Tree Series

Date: Page:

Binary Tree

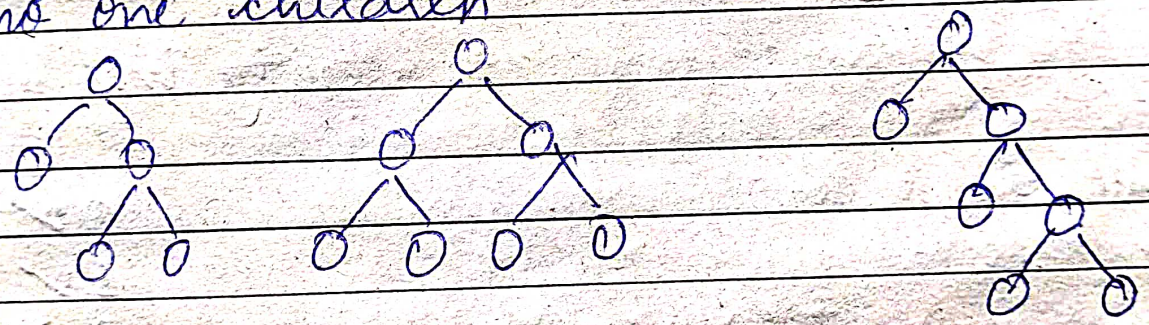
→ Hierarchial data structure
→ Binary tree: at most 2 children.
starting point: root.
→ one level down of node: children of node
→ leaf node: node with no children
→ sub-tree: tree inside another tree
→ ancestor: parents/grandparent .... for any node

Types of Binary Tree

① Full binary tree:
every node will have either 0 or 2 node
→ no one children

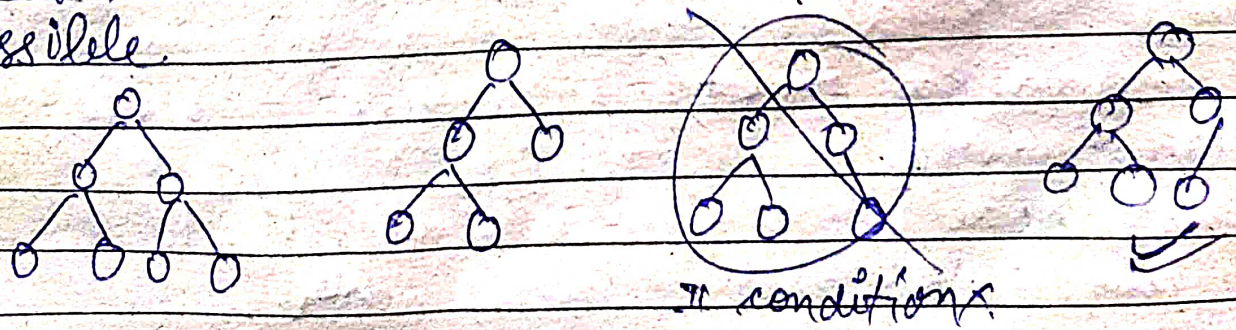
0 ✓
1 ✗
2 ✓

eg.

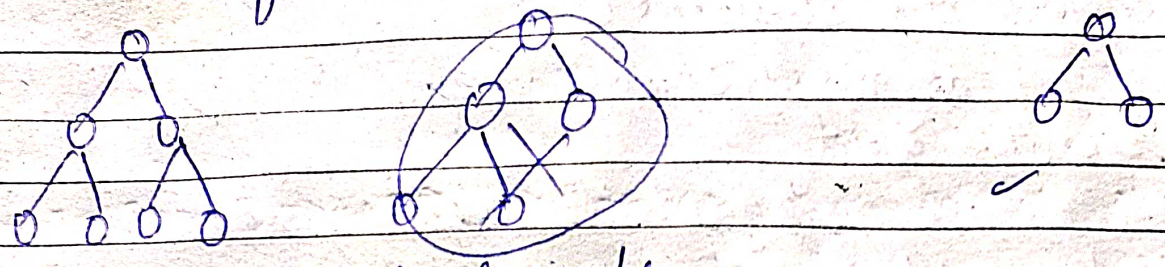


② complete binary tree
→ all level are ~~completed~~ filled except last level.
→ last level has all nodes on as left as possible.

eg



③ Perfect binary tree
→ all leaf nodes are at same level.



leaf needs to be on same level

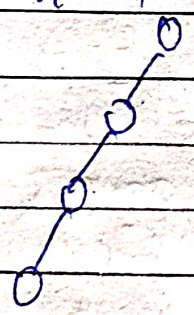
④ Balanced binary tree
→ height of binary tree is at max $\log(N)$
nodes

$n = 8$ $\log_2 8 = 3$.
→ done to improve time complexity in searching

⑤ degenerate tree
→ skew tree

0 ✗
1 ✓
2 ✗

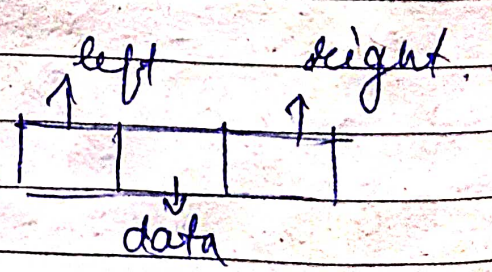
$n = 4$



→ each node has a single children

Representation in C++
→ struct node

```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
}
```



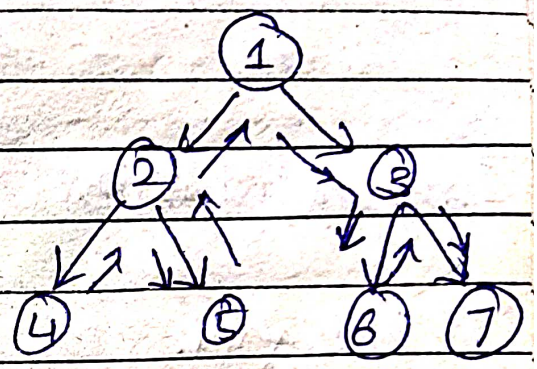

```
Node(int val){
data = val;
left = right = null;
}
```

Date : Page :

Traversal techniques
(BFS and DFS)

① DFS (Depth first search) → go depthwise
→ Inorder Traversal (Left root Right)
• go to extreme left, go to ~~center~~ root and then right extreme.

Traversal :



4 → 2 → 5 → 1 → 6 → 3 → 7

→ Preorder Traversal (Root Left Right)

• traverse root, go to left and then right.
Traversal :

1 → 2 → 4 → 5 → 3 → 6 → 7

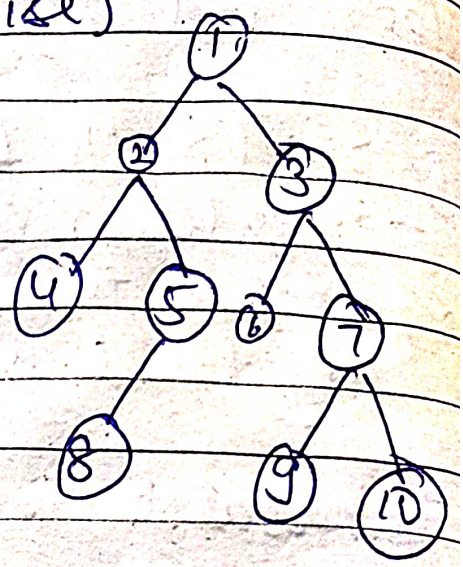
→ Postorder Traversal (Left Right Root)
• traverse left, traverse right and finally root.

Traversal :

4 → 5 → 2 → 6 → 7 → 3 → 1

Date : Page :

② BFS (Breadth first search)
→ go level wise (breadth wise)



Traversal:

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

Implementation

① DFS Q101.
ⓐ Preorder Traversal (Root left Right)

```
void preorder (node) {
    if (node == NULL) return;
    print (node → data);
    preorder (node → left);
    preorder (node → right);
}
```

TC ⇒ O(N) SC ⇒ O(N)

Q100
ⓑ Inorder Traversal (Left Root ~~node~~ right)

```
void inorder (node) {
    if (node == NULL) return;
    inorder (node → left);
    print (node → data);
    inorder (node → right);
}
```

Q.102.

Date Page

Post order Traversal. (Left Right Root)

```
void postorder (node) {
    if (node = = NULL) return ;
    post order (node → left) ;
    post order (node → right) ;
    print (node → data) ;
    }.
```

Q106 Level Order Traversal.

→ print levelwise, i.e first all level x then all level x+1 nodes from left to right

requirements = queue and vector < vector<int>>

→ pop one element from queue and push its children into queue.

→ put all elements of 1 level into vector of vectors.

```
vector < vector < int > > solve (Node* root) {
    ~~vector~~ vector < vector<int>> ans ;
    if (root = = NULL) return ans ;
    queue < Node* > q ;
    q. push (root) ;
    while (!q. empty ()) {
        int n = q. size () ; vector <int> lvl ;
        for (int i = 0 ; i < n ; i++) {
            Node* temp = q.front () ;
            q. pop () ;
            if (temp → left ! = NULL) q. push (temp → left) ;
            if (temp → right ! = NULL) q. push (temp → right) ;
            lvl. push_back (temp → val) ;
        }
    ans. push_back ( lvl) ; } return ans ;
```

Date: Page:

Q101. Iterative Preorder traversal (Root Left Right)

→ use of stack data structure

→ pop a node and push right and left node of tree in stack

→ do this untill stack is empty

```
vector<int> preorder (Node* root) {
    vector<int> res;
    if (root == NULL) return res;
    stack<Node*> st;
    st.push(root);
    while (!st.empty()) {
        root = st.front();
        st.pop()
        res.push_back(root->data);
        if (root->right != NULL) st.push(root->right);
        if (root->left != NULL) st.push(root->left);
    }
    return res;
}
```

Q100.

Iterative Inorder traversal (Left Root Right)

→ use of stack data structure

→ push root untill any node has no left children

→ ~~no~~ more auxiliary recursive space

→ pop the node if it has no child/leftchild is traversed

```
vector < int > inorder (Node * root) {
  stack < int > st;
  Node * node = root;
  vector <int> ans;
  while (true) {
    if (node -> left != NULL) {
        st. push(node);
        node = node -> left;
    }
    else {
        ~~node = st.top()~~
        if (st.empty == true) break;
        node = st. top ()
        st. pop ()
        ~~inorde~~ ans. push_back (node -> val);
        node = node -> right;
    }
  }
  return ans;
```

Date: Page:

Q10 22.

Iterative Postorder (Left Right root)

→ use of 2 stacks

→ ~~for any~~ initially, st1 has root node and st2 is having 0 elements.

→ take top element of st1, add it to st2, pop from st1 and then add its ~~right~~ and left children to st1.

→ do this untill st1 is not empty

→ then pop elements from st2 and push it to answer vector

TC → O(N) SC → O(N)

Approach 2

→ use of 1 stack

→ we will make use of curr too.

→ curr tells which node we are currently

→ we will go left untill curr is not null along with adding all elements to stack.

→ now we will go right ~~untill we don't gets~~ and assign curr to right

→ moment we reach to ~~left node~~ null

→ moving right, we will ~~che~~ remove top element and add it to our answer

→ to avoid retraversal of right/left, we check if temp element is right of top→right, if it is so then we add top element to ans and pop it.

TC → O(2N) SC → O(N)

Date : Page

Inorder / Preorder / Postorder Traversal

→ use of single stack of pair

→ first element of node will be treenode whereas other one is the deciding integer that determines the element must be pushed to which list.

→ initially root with integer 1 is being placed in stack

now we will performed below operations continuously until stack is non empty.

→ if top element integer value is 1, treenode will be added to preorder list, integer value will be incremented and if its left exist, then it will be added to stack with integer 1

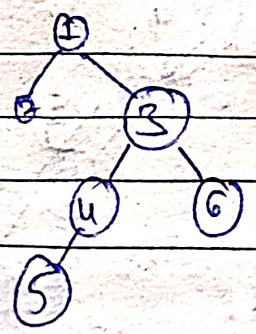
→ else if top element's integer value is 2, treenode will be added to inorder list, integer value will be incremented and if its right exist, then it will be added to stack with integer value 1

→ else top element is 3, the treenode is added to post order list and the pop operation will be performed on stack.

TC : O(3N) SC : O(N)

Q107.

Maximum Depth of Binary tree



height = 4

→ Recursive / Level order Traversal.
O(Height) ↳ O(N)

Date: Page:

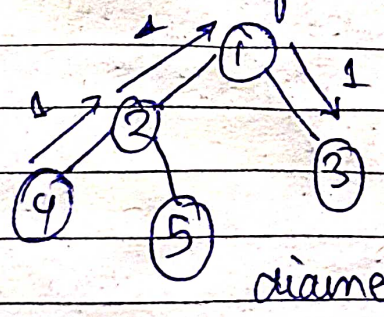
My approach recursive
→ if root is null return 0.
→ else return 1 + max(left subtree dep., ~~max~~ right subtree dep.)

check for Balanced Binary Tree.

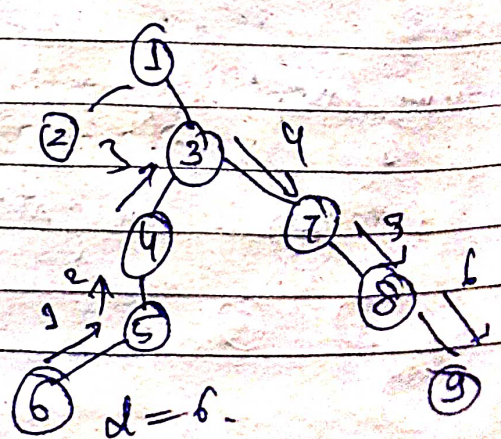
Q109.
Balanced tree height(left) − height (right) <= 1
Approach: Recursive solution.
check whether left subtree & right subtree height diff is less than equal to 1?
if it is so then check whether are left subtree and right subtree are balanced or not, if its so then return True
→ else return false.
TC → O(N) × O(N) = O(N²)

Optimized solution
changes to the finding height solution
→ finding height function also checks whether the left or right subtree is balanced or not, if it's not balanced then it returns −1.
→ −1 denotes tree isn't balanced and thus there's no sense of finding height.

Q108.
Diameter of Binary Tree.



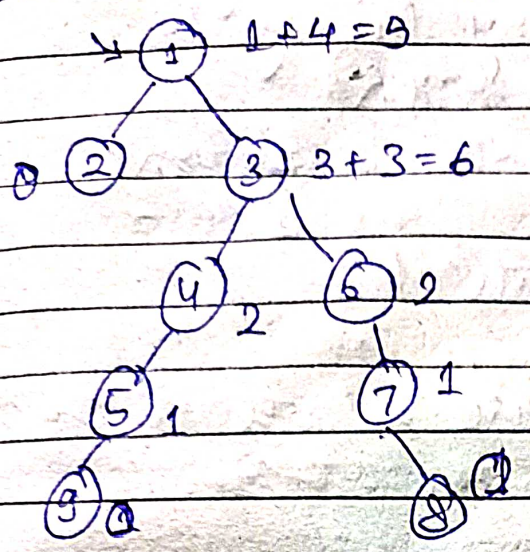
diameter = 3



d = 6.

Date : Page :

Approach 1 Naive approach.



diameter at node = ~~length~~ height of lt + height of rt.
find for every node and max one will be diameter.

TC → $O(N^2)$ SC → $O(N)$

Approach 2. Optimized Approach.
→ make changes in the function that is finding height of subtree.
→ find maximum diameter within them so as to avoid computation
TC → $O(N)$ SC → $O(N)$.

Q112.
Maximum Path Sum in binary tree.
Path: from any node a to node b is called path.
→ each intermediate node in path occurs once.

Approach 1: Brute force
→ every possible combination of node A and B.
→ N^2 combinations
TC → $O(N^3)$

Date : Page :

Approach 2.
Height and width of any binary tree.
max path = val + (maxL + maxR)
→ have curved path as the backbone.

Q111.
Check if 2 trees are identical
→ given 2 trees
→ find if both trees are identical or not.

Approach
→ do any traversal on both the trees and check if both are same or not.
Recursive solution

Q106.
Binary Tree zigzag level order traversal

→ given a tree, find zigzag order traversal. ie level n from left to right, n+1 from right to left, level n+2 from left to right and so on.

My approach.
→ use a deque and maintain a counter element
- that counts our current level of traversal in tree
→ if level count is even, we have to move from left to right else right to left
TC = O(N) SC → O(N)

Approach 2.

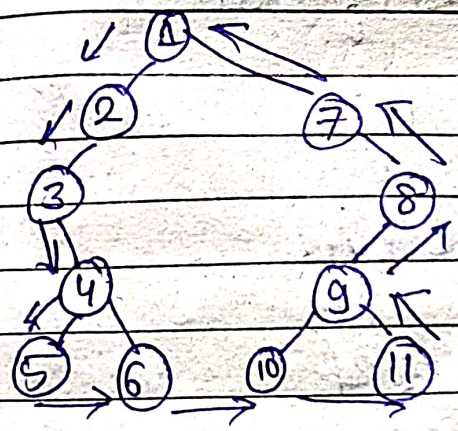
→ use of ~~stack~~ queue.

→ and a left to right bool variable.

→ if left to right is true, we iterate from 1 to n else n to 1. This variable decides the starting point.

TC → O(N) SC → O(N)

Boundary Traversal in Binary Tree



1 2 3 4 5 6 10 11 9 8 7

→ Left boundary excluding left

→ Leaf nodes

→ Right boundary in reverse

	→ check for leaf nodes
4 3 2 1	→ left boundary

→ for leaf nodes, use inorder traversal.

from inorder traversal our ds

11
10
6
5
4
3
2
1

ds.

9
8
7

stack

→ for right leaf excluding leaf

→ Start with root's → right

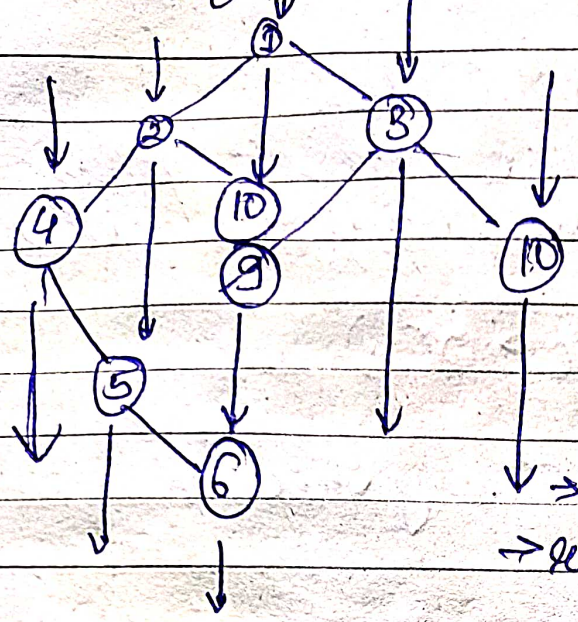
→ If no root → right, go left and add the ans in new ~~ds~~ stack

→ add all stack elements in new ds.

Date : Page :

Vertical order Traversal in binary Tree

→



4
2 5
1 9 10 6
3
10

→ Top to bottom.
→ returns list of list

→ consider lines as points on x axis and level as y axis.
→ iterate in ascending level of x axis
→ data structure → Queue.
store node, vertical and level
② map < int, map < int, multiset < int > >>

vertical level sorted order + multiple values

→ store all vertical and levels

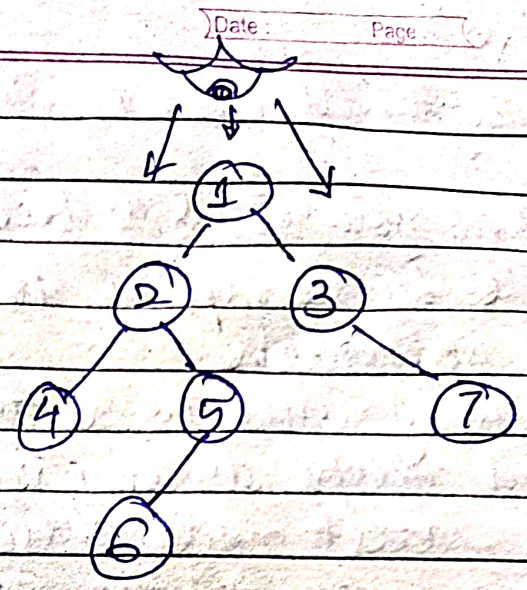
dry Run.

iteration 1 | 1 | {1, 0, 0}

iteration 2 {2, -1, 1} {3, 1, 1}
iteration 3 {3, 1, 1} {4, -2, 2} {10, 0, 2}
iteration 4 {4, -2, 2} {10, 0, 2} {9, 0, 2} {10,
iteration 5 {10, 0, 2} {9, 0, 2} {10, 2, 2} {5, -1, 3}
iteration 6 {9, 0, 2} {10, 2, 2} {5, -1, 3}
iteration 9 {6, 0, 4}
iteration 10 end.

Date : Page :

Q105 Top View of Binary Tree



→ use of line order concept.
→ first line node.
→ need of queue.
→ map data structure
que and map.

queue	map
~~(6, 4)~~	
~~(7, 2)~~	
~~(5, 0)~~	
~~(4, -2)~~	~~7 → 7~~
~~(3, +1)~~	~~-2 → 4~~
~~(2, -1)~~	~~1 → 3~~
~~(1, 0)~~	~~-1 → 2~~
	~~0 → 1~~

queue data ~~structure~~

map storing all result in sorted order.

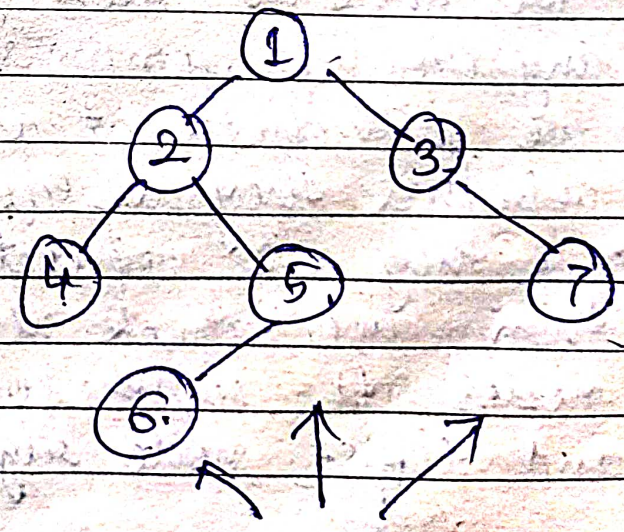
4, 2, 1, 3, 7.

node = ~~1~~ ~~2~~ 3 4 7
~~0~~ ~~-1~~ ~~+1~~ ~~-2~~ 2

TC : O(N)
SC : O(2N)

Q104. Bottom view of binary tree.

→ same as top view of binary tree
→ same data structures
→ difference : update map if you get another node in same line



queue	map
~~(6, -1)~~	
~~(5, 0)~~	
~~(4, -2)~~	-2 → 4
~~(7, 2)~~	-1 → ~~2~~ 6
~~(3, 1)~~	2 → 7
~~(2, -1)~~	1 → 3
~~(1, 0)~~	0 → ~~1~~ 5

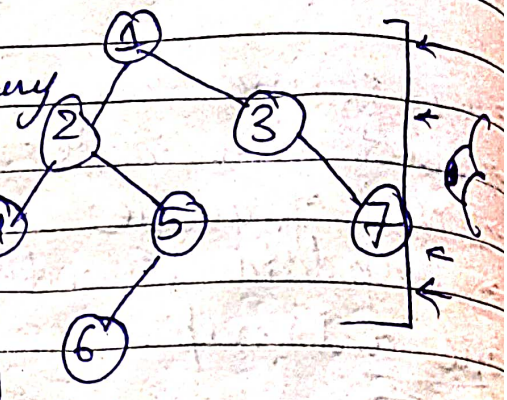
[4 6 5 3 7]
Ans

TC → O(N)
SC → O(2N)

Date : Page :

Q. 103

Right / Left view of Binary Tree



→ how you view a tree from any side from up to down.

My approach

→ apply level wise traversal and keep storing values of latest node being visited at any particular level. 1 3 7 6

→ latest update prev value with new value

→ TC : O(N) SC : O(2N)

Approach 2 # preferred)

→ Recursive solution (very short code)

→ Reverse pre-order traversal (Root Right Left)

TC : O(N) SC : O(H)

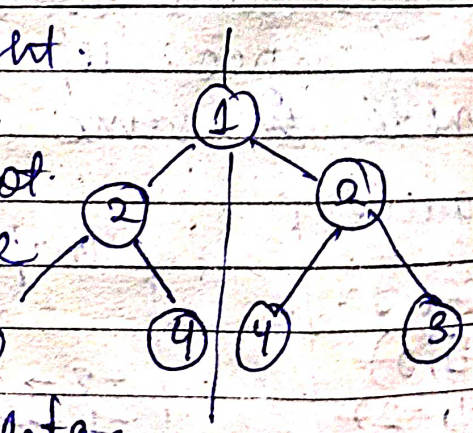
Q117 → 115 Symmetric or not

→ mirror image of left = right.

→ we have to check whether any tree is symmetric or not

→ we will check at any node whether they are equal

→ if they are equal then we check if left subtree nodes left = right subtree nodes right and leftst node right == right subtree node left

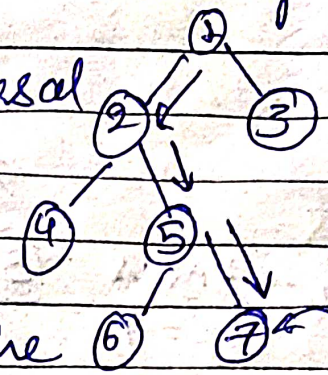


→ this will be recursive call

~~TC O(N)~~ TC → O(N) SC → O(N)

Date : Page

Print Root to Node Path in Binary Tree.

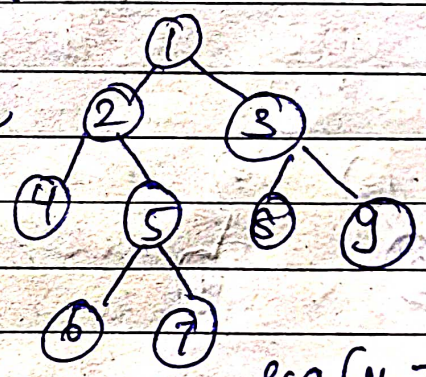


Node = 7

{1, 2, 5, 7}

→ Make use of Inorder Traversal
→ Recursive solution
→ go left, check for the element, ~~if found~~ if not found element, add the unequal elements
→ ~~out~~ if you reach false ans, remove the elements from back until you didn't get true value.
→ when you find element, return the path. (Backtracking Problem).

Q40. Lowest common Ancestor.



→ ancestor existing deepest at the level.

O(N) → find path O(N)

O(N) → space O(N)

extra S.C.

lca(4, 7) = 2

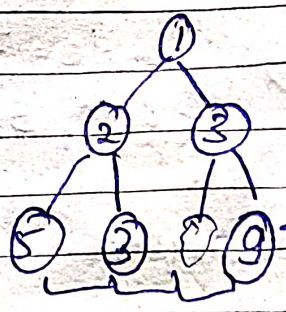
lca(5, 9) = 1

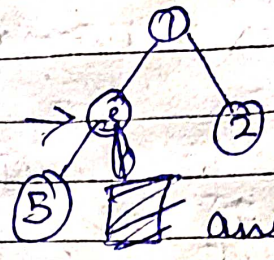
lca(2, 6) = 2

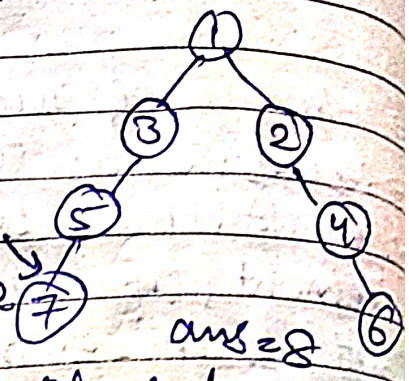
→ Recursive traversal technique
→ DFS until you dont get both the nodes as answer.

Date : Page :

Q112 Maximum width of Binary Tree.



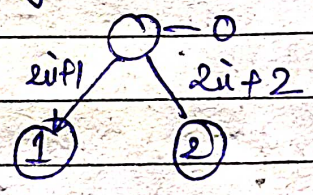


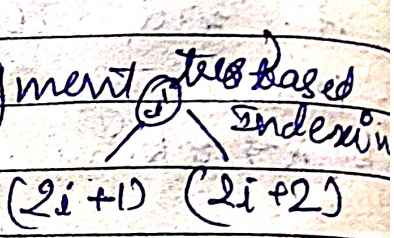


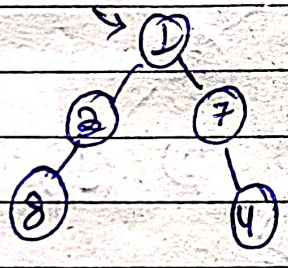
width → no. of nodes in a level between any 2 nodes

→ Level order Traversal.

→ do indexing too. (used in segment tree based indexing)







~~(4, 4)~~
~~(8, 1)~~
~~(7, 2)~~
~~(3, 1)~~
~~(1, 0)~~

width = ~~2~~ 4

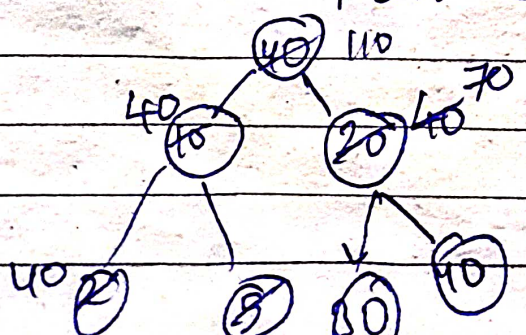
Children sum property

→ given binary tree → maintain children sum

→ node = left + right

→ can increment nodes by +1 to make property

→ Recursive Traversal

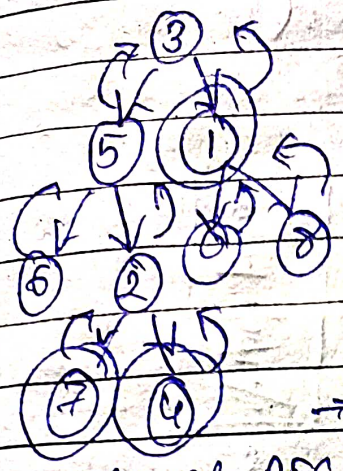


10 + 20 = 30 < 40

2 + 5 ≤ 7 < 40

30 + 40 ≠ 70 > 40

Print all nodes at a distance K in binary Tree



K=2 Target = 5

ans ⇒ 7, 4, 1

→ nodes can be at top as well as bottom

→ we can't traverse upward.

→ we thus mark parent Node

→ use of BFS (queue)

→ with BFS find all parent nodes for each parent

→ if value is given, find the node with any traversal technique

~~7~~ 4
~~8~~
~~0~~
~~2~~
~~6~~
~~1~~
~~5~~
~~3~~

→ once you find the node, perform DFS upward & downward to find the nodes

→ keep visited node to avoid retraversal.

vis	q
8	4
2	3
5	6
	2
	5

distance = = reqd.

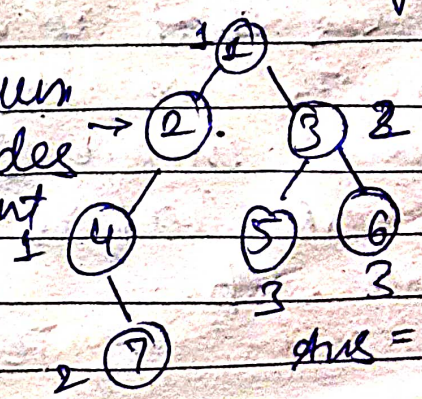
→ take all 3 nodes together, move radially outward.

→ check vis for avoiding loops.

do the steps until you don't get req'd distance.

Minimum Time taken to burn a binary Tree.

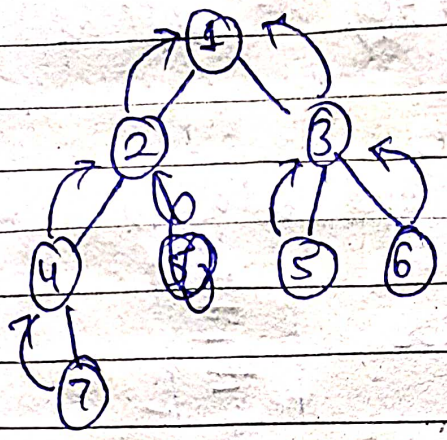
→ given a node, find minimum time by fire to burn nodes if it can burn adjacent 2 nodes at 1 s rate.



ans = 3 sec

→ BFS → go radially outwards so as to burn tree → Problem: links pointing downward
→ need of parent pointers

I. Parent Pointer



7 → 4
6 → 3
5 → 3
4 → 2
3 → 1
2 → 1

6
~~5~~
~~4~~
~~3~~
~~2~~
~~1~~

Q

II BFS Traversal.
→ if value is given, find the value using any traversal technique.
→ no need if address is given

Burning Process

6 → 1
5 → 1
7 → 1
3 → 1
4 → 1
1 → 1
2 → 1

visited map

6
5
7
3
~~4~~
~~1~~
~~2~~

Q

→ same time
→ same time

→ at a time burn left, right and parent and at the same time, mark everyone visited.

t = 0 1 2 3

→ we create a bool type var that ensures that if any one of the node is burnt, we assume its time to be taken in consideration.
→ if no leaf is burnt, we don't increment and stop.

TC = O(N) + O(N)
= O(N)

SC = O(N)

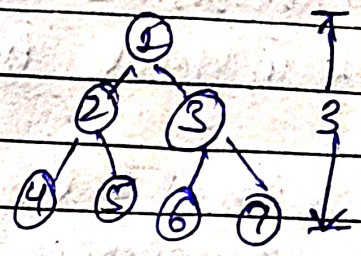
Date Page

count total no of nodes in complete binary tree

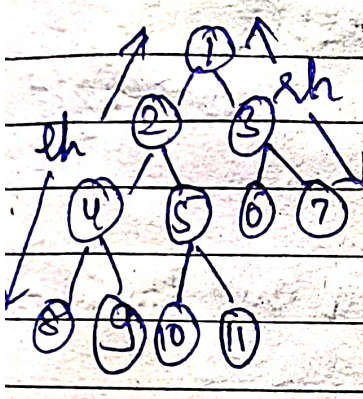
<O(N) time complete tree.

Approach 1. any tree traversal TC→O(N) SC→O(h)
height of complete tree at worst case = log N.

Approach 2.
→ check for every subtree without traversal.



no. of nodes = $2^3 - 1$



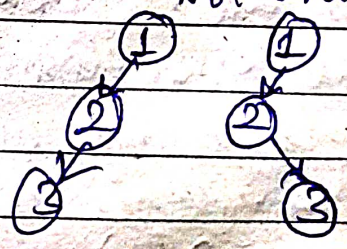
lh = 4 ✗
rh = 3

1 + check(lst) + check(rst)
if lh == rh return $(2^{lh} - 1)$

TC → $O(\log N)^2$ SC → O(log N)
recursive space.

Q113 construct a binary tree with:
(a) Preorder and Postorder.
Preorder → 1 2 3 (RLR
✗ Postorder → 3 2 1. (LRR)

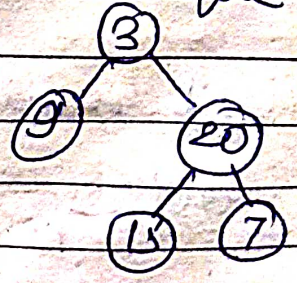
not unique



(b) Inorder and Preorder

✓ Inorder → [9 3 15 20 7]
Preorder → [3 9 20 15 7]

unique



Date : Page :

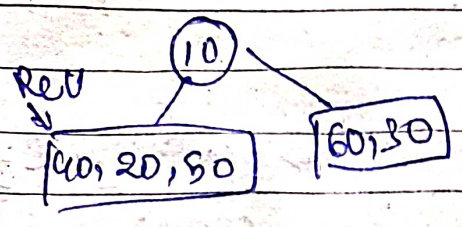
Q119 Construct a Tree from Preorder and Inorder

(~~Root Left~~ Right)

Pre: [40 20 50 10 60 30]

In: [10 20 40 50 30 60]

(~~Root left~~ Right)
left root



→ Recursive Tree Solution

if inorder has nothing at right & left, it's leaf

Inorder [x ___ o ___ y] x elements left subtree

Preorder [o ___ ___] y elements → right

x y

TC ⇒ O(N) SC ⇒ O(N)

Q114 Construct a Tree from Post order and Inorder

Inor : [40 20 50 10 60 30]

Postor : [40 50 20 60 30 10]

Last one is root of tree

→ all the elements before root in inorder are of ~~tree~~ left subtree and rest are right subt

→ TC → O(N) SC → O(N)

Q129 Serialize and Deserialize Binary tree

root → [serialize] → string

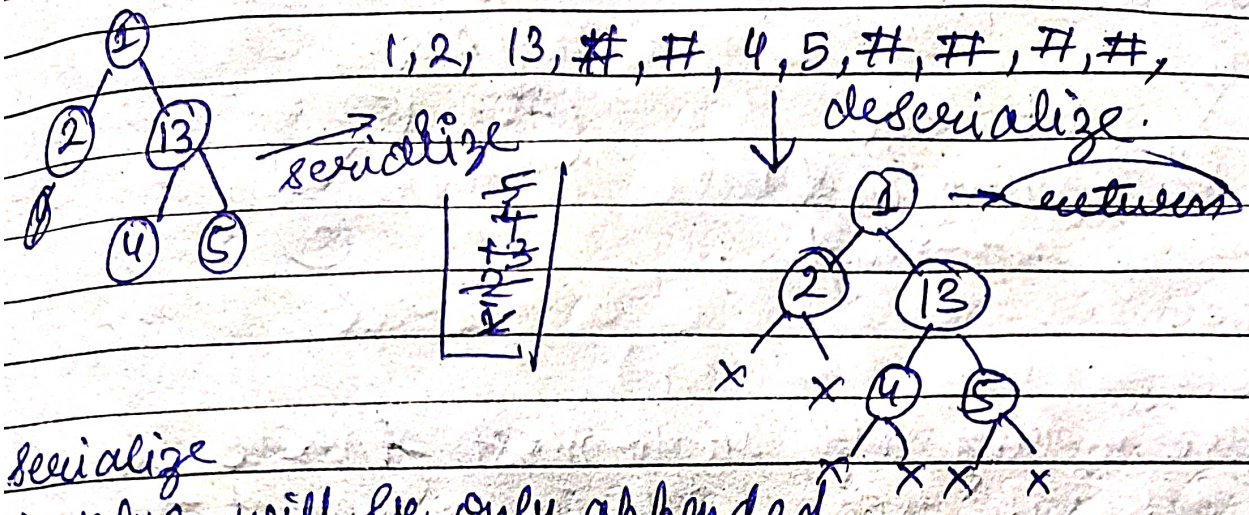
→ [deserialize] → root

→ 2 functions : serialize & deserializ

→ serialize returns string out of root

→ deserialize return root out of string.

→ need entire binary tree back.
→ serialize → reverse → deserialize.
→ level order traversal.



Serialize
→ value will be only appended if the top node is not null.
→ we will add nodes to queue irrespective of they are null or not.
→ in case we get null, we add "#," to our one and return ans.

TC ⇒ O(N) SC ⇒ O(N)

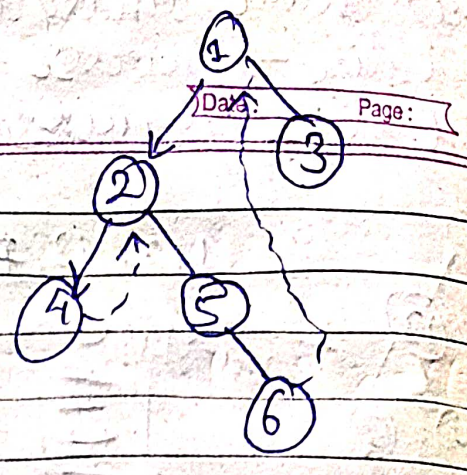
Deserialize
→ use of stringstream → we'll create nodes and one by one add them to queue.
→ this will be done untill queue is not null.
→ in case of #, will assign null to next node to the current front element of node

TC → O(N) SC → O(N).

Morris Traversal Preorder
→ no extra space
TC → O(N) SC → O(1)
→ Makes use of threaded Binary Tree.

Ex

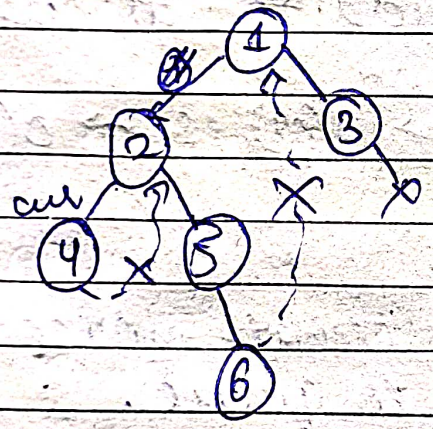
Inorder Traversal.
→ backward Pointer



I case no left.

left→null
print (root)
root→right

II case.
→ before moving left, right most ~~g~~ node in ~~lft~~ left ~~sub~~ tree will be connected to current.
→ current got to left
→ if thread already exist, remove thread



4 2 5 6 1 3

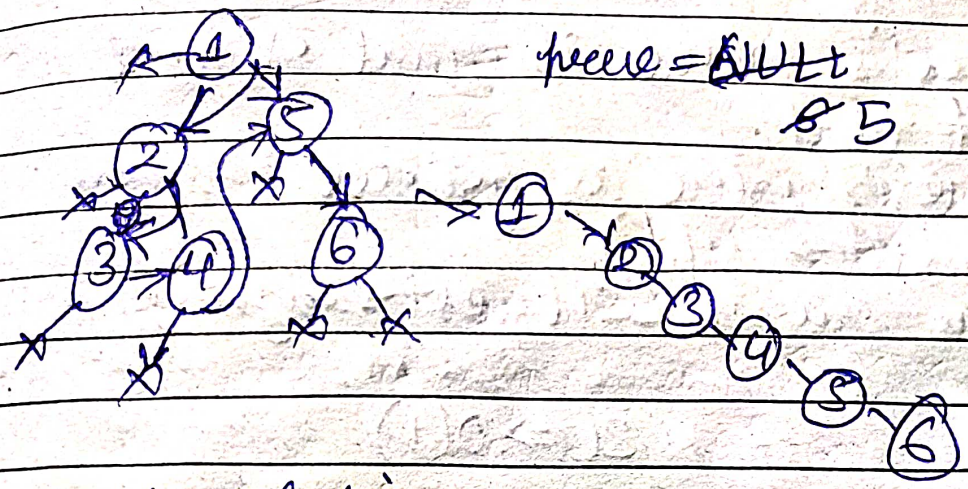
*don't leave threads.

Q116 Flatten Binary tree to Linked list.

→ convert a linked list representing the preorder traversal.
→ no external space, rearrange nodes.

initial = prev = NULL.
→ at each step you'll call solution first for right subtree and then left subtree
→ afterwards node's right contains prev value and node's left initialized to NULL
→ in case node is null, return;

→ before ending recursive call make sure to initialize prev with node



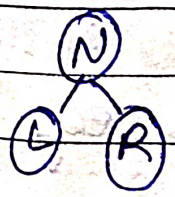
prev = NULL
& 5

TC ⇒ O(N)
SC ⇒ O(N)

Stack solution

```
st.push(root)
while (!st.empty()) {
    curr = st.top(); st.pop();
    if (curr->right) st.push(curr->right)
    if (curr->left) st.push(curr->left)
    if (!st.empty) curr->right = st.top();
    curr->left = NULL
```

Binary search Tree



→ L < N < R in every BST
→ left subtree must be BST
→ right subtree must be BST.

Why BST?
→ searching any node takes $O(\log_2 N)$ in all cases
→ but in BT normal, it takes $O(N)$

Q119 Search in BST

→ return address of node that needs to be find out

→ if ~~node~~ root's value is equal or null return ~~null~~ root

→ else if root's value is greater return searchBST (root → left)

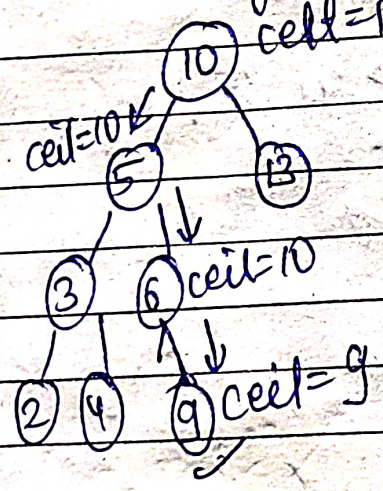
→ else return search BST (root → right)

→ recursion

TC ⇒ $O(\log_2 N)$ SC ⇒ $O(1)$

Q124 Ceil in BST.

→ node that is least greater or equal to the given key



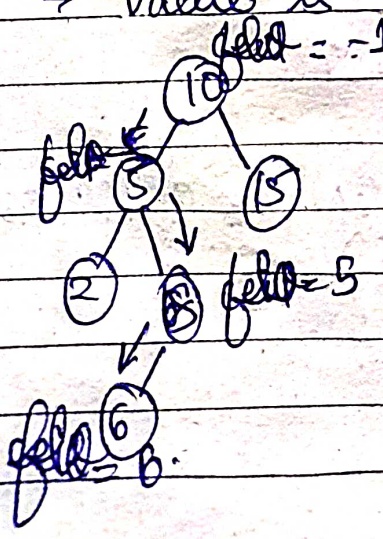
key = 8 initial ceil = -1

Ans ceil = 9

TC ⇒ $O(\log_2 N)$ SC ⇒ $O(1)$

Q125 Floor in BST

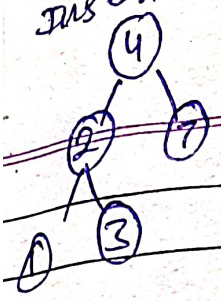
→ value is largest smaller or equal to key



key = 6 floor = 5

Insert node in BST

node = 5



→ inserted nodes at leaf.

→ where you face null, place it as leaf node

Delete a node in BST

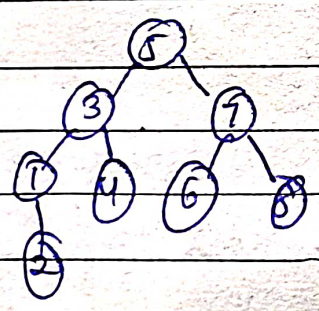
→ search node

→ deleting

after searching the node, detach it with parent and its left is being attached with

K^{th} smallest/largest element in BST.

smallest K^{th} element.



K = 3. Ans = 3.

if we sort the elements and find 3rd element, we get 3.

Naive approach → do any traversal technique storing each value in container, sort the containers and find K^{th} smallest element

Problem: TC = $O(N) + O(N \log N)$

SC = $O(N)$

Expected TC : $O(N)$ SC : $O(1)$.

efficient Approach.

→ Inorder traversal of any BST is always in sorted order and thus 3rd visited node will be answer.

But still we use SC → O(N)
when you hit left ~~mode~~ right, do counter++
moment we get count == k, that's answer.

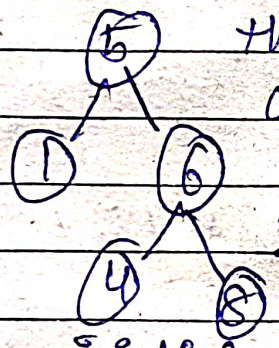
Recursive : TC → O(N)
SC → O(N)

Iterative : TC → O(N)
SC → O(N)

Morris Trav TC → O(N)
SC → O(1)

Kth largest ~~traversal~~ node ??
do one traversal : ~~of~~ to find N.
find N-Kth smallest node.

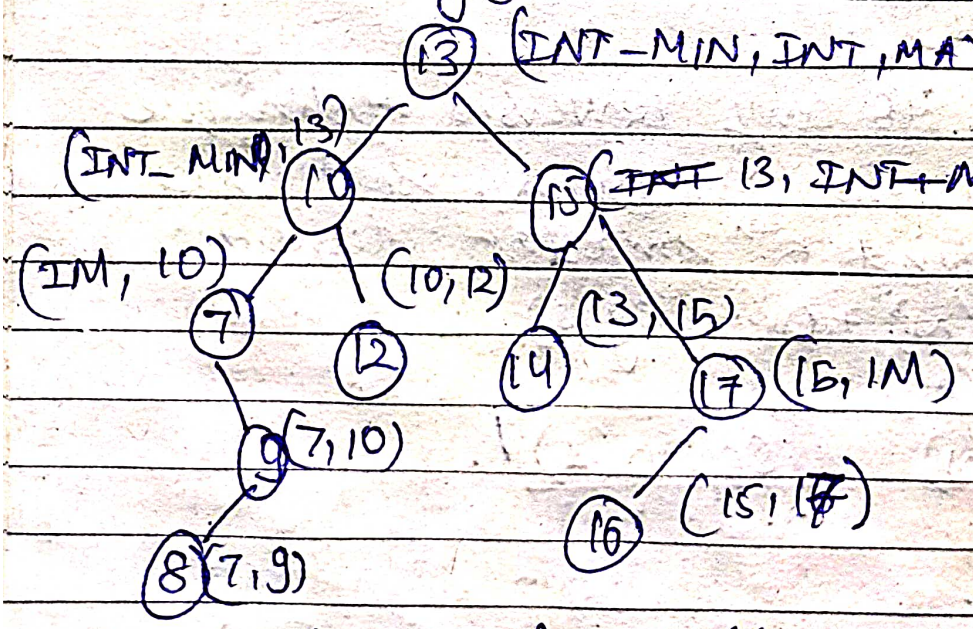
Q[2]. Check if tree is BST or BT (validate BST)
BST → all elements in LST ≤ node ≤ all elements in R



this isn't a binary search tree.

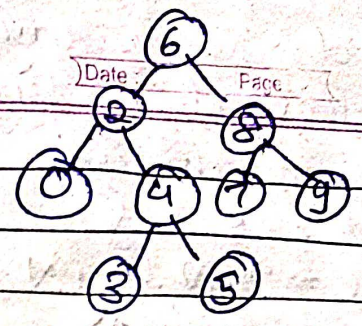
→ provide range to particular range.

→ initial range → INT_MIN to INT_MAX



→ recursive calls

Q122. Find LCA of 2 nodes in BST
→ Lowest common ancestor.



Approach 1 Naive Approach
TC: O(N) ~~for~~ ~~f~~ → same as we found LCA of BS. using DFS.

Approach 2. (My approach)
→ recursive solution (DFS)
→ if root is null or it is equal to any of the nodes, return root.
→ if p and q both the values are lesser than root, return recursive ~~solutio~~ calls for root → left
→ if p and q both the values are greater than root, return recursive call for root → right
→ if none gets followed, return root.

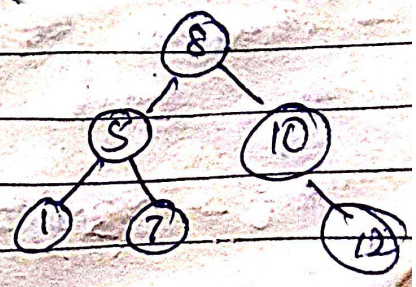
→ find the point from where your path splits
TC → O(H)
SC → O(1)
ASC → O(H)

~~Q123.~~

Construct BST from given preorder Traversal

preorder → {8 5 1 7 10 12}

Naive approach:
→ at each iteration, determine where the particular element will be placed in the ~~tree~~ BST agreeing the property of BST.



TC → O(N×N) SC → O(1)

BST Inorder Traversal.

→ sort the array, this will be inorder traversal of tree.

→ create unique Binary tree from preorder and inorder

TC → $O(N \log N) + O(N)$

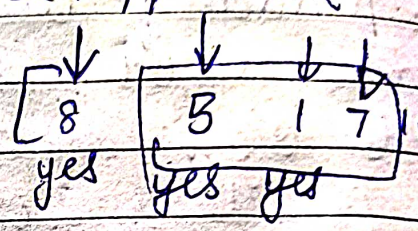
SC → $O(N)$

Approach 3. (efficient) Tree

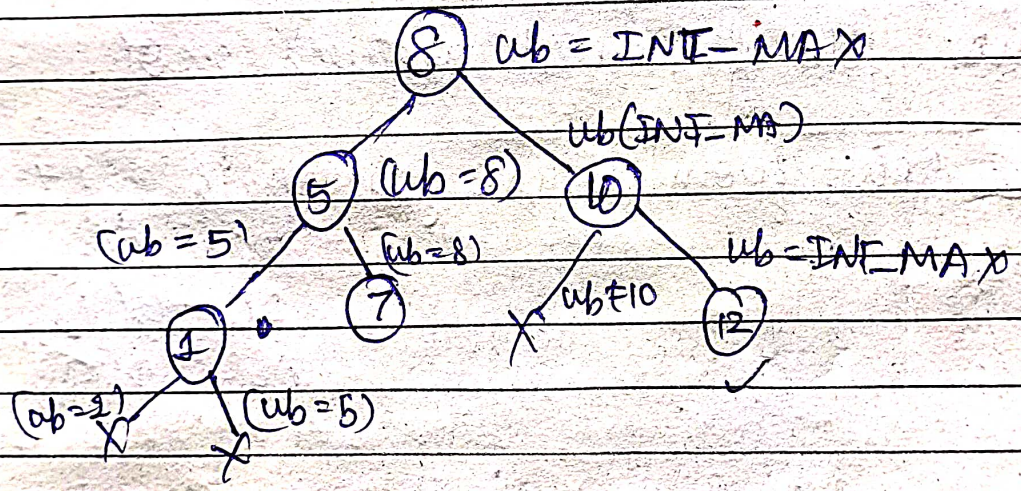
* Recall validate Binary search Tree question

* ~~reduce~~ adjust search space approach

ub = INT_MAX



→ is element in upper bound.



→ if it is within upper bound, increment the count of array iterator ~~before~~ after creating node at the correct place.

→ for left subtree upper bound will be current node

→ whereas for right subtree, upper bound will be same as that for current node.

TC → $O(3N) \approx O(N)$ SC → $O(1)$

ASC → stack space

Date. Page.

Inorder successor and Predecessor in BST

→ write the inorder of BST and whichever is the next value to it in inorder way, return it

Approach 1. Brute force.
→ store inorder way in array, locate the needed value and return its next one as result.
→ if there is no successor, return NULL
TC → O(N) + O(N) sorting SC → O(N)

Approach 2
→ do inorder traversal normally and return first value greater than 8.
→ TC : O(N) SC: O(1)

Approach 3
initially → successor = NULL
if ~~value~~ <= req. → move right
→ if you find value > req, update successor with ~~val~~ and move left.

→ when you reach null, return successor.

TC → O(H) SC → O(1)

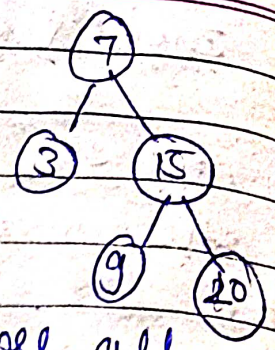
Binary search Tree Iterators..

→ given some steps and you have to perform

Date : Page :

→ given BST Iterator (7)

similar to begin(), end()



next → 3

next → 7

hasnext → true

next → 9

hasnext → true

next → 15

hasnext → true

next → 20

hasnext → false

Approach 1

use vector and store all the inorder traversal.

TC : O(1) SC : O(N)

Inorder → left

node

right

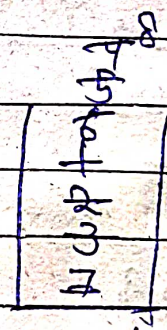
required space complexity ⇒ O(H) ~~&~~ TC ≈ O(1)

Approach 2.

→ constructor

BST Iterator(7)

→ take root and go extreme left ~~going~~ pushing elements in stack



stack → LIFO

→ ~~fast next~~ next will look at top most element out of stack.

→ after removing push right of the top element of stack and all left most elements of stack

→ if it doesn't have right, leave it

→ hasnext() = st.empty()

TC ⇒ O(1) SC → O(H)

Two Sum 4:

→ prerequisite = 2Sum

→ given BST and target K.
→ find 2 integers in BST with sum = K.

Approach 1 Brute force
→ generate Inorder Traversal that is always sorted in nature
→ keep 2 pointers on left and right
TC: O(N) + O(N) SC: O(N)

Approach 2.
reduced Space complexity.
find next() and before() function with the help of BST iterator.
increase sum → next()
reduce sum → before()

TC: O(N) SC: O(H) × 2

Recover a BST
→ given Root of BST with 2 nodes that are swapped. that converted it into normal BT
→ you need to make BT into BST

Approach 1. (Brute)
* Brute → do inorder traversal/any traversal.
→ sort the traversal and generate
→ now do inorder traversal along with the traversal in vector.
→ if there are non equality change node values.
* TC: O(N) + O(N log N) + O(N)
SC: O(N) excluding ASC with recursion

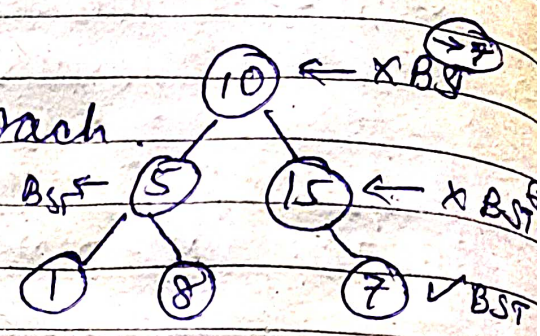
Approach 2.

→ inorder of given Binary Tree.

Date: Page:

→ in sorted array, we have to find 2 arrays that are not sorted

Largest BST in Binary Tree.



Approach 1. Brute force Approach.

Pre. : Validate BST.

→ any traversal is well and good.

→ traverse every node and validate BST

7 ✓ BST size = 1

5 ✓ BST size = 3

TC: $O(N^2)$ $O(N)$ → every node

$O(N)$ → validate a BST.

Approach 2. (Optimized)

→ if we have node, from left subtree, find maximum.

→ if max(LST) < node

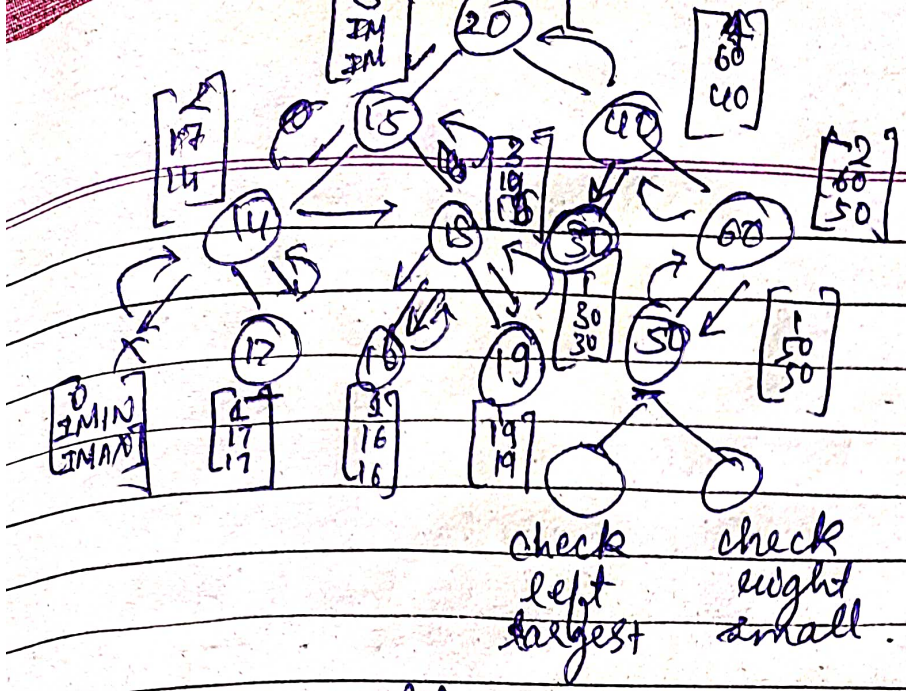
→ and from RST, find min. and if min(LST) > node

if max(LST) < node < min(LST) → ✓

→ carry largest & smallest.

→ check for size? $1 + x + y$

→ Build BST from bottom. and go up. and at the same time find largest, smallest and size.



which traversal? post order traversal.

- single node is a BST.
- at every node, check largest from LST and smallest from RST
- size = 1 + x + y if its BST
- if BST fails, pass only size that too from larger value.

TC: O(N) SC: O(1)