Binary Search tree

# 197. Search and Insert in a BST

**Searching a key**

// C function to search a given key in a given BST

struct node\* search(struct node\* root, int key)

{

// Base Cases: root is null or key is present at root

if (root == NULL || root->key == key)

return root;

// Key is greater than root's key

if (root->key < key)

return search(root->right, key);

// Key is smaller than root's key

return search(root->left, key);

}

**Illustration to search 6 in below tree:**   
1. Start from the root.   
2. Compare the searching element with root, if less than root, then recurse for left, else recurse for right.   
3. If the element to search is found anywhere, return true, else return false. 

bstsearch

**Insertion of a key**

A new key is always inserted at the leaf. We start searching a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. 

100 100

/ \ Insert 40 / \

20 500 ---------> 20 500

/ \ / \

10 30 10 30

\

40

// C++ program to demonstrate insertion

// in a BST recursively.

#include <iostream>

using namespace std;

class BST

{

int data;

BST \*left, \*right;

public:

// Default constructor.

BST();

// Parameterized constructor.

BST(int);

// Insert function.

BST\* Insert(BST\*, int);

// Inorder traversal.

void Inorder(BST\*);

};

// Default Constructor definition.

BST ::BST()

: data(0)

, left(NULL)

, right(NULL)

{

}

// Parameterized Constructor definition.

BST ::BST(int value)

{

data = value;

left = right = NULL;

}

// Insert function definition.

BST\* BST ::Insert(BST\* root, int value)

{

if (!root)

{

// Insert the first node, if root is NULL.

return new BST(value);

}

// Insert data.

if (value > root->data)

{

// Insert right node data, if the 'value'

// to be inserted is greater than 'root' node data.

// Process right nodes.

root->right = Insert(root->right, value);

}

else

{

// Insert left node data, if the 'value'

// to be inserted is greater than 'root' node data.

// Process left nodes.

root->left = Insert(root->left, value);

}

// Return 'root' node, after insertion.

return root;

}

// Inorder traversal function.

// This gives data in sorted order.

void BST ::Inorder(BST\* root)

{

if (!root) {

return;

}

Inorder(root->left);

cout << root->data << endl;

Inorder(root->right);

}

// Driver code

int main()

{

BST b, \*root = NULL;

root = b.Insert(root, 50);

b.Insert(root, 30);

b.Insert(root, 20);

b.Insert(root, 40);

b.Insert(root, 70);

b.Insert(root, 60);

b.Insert(root, 80);

b.Inorder(root);

return 0;

}

// This code is contributed by pkthapa

**Output**

20

30

40

50

60

70

80

**Illustration to insert 2 in below tree:**   
1. Start from the root.   
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.   
3. After reaching the end, just insert that node at left(if less than current) else right. 

bstsearch

**Time Complexity:**The worst-case time complexity of search and insert operations is O(h) where h is the height of the Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

**Insertion using loop:**

* Java

import java.util.\*;

import java.io.\*;

class GFG {

public static void main (String[] args) {

BST tree=new BST();

tree.insert(30);

tree.insert(50);

tree.insert(15);

tree.insert(20);

tree.insert(10);

tree.insert(40);

tree.insert(60);

tree.inorder();

}

}

class Node{

Node left;

int val;

Node right;

Node(int val){

this.val=val;

}

}

class BST{

Node root;

public void insert(int key){

Node node=new Node(key);

if(root==null) {

root = node;

return;

}

Node prev=null;

Node temp=root;

while (temp!=null){

if(temp.val>key){

prev=temp;

temp=temp.left;

}

else if (temp.val<key){

prev=temp;

temp=temp.right;

}

}

if(prev.val>key)

prev.left=node;

else prev.right=node;

}

public void inorder(){

Node temp=root;

Stack<Node> stack=new Stack<>();

while (temp!=null||!stack.isEmpty()){

if(temp!=null){

stack.add(temp);

temp=temp.left;

}

else {

temp=stack.pop();

System.out.print(temp.val+" ");

temp=temp.right;

}

}

}

}

**Output**

10 15 20 30 40 50 60

**Some Interesting Facts:**

* Inorder traversal of BST always produces sorted output.
* We can construct a BST with only Preorder or Postorder or Level Order traversal. Note that we can always get inorder traversal by sorting the only given traversal.

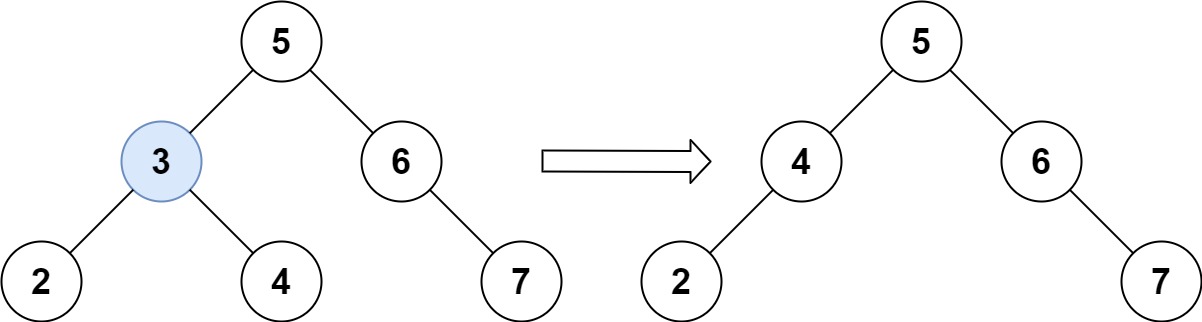
# 198. [Deletion of a node in a BST](https://leetcode.com/problems/delete-node-in-a-bst/)

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

**Example 1:**



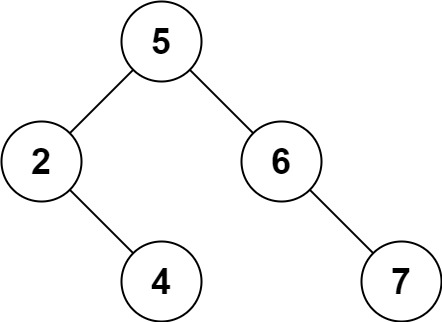
**Input:** root = [5,3,6,2,4,null,7], key = 3

**Output:** [5,4,6,2,null,null,7]

**Explanation:** Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the above BST.

Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.



**Example 2:**

**Input:** root = [5,3,6,2,4,null,7], key = 0

**Output:** [5,3,6,2,4,null,7]

**Explanation:** The tree does not contain a node with value = 0.

**Example 3:**

**Input:** root = [], key = 0

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -105 <= Node.val <= 105
* Each node has a **unique** value.
* root is a valid binary search tree.
* -105 <= key <= 105

# Solution:

When we delete a node, three possibilities arise.   
**1)*Node to be deleted is the*** ***leaf:*** Simply remove from the tree.

50 50

/ \ delete(20) / \

30 70 ---------> 30 70

/ \ / \ \ / \

20 40 60 80 40 60 80

**2) *Node to be deleted has only one child:*** Copy the child to the node and delete the child

50 50

/ \ delete(30) / \

30 70 ---------> 40 70

\ / \ / \

40 60 80 60 80

**3) *Node to be deleted has two children:***Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

50 60

/ \ delete(50) / \

40 70 ---------> 40 70

/ \ \

60 80 80

The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

**My Implementation:**

class Solution {

public:

TreeNode\* deleteNode(TreeNode\* root, int key) {

if(!root)

return root;

TreeNode\* toDelete = root, \*parent = NULL;

while(toDelete && toDelete->val != key){

parent = toDelete;

if(toDelete->val < key)

toDelete = toDelete->right;

else

toDelete = toDelete->left;

}

if(!toDelete)

return root;

if(!toDelete->left){

if(toDelete == root)

return toDelete->right;

if(toDelete == parent->left)

parent->left = toDelete->right;

else

parent->right = toDelete->right;

}

else if(!toDelete->right){

if(toDelete == root)

return toDelete->left;

if(toDelete == parent->left)

parent->left = toDelete->left;

else

parent->right = toDelete->left;

}

else{

TreeNode\* inorder\_succ = toDelete->right, \*succ\_parent = toDelete;

while(inorder\_succ->left){

succ\_parent = inorder\_succ;

inorder\_succ = inorder\_succ->left;

}

swap(inorder\_succ->val, toDelete->val);

if(succ\_parent == toDelete)

succ\_parent->right = inorder\_succ->right;

else

succ\_parent->left = inorder\_succ->right;

delete(inorder\_succ);

}

return root;

}

};

**GFG:**

// C++ program to demonstrate

// delete operation in binary

// search tree

#include <bits/stdc++.h>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

// A utility function to create a new BST node

struct node\* newNode(int item)

{

struct node\* temp

= (struct node\*)malloc(sizeof(struct node));

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// A utility function to do

// inorder traversal of BST

void inorder(struct node\* root)

{

if (root != NULL) {

inorder(root->left);

cout << root->key;

inorder(root->right);

}

}

/\* A utility function to

insert a new node with given key in

\* BST \*/

struct node\* insert(struct node\* node, int key)

{

/\* If the tree is empty, return a new node \*/

if (node == NULL)

return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

/\* return the (unchanged) node pointer \*/

return node;

}

/\* Given a non-empty binary search tree, return the node

with minimum key value found in that tree. Note that the

entire tree does not need to be searched. \*/

struct node\* minValueNode(struct node\* node)

{

struct node\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current && current->left != NULL)

current = current->left;

return current;

}

/\* Given a binary search tree and a key, this function

deletes the key and returns the new root \*/

struct node\* deleteNode(struct node\* root, int key)

{

// base case

if (root == NULL)

return root;

// If the key to be deleted is

// smaller than the root's

// key, then it lies in left subtree

if (key < root->key)

root->left = deleteNode(root->left, key);

// If the key to be deleted is

// greater than the root's

// key, then it lies in right subtree

else if (key > root->key)

root->right = deleteNode(root->right, key);

// if key is same as root's key, then This is the node

// to be deleted

else {

// node has no child

if (root->left==NULL and root->right==NULL)

return NULL;

// node with only one child or no child

else if (root->left == NULL) {

struct node\* temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL) {

struct node\* temp = root->left;

free(root);

return temp;

}

// node with two children: Get the inorder successor

// (smallest in the right subtree)

struct node\* temp = minValueNode(root->right);

// Copy the inorder successor's content to this node

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

return root;

}

// Driver Code

int main()

{

/\* Let us create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

struct node\* root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

cout << "Inorder traversal of the given tree \n";

inorder(root);

cout << "\nDelete 20\n";

root = deleteNode(root, 20);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 30\n";

root = deleteNode(root, 30);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 50\n";

root = deleteNode(root, 50);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

return 0;

}

**Output:**

Inorder traversal of the given tree

20 30 40 50 60 70 80

Delete 20

Inorder traversal of the modified tree

30 40 50 60 70 80

Delete 30

Inorder traversal of the modified tree

40 50 60 70 80

Delete 50

Inorder traversal of the modified tree

40 60 70 80

**Illustration:**

bst-delete

bst-delete2

**Time Complexity:** The worst case time complexity of delete operation is O(h) where h is the height of the Binary Search Tree. In worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

**Optimization to above code for two children case :**  
In the above recursive code, we recursively call delete() for the successor. We can avoid recursive calls by keeping track of the parent node of the successor so that we can simply remove the successor by making the child of a parent NULL. We know that the successor would always be a leaf node.

// C++ program to implement optimized delete in BST.

#include <bits/stdc++.h>

using namespace std;

struct Node {

int key;

struct Node \*left, \*right;

};

// A utility function to create a new BST node

Node\* newNode(int item)

{

Node\* temp = new Node;

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// A utility function to do inorder traversal of BST

void inorder(Node\* root)

{

if (root != NULL) {

inorder(root->left);

printf("%d ", root->key);

inorder(root->right);

}

}

/\* A utility function to insert a new node with given key in

\* BST \*/

Node\* insert(Node\* node, int key)

{

/\* If the tree is empty, return a new node \*/

if (node == NULL)

return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

/\* return the (unchanged) node pointer \*/

return node;

}

/\* Given a binary search tree and a key, this function

deletes the key and returns the new root \*/

Node\* deleteNode(Node\* root, int k)

{

// Base case

if (root == NULL)

return root;

// Recursive calls for ancestors of

// node to be deleted

if (root->key > k) {

root->left = deleteNode(root->left, k);

return root;

}

else if (root->key < k) {

root->right = deleteNode(root->right, k);

return root;

}

// We reach here when root is the node

// to be deleted.

// If one of the children is empty

if (root->left == NULL) {

Node\* temp = root->right;

delete root;

return temp;

}

else if (root->right == NULL) {

Node\* temp = root->left;

delete root;

return temp;

}

// If both children exist

else {

Node\* succParent = root;

// Find successor

Node\* succ = root->right;

while (succ->left != NULL) {

succParent = succ;

succ = succ->left;

}

// Delete successor. Since successor

// is always left child of its parent

// we can safely make successor's right

// right child as left of its parent.

// If there is no succ, then assign

// succ->right to succParent->right

if (succParent != root)

succParent->left = succ->right;

else

succParent->right = succ->right;

// Copy Successor Data to root

root->key = succ->key;

// Delete Successor and return root

delete succ;

return root;

}

}

// Driver Code

int main()

{

/\* Let us create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

Node\* root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

printf("Inorder traversal of the given tree \n");

inorder(root);

printf("\nDelete 20\n");

root = deleteNode(root, 20);

printf("Inorder traversal of the modified tree \n");

inorder(root);

printf("\nDelete 30\n");

root = deleteNode(root, 30);

printf("Inorder traversal of the modified tree \n");

inorder(root);

printf("\nDelete 50\n");

root = deleteNode(root, 50);

printf("Inorder traversal of the modified tree \n");

inorder(root);

return 0;

}

**Output**

Inorder traversal of the given tree

20 30 40 50 60 70 80

Delete 20

Inorder traversal of the modified tree

30 40 50 60 70 80

Delete 30

Inorder traversal of the modified tree

40 50 60 70 80

Delete 50

Inorder traversal of the modified tree

40 60 70 80

# 199. Find min value in a BST

Given a **Binary Search Tree**. The task is to find the minimum element in this given BST.

**Example 1:**

**Input:**

           5

        /    \

       4      6

     /        \

   3          7

   /

    1

**Output:** 1

**Example 2:**

**Input:**

             9

             \

              10

              \

                11

**Output:** 9

**Your Task:**  
The task is to complete the function **minValue()** which takes root as the argument and returns the minimum element of BST. If the tree is empty, there is no minimum elemnt, so retutn **-1** in that case.

**Expected Time Complexity:**O(Height of the BST)  
**Expected Auxiliary Space:**O(Height of the BST).

**Constraints:**  
1 <= N <= 104

## Solution:

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value. 



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

int minValue(struct node\* node)

{

struct node\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left != NULL)

{

current = current->left;

}

return(current->data);

}

# 200. Find inorder successor and inorder predecessor in a BST

There is BST given with root node with key part as integer only. You need to find the inorder successor and predecessor of a given key. In case, if the either of predecessor or successor is not found print -1.

**Input:**  
The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. Each test case contains n denoting the number of edges of the BST. The next line contains the edges of the BST. The last line contains the key.

**Output:**  
Print the predecessor followed by successor for the given key. If the predecessor or successor is not found print -1.

**Constraints:**  
1<=T<=100  
1<=n<=100  
1<=data of node<=100  
1<=key<=100

**Example:  
Input:**  
2  
6  
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R  
65  
6  
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R  
100

**Output:**  
60 70  
80 -1

## Solution:

Following is the algorithm to reach the desired result. Its a recursive method:

Input: root node, key

output: predecessor node, successor node

1. If root is NULL

then return

2. if key is found then

a. If its left subtree is not null

Then predecessor will be the right most

child of left subtree or left child itself.

b. If its right subtree is not null

The successor will be the left most child

of right subtree or right child itself.

return

3. If key is smaller then root node

set the successor as root

search recursively into left subtree

else

set the predecessor as root

search recursively into right subtree

Following is the implementation of the above algorithm:

// This function finds predecessor and successor of key in BST.

// It sets pre and suc as predecessor and successor respectively

void findPreSuc(Node\* root, Node\*& pre, Node\*& suc, int key)

{

// Base case

if (root == NULL) return ;

// If key is present at root

if (root->key == key)

{

// the maximum value in left subtree is predecessor

if (root->left != NULL)

{

Node\* tmp = root->left;

while (tmp->right)

tmp = tmp->right;

pre = tmp ;

}

// the minimum value in right subtree is successor

if (root->right != NULL)

{

Node\* tmp = root->right ;

while (tmp->left)

tmp = tmp->left ;

suc = tmp ;

}

return ;

}

// If key is smaller than root's key, go to left subtree

if (root->key > key)

{

suc = root ;

findPreSuc(root->left, pre, suc, key) ;

}

else // go to right subtree

{

pre = root ;

findPreSuc(root->right, pre, suc, key) ;

}

**Output:**

Predecessor is 60

Successor is 70

**Another Approach :**  
We can also find the inorder successor and inorder predecessor using inorder traversal . Check if the current node is smaller than the given key for predecessor and for successor, check if it is greater than the given key. If it is greater than the given key then, check if it is smaller than the already stored value in successor then, update it. At last, get the predecessor and successor stored in q(successor) and p(predecessor).

/\*

since inorder traversal results in

ascending order visit to node , we

can store the values of the largest

no which is smaller than a (predecessor)

and smallest no which is large than

a (successor) using inorder traversal

\*/

void find\_p\_s(Node\* root,int a,

Node\*\* p, Node\*\* q)

{

// If root is null return

if(!root)

return ;

// traverse the left subtree

find\_p\_s(root->left, a, p, q);

// root data is greater than a

if(root&&root->data > a)

{

// q stores the node whose data is greater

// than a and is smaller than the previously

// stored data in \*q which is successor

if((!\*q) || (\*q) && (\*q)->data > root->data)

\*q = root;

}

// if the root data is smaller than

// store it in p which is predecessor

else if(root && root->data < a)

{

\*p = root;

}

// traverse the right subtree

find\_p\_s(root->right, a, p, q);

}

**My Implemenatation using above approach:**

void findPreSuc(Node\* root, Node\*& pre, Node\*& suc, int key)

{

if(!root)

return;

findPreSuc(root->left, pre, suc, key);

if(root->key < key)

pre = root;

if(!suc)

if(root->key > key)

suc = root;

findPreSuc(root->right, pre, suc, key);

}

# 201. Check if a tree is a BST or not

Given the root of a binary tree. Check whether it is a BST or not.  
**Note:**We are considering that BSTs can not contain duplicate Nodes.  
A **BST** is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

**Example 1:**

**Input:**

   2

/    \

1      3

**Output:** 1

**Explanation:**

The left subtree of root node contains node

with key lesser than the root nodes key and

the right subtree of root node contains node

with key greater than the root nodes key.

Hence, the tree is a BST.

**Example 2:**

**Input:**

2

  \

  7

  \

  6

  \

  5

  \

  9

  \

  2

  \

  6

**Output:** 0

**Explanation:**

Since the node with value 7 has right subtree

nodes with keys less than 7, this is not a BST.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **isBST()** which takes the root of the tree as a parameter and returns **true** if the given binary tree is BST, else returns **false**.

**Expected Time Complexity:** O(N).  
**Expected Auxiliary Space:** O(Height of the BST).

**Constraints:**  
0 <= Number of edges <= 100000

## Solution:

**METHOD 1 (Simple but Wrong)**   
Following is a simple program. For each node, check if the left node of it is smaller than the node and right node of it is greater than the node.

int isBST(struct node\* node)

{

if (node == NULL)

return 1;

/\* false if left is > than node \*/

if (node->left != NULL && node->left->data > node->data)

return 0;

/\* false if right is < than node \*/

if (node->right != NULL && node->right->data < node->data)

return 0;

/\* false if, recursively, the left or right is not a BST \*/

if (!isBST(node->left) || !isBST(node->right))

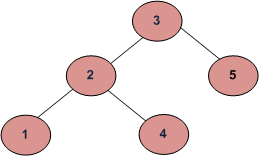
return 0;

/\* passing all that, it's a BST \*/

return 1;

}

**This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)**



**METHOD 2 (Correct but not efficient)**   
For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

/\* Returns true if a binary tree is a binary search tree \*/

int isBST(struct node\* node)

{

if (node == NULL)

return 1;

/\* false if the max of the left is > than us \*/

if (node->left != NULL && maxValue(node->left) >= node->data)

return 0;

/\* false if the min of the right is <= than us \*/

if (node->right != NULL && minValue(node->right) <= node->data)

return 0;

/\* false if, recursively, the left or right is not a BST \*/

if (!isBST(node->left) || !isBST(node->right))

return 0;

/\* passing all that, it's a BST \*/

return 1;

}

It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

**METHOD 3 (Correct and Efficient)**:   
Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTUtil(struct node\* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT\_MIN and INT\_MAX — they narrow from there.

Note: This method is not applicable if there are duplicate elements with value INT\_MIN or INT\_MAX.

Below is the implementation of the above approach:

#include<bits/stdc++.h>

using namespace std;

/\* A binary tree node has data,

pointer to left child and

a pointer to right child \*/

class node

{

public:

int data;

node\* left;

node\* right;

/\* Constructor that allocates

a new node with the given data

and NULL left and right pointers. \*/

node(int data)

{

this->data = data;

this->left = NULL;

this->right = NULL;

}

};

int isBSTUtil(node\* node, int min, int max);

/\* Returns true if the given

tree is a binary search tree

(efficient version). \*/

int isBST(node\* node)

{

return(isBSTUtil(node, INT\_MIN, INT\_MAX));

}

/\* Returns true if the given

tree is a BST and its values

are >= min and <= max. \*/

int isBSTUtil(node\* node, int min, int max)

{

/\* an empty tree is BST \*/

if (node==NULL)

return 1;

/\* false if this node violates

the min/max constraint \*/

if (node->data < min || node->data > max)

return 0;

/\* otherwise check the subtrees recursively,

tightening the min or max constraint \*/

return

isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values

isBSTUtil(node->right, node->data+1, max); // Allow only distinct values

}

/\* Driver code\*/

int main()

{

node \*root = new node(4);

root->left = new node(2);

root->right = new node(5);

root->left->left = new node(1);

root->left->right = new node(3);

if(isBST(root))

cout<<"Is BST";

else

cout<<"Not a BST";

return 0;

}

It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

**METHOD 3 (Correct and Efficient)**:   
Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTUtil(struct node\* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT\_MIN and INT\_MAX — they narrow from there.

Note: This method is not applicable if there are duplicate elements with value INT\_MIN or INT\_MAX.

Below is the implementation of the above approach:

#include<bits/stdc++.h>

using namespace std;

/\* A binary tree node has data,

pointer to left child and

a pointer to right child \*/

class node

{

public:

int data;

node\* left;

node\* right;

/\* Constructor that allocates

a new node with the given data

and NULL left and right pointers. \*/

node(int data)

{

this->data = data;

this->left = NULL;

this->right = NULL;

}

};

int isBSTUtil(node\* node, int min, int max);

/\* Returns true if the given

tree is a binary search tree

(efficient version). \*/

int isBST(node\* node)

{

return(isBSTUtil(node, INT\_MIN, INT\_MAX));

}

/\* Returns true if the given

tree is a BST and its values

are >= min and <= max. \*/

int isBSTUtil(node\* node, int min, int max)

{

/\* an empty tree is BST \*/

if (node==NULL)

return 1;

/\* false if this node violates

the min/max constraint \*/

if (node->data < min || node->data > max)

return 0;

/\* otherwise check the subtrees recursively,

tightening the min or max constraint \*/

return

isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values

isBSTUtil(node->right, node->data+1, max); // Allow only distinct values

}

/\* Driver code\*/

int main()

{

node \*root = new node(4);

root->left = new node(2);

root->right = new node(5);

root->left->left = new node(1);

root->left->right = new node(3);

if(isBST(root))

cout<<"Is BST";

else

cout<<"Not a BST";

return 0;

}

**Output:**

IS BST

**Time Complexity:** O(n)   
**Auxiliary Space:** O(1) if Function Call Stack size is not considered, otherwise O(n)  
    
**Simplified Method 3**   
We can simplify method 2 using NULL pointers instead of INT\_MIN and INT\_MAX values.

// C++ program to check if a given tree is BST.

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to

left child and a pointer to right child \*/

struct Node

{

int data;

struct Node\* left, \*right;

};

// Returns true if given tree is BST.

bool isBST(Node\* root, Node\* l=NULL, Node\* r=NULL)

{

// Base condition

if (root == NULL)

return true;

// if left node exist then check it has

// correct data or not i.e. left node's data

// should be less than root's data

if (l != NULL and root->data <= l->data)

return false;

// if right node exist then check it has

// correct data or not i.e. right node's data

// should be greater than root's data

if (r != NULL and root->data >= r->data)

return false;

// check recursively for every node.

return isBST(root->left, l, root) and

isBST(root->right, root, r);

}

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

struct Node\* newNode(int data)

{

struct Node\* node = new Node;

node->data = data;

node->left = node->right = NULL;

return (node);

}

/\* Driver program to test above functions\*/

int main()

{

struct Node \*root = newNode(3);

root->left = newNode(2);

root->right = newNode(5);

root->left->left = newNode(1);

root->left->right = newNode(4);

if (isBST(root,NULL,NULL))

cout << "Is BST";

else

cout << "Not a BST";

return 0;

}

**Output:**

Not a BST

**METHOD 4(Using In-Order Traversal)**

1) Do In-Order Traversal of the given tree and store the result in a temp array.

2) This method assumes that there are no duplicate values in the tree  
3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.  
Time Complexity: O(n)  
We can avoid the use of a Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to *ygos* for this space optimization.

bool isBST(node\* root)

{

static node \*prev = NULL;

// traverse the tree in inorder fashion

// and keep track of prev node

if (root)

{

if (!isBST(root->left))

return false;

// Allows only distinct valued nodes

if (prev != NULL &&

root->data <= prev->data)

return false;

prev = root;

return isBST(root->right);

}

return true;

}

The use of a static variable can also be avoided by using a reference to the prev node as a parameter.

// C++ program to check if a given tree is BST.

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to

left child and a pointer to right child \*/

struct Node

{

int data;

struct Node\* left, \*right;

Node(int data)

{

this->data = data;

left = right = NULL;

}

};

bool isBSTUtil(struct Node\* root, Node \*&prev)

{

// traverse the tree in inorder fashion and

// keep track of prev node

if (root)

{

if (!isBSTUtil(root->left, prev))

return false;

// Allows only distinct valued nodes

if (prev != NULL && root->data <= prev->data)

return false;

prev = root;

return isBSTUtil(root->right, prev);

}

return true;

}

bool isBST(Node \*root)

{

Node \*prev = NULL;

return isBSTUtil(root, prev);

}

/\* Driver program to test above functions\*/

int main()

{

struct Node \*root = new Node(3);

root->left = new Node(2);

root->right = new Node(5);

root->left->left = new Node(1);

root->left->right = new Node(4);

if (isBST(root))

cout << "Is BST";

else

cout << "Not a BST";

return 0;

}

**Output**:

Not a BST

# 202. Populate Inorder successor of all nodes

Given a Binary Tree, write a function to populate next pointer for all nodes. The next pointer for every node should be set to point to inorder successor.

**Example 1:**

**Input:**

10

  / \

  8 12

  /

  3

**Output:** 3->8 8->10 10->12 12->-1

**Explanation:** The inorder of the above tree is :

3 8 10 12. So the next pointer of node 3 is

pointing to 8 , next pointer of 8 is pointing

to 10 and so on.And next pointer of 12 is

pointing to -1 as there is no inorder successor

of 12.

**Example 2:**

**Input:**

1

  / \

  2 3

**Output:** 2->1 1->3 3->-1

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function**populateNext()**that takes the root node of the binary tree as input parameter.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)  
**Constraints:**  
1<=n<=10^5  
1<=data of the node<=10^5

## Solution:

**Solution (Use Reverse Inorder Traversal)**   
Traverse the given tree in reverse inorder traversal and keep track of previously visited node. When a node is being visited, assign a previously visited node as next.

// An implementation that doesn't use static variable

// A wrapper over populateNextRecur

void populateNext(node \*root)

{

// The first visited node will be the rightmost node

// next of the rightmost node will be NULL

node \*next = NULL;

populateNextRecur(root, &next);

}

/\* Set next of all descendants of p by

traversing them in reverse Inorder \*/

void populateNextRecur(node\* p, node \*\*next\_ref)

{

if (p)

{

// First set the next pointer in right subtree

populateNextRecur(p->right, next\_ref);

// Set the next as previously visited

// node in reverse Inorder

p->next = \*next\_ref;

// Change the prev for subsequent node

\*next\_ref = p;

// Finally, set the next pointer in right subtree

populateNextRecur(p->left, next\_ref);

}

}

**Time Complexity:**O(n)

**My Implementation using Inorder traversal**

class Solution

{

public:

Node\* prev = NULL;

void populateNext(Node \*root)

{

if(!root)

return;

populateNext(root->left);

if(prev)

prev->next = root;

prev = root;

populateNext(root->right);

}

};

**Time Complexity:**O(n)

# 203. Find LCA of 2 nodes in a BST

Given a Binary Search Tree (with all values unique) and two node values. Find the Lowest Common Ancestors of the two nodes in the BST.

**Example 1:**

**Input:**

              5

          /    \

         4       6

     /        \

   3           7

                 \

                    8

n1 = 7, n2 = 8

**Output:** 7

**Example 2:**

**Input:**

2

  / \

  1 3

n1 = 1, n2 = 3

**Output:** 2

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **LCA()** which takes the root Node of the BST and two integer values n1 and n2 as inputs and returns the Lowest Common Ancestor of the Nodes with values n1 and n2 in the given BST.

**Expected Time Complexity:** O(Height of the BST).  
**Expected Auxiliary Space:** O(Height of the BST).

**Constraints:**  
1 <= N <= 104

## Solution:

**Approach:** For Binary search tree, while traversing the tree from top to bottom the first node which lies in between the two numbers n1 and n2 is the LCA of the nodes, i.e. the first node n with the lowest depth which lies in between n1 and n2 (n1<=n<=n2) n1 < n2. So just recursively traverse the BST in, if node’s value is greater than both n1 and n2 then our LCA lies in the left side of the node, if it’s is smaller than both n1 and n2, then LCA lies on the right side. Otherwise, the root is LCA (assuming that both n1 and n2 are present in BST).

**Algorithm:**

1. Create a recursive function that takes a node and the two values n1 and n2.
2. If the value of the current node is less than both n1 and n2, then LCA lies in the right subtree. Call the recursive function for the right subtree.
3. If the value of the current node is greater than both n1 and n2, then LCA lies in the left subtree. Call the recursive function for the left subtree.
4. If both the above cases are false then return the current node as LCA.

**Implementation:**

// A recursive CPP program to find

// LCA of two nodes n1 and n2.

#include <bits/stdc++.h>

using namespace std;

class node

{

public:

int data;

node\* left, \*right;

};

/\* Function to find LCA of n1 and n2.

The function assumes that both

n1 and n2 are present in BST \*/

node \*lca(node\* root, int n1, int n2)

{

if (root == NULL) return NULL;

// If both n1 and n2 are smaller

// than root, then LCA lies in left

if (root->data > n1 && root->data > n2)

return lca(root->left, n1, n2);

// If both n1 and n2 are greater than

// root, then LCA lies in right

if (root->data < n1 && root->data < n2)

return lca(root->right, n1, n2);

return root;

}

/\* Helper function that allocates

a new node with the given data.\*/

node\* newNode(int data)

{

node\* Node = new node();

Node->data = data;

Node->left = Node->right = NULL;

return(Node);

}

/\* Driver code\*/

int main()

{

// Let us construct the BST

// shown in the above figure

node \*root = newNode(20);

root->left = newNode(8);

root->right = newNode(22);

root->left->left = newNode(4);

root->left->right = newNode(12);

root->left->right->left = newNode(10);

root->left->right->right = newNode(14);

int n1 = 10, n2 = 14;

node \*t = lca(root, n1, n2);

cout << "LCA of " << n1 << " and " << n2 << " is " << t->data<<endl;

n1 = 14, n2 = 8;

t = lca(root, n1, n2);

cout<<"LCA of " << n1 << " and " << n2 << " is " << t->data << endl;

n1 = 10, n2 = 22;

t = lca(root, n1, n2);

cout << "LCA of " << n1 << " and " << n2 << " is " << t->data << endl;

return 0;

}

**Output**

LCA of 10 and 14 is 12

LCA of 14 and 8 is 8

LCA of 10 and 22 is 20

**Complexity Analysis:**

* **Time Complexity:** O(h).   
  The time Complexity of the above solution is O(h), where h is the height of the tree.
* **Space Complexity:** O(h).   
  If recursive stack space is ignored, the space complexity of the above solution is constant.

**Iterative Implementation:** The above solution uses recursion. The recursive solution requires extra space in the form of the function call stack. So an iterative solution can be implemented which does not occupy space in the form of the function call stack.

**Implementation:**

// A recursive CPP program to find

// LCA of two nodes n1 and n2.

#include <bits/stdc++.h>

using namespace std;

class node

{

public:

int data;

node\* left, \*right;

};

/\* Function to find LCA of n1 and n2.

The function assumes that both n1 and n2

are present in BST \*/

node \*lca(node\* root, int n1, int n2)

{

while (root != NULL)

{

// If both n1 and n2 are smaller than root,

// then LCA lies in left

if (root->data > n1 && root->data > n2)

root = root->left;

// If both n1 and n2 are greater than root,

// then LCA lies in right

else if (root->data < n1 && root->data < n2)

root = root->right;

else break;

}

return root;

}

/\* Helper function that allocates

a new node with the given data.\*/

node\* newNode(int data)

{

node\* Node = new node();

Node->data = data;

Node->left = Node->right = NULL;

return(Node);

}

/\* Driver code\*/

int main()

{

// Let us construct the BST

// shown in the above figure

node \*root = newNode(20);

root->left = newNode(8);

root->right = newNode(22);

root->left->left = newNode(4);

root->left->right = newNode(12);

root->left->right->left = newNode(10);

root->left->right->right = newNode(14);

int n1 = 10, n2 = 14;

node \*t = lca(root, n1, n2);

cout << "LCA of " << n1 << " and " << n2 << " is " << t->data<<endl;

n1 = 14, n2 = 8;

t = lca(root, n1, n2);

cout<<"LCA of " << n1 << " and " << n2 << " is " << t->data << endl;

n1 = 10, n2 = 22;

t = lca(root, n1, n2);

cout << "LCA of " << n1 << " and " << n2 << " is " << t->data << endl;

return 0;

}

**Output**

LCA of 10 and 14 is 12

LCA of 14 and 8 is 8

LCA of 10 and 22 is 20

**Complexity Analysis:**

* **Time Complexity:** O(h).   
  The Time Complexity of the above solution is O(h), where h is the height of the tree.
* **Space Complexity:**O(1).   
  The space complexity of the above solution is constant.

# 204. Construct BST from preorder traversal

Given preorder traversal of a binary search tree, construct the BST.

**For example**, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be the root of the following tree.

10

/ \

5 40

/ \ \

1 7 50

## Solution:

**Method 1 ( O(n2) time complexity )**   
The first element of preorder traversal is always root. We first construct the root. Then we find the index of the first element which is greater than the root. Let the index be ‘i’. The values between root and ‘i’ will be part of the left subtree, and the values between ‘i'(inclusive) and ‘n-1’ will be part of the right subtree. Divide given pre[] at index “i” and recur for left and right sub-trees.

**For example** in {10, 5, 1, 7, 40, 50}, 10 is the first element, so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as following.

10

/ \

/ \

{5, 1, 7} {40, 50}

We recursively follow above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.

/\* A O(n^2) program for construction of BST from preorder

\* traversal \*/

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to left child

and a pointer to right child \*/

class node {

public:

int data;

node\* left;

node\* right;

};

// A utility function to create a node

node\* newNode(int data)

{

node\* temp = new node();

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// A recursive function to construct Full from pre[].

// preIndex is used to keep track of index in pre[].

node\* constructTreeUtil(int pre[], int\* preIndex, int low,

int high, int size)

{

// Base case

if (\*preIndex >= size || low > high)

return NULL;

// The first node in preorder traversal is root. So take

// the node at preIndex from pre[] and make it root, and

// increment preIndex

node\* root = newNode(pre[\*preIndex]);

\*preIndex = \*preIndex + 1;

// If the current subarray has only one element, no need

// to recur

if (low == high)

return root;

// Search for the first element greater than root

int i;

for (i = low; i <= high; ++i)

if (pre[i] > root->data)

break;

// Use the index of element found in preorder to divide

// preorder array in two parts. Left subtree and right

// subtree

root->left = constructTreeUtil(pre, preIndex, \*preIndex,

i - 1, size);

root->right

= constructTreeUtil(pre, preIndex, i, high, size);

return root;

}

// The main function to construct BST from given preorder

// traversal. This function mainly uses constructTreeUtil()

node\* constructTree(int pre[], int size)

{

int preIndex = 0;

return constructTreeUtil(pre, &preIndex, 0, size - 1,

size);

}

// A utility function to print inorder traversal of a Binary

// Tree

void printInorder(node\* node)

{

if (node == NULL)

return;

printInorder(node->left);

cout << node->data << " ";

printInorder(node->right);

}

// Driver code

int main()

{

int pre[] = { 10, 5, 1, 7, 40, 50 };

int size = sizeof(pre) / sizeof(pre[0]);

node\* root = constructTree(pre, size);

cout << "Inorder traversal of the constructed tree: \n";

printInorder(root);

return 0;

}

**Output**

Inorder traversal of the constructed tree:

1 5 7 10 40 50

**Time Complexity:** O(n2)

**Method 2 ( O(n) time complexity )**   
The idea used here is inspired by method 3 of [this](https://www.geeksforgeeks.org/archives/3042)post. The trick is to set a range {min .. max} for every node. Initialize the range as {INT\_MIN .. INT\_MAX}. The first node will definitely be in range, so create a root node. To construct the left subtree, set the range as {INT\_MIN …root->data}. If a value is in the range {INT\_MIN .. root->data}, the values are part of the left subtree. To construct the right subtree, set the range as {root->data..max .. INT\_MAX}.

Below is the implementation of the above idea:

/\* A O(n) program for construction

of BST from preorder traversal \*/

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to left child

and a pointer to right child \*/

class node {

public:

int data;

node\* left;

node\* right;

};

// A utility function to create a node

node\* newNode(int data)

{

node\* temp = new node();

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// A recursive function to construct

// BST from pre[]. preIndex is used

// to keep track of index in pre[].

node\* constructTreeUtil(int pre[], int\* preIndex, int key,

int min, int max, int size)

{

// Base case

if (\*preIndex >= size)

return NULL;

node\* root = NULL;

// If current element of pre[] is in range, then

// only it is part of current subtree

if (key > min && key < max) {

// Allocate memory for root of this

// subtree and increment \*preIndex

root = newNode(key);

\*preIndex = \*preIndex + 1;

if (\*preIndex < size) {

// Construct the subtree under root

// All nodes which are in range

// {min .. key} will go in left

// subtree, and first such node

// will be root of left subtree.

root->left = constructTreeUtil(pre, preIndex,

pre[\*preIndex],

min, key, size);

}

if (\*preIndex < size) {

// All nodes which are in range

// {key..max} will go in right

// subtree, and first such node

// will be root of right subtree.

root->right = constructTreeUtil(pre, preIndex,

pre[\*preIndex],

key, max, size);

}

}

return root;

}

// The main function to construct BST

// from given preorder traversal.

// This function mainly uses constructTreeUtil()

node\* constructTree(int pre[], int size)

{

int preIndex = 0;

return constructTreeUtil(pre, &preIndex, pre[0],

INT\_MIN, INT\_MAX, size);

}

// A utility function to print inorder

// traversal of a Binary Tree

void printInorder(node\* node)

{

if (node == NULL)

return;

printInorder(node->left);

cout << node->data << " ";

printInorder(node->right);

}

// Driver code

int main()

{

int pre[] = { 10, 5, 1, 7, 40, 50 };

int size = sizeof(pre) / sizeof(pre[0]);

// Function call

node\* root = constructTree(pre, size);

cout << "Inorder traversal of the constructed tree: \n";

printInorder(root);

return 0;

}

**Output**

Inorder traversal of the constructed tree:

1 5 7 10 40 50

**Time Complexity:** O(n)

We will soon publish a O(n) iterative solution as a separate post.  
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Method 3 ( O(n2) time complexity ):**

Simply do that just by using the recursion concept and iterating through the array of the given elements like below.

/\*Construct a BST from given pre-order traversal

for example if the given traversal is {10, 5, 1, 7, 40, 50},

then the output should be the root of the following tree.

10

/ \

5 40

/ \ \

1 7 50 \*/

class Node {

int data;

Node left, right;

Node(int data)

{

this.data = data;

this.left = this.right = null;

}

}

class CreateBSTFromPreorder {

private static Node node;

// This will create the BST

public static Node createNode(Node node, int data)

{

if (node == null)

node = new Node(data);

if (node.data > data)

node.left = createNode(node.left, data);

if (node.data < data)

node.right = createNode(node.right, data);

return node;

}

// A wrapper function of createNode

public static void create(int data)

{

node = createNode(node, data);

}

// A function to print BST in inorder

public static void inorderRec(Node root)

{

if (root != null) {

inorderRec(root.left);

System.out.println(root.data);

inorderRec(root.right);

}

}

// Driver Code

public static void main(String[] args)

{

int[] nodeData = { 10, 5, 1, 7, 40, 50 };

for (int i = 0; i < nodeData.length; i++) {

create(nodeData[i]);

}

inorderRec(node);

}

}

**Output**

1

5

7

10

40

50

Following is a stack based iterative solution that works in O(n) time.  
**1.** Create an empty stack.  
**2.** Make the first value as root. Push it to the stack.  
**3.**Keep on popping while the stack is not empty and the next value is greater than stack’s top value. Make this value as the right child of the last popped node. Push the new node to the stack.  
**4.**If the next value is less than the stack’s top value, make this value as the left child of the stack’s top node. Push the new node to the stack.  
**5.** Repeat steps 2 and 3 until there are items remaining in pre[].

// A O(n) iterative program for construction of BST from preorder traversal

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to left child

and a pointer to right child \*/

class Node

{

public:

int data;

Node \*left, \*right;

} node;

// A Stack has array of Nodes, capacity, and top

class Stack

{

public:

int top;

int capacity;

Node\*\* array;

} stack;

// A utility function to create a new tree node

Node\* newNode( int data )

{

Node\* temp = new Node();

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// A utility function to create a stack of given capacity

Stack\* createStack( int capacity )

{

Stack\* stack = new Stack();

stack->top = -1;

stack->capacity = capacity;

stack->array = new Node\*[stack->capacity \* sizeof( Node\* )];

return stack;

}

// A utility function to check if stack is full

int isFull( Stack\* stack )

{

return stack->top == stack->capacity - 1;

}

// A utility function to check if stack is empty

int isEmpty( Stack\* stack )

{

return stack->top == -1;

}

// A utility function to push an item to stack

void push( Stack\* stack, Node\* item )

{

if( isFull( stack ) )

return;

stack->array[ ++stack->top ] = item;

}

// A utility function to remove an item from stack

Node\* pop( Stack\* stack )

{

if( isEmpty( stack ) )

return NULL;

return stack->array[ stack->top-- ];

}

// A utility function to get top node of stack

Node\* peek( Stack\* stack )

{

return stack->array[ stack->top ];

}

// The main function that constructs BST from pre[]

Node\* constructTree ( int pre[], int size )

{

// Create a stack of capacity equal to size

Stack\* stack = createStack( size );

// The first element of pre[] is always root

Node\* root = newNode( pre[0] );

// Push root

push( stack, root );

int i;

Node\* temp;

// Iterate through rest of the size-1 items of given preorder array

for ( i = 1; i < size; ++i )

{

temp = NULL;

/\* Keep on popping while the next value is greater than

stack's top value. \*/

while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )

temp = pop( stack );

// Make this greater value as the right child

// and push it to the stack

if ( temp != NULL)

{

temp->right = newNode( pre[i] );

push( stack, temp->right );

}

// If the next value is less than the stack's top

// value, make this value as the left child of the

// stack's top node. Push the new node to stack

else

{

peek( stack )->left = newNode( pre[i] );

push( stack, peek( stack )->left );

}

}

return root;

}

// A utility function to print inorder traversal of a Binary Tree

void printInorder (Node\* node)

{

if (node == NULL)

return;

printInorder(node->left);

cout<<node->data<<" ";

printInorder(node->right);

}

// Driver program to test above functions

int main ()

{

int pre[] = {10, 5, 1, 7, 40, 50};

int size = sizeof( pre ) / sizeof( pre[0] );

Node \*root = constructTree(pre, size);

cout<<"Inorder traversal of the constructed tree: \n";

printInorder(root);

return 0;

}

Here is another method to construct binary search tree when given preorder traversal.

We know that the inorder traversal of the BST gives the element in non-decreasing manner. Hence we can sort the given preorder traversal to obtain the inorder traversal of the binary search tree.

We have already learnt the method to construct tree when given preorder and inorder traversals in [this](https://www.geeksforgeeks.org/construct-tree-from-given-inorder-and-preorder-traversal/)post. We will now use the same method to construct the BST.

#include <bits/stdc++.h>

using namespace std;

// A BST node has data, pointer to left

// child and pointer to right child

struct Node {

int data;

Node \*left, \*right;

};

// A utility function to create new node

Node\* getNode(int data)

{

Node\* temp = new Node();

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

/\* Recursive function to construct BST

Inorder traversal in[] and Preorder traversal

pre[]. Initial values of inStart and inEnd should be

0 and n -1.\*/

Node\* buildBTRec(int in[], int pre[], int inStart,

int inEnd, unordered\_map<int,int>& m)

{

static int preIdx = 0;

if (inStart > inEnd)

return NULL;

// Pick current node from Preorder traversal

// using preIndex and increment preIndex

int curr = pre[preIdx];

++preIdx;

Node\* temp = getNode(curr);

// If this node has no children then return

if (inStart == inEnd)

return temp;

// Else find the index of this node in

// inorder traversal

int idx = m[curr];

// Using this index construct left and right subtrees

temp->left = buildBTRec(in, pre, inStart, idx - 1, m);

temp->right = buildBTRec(in, pre, idx + 1, inEnd, m);

return temp;

}

// This function mainly creates a map to store

// the indices of all items so we can quickly

// access them later.

Node\* buildBST(int pre[], int n)

{

// Copy pre[] to in[] and sort it

int in[n];

for (int i = 0; i < n; i++)

in[i] = pre[i];

sort(in, in + n);

unordered\_map<int,int> m;

for(int i=0;i<n;i++)

{

m[in[i]] = i;

}

return buildBTRec(in, pre, 0, n-1,m);

}

// Inorder Traversal of tree

void inorderTraversal(Node\* node)

{

if(node==NULL)

return ;

inorderTraversal(node->left);

cout << node->data << " ";

inorderTraversal(node->right);

}

// Driver Program

int main()

{

int pre[] = { 100, 20, 10, 30, 200, 150, 300 };

int n = sizeof(pre) / sizeof(pre[0]);

Node\* root = buildBST(pre, n);

// Let's test the built tree by printing its

// Inorder traversal

cout << "Inorder traversal of the tree is \n";

inorderTraversal(root);

return 0;

}

**Output:**

Inorder traversal of the tree is

10 20 30 100 150 200 300

**Time Complexity:** Sorting takes O(nlogn) time for sorting and constructing using preorder and inorder traversals takes linear time. Hence overall time complexity of the above solution is O(nlogn).   
**Auxiliary Space:** O(n).

# 205. Convert Binary tree into BST keeping the original structure of Binary Tree intact

Given a Binary Tree, convert it to Binary Search Tree in such a way that keeps the original structure of Binary Tree intact.

**Example 1:**

**Input:**

1

  / \

2 3

**Output:** 1 2 3

**Example 2:**

**Input:**

1

/ \

2 3

/

4

**Output:** 1 2 3 4

**Explanation:**

The converted BST will be

3

/ \

2 4

/

1

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **binaryTreeToBST()** which takes the root of the Binary tree as input and returns the root of the BST. The driver code will print**inorder** traversal of the converted BST.

**Expected Time Complexity:** O(NLogN).  
**Expected Auxiliary Space:** O(N).

**Constraints:**  
1 <= Number of nodes <= 1000

## Solution:

Following is a 3 step solution for converting Binary tree to Binary Search Tree.

1) Create a temp array arr[] that stores inorder traversal of the tree. This step takes O(n) time.

2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes (n^2) time. This can be done in O(nLogn) time using Heap Sort or Merge Sort.

3) Again do inorder traversal of tree and copy array elements to tree nodes one by one. This step takes O(n) time.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

/\* A program to convert Binary Tree to Binary Search Tree \*/

#include <iostream>

using namespace std;

/\* A binary tree node structure \*/

struct node {

int data;

struct node\* left;

struct node\* right;

};

/\* A helper function that stores inorder traversal of a tree rooted

with node \*/

void storeInorder(struct node\* node, int inorder[], int\* index\_ptr)

{

// Base Case

if (node == NULL)

return;

/\* first store the left subtree \*/

storeInorder(node->left, inorder, index\_ptr);

/\* Copy the root's data \*/

inorder[\*index\_ptr] = node->data;

(\*index\_ptr)++; // increase index for next entry

/\* finally store the right subtree \*/

storeInorder(node->right, inorder, index\_ptr);

}

/\* A helper function to count nodes in a Binary Tree \*/

int countNodes(struct node\* root)

{

if (root == NULL)

return 0;

return countNodes(root->left) + countNodes(root->right) + 1;

}

// Following function is needed for library function qsort()

int compare(const void\* a, const void\* b)

{

return (\*(int\*)a - \*(int\*)b);

}

/\* A helper function that copies contents of arr[] to Binary Tree.

This function basically does Inorder traversal of Binary Tree and

one by one copy arr[] elements to Binary Tree nodes \*/

void arrayToBST(int\* arr, struct node\* root, int\* index\_ptr)

{

// Base Case

if (root == NULL)

return;

/\* first update the left subtree \*/

arrayToBST(arr, root->left, index\_ptr);

/\* Now update root's data and increment index \*/

root->data = arr[\*index\_ptr];

(\*index\_ptr)++;

/\* finally update the right subtree \*/

arrayToBST(arr, root->right, index\_ptr);

}

// This function converts a given Binary Tree to BST

void binaryTreeToBST(struct node\* root)

{

// base case: tree is empty

if (root == NULL)

return;

/\* Count the number of nodes in Binary Tree so that

we know the size of temporary array to be created \*/

int n = countNodes(root);

// Create a temp array arr[] and store inorder traversal of tree in arr[]

int\* arr = new int[n];

int i = 0;

storeInorder(root, arr, &i);

// Sort the array using library function for quick sort

qsort(arr, n, sizeof(arr[0]), compare);

// Copy array elements back to Binary Tree

i = 0;

arrayToBST(arr, root, &i);

// delete dynamically allocated memory to avoid memory leak

delete[] arr;

}

/\* Utility function to create a new Binary Tree node \*/

struct node\* newNode(int data)

{

struct node\* temp = new struct node;

temp->data = data;

temp->left = NULL;

temp->right = NULL;

return temp;

}

/\* Utility function to print inorder traversal of Binary Tree \*/

void printInorder(struct node\* node)

{

if (node == NULL)

return;

/\* first recur on left child \*/

printInorder(node->left);

/\* then print the data of node \*/

cout <<" "<< node->data;

/\* now recur on right child \*/

printInorder(node->right);

}

/\* Driver function to test above functions \*/

int main()

{

struct node\* root = NULL;

/\* Constructing tree given in the above figure

10

/ \

30 15

/ \

20 5 \*/

root = newNode(10);

root->left = newNode(30);

root->right = newNode(15);

root->left->left = newNode(20);

root->right->right = newNode(5);

// convert Binary Tree to BST

binaryTreeToBST(root);

cout <<"Following is Inorder Traversal of the converted BST:" << endl ;

printInorder(root);

return 0;

}

**Output**

Following is the inorder traversal of the converted BST

5 10 15 20 30

**Complexity Analysis:**

* **Time Complexity:** O(nlogn). This is the complexity of the sorting algorithm which we are using after first in-order traversal, rest of the operations take place in linear time.
* **Auxiliary Space:** O(n). Use of data structure ‘array’ to store in-order traversal.

# 206. Convert a normal BST into a Balanced BST

Given a Binary Search Tree**,** modify the given BST such that itis balanced and has minimum possible height.

Examples :

Input:

30

/

20

/

10

Output:

20

/ \

10 30

Input:

4

/

3

/

2

/

1

Output:

3 3 2

/ \ / \ / \

1 4 OR 2 4 OR 1 3 OR ..

\ / \

2 1 4

**Your Task:**  
The task is to complete the function **buildBalancedTree()** which takes root as the input argument, and returns the root of tree after converting the given BST into a balanced BST that has minimum possible height. The driver code will print the height of the updated tree in output itself.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(N)  
Here N denotes total number of nodes in given BST.

**Constraints:**  
1<=N<=200

## Solution:

A **Simple Solution** is to traverse nodes in Inorder and one by one insert into a self-balancing BST like AVL tree. Time complexity of this solution is O(n Log n) and this solution doesn’t guarantee   
An **Efficient Solution** can construct balanced BST in O(n) time with minimum possible height. Below are steps. 

1. Traverse given BST in inorder and store result in an array. This step takes O(n) time. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.
2. Build a balanced BST from the above created sorted array using the recursive approach discussed [here](https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/). This step also takes O(n) time as we traverse every element exactly once and processing an element takes O(1) time.

Below is the implementation of above steps.

// C++ program to convert a left unbalanced BST to

// a balanced BST

#include <bits/stdc++.h>

using namespace std;

struct Node

{

int data;

Node\* left, \*right;

};

/\* This function traverse the skewed binary tree and

stores its nodes pointers in vector nodes[] \*/

void storeBSTNodes(Node\* root, vector<Node\*> &nodes)

{

// Base case

if (root==NULL)

return;

// Store nodes in Inorder (which is sorted

// order for BST)

storeBSTNodes(root->left, nodes);

nodes.push\_back(root);

storeBSTNodes(root->right, nodes);

}

/\* Recursive function to construct binary tree \*/

Node\* buildTreeUtil(vector<Node\*> &nodes, int start,

int end)

{

// base case

if (start > end)

return NULL;

/\* Get the middle element and make it root \*/

int mid = (start + end)/2;

Node \*root = nodes[mid];

/\* Using index in Inorder traversal, construct

left and right subtress \*/

root->left = buildTreeUtil(nodes, start, mid-1);

root->right = buildTreeUtil(nodes, mid+1, end);

return root;

}

// This functions converts an unbalanced BST to

// a balanced BST

Node\* buildTree(Node\* root)

{

// Store nodes of given BST in sorted order

vector<Node \*> nodes;

storeBSTNodes(root, nodes);

// Constructs BST from nodes[]

int n = nodes.size();

return buildTreeUtil(nodes, 0, n-1);

}

// Utility function to create a new node

Node\* newNode(int data)

{

Node\* node = new Node;

node->data = data;

node->left = node->right = NULL;

return (node);

}

/\* Function to do preorder traversal of tree \*/

void preOrder(Node\* node)

{

if (node == NULL)

return;

printf("%d ", node->data);

preOrder(node->left);

preOrder(node->right);

}

// Driver program

int main()

{

/\* Constructed skewed binary tree is

10

/

8

/

7

/

6

/

5 \*/

Node\* root = newNode(10);

root->left = newNode(8);

root->left->left = newNode(7);

root->left->left->left = newNode(6);

root->left->left->left->left = newNode(5);

root = buildTree(root);

printf("Preorder traversal of balanced "

"BST is : \n");

preOrder(root);

return 0;

}

**Output :**

Preorder traversal of balanced BST is :

7 5 6 8 10

# 207. Merge two BST [ V.V.V>IMP ]

You are given two balanced binary search trees e.g., AVL or Red-Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in the first tree and n elements in the other tree. Your merge function should take O(m+n) time.  
In the following solutions, it is assumed that the sizes of trees are also given as input. If the size is not given, then we can get the size by traversing the tree.

## Solution:

**Method 1 (Insert elements of**the **first tree to second)**  
Take all elements of first BST one by one, and insert them into the second BST. Inserting an element to a self balancing BST takes Logn time (See [this](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/)) where n is size of the BST. So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1). The value of this expression will be between mLogn and mLog(m+n-1). As an optimization, we can pick the smaller tree as first tree.

**Method 2 (Merge Inorder Traversals)**  
1) Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.   
2) Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.   
3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.   
4) Construct a balanced tree from the merged array using the technique discussed in [this](https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/) post. This step takes O(m+n) time.  
Time complexity of this method is O(m+n) which is better than method 1. This method takes O(m+n) time even if the input BSTs are not balanced.   
Following is implementation of this method.

// C++ program to Merge Two Balanced Binary Search Trees

#include<bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to left child

and a pointer to right child \*/

class node

{

public:

int data;

node\* left;

node\* right;

};

// A utility function to merge two sorted arrays into one

int \*merge(int arr1[], int arr2[], int m, int n);

// A helper function that stores inorder

// traversal of a tree in inorder array

void storeInorder(node\* node, int inorder[],

int \*index\_ptr);

/\* A function that constructs Balanced

Binary Search Tree from a sorted array

See https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/ \*/

node\* sortedArrayToBST(int arr[], int start, int end);

/\* This function merges two balanced

BSTs with roots as root1 and root2.

m and n are the sizes of the trees respectively \*/

node\* mergeTrees(node \*root1, node \*root2, int m, int n)

{

// Store inorder traversal of

// first tree in an array arr1[]

int \*arr1 = new int[m];

int i = 0;

storeInorder(root1, arr1, &i);

// Store inorder traversal of second

// tree in another array arr2[]

int \*arr2 = new int[n];

int j = 0;

storeInorder(root2, arr2, &j);

// Merge the two sorted array into one

int \*mergedArr = merge(arr1, arr2, m, n);

// Construct a tree from the merged

// array and return root of the tree

return sortedArrayToBST (mergedArr, 0, m + n - 1);

}

/\* Helper function that allocates

a new node with the given data and

NULL left and right pointers. \*/

node\* newNode(int data)

{

node\* Node = new node();

Node->data = data;

Node->left = NULL;

Node->right = NULL;

return(Node);

}

// A utility function to print inorder

// traversal of a given binary tree

void printInorder(node\* node)

{

if (node == NULL)

return;

/\* first recur on left child \*/

printInorder(node->left);

cout << node->data << " ";

/\* now recur on right child \*/

printInorder(node->right);

}

// A utility function to merge

// two sorted arrays into one

int \*merge(int arr1[], int arr2[], int m, int n)

{

// mergedArr[] is going to contain result

int \*mergedArr = new int[m + n];

int i = 0, j = 0, k = 0;

// Traverse through both arrays

while (i < m && j < n)

{

// Pick the smaller element and put it in mergedArr

if (arr1[i] < arr2[j])

{

mergedArr[k] = arr1[i];

i++;

}

else

{

mergedArr[k] = arr2[j];

j++;

}

k++;

}

// If there are more elements in first array

while (i < m)

{

mergedArr[k] = arr1[i];

i++; k++;

}

// If there are more elements in second array

while (j < n)

{

mergedArr[k] = arr2[j];

j++; k++;

}

return mergedArr;

}

// A helper function that stores inorder

// traversal of a tree rooted with node

void storeInorder(node\* node, int inorder[], int \*index\_ptr)

{

if (node == NULL)

return;

/\* first recur on left child \*/

storeInorder(node->left, inorder, index\_ptr);

inorder[\*index\_ptr] = node->data;

(\*index\_ptr)++; // increase index for next entry

/\* now recur on right child \*/

storeInorder(node->right, inorder, index\_ptr);

}

/\* A function that constructs Balanced

// Binary Search Tree from a sorted array

See https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/ \*/

node\* sortedArrayToBST(int arr[], int start, int end)

{

/\* Base Case \*/

if (start > end)

return NULL;

/\* Get the middle element and make it root \*/

int mid = (start + end)/2;

node \*root = newNode(arr[mid]);

/\* Recursively construct the left subtree and make it

left child of root \*/

root->left = sortedArrayToBST(arr, start, mid-1);

/\* Recursively construct the right subtree and make it

right child of root \*/

root->right = sortedArrayToBST(arr, mid+1, end);

return root;

}

/\* Driver code\*/

int main()

{

/\* Create following tree as first balanced BST

100

/ \

50 300

/ \

20 70

\*/

node \*root1 = newNode(100);

root1->left = newNode(50);

root1->right = newNode(300);

root1->left->left = newNode(20);

root1->left->right = newNode(70);

/\* Create following tree as second balanced BST

80

/ \

40 120

\*/

node \*root2 = newNode(80);

root2->left = newNode(40);

root2->right = newNode(120);

node \*mergedTree = mergeTrees(root1, root2, 5, 3);

cout << "Following is Inorder traversal of the merged tree \n";

printInorder(mergedTree);

return 0;

}

**Output:**

Following is Inorder traversal of the merged tree

20 40 50 70 80 100 120 300

**Method 3 (In-Place Merge using DLL)**

Time complexity of this method is also O(m+n) and this method does conversion in place.  
We can use a Doubly Linked List to merge trees in place. Following are the steps.

1) Convert the given two Binary Search Trees into doubly linked list in place.  
2) Merge the two sorted Linked Lists.  
3) Build a Balanced Binary Search Tree from the merged list created in step 2.

1) Convert the given two Binary Search Trees into doubly linked list in place (Refer [this post](https://www.geeksforgeeks.org/the-great-tree-list-recursion-problem/) for this step).

The idea is to do in order traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable, say *prev*. For every visited node, make it next to the *prev*and previous of this node as *prev*.

// A C++ program for in-place conversion of Binary Tree to DLL

#include <iostream>

using namespace std;

/\* A binary tree node has data, and left and right pointers \*/

struct node

{

int data;

node\* left;

node\* right;

};

// A simple recursive function to convert a given Binary tree to Doubly

// Linked List

// root --> Root of Binary Tree

// head --> Pointer to head node of created doubly linked list

void BinaryTree2DoubleLinkedList(node \*root, node \*\*head)

{

// Base case

if (root == NULL) return;

// Initialize previously visited node as NULL. This is

// static so that the same value is accessible in all recursive

// calls

static node\* prev = NULL;

// Recursively convert left subtree

BinaryTree2DoubleLinkedList(root->left, head);

// Now convert this node

if (prev == NULL)

\*head = root;

else

{

root->left = prev;

prev->right = root;

}

prev = root;

// Finally convert right subtree

BinaryTree2DoubleLinkedList(root->right, head);

}

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

node\* newNode(int data)

{

node\* new\_node = new node;

new\_node->data = data;

new\_node->left = new\_node->right = NULL;

return (new\_node);

}

/\* Function to print nodes in a given doubly linked list \*/

void printList(node \*node)

{

while (node!=NULL)

{

cout << node->data << " ";

node = node->right;

}

}

/\* Driver program to test above functions\*/

int main()

{

// Let us create the tree shown in above diagram

node \*root = newNode(10);

root->left = newNode(12);

root->right = newNode(15);

root->left->left = newNode(25);

root->left->right = newNode(30);

root->right->left = newNode(36);

// Convert to DLL

node \*head = NULL;

BinaryTree2DoubleLinkedList(root, &head);

// Print the converted list

printList(head);

return 0;

}

**Output:**

25 12 30 10 36 15

Note that the use of static variables like above is not a recommended practice (we have used static for simplicity). Imagine a situation where the same function is called for two or more trees. The old value of *prev*would be used in the next call for a different tree. To avoid such problems, we can use a double-pointer or reference to a pointer.  
**Time Complexity:** The above program does a simple inorder traversal, so time complexity is O(n) where n is the number of nodes in given binary tree.

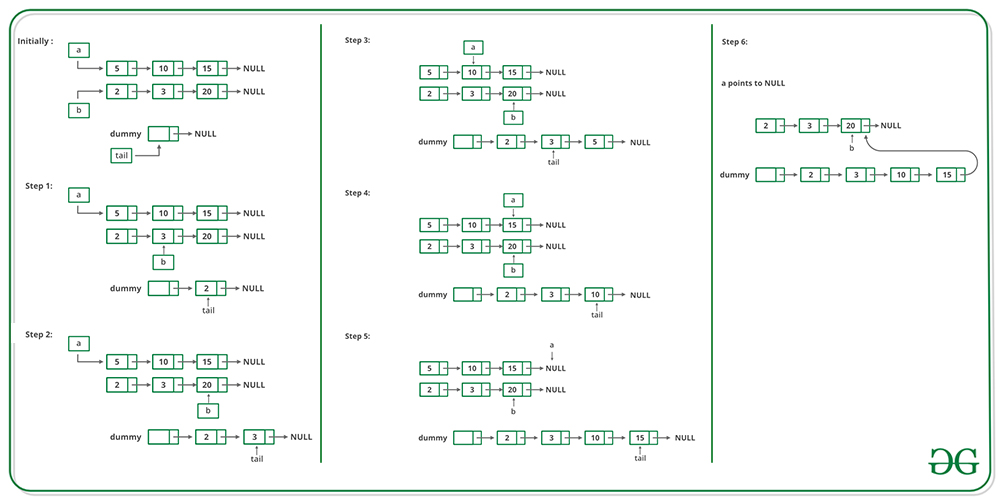
2) Merge the two sorted Linked Lists (Refer [this post](https://www.geeksforgeeks.org/merge-two-sorted-linked-lists/) for this step).

**Method 1 (Using Dummy Nodes)**   
The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy.

The dummy node gives the tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to the tail. When

We are done, the result is in dummy.next.

The below image is a dry run of the above approach:



Below is the implementation of the above approach:

/\* C++ program to merge two sorted linked lists \*/

#include <bits/stdc++.h>

using namespace std;

/\* Link list node \*/

class Node

{

public:

int data;

Node\* next;

};

/\* pull off the front node of

the source and put it in dest \*/

void MoveNode(Node\*\* destRef, Node\*\* sourceRef);

/\* Takes two lists sorted in increasing

order, and splices their nodes together

to make one big sorted list which

is returned. \*/

Node\* SortedMerge(Node\* a, Node\* b)

{

/\* a dummy first node to hang the result on \*/

Node dummy;

/\* tail points to the last result node \*/

Node\* tail = &dummy;

/\* so tail->next is the place to

add new nodes to the result. \*/

dummy.next = NULL;

while (1)

{

if (a == NULL)

{

/\* if either list runs out, use the

other list \*/

tail->next = b;

break;

}

else if (b == NULL)

{

tail->next = a;

break;

}

if (a->data <= b->data)

MoveNode(&(tail->next), &a);

else

MoveNode(&(tail->next), &b);

tail = tail->next;

}

return(dummy.next);

}

/\* UTILITY FUNCTIONS \*/

/\* MoveNode() function takes the

node from the front of the source,

and move it to the front of the dest.

It is an error to call this with the

source list empty.

Before calling MoveNode():

source == {1, 2, 3}

dest == {1, 2, 3}

After calling MoveNode():

source == {2, 3}

dest == {1, 1, 2, 3} \*/

void MoveNode(Node\*\* destRef, Node\*\* sourceRef)

{

/\* the front source node \*/

Node\* newNode = \*sourceRef;

assert(newNode != NULL);

/\* Advance the source pointer \*/

\*sourceRef = newNode->next;

/\* Link the old dest off the new node \*/

newNode->next = \*destRef;

/\* Move dest to point to the new node \*/

\*destRef = newNode;

}

/\* Function to insert a node at

the beginning of the linked list \*/

void push(Node\*\* head\_ref, int new\_data)

{

/\* allocate node \*/

Node\* new\_node = new Node();

/\* put in the data \*/

new\_node->data = new\_data;

/\* link the old list off the new node \*/

new\_node->next = (\*head\_ref);

/\* move the head to point to the new node \*/

(\*head\_ref) = new\_node;

}

/\* Function to print nodes in a given linked list \*/

void printList(Node \*node)

{

while (node!=NULL)

{

cout<<node->data<<" ";

node = node->next;

}

}

/\* Driver code\*/

int main()

{

/\* Start with the empty list \*/

Node\* res = NULL;

Node\* a = NULL;

Node\* b = NULL;

/\* Let us create two sorted linked lists

to test the functions

Created lists, a: 5->10->15, b: 2->3->20 \*/

push(&a, 15);

push(&a, 10);

push(&a, 5);

push(&b, 20);

push(&b, 3);

push(&b, 2);

/\* Remove duplicates from linked list \*/

res = SortedMerge(a, b);

cout << "Merged Linked List is: \n";

printList(res);

return 0;

}

**Output :**

Merged Linked List is:

2 3 5 10 15 20

**Method 2 (Using Recursion)**   
Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn’t want to use the recursive version for production code, however, because it will use stack space which is proportional to the length of the lists.

Node\* SortedMerge(Node\* a, Node\* b)

{

Node\* result = NULL;

/\* Base cases \*/

if (a == NULL)

return(b);

else if (b == NULL)

return(a);

/\* Pick either a or b, and recur \*/

if (a->data <= b->data)

{

result = a;

result->next = SortedMerge(a->next, b);

}

else

{

result = b;

result->next = SortedMerge(a, b->next);

}

return(result);

}

**Time Complexity:**Since we are traversing through the two lists fully. So, the time complexity is **O(m+n)** where m and n are the lengths of the two lists to be merged.

3) Build a Balanced Binary Search Tree from the merged list created in step 2. (Refer [this post](https://www.geeksforgeeks.org/in-place-conversion-of-sorted-dll-to-balanced-bst/) for this step)

In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as they appear in Doubly Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.   
While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

Following is the implementation of method 2. The main code which creates Balanced BST is highlighted.

#include <bits/stdc++.h>

using namespace std;

/\* A Doubly Linked List node that

will also be used as a tree node \*/

class Node

{

public:

int data;

// For tree, next pointer can be

// used as right subtree pointer

Node\* next;

// For tree, prev pointer can be

// used as left subtree pointer

Node\* prev;

};

// A utility function to count nodes in a Linked List

int countNodes(Node \*head);

Node\* sortedListToBSTRecur(Node \*\*head\_ref, int n);

/\* This function counts the number of

nodes in Linked List and then calls

sortedListToBSTRecur() to construct BST \*/

Node\* sortedListToBST(Node \*head)

{

/\*Count the number of nodes in Linked List \*/

int n = countNodes(head);

/\* Construct BST \*/

return sortedListToBSTRecur(&head, n);

}

/\* The main function that constructs

balanced BST and returns root of it.

head\_ref --> Pointer to pointer to

head node of Doubly linked list

n --> No. of nodes in the Doubly Linked List \*/

Node\* sortedListToBSTRecur(Node \*\*head\_ref, int n)

{

/\* Base Case \*/

if (n <= 0)

return NULL;

/\* Recursively construct the left subtree \*/

Node \*left = sortedListToBSTRecur(head\_ref, n/2);

/\* head\_ref now refers to middle node,

make middle node as root of BST\*/

Node \*root = \*head\_ref;

// Set pointer to left subtree

root->prev = left;

/\* Change head pointer of Linked List

for parent recursive calls \*/

\*head\_ref = (\*head\_ref)->next;

/\* Recursively construct the right

subtree and link it with root

The number of nodes in right subtree

is total nodes - nodes in

left subtree - 1 (for root) \*/

root->next = sortedListToBSTRecur(head\_ref, n-n/2-1);

return root;

}

/\* UTILITY FUNCTIONS \*/

/\* A utility function that returns

count of nodes in a given Linked List \*/

int countNodes(Node \*head)

{

int count = 0;

Node \*temp = head;

while(temp)

{

temp = temp->next;

count++;

}

return count;

}

/\* Function to insert a node at

the beginning of the Doubly Linked List \*/

void push(Node\*\* head\_ref, int new\_data)

{

/\* allocate node \*/

Node\* new\_node = new Node();

/\* put in the data \*/

new\_node->data = new\_data;

/\* since we are adding at the beginning,

prev is always NULL \*/

new\_node->prev = NULL;

/\* link the old list off the new node \*/

new\_node->next = (\*head\_ref);

/\* change prev of head node to new node \*/

if((\*head\_ref) != NULL)

(\*head\_ref)->prev = new\_node ;

/\* move the head to point to the new node \*/

(\*head\_ref) = new\_node;

}

/\* Function to print nodes in a given linked list \*/

void printList(Node \*node)

{

while (node!=NULL)

{

cout<<node->data<<" ";

node = node->next;

}

}

/\* A utility function to print

preorder traversal of BST \*/

void preOrder(Node\* node)

{

if (node == NULL)

return;

cout<<node->data<<" ";

preOrder(node->prev);

preOrder(node->next);

}

/\* Driver code\*/

int main()

{

/\* Start with the empty list \*/

Node\* head = NULL;

/\* Let us create a sorted linked list to test the functions

Created linked list will be 7->6->5->4->3->2->1 \*/

push(&head, 7);

push(&head, 6);

push(&head, 5);

push(&head, 4);

push(&head, 3);

push(&head, 2);

push(&head, 1);

cout<<"Given Linked List\n";

printList(head);

/\* Convert List to BST \*/

Node \*root = sortedListToBST(head);

cout<<"\nPreOrder Traversal of constructed BST \n ";

preOrder(root);

return 0;

}

**Output:**

Given Linked List

1 2 3 4 5 6 7

Pre-Order Traversal of constructed BST

4 2 1 3 6 5 7

**Time Complexity:** O(n)

# 208. Find Kth largest element in a BST

Given a Binary search tree. Your task is to complete the function which will return the Kth largest element without doing any modification in Binary Search Tree.

**Example 1:**

**Input:**

**4**

  / \

2 9

k = 2

**Output: 4**

**Example 2:**

**Input:**

       9

       \

       10

K = 1

**Output:** 10

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **kthLargest()** which takes the root of the BST and an integer K as inputs and returns the Kth largest element in the given BST.

**Expected Time Complexity:** O(H + K).  
**Expected Auxiliary Space:** O(H)

**Constraints:**  
1 <= N <= 1000  
1 <= K <= N

## Solution:

**Approach:** 

1. The idea is to do reverse inorder traversal of BST. Keep a count of nodes visited.
2. The reverse inorder traversal traverses all nodes in decreasing order, i.e, visit the right node then centre then left and continue traversing the nodes recursively.
3. While doing the traversal, keep track of the count of nodes visited so far.
4. When the count becomes equal to k, stop the traversal and print the key.

// C++ program to find k'th largest element in BST

#include<bits/stdc++.h>

using namespace std;

struct Node

{

int key;

Node \*left, \*right;

};

// A utility function to create a new BST node

Node \*newNode(int item)

{

Node \*temp = new Node;

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// A function to find k'th largest element in a given tree.

void kthLargestUtil(Node \*root, int k, int &c)

{

// Base cases, the second condition is important to

// avoid unnecessary recursive calls

if (root == NULL || c >= k)

return;

// Follow reverse inorder traversal so that the

// largest element is visited first

kthLargestUtil(root->right, k, c);

// Increment count of visited nodes

c++;

// If c becomes k now, then this is the k'th largest

if (c == k)

{

cout << "K'th largest element is "

<< root->key << endl;

return;

}

// Recur for left subtree

kthLargestUtil(root->left, k, c);

}

// Function to find k'th largest element

void kthLargest(Node \*root, int k)

{

// Initialize count of nodes visited as 0

int c = 0;

// Note that c is passed by reference

kthLargestUtil(root, k, c);

}

/\* A utility function to insert a new node with given key in BST \*/

Node\* insert(Node\* node, int key)

{

/\* If the tree is empty, return a new node \*/

if (node == NULL) return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

/\* return the (unchanged) node pointer \*/

return node;

}

// Driver Program to test above functions

int main()

{

/\* Let us create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

Node \*root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

int c = 0;

for (int k=1; k<=7; k++)

kthLargest(root, k);

return 0;

}

**Output:** 

K'th largest element is 80

K'th largest element is 70

K'th largest element is 60

K'th largest element is 50

K'th largest element is 40

K'th largest element is 30

K'th largest element is 20

**Complexity Analysis:** 

1. **Time Complexity:** O(h + k).   
   The code first traverses down to the rightmost node which takes O(h) time, then traverses k elements in O(k) time. Therefore overall time complexity is O(h + k).
2. **Auxiliary Space:** O(h).   
   Max recursion stack of height h at a given time.

# 209. [Find Kth smallest element in a BST](https://practice.geeksforgeeks.org/problems/find-k-th-smallest-element-in-bst/1)

Given a BST and an integer K. Find the Kth Smallest element in the BST.

**Example 1:**

**Input:**

2

  / \

  1 3

K = 2

**Output:** 2

**Example 2:**

**Input:**

2

  / \

  1 3

K = 5

**Output:** -1

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **KthSmallestElement()**which takes the root of the BST and integer K as inputs and return the Kth smallest element in the BST, if no such element exists return -1.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).  
  
**Constraints:**  
1<=Number of nodes<=100000

## Solution:

**Using Inorder traversal**

class Solution {

public:

int res=-1;

void fun(Node \*root, int &k){

if(!root || k<=0)

return;

fun(root->left, k);

if(!--k){

res = root->data;

return;

}

fun(root->right, k);

}

// Return the Kth smallest element in the given BST

int KthSmallestElement(Node \*root, int K) {

// add code here.

fun(root, K);

return res;

}

**Complexity Analysis:** 

**Time Complexity:** O(h + k).

**Auxiliary Space:** O(h).   
Max recursion stack of height h at a given time

# K’th smallest element in BST using O(1) Extra Space

The idea is to use [Morris Traversal](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/). In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree. See [this](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/)for more details.  
Below is the implementation of the idea.

// C++ program to find k'th largest element in BST

#include<bits/stdc++.h>

using namespace std;

// A BST node

struct Node

{

int key;

Node \*left, \*right;

};

// A function to find

int KSmallestUsingMorris(Node \*root, int k)

{

// Count to iterate over elements till we

// get the kth smallest number

int count = 0;

int ksmall = INT\_MIN; // store the Kth smallest

Node \*curr = root; // to store the current node

while (curr != NULL)

{

// Like Morris traversal if current does

// not have left child rather than printing

// as we did in inorder, we will just

// increment the count as the number will

// be in an increasing order

if (curr->left == NULL)

{

count++;

// if count is equal to K then we found the

// kth smallest, so store it in ksmall

if (count==k)

ksmall = curr->key;

// go to current's right child

curr = curr->right;

}

else

{

// we create links to Inorder Successor and

// count using these links

Node \*pre = curr->left;

while (pre->right != NULL && pre->right != curr)

pre = pre->right;

// building links

if (pre->right==NULL)

{

//link made to Inorder Successor

pre->right = curr;

curr = curr->left;

}

// While breaking the links in so made temporary

// threaded tree we will check for the K smallest

// condition

else

{

// Revert the changes made in if part (break link

// from the Inorder Successor)

pre->right = NULL;

count++;

// If count is equal to K then we found

// the kth smallest and so store it in ksmall

if (count==k)

ksmall = curr->key;

curr = curr->right;

}

}

}

return ksmall; //return the found value

}

// A utility function to create a new BST node

Node \*newNode(int item)

{

Node \*temp = new Node;

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

/\* A utility function to insert a new node with given key in BST \*/

Node\* insert(Node\* node, int key)

{

/\* If the tree is empty, return a new node \*/

if (node == NULL) return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

/\* return the (unchanged) node pointer \*/

return node;

}

// Driver Program to test above functions

int main()

{

/\* Let us create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

Node \*root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

for (int k=1; k<=7; k++)

cout << KSmallestUsingMorris(root, k) << " ";

return 0;

}

**Output:**

20 30 40 50 60 70 80

# 210. Count pairs from 2 BST whose sum is equal to given value "X"

Given two BSTs containing N**1** and N**2** distinct nodes respectively and given a value **x**. Your task is to complete the function **countPairs()**, that returns the count of all pairs from both the BSTs whose sum is equal to **x**.

**Example 1:**

**Input:**

**BST1:**

  5

/ \

  3 7

  / \ / \

2 4 6 8

**BST2:**

  10

  / \

  6 15

  / \ / \

  3 8 11 18

**x** = 16

**Output:**

3

**Explanation:**

The pairs are: **(5, 11), (6, 10)** and **(8, 8)**

**Example 2:**

**Input:**

**BST1:**

1

\

3

/

2

**BST2:**

3

  / \

  2 4

/

1

**x** = 4

**Output:**

3

**Explanation:**

The pairs are: **(2, 2), (3, 1)** and **(1, 3)**

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **countPairs()**, which takes 2 BST's as parameter in form of **root1** and **root2** and the integer **x**, that returns the count of all pairs from both the BSTs whose sum is equal to **x**.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 ≤ Number of nodes ≤ 105  
1 ≤ Data of a node ≤ 106

## Solution:

**Method 1:** Traverse BST 1 from smallest value to node to largest. This can be achieved with the help of [iterative inorder traversal](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/). Traverse BST 2 from largest value node to smallest. This can be achieved with the help of reverse inorder traversal. Perform these two traversals simultaneously. Sum up the corresponding node’s value from both the BSTs at a particular instance of traversals. If sum == x, then increment **count**. If x > sum, then move to the inorder successor of the current node of BST 1, else move to the inorder predecessor of the current node of BST 2. Perform these operations until either of the two traversals gets completed.

// C++ implementation to count pairs from two

// BSTs whose sum is equal to a given value x

#include <bits/stdc++.h>

using namespace std;

// structure of a node of BST

struct Node {

int data;

Node\* left, \*right;

};

// function to create and return a node of BST

Node\* getNode(int data)

{

// allocate space for the node

Node\* new\_node = (Node\*)malloc(sizeof(Node));

// put in the data

new\_node->data = data;

new\_node->left = new\_node->right = NULL;

}

// function to count pairs from two BSTs

// whose sum is equal to a given value x

int countPairs(Node\* root1, Node\* root2, int x)

{

// if either of the tree is empty

if (root1 == NULL || root2 == NULL)

return 0;

// stack 'st1' used for the inorder

// traversal of BST 1

// stack 'st2' used for the reverse

// inorder traversal of BST 2

stack<Node\*> st1, st2;

Node\* top1, \*top2;

int count = 0;

// the loop will break when either of two

// traversals gets completed

while (1) {

// to find next node in inorder

// traversal of BST 1

while (root1 != NULL) {

st1.push(root1);

root1 = root1->left;

}

// to find next node in reverse

// inorder traversal of BST 2

while (root2 != NULL) {

st2.push(root2);

root2 = root2->right;

}

// if either gets empty then corresponding

// tree traversal is completed

if (st1.empty() || st2.empty())

break;

top1 = st1.top();

top2 = st2.top();

// if the sum of the node's is equal to 'x'

if ((top1->data + top2->data) == x) {

// increment count

count++;

// pop nodes from the respective stacks

st1.pop();

st2.pop();

// insert next possible node in the

// respective stacks

root1 = top1->right;

root2 = top2->left;

}

// move to next possible node in the

// inorder traversal of BST 1

else if ((top1->data + top2->data) < x) {

st1.pop();

root1 = top1->right;

}

// move to next possible node in the

// reverse inorder traversal of BST 2

else {

st2.pop();

root2 = top2->left;

}

}

// required count of pairs

return count;

}

// Driver program to test above

int main()

{

// formation of BST 1

Node\* root1 = getNode(5); /\* 5 \*/

root1->left = getNode(3); /\* / \ \*/

root1->right = getNode(7); /\* 3 7 \*/

root1->left->left = getNode(2); /\* / \ / \ \*/

root1->left->right = getNode(4); /\* 2 4 6 8 \*/

root1->right->left = getNode(6);

root1->right->right = getNode(8);

// formation of BST 2

Node\* root2 = getNode(10); /\* 10 \*/

root2->left = getNode(6); /\* / \ \*/

root2->right = getNode(15); /\* 6 15 \*/

root2->left->left = getNode(3); /\* / \ / \ \*/

root2->left->right = getNode(8); /\* 3 8 11 18 \*/

root2->right->left = getNode(11);

root2->right->right = getNode(18);

int x = 16;

cout << "Pairs = "

<< countPairs(root1, root2, x);

return 0;

}

**Output**

Pairs = 3

**Time Complexity:** O(n1 + n2)   
**Auxiliary Space:** O(h1 + h2) Where h1 is height of first tree and h2 is height of second tree

**Method 2 :**

1. Recursive approach to solving this question.
2. Traverse the BST1 and for each node find the diff i.e. (x – root1.data) in BST2 and increment the count.

// Java implementation to count pairs from two

// BSTs whose sum is equal to a given value x

import java.util.Stack;

public class GFG {

// structure of a node of BST

static class Node {

int data;

Node left, right;

// constructor

public Node(int data)

{

this.data = data;

left = null;

right = null;

}

}

static Node root1;

static Node root2;

// function to count pairs from two BSTs

// whose sum is equal to a given value x

public static int pairCount = 0;

public static void traverseTree(Node root1, Node root2,

int sum)

{

if (root1 == null || root2 == null) {

return;

}

traverseTree(root1.left, root2, sum);

traverseTree(root1.right, root2, sum);

int diff = sum - root1.data;

findPairs(root2, diff);

}

private static void findPairs(Node root2, int diff)

{

if (root2 == null) {

return;

}

if (diff > root2.data) {

findPairs(root2.right, diff);

}

else {

findPairs(root2.left, diff);

}

if (root2.data == diff) {

pairCount++;

}

}

public static int countPairs(Node root1, Node root2,

int sum)

{

traverseTree(root1, root2, sum);

return pairCount;

}

// Driver program to test above

public static void main(String args[])

{

// formation of BST 1

root1 = new Node(5); /\* 5 \*/

root1.left = new Node(3); /\* / \ \*/

root1.right = new Node(7); /\* 3 7 \*/

root1.left.left = new Node(2); /\* / \ / \ \*/

root1.left.right = new Node(4); /\* 2 4 6 8 \*/

root1.right.left = new Node(6);

root1.right.right = new Node(8);

// formation of BST 2

root2 = new Node(10); /\* 10 \*/

root2.left = new Node(6); /\* / \ \*/

root2.right = new Node(15); /\* 6 15 \*/

root2.left.left = new Node(3); /\* / \ / \ \*/

root2.left.right

= new Node(8); /\* 3 8 11 18 \*/

root2.right.left = new Node(11);

root2.right.right = new Node(18);

int x = 16;

System.out.println("Pairs = "

+ countPairs(root1, root2, x));

}

}

**Output**

Pairs = 3

Time complexity: O(n1 \* h2), here n1 is number of nodes in first BST and h2 is height of second BST.

**My Implementation:**

class Solution

{

public:

void storeInorder(Node\* root, vector<int> &arr){

if(!root)

return;

storeInorder(root->left, arr);

arr.push\_back(root->data);

storeInorder(root->right, arr);

}

int countPairs(Node\* root1, Node\* root2, int x)

{

vector<int> arr1, arr2;

storeInorder(root1, arr1);

storeInorder(root2, arr2);

int m = arr1.size(), n = arr2.size(), i=0, j=n-1, res=0;

//cout<<m<<" "<<n<<" "<<arr1[0]<<" "<<arr1[m-1]<<" "<<arr2[0]<<" "<<arr2[n-1]<<endl;

while(i<m && j>=0){

//cout<<"loop";

if((arr1[i]+arr2[j])==x){

res++;

i++;

j--;

//cout<<"equal "<<arr1[i]+arr2[j];

}

else if((arr1[i]+arr2[j])<x){

i++;

//cout<<"less "<<arr1[i]+arr2[j];

}

else{

j--;

//cout<<"greater "<<arr1[i]+arr2[j];

}

}

return res;

}

};

**Time Complexity:** O(n1+n2)

**Space Complexity:** O(n1+n2)

# 211. Find the median of BST in O(n) time and O(1) space

Given a Binary Search Tree of size N, find the Median of its Node values.

**Example 1:**

**Input:**

       6

     /   \

   3      8

 /  \    /  \

1    4  7  9

**Output:** 6

**Explanation:** Inorder of Given BST will be:

1, 3, 4, 6, 7, 8, 9. So, here median will 6.

**Example 2:**

**Input:**

       6

     /   \

   3      8

 /   \    /

1    4  7

**Output:** 5

**Explanation:**Inorder of Given BST will be:

1, 3, 4, 6, 7, 8. So, here median will

(4 + 6)/2 = 10/2 = 5.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findMedian()** which takes the root of the Binary Search Tree as input and returns the Median of Node values in the given BST.  
Median of the BST is:

* If number of nodes are even: then median = (N/2 th node + (N/2)+1 th node)/2
* If number of nodes are odd : then median = (N+1)/2th node.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1<=N<=1000

## Solution:

To find the median, we need to find the Inorder of the BST because its Inorder will be in sorted order and then find the median i.e.   
The idea is based on [K’th smallest element in BST using O(1) Extra Space](https://www.geeksforgeeks.org/kth-smallest-element-in-bst-using-o1-extra-space/)  
The task is very simple if we are allowed to use extra space but Inorder traversal using recursion and stack both use Space which is not allowed here. So, the solution is to do [Morris Inorder traversal](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/) as it doesn’t require any extra space.

**Implementation:**

1- Count the no. of nodes in the given BST

using Morris Inorder Traversal.

2- Then Perform Morris Inorder traversal one

more time by counting nodes and by checking if

count is equal to the median point.

To consider even no. of nodes an extra pointer

pointing to the previous node is used.

/\* C++ program to find the median of BST in O(n)

time and O(1) space\*/

#include<bits/stdc++.h>

using namespace std;

/\* A binary search tree Node has data, pointer

to left child and a pointer to right child \*/

struct Node

{

int data;

struct Node\* left, \*right;

};

// A utility function to create a new BST node

struct Node \*newNode(int item)

{

struct Node \*temp = new Node;

temp->data = item;

temp->left = temp->right = NULL;

return temp;

}

/\* A utility function to insert a new node with

given key in BST \*/

struct Node\* insert(struct Node\* node, int key)

{

/\* If the tree is empty, return a new node \*/

if (node == NULL) return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->data)

node->left = insert(node->left, key);

else if (key > node->data)

node->right = insert(node->right, key);

/\* return the (unchanged) node pointer \*/

return node;

}

/\* Function to count nodes in a binary search tree

using Morris Inorder traversal\*/

int counNodes(struct Node \*root)

{

struct Node \*current, \*pre;

// Initialise count of nodes as 0

int count = 0;

if (root == NULL)

return count;

current = root;

while (current != NULL)

{

if (current->left == NULL)

{

// Count node if its left is NULL

count++;

// Move to its right

current = current->right;

}

else

{

/\* Find the inorder predecessor of current \*/

pre = current->left;

while (pre->right != NULL &&

pre->right != current)

pre = pre->right;

/\* Make current as right child of its

inorder predecessor \*/

if(pre->right == NULL)

{

pre->right = current;

current = current->left;

}

/\* Revert the changes made in if part to

restore the original tree i.e., fix

the right child of predecessor \*/

else

{

pre->right = NULL;

// Increment count if the current

// node is to be visited

count++;

current = current->right;

} /\* End of if condition pre->right == NULL \*/

} /\* End of if condition current->left == NULL\*/

} /\* End of while \*/

return count;

}

/\* Function to find median in O(n) time and O(1) space

using Morris Inorder traversal\*/

int findMedian(struct Node \*root)

{

if (root == NULL)

return 0;

int count = counNodes(root);

int currCount = 0;

struct Node \*current = root, \*pre, \*prev;

while (current != NULL)

{

if (current->left == NULL)

{

// count current node

currCount++;

// check if current node is the median

// Odd case

if (count % 2 != 0 && currCount == (count+1)/2)

return current->data;

// Even case

else if (count % 2 == 0 && currCount == (count/2)+1)

return (prev->data + current->data)/2;

// Update prev for even no. of nodes

prev = current;

//Move to the right

current = current->right;

}

else

{

/\* Find the inorder predecessor of current \*/

pre = current->left;

while (pre->right != NULL && pre->right != current)

pre = pre->right;

/\* Make current as right child of its inorder predecessor \*/

if (pre->right == NULL)

{

pre->right = current;

current = current->left;

}

/\* Revert the changes made in if part to restore the original

tree i.e., fix the right child of predecessor \*/

else

{

pre->right = NULL;

prev = pre;

// Count current node

currCount++;

// Check if the current node is the median

if (count % 2 != 0 && currCount == (count+1)/2 )

return current->data;

else if (count%2==0 && currCount == (count/2)+1)

return (prev->data+current->data)/2;

// update prev node for the case of even

// no. of nodes

prev = current;

current = current->right;

} /\* End of if condition pre->right == NULL \*/

} /\* End of if condition current->left == NULL\*/

} /\* End of while \*/

}

/\* Driver program to test above functions\*/

int main()

{

/\* Let us create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

struct Node \*root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

cout << "\nMedian of BST is "

<< findMedian(root);

return 0;

}

**Time Complexity:** O(n)

**Space Complexity:** O(1)

# 212. Count BST ndoes that lie in a given range

Given a Binary Search Tree (BST) and a range **l-h(inclusive)**, count the number of nodes in the BST that lie in the given range.

* The values smaller than root go to the left side
* The values greater and equal to the root go to the right side

**Example 1:**

**Input:**

10

  / \

  5 50

  / / \

  1 40 100

l = 5, h = 45

**Output:** 3

**Explanation:** 5 10 40 are the node in the

range

**Example 2:**

**Input:**

5

  / \

  4 6

  / \

 3 7

l = 2, h = 8

**Output:** 5

**Explanation:** All the nodes are in the

given range.

**Your Task:**  
This is a function problem. You don't have to take input. You are required to complete the function **getCountOfNode()**that takes root, l ,h as parameters and returns the **count**.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(Height of the BST).

**Constraints:**  
1 <= Number of nodes <= 100  
1 <= l < h < 103

## Solution:

The idea is to traverse the given binary search tree starting from root. For every node being visited, check if this node lies in range, if yes, then add 1 to result and recur for both of its children. If current node is smaller than low value of range, then recur for right child, else recur for left child.  
Below is the implementation of above idea.

// C++ program to count BST nodes withing a given range

#include<bits/stdc++.h>

using namespace std;

// A BST node

struct node

{

int data;

struct node\* left, \*right;

};

// Utility function to create new node

node \*newNode(int data)

{

node \*temp = new node;

temp->data = data;

temp->left = temp->right = NULL;

return (temp);

}

// Returns count of nodes in BST in range [low, high]

int getCount(node \*root, int low, int high)

{

// Base case

if (!root) return 0;

// Special Optional case for improving efficiency

if (root->data == high && root->data == low)

return 1;

// If current node is in range, then include it in count and

// recur for left and right children of it

if (root->data <= high && root->data >= low)

return 1 + getCount(root->left, low, high) +

getCount(root->right, low, high);

// If current node is smaller than low, then recur for right

// child

else if (root->data < low)

return getCount(root->right, low, high);

// Else recur for left child

else return getCount(root->left, low, high);

}

// Driver program

int main()

{

// Let us construct the BST shown in the above figure

node \*root = newNode(10);

root->left = newNode(5);

root->right = newNode(50);

root->left->left = newNode(1);

root->right->left = newNode(40);

root->right->right = newNode(100);

/\* Let us constructed BST shown in above example

10

/ \

5 50

/ / \

1 40 100 \*/

int l = 5;

int h = 45;

cout << "Count of nodes between [" << l << ", " << h

<< "] is " << getCount(root, l, h);

return 0;

}

**Output:**

Count of nodes between [5, 45] is 3

Time complexity of the above program is O(h + k) where h is height of BST and k is number of nodes in given range.

# 213. Replace every element with the least greater element on its right

Given an array of integers, replace every element with the least greater element on its right side in the array. If there are no greater elements on the right side, replace it with -1.

Examples:

**Input:** [8, 58, 71, 18, 31, 32, 63, 92,

43, 3, 91, 93, 25, 80, 28]

**Output:** [18, 63, 80, 25, 32, 43, 80, 93,

80, 25, 93, -1, 28, -1, -1]

## Solution:

A naive method is to run two loops. The outer loop will one by one pick array elements from left to right. The inner loop will find the smallest element greater than the picked element on its right side. Finally, the outer loop will replace the picked element with the element found by inner loop. The time complexity of this method will be O(n2).

A tricky solution would be to use Binary Search Trees. We start scanning the array from right to left and insert each element into the BST. For each inserted element, we replace it in the array by its inorder successor in BST. If the element inserted is the maximum so far (i.e. its inorder successor doesn’t exist), we replace it by -1.

Below is the implementation of the above idea –

// C++ program to replace every element with the

// least greater element on its right

#include <bits/stdc++.h>

using namespace std;

// A binary Tree node

struct Node {

int data;

Node \*left, \*right;

};

// A utility function to create a new BST node

Node\* newNode(int item)

{

Node\* temp = new Node;

temp->data = item;

temp->left = temp->right = NULL;

return temp;

}

/\* A utility function to insert a new node with

given data in BST and find its successor \*/

Node\* insert(Node\* node, int data, Node\*& succ)

{

/\* If the tree is empty, return a new node \*/

if (node == NULL)

return node = newNode(data);

// If key is smaller than root's key, go to left

// subtree and set successor as current node

if (data < node->data) {

succ = node;

node->left = insert(node->left, data, succ);

}

// go to right subtree

else if (data > node->data)

node->right = insert(node->right, data, succ);

return node;

}

// Function to replace every element with the

// least greater element on its right

void replace(int arr[], int n)

{

Node\* root = NULL;

// start from right to left

for (int i = n - 1; i >= 0; i--) {

Node\* succ = NULL;

// insert current element into BST and

// find its inorder successor

root = insert(root, arr[i], succ);

// replace element by its inorder

// successor in BST

if (succ)

arr[i] = succ->data;

else // No inorder successor

arr[i] = -1;

}

}

// Driver Program to test above functions

int main()

{

int arr[] = { 8, 58, 71, 18, 31, 32, 63, 92,

43, 3, 91, 93, 25, 80, 28 };

int n = sizeof(arr) / sizeof(arr[0]);

replace(arr, n);

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

return 0;

}

**Output**

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

The **worst-case time complexity** of the above solution is also O(n2) as it uses BST. The worst-case will happen when array is sorted in ascending or descending order. The complexity can easily be reduced to O(nlogn) by using balanced trees like red-black trees.

**Another Approach:**

We can use the [**Next Greater Element using stack**](https://www.geeksforgeeks.org/next-greater-element/)algorithmto solve this problem in **O(Nlog(N))** time and **O(N)**space.

Algorithm:

1. *First, we take an array of pairs namely temp, and store each element and its index in this array,i.e.****temp[i] will be storing {arr[i],i}****.*
2. [***Sort the array***](https://www.geeksforgeeks.org/sorting-algorithms/)*according to the array elements.*
3. *Now get the next greater index for each and every index of the temp array in an array namely index by using*[***Next Greater Element***](https://www.geeksforgeeks.org/next-greater-element/)*using stack.*
4. *Now index[i] stores the index of the next least greater element of the element temp[i].first and if index[i] is -1, then it means that there is no least greater element of the element temp[i].second at its right side.*
5. *Now take a result array where result[i] will be equal to****a[indexes[temp[i].second]]****if index[i] is not -1 otherwise result[i] will be equal to -1.*

Below is the implementation of the above approach

#include <bits/stdc++.h>

using namespace std;

// function to get the next least greater index for each and

// every temp.second of the temp array using stack this

// function is similar to the Next Greater element for each

// and every element of an array using stack difference is

// we are finding the next greater index not value and the

// indexes are stored in the temp[i].second for all i

vector<int> nextGreaterIndex(vector<pair<int, int> >& temp)

{

int n = temp.size();

// initially result[i] for all i is -1

vector<int> res(n, -1);

stack<int> stack;

for (int i = 0; i < n; i++) {

// if the stack is empty or this index is smaller

// than the index stored at top of the stack then we

// push this index to the stack

if (stack.empty() || temp[i].second < stack.top())

stack.push(

temp[i].second); // notice temp[i].second is

// the index

// else this index (i.e. temp[i].second) is greater

// than the index stored at top of the stack we pop

// all the indexes stored at stack's top and for all

// these indexes we make this index i.e.

// temp[i].second as their next greater index

else {

while (!stack.empty()

&& temp[i].second > stack.top()) {

res[stack.top()] = temp[i].second;

stack.pop();

}

// after that push the current index to the stack

stack.push(temp[i].second);

}

}

// now res will store the next least greater indexes for

// each and every indexes stored at temp[i].second for

// all i

return res;

}

// now we will be using above function for finding the next

// greater index for each and every indexes stored at

// temp[i].second

vector<int> replaceByLeastGreaterUsingStack(int arr[],

int n)

{

// first of all in temp we store the pairs of {arr[i].i}

vector<pair<int, int> > temp;

for (int i = 0; i < n; i++) {

temp.push\_back({ arr[i], i });

}

// we sort the temp according to the first of the pair

// i.e value

sort(temp.begin(), temp.end());

// now indexes vector will store the next greater index

// for each temp[i].second index

vector<int> indexes = nextGreaterIndex(temp);

// we initialize a result vector with all -1

vector<int> res(n, -1);

for (int i = 0; i < n; i++) {

// now if there is no next greater index after the

// index temp[i].second the result will be -1

// otherwise the result will be the element of the

// array arr at index indexes[temp[i].second]

if (indexes[temp[i].second] != -1)

res[temp[i].second]

= arr[indexes[temp[i].second]];

}

// return the res which will store the least greater

// element of each and every element in the array at its

// right side

return res;

}

// driver code

int main()

{

int arr[] = { 8, 58, 71, 18, 31, 32, 63, 92,

43, 3, 91, 93, 25, 80, 28 };

int n = sizeof(arr) / sizeof(int);

auto res = replaceByLeastGreaterUsingStack(arr, n);

cout << "Least Greater elements on the right side are "

<< endl;

for (int i : res)

cout << i << ' ';

cout << endl;

return 0;

}

**Output**

Least Greater elements on the right side are

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

**Another approach with set**

A different way to think about the problem is listing our requirements and then thinking over it to find a solution. If we traverse the array from backwards, we need  a data structure(ds) to support:

1. Insert an element into our ds in sorted order (so at any point of time the elements in our ds are sorted)

2. Finding the upper bound of the current element (upper bound will give just greater element from our ds if present)

Carefully observing at our requirements, a set is what comes in mind.

Why not multiset? Well we can use a multiset but there is no need to store an element more than once.

Let’s code our approach

**Time and space complexity**: We insert each element in our set and find upper bound for each element using a loop so its time complexity is O(n\*log(n)). We are storing each element in our set so space complexity is O(n)

#include <iostream>

#include <vector>

#include <set>

using namespace std;

void solve(vector<int>& arr) {

set<int> s;

vector<int> ans(arr.size());

for(int i=arr.size()-1;i>=0;i--) { //traversing the array backwards

s.insert(arr[i]); // inserting the element into set

auto it = s.upper\_bound(arr[i]); // finding upper bound

if(it == s.end()) arr[i] = -1; // if upper\_bound does not exist then -1

else arr[i] = \*it; // if upper\_bound exists, lets take it

}

}

void printArray(vector<int>& arr) {

for(int i : arr) cout << i << " ";

cout << "\n";

}

int main() {

vector<int> arr = {8, 58, 71, 18, 31, 32, 63, 92, 43, 3, 91, 93, 25, 80, 28};

printArray(arr);

solve(arr);

printArray(arr);

return 0;

}

**Output**

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

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

# 214. [Given "n" appointments, find the conflicting appointments](https://www.geeksforgeeks.org/given-n-appointments-find-conflicting-appointments/)

Given n appointments, find all conflicting appointments.

Examples:

Input: appointments[] = { {1, 5} {3, 7}, {2, 6}, {10, 15}, {5, 6}, {4, 100}}

Output: Following are conflicting intervals

[3,7] Conflicts with [1,5]

[2,6] Conflicts with [1,5]

[5,6] Conflicts with [3,7]

[4,100] Conflicts with [1,5]

An appointment is conflicting if it conflicts with any of the previous appointments in the array.

## Solution:

A **Simple Solution** is to one by one process all appointments from the second appointment to last. For every appointment i, check if it conflicts with i-1, i-2, … 0. The time complexity of this method is O(n2).   
We can use [**Interval Tree**](https://www.geeksforgeeks.org/interval-tree/) to solve this problem in O(nLogn) time.

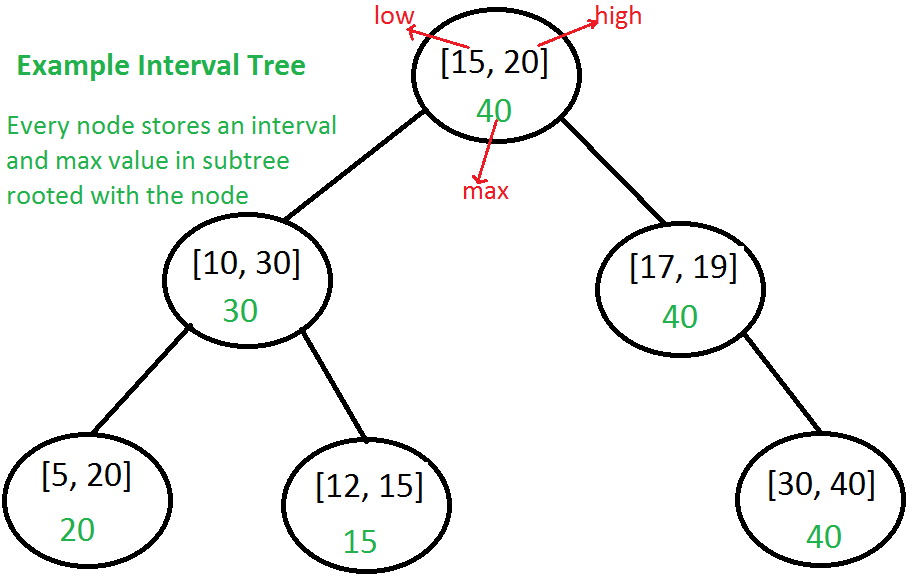
# Interval Tree

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently.  
**1)**Add an interval  
**2)**Remove an interval  
**3)** Given an interval x, find if x overlaps with any of the existing intervals.

***Interval Tree:*** The idea is to augment a self-balancing Binary Search Tree (BST) like [Red Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/), [AVL Tree](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/), etc to maintain set of intervals so that all operations can be done in O(Logn) time.

Every node of Interval Tree stores following information.  
a) **i**: An interval which is represented as a pair [low, high]  
b) **max**: Maximum high value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/IntervalSearcTree.png)

The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval x in an Interval tree rooted with root.

Interval overlappingIntervalSearch(root, x)

**1)** If x overlaps with root's interval, return the root's interval.

**2)** If left child of root is not empty and the max in left child

is greater than x's low value, recur for left child

**3)** Else recur for right child.

***How does the above algorithm work?***  
Let the interval to be searched be x. We need to prove this in for following two cases.

***Case 1:*** When we go to right subtree, one of the following must be true.  
a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.  
b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than x.low. So the interval cannot be present in left subtree.

***Case 2:***When we go to left subtree, one of the following must be true.  
a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.  
b) There is no overlap in either subtree: This is the most important part. We need to consider following facts.  
… We went to left subtree because x.low <= max in left subtree  
…. max in left subtree is a high of one of the intervals let us say [a, max] in left subtree.  
…. Since x doesn’t overlap with any node in left subtree x.low must be smaller than ‘a‘.  
…. All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than ‘a‘.  
…. From above two facts, we can say all intervals in right subtree have low value greater than x.low. So x cannot overlap with any interval in right subtree.

**Implementation of Interval Tree:**  
Following is C++ implementation of Interval Tree. The implementation uses basic [insert operation of BST](http://geeksquiz.com/binary-search-tree-set-1-search-and-insertion/) to keep things simple. Ideally it should be [insertion of AVL Tree](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/) or [insertion of Red-Black Tree](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/). [Deletion from BST](http://geeksquiz.com/binary-search-tree-set-2-delete/) is left as an exercise.

#include <iostream>

using namespace std;

// Structure to represent an interval

struct Interval

{

int low, high;

};

// Structure to represent a node in Interval Search Tree

struct ITNode

{

Interval \*i; // 'i' could also be a normal variable

int max;

ITNode \*left, \*right;

};

// A utility function to create a new Interval Search Tree Node

ITNode \* newNode(Interval i)

{

ITNode \*temp = new ITNode;

temp->i = new Interval(i);

temp->max = i.high;

temp->left = temp->right = NULL;

return temp;

};

// A utility function to insert a new Interval Search Tree Node

// This is similar to BST Insert. Here the low value of interval

// is used tomaintain BST property

ITNode \*insert(ITNode \*root, Interval i)

{

// Base case: Tree is empty, new node becomes root

if (root == NULL)

return newNode(i);

// Get low value of interval at root

int l = root->i->low;

// If root's low value is smaller, then new interval goes to

// left subtree

if (i.low < l)

root->left = insert(root->left, i);

// Else, new node goes to right subtree.

else

root->right = insert(root->right, i);

// Update the max value of this ancestor if needed

if (root->max < i.high)

root->max = i.high;

return root;

}

// A utility function to check if given two intervals overlap

bool doOVerlap(Interval i1, Interval i2)

{

if (i1.low <= i2.high && i2.low <= i1.high)

return true;

return false;

}

// The main function that searches a given interval i in a given

// Interval Tree.

Interval \*overlapSearch(ITNode \*root, Interval i)

{

// Base Case, tree is empty

if (root == NULL) return NULL;

// If given interval overlaps with root

if (doOVerlap(\*(root->i), i))

return root->i;

// If left child of root is present and max of left child is

// greater than or equal to given interval, then i may

// overlap with an interval is left subtree

if (root->left != NULL && root->left->max >= i.low)

return overlapSearch(root->left, i);

// Else interval can only overlap with right subtree

return overlapSearch(root->right, i);

}

void inorder(ITNode \*root)

{

if (root == NULL) return;

inorder(root->left);

cout << "[" << root->i->low << ", " << root->i->high << "]"

<< " max = " << root->max << endl;

inorder(root->right);

}

// Driver program to test above functions

int main()

{

// Let us create interval tree shown in above figure

Interval ints[] = {{15, 20}, {10, 30}, {17, 19},

{5, 20}, {12, 15}, {30, 40}

};

int n = sizeof(ints)/sizeof(ints[0]);

ITNode \*root = NULL;

for (int i = 0; i < n; i++)

root = insert(root, ints[i]);

cout << "Inorder traversal of constructed Interval Tree is\n";

inorder(root);

Interval x = {6, 7};

cout << "\nSearching for interval [" << x.low << "," << x.high << "]";

Interval \*res = overlapSearch(root, x);

if (res == NULL)

cout << "\nNo Overlapping Interval";

else

cout << "\nOverlaps with [" << res->low << ", " << res->high << "]";

return 0;

}

Output:

Inorder traversal of constructed Interval Tree is

[5, 20] max = 20

[10, 30] max = 30

[12, 15] max = 15

[15, 20] max = 40

[17, 19] max = 40

[30, 40] max = 40

Searching for interval [6,7]

Overlaps with [5, 20]

**Applications of Interval Tree:**  
Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene (Source [Wiki](http://en.wikipedia.org/wiki/Interval_tree)).

**Interval Tree vs**[Segment Tree](https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/)  
Both segment and interval trees store intervals. Segment tree is mainly optimized for queries for a given point, and interval trees are mainly optimized for overlapping queries for a given interval.

Now to solve the question:

Following is a detailed algorithm.

1) Create an Interval Tree, initially with the first appointment.

2) Do following for all other appointments starting from the second one.

a) Check if the current appointment conflicts with any of the existing

appointments in Interval Tree. If conflicts, then print the current

appointment. This step can be done O(Logn) time.

b) Insert the current appointment in Interval Tree. This step also can

be done O(Logn) time.

Following is the implementation of the above idea.

// C++ program to print all conflicting appointments in a

// given set of appointments

#include <bits/stdc++.h>

using namespace std;

// Structure to represent an interval

struct Interval

{

int low, high;

};

// Structure to represent a node in Interval Search Tree

struct ITNode

{

Interval \*i; // 'i' could also be a normal variable

int max;

ITNode \*left, \*right;

};

// A utility function to create a new Interval Search Tree Node

ITNode \* newNode(Interval i)

{

ITNode \*temp = new ITNode;

temp->i = new Interval(i);

temp->max = i.high;

temp->left = temp->right = NULL;

return temp;

};

// A utility function to insert a new Interval Search Tree

// Node. This is similar to BST Insert. Here the low value

// of interval is used tomaintain BST property

ITNode \*insert(ITNode \*root, Interval i)

{

// Base case: Tree is empty, new node becomes root

if (root == NULL)

return newNode(i);

// Get low value of interval at root

int l = root->i->low;

// If root's low value is smaller, then new interval

// goes to left subtree

if (i.low < l)

root->left = insert(root->left, i);

// Else, new node goes to right subtree.

else

root->right = insert(root->right, i);

// Update the max value of this ancestor if needed

if (root->max < i.high)

root->max = i.high;

return root;

}

// A utility function to check if given two intervals overlap

bool doOVerlap(Interval i1, Interval i2)

{

if (i1.low < i2.high && i2.low < i1.high)

return true;

return false;

}

// The main function that searches a given interval i

// in a given Interval Tree.

Interval \*overlapSearch(ITNode \*root, Interval i)

{

// Base Case, tree is empty

if (root == NULL) return NULL;

// If given interval overlaps with root

if (doOVerlap(\*(root->i), i))

return root->i;

// If left child of root is present and max of left child

// is greater than or equal to given interval, then i may

// overlap with an interval is left subtree

if (root->left != NULL && root->left->max >= i.low)

return overlapSearch(root->left, i);

// Else interval can only overlap with right subtree

return overlapSearch(root->right, i);

}

// This function prints all conflicting appointments in a given

// array of appointments.

void printConflicting(Interval appt[], int n)

{

// Create an empty Interval Search Tree, add first

// appointment

ITNode \*root = NULL;

root = insert(root, appt[0]);

// Process rest of the intervals

for (int i=1; i<n; i++)

{

// If current appointment conflicts with any of the

// existing intervals, print it

Interval \*res = overlapSearch(root, appt[i]);

if (res != NULL)

cout << "[" << appt[i].low << "," << appt[i].high

<< "] Conflicts with [" << res->low << ","

<< res->high << "]\n";

// Insert this appointment

root = insert(root, appt[i]);

}

}

// Driver program to test above functions

int main()

{

// Let us create interval tree shown in above figure

Interval appt[] = { {1, 5}, {3, 7}, {2, 6}, {10, 15},

{5, 6}, {4, 100}};

int n = sizeof(appt)/sizeof(appt[0]);

cout << "Following are conflicting intervals\n";

printConflicting(appt, n);

return 0;

}

**Output:**

Following are conflicting intervals

[3,7] Conflicts with [1,5]

[2,6] Conflicts with [1,5]

[5,6] Conflicts with [3,7]

[4,100] Conflicts with [1,5]

Note that the above implementation uses a simple Binary Search Tree insert operations. Therefore, the time complexity of the above implementation is more than O(nLogn). We can use [Red-Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/) or [AVL Tree](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/) balancing techniques to make the above implementation O(nLogn).

# 215. Check preorder is valid or not

Given an array **arr[ ]**of size **N**consisting of **distinct** integers, write a program that returns **1** if given array can represent preorder traversal of a possible BST, else returns**0**.

**Example 1:**

**Input:**

N = 3

arr = {2, 4, 3}

**Output:** 1

**Explaination:** Given arr[] can represent

preorder traversal of following BST:

  2

  \

  4

  /

  3

**Example 2:**

**Input:**

N = 3

Arr = {2, 4, 1}

**Output:** 0

**Explaination:** Given arr[] cannot represent

preorder traversal of a BST.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **canRepresentBST()** which takes the array a**rr[]** and its size **N**as input parameters and returns **1** if given array can represent preorder traversal of a BST, else returns**0**.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 ≤ N ≤ 105  
0 ≤ arr[i] ≤ 105

## Solution:

A **Simple Solution** is to do following for every node pre[i] starting from first one.

1) Find the first greater value on right side of current node.

Let the index of this node be j. Return true if following

conditions hold. Else return false

(i) All values after the above found greater value are

greater than current node.

(ii) Recursive calls for the subarrays pre[i+1..j-1] and

pre[j+1..n-1] also return true.

Time Complexity of the above solution is O(n2)

An **Efficient Solution** can solve this problem in O(n) time. The idea is to use a stack. This problem is similar to [Next (or closest) Greater Element problem](https://www.geeksforgeeks.org/next-greater-element/). Here we find the next greater element and after finding next greater, if we find a smaller element, then return false.

1) Create an empty stack.

2) Initialize root as INT\_MIN.

3) Do following for every element pre[i]

a) If pre[i] is smaller than current root, return false.

b) Keep removing elements from stack while pre[i] is greater

then stack top. Make the last removed item as new root (to

be compared next).

At this point, pre[i] is greater than the removed root

(That is why if we see a smaller element in step a), we

return false)

c) push pre[i] to stack (All elements in stack are in decreasing

order)

Below is the implementation of above idea.

// C++ program for an efficient solution to check if

// a given array can represent Preorder traversal of

// a Binary Search Tree

#include<bits/stdc++.h>

using namespace std;

bool canRepresentBST(int pre[], int n)

{

// Create an empty stack

stack<int> s;

// Initialize current root as minimum possible

// value

int root = INT\_MIN;

// Traverse given array

for (int i=0; i<n; i++)

{

// If we find a node who is on right side

// and smaller than root, return false

if (pre[i] < root)

return false;

// If pre[i] is in right subtree of stack top,

// Keep removing items smaller than pre[i]

// and make the last removed item as new

// root.

while (!s.empty() && s.top()<pre[i])

{

root = s.top();

s.pop();

}

// At this point either stack is empty or

// pre[i] is smaller than root, push pre[i]

s.push(pre[i]);

}

return true;

}

// Driver program

int main()

{

int pre1[] = {40, 30, 35, 80, 100};

int n = sizeof(pre1)/sizeof(pre1[0]);

canRepresentBST(pre1, n)? cout << "true\n":

cout << "false\n";

int pre2[] = {40, 30, 35, 20, 80, 100};

n = sizeof(pre2)/sizeof(pre2[0]);

canRepresentBST(pre2, n)? cout << "true\n":

cout << "false\n";

return 0;

}

**Output**

true

false

# 216. Check whether BST contains Dead end

Given a [Binary search Tree](http://quiz.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/) that contains positive integer values greater then 0. The task is to complete the function **isDeadEnd** which returns true if the BST contains a dead end else returns false. Here Dead End means, we are not able to insert any element after that node.

Examples:

Input :

  8

/ \

5 9

/ \

2 7

/

1

Output : Yes

Explanation : Node "1" is the dead End because after that

  we cant insert any element.

Input :

  8

/ \

7 10

/ / \

2 9 13

Output : Yes

Explanation : We can't insert any element at

node 9.

**Input:**  
The first line of the input contains an integer 'T' denoting the number of test cases. Then 'T' test cases follow. Each test case consists of three lines. First line of each test case contains an integer N denoting the no of nodes of the BST . Second line of each test case consists of 'N' space separated integers denoting the elements of the BST. These elements are inserted into BST in the given order.  
  
**Output:**  
The output for each test case will be 1 if the BST contains a dead end else 0.

**Constraints:**  
1<=T<=100  
1<=N<=200

**Example(To be used only for expected output):**  
**Input:**  
2  
6  
8 5 9 7 2 1  
6  
8 7 10 9 13 2  
**Output:**  
1  
1

## Solution:

If we take a closer look at the problem, we can notice that we basically need to check if there is a leaf node with value x such that x+1 and x-1 exist in BST with the exception of x = 1. For x = 1, we can’t insert 0 as the problem statement says BST contains positive integers only.  
To implement the above idea we first traverse the whole BST and store all nodes in a set. We also store all leaves in a separate hash to avoid re-traversal of BST. Finally, we check for every leaf node x, if x-1 and x+1 are present in set or not.  
Below is a C++ implementation of the above idea.

// C++ program check weather BST contains

// dead end or not

#include<bits/stdc++.h>

using namespace std;

// A BST node

struct Node

{

int data;

struct Node \*left, \*right;

};

// A utility function to create a new node

Node \*newNode(int data)

{

Node \*temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

/\* A utility function to insert a new Node

with given key in BST \*/

struct Node\* insert(struct Node\* node, int key)

{

/\* If the tree is empty, return a new Node \*/

if (node == NULL) return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->data)

node->left = insert(node->left, key);

else if (key > node->data)

node->right = insert(node->right, key);

/\* return the (unchanged) Node pointer \*/

return node;

}

// Function to store all node of given binary search tree

void storeNodes(Node \* root, unordered\_set<int> &all\_nodes,

unordered\_set<int> &leaf\_nodes)

{

if (root == NULL)

return ;

// store all node of binary search tree

all\_nodes.insert(root->data);

// store leaf node in leaf\_hash

if (root->left==NULL && root->right==NULL)

{

leaf\_nodes.insert(root->data);

return ;

}

// recur call rest tree

storeNodes(root-> left, all\_nodes, leaf\_nodes);

storeNodes(root->right, all\_nodes, leaf\_nodes);

}

// Returns true if there is a dead end in tree,

// else false.

bool isDeadEnd(Node \*root)

{

// Base case

if (root == NULL)

return false ;

// create two empty hash sets that store all

// BST elements and leaf nodes respectively.

unordered\_set<int> all\_nodes, leaf\_nodes;

// insert 0 in 'all\_nodes' for handle case

// if bst contain value 1

all\_nodes.insert(0);

// Call storeNodes function to store all BST Node

storeNodes(root, all\_nodes, leaf\_nodes);

// Traversal leaf node and check Tree contain

// continuous sequence of

// size tree or Not

for (auto i = leaf\_nodes.begin() ; i != leaf\_nodes.end(); i++)

{

int x = (\*i);

// Here we check first and last element of

// continuous sequence that are x-1 & x+1

if (all\_nodes.find(x+1) != all\_nodes.end() &&

all\_nodes.find(x-1) != all\_nodes.end())

return true;

}

return false ;

}

// Driver program

int main()

{

/\* 8

/ \

5 11

/ \

2 7

\

3

\

4 \*/

Node \*root = NULL;

root = insert(root, 8);

root = insert(root, 5);

root = insert(root, 2);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 11);

root = insert(root, 4);

if (isDeadEnd(root) == true)

cout << "Yes " << endl;

else

cout << "No " << endl;

return 0;

}

Output: 

Yes

Time Complexity : O(n)

Space Complexity: O(n)

**My approach:**

bool fun(Node\* root, stack<int> &st, int &min){

if(!root)

return false;

int max = st.top();

if(max==1 || max-min==1)

return true;

if(!root->left && !root->right){

if((root->data-min==1 || root->data==1) && max-root->data==1)

return true;

return false;

}

st.push(root->data);

bool l = fun(root->left, st, min);

if(l)

return l;

min = st.top();

st.pop();

return fun(root->right, st, min);

}

Time Complexity : O(n)

Space Complexity: O(n)

# 217. Largest BST in a Binary Tree [ V.V.V.V.V IMP ]

Given a binary tree. Find the size of its largest subtree that is a Binary Search Tree.  
**Note:**Here Size is equal to the number of nodes in the subtree.

**Example 1:**

**Input:**

  1

  / \

  4 4

  / \

  6 8

**Output:** 1

**Explanation:** There's no sub-tree with size

greater than 1 which forms a BST. All the

leaf Nodes are the BSTs with size equal

to 1.

**Example 2:**

**Input:** 6 6 3 N 2 9 3 N 8 8 2

  6

  / \

  6 3

  \ / \

  2 9 3

  \ / \

  8 8 2

**Output:** 2

**Explanation:** The following sub-tree is a

BST of size 2:

      2

    /    \

  N      8

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function**largestBst()**that takes the root node of the Binary Treeas its input and returns the size of the largest subtree which is also the BST. If the complete Binary Tree is a BST, return the size of the complete Binary Tree.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the BST).

**Constraints:**  
1 ≤ Number of nodes ≤ 105  
1 ≤ Data of a node ≤ 106

## Solution:

**Method 1 (Simple but inefficient)**   
Start from root and do an inorder traversal of the tree. For each node N, check whether the subtree rooted with N is BST or not. If BST, then return size of the subtree rooted with N. Else, recur down the left and right subtrees, and return the maximum of values returned by left and right subtrees.

/\*

See https://www.geeksforgeeks.org/write-a-c-program-to-calculate-size-of-a-tree/ for implementation of size()

See Method 3 of https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/ for

implementation of isBST()

max() returns maximum of two integers

\*/

int largestBST(struct node \*root)

{

// Base Case

if(root == NULL)

return 0;

if (isBST(root))

return size(root);

else

return max(largestBST(root->left), largestBST(root->right));

}

Time Complexity: The worst-case time complexity of this method will be O(n^2). Consider a skewed tree for worst case analysis.

**Method 2 (Tricky and Efficient)**   
In method 1, we traverse the tree in top-down manner and do BST test for every node. If we traverse the tree in bottom-up manner, then we can pass information about subtrees to the parent. The passed information can be used by the parent to do BST test (for parent node) only in constant time (or O(1) time). A left subtree need to tell the parent whether it is BST or not and also needs to pass maximum value in it. So that we can compare the maximum value with the parent’s data to check the BST property. Similarly, the right subtree need to pass the minimum value up the tree. The subtrees need to pass the following information up the tree for the finding the largest BST.   
1) Whether the subtree itself is BST or not (In the following code, is\_bst\_ref is used for this purpose)   
2) If the subtree is left subtree of its parent, then maximum value in it. And if it is right subtree then minimum value in it.   
3) Size of this subtree if this subtree is BST (In the following code, return value of largestBSTtil() is used for this purpose)  
max\_ref is used for passing the maximum value up the tree and min\_ptr is used for passing minimum value up the tree. 

// C++ program of above approach

#include<bits/stdc++.h>

using namespace std;

/\* A binary tree node has data,

pointer to left child and a pointer

to right child \*/

class node

{

public:

int data;

node\* left;

node\* right;

/\* Constructor that allocates

a new node with the given data

and NULL left and right pointers. \*/

node(int data)

{

this->data = data;

this->left = NULL;

this->right = NULL;

}

};

int largestBSTUtil(node\* node, int \*min\_ref, int \*max\_ref,

int \*max\_size\_ref, bool \*is\_bst\_ref);

/\* Returns size of the largest BST

subtree in a Binary Tree

(efficient version). \*/

int largestBST(node\* node)

{

// Set the initial values for

// calling largestBSTUtil()

int min = INT\_MAX; // For minimum value in right subtree

int max = INT\_MIN; // For maximum value in left subtree

int max\_size = 0; // For size of the largest BST

bool is\_bst = 0;

largestBSTUtil(node, &min, &max,

&max\_size, &is\_bst);

return max\_size;

}

/\* largestBSTUtil() updates \*max\_size\_ref

for the size of the largest BST subtree.

Also, if the tree rooted with node is

non-empty and a BST, then returns size

of the tree. Otherwise returns 0.\*/

int largestBSTUtil(node\* node, int \*min\_ref, int \*max\_ref,

int \*max\_size\_ref, bool \*is\_bst\_ref)

{

/\* Base Case \*/

if (node == NULL)

{

\*is\_bst\_ref = 1; // An empty tree is BST

return 0; // Size of the BST is 0

}

int min = INT\_MAX;

/\* A flag variable for left subtree property

i.e., max(root->left) < root->data \*/

bool left\_flag = false;

/\* A flag variable for right subtree property

i.e., min(root->right) > root->data \*/

bool right\_flag = false;

int ls, rs; // To store sizes of left and right subtrees

/\* Following tasks are done by

recursive call for left subtree

a) Get the maximum value in left

subtree (Stored in \*max\_ref)

b) Check whether Left Subtree is

BST or not (Stored in \*is\_bst\_ref)

c) Get the size of maximum size BST

in left subtree (updates \*max\_size) \*/

\*max\_ref = INT\_MIN;

ls = largestBSTUtil(node->left, min\_ref, max\_ref,

max\_size\_ref, is\_bst\_ref);

if (\*is\_bst\_ref == 1 && node->data > \*max\_ref)

left\_flag = true;

/\* Before updating \*min\_ref, store the min

value in left subtree. So that we have the

correct minimum value for this subtree \*/

min = \*min\_ref;

/\* The following recursive call

does similar (similar to left subtree)

task for right subtree \*/

\*min\_ref = INT\_MAX;

rs = largestBSTUtil(node->right, min\_ref,

max\_ref, max\_size\_ref, is\_bst\_ref);

if (\*is\_bst\_ref == 1 && node->data < \*min\_ref)

right\_flag = true;

// Update min and max values for

// the parent recursive calls

if (min < \*min\_ref)

\*min\_ref = min;

if (node->data < \*min\_ref) // For leaf nodes

\*min\_ref = node->data;

if (node->data > \*max\_ref)

\*max\_ref = node->data;

/\* If both left and right subtrees are BST.

And left and right subtree properties hold

for this node, then this tree is BST.

So return the size of this tree \*/

if(left\_flag && right\_flag)

{

if (ls + rs + 1 > \*max\_size\_ref)

\*max\_size\_ref = ls + rs + 1;

return ls + rs + 1;

}

else

{

// Since this subtree is not BST,

// set is\_bst flag for parent calls

\*is\_bst\_ref = 0;

return 0;

}

}

/\* Driver code\*/

int main()

{

/\* Let us construct the following Tree

50

/ \

10 60

/ \ / \

5 20 55 70

/ / \

45 65 80

\*/

node \*root = new node(50);

root->left = new node(10);

root->right = new node(60);

root->left->left = new node(5);

root->left->right = new node(20);

root->right->left = new node(55);

root->right->left->left = new node(45);

root->right->right = new node(70);

root->right->right->left = new node(65);

root->right->right->right = new node(80);

/\* The complete tree is not BST

as 45 is in right subtree of 50.

The following subtree is the largest BST

60

/ \

55 70

/ / \

45 65 80

\*/

cout<<" Size of the largest BST is "<< largestBST(root);

return 0;

}

**Output:**

Size of largest BST is 6

**Time Complexity:** O(n) where n is the number of nodes in the given Binary Tree.

**Approach 3:**

A Tree is BST if following is true for every node x.

1. The largest value in left subtree (of x) is smaller than value of x.
2. The smallest value in right subtree (of x) is greater than value of x.

We traverse tree in bottom up manner. For every traversed node, we return maximum and minimum values in subtree rooted with it. If any node follows above properties and size of

 // C++ program to find largest BST in a

// Binary Tree.

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data,

pointer to left child and a

pointer to right child \*/

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

};

/\* Helper function that allocates a new

node with the given data and NULL left

and right pointers. \*/

struct Node\* newNode(int data)

{

struct Node\* node = new Node;

node->data = data;

node->left = node->right = NULL;

return(node);

}

// Information to be returned by every

// node in bottom up traversal.

struct Info

{

int sz; // Size of subtree

int max; // Min value in subtree

int min; // Max value in subtree

int ans; // Size of largest BST which

// is subtree of current node

bool isBST; // If subtree is BST

};

// Returns Information about subtree. The

// Information also includes size of largest

// subtree which is a BST.

Info largestBSTBT(Node\* root)

{

// Base cases : When tree is empty or it has

// one child.

if (root == NULL)

return {0, INT\_MIN, INT\_MAX, 0, true};

if (root->left == NULL && root->right == NULL)

return {1, root->data, root->data, 1, true};

// Recur for left subtree and right subtrees

Info l = largestBSTBT(root->left);

Info r = largestBSTBT(root->right);

// Create a return variable and initialize its

// size.

Info ret;

ret.sz = (1 + l.sz + r.sz);

// If whole tree rooted under current root is

// BST.

if (l.isBST && r.isBST && l.max < root->data &&

r.min > root->data)

{

ret.min = min(l.min, min(r.min, root->data));

ret.max = max(r.max, max(l.max, root->data));

// Update answer for tree rooted under

// current 'root'

ret.ans = ret.sz;

ret.isBST = true;

return ret;

}

// If whole tree is not BST, return maximum

// of left and right subtrees

ret.ans = max(l.ans, r.ans);

ret.isBST = false;

return ret;

}

/\* Driver program to test above functions\*/

int main()

{

/\* Let us construct the following Tree

60

/ \

65 70

/

50 \*/

struct Node \*root = newNode(60);

root->left = newNode(65);

root->right = newNode(70);

root->left->left = newNode(50);

printf(" Size of the largest BST is %d\n",

largestBSTBT(root).ans);

return 0;

}

**Output**

Size of the largest BST is 2

**Time Complexity :** O(n)

**My Implementation:**

class Solution{

public:

bool fun(Node\* root, int &size, int &maxi, int &mini, int &max\_size){

if(!root){

maxi = 0;

mini = INT\_MAX;

size = 0;

return true;

}

int l\_size=0, r\_size=0, l\_max=0, r\_max=0, l\_min=INT\_MAX, r\_min=INT\_MAX;

bool l,r;

l = fun(root->left, l\_size, l\_max, l\_min, max\_size);

r = fun(root->right, r\_size, r\_max, r\_min, max\_size);

size = l\_size + r\_size + 1;

maxi = max(max(l\_max, root->data), r\_max);

mini = min(min(l\_min, root->data), r\_min);

if(l && r && (l\_max < root->data) && (root->data < r\_min) ){

max\_size = max(max\_size, size);

return true;

}

return false;

}

/\*You are required to complete this method \*/

// Return the size of the largest sub-tree which is also a BST

int largestBst(Node \*root)

{

int size = 0, maxi=0, mini=INT\_MAX, max\_size = 0;

fun(root, size, maxi, mini, max\_size);

return max\_size;

}

};

**Time Complexity :** O(n)

# 218. Flatten BST to sorted list

Given a binary search tree, the task is to flatten it to a sorted list. Precisely, the value of each node must be lesser than the values of all the nodes at its right, and its left node must be NULL after flattening. We must do it in O(H) extra space where ‘H’ is the height of BST.

**Examples:**

**Input:**

5

/ \

3 7

/ \ / \

2 4 6 8

**Output:** 2 3 4 5 6 7 8

**Input:**

1

\

2

\

3

\

4

\

5

**Output:** 1 2 3 4 5

## Solution:

**Approach:** A simple approach will be to recreate the BST from its [in-order](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) traversal. This will take O(N) extra space where N is the number of nodes in BST.

To improve upon that, we will simulate in-order traversal of a binary tree as follows:

1. Create a dummy node.
2. Create a variable called ‘prev’ and make it point to the dummy node.
3. Perform in-order traversal and at each step.
   * Set prev -> right = curr
   * Set prev -> left = NULL
   * Set prev = curr

This will improve the space complexity to O(H) in worst case as in-order traversal takes O(H) extra space.

Below is the implementation of the above approach:

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

// Node of the binary tree

struct node {

int data;

node\* left;

node\* right;

node(int data)

{

this->data = data;

left = NULL;

right = NULL;

}

};

// Function to print flattened

// binary Tree

void print(node\* parent)

{

node\* curr = parent;

while (curr != NULL)

cout << curr->data << " ", curr = curr->right;

}

// Function to perform in-order traversal

// recursively

void inorder(node\* curr, node\*& prev)

{

// Base case

if (curr == NULL)

return;

inorder(curr->left, prev);

prev->left = NULL;

prev->right = curr;

prev = curr;

inorder(curr->right, prev);

}

// Function to flatten binary tree using

// level order traversal

node\* flatten(node\* parent)

{

// Dummy node

node\* dummy = new node(-1);

// Pointer to previous element

node\* prev = dummy;

// Calling in-order traversal

inorder(parent, prev);

prev->left = NULL;

prev->right = NULL;

node\* ret = dummy->right;

// Delete dummy node

delete dummy;

return ret;

}

// Driver code

int main()

{

node\* root = new node(5);

root->left = new node(3);

root->right = new node(7);

root->left->left = new node(2);

root->left->right = new node(4);

root->right->left = new node(6);

root->right->right = new node(8);

// Calling required function

print(flatten(root));

return 0;

}

**Output:**

2 3 4 5 6 7 8

**Time Complexity:** O(N)