Bit Manipulation

**Tricks With Bits**

* x & (x-1) will clear the lowest set bit of x
* x & ~(x-1) extracts the lowest set bit of x (all others are clear). Pretty patterns when applied to a linear sequence.
* x & (x + (1 << n)) = x, with the run of set bits (possibly length 0) starting at bit n cleared.
* x & ~(x + (1 << n)) = the run of set bits (possibly length 0) in x, starting at bit n.
* x | (x + 1) = x with the lowest cleared bit set.
* x | ~(x + 1) = extracts the lowest cleared bit of x (all others are set).
* x | (x - (1 << n)) = x, with the run of cleared bits (possibly length 0) starting at bit n set.
* x | ~(x - (1 << n)) = the lowest run of cleared bits (possibly length 0) in x, starting at bit n are the only clear bits.

## Reverse Bits

* Asked in:
* [Nvidia](https://www.interviewbit.com/search/?q=Nvidia)
* [HCL](https://www.interviewbit.com/search/?q=HCL)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

unsigned int Solution::reverse(unsigned int A) {

// Do not write main() function.

// Do not read input, instead use the arguments to the function.

// Do not print the output, instead return values as specified unsigned int res=0;

int i=0;

while(A){

if(A%2){

res=res+pow(2,(31-i));

}

A/=2;

i++;

}

return res;

}

## Single Number II

* Asked in:
* [Google](https://www.interviewbit.com/search/?q=Google)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

int Solution::singleNumber(const vector<int> &A) {

int ones=0, twos=0, threes=0;

for(int i=0; i<A.size(); i++){

twos |=ones & A[i];

ones= ones ^ A[i];

threes= ones & twos;

ones &=~threes;

twos &=~threes;

}

return ones;

}

## Single Number I

**class** **Solution** {

**public:**

**unsigned** **int** numSetBits(**unsigned** **int** x) {

**unsigned** **int** total\_ones **=** 0;

**while** (x **!=** 0) {

x **=** x **&** (x**-**1);

total\_ones**++**;

}

**return** total\_ones;

}

};

# Trees

## Next Greater Number BST / Inorder Successor

Given a BST node, return the node which has value just greater than the given node.

**1)** If right subtree of node is not NULL, then succ lies in right subtree. Do following.  
Go to right subtree and return the node with minimum key value in right subtree.  
**2)**If right sbtree of node is NULL, then start from root and use search like technique. Do following.  
Travel down the tree, if a node’s data is greater than root’s data then go right side, otherwise go to left side.

TreeNode**\*** Solution**::**getSuccessor(TreeNode**\***root, **int** B) {

TreeNode **\*** curr **=** root;

TreeNode **\***suc **=** NULL;

**while**(curr)

{

**if**(curr**->**val **>** B)

{

suc **=** curr;

curr **=** curr**->**left;

}**else**

{

curr **=** curr**->**right;

}

}

**return** suc;

}

Here condition 1) is also covered in condition 2).

## Valid BST

Efficient Method 1:

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

}

Method 2:

int leftMax(TreeNode \*node){

if(!node) return INT\_MIN;

while(node->right)

node=node->right;

return node->val;

}

int rightMin(TreeNode \*node){

if(!node) return INT\_MAX;

while(node->left)

node=node->left;

return node->val;

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::isValidBST(TreeNode\* root) {

if(!root) return true; //Null tree is valid BST

if(root->left && root->left->val>=root->val) return false;

if(root->right && root->right->val<=root->val) return false;

if(!isValidBST(root->left) || !(isValidBST(root->right)))

return false;

int lmax=leftMax(root->left);

int rmin=rightMin(root->right);

if(lmax>=root->val || rmin<=root->val) return false;

return true;

}

## Inorder without recursion

Method 1:

/\* Iterative function for inorder tree

   traversal \*/

void inOrder(struct Node \*root)

{

    stack<Node \*> s;

    Node \*curr = root;

    while (curr != NULL || s.empty() == false)

    {

        /\* Reach the left most Node of the

           curr Node \*/

        while (curr !=  NULL)

        {

            /\* place pointer to a tree node on

               the stack before traversing

              the node's left subtree \*/

            s.push(curr);

            curr = curr->left;

        }

        /\* Current must be NULL at this point \*/

        curr = s.top();

        s.pop();

        cout << curr->data << " ";

        /\* we have visited the node and its

           left subtree.  Now, it's right

           subtree's turn \*/

        curr = curr->right;

    } /\* end of while \*/

}

/\* Function to traverse the binary tree without recursion and

   without stack \*/

void MorrisTraversal(struct tNode\* root)

{

    struct tNode \*current, \*pre;

    if (root == NULL)

        return;

    current = root;

    while (current != NULL) {

        if (current->left == NULL) {

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

            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 the right child of its inorder

               predecessor \*/

            if (pre->right == NULL) {

                pre->right = current;

                current = current->left;

            }

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

               the original tree i.e., fix the right child

               of predecessor \*/

            else {

                pre->right = NULL;

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

                current = current->right;

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

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

    } /\* End of while \*/

}

## Preorder Traversal

// An iterative process to print preorder traversal of Binary tree

void iterativePreorder(node \*root)

{

    // Base Case

    if (root == NULL)

       return;

    // Create an empty stack and push root to it

    stack<node \*> nodeStack;

    nodeStack.push(root);

    /\* Pop all items one by one. Do following for every popped item

       a) print it

       b) push its right child

       c) push its left child

    Note that right child is pushed first so that left is processed first \*/

    while (nodeStack.empty() == false)

    {

        // Pop the top item from stack and print it

        struct node \*node = nodeStack.top();

        printf ("%d ", node->data);

        nodeStack.pop();

        // Push right and left children of the popped node to stack

        if (node->right)

            nodeStack.push(node->right);

        if (node->left)

            nodeStack.push(node->left);

    }

}

## Postorder Traversal

Asked in:

* [Amazon](https://www.interviewbit.com/search/?q=Amazon)
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)

BookmarkSuggest Edit

Given a binary tree, return the postorder traversal of its nodes’ values.

**Example :**

Given binary tree

1

\

2

/

3

return [3,2,1].

**Using recursion is not allowed.**

void postOrderIterative(Node\* root)

{

    if (root == NULL)

        return;

    // Create two stacks

    stack<Node \*> s1, s2;

    // push root to first stack

    s1.push(root);

    Node\* node;

    // Run while first stack is not empty

    while (!s1.empty()) {

        // Pop an item from s1 and push it to s2

        node = s1.top();

        s1.pop();

        s2.push(node);

        // Push left and right children

        // of removed item to s1

        if (node->left)

            s1.push(node->left);

        if (node->right)

            s1.push(node->right);

    }

    // Print all elements of second stack

    while (!s2.empty()) {

        node = s2.top();

        s2.pop();

        cout << node->data << " ";

    }

}

Method 2:

Create an empty stack, Push root node to the stack.  
Do following while stack is not empty.

2.1. pop an item from the stack and print it.

2.2. push the left child of popped item to stack.

2.3. push the right child of popped item to stack.

class Solution {

public:

vector<int> postorderTraversal(TreeNode \*root) {

stack<TreeNode\*> nodeStack;

vector<int> result;

//base case

if(root==NULL)

return result;

nodeStack.push(root);

while(!nodeStack.empty()) {

TreeNode\* node= nodeStack.top();

result.push\_back(node->val);

nodeStack.pop();

if(node->left)

nodeStack.push(node->left);

if(node->right)

nodeStack.push(node->right);

}

reverse(result.begin(),result.end());

return result;

}

## Vertical Order traversal of Binary Tree

Asked in:

* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

Problem Setter: yashpal1995 Problem Tester: [RAMBO\_tejasv](https://www.interviewbit.com/profile/RAMBO_tejasv)

BookmarkSuggest Edit

Given a binary tree, return a 2-D array with vertical order traversal of it.  
Go through the example and image for more details.

**Example :**  
Given binary tree:

6

/ \

3 7

/ \ \

2 5 9

returns

[

[2],

[3],

[6 5],

[7],

[9]

]

*/\**

*\* This problem can also be done in O(n) instead of O(n \* logn) by adding*

*\* one more traversal to compute minimum & maximum horizontal distance, start root's distance with (-minimumDistance) instead of '0'.*

*\*/*

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

vector**<**vector**<int>** **>** Solution**::**verticalOrderTraversal(TreeNode**\*** A) {

vector**<**vector**<int>** **>** result;

*// Base case*

**if** (**!**A) **return** result;

*// Create a map of horizontalDistance, ListOfTreeNodes(at that horizontalDistance)*

map **<int**,vector**<int>** **>** m;

*// Create queue to do level order traversal.*

*// Every item of queue contains node and horizontal distance.*

queue**<**pair**<**TreeNode**\***, **int>** **>** q;

q.push(make\_pair(A, 0)); *// root's horizontalDistance is zero*

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

*// get the queue front*

pair**<**TreeNode**\***,**int>** front **=** q.front();

q.pop();

**int** currentHorizontalDistance **=** front.second;

TreeNode**\*** currentTreeNode **=** front.first;

*// insert current node to hash map*

m[currentHorizontalDistance].push\_back(currentTreeNode **->** val);

**if** (currentTreeNode **->** left **!=** NULL) {

q.push(make\_pair(currentTreeNode **->** left, currentHorizontalDistance **-** 1));

}

**if** (currentTreeNode **->** right **!=** NULL) {

q.push(make\_pair(currentTreeNode **->** right, currentHorizontalDistance **+** 1));

}

}

*// Traverse the map and print nodes at every horigontal distance*

**for** (map**<int**,vector**<int>** **>** **::** iterator it **=** m.begin(); it **!=** m.end(); it**++**) {

result.push\_back(it **->** second);

}

**return** result;

}

## ZigZag Level Order Traversal BT

Asked in:

* [Amazon](https://www.interviewbit.com/search/?q=Amazon)
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)

Given a binary tree, return the zigzag level order traversal of its nodes’ values. (ie, from left to right, then right to left for the next level and alternate between).

**Example :**   
Given binary tree

3

/ \

9 20

/ \

15 7

return

[

[3],

[20, 9],

[15, 7]

]

Sol1:

\* };

\*/

vector<vector<int> > Solution::zigzagLevelOrder(TreeNode\* root) {

vector<vector<int>> res;

if(!root) return res;

map<int, vector<int>> m;

queue<pair<TreeNode\*,int>> q;

q.push(make\_pair(root,0));

while(!q.empty()){

pair<TreeNode\*,int> front=q.front();

TreeNode \*node=front.first;

int level=front.second;

q.pop();

m[level].push\_back(node->val);

if(node->left)

q.push(make\_pair(node->left, level+1));

if(node->right)

q.push(make\_pair(node->right, level+1));

}

for(map<int, vector<int>>::iterator it=m.begin(); it!=m.end(); ++it){

if(it->first%2)

reverse(it->second.begin(), it->second.end());

res.push\_back(it->second);

}

return res;

}

Sol2: Simple

//Sol 2:

vector<vector<int>> res;

if(!root) return res;

stack<TreeNode\*> s1, s2;

int level=1;

s1.push(root);

vector<int> vec;

while(!s1.empty()){

TreeNode \*node=s1.top();

s1.pop();

vec.push\_back(node->val);

if(!level){

if(node->right) s2.push(node->right);

if(node->left) s2.push(node->left);

}

else{

if(node->left) s2.push(node->left);

if(node->right) s2.push(node->right);

}

if(s1.empty()){

s1.swap(s2);

level ^=1;

res.push\_back(vec);

vec.clear();

}

}

return res;

## Populate Next Right Pointers Tree / Connect nodes at same level

* Asked in:
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

BookmarkSuggest Edit

Given a binary tree

**struct** TreeLinkNode {

TreeLinkNode **\***left;

TreeLinkNode **\***right;

TreeLinkNode **\***next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**Example :**

Given the following binary tree,

1

/ \

2 3

/ \ / \

4 5 6 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

*/\*\**

*\* Definition for binary tree with next pointer.*

*\* struct TreeLinkNode {*

*\* int val;*

*\* TreeLinkNode \*left, \*right, \*next;*

*\* TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}*

*\* };*

*\*/*

**void** Solution**::**connect(TreeLinkNode**\*** root) {

**while**(root)//loop run for number of levels times

{

TreeLinkNode**\*** s**=new** TreeLinkNode(0);

TreeLinkNode**\*** n**=**s;

**while**(root) //loop to traverse all nodes at a level

{

**if**(root**->**left)

{

n**->**next**=**root**->**left;

n**=**n**->**next;

}

**if**(root**->**right)

{

n**->**next**=**root**->**right;

n**=**n**->**next;

}

root**=**root**->**next;

}

root**=**s**->**next;

}

}

Sol2:

void Solution::connect(TreeLinkNode\* root) {

**if**(!root) return;

queue<TreeLinkNode\*>s1, s2;

s1.push(root);

**while**(!s1.empty()){

TreeLinkNode \*node=s1.front();

s1.pop();

**if**(node->left) s2.push(node->left);

**if**(node->right) s2.push(node->right);

**if**(s1.empty()){

node->next=NULL;

s1.swap(s2);

}

**else**{

node->next=s1.front();

}

}

}

## Balanced Binary Tree

* Asked in:
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

BookmarkSuggest Edit

Given a binary tree, determine if it is height-balanced.

***Height-balanced binary tree****: is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.*

Return 0 / 1 ( 0 for false, 1 for true ) for this problem

bool balance\_flag;

// depth function of binary tree is modified here

int depthCheck(TreeNode \*node){

if(!balance\_flag) return 0;

if(!node) return 0;

int leftdepth=depthCheck(node->left);

int rightdepth=depthCheck(node->right);

if(abs(leftdepth-rightdepth) >= 2){

balance\_flag=false;

return 0;

}

return 1+max(leftdepth,rightdepth);

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::isBalanced(TreeNode\* root) {

balance\_flag=true;

depthCheck(root);

return balance\_flag;

}

## Identical Binary Trees

* Asked in:
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

BookmarkSuggest Edit

Given two binary trees, write a function to check if they are equal or not.

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

Return 0 / 1 ( 0 for false, 1 for true ) for this problem

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::isSameTree(TreeNode\* A, TreeNode\* B) {

if(!A && !B) return 1;

if(!A || !B) return 0; //1&&1 is already covered above

return (A->val==B->val && isSameTree(A->left,B->left) && isSameTree(A->right, B->right));

}

## Symmetric Binary Tree Asked in:

* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

BookmarkSuggest Edit

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

**Example :**

1

/ \

2 2

/ \ / \

3 4 4 3

The above binary tree is symmetric.   
But the following is not:

1

/ \

2 2

\ \

3 3

//Solution

int isMirror(TreeNode \*A, TreeNode \*B){

if(!A && !B) return 1;

if(!A || !B) return 0;

return ((A->val == B->val) && isMirror(A->left, B->right) && isMirror(A->right, B->left));

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::isSymmetric(TreeNode\* A) {

if(!A) return 1;

return isMirror(A->left, A->right);

}

## Inorder Traversal of Cartesian Tree

Asked in:

* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

BookmarkSuggest Edit

Given an inorder traversal of a cartesian tree, construct the tree.

***Cartesian tree****: is a heap ordered binary tree, where the root is greater than all the elements in the subtree.*

***Note:****You may assume that duplicates do not exist in the tree.*

**Example :**

Input : [1 2 3]

Return :

3

/

2

/

1

void printTree(TreeNode \* root){

if (!root) return;

printTree(root->left);

cout<<root->val<<" ";

printTree(root->right);

}

int findMax(vector<int> &A, int l, int r){

int max\_idx=l;

for(int i=l; i<=r; i++)

max\_idx=(A[max\_idx]>A[i])?max\_idx:i;

return max\_idx;

}

TreeNode \* formTree(vector<int> &A, int l, int r){

TreeNode \*root=NULL;

if(l <= r){

int root\_idx = findMax(A, l, r);

root=new TreeNode(A[root\_idx]);

root->left=formTree(A,l, root\_idx-1);

root->right=formTree(A, root\_idx+1, r);

}

return root;

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

TreeNode\* Solution::buildTree(vector<int> &A) {

TreeNode\* root=formTree(A, 0, A.size()-1);

return root;

}

## Binary Tree From Inorder And Postorder

* Asked in:
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)

Given inorder and postorder traversal of a tree, construct the binary tree.

***Note:****You may assume that duplicates do not exist in the tree.*

**Example :**

Input :

Inorder : [2, 1, 3]

Postorder : [2, 3, 1]

Return :

1

/ \

2 3

TreeNode\* buildTreeUtil(vector<int> &in, int l, int r, vector<int> &po, int &pIndex, map<int,int> &m){

if(l>r) return NULL;

int root\_val=po[pIndex];//root of subtree found

TreeNode \*root= new TreeNode(root\_val); //create root node

(pIndex)--; //do not consider this index in further BT creation

if(l==r) return root;

int iIndex=m[root\_val];

root->right=buildTreeUtil(in, iIndex+1, r, po, pIndex, m);

root->left=buildTreeUtil(in, l, iIndex-1, po, pIndex, m);

return root;

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

TreeNode\* Solution::buildTree(vector<int> &inOrder, vector<int> &postOrder) {

map<int,int> m; //mapping inorder elements to their indices

//for easy search

int n=inOrder.size();

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

m[inOrder[i]]=i;

int pIndex=n-1; //post order index where root lies

return buildTreeUtil(inOrder, 0, n-1, postOrder, pIndex, m);

}

## Construct Binary Tree From Inorder And Preorder

* Asked in:
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)

BookmarkSuggest Edit

Given preorder and inorder traversal of a tree, construct the binary tree.

***Note:****You may assume that duplicates do not exist in the tree.*

**Example :**

Input :

Preorder : [1, 2, 3]

Inorder : [2, 1, 3]

Return :

1

/ \

2 3

TreeNode \*buildTreeUtil(vector<int> &in, int l, int r, vector<int> &pre, int &preIndex, map<int,int> &m){

if(l > r) return NULL;

int root\_val=pre[preIndex];

TreeNode \* root=new TreeNode(root\_val);

preIndex++;

if(l==r) return root;

int iIndex=m[root\_val];

root->left=buildTreeUtil(in, l, iIndex-1, pre, preIndex, m);

//cout<<root->left->val<<endl;

root->right=buildTreeUtil(in, iIndex+1, r, pre, preIndex, m);

return root;

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

TreeNode\* Solution::buildTree(vector<int> &pre, vector<int> &in) {

int preIndex=0;

int n=pre.size();

map<int,int> m;

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

m[in[i]]=i;

}

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

}

## Path Sum

* Asked in:
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)
* [Yahoo](https://www.interviewbit.com/search/?q=Yahoo)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

**Example :**

Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

**Return 0 / 1 ( 0 for false, 1 for true ) for this problem**

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::hasPathSum(TreeNode\* root, int sum) {

if(!root) return (sum==0);

if((sum==root->val) && !root->left && !root->right) return 1;

int ans=0;

if(root->left)

ans = ans || hasPathSum(root->left, sum-root->val);

if(root->right)

ans = ans || hasPathSum(root->right, sum-root->val);

return ans;

}

## Root to Leaf Paths With Sum

* Asked in:
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)
* [Yahoo](https://www.interviewbit.com/search/?q=Yahoo)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

Given a binary tree and a sum, find all root-to-leaf paths where each path’s sum equals the given sum.

For example:  
Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ / \

7 2 5 1

return

[

[5,4,11,2],

[5,8,4,5]

]

vector<int> path;

void pathSumUtil(TreeNode \*node, int sum, vector<vector<int>> &res, int sum\_so\_far)

{

if(!node) return;

path.push\_back(node->val);

sum\_so\_far +=node->val;

if(!node->left && !node->right && sum\_so\_far==sum)

res.push\_back(path);

if(node->left)

pathSumUtil(node->left, sum, res, sum\_so\_far);

if(node->right)

pathSumUtil(node->right, sum, res, sum\_so\_far);

path.pop\_back();

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

vector<vector<int> > Solution::pathSum(TreeNode\* root, int sum) {

vector<vector<int>> res;

int sum\_so\_far=0;

if(!root) return res;

pathSumUtil(root, sum, res, sum\_so\_far);

return res;

}

## Min Depth of Binary Tree

* Asked in:
* [Facebook](https://www.interviewbit.com/search/?q=Facebook)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

***NOTE :****The path has to end on a leaf node.*

**Example :**

1

/

2

min depth = 2.

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::minDepth(TreeNode\* root) {

if(!root) return 0;

if(!root->left && !root->right) return 1;

int ld=INT\_MAX, rd=INT\_MAX;

if(root->left) ld=minDepth(root->left);

if(root->right) rd=minDepth(root->right);

//cout<<(1+min(ld,rd))<<endl;

return 1+min(ld,rd);

}

## Sum Root to Leaf Numbers

* Asked in:
* [Google](https://www.interviewbit.com/search/?q=Google)
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)
* Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.
* An example is the root-to-leaf path 1->2->3 which represents the number 123.
* Find the total sum of all root-to-leaf numbers % 1003.
* **Example :**
* 1
* / \
* 2 3
* The root-to-leaf path 1->2 represents the number 12.  
  The root-to-leaf path 1->3 represents the number 13.
* Return the sum = (12 + 13) % 1003 = 25 % 1003 = 25.

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int sum(TreeNode\* root, int val){

if(root == NULL){

return 0;

}

val = (((val)%1003\*10)%1003 + (root->val)%1003)%1003;

if(root->left == NULL && root->right == NULL){

return val;

}

return (sum(root->left, val)%1003 + sum(root->right, val)%1003)%1003;

}

int Solution::sumNumbers(TreeNode\* A) {

// Do not write main() function.

// Do not read input, instead use the arguments to the function.

// Do not print the output, instead return values as specified

// Still have a doubt. Checkout www.interviewbit.com/pages/sample\_codes/ for more details

return sum(A, 0);

}

## Shortest Unique Prefix

Asked in:

* [Google](https://www.interviewbit.com/search/?q=Google)

Find shortest unique prefix to represent each word in the list.

**Example:**

Input: [zebra, dog, duck, dove]

Output: {z, dog, du, dov}

where we can see that

zebra = z

dog = dog

duck = du

dove = dov

#define ALPHABET\_SIZE 26

struct Trie{

int repeat;

Trie \*child[ALPHABET\_SIZE];

Trie(){

repeat=0;

for(int i=0; i<ALPHABET\_SIZE; i++)

child[i]=NULL;

}

};

void insert(Trie \*root, string s){

int n=s.size();

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

if(!root->child[s[i]-'a']) root->child[s[i]-'a']=new Trie();

root=root->child[s[i]-'a'];

root->repeat++;

}

}

int prefixFind(Trie \*node, string word){

for(int i=0; i<word.size(); i++){

node=node->child[word[i]-'a'];

if(node->repeat==1)

return i;

}

}

vector<string> Solution::prefix(vector<string> &input) {

vector<string> output;

Trie \*trie=new Trie();

for(int i=0; i<input.size(); i++){

insert(trie, input[i]);

}

//printTrie(trie);

for(vector<string>::iterator it=input.begin(); it!=input.end(); ++it){

string word=\*it;

int count=prefixFind(trie, word);

output.push\_back(word.substr(0,count+1)) ;

}

//print(output);

return output;

}

## Kth Smallest Element In Tree

Asked in:

* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

Given a binary search tree, write a function to find the kth smallest element in the tree.

**Example :**

Input :

2

/ \

1 3

and k = 2

Return : 2

As 2 is the second smallest element in the tree.

int countSubTree(TreeNode \*node){

if(!node) return 0;

return countSubTree(node->left)+countSubTree(node->right)+1;

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::kthsmallest(TreeNode\* root, int k) {

if(!root) return -1;

int lcount = countSubTree(root->left);

if(lcount+1 == k) return root->val;

if(k <= lcount)

return kthsmallest(root->left, k);

else

return kthsmallest(root->right, k-lcount-1);

}

Method 2:

**class** **Solution** {

**public:**

**int** find(TreeNode**\*** root, **int** **&**k) {

**if** (**!**root) **return** **-**1;

*// We do an inorder traversal here.*

**int** k1 **=** find(root**->**left, k);

**if** (k **==** 0) **return** k1; *// left subtree has k or more elements.*

k**--**;

**if** (k **==** 0) **return** root**->**val; *// root is the kth element.*

**return** find(root**->**right, k); *// answer lies in the right node.*

}

**int** kthsmallest(TreeNode**\*** root, **int** k) {

**return** find(root, k); *// Call another function to pass k by reference.*

}

};

## 2-Sum Binary Tree

* Asked in:
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

BookmarkSuggest Edit

Given a binary search tree T, where each node contains a positive integer, and an integer K, you have to find whether or not there exist two different nodes A and B such that A.value + B.value = K.

Return 1 to denote that two such nodes exist. Return 0, otherwise.

**Notes**

* Your solution should run in linear time and not take memory more than O(height of T).
* Assume all values in BST are distinct.

**Example :**

Input 1:

T : 10

/ \

9 20

K = 19

Return: 1

Input 2:

T: 10

/ \

9 20

K = 40

Return: 0

int find(TreeNode \*node, int sum, set<int> &s){

if(!node) return 0;

if(s.find(sum-node->val)!=s.end())

return 1;

s.insert(node->val);

return find(node->left, sum, s) || find(node->right, sum, s);

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::t2Sum(TreeNode\* root, int sum) {

if(!root) return 0;

set<int> nums;

return find(root, sum, nums);

}

## BST Iterator

* Asked in:
* [Apple](https://www.interviewbit.com/search/?q=Apple)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)
* [Facebook](https://www.interviewbit.com/search/?q=Facebook)

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

The first call to next() will return the smallest number in BST. Calling next() again will return the next smallest number in the BST, and so on.

***Note:****next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.  
Try to optimize the additional space complexity apart from the amortized time complexity.*

Push leftmost subtree to stack. If a node has right child, when popping the same, push its right leftmost subtree to stack.

stack<TreeNode\*> st;

void leftmostInorder(TreeNode \*node){

while(node){

st.push(node);

node=node->left;

}

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

BSTIterator::BSTIterator(TreeNode \*root) {

while(!st.empty())

st.pop();

leftmostInorder(root);

}

/\*\* @return whether we have a next smallest number \*/

bool BSTIterator::hasNext() {

if(!st.empty()) return true;

return false;

}

/\*\* @return the next smallest number \*/

int BSTIterator::next() {

if(st.empty()) return -1;

TreeNode \*top=st.top();

st.pop();

leftmostInorder(top->right);

return top->val;

}

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = BSTIterator(root);

\* while (i.hasNext()) cout << i.next();

\*/

## Recover Binary Search Tree

* Asked in:
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)

Two elements of a binary search tree (BST) are swapped by mistake.  
Tell us the 2 values swapping which the tree will be restored.

***Note:*** *A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?*

**Example :**

Input :

1

/ \

2 3

Output :

[1, 2]

Explanation : Swapping 1 and 2 will change the BST to be

2

/ \

1 3

which is a valid BST

// This function does inorder traversal to find out the two swapped nodes.

// It sets three pointers, first, middle and last.  If the swapped nodes are

// adjacent to each other, then first and middle contain the resultant nodes

// Else, first and last contain the resultant nodes

void correctBSTUtil( struct node\* root, struct node\*\* first,

                     struct node\*\* middle, struct node\*\* last,

                     struct node\*\* prev )

{

    if( root )

    {

        // Recur for the left subtree

        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating

        // the BST rule.

        if (\*prev && root->data < (\*prev)->data)

        {

            // If this is first violation, mark these two nodes as

            // 'first' and 'middle'

            if ( !\*first )

            {

                \*first = \*prev;

                \*middle = root;

            }

            // If this is second violation, mark this node as last

            else

                \*last = root;

        }

        // Mark this node as previous

        \*prev = root;

        // Recur for the right subtree

        correctBSTUtil( root->right, first, middle, last, prev );

    }

}

// A function to fix a given BST where two nodes are swapped.  This

// function uses correctBSTUtil() to find out two nodes and swaps the

// nodes to fix the BST

void correctBST( struct node\* root )

{

    // Initialize pointers needed for correctBSTUtil()

    struct node \*first, \*middle, \*last, \*prev;

    first = middle = last = prev = NULL;

    // Set the poiters to find out two nodes

    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree

    if( first && last )

        swap( &(first->data), &(last->data) );

    else if( first && middle ) // Adjacent nodes swapped

        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.

}

## Least Common Ancestor

* Asked in:
* [Facebook](https://www.interviewbit.com/search/?q=Facebook)
* [Adobe](https://www.interviewbit.com/search/?q=Adobe)
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)
* [Google](https://www.interviewbit.com/search/?q=Google)

Find the lowest common ancestor in an unordered binary tree given two values in the tree.

***Lowest common ancestor :****the lowest common ancestor (LCA) of two nodes v and w in a tree or directed acyclic graph (DAG) is the lowest (i.e. deepest) node that has both v and w as descendants.*

**Example :**

\_\_\_\_\_\_\_3\_\_\_\_\_\_

/ \

\_\_\_5\_\_ \_\_\_1\_\_

/ \ / \

6 \_2\_ 0 8

/ \

7 4

For the above tree, the LCA of nodes 5 and 1 is 3.

***LCA****= Lowest common ancestor*

Please note that LCA for nodes 5 and 4 is 5.

Sol1:

void printvec(vector<int> v){

for(int i=0; i<v.size(); i++)

cout<<v[i]<<" ";

cout<<endl;

}

bool findPath(TreeNode \*node, int n, vector<int> &res){

if(!node) return false;

res.push\_back(node->val);

if(node->val == n) return true;

if(findPath(node->left, n, res) || findPath(node->right, n, res))

return true;

res.pop\_back();

return false;

}

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

int Solution::lca(TreeNode\* root, int A, int B) {

vector<int> a, b;

findPath(root, A, a);

findPath(root, B, b);

//printvec(a);

//printvec(b);

if(a.size()>0 && b.size()>0){

for(int i=a.size()-1; i>=0; i--){

if(find(b.rbegin(), b.rend(),a[i])!=b.rend())

return a[i];

}

}

return -1;

}

Sol2:

class Solution {

public:

TreeNode \*LCA(TreeNode \*root, int val1, int val2) {

if (!root) return NULL;

if (root->val == val1 || root->val == val2) return root;

TreeNode \*L = LCA(root->left, val1, val2);

TreeNode \*R = LCA(root->right, val1, val2);

if (L && R) return root; // If val1, val2 are on both sides

return L ? L : R; // either one of val1, val2 is on one side OR val1, val2 is not in L&R subtrees

}

bool find(TreeNode \*root, int val1) {

if (!root) return false;

if (root->val == val1) return true;

return (find(root->left, val1) || find(root->right, val1));

}

int lca(TreeNode \*root, int val1, int val2) {

if (!find(root, val1) || !find(root, val2)) return -1;

TreeNode \*ans = LCA(root, val1, val2);

if (!ans) return -1;

return ans->val;

}

};

## Flatten Binary Tree to Linked List

* Asked in:
* [Adobe](https://www.interviewbit.com/search/?q=Adobe)
* [Amazon](https://www.interviewbit.com/search/?q=Amazon)
* [Microsoft](https://www.interviewbit.com/search/?q=Microsoft)

Given a binary tree, flatten it to a linked list in-place.

**Example :**  
Given

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

6

Note that the left child of all nodes should be NULL.

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

TreeNode\* Solution::flatten(TreeNode\* root) {

// Do not write main() function.

// Do not read input, instead use the arguments to the function.

// Do not print the output, instead return values as specified

// Still have a doubt. Checkout www.interviewbit.com/pages/sample\_codes/ for more details

if(!root) return NULL;

root->left=flatten(root->left);//flatten left subtree

root->right=flatten(root->right);//flatten right subtree

//attach left subtree to right child of root and

//attach right subtree to right child of left subtree

if(root->left){

TreeNode \*newpos=root->right;

root->right=root->left;

TreeNode \*tail=root->left;

root->left=NULL;

while(tail->right)

tail=tail->right;

tail->right=newpos;

}

return root;

}

Sol2:

lass Solution {

public:

void flatten(TreeNode \*root) {

if (!root) return;

TreeNode\* node = root;

while (node) {

// Attatches the right sub-tree to the rightmost leaf of the left sub-tree:

if (node->left) {

TreeNode \*rightMost = node->left;

while (rightMost->right) {

rightMost = rightMost->right;

}

rightMost->right = node->right;

// Makes the left sub-tree to the right sub-tree:

node->right = node->left;

node->left = NULL;

}

// Flatten the rest of the tree:

node = node->right;

}

}

};

## Order of People Heights

* Asked in:
* [Google](https://www.interviewbit.com/search/?q=Google)

You are given the following :

* A positive number N
* Heights : A list of heights of N persons standing in a queue
* Infronts : A list of numbers corresponding to each person (P) that gives the **number of persons**who are **taller** than P and standing in front of P

You need to return list of actual order of persons’s height

**Consider that heights will be unique**

**Example**

Input :

Heights: 5 3 2 6 1 4

InFronts: 0 1 2 0 3 2

Output :

actual order is: 5 3 2 1 6 4

So, you can see that for the person with height 5, there is no one taller than him who is in front of him, and hence Infronts has 0 for him.

For person with height 3, there is 1 person ( Height : 5 ) in front of him who is taller than him.

You can do similar inference for other people in the list.

Please read the previous hint if you haven’t done so already.

Here, we will explore how to efficiently answer the query of finding the ith empty space.

The query can be solved using **segment / interval tree**.  
The root contains the number of elements in [0, N].  
Left node contains the number of elements in [0, N/2]  
Right node contains the number of elements in [N/2 + 1, N]

Lets say we need to find the ith empty position.  
We look at the number of elements X in [0, N/2].

If

N / 2 - X >= i, the position lies in the left part of array and we move down to the left node.

N / 2 - X < i, we now look for i - (N / 2 - X) th position in the right part of the array and move to the right node in the tree.

This is a fairly standard use of the segment tree.

vector<int> Solution::order(vector<int> &height, vector<int> &infront) {

vector<int> res(height.size(),-1);

map<int, int> mp;

for(int i=0; i<height.size(); i++)

mp[height[i]]=infront[i]; //pairs of height and infront with increasting order of height

for(auto it: mp){

int empty\_pos=0, pos=0; //empty\_pos tells how my empty positions are

//pos tells at which position the person with given height to place

while(empty\_pos<=it.second){ //increase pos unless empty positions aren't reached what infront has

if(res[pos]==-1)

empty\_pos++;

pos++;

}

res[pos-1]=it.first; //subtract 1 from pos as position starts from 0

}

return res;

}

## Invert the Binary Tree

* Asked in:
* [Google](https://www.interviewbit.com/search/?q=Google)

Given a binary tree, invert the binary tree and return it.   
Look at the example for more details.

**Example :**   
Given binary tree

1

/ \

2 3

/ \ / \

4 5 6 7

invert and return

1

/ \

3 2

/ \ / \

7 6 5 4

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

TreeNode\* Solution::invertTree(TreeNode\* root) {

if(!root) return NULL;

root->left=invertTree(root->left);

root->right=invertTree(root->right);

TreeNode \*temp=root->left;

root->left=root->right;

root->right=temp;

return root;

}