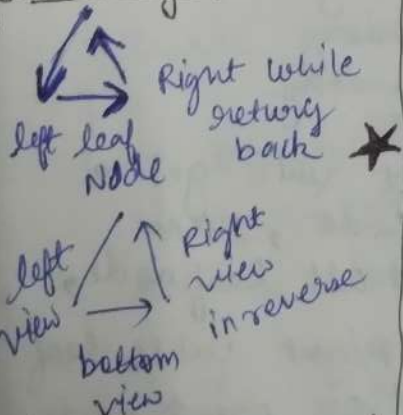
Print all leaf nodes

Inorder traversal ~~में leaf node~~ ~~पे~~ करो, leaf node ~~मिले~~ तो print करो।

zig-zag :-

Through level order ~~one~~ level. then push right then left
odd level then push left then right.
Also through Stack

Boundary Traversal

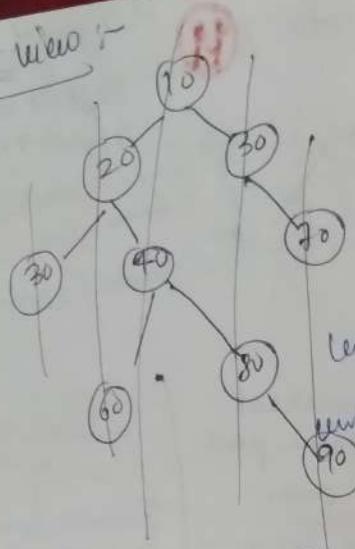


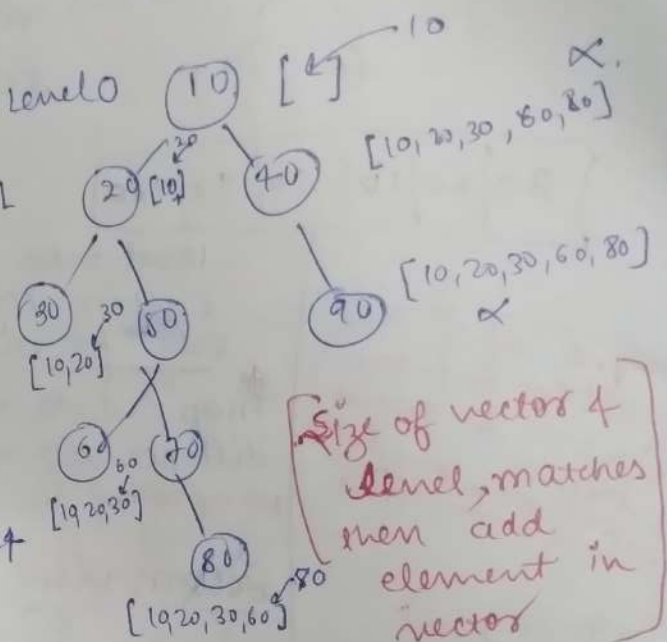
Diagonal traversal

12

(a) Through Level order
(b) Through Recursion

Left view :-

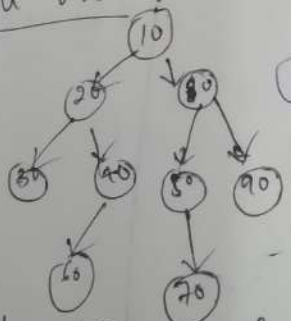




10 20 30 60 80

Size of vector & level, matches then add element in vector

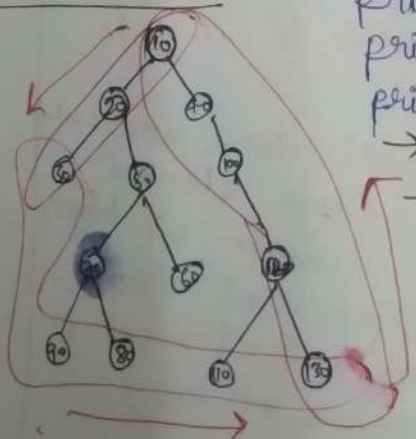
Right view :-



10 80 90 70

Same logic, only right call first then left call.

Boundary Traversal :-

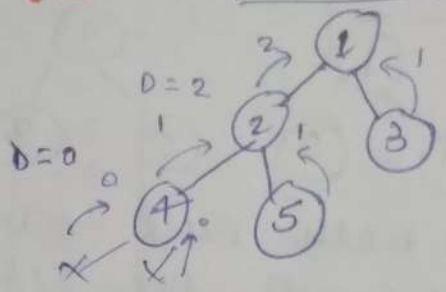


printLeftBoundary
printLeafBoundary
printRightBoundary
→ Print root
→ printleft until get a leaf node, then print all leaf node, move right until leaf node then print node while returning.

fast way to find diameter :-

13

D=3, h=2+1=3



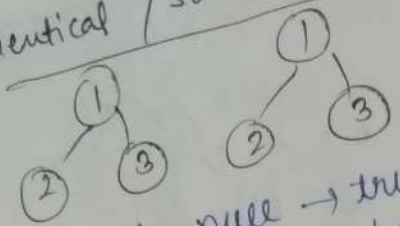

```
class Solution {
public:
  int D=0;
  int height (TreeNode* root){
   if(!root) return 0;
    int lh= height(root->left);
    int rh= height(root->right);
    int curD = lh+rh;
      D= max(D, curD);
    return max(lh+rh)+1 ;
  }
int diameter (TreeNode* root) {
    height(root);
    return D;
}
```

Fast way to find right balanced Tree:-

Follow up: How to convert into Balanced Tree?

```
bool isbalanced= true;
int height (TreeNode* root){
  if(!root) return 0;
 int lh= height(root->left);
  int rh= height(root->right);
  if(isbalanced && abs(lh-rh)>1)
      isbalanced = false;
return max(lh,rh)+1;
}
bool Checkbalance(TreeNode* root){
   height(root);
   return isbalanced;
}
```

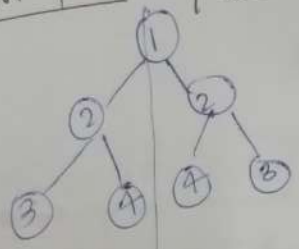
Identical / same tree :-



(a) both null → true
(b) Not null then check value && left of both && right of both & return

Mirror Tree :- / symmetric tree

(c) return false

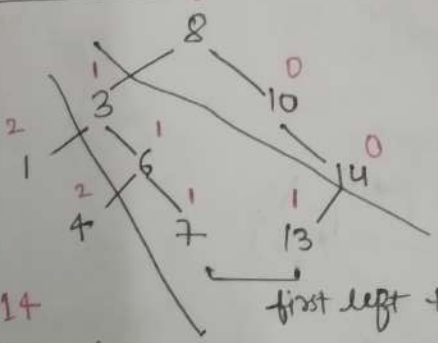


same as identical, but one's left → 2nd right

```
isMirror(root1->left, root2->right);
&& isMirror(root1->right, root2->left);
```

Diagonal traversal :-

(I)



going left +1, going right do nothing

d:0 → 8 10 14
d:1 → 3 6 7 13
d:2 → 1 4

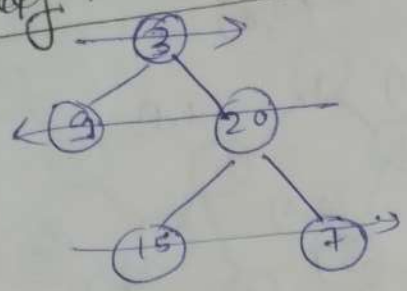
d≠0, use map<int, vector<int>>mp; so that it can be stored in sorted order acc. to d.

⇒ Then print the values of map.

(II)

```
vector<int> ans;
if(!root) return ans;
queue<Node*> q;
q.push(root);
while(!q.empty()){
  Node *temp = q.front(); q.pop();
  while(temp){
    ans.push_back(temp->data);
    if(temp->left){
      q.push(temp->left);
    }
    temp = temp->right;
  }
}
return ans;
```

zig-zag level order Traversal :-



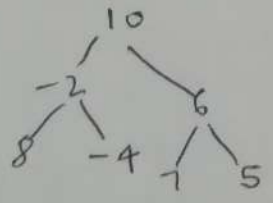
[3, 20, 9, 15, 7]

7
15
20
9
3

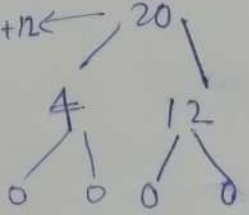
flag = left to right

Transform to sum tree :-

each node is sum of LC & RC
& leaf nodes = 0

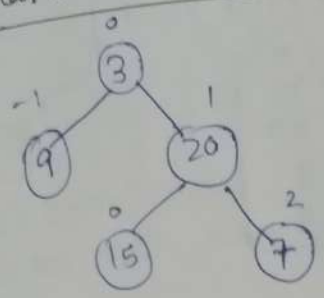


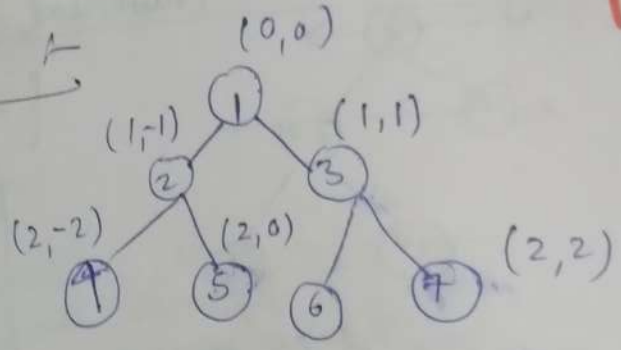
4 - 2 + 6 + 12 ←



16

Vertical order Traversal



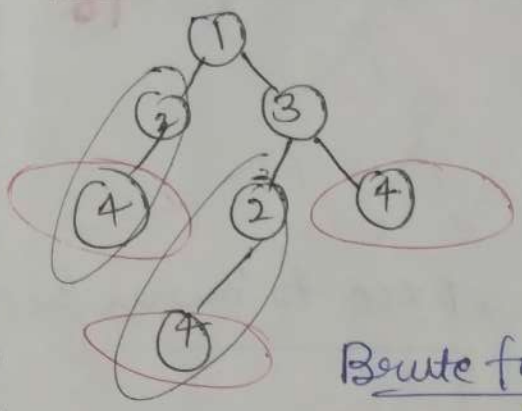


map <col, map <row, multiset > > mp;

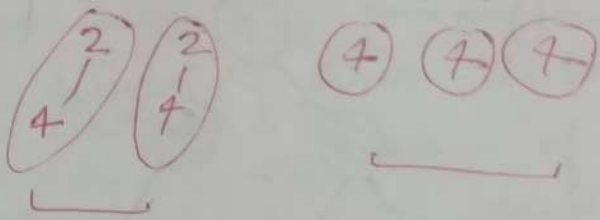
Sorted

Duplicate also

Find Duplicate Subtrees :-



01 02 03 04 05



Ans:- | 01 | 03 |

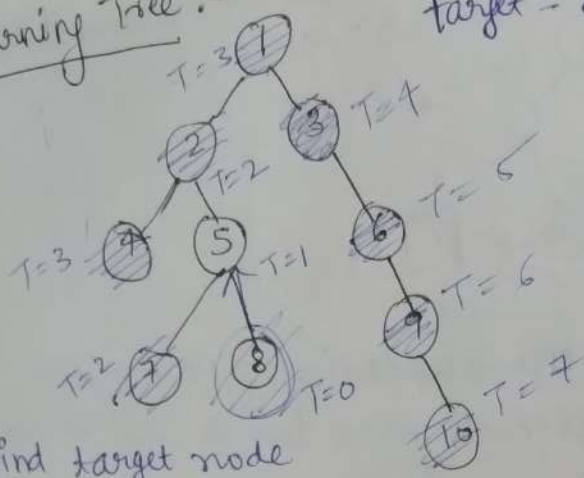
Brute force:- isIdentical (Node *a , Node *b)

⇒ for each node traverse all nodes of tree.

⇒ $O(n^2)$

Burning Tree :-

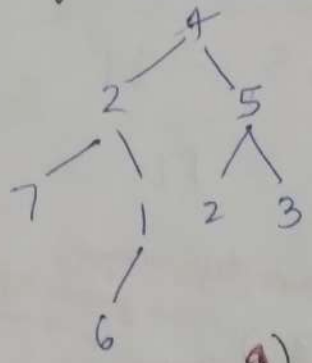
target = 8



7 sec to burn full

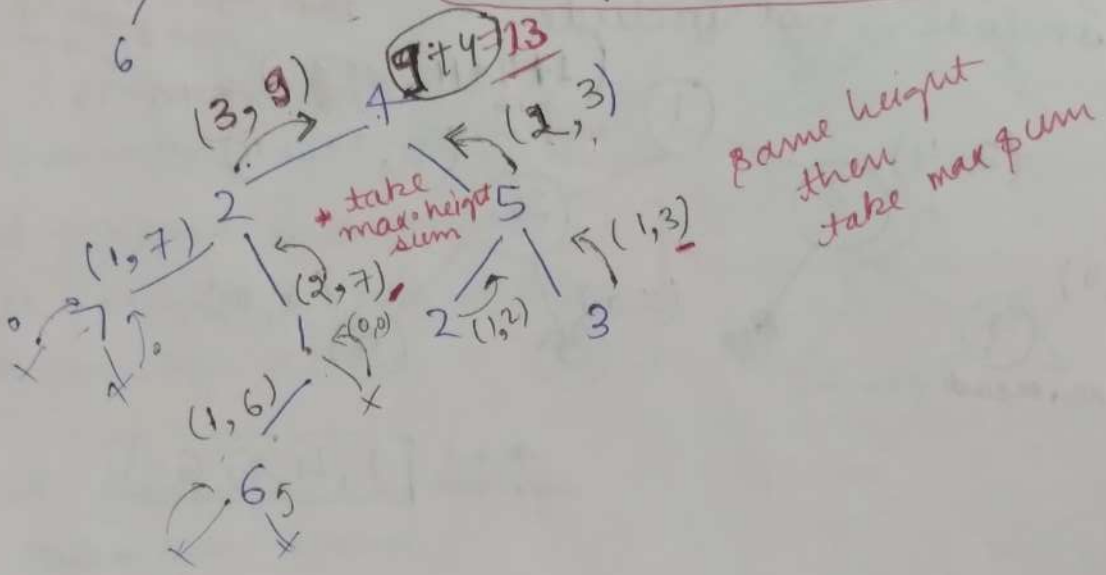
(a) Find target node
(b) make node to parent node mapping

Sum of the longest bloodline of a tree. 19



⇒ height of tree & take sum of all nodes

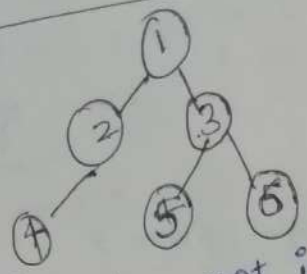
Pair < height, sum >



Maximum sum of Non-adjacent Nodes :— 20

1, 1, 4, 5 = 11

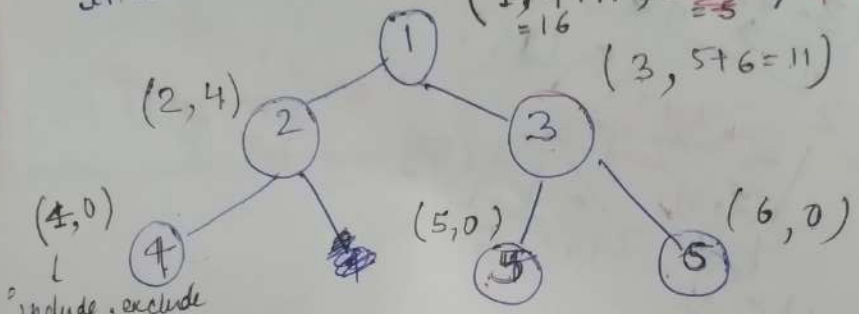
2, 4, 5 = 11



include , not include

~~max(left, right)~~ max from left + max from right

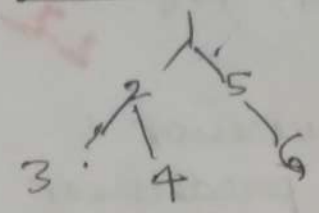
(1+4+11 = 16, ~~2+3+5 = 5~~) 4 + 11 = 15



Ans 1, 4, 5, 6

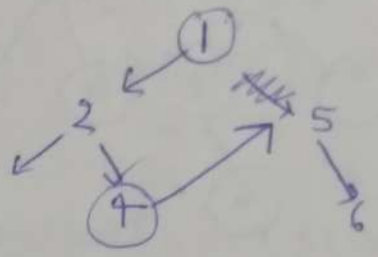
Flatten Binary tree to linked list

2)



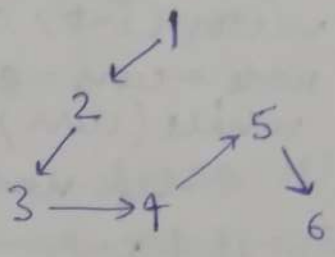
NLR

→ 1 2 3 4 5 6 (Preorder)



predecessor → right = curr → right

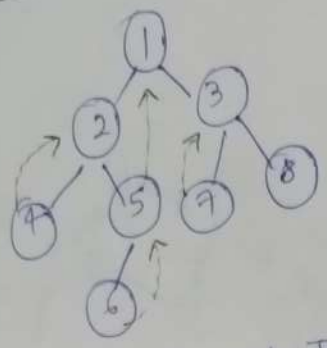
```
if (curr → left)
Pred → right = curr → right
Curr → right = curr → left
Curr → left = NULL
```




```
void flatten (Node* root){
    Node* curr = root;
    while (curr){
        if (curr → left){
            Node* pred = curr → left;
            while (pred → right){
                pred = pred → right;
            }
            pred → right = curr → right;
            curr → right = curr → left;
            curr → left = NULL;
        }
        curr = curr → right;
    }
}
```

Morris Traversal

Same as inorder but in O(1) S.C , O(N) T.C



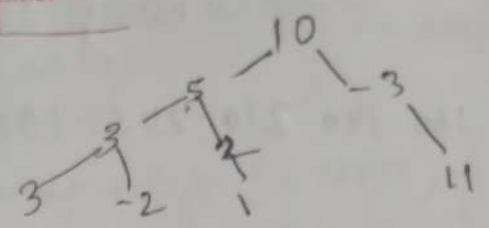
→ concept of successor & predecessor

→ link Establishment through right pointer.

```
vector<int> inorderTraversal(Node* root) {
    vector<int> ans;
    Node* curr = root;
    while(curr) {
        // left node is null, then visit it & go right
        if(curr->left == NULL) {
            ans.push_back(curr->value)
            curr = curr->right;
        }
        else {
            // find inorder predecessor.
            Node* pred = curr->left;
            while(pred->right != curr && pred->right) {
                pred = pred->right;
            }
            if(pred->right == NULL) {
                pred->right = curr;
                curr = curr->left;
            }
            // left is already visited, go right after visiting curr node, while removing the link
            else {
                pred->right = null;
                ans.push_back(curr->val);
                curr = curr.right;
            }
        }
    }
```

// if right node null then go left after establishing link from pred to curr

K-Sum path :-



Target = 8

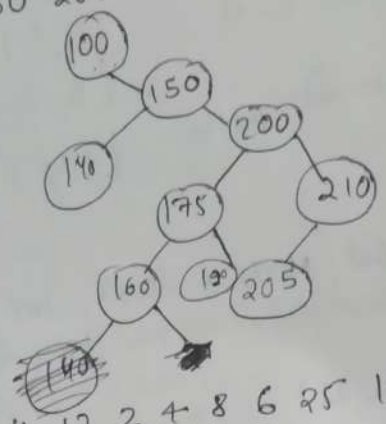
[5, 3]
[5, 2, 1]
[11, 3]

Break it down :- root से path

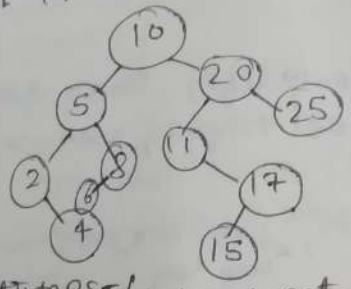
```
{ long long
  int ans = 0;
  void pathFromOneRoot(Node * root, int &sum) {
                                    long long
    if(!root) return;
    if(sum == root->val)
      ++ans;
    pathFromOneRoot(root->left, sum - root->val);
    pathFromOneRoot(root->right, sum - root->val);
  }
  int pathSum(Node * root, int targetSum) {
                           long long
    if(root) {
      pathFromOneRoot(root, targetSum);
      pathSum(root->left, targetSum);
      pathSum(root->right, targetSum);
    }
    return ans;
  }
};
```

Binary Search Tree

i/p 100 150 200 175 160 140 210 205 190



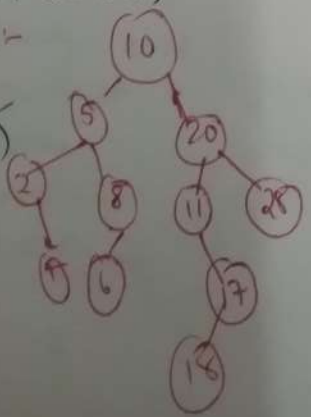
10 20 5 11 17 2 4 8 6 25 15




```
Node * insertIntoBST(Node* root, int data){
  if(root==NULL){
     root= new Node(data);
     return root;
  }
  if (root→data>data){
     root→left = insertIntoBST(root→left, data);
  }
  else
     { root→right = insertIntoBST (root→right, data);
  }
  return root;
```

Searching :-

avg. case
O(H) = O(log N)




```
Bool findNodeinBST (Node *root, int target)
  if (root==NULL){
     return false;
  }
  if(root→data == target)
     return true;
  if (target > root→data)
     return findNodeinBST(root→right, target)
  else
     return findNodeinBST(root→left, target)
}
```

```
int minVal (Node* root){
if (temp == NULL)
    return -1;
while (temp->left != NULL) {
    temp = temp->left;
}
return temp->data;
}
int maxVal (Node* root) {
Node* temp = root;
if (temp == NULL)
  return -1;
while (temp->right != NULL) {
  temp = temp->right;
}
return temp->data;
}
```

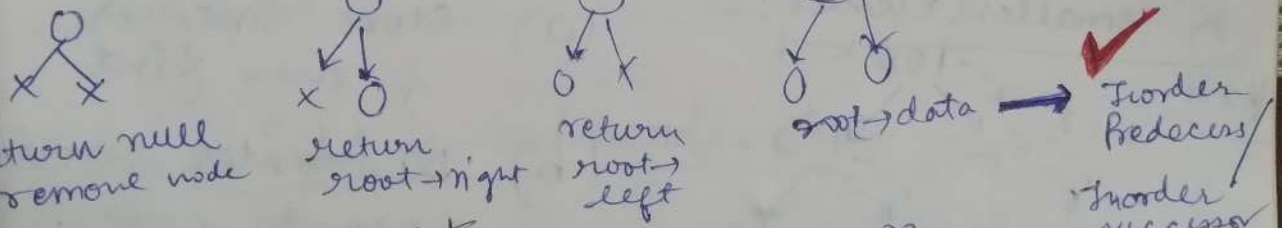
Inorder of BST is sorted.

Inorder Predecessor & Successor

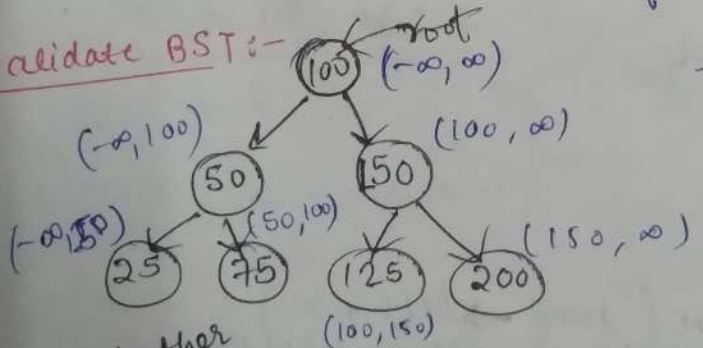
- find inorder then find predecessor & successor.

leftTree ka max → ← right subtree min

Deletion in BST :-



Validate BST :-

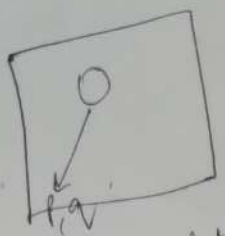


$-2^{32} \rightarrow 2^{32}$

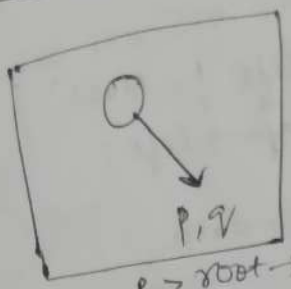
Check whether valid BST or not

LCA → lowest common Ancestor

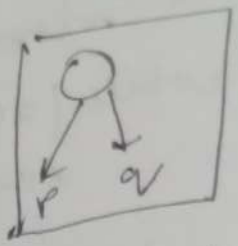
log(n) 26



p < root → data
q < root → data



p > root → data
q > root → data

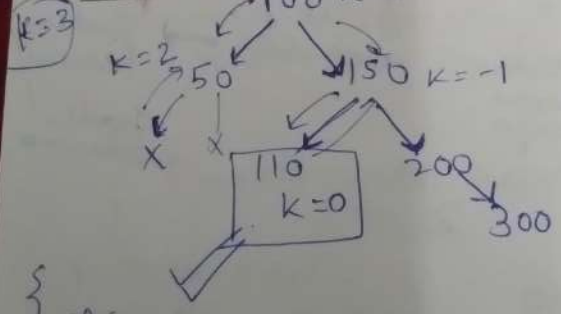


p < root → data
q > root → data

q < root → data
p > root → data

```
{ if (root == NULL)
    return NULL;
  if (p → data < root → data && q → data < root → data) {
    return LCA(root → left, p, q);
  if (p → data > root → data && q → data > root → data) {
    return LCA(root → right, p, q);
  return root;
}
```

K^th smallest element :-

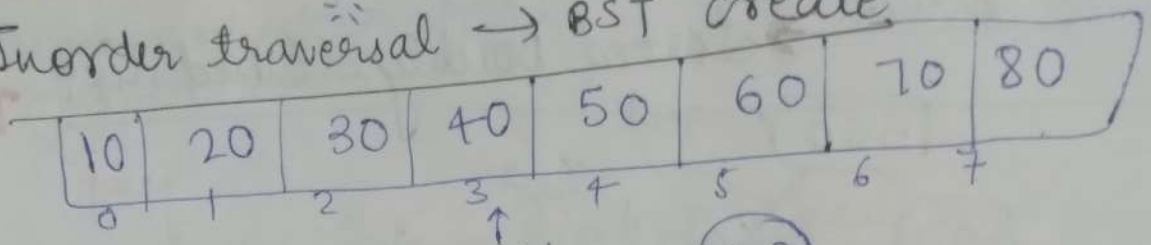


(a) Store inorder & then find.

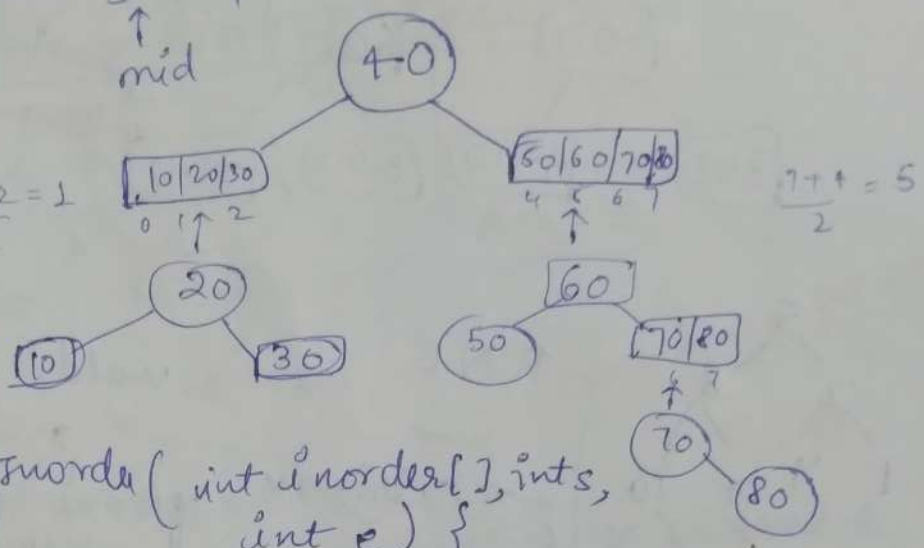
(b) DO inorder & reduce k to zero.

```
{
  if(root == NULL)
    return -1;
  int leftAns = solve(root → left, k);
  k--;
  if(leftAns != -1) return leftAns;
  if (k == 0) { return root → data; }
  int rightAns = solve(root → right, k);
  return rightAns;
}
```

Inorder traversal → BST create



$\frac{0+7}{2} = 3$ mid



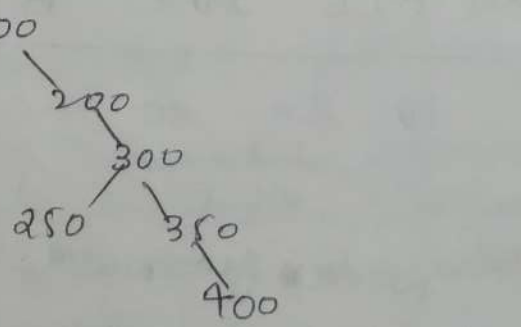
O(n)

```
Node * bstUsingInorder( int inorder[], int s,
                        int e) {
if (s > e)
    return NULL;
int mid = (s+e)/2;
int element = inorder[mid];
Node* root = new Node(element);
root->left = bstUsingInorder(inorder, s, mid-1);
root->right = bstUsingInorder(inorder, mid+1, e);
return root;
}
```

N.W Convert into a balanced BST :-

2-Sum target :-

(a) Take one node & search (sum - node->val) in tree



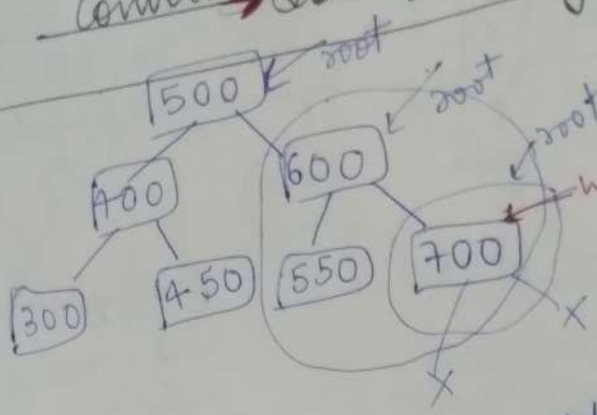
O(n log n)

(b) Store inorder, then 2-point approch of start & end, if sum (start & end value) > req. then e--; else s++;

O(n) + O(n) = O(n)

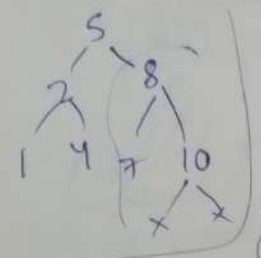
28

BST convert → Sorted Doubly linked list

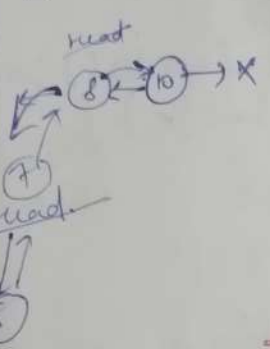


head
600 ⟷ 700

T.C O(N) else O(H)
S.C O(N) Skew Tree

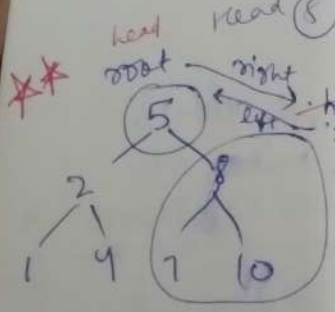


RNL




```
void func (Node * root,
           Node * head)
{
if(root == NULL)
    return ;
func (root->right, head);
root->right = head
If (head != NULL)
   head -> left = root;
head = root ;
func (root -> left , head);
}
```

**



5 ⟷ 8 ⟷ 10

assume
left = prev
right = next

reverse inorder & move head.

Sorted DLL to BST :—

10 20 30 40 50 60 70 80

$n/2 - 1$ | 1 | $n/2$

inplace

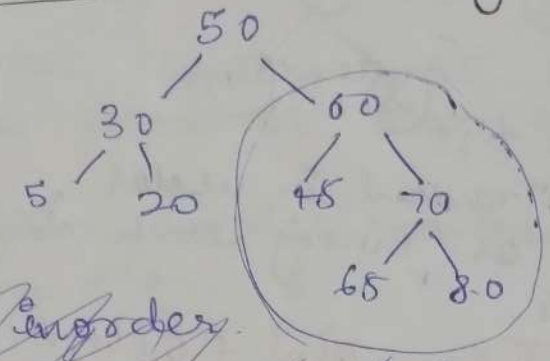
```
Node * func (Node *& head, int n) {
if (n<=0 || head = null)
    return null;
Node * leftSubtree = func (head, n/2);
Node * root = head;
root -> left = leftSubtree;
head = head -> right;
}
```

T.C = O(N)
S.C = O(H)

**

```
root->right = fun(head, n-1-n/2);
return root;
}
```

Largest BST in a Binary Tree :-



Ans = 5

Bottom up pattern

T.C O(N)

(a) ~~Find inorder~~

5 30 20 50 45 60 65 70 80

~~They select all subsets that are sorted & find max one.~~

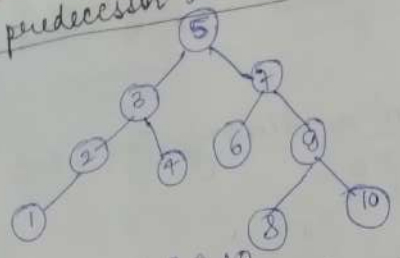
maybe correct

(b) [size, maxV, minV, validBST] → NodeClass

```
NodeD* fun(Node* root, int &ans){
if(root == null){
    NodeD* temp = new NodeData(0, INT_MIN, INT_MAX, true);
    return temp;
}
NodeD* leftAns = fun(root->left, ans);
NodeD* rightAns = fun(root->right, ans);
                                        LRN
NodeD* currNodeAns;
currNodeAns->size = leftAns->size + rightAns->size + 1;
currNodeAns->maxV = max(root->data, rightAns->maxV);
currNodeAns->minV = min(root->data, leftAn->minV);
if(leftAns->validBST && rightAns->validBST && (root->data > leftAns->maxV
    && root->data < rightAns->minV)){
    currNodeAns->validBST = true;
}
else{ currNodeAns->validBST = false; }
if(currNodeAns->validBST){ ans = max(ans, currNodeAns->size); }
return currNodeAns;
}
```

Inorder predecessor in BST

30



1 2 3 4 5 6 7 8 9 10

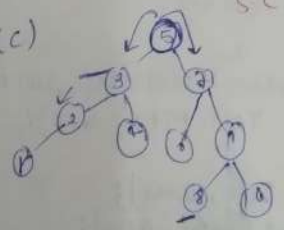
(a) → Store inorder Traversal in vector ले लो x को
→ Linear Search or Binary Search just left node.

T.C = O(n) + O(log n)
S.C = O(n)

(b) Perform inorder Traversal using Recursion & find value just less than x.

T.C = O(n)
S.C = O(n) | O(1) Morriss Traversal

(c)



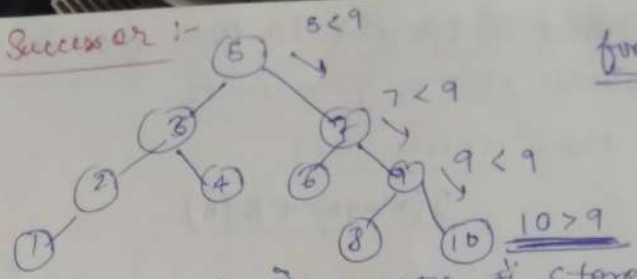
for (3) → 2
for (8) → 7

T.C = O(H)
S.C = O(1)

```
Node* pred = 0;
Node* curr = root;
while (curr) {
    if (curr->data < p->data) {
        pred = curr
        curr = curr->right;
    }
    else {
        curr = curr->left;
    }
}
return pred;
```

Inorder Successor :-

for 9 31



(a) Inorder Traversal को vector में store, then linear search or Binary search. → first value greater than x.

(b) Perform Inorder Traversal and find first value greater than 6.

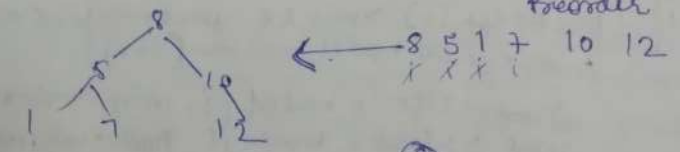
T.C = O(n)
S.C = O(n) | O(1) Morris Traversal
or recursion

(c)

```
{ Node * succ = 0
  Node * curr = 0
  while (curr) {
    if (curr -> data > x -> data) {
        succ = curr;
        curr = curr -> left;
    }
    else {
        curr = curr -> right;
  } }
  return succ;
}
```

T.C => O(H)
S.C => O(1)

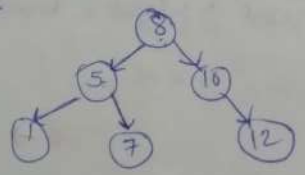
Build BST using Preorder Traversal :-



(a)

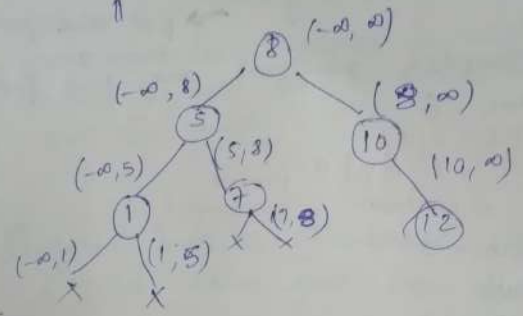
for every node we are traversing from root.

$O(n^2)$



(b) preorder :- 8 5 1 7 10 12 (sort it)
inorder :- 1 5 7 8 10 12
then make BST.
$O(n \log n) + O(n)$

(c) 8 5 1 7 10 12
↑



T.C → O(n)

func preorderBST

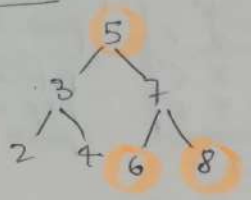
```
Int min = INT_MIN
int max = INT_MAX
int i=0;
return build (i, min, max, preorder);
```

```
build( int& i, int min, int max, vector<int>& preorder ) {
    if (i >= preorder.size())
        return NULL;
    Node * root = null
    if (preorder[i] > min && preorder[i] < max) {
        root = new Node(preorder[i++]);
        root → left = build (i, min, root→val, preorder);
        root → right = build (i, root→val, max, preorder);
    }
    return root;
}
```

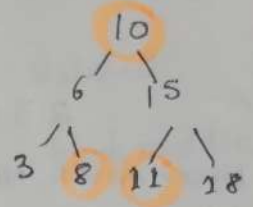
T.C → O(n)

Brothers from diff. roots / Count of pairs from two BST whose sum is equal to given x.

BST 1



BST 2



$x = 16$

(a, b) sum ⇒ x.

(5, 11), (6, 10), (8, 8)

(a) for each node in BST 1, search (sum − a) from BST2. [O(n²)]

(b) Store BST2 in map then search for (sum − a).

(c) BST1 → 2 3 4 5 6 7 8

BST2 → (reverse inorder) → descending order.

RNL

18 15 11 10 8 6 3

take inorder and start pointer from Back.

BST1 2 3 4 5 6 7 8 ×

BST2 18 15 11 10 8 6 3

$x = 16$

(2+18) = 20

(15+2) = 17

(11+2) = 13

(11+3) = 14

(11+4) = 15

(11+5) = 16 ✓

(10+6) = 16 ✓

(8+7) = 15

(8+8) = 16 ✓

if sum < x

BST1 pointer moves to its successor

else if (sum > x)

BST2 pointer moves to its predecessor.

else

BST1 ptr++

BST2 ptr++

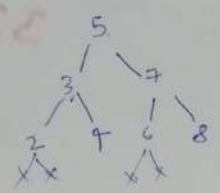
⇒ any pointer out of bound then Stop.

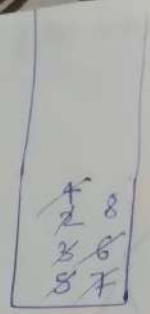
T.C → O(n), S.C → O(n) + O(n)

(d) Do not use space, :-

∴ Inorder through stack.

Inorder :-





push root then push left.

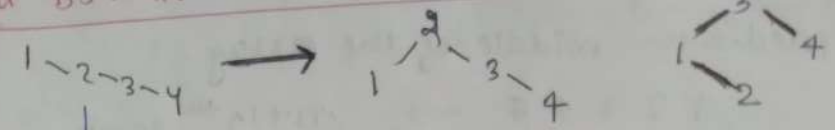
pop left
pop root
push right

2 3 4 5 6 7 8

L N R
R N L

```
int countPairs(Node* root1, Node* root2, int x) {
    int ans = 0;
    stack<Node*> s1, s2;
    Node* a = root1;
    Node* b = root2;
    while(1) {
        while(a) {
            s1.push(a);
            a = a->left;
        }
        while(b) {
            s2.push(b);
            b = b->right;
        }
        if (s1.empty() || s2.empty()){
            break;
        }
        auto atop = s1.top();
        auto btop = s2.top();
        int sum = atop->data + btop->data
        if (sum == x) {
            ++ans; s1.pop(); s2.pop();
            a = atop->right,
            b = btop->left,
        }
        else if (sum < x) {
            s1.pop();
            a = atop->right;
        }
        else {
            s2.pop();
            b = btop->left;} } return ans; };
```

Convert BST to a Balanced BST :-



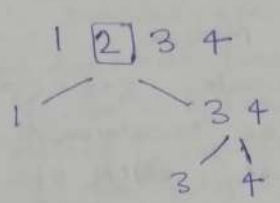
↓

Inorder Traversal

↓

BST should be balanced.

BT to BST??
Postorder → BST
???



find middle, then make it node.

★ Tree with Binary Search

```
~~Tree~~Node* balanceBST(Node* root) {
    vector<int> in;
    inorder(root, in)
    return buildTree(in, 0, in.size()-1);
}

Node* buildTree(vector<int>& in, int start, int end) {
    if (start > end)
        return null;
    int mid = (start + end)/2
    Node* ~~root~~ = new Node(in[mid]);
    root->left = buildTree(in, start, mid-1);
    root->right = buildTree(in, mid+1, end);
    return root;
}

void inorder(Node* root, vector<int>& in){
    if (!root) return;
    inorder(root->left, in);
    in.push_back(root->val);
    inorder(root->right, in);
}
```

36

Find Median of BST :-

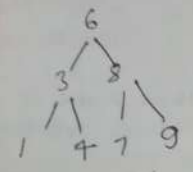
Median :- middle of the array.

1 2 3 4 5 → $\left(\frac{n+1}{2}\right)^{th}$ term = 3

1 2 3 4 5 6 → $\frac{\frac{n^{th}}{2} + \left(\frac{n}{2}+1\right)^{th}}{2}$ $= \frac{\frac{6}{2} + \left(\frac{6}{2}+1\right)^{th}}{2}$

$= \frac{3^{th} + 4^{th}}{2}$

$= \frac{3+4}{2}$ ⇐ 3.5



(a) Inorder Traversal then apply Median formula.

To find inorder → (a) use recursion O(N), O(N) vector space O(N)

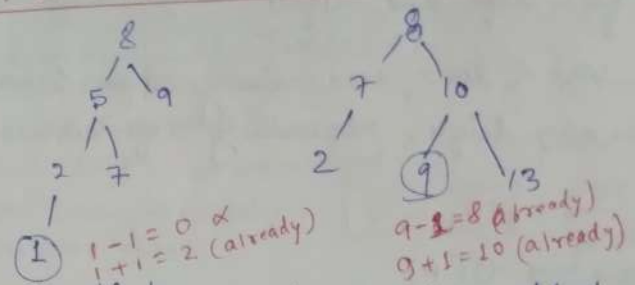
← (b) use Stack O(N), O(N) No vector

← (c) use Morris Traversal O(N), O(1)

(i) Morris traversal ∄ no. of nodes T.C

(ii) Morris traversal → desired element. S.C

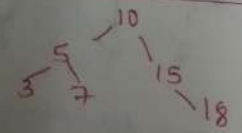
Check whether has Dead End :-



Dead End → No values can be added to it.

```
void fun(root, map<int, bool>& visited, bool& ans) {
  if(root==0) return;
  visited[root->data] = 1;
  if(root->left==0 && root->right==0){
     int xp1 = root->data+1;
     int xm1 = root->data-1 >= 0 ? root->data : root->data-1;
     if(visited.find(xp1)!=visited.end() && visited.find(xm1)!=visited.end())
     {
     ans = true;
     return;
  }}
  fun(root->left, visited, ans);
  fun(root->right, visited, ans);
}

{ ans = false;
  unordered_map<int, bool>visited;
  fun(root, visited, ans);
} return ans;
```

Count BST nodes lying in a Range



[7,15]

(a) Travel each node and compare lower & upper range & sum them

(b) Take inorder, then choose elements in range & sum them.

(c) →

(c) case 1:- val in range, → add that node & call Recursively for left and right.

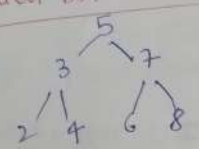
case 2:- val < low, recursively right subtree

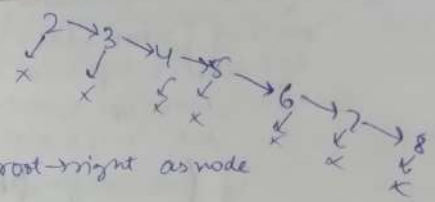
case 3:- val > high, recursively left subtree

[7,15] → [10,7,15]

Worst case → O(n)
Avg case → O(H)

Flatten BST to sorted linked list :-



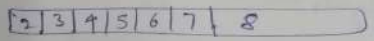


(a) store inorder and make root→right as node & left as null.

T.C → O(N)
S.C → O(N)

(b) Sort inorder in Node* vector & create the list.

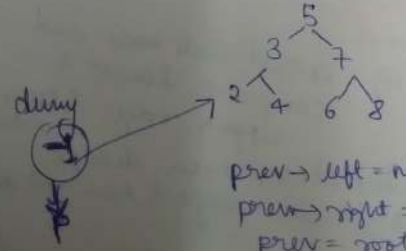
vector <node*> V



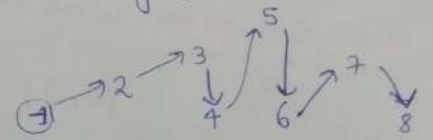
V[i] → left = null
V[i] → right = V[i+1]
last elemt → right = null

T.C → O(N)
S.C → O(N)

(c) flatten inorder wise on the go.



prev → left = null;
prev → right = root;
prev = root



return dummy→right.

```
Node* flatten(Node* root){
    Node* dum = new Node(-1);
    node* prev = dum;
    in( root, prev );
    prev->left = prev->right = 0;   // for last leaf node
    root = dum->right;
    return root;
}
```

```
void in(root, &prev){
    if (!root) return;
    in(root->left, prev);
    prev->left = 0
    prev->right = root;
    prev = root;
    in(root->right, prev);
}
```

59

* Sort in decreasing order:- Do reverse inorder.

Replace every element with the least greater element on its right :- If not greater then -1.

8 58 71 18 31 32 63 92 43 3 91 93 25 80 28

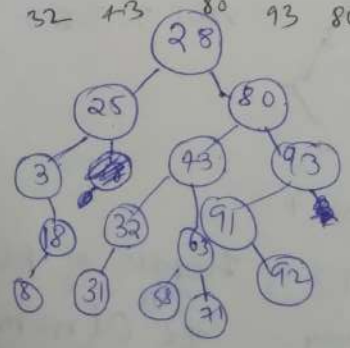
18 63 80 25 - - - - - - - - - - - -1

(a) Run two loop $O(N^2)$, $O(1)$

(b) move from right to left & store inorder successor. (If move to left then upper right one is successor. else if move to right then may not have successor)

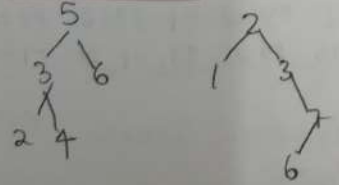
8 58 71 18 31 32 63 92 43 3 91 93 25 80 28

18 63 80 25 32 43 80 93 80 25 93 -1 28 -1 -1



40

Merge Two BST :-



1 2 2 3 3 4 5 6 6 7

V1:- 2 3 4 5 6
V2:- 1 2 3 6 7

Merge Two Sorted Array,
Merge Two Sorted LL
Merge Two LL

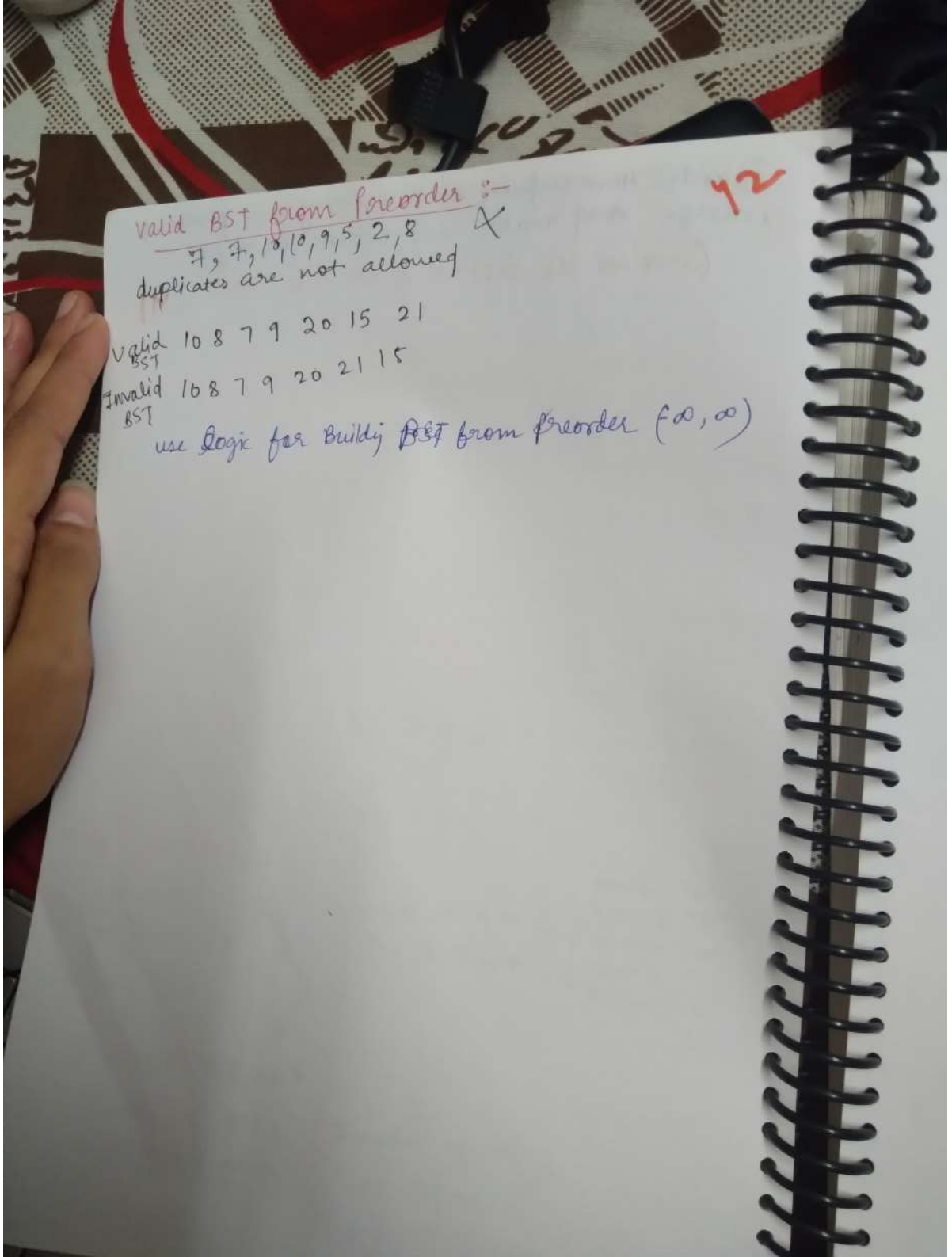
(9) Inorder of BST 1, Inorder of BST 2
Merge Sorted array.

T.C O(n+m)
S.C O(n+m)

inorder traversal using stack & keeping two pointers & merge and make ans vector on the go.

(Similar to Brothers from diff. Trees)

41


Valid BST from Preorder :-
7, 7, 18, 10, 9, 5, 2, 8
duplicates are not allowed
Valid BST 10 8 7 9 20 15 21
Invalid BST 10 8 7 9 20 21 15
use logic for Building BST from Preorder (∞, ∞)
42