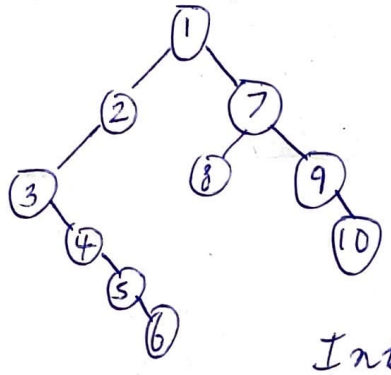
Iterative Postorder Traversal using 1 stack →



1) We'll go left until left != NULL
2) Then we'll go right, then we will again go left left until != NULL
3) Then again right, and so on

In this way, we will perform the traversal

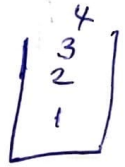
We are directly writing the code w/o writing algo.

```
cur = root
while(cur != NULL || !st.empty())
{
  if(cur != NULL)
    st.push(cur)
    cur = cur -> left
  else
    temp = st.top() -> right
    if (temp == null)
      temp = st.top()
      st.pop()
      post.add(temp);
      while(!st.empty() && temp == st.top() -> right)
          temp = st.top();
          st.pop();
          post.add(temp->val);
    else
      cur = temp;
```

Take a stack

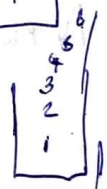
cur = 1 → 2 → 3 → null

Now cur = null
temp = 3 -> right = 4
As temp != null, cur = 4 → null

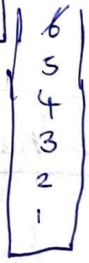


Now cur = null,
temp = 4 -> right = 5

cur = 5 → null,
Now, cur = null,
temp = 5 -> right = 6,



cur = 6 → null
Now, cur = null,
temp = 6 -> right = null
As temp == null,
temp = 6, pop 6



Print 6,
while (temp == 5 -> right) Yes it is true
temp = 5, pop 5
Print 5

This loop will print 6 5 4 3

Now also, cur = null, temp = 2 -> right
temp == null, print 2 & pop

6 5 4 3 2

Again we will come back, cur = null, temp = 1 -> right = 7

cur = 7

Now, cur = 7, push 7
& cur = 8

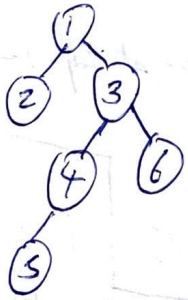
cur = 8, push 8
& cur = NULL

Now, cur = null
temp = 8 -> right = null
temp == null, yes
temp = 8, pop & print 8
8 == 7 -> right, No,
Now, its comes back & again cur = null
temp = 7 -> right = null
if (temp = null) {
temp = 7, pop & print
while (7 == 1 -> right)
temp = 1, pop & print

Now, both cur = null & st is empty.

TC - O(2N)
SC - O(N)

Maximum Depth of Binary Tree



height = 4

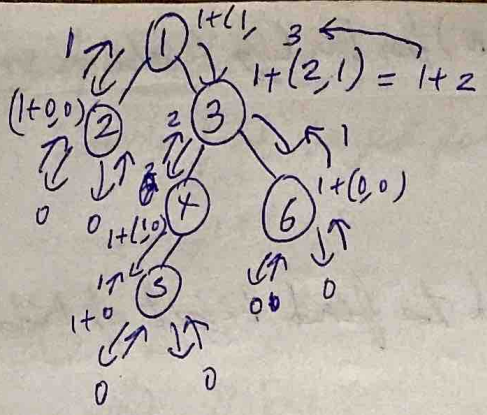
We have 2 approaches,
i.e. Recursive / Level Order Traversal

Here, we will be using recursive approach

If we are standing at any given node, then height will be

1 + max(height(node->left), height(node->right))

1 + max(l, r)



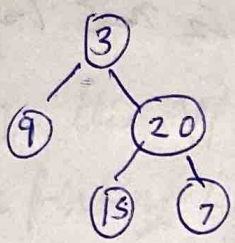
So, 1+(1,3) = 1+3 = 4

So, max height = 4

Check for Balanced Binary Tree →

For every node, height(left) - height(right) <= 1

E.g



For ③,
left height = 1
right height = 2
diff = 1

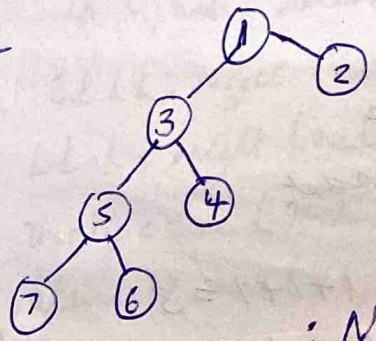
For ⑨, no left & right subtree

For ⑳, left height = 1 & right height = 1
diff = 0

For ⑮ & ⑦, no subtrees.

∴ This tree is a balanced binary tree

E.g 2



For ①
left height = 3
right height = 1
diff = 2

∴ Not a balanced binary tree

Naive approach

Go to every node & figure out left height & right height & compare difference, if it is >1 return false

Pseudo code →

```
if(node == null)
    return true
Lh = height(node->left)
Rh = height(node->right)
if(abs(Lh - Rh) > 1) return false;
bool left = check(node->left);
bool right = check(node->right);
if(!left || !right) return false;
```

TC- O(N) for traversal × O(N) for finding height
≈ O(N²)

Optimal solu →

We will use the code we used to find height of binary tree in O(N) time

But we want the answer in true / false.

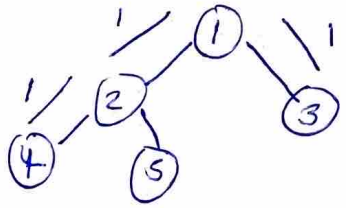
So, we will right a function, which will return height of tree if our tree is balanced,
Otherwise it will return -1, if it is not balanced.

```
int check(node) {
  if(!node) return 0;
  lh = check(node->left);
  rh = check(node->right);   ← if(lh || rh == -1) return -1;
  if(abs(lh-rh)>1) return -1;
  return max(lh, rh)+1;
}
```

Diameter of Binary Tree →

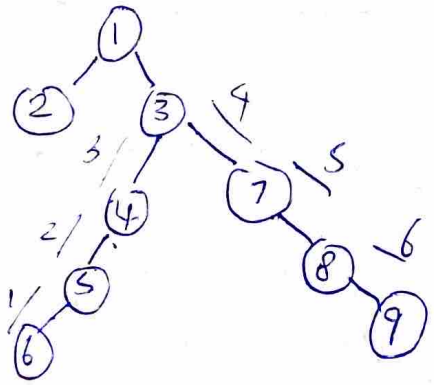
→ Longest path b/w any 2 nodes
→ path does not need to pass via root

E.g -



1+1+1 = 3
So, diameter = 3

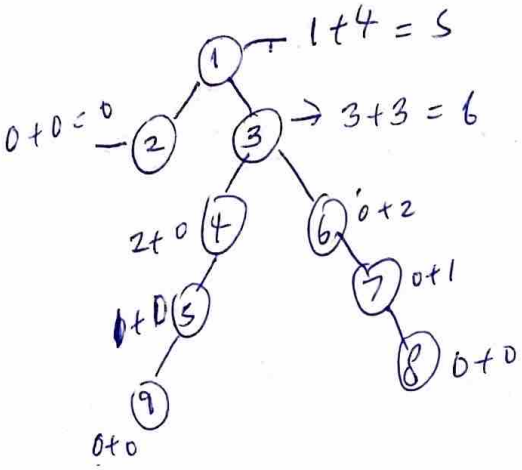
E.g -



diameter = 6

Naive Approach -

Simply find largest path by summing lh + rh



So, we will find sum of lh & rh at every node

∴ This approach will take $O(N^2)$ TC

Optimal -

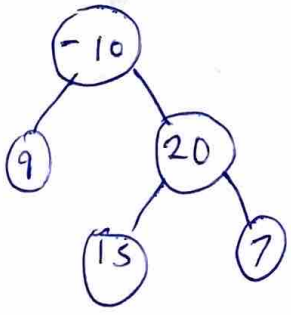
In this approach ~~algo~~, we will modify the code for finding height of tree which will take $O(N)$ Time complexity

We will calculate sum of left height & right height after making recursion calls for node -> left & node -> right, and compare the sum with max variable.

```
int find(node, maxi)
{
if (!node) return 0;
int lh = height(node -> left);
int rh = height(node -> right);
maxi = max(maxi, lh+rh);
return 1+ max(lh, rh);
}
```

TC - O(N) recursion.
SC - O(N) for stack

Maximum Path Sum →



15+20+7 = 42 ← Max

9 + (-10) + 20 + 15 = 34

Brute Force- Trying every possible pair in the tree of node A & B, where A can be anything & B can be anything.

But this solution is complicated, & for N nodes we will have to try N^2 combinations.

Better- We will use the same approach we used for finding max height of binary tree & finding max width of binary tree.

If we are standing at any node, we compute maxLeft & maxRight which represent max path sum in left & right nodes of curr node

So, ~~ans for the~~
ans for this node = val + (maxL + maxR)

And we will also maintain a max variable, which will store max path sum

```
int maxSum (node, maxi) {
    if (!node) return 0;
    lSum = maxSum (node -> left);
    rSum = maxSum (node -> right);
    maxi = max(maxi, node -> val + lSum + rSum);
    return node -> val + max (lSum, rSum);
}
```

Check if two trees are identical or not

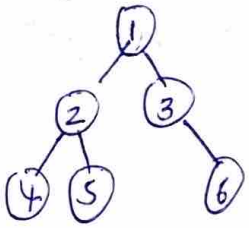
Two trees are same if they are structurally identical and the nodes have same value.

Approach:

We can do traversals of both the trees and the traversal of both the trees should be equal.

```
{
if (p==NULL || q==NULL) { return (p==q); }

return (p->val == q->val) && isSameTree(p->left, q->left)
        && isSameTree(p->right, q->right);

}
```

Zigzag Level Order Traversal →



→ 1
3 2
4 5 6

We will follow same technique which we used in Level Order Traversal.

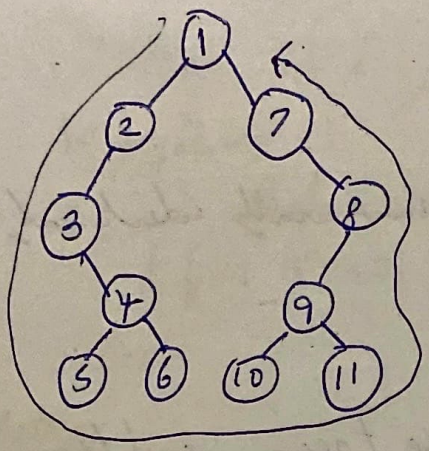
Take a <queue>, store root

Introduce a flag variable, flag = 0, which tells direction is L→R

flag = 1 tells, direction is R→L

Wherever we insert a level in our array, ~~level = 1~~ toggle level variable i.e. change it to 0 & 1 alternately

Boundary Traversal →



1 2 3 4 5 6 10 11 9 8 7

Anti clockwise

Approach: First we will take left boundary excluding leaf node
Then we will take leaf nodes
Then we will take right boundary in the reverse direction, excluding leaf & root node.

Initially store the root in an array

ds [1]

Then start from left boundary i.e. root -> left
Again we will check its left, if it exists then move left, otherwise move right

ds [1 2 3 4]

Now 4's left & right both are leaf nodes, we will stop here.

Now, we have stored the left boundary, we'll move towards leaf nodes.
To store leaf nodes we will do inorder traversal.
Then our ds will be

[1 2 3 4 5 6 10 11]

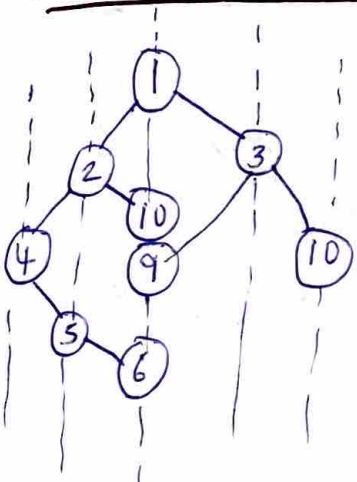
Now, Right boundary in reverse excluding leaf.
Start from root → right.
We'll use a temp stack to store values now
We'll keep storing until it reaches leaf nodes

Now store the stack in final ans array

ds | 1 2 3 4 5 6 10 11 9 8 7 |

This is the final ans.

Vertical Order Traversal



Vertical order Traversal is →

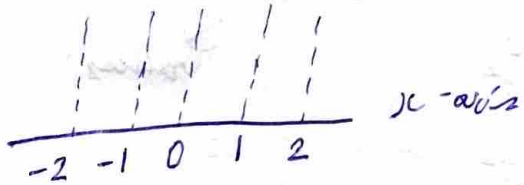
~~4 2 5 1 9 10 6 3~~

4
2 5
1 9 10 6
3
10

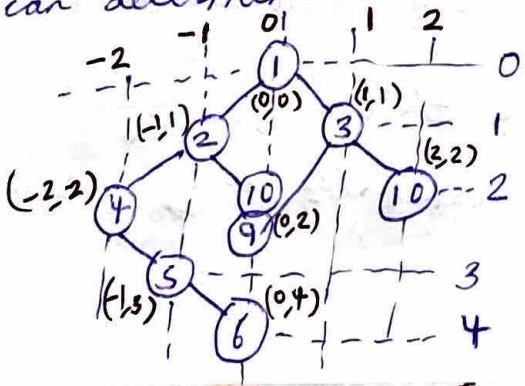
9 & 10 were on same level, so we will write smaller one first.

Approach If we consider these lines as points on the x-axis

i.e.,

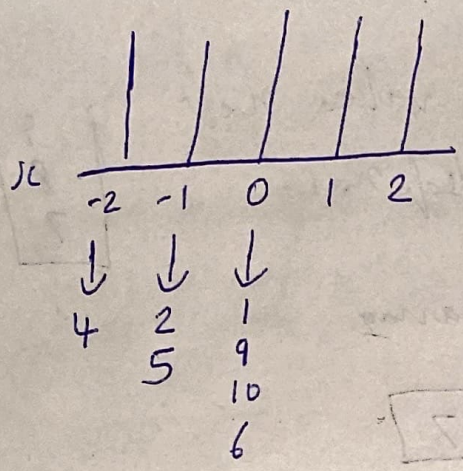


And if we consider levels as points on y-axis
Then we can determine the coordinates of node.



So, coordinates of a node are written in (x, y) format.

We can write this more simply now



This way, we can get every vertical,
Suppose, we have x, then we iterate in the ascending order of y, and similarly in y, we iterate in ascending order of x.

So, our first step will be to assign (x,y) to every node.
We can do this task with any traversal method.
Here, we will use Level order traversal to assign ~~(x,y)~~ coordinates

→ We will take a Queue which will store
(node, x, y)
x - Vertical position
y - Level position

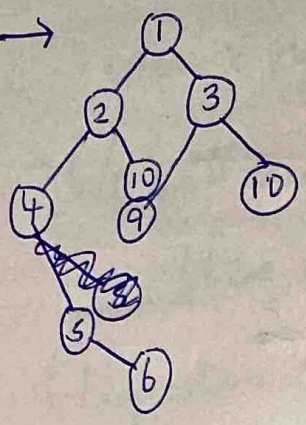
→ We will use another data structure map,
map<int, map<int, multiset<int>>>
x, y, values
This will store,
x → y → values in multiset

We are using multiset because, we can have same values.
And if we use set, then we will only have unique values.

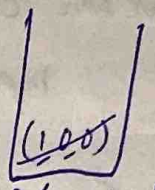
So, our map will store the values, whereas
Queue will store coordinates along with node

~~map<int~~

Now, we will start the process →



Initially while starting, store (1,0,0) in Queue before beginning loop.



Q(node, x, y)

Then we will take out this q.front()

node = 1 , (0,0)

Enter this in map,

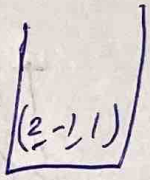
0 → 0 → {1}

For node=1, do level order Traversal

i.e. Take left node → left = 2,

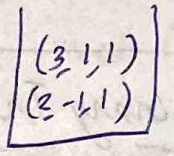
now if we are moving left, the our x will decrement & ~~x~~ if we are moving downward, y will increment

So, we will get (2, -1, 1) & store in queue



Now, Take right & increment x & y

We get (3, 1, 1) & store in queue



Now go to next iteration, get q.front()

which is (2, -1, 1)

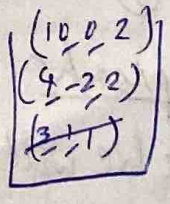
node = 2, x = -1 & y = 1

Store this in map,

-1 → 1 → {2}

& perform Level order Traversal on this

We get (4, -2, 2) & (10, 0, 2) store in q

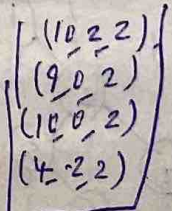


Now, get q.front() → (3, 1, 1), perform same steps

~~5~~ → 1 → 1 → {3}

& perform Level order traversal, we get

(9, 0, 2) (10, 2, 2) store in q



This way we will iterate over the queue & store it in map.

Code → We can easily store in map using

```
mp[x][y].insert(node->val);
```

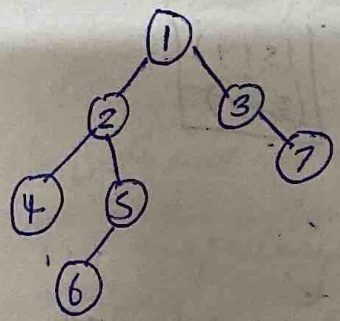
Now to store data from map in vector, code is

```
for(auto p: mp)
{
    vector<int> col;
    for(auto q: p.second){
        col.insert(col.end(), q.second.begin(), q.second.end());
    }
    ans.push_back(col);
}
```

$T.C \rightarrow O(N) * O(\log N)$

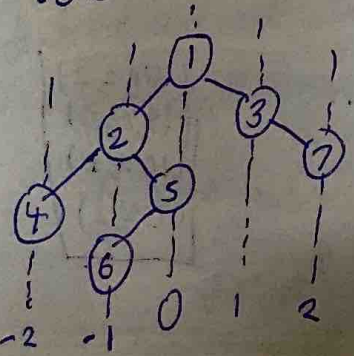
$S.C \rightarrow O(N)$

Top View of Binary Tree →



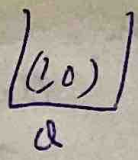
Top View → 4 2 1 3 7

Approach - We will use Level order traversal for this. & We will also use the vertical line concept from Vertical order traversal.



On every line, whatever is the 1st node from top will be our answer

→ We will use a queue to store (Node, x) where x represent vertical pos.



→ We will use a map data structure to store (Line, node) i.e. (x, node)

(Map store everything in a sorted fashion of keys)

Initially insert (1,0) in q

Take out (1,0)

node = 1 x = 0

We will check whether on line 0, do we have anything in map, i.e. mp[0] exists in map or not.

If no, then insert in map

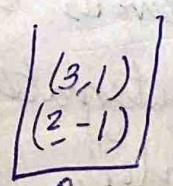
0 → 1

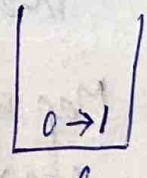
Now, ~~go~~ do level order traversal of node ①, i.e. go left & right

On left we have node = 2 & x will decrement, it will became x = -1

Push it in queue

Similarly, on right push (3,1) in q



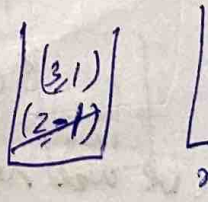


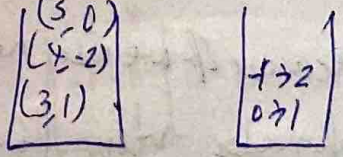
Now take, node = 2, x = -1

check if mp[x] exist in map, if no, then create entry

start its left & right in q.

i.e. (4,-2) & (5,0)





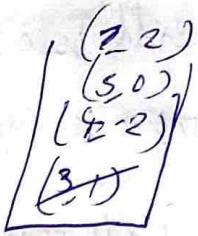
Now, node = 3, hd = 1

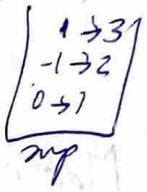
Check whether mp[1] exists in map.

If not create entry

& push its children in q

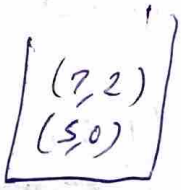
i.e. (7, 2)

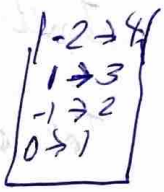




Now, node = 4, hd = -2

Put it in map & add its children in q

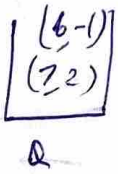


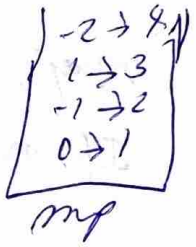


Now, node = 5, hd = 0

mp[0] already exist in map, so we will not insert this node 5 in map

Push 5's children





Similarly store all nodes in map.

This method will store only first occurrence/node of a vertical line in map.

This will give us top-view of tree.

Map will automatically sort keys, so answer will be in correct order.

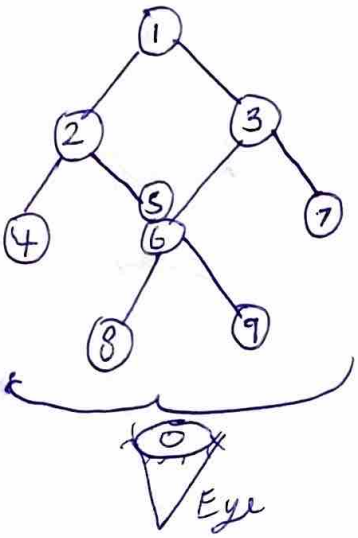
TC - O(N)

SC - O(N)

Can we use recursive traversal to solve this?

No because if we use recursive traversal, it will store leaf nodes view first, and we will not be able to get top view.

Bottom View of Binary Tree



Bottom view will be

4 8 6 9 7

5 & 6 both are visible from bottom & both lie on same vertical line, So, in this case we will take the right ~~Node~~ node, i.e. 6

Approach: We will use the concept of vertical line, just like we used in Vertical order traversal

On every line, the last node, we will store that

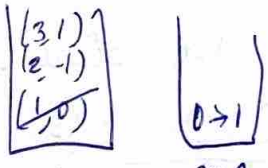
Take a Queue <TreeNode*, int> which will store (Node, x) where x represents vertical line

We will also use a map to store (x, node->val)

Initially start with (1, 0)

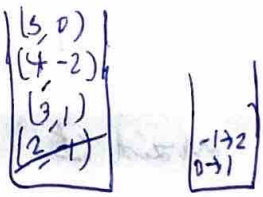
Node = 1, x = 0

We will insert mp[x] = Node->val & insert its children in the queue i.e., (2, -1) & (3, 1)



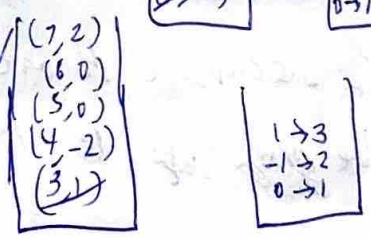
Now, Node = 2, x = -1

Insert in map, push its children in queue

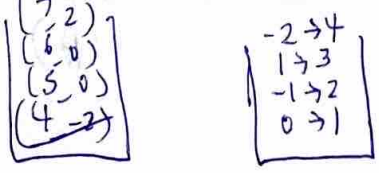


Now, node = 3, x = 1

Perform same task



Now, node = 4, x = -2



Now, node = 5, x = 0

Update mp[x]

(9, 1)
(8, -1)
(7, 2)
(6, 1)
(5, 0)

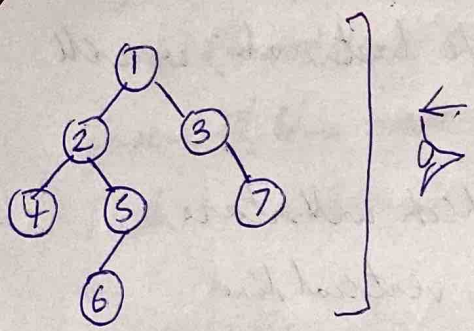
-2 -> 4
1 -> 3
-1 -> 2
0 -> 5

Similarly in this way we will proceed, we will overwrite previous entry if it exists, which will give us the bottom view in the end.

TC - O(N)
SC - O(N)

Here, also we cannot use recursion

Right / Left Side View of Binary Tree



Right view

1 3 7 6

Approach: We have to traverse in such a way that we store the last node of every level will be Right View

We can use any traversal i.e Recursive / Iterative

Level order traversal will use more space, so we will avoid it.

But, in recursive traversal, then space complexity will be ~~gre~~ ~~dot~~ ⊗ height of tree

TC - O(N)
SC - O(H)

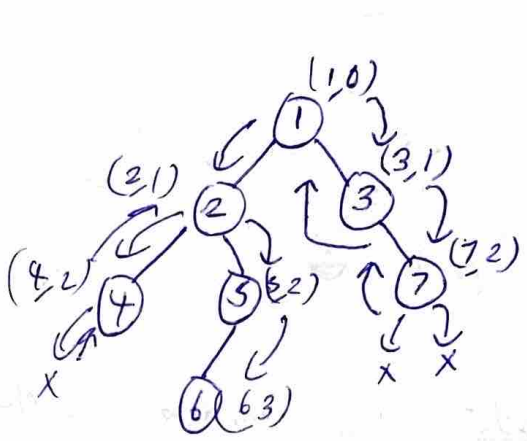
Here we will use recursive approach, because code is short

Here we will use Reverse Preorder Traversal, i.e.

Root Right Left

```
if (node, level)
{
  if (!node)
    return;
  if (level == ds.size())
    ds.add(node)
  if (node->right, level+1);
  if (node->left, level+1);
}
```

ds me answer store krenge



ds

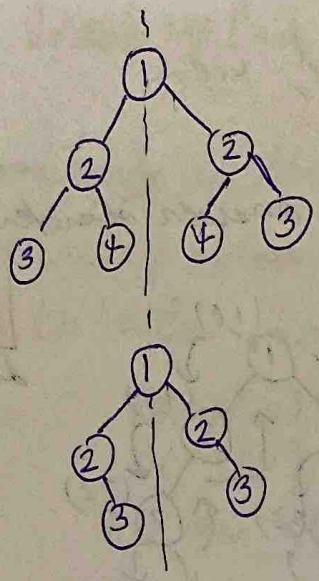
We will only store 1 element at each level, so we will check whether (level == ds.size())

Suppose, level = 0 & size = 0, we'll add
Now, level = 1 & size = 1, we'll add
level = 2 & size = 2, we'll add
Now, it will go back to root, level became 0,
Now, root->left, level = 1, & size = 3 we'll not add
root->right, level = 2, size = 3, we'll not add
root->left, level = 3, size = 3, then only we'll add

TC - O(N) SC - O(H)

For left view, we'll go left first then right

Check for Symmetric Binary Tree



If form a mirror of itself around the center

This is a symmetric binary tree

This is not symmetric

Approach In a mirror Left → Right & Right → Left

If we ignore the root, left subtree will start from root → left & its mirror will be root → right subtree

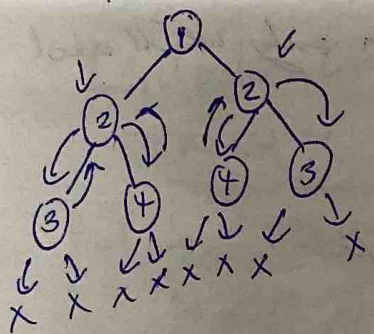
(Root → left)	(Root → Right)
Root Left Right	Root Right Left

If we are doing preorder traversal i.e.

Then this will be the mirror

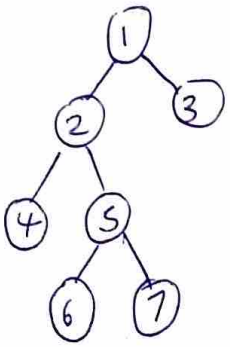
We will do both these traversals simultaneously on both left & right subtrees

We will just move opposite in both subtrees



If everyone was matching then it is symmetric

Print Root to Node Path in Binary Tree



Node = 7

From ①, we have to find path till ⑦

O/P → {1, 2, 5, 7}

Approach- We will use Inorder Traversal here

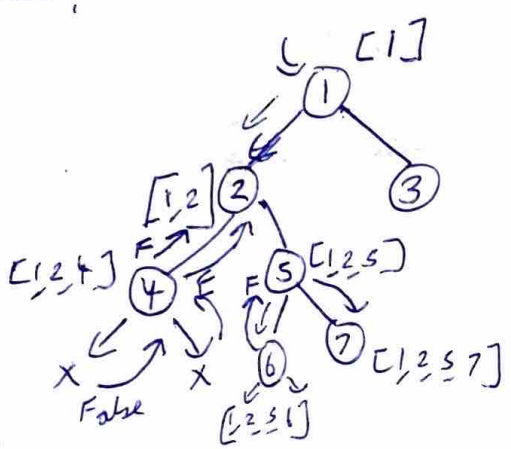
Preorder & Postorder can also be used, but they are complicated.

1) Initially we will start with the root, when we ~~reach~~ are at root, we'll be have a data structure which will be empty initially.

If node != null & ~~node != target~~ push it into ds

First left recursion then right recursion

When a node returns false it means that ⑦ is not present in this path, so we will go back & pop that element from our data structure.



Here ④ returned a false to ②, so it was removed from ds.

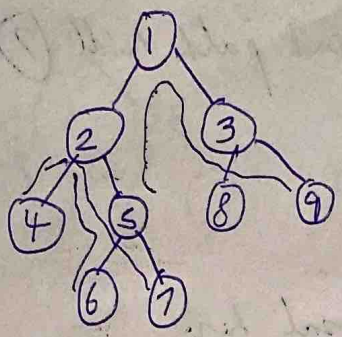
Now ② goes to right

Now when we reach node ⑦, our ds has [1, 2, 5, 7] as we have got our required node, we will return True with array as it is.

Code →

```
arr.push_back(root->val);
if(root->val == k) return true;
if(getPath(root->left, arr, k) || getPath(root->right, arr, k))
  return true;
arr.pop_back();
return false;
```

Lowest Common Ancestor in Binary Tree



lca(4,7) = 2

lca(5,4) = 1

lca(2,6) = 2

The ancestor that exist at deepest node

Brute Force - We can store path of both the nodes and then compare paths.

e.g. :- node = 4 (1 2 4)

lca(4,7) node = 7 (1 2 5 7)

We can see both paths match till ②. So, ② will be the LCA

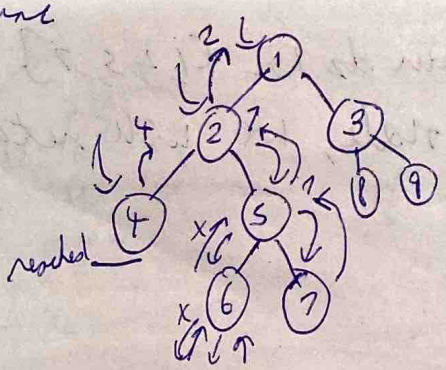
TC - O(N) + O(N)

SC - O(N) + O(N)

Better - We will optimise space in this approach.

First we will go left, when we reach one of the node we are looking for, we'll return that node to previous func

lca(4,7)



Now at ② we are getting both ④ & ⑦.

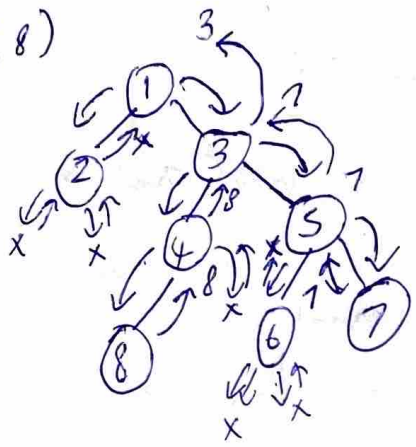
Which means 2 is LCA

When we returned ④, we will make a recursion call in the right to check its right children.

Node ② will return 2, now ① will go in right & look for nodes again it will get back NULL.

So, answer will be 2

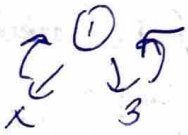
lca (7, 8)



As soon as we reach 8, return

3 is LCA

We get



So, answer is ③

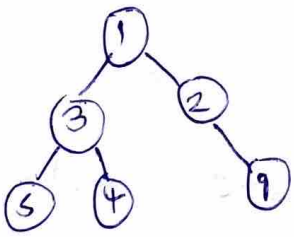
Code →

```
if(!root || root==p || root==q) return root;

TreeNode *l = solve(root->left)
     r = solve(root->right)

if(!l) return r;
else if(!r) return l;
else return root;
}
```

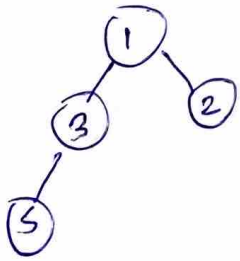
TC - O(N)
SC - O(N)

Maximum Width of a Binary Tree →

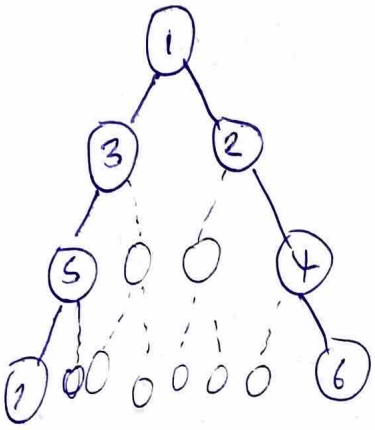
Width means no. of nodes in a level b/w any 2 nodes



Max width of this tree = 4
because if 2's left exist, then it would have been 4.



Here max width = 2
We cannot consider last level here because our task is to find max width b/w any 2 nodes & there is no node in right side of last level.



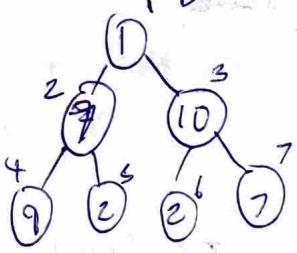
Max width = 8

Approach- Width is dependent on the level

We can say, width can be calculated only b/w first node in that level & last node in that level,

We should follow Level Order Traversal.

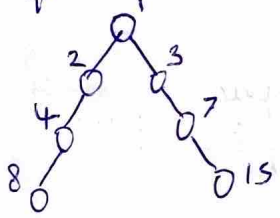
If we can index the tree based on their position, then Soln will be very simple



Then we'll have to go to each level.
Pick up the first node index
Pick up the last node index
Subtract it and add +1.

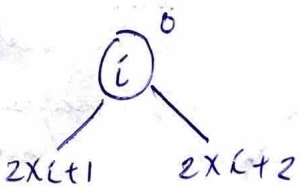
e.g - 7 - 4 + 1 = 8 ← max width

Even if there are no nodes in b/w, we'll index in proper format.



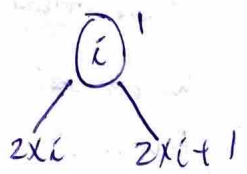
Now, our main task is How to index the nodes?

If we consider 0 based indexing, then



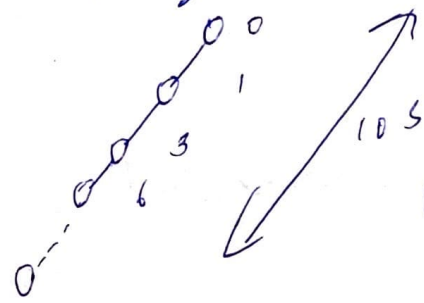
lchild = 2Xi+1
rchild = 2Xi+2

If we consider 1 based indexing, then



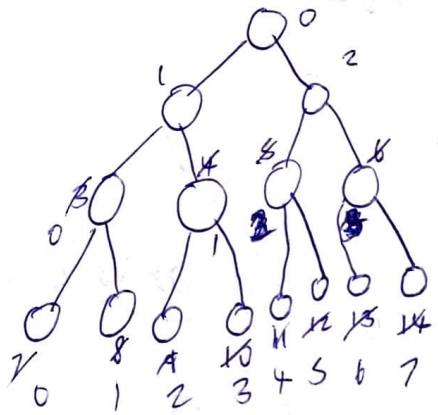
lchild = 2Xi
rchild = 2Xi+1

But this method also has one issue, suppose if we have a skew tree of ~~height~~ 10^5 nodes. then



then integer will overflow because indexes increase by this

So, in order to prevent overflow

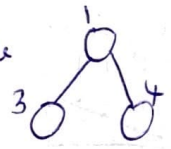


instead of 3 4 5 6 indexes we'll make it 0 1 2 3 in this level

In this level we'll change 7 8 9 10 11 12 13 14 to 0 1 2 3 4 5 6 7

Q. How will we do this?

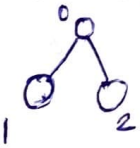
Sol:- In this node



i = 1
lchild = 3
rchild = 4

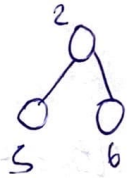
We'll follow formula of 0 based indexing

What if we do i-1, i.e. make it i = 0



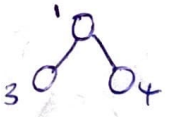
Then lchild = 1
rchild = 2

Eg2



i = 2
lchild = 5
rchild = 6

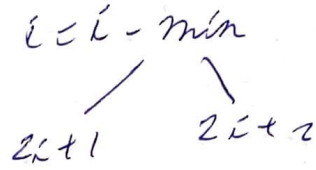
→ we'll do i = i-1 = 1, then



Now we have got

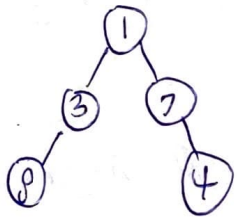
1 2 3 4 in level 3,

We'll perform same steps in calculating indexes of level 4



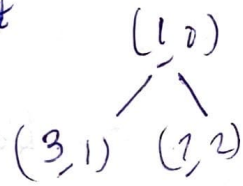
Then On every level take the min & max index, subtract & add +1.

We'll follow Level Order Traversal,
We'll store (node, index) in Queue.



max width = 1

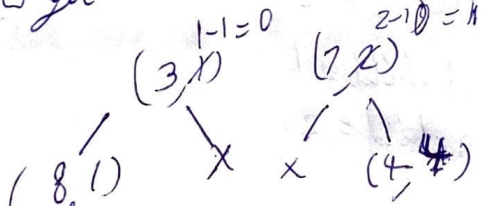
node = 1, ind = 0



2*i+1 → l child
2*i+2 → r child

Push in queue

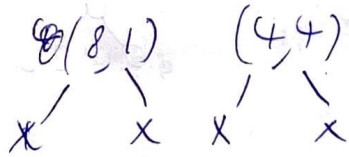
Now get



max width = 2-1+1 = 2

Pop them from q

Now,



max width = 4-1+1 = 4

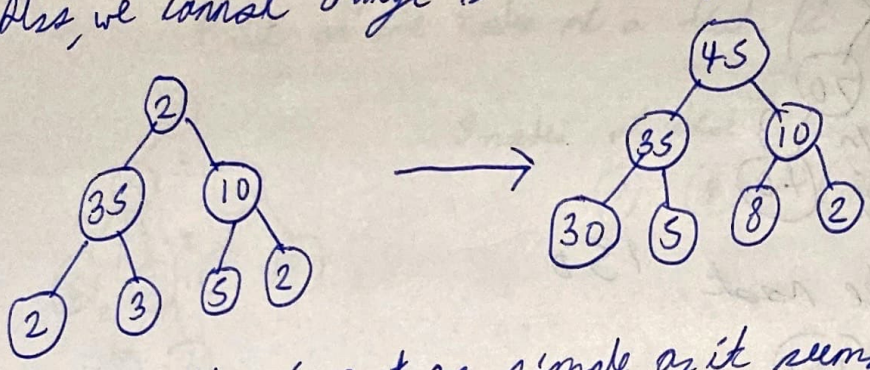
Code →

Children Sum Property -

According to this property, ~~Node->val = left~~

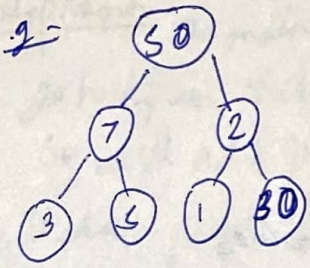
node->val = lnode->val + rnode->val

If binary tree does not follow this property, then we can increment any node by +1, any no. of times
To make it follow this property
Also, we cannot change structure of tree.



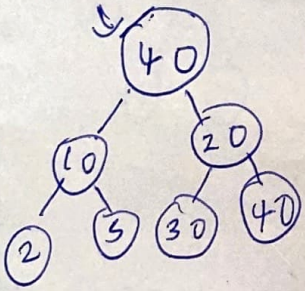
This problem is not as simple as it seems to be

E.g.-

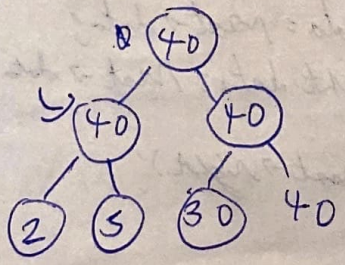


First thing which comes to our mind is that
we make 7 → 8
& 2 → 31
but we cannot just change this, because above these 2 nodes, their children sum property will be disturbed.

Approach- We'll use recursive traversal

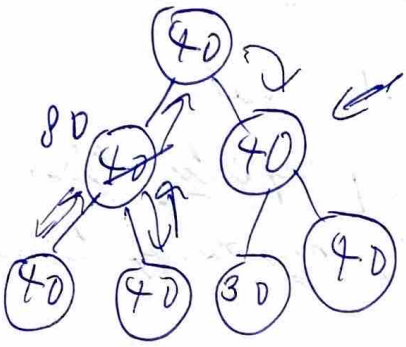


10+20 = 30 < 40
Now, if sum < 40, then we replace both 10 & 20 by 40



Now 2+5 = 7 < 40
Now, again if sum < 40

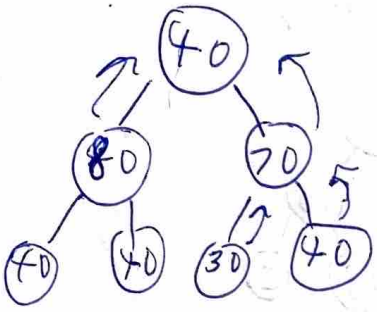
We'll change 2 & 3 to 40 & when we go back make 40 to 80 in their parent



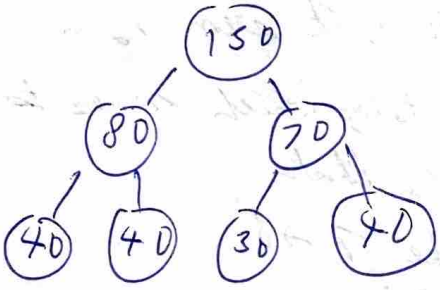
Now,

30 + 40 = 70 >= 40

Now if sum > 40 then make root 70



Now make root 150



So, while going down increase the node values & while coming back up, change node values accordingly.

Code →

```
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;
else {
  if (root->left) root->left->data = root->data;
  else if (root->right) root->right data = root->data;
}
solve(root->left); solve(root->right);
int tot = 0;
if (root->left) tot += root->left->data;
```

if (root->right)) tot += root->right->data;

if (root->left || root->right) root->data = tot;

}

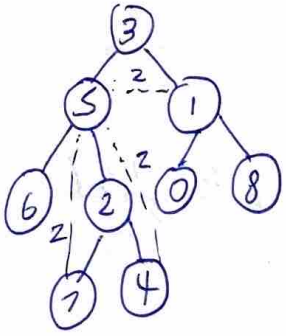
TC - O(N)

SC - O(1)

Nodes at a distance K

K = 2, target = 5

Print all the nodes at a dist. 2 from node (5)

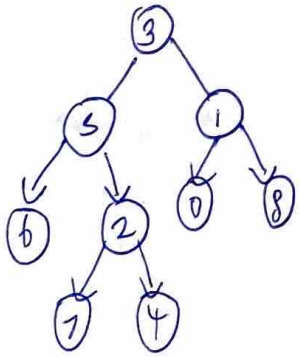


3 nodes at dist 2 are (7) (4) & (4)

Approach The main issue in solving this is that we cannot go back from child node to parent node.

So, we'll add parent ~~node~~ pointer to all nodes, i.e. we'll store parent of all nodes in a map or array

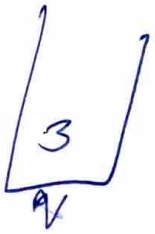
We'll use a BFS to perform this task



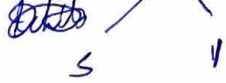
Initially take a queue

Push root in it

Now start iterating over queue's elements



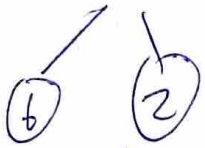
Node = 3

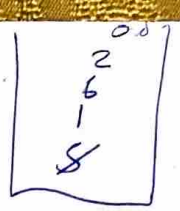


We'll push l & r in q & store both's parents as (3)

We can store it in a map.

Now, node = 5

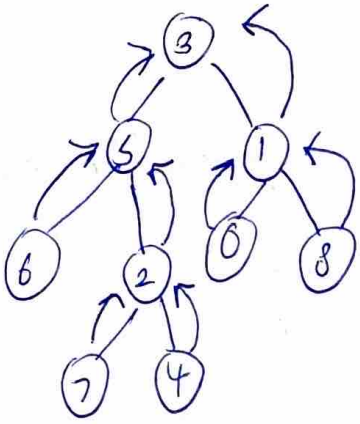




Pushing q & ~~upto~~ store parent node of both

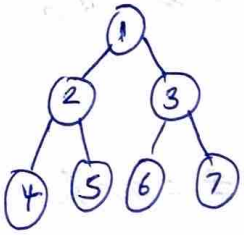
Similarly we'll proceed & store parent's of all nodes

K = 2, target = 5



Now, ~~if~~ 5 target is given, we have to find other nodes at dist = 2

Count total Nodes in a Complete Binary Tree



cnt = 7

We can simply count by traversing the tree

This is Naive approach.

TC - O(N) SC - O(h)

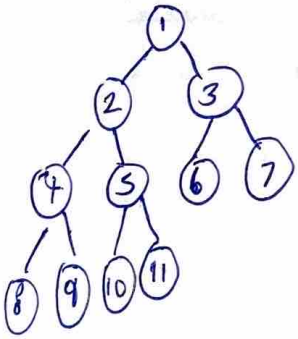
Optimal approach -

If height of tree = 3

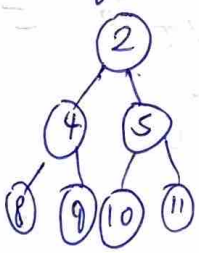
Then no. of nodes = $2^3 - 1 = 7$

This is a property of Complete Binary Tree.

But what if some nodes are NULL in last level



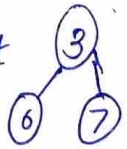
If we are standing at



then height of subtree = 3

& no. of nodes = 7

If we are at



then height = 2

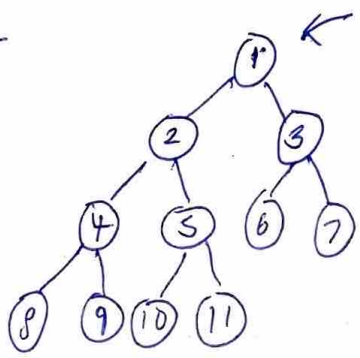
& no. of nodes = $2^2 - 1 = 3$

Now, if we are at root, then

total nodes = 1 + 7 + 3 = 11

So, we'll check for every subtree in this way.

Dry Run



At root, calculate leftheight & right heigh

Here, lh = 4

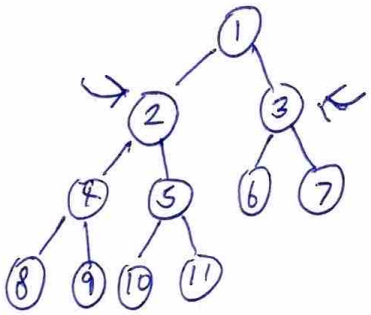
rh = 3

We can see lh ≠ rh

Then we cannot use this formula here.

So, we'll add 1 + leftrec + right rec.

We'll move left & right subtrees, we'll check whether lh == rh in their subtrees.



In ② lh = 3

rh = 3

lh == rh

We can use the formula

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

= 7

In ③ lh = 2

rh = 2, lh == rh. We can use the formula

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

So, we get the answer 1 + 7 + 3 = 11

TC - O(log N^2) SC - O(log N)

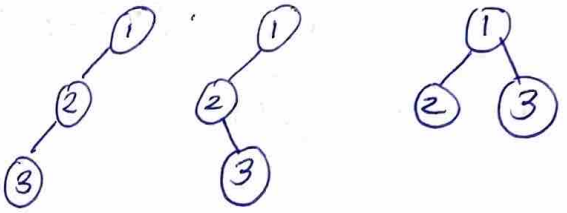
Requirements needed to construct a Unique Binary Tree

a) Preorder & Postorder –

PreOrder → 1 2 3

PostOrder → 3 2 1

Many trees are possible



So, if we are given a preorder & postorder traversal then we cannot construct a unique binary tree

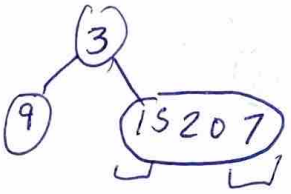
b) Inorder & Preorder –

Preorder → 3 9 20 15 7

Inorder → 9 3 15 20 7

(Left) (Right)

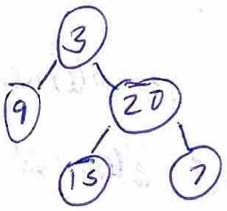
(3) is root of tree, then



Now (9) becomes root

Now (20) becomes root

We are determining root by checking how elements are coming in preorder traversal



c) Inorder & Postorder can also be used

Construct a Binary Tree from Inorder & Preorder

inorder → [40 20 50 10 60 30]

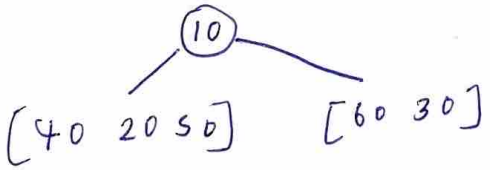
preorder → [10 20 40 50 30 60]

(root)

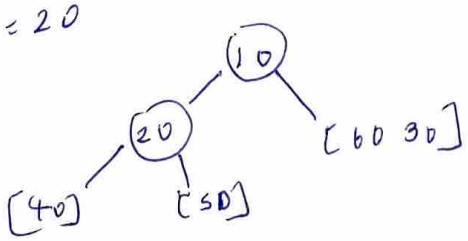
Left Root right

Root Left Right

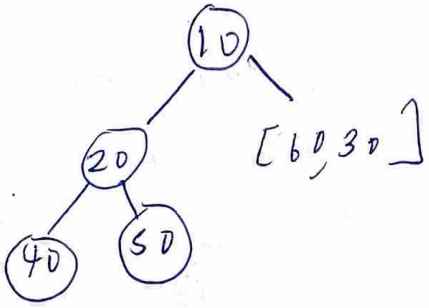
We split using inorder traversal



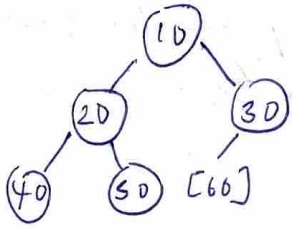
Now, root = 20



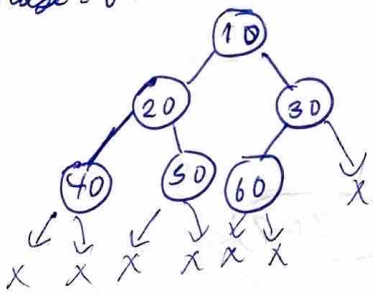
Now root = 40, then 50



Now root = 30

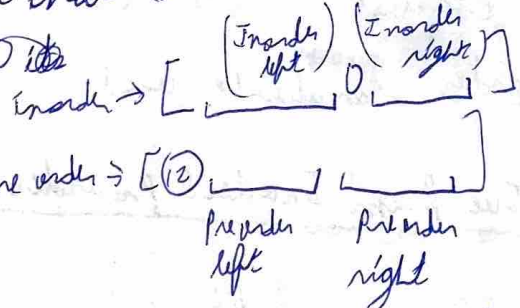


Now root = 60



Code Approach-

We are chosing first element of preorder as root, then we search for that element in inorder traversal



Make a recursion call with left & right subarrays

TC - O(N)

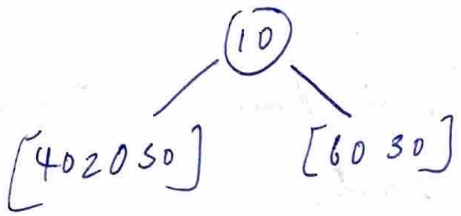
SC - O(N)

Construct Binary Tree from Postorder & Inorder

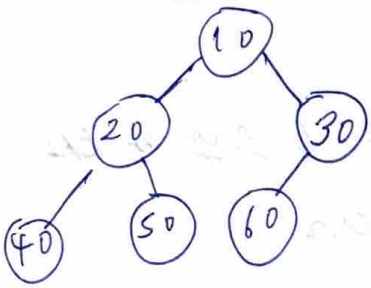
Inorder → [40 20 50 10 60 30]

Postorder → [40 50 20 60 30 10]

Last element is root in postorder



Here, 20 will be root



This will be final tree

Map

40 - 0
20 - 1
50 - 2
10 - 3
60 - 4
30 - 5

iRoot = 3

numLeft = 3 - 0 = 3

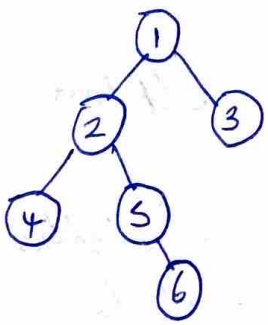
Find index using map

Same code as previous

Morris Traversal

TC - O(N)
SC - O(1)

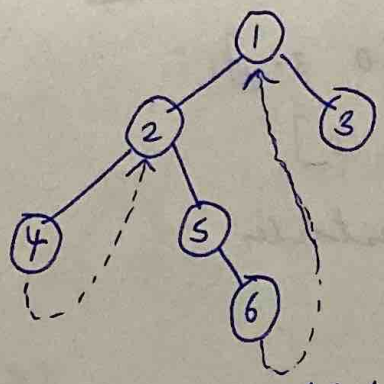
- It uses constant space
- It uses Threaded Binary Tree



Inorder - 4 2 5 6 1 3

Preorder - 1 2 4 5 6 3

Suppose, we had a backward pointer, like



We can observe a pattern, that from last node of a subtree we go back to root.

Case 1 - If left == null, then print() & go right

Case 2 - Before moving left, rightmost guy on the left subtree ↗ cur

Connect it.

Then after that cur = cur -> left;

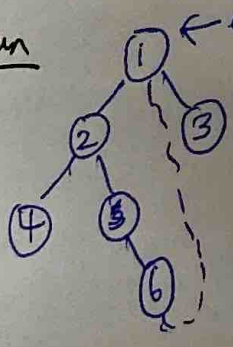
Q. Now, we have a doubt, that when we keep going left, then our threaded binary tree will again come back to root eventually. In this case, how will we determine that we don't have to go left again, we have to move right?

Sol: Case 3 - If threaded binary tree already exist then we'll remove thread,

then we'll move right

i.e. cur = cur -> right

Dry Run

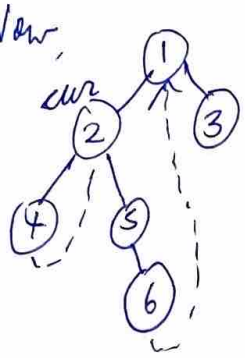


Now, for inorder traversal, we'll have to go to left.

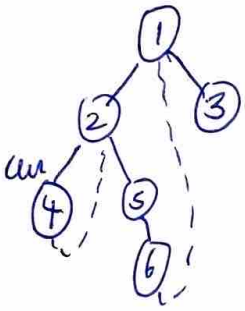
Before going left, we'll make the rightmost ptr in the left subtree & point it to cur.

Once the thread is created, move left

Now,



Again, create a thread from last guy in left subtree to cur node
& move cur = cur -> left



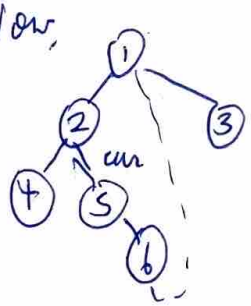
Now, (4) don't have a left, then this guy will be the root itself.
In this case, print 4

Now, we'll go back to the root via the thread we created.

the moment we go back to root, it has to be printed.
We'll print it.

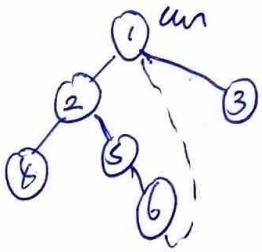
Now, this is the scenario where we will check whether a thread exists or not in the left subtree. If it exists, then we'll go left remove thread & once thread is removed, we'll move cur = cur -> right.

Now,



cur is at (5)
5 has no left, which means this node itself is a root
So, print 5 & go to right
Now, cur at (6). Print it,

Now, Go to root via thread



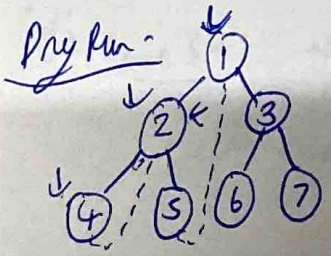
Print root.
Now, check whether left thread exists,
if yes, then we'll remove the thread
then move right & print it

Code →

```
cur = root;
while (cur != NULL) {
    if (cur->left == NULL) {
        ans.add(cur->val);
        cur = cur->right;
    }
    else {
        TreeNode* thread = cur->left;
        while (thread->right && thread->right != cur) {
            thread = thread->right;
        }
        if (thread->right == NULL) {
            thread->right = cur;
            cur = cur->left;
        }
        else if (thread->right == cur) {
            thread->right = NULL;
            ans.add(cur->val);
            cur = cur->right;
        }
    }
}
return ans;
```

Inorder - 4 2 5 1 6 3 7 ✓

Dry Run:



cur = ①, thread = ②

② → right = ① , cur = ②

cur = ② thread = ①

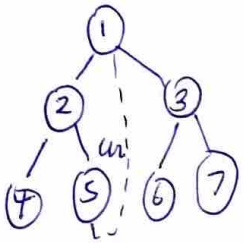
① → right = ② , cur = ④

④ → left == NULL , print 4

cur = ②

thread = ④, thread → right = NULL , print ②

cur = ⑤



⑤ cur -> left == NULL

print ⑤, move cur to ①

Now, cur = ①

cur -> left != NULL

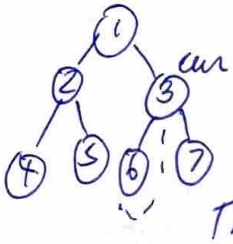
thread = ②, ~~thread = thread -> right~~

thread = ⑤, now

thread -> right == cur, i.e. ⑤ -> right == ①

~~⑤~~ ⑤ -> right = NULL, print ①, cur = cur -> right

Now, cur = ③



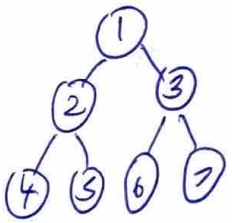
Again follow same procedure.

This will get us the inorder traversal.

TC - O(N) SC - O(1)

~~Now~~

Now, for Preorder Traversal →

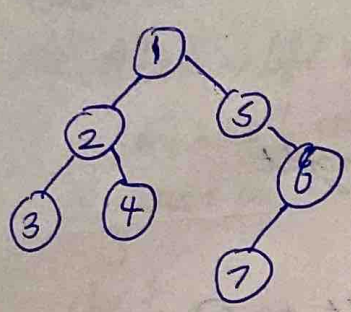


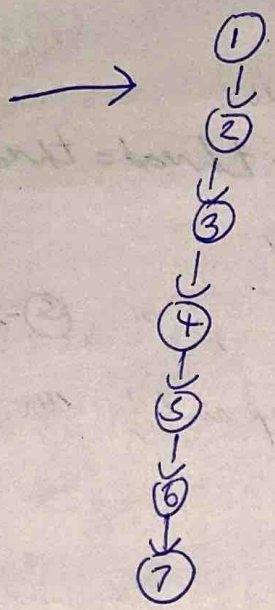
Preorder ⇒ 1 2 4 5 3 6 7

Only thing we need to change in code is that when we are pointing thread -> right = cur, then only we'll print the cur value.

Then move cur = cur -> left.

Flatten a Binary Tree to Linked List





Preorder -
1 2 3 4 5 6 7

We don't have to make a separate LL
We have to rearrange the nodes such that
① → right = ② , and so on

Recursive Approach -

Right Left Root

We'll arrange right subtree first, then left subtree then root

```
prev = null
flatten (node)
{ if (node == X) return;
  flatten (node -> right)
  flatten (node -> left)
  node -> right = prev;
  node -> left = null;
  prev = node
}
```

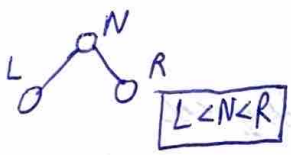
TC - O(N)
SC - O(N)

Iterative Approach

We'll use stack

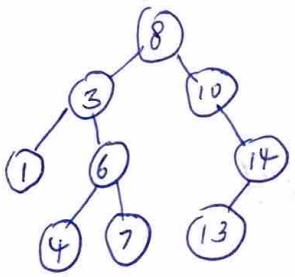
```
st.push(root)
while(!st.empty()){
  cur = st.top(), st.pop();
  if(cur->right) st.push(cur->right)
  if(cur->left) st.push(cur->left)
  if(!st.empty()) cur->right = st.top();
  cur->left = null;
```

Binary Search Tree -



- Left subtree → BST
- Right subtree → BST

Eg:



Q. Are duplicates allowed?

Ans - No

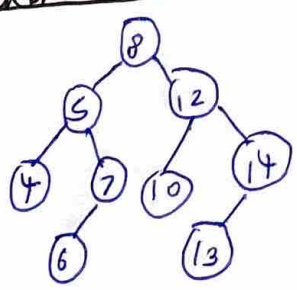
But if someone still insists on inserting duplicates, then we'll modify this condition of BST, i.e. L <= N < R

or we can store (node, freq) together

Why BST?

Ans - In most BST, height is always $\log_2 N$

Search in a BST



node = 10

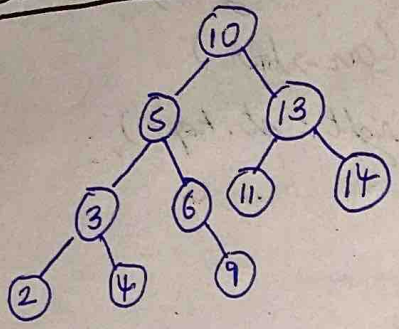
start at (8), 8 < 10, so move cur -> right

Now (12) > 10, so move cur -> left

node = (10), return true

We are not traversing the entire tree to search for an element, so we can say max Time Complexity to search is $O(\log N)$

Ceil in a BST



Suppose we are given a key = 8
We have to find node val >= key
But that val should be smallest.
E.g Here it is 9

key = 8

Approach:

Initialize a variable ceil = -1

Start with cur = 10.

Now cur > key, change ceil = 10
& move left

Now cur = 5

Now cur < key, don't modify ceil
& move right

Now cur = 6, again go right

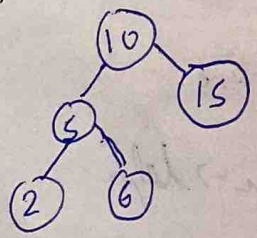
Now cur = 9, cur > key, update ceil
go left

Now, cur = NULL, we can say 9 is the answer
Return ceil

TC - $O(\log N)$

Floor in a BST

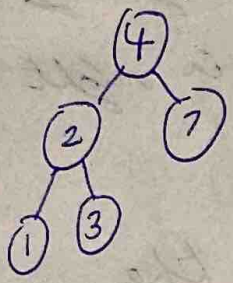
key = 7, We have to find greatest value which is <= 7



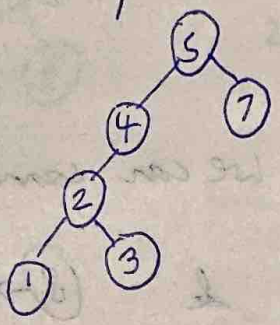
Ans - 6

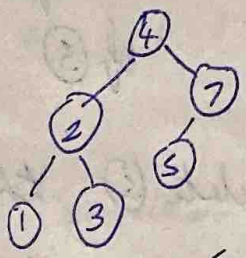
Same approach as above

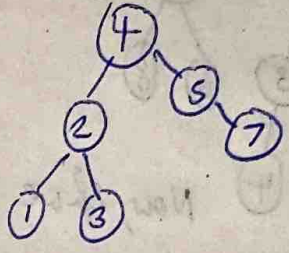
Insert a given node in BST



node = 5,
Multiple trees are possible



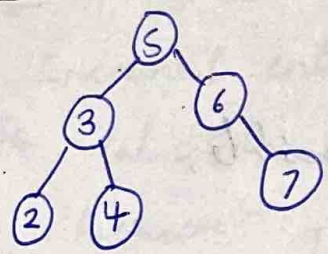




Approach-
We'll insert (5), at whichever leaf is possible.
Find where it can be inserted then insert
We'll try to find a leaf position, because it is easier to insert in leaf.

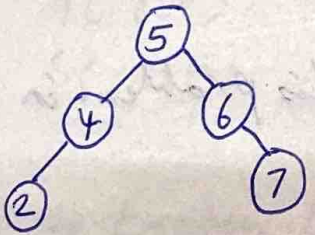
$TC - O(\log_2 N)$

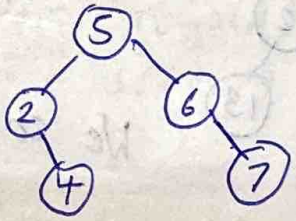
Delete a Node in BST



node = 3

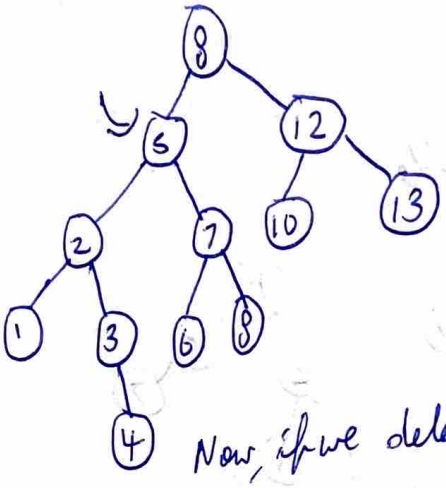
If we delete (3), then we have 2 possible trees





There can be multiple answers

Approach- We have to do deletion in parts, i.e.,
- First, we will search whether mentioned node exist in the tree or not. (SEARCH)
- Then we will delete

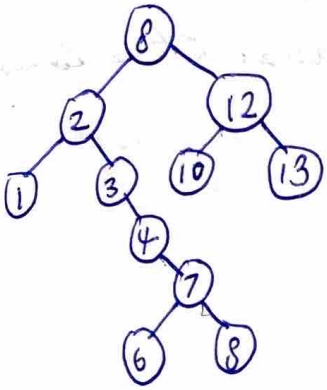


node = 5

Now, we can observe that everything in left of (5) < everything in right of (5)

Now, if we delete (5), then we can connect the subtree to (4) -> right & (8) -> left should become (2)

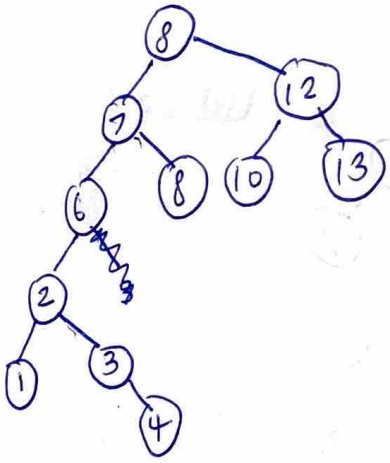
The tree should look like this,



Now, other way of solving, if we delete (5) & Put (8) -> left ~~ϕ~~ = (7) & its subtree.

& in the left of (6) we can attach (2)

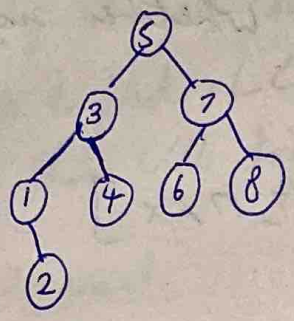
The tree should look like this



So, 2 ways are possible.

We can do this problem in whatever way

K^th Smallest Element in BST



k = 3

1 2 3 4 5 6 7 8

↑

This is 3rd smallest

Naive Approach- Do a recursive DFS traversal and store all the nodes in a vector. Then we can easily find k th smallest element.

TC - O(N) + O(N log N)

SC - O(N)

Efficient Approach-

Generally, inorder traversal of BST is in sorted order. We will do inorder traversal, maintain cnt = = 0.

Increment cnt whenever we encounter a node.

When cnt = = k, Return that node

Recursive - TC - O(N) SC - O(N)

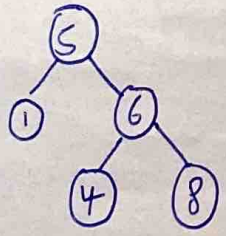
Morris - TC - O(N) SC - O(1)

K^th largest → Do one traversal to count no. of nodes in BST.

Then K^th largest = (N - K^th) Smallest

So, 2 traversals required here.

Validate a BST



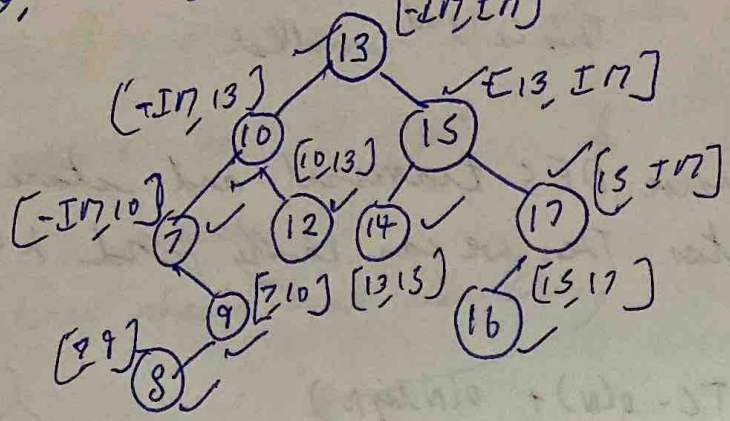
This is not a valid BST, because 4 < 5

Approach-

For a node to be valid, it has to be < (6) & > (5). So, we will

provide a range to particular node,
e.g. [5 6], then we can see whether node lies in particular range or not,

Initially, let range be [INT_MIN, INT_MAX]



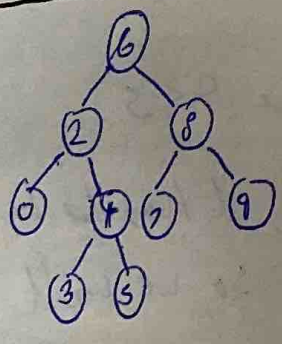
We will check whether, (13) lies in range or not. If yes, then it is a valid node. Update range and move left & right.
For left, range will be [-IM, 13) for left child
& [13, IM] for right child.
Now (10) is lying in range
Hence, this tree is valid

TC - O(N) SC - O(1)

To initialize long int min & long int max →

long int minVal = numeric_limits<long int>::min();
" " maxVal = numeric_limits<long int>::max();

LCA in BST



LCA(5, 0) = (2)
LCA(2, 5) = (2)

Naive Approach

* Simply ~~traverse~~ do the same approach, as we did to find LCA in binary tree.
But this will take O(n) time complexity.

Better Approach

There are 3 possibilities for 2 nodes to exist that are,

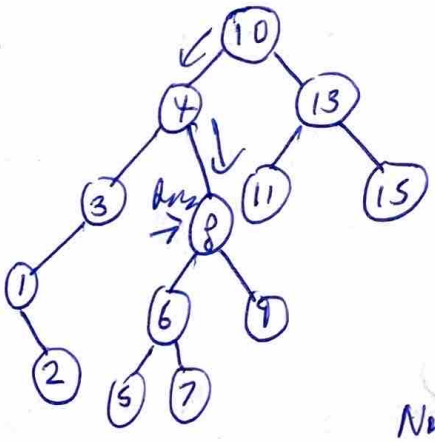
Both on left

Both on Right

One left & One right

In last case, we can say that the point where they split is our LCA.

E.g.



LCA(5, 9)

Start from root

(5 & 9) < (10), move left

(5, 9) > (4), move right

Now, 5 < (8) & (8) < 9, which means

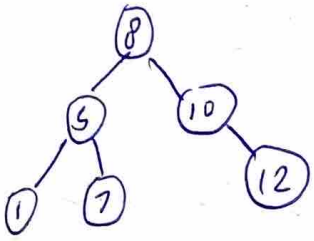
This is our LCA.

So, the point where we split, is our LCA

TC - O(H) SC - O(1)

Construct BST from preorder traversal

preorder → {8 5 1 7 10 12}

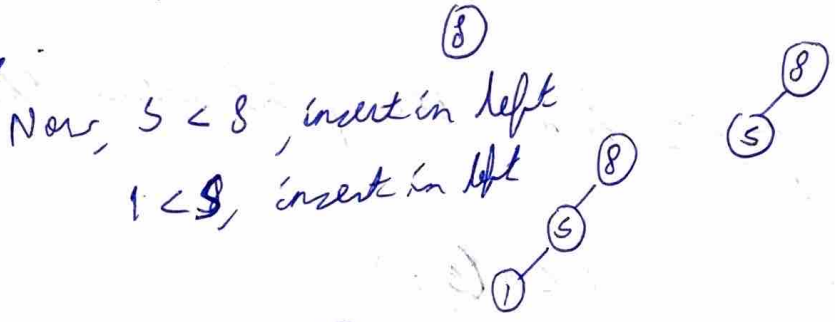


Naive approach

Naively create Binary tree

As we know (8) is root, because first element is root. So, insert root

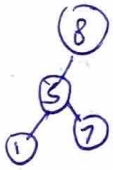
Now,



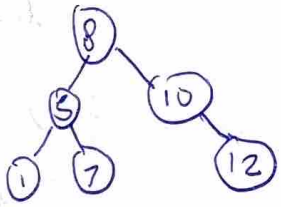
Now, 5 < 8, insert in left

1 < 8, insert in left

7 < 8 & 7 > 5,



And similarly we get



TC- O(N) x O(N)

Better Approach

Sort the given preorder traversal, which will eventually become inorder traversal of BST

Inorder → {1 5 7 8 10 12}

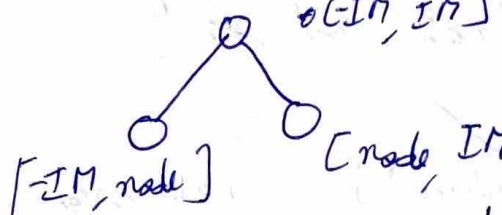
Now, with inorder & preorder, we can create a BST, just like we created a Binary tree

TC- O(N log N) + O(N)

SC- O(N)

Optimal approach

We'll use the same approach we used to Validate a BST, i.e., we'll use INT_MIN & INT_MAX



But, in this approach, we don't need the Lower Bound Range, We can just solve with INT_MAX

Upper_Bound → ub = INT_MAX

Initially,

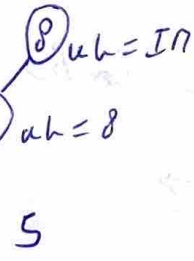
preorder → [8 5 1 7 10 12]

node = 8, is 8 < ub, Yes it is

Now, insert (8).

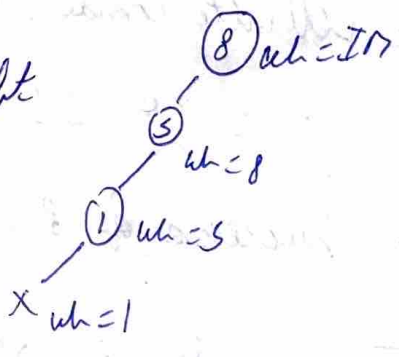
Now, go left & update ub = 8, we can observe that 5 1 7 are < 8.

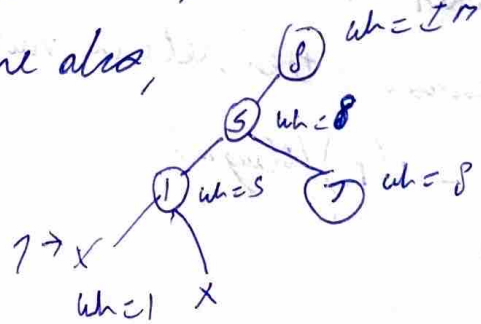
Now, ele = 5, 5 < ub, yes we can insert, update ub = 5



Now, ele = 1, insert & update ub = 1, Now, again go left

Now, ele = 7, it is not less than (ub=1) so return NULL, now it goes back to (1) & ub=5 here, it cannot be inserted here also, so return NULL





This way we'll keep inserting all the elements.
We just have to maintain upper bound,

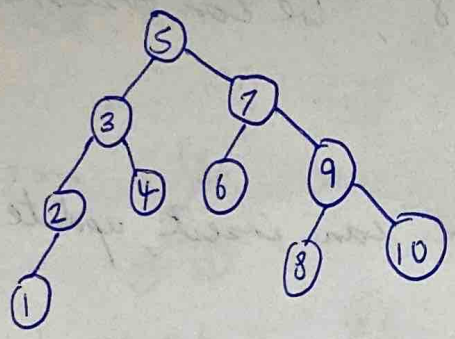
(node, ~~INT_MAX~~ upper bound)

(node->left, node->val) (node->right, @ bound)

We'll also have to maintain at which element we are in the array, then return.

TC - O(3N) ≈ O(N)
SC - O(1)

Inorder Successor in BST



val = 8

We have to find inorder successor in BST

First we will write inorder traversal →

1 2 3 4 5 6 7 8 9 10

Inorder successor of 8 is 9 here

Brute Force-

Store inorder in vector & then find successor
If no successor then return null.

TC - O(N) + O(N)/O(log N)
SC - O(N)

Better Optimal approach

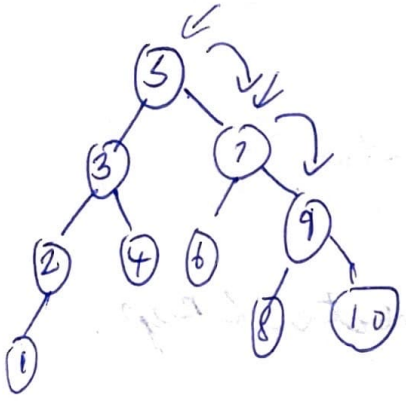
Do inorder traversal, and the first value which we encounter > 8, is our answer

$TC - O(N)$
$SC - O(1)$

We will optimize Time to $O(H)$

Optimal approach

Initialise a variable successor = NULL.
First node is (5), which is < 8, so ok this can't be successor & $8 > 5$, so move right.



Again on 7, $8 > 7$, move right & successor is still NULL.

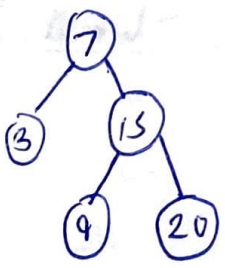
Now, node is (9), $9 > 8$, successor = 9
& $8 < 9$, move left

Now, node is (8), when we reach the node itself we'll go to its right, if its null then return successor

$TC - O(H)$ $SC - O(1)$

BST Iterator

BST iterator (7) — This will be root
next - Initially next ptr points to NULL. The moment we ask next it should return its next node in inorder traversal.



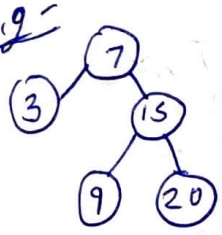
Inorder - 3 7 9 15 20

next (initially) next

When next is called it will return 3 & move to 3

hasnext - return True or False whether next elements or not.

E.g:



3 7 9 15 20

BST iterator (7) — represents root

next → 3
next → 7
hasnext → True
next → 9
hasnext → True
next → 15
hasnext → True
next → 20
hasnext → False

So we have to design this BST iterator

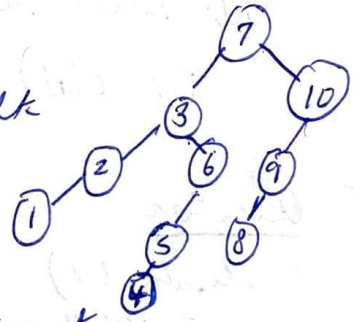
Naive Approach

Store the inorder traversal in a vector & keep next pointer at arr[0].

TC - O(1) + O(N)
SC - O(N)

Better

Inorder is → Left Root Right



SC - O(H)
TC - O(1)

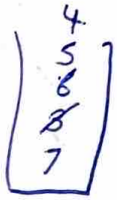
We will have to create a constructor BST iterator (7), which will be initially the root

We will take a stack.
Go to the extreme left & put everything in stack

next - Return st.top() this will be our next, (1 gets popped)
After popping go to its right & push elements

next - Returns 2
1 & 2 has no right pointer, so we didn't push anything

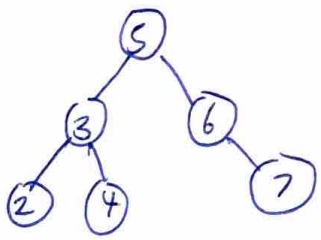
next - Returns 3
Now, 3 has right, go to right & put everything in stack



We'll follow the same procedure.

TC - O(1)
SC - O(H)

Two Sum in BST



K = 9

5+4 = 9
3+6 = 9
2+7 = 9 ✓

Yes there exist 2 node to solve this

Brute Force -
Write inorder traversal in vector. Then perform normal 2 sum approach using 2 pointers

TC - O(N) + O(N) SC - O(N)

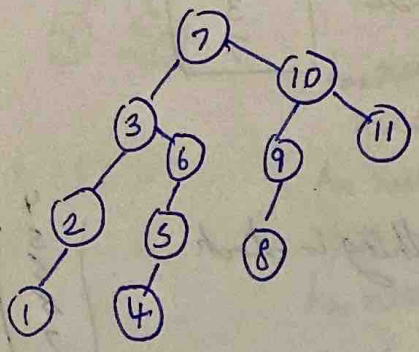
Optimal - We'll follow some approach we used in BST iterator.
We will take 2 stacks to determine next() & before()
element, i = next() j = before()

We will determine before() using opposite approach, we used to determine next().

i.e. We will keep going node->right & push it in stack.

Now, i = next() & j = before()

k = 16

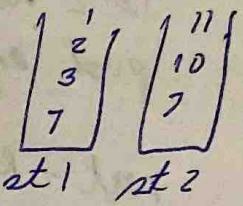


In this case,

i = 1 & j = 11

i + j = 12 < 16,

So, we have to increase the value



To increase the value we'll increase the next() guy.

Now, i = 2 & j = 11 < 16

i = 3 & j = 11 < 16

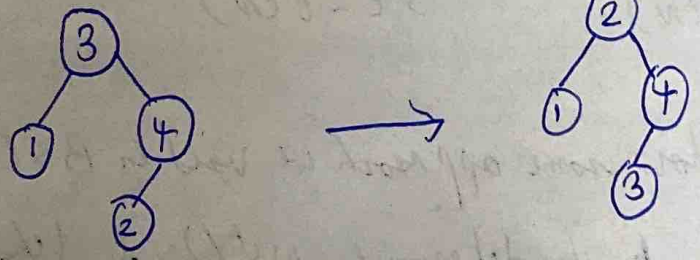
Again i = 4 & j = 11 < 16,

Again i = 5 & j = 10 == 16. Return true

TC - O(N) SC - 2 x O(H)

Recover BST

Two nodes are swapped in given BST, we have to swap them to recover our BST,



Brute Force-

Do inorder traversal & store it in vector

Then sort the traversal. Simply

Now simply traverse the tree again & compare nodes values with values inside tree.

If not same then change

TC - 2N + N log N

SC - O(N)

Better Soln

Write the inorder of the given incorrect BST.

This will take O(N) time

Now, as we know 2 nodes are swapped in the sorted array.

So, we have to figure out the swapped elements

Case I - Swapped nodes are not adjacent

3 25 7 8 10 15 20 5

Case II: Swapped nodes are adjacent

3 5 8 7 10 15 20 25

We don't need to store the inorder traversal in any array, we just have to figure out what nodes are swapped.

Now, in case I; start traversing

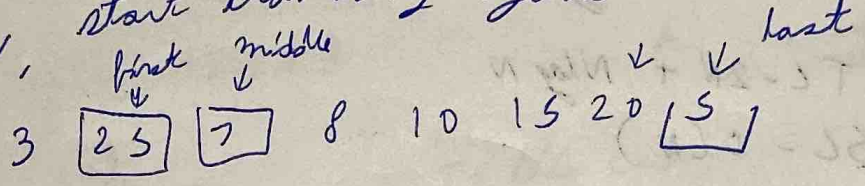
3 25 7 8 10 15 20 5

Now as soon as we see that 25 > 7, or 7 is not greater than prev element.

So, we can mark 25 as the first node which needs to be swapped.

We'll also store 7 as second element which is violated/swapped.

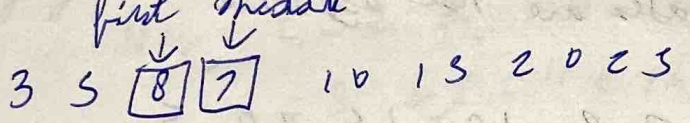
Now, start traversing again



We get 5 is < 20. this is second violation.
We'll store 5.

Now, we'll simply ~~swap~~ swap 25 & 5

Now Case II - If there is no second violation

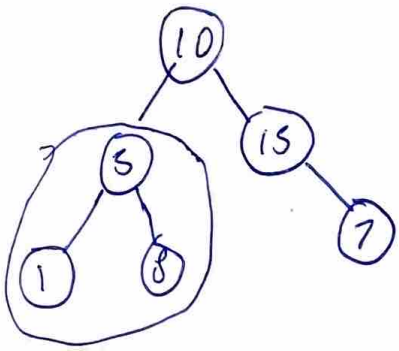


We'll not find any 2nd violation in this case.
So, we'll simply swap 7 & 8.

We only swap first & last if both are found.
If not found, we only swap first & middle.

TC - O(N)
SC - O(1)

Largest BST in Binary Tree



largest BST is of size = 3

Brute Force

We apply the same approach we used to Validate a BST.

TC - $O(N^2)$

SC - $O(N)$

Better approach -

If we can find largest ele in the left of node & smallest ele in the right of node

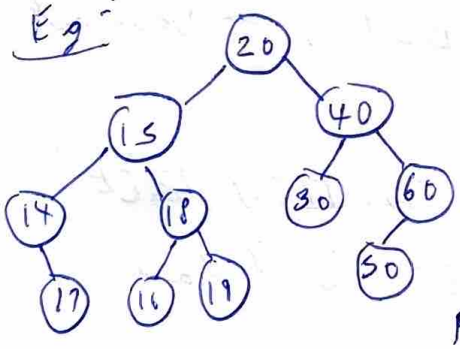
largest < node < smallest, then it is a Valid BST.

And for size we say, left subtree has x elements & right has y elements. Then total size = 1 + x + y.

(largest, smallest, size)

We'll follow Postorder traversal to go left, right & then compute root.

Eg:



For a leaf node, we can say it is definitely a BST & its size = 1.

We go left & reach (14), (14) → left == NULL. So, we go right

Did we reach node (17)

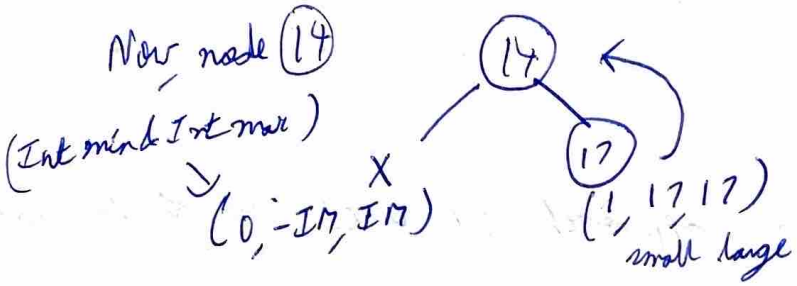
we update our values

(17, 17, 1)

large small size

Now, node (14)

(Int min & Int max)

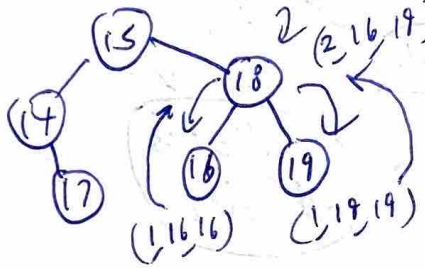


This will return to 14.

14 will compare both values & update its values accordingly. Now, 14 will become smallest & update size

At (14) (2, 14, 17)

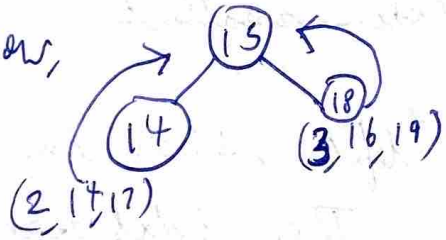
Now, left of 15 is computed, we'll go right.



(2, 16, 19)

We'll move to 18

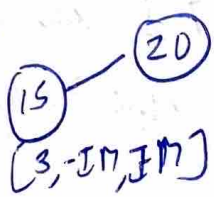
Now,



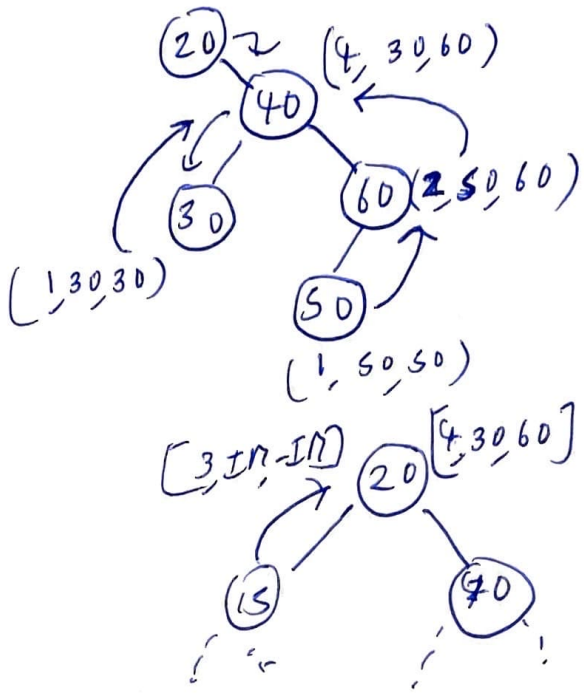
As largest in left is 17 which is not less than 15, so, this is not a valid BST, so we'll not update size, we'll not do size+1.

And we'll pass INT MAX & INT MIN to it

20 → left is not a valid BST. We'll have to look in right only.



Now we'll compute 20 → right



Yes it is a valid BST & its size = 4

20 gets 2 values

So, max size = 4

So, if valid bst, size = left + right + 1

if not, size = max(left, right)

& update largest & smallest.

TC - O(N) SC - O(1)

Strings

Remove Outermost Parentheses

(()()) (()) → ()()()

(()()) (()(())) → ()()()()(())

()() → "" empty

Approach:

We will maintain a cnt variable while traversing the string, whenever we get '(' we increment cnt & whenever we get ')' we decrement cnt.

And in each loop we will check whether cnt != 0 then

ans += s[i];

E.g:

(()()) (())

cnt = 0, ans = ""

↓

(()()) (()) cnt = 1, ans = ""

2, ans = "("

1, ans = "()"

2, ()(

1, ()()

0

1, ()()()

2

1

0

Reverse words in a string →

" hello world! " ⟶ "world! hello"

Approach -

1) Start Using 2 pointer approach.

- For leading and trailing blank spaces we'll use a inner while loop which will keep incrementing until i does not encounter a non-space character.
- As soon as i encounter non space character, we will initialize j = i + 1. And move j until j again encounters a space character. This space will embark end of 1 word.
- Now we will store string word = s.substr(i, j - i)

* substr takes 2 parameters (i, length of substring)

And we will add this result to the front of result/ans

ans = word + " " + ans;

Dry Run

" ---hello_world!--- "
 i

i = 0 res = " "

- blank character (space)

i will keep incrementing until it encounters first alphabet

" ---hello_world!--- "
 i ↑ j

Now, initialize j = i + 1 & increment j until space

" ---hello_world!--- "
 i ↑ j

word = "hello"

res = word + res = "hello"

Move i to j + 1

Again follow same procedure

T.C - O(N) S.C - O(N) i.e. Auxiliary space for answer & substring

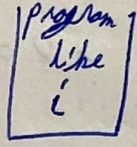
In the previous approach, we will get "memory Limit Exceeded error", because we have to solve this in O(1) constant space.

Approach II - This approach also, taken extra space i.e. O(N) space.

We'll use a stack in this approach.

E.g. - " i like program "

Push all these in stack in the traversal



Then in second traversal pop the strings from stack.

Optimal O(1) approach -

1) ~~Store~~ the words in vector<~~string~~>.

2) Traverse ~~backward~~ in the vector and store the words in ~~string~~ one with extra space appended to each string except the first ~~string~~ (i == 0)

Store the answer in original string itself.

Initially before entering the loop do s += " ";

Then in loop do s += s.substr(i, j - i)

After loop

s = s.substr(len+1, s.size() - (len+2));

return s

TC - O(N) SC - O(1)

Largest Odd No in String

		o/p
num = "52"	→	"5"
num = "4206"	→	""
num = "35427"	→	"35427"

Approach-

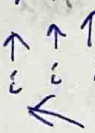
Last digit in string decides whether the given string is odd or even,

e.g. - "11740" is even

"70089S" is odd

So, we'll traverse the string from Right to Left

e.g. - 11740 If even decrement i



← Here it is odd, so extract the substring till current i position.

e.g. - 700895 odd digit then return whole string

Now we don't need to convert char to int to check whether a digit is odd or even.
We can check with ASCII value also.
Ascii value of even digit is even.

Longest Common Prefix

I/p → ["apple", "appx", "apc"]

o/p → "ap"

Brute Force

T.C - O(N) * O(min string length)
S.C - O(1)

Approach-

We will compare 2 strings at once. And then what substring is common in them. We will compare that substring with remaining strings.

compare (s1, s2)

Isomorphic strings -

Two strings s & t are isomorphic if the characters in s can be replaced to get t.

Eg: s = "egg", t = "add"
o/p - true

s = "foo", t = "bar"
o/p - false

2 characters may not map to same character, but a character may map to itself.

Approach -

If all characters in both strings are One-to-One mapped, then these strings are isomorphic

We will use 2 maps, one will map char - char
& other will map char - bool

There are 2 cases in which our soln can fail

Case 1: f o o
b a r
o is mapped with 2 char
i.e. a & r

Case 2: f o r
b o b.
b is mapped with 2 char i.e. f & r

For case 2, we will use a second map which will check whether a char in t string has already been mapped or not.

Find if one string is rotation of another string.

s = "abcde" goal = "cdeab"

O/P → true

Approach - If left rotation has been done by k places, then by right rotation of n-k places, we can get original string.

Method - We do

str temp = s + goal;

Concat string with original string

i.e. s = abcdeabcde

Then start traversing this string and look for goal.

If found, then return true.

We can search using

```
if (temp.find(goal) != string::npos)
    return true;
else
return false;
```

Valid Anagram -

Given 2 strings s & t, return true if t is an anagram of s, and false otherwise.

e.g. - s = "anagram", t = "nagaram" O/P → TRUE

s = "rat", t = "car" O/P → FALSE

Sort characters by Frequency

S = tree
O/P → eetr / eetr

S = Aabb
res = bbaA / bbAa

Approach -
We will use Hashmap & then sort the hashmap according to values

TC - len(string) + S2 log S2

(S2 for S2 alphabets uppercase & lowercase

use a vector<pair<int, char>> hash('z'+1) {0, 0}

Insert frequency & then sort in ascending order.

Roman to Integer

Use a map to map roman characters to their corresponding values.
A special cond we need to take care of is that if smaller value precedes larger value then we subtract smaller value from ans. Not add it.

E.g - IV → -1 + 5 = 4

TC - O(N) SC - O(2*7) ≈ O(1)

Integer to Roman

9 → "IX"

We will map all values i.e. M, CM, D, IV, IX, etc to their corresponding values.

Start traversing the map

While num >= mp.first

ans += mp[i].second

num -= mp[i].first

string ans = ""

Count Number of Substrings with Exactly K distinct characters →

Input : abc k = 2

o/p → 2

Substrings are {"ab", "bc"} with 2 distinct characters

Input → aba k = 2

o/p → 3 Substrings are {"ab", "ba", "aba"}

Input → aa k = 1

o/p → 3 Substrings are {"a", "a", "aa"}

Brute Force Approach -

Generate all substrings which will take $O(n^2)$ time

Then for each substring count distinct characters in it.

Total TC - $O(n^3)$.

Better Approach -

We'll maintain a hash table while generating substring and check the number of unique characters using that hash table.

Optimal Approach -

We will use 2 hashmaps m1 & m2

"abc abd abbcfa" k = 3

m1 hashmap - data hashmap (till k-1 characters)

m2 hashmap - data hashmap (till k characters)

↓ - adding in m1 hashmap

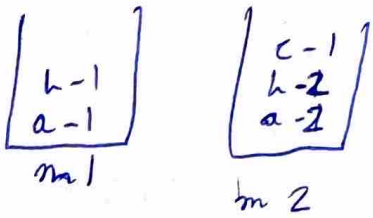
↙ - adding in m2 hashmap

↑ - release from hashmap / pop

Here, k=3, m1 hashmap will go till k-1 i.e. 2 characters, it means it will store only 2 characters at max

Now, let's do dry run

a b c a b d a b b c f a



We will keep pushing characters in m1 until its size != k-1

& we will keep pushing in m2 until size != k

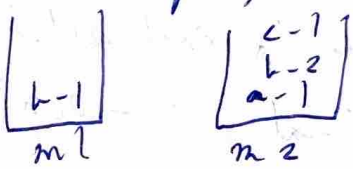
Here, size of hashmap denotes how many distinct characters it has in it.

Now we have 3 substrings with k unique characters in window

that are, "abc" "abca" "abcab"

Now, we'll release first character from both maps, i.e. delete it

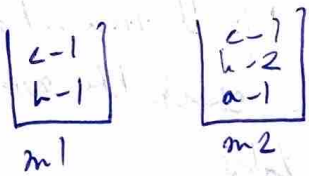
a b c a b d a b b c f a



Now our m1 is invalid as its size != k-1,

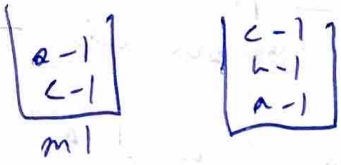
So, insert next character in m1

Now, we get 2 strings, "bca" & "bcab"



Now again move character pointer to next i.e. pop b from both

a b c a b d a b b c f a



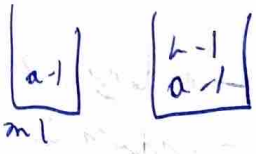
m1 size again < k-1, So, push next char

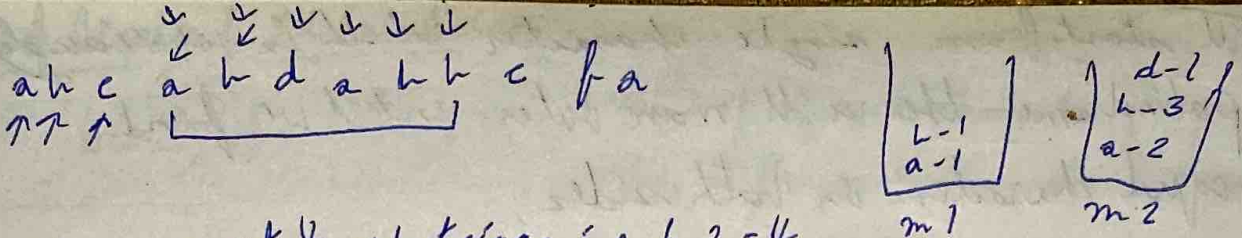
Now strings are "cab"

Now release c from both maps.

Both maps got invalid as

push next characters until size = k in m2





Now we get 4 substrings, i.e. 6-2=4 in this window
That are "abdabb" "abdab" "abda", "abd"

Now move a

X

Longest Palindromic Substring -

I/P → "babad" O/P → "bab" / "aba"

Brute Force - Generate all possible strings, and check whether it is a palindrome or not.
So, TC - $O(n^3)$

Better - Earlier we used to determine palindromic string by starting from edge characters, if they match then we moved inwards, e.g. b o b

But here we will do it opposite way, i.e. b o b

we'll start from single character which is obviously a palindrome, then we'll move outer until we find equal characters on both sides.

Dry Run- "babad" longest = ""
i↑

First "b" as center, we cannot move left as this is longest palindrome & increment i to next char

"babad" longest = "b"
i↑

Now 'a' is center, move left & right until character match,
Now, "bab" is longest. longest = "bab"

Now, "babad"
i↑

"aba" is a palindrome
So, we can update longest or keep it same it's on us.

But, this method only calculates palindrome of odd lengths.
What if our palindrome is of even length?

Eg- "cbbd" We'll modify our code a little bit to handle this edge case
O/P → "bb"

To handle this edge case, we will initialize left = i & right = i+1
Whereas to check for odd length, we do left = i & right = i

Let's do a dry run

"cbbd" l=0 r=1 longest = "c"
i↑

"cbbd" l=1 r=2 longest = "bb"
i

TC - $O(n^2)$
SC - $O(1)$

Sum of Beauty of All Substrings -

Beauty of a string is = Most frequent character frequency
- Least frequent character frequency

E.g = Beauty of "abaacc" = 3 - 1 = 2

So, given a string s, return sum of beauty of all of its substrings

I/P → "aabcb"

Beauty of substring aab = 2-1 = 1
" " " aabc = 2-1 = 1
" " " aabcb = 2-1 = 1
" " " abcb = 2-1 = 1
" " " bcb = 2-1 = 1

Sum = 1+1+1+1+1 = 5 ← O/P

Approach -

Generate all substrings and insert the characters in map. And then add a third loop to traverse the map which will determine max & least frequency

TC - $O(n^2) * O(26) \approx O(n^2)$
SC - $O(26) \approx O(1)$

```
for (i = 0 → n) {
    map<char, int> mp
    for (j = i → n) {
        mp[s[j]]++; int mf = 0, lf = INT_MAX
        for (auto i : mp) {
            mf = max(mf, i.second)
            lf = min(lf, i.second)
        }
        ans += (mf - lf);
    }
}
```

Better approach -

We can use a multiset instead of using map because we only need to know frequency, not character. Map would have failed if the question was of mapping integers.

multiset < int > ms; (Sorted & can store duplicates)

First ele of multiset - lowest frequency
Last " " " - highest "

(Frequency can be same of 2 characters so we take multiset)

So, we won't have to traverse whole set to find max & min frequencies.

```
for (i=0 -> n) {
  map< char, int > m;
  multiset < int > st;
  for(j = i -> n) {
    if(m.find(s[j]) != m.end())
    st.erase(st.find(m[s[j]]));

    m[s[j]]++
    st.insert(m[s[j]]);
    ans += (*st.rbegin() - *st.begin());
  }
}
```

We we delete previous frequency of a char if it has already been inserted in set, Because we have to update new frequency in set

Rabin Karp Algorithm

If a string is given, and a pattern is given. We have to determine whether pattern exists in string or not.

Text: a a a a a b

Pattern: a a b

Approach: We'll not compare each & every alphabet in this algo. In pattern, we see 3 alphabets, instead of checking single character each time, we'll make it as a single value. We'll sum it up with ASCII values, but for the sake of understanding this algo easily, we'll make our own values

a-1 g-7
b-2 h-8
c-3 i-9
d-4 j-10
e-5
f-6

Pattern: a a b

1+1+2 = 4

1+1+2 ← This is hash f'n h(p)

4 ← This is hashcode

We use this hash f'n to find pattern inside this text.

Text: a a a a a b

n=6 1+1+1=3

Depending on size of pattern (here m=3), we'll take 3 letters in text.

Pattern: a a b

m=3 1+1+2=4

⊗ Both values are not matching, using constant time we found value for above 3 characters.

Then we'll move further until we find 4 in Text

a a a a a b

3 → 4

3 3 2

This sliding is Rolling Hash F'n.

Eg:

Text: a b c d a b c e

n = 8

6

6 - 1 + 4 = 9

9 - 2 + 1 = 8

8 - 3 + 2 = 7

6

10

Pattern: b c e

m = 3 2 + 3 + 5 = 10

When we slide our window further, subtract removed character & add next character ASCII value in the total.

Drawback - We were going through the main string only once & we have checked for pattern only once,

So, TC - O(n - m + 1). This is Average case

But, in worst case scenario like below

Text: c c a c c a a c d b a

n = 11

3 + 3 + 1 = 7

3 + 1 + 3

pattern: d b a

m = 3 4 + 2 + 1 = 7

Both values match, we'll compare character, character do not match so we move our window

Again values match, we'll again compare character

So, if values come same, we'll have to compare pattern everytime.

These are called as Spurious Hits.

And Max Time - O(m * n)

This is happening because of very simple hash fn.

Now we'll study hash fn by Rabin Karp.

Main algo -

We define a hetter hash fn.

pattern : d h a

$4\times10^2 + 2\times10^1 + 1\times10^0 = 400 + 20 + 1$

Here we have taken 10 because we are considering alphabets from a - j i.e. 1 - 10.

It is completely dependent on us, what value we take as base to define hash fn.

Text: c c a c c a a e d h a

$3\times10^2 + 3\times10^1 + 1\times10^0$

$300 + 30 + 1 = 331$

Now to move the window forward

$[3\times10^2 + 3\times10^1 + 1\times10^0] - 3\times10^2$

$[3\times10^1 + 1\times10^0]\times10 \longrightarrow 3\times10^2 + 1\times10^1$

$3\times10^2 + 1\times10^1 + 3\times10^0$

Similarly on moving forward we'll find 'd h a' pattern

Now TC - $O(n-m+1)$ avg case

TC - $O(mn)$ worst case

But now chances of worst case are very less

How to code?

Hash at next shift must be efficiently computable ($O(1)$) from the current hash value and next character in text

Rolling hash fn:

$$hash(txt[s+1 \ldots s+m]) = d(hash(txt[s \ldots s+m-1]) - txt[s]*h) + txt[s+m] \ \%\ q$$

txt[s+1] - next char

txt[s] - curr char

d - No of characters in the alphabet

q - A prime no.

h - $d^{(m-1)}$

m - length of pattern.

Simply written

$hash(s[i+1]) = d(hash(s[i]) - s[i] * h) + s[i+m] \% q$

d → 256 q → Prime no. (here 11)

```
int p = 0;  // hash value for pattern
int t = 0;  //  "    "   "  "  text
int h = 1;
```

Calculate h - `for (i=0 → m-1) h = (h*d) % q;`

Calculate hash value for both strings -

```
for (i=0 → m)
    p = d*p + pat[i] % q;
    t = d*t + txt[i] % q;
```

Now we'll search in main string

```
for (i = 0 → N-m)
  if (p == t) {
    for (j = 0 → m) {
      if (txt[i+j] != pat[j]) break;
    }
    if (j == m)  Found
  }
  if (i < N-m) {
    t = (d*(t - txt[i]*h) + txt[i+M]) % q;
    if (t < 0) t = (t+q);
```

Naive String Matching Algorithm

String: a b c d e f g h

pattern: d e f

We have to search for the pattern in the string.

Sliding window of size m, where m is length of pattern.

If it matches then return index, otherwise shift the window towards right

Eg 2

str a b c d a b c a b c d f
1 2 3 4 5 6 7 8 9 10 11 12

We'll use i & j here

pat a b c d f
1 2 3 4 5

str[i] == pat[j]
i++, j++

At index 4, there is a mismatch

So, now j will go at index 1, i.e. starting
But i will go at index 2, i.e. starting + 1

Now, (mismatch) (from here it starts matching)
a b c d a b c a b c d f

So, this i going back to index 2 is a waste of time. This is a drawback of Naive algo.

a b c d f

Again pat[j] ≠ str[i]

i will move to index 6, and j will move to start of pattern.
i will move to that index+1, where both characters matched for the first time.
So, drawback of this algo is that some characters are compared repeatedly

TC - O(m * n)

KMP Algorithm (Knuth - Morris - Pratt)

We'll find prefix & suffix of a pattern

Pattern: a b c d a b c

prefix: a, ab, abc, abcd, ...

suffix: c, bc, abc, dabc, ...