1. **Left View of Binary Tree**

Given a Binary Tree, print Left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from Left side. The task is to complete the function **leftView()**, which accepts root of the tree as argument.

Left view of following tree is 1 2 4 8.

          1  
       /     \  
     2        3  
   /     \    /    \  
  4     5   6    7  
   \  
     8

**Example 1:**

**Input:**

  1

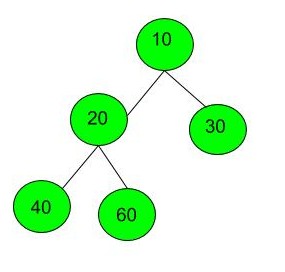
 /  \

3    2

**Output:** 1 3

**Example 2:**

**Input:**



**Output:** 10 20 40

**Your Task:**  
You just have to **complete**the function **leftView()**that prints the left view. The newline is automatically appended by the driver code.  
**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 <= Number of nodes <= 100  
1 <= Data of a node <= 1000

Code:

#include <bits/stdc++.h>

using namespace std;

// Tree Node

struct Node

{

int data;

Node\* left;

Node\* right;

};

void leftView(struct Node \*root);

// Utility function to create a new Tree Node

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// for(string i:ip)

// cout<<i<<" ";

// cout<<endl;

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

int main() {

int t;

scanf("%d ",&t);

while(t--)

{

string s;

getline(cin,s);

Node\* root = buildTree(s);

leftView(root);

cout << endl;

}

return 0;

}

// } Driver Code Ends

/\* A binary tree node

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

Node(int x){

data = x;

left = right = NULL;

}

};

\*/

int flag=0;

vector<int> v;

int height(Node\* root)

{

if(root==NULL)

return 0;

int hl=height(root->left);

int hr=height(root->right);

return max(hl,hr)+1;

}

void perlevel(Node\* root,int h)

{

if(root==NULL)

return;

if(h==1)

{

if(!flag)

{

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

flag=1;

}

return;

}

perlevel(root->left,h-1);

perlevel(root->right,h-1);

}

void level(Node\* root)

{

int h=height(root);

for(int i=1;i<=h;i++)

{

flag=0;

perlevel(root,i);

}

return;

}

// A wrapper over leftViewUtil()

void leftView(Node \*root)

{

level(root);

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

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

v.clear();

}

2. **Check for BST**

Given a binary tree. Check whether it is a BST or not.

**Example 1:**

**Input:**

    2

/    \

1      3

**Output:** 1

**Example 2:**

**Input:**

2

  \

  7

  \

  6

  \

  5

  \

  9

  \

  2

  \

  6

**Output:** 0

**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. The printing is done by the driver's code.

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

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

Code:

#include <bits/stdc++.h>

using namespace std;

#define MAX\_HEIGHT 100000

// Tree Node

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

bool isBST(struct Node\* node);

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

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

Node\* root = new Node(stoi(ip[0]));

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

void inorder(Node \*root, vector<int> &v)

{

if(root==NULL)

return;

inorder(root->left, v);

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

inorder(root->right, v);

}

int main() {

int t;

string tc;

getline(cin, tc);

t=stoi(tc);

while(t--)

{

string s;

getline(cin, s);

Node\* root = buildTree(s);

cout << isBST(root) << endl;

}

return 0;

}

// } Driver Code Ends

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

and a pointer to right child

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

\*/

bool checkNode(Node\* node,int min,int max)

{

if(node==NULL)

return 1;

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

return 0;

else

{

checkNode(node->left,min,node->data-1)&&

checkNode(node->right,node->data+1,max);

}

}

bool isBST(Node\* root) {

return checkNode(root,INT\_MIN,INT\_MAX);

}

// { Driver Code Starts

// } Driver Code Ends

3. **Bottom View of Binary Tree**

Given a binary tree, print the bottom view from left to right.  
A node is included in bottom view if it can be seen when we look at the tree from bottom.

                      20  
                    /    \  
                  8       22  
                /   \        \  
              5      3       25  
                    /   \        
                  10    14

For the above tree, the bottom view is 5 10 3 14 25.  
If there are **multiple**bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottommost nodes at horizontal distance 0, we need to print 4.

                      20  
                    /    \  
                  8       22  
                /   \     /   \  
              5      3 4     25  
                     /    \        
                 10       14

For the above tree the output should be 5 10 4 14 25.

**Example 1:**

**Input:**

1

  / \

  3 2

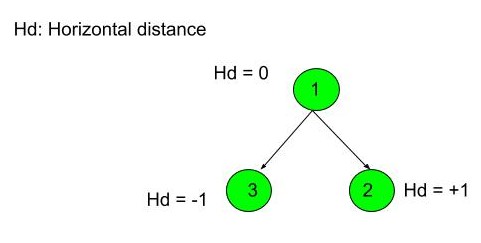
**Output:** 3 1 2

**Explanation:**

First case represents a tree with 3 nodes

and 2 edges where root is 1, left child of

1 is 3 and right child of 1 is 2.



Thus nodes of the binary tree will be

printed as such 3 1 2.

**Example 2:**

**Input:**

10

  / \

  20 30

  / \

  40 60

**Output:** 40 20 60 30

**Your Task:**  
This is a functional problem, you **don't**need to care about input, just complete the function **bottomView**() which takes the root node of the tree as input and returns an array containing the bottom view of the given tree.

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

**Constraints:**  
1 <= Number of nodes <= 105  
1 <= Data of a node <= 105

Code:

#include <bits/stdc++.h>

using namespace std;

#define MAX\_HEIGHT 100000

// Tree Node

struct Node

{

int data;

Node\* left;

Node\* right;

};

// Utility function to create a new Tree Node

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

vector <int> bottomView(Node \*root);

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

int main() {

int t;

string tc;

getline(cin, tc);

t=stoi(tc);

while(t--)

{

string s ,ch;

getline(cin, s);

Node\* root = buildTree(s);

vector <int> res = bottomView(root);

for (int i : res) cout << i << " ";

cout << endl;

}

return 0;

}

// } Driver Code Ends

/\* Tree node class

struct Node

{

int data; //data of the node

Node \*left, \*right; //left and right references

// Constructor of tree node

Node(int key)

{

data = key;

left = right = NULL;

}

}; \*/

// Method that returns the bottom view.

vector <int> bottomView(Node \*root)

{

vector<int> ans;

if(root==NULL)

return ans;

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

map<int,int> mp;

int d=0;

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

mp[d]=root->data;

while(!q.empty())

{

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

q.pop();

d=temp.second;

mp[d]=temp.first->data;

if(temp.first->left!=NULL)

{

q.push(make\_pair(temp.first->left,d-1));

}

if(temp.first->right!=NULL)

{

q.push(make\_pair(temp.first->right,d+1));

}

}

map<int,int> ::iterator it=mp.begin();

while(it!=mp.end())

{

//cout<<it->first<<" - "<<it->second<<endl;

if(it->second!=0)

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

it++;

}

return ans;

}

4. **Vertical Traversal of Binary Tree**

Given a Binary Tree, find the vertical traversal of it starting from the leftmost level to the rightmost level.  
If there are multiple nodes passing through a vertical line, then they should be printed as they appear in **level order** traversal of the tree.

**Example 1:**

**Input:**

2

     \

      3

     /

    4

**Output:** 2 4 3

**Example 2:**

**Input:**

1

   /    \

  2      3

/   \      \

4     5     6

**Output:** 4 2 1 5 3 6

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **verticalOrder()**which takes the root node as input and returns an array containing the vertical order traversal of the tree from the leftmost to the rightmost level. If 2 nodes lie in the same vertical level, they should be printed in the order they appear in the level order traversal of the tree.

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

**Constraints:**  
1 <= Number of nodes <= 5000

Code:

// Tree Node

struct Node

{

int data;

Node\* left;

Node\* right;

};

// Utility function to create a new Tree Node

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

vector <int> verticalOrder(Node \*root);

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

// Function for Inorder Traversal

void printInorder(Node\* root)

{

if(!root)

return;

printInorder(root->left);

cout<<root->data<<" ";

printInorder(root->right);

}

int main() {

int t;

string tc;

getline(cin,tc);

t=stoi(tc);

while(t--)

{

string s;

getline(cin,s);

// string c;

// getline(cin,c);

Node\* root = buildTree(s);

vector <int> res = verticalOrder(root);

for (int i : res) cout << i << " ";

cout << endl;

}

return 0;

}

// } Driver Code Ends

/\* 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;

Node(int x){

data = x;

left = right = NULL;

}

};

\*/

bool comparator(pair<Node\*,int> p1,pair<Node\*,int>p2)

{

if(p1.second<p2.second)

return true;

return false;

}

// root: root node of the tree

vector<int> verticalOrder(Node \*root)

{

vector<int> res;

if(root==NULL)

return res;

multimap<int,int> m;

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

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

while(!q.empty())

{

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

q.pop();

m.insert(pair<int,int>(curr.second,curr.first->data));

if(curr.first->left)

q.push(make\_pair(curr.first->left,curr.second-1));

if(curr.first->right)

q.push(make\_pair(curr.first->right,curr.second+1));

}

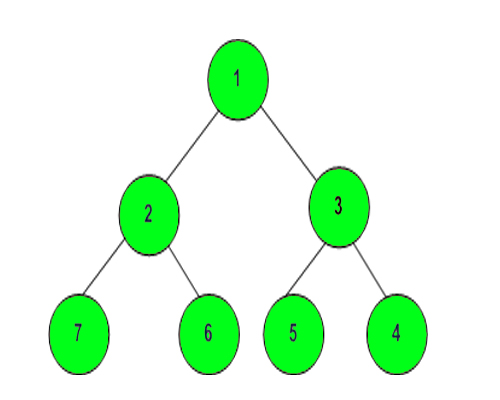
for(auto i:m)

res.push\_back(i.second);

return res;

}

5. **Level order traversal in spiral form**

Complete the function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.  
  
  
****

**Example 1:**

**Input:**

      1

   /   \

  3     2

**Output:**1 3 2

**Example 2:**

**Input:**

           10

        /     \

       20     30

     /    \

   40     60

**Output:** 10 20 30 60 40

**Your Task:**  
The task is to complete the function **printSpiral**() which prints the elements in spiral form of level order traversal. The newline is automatically appended by the driver code.  
**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(N).

**Constraints:**  
0 <= Number of nodes <= 105  
1 <= Data of a node <= 105

Code:

#include <bits/stdc++.h>

using namespace std;

// Tree Node

struct Node

{

int data;

Node\* left;

Node\* right;

};

// Utility function to create a new Tree Node

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

void printSpiral(Node \*root);

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

int main() {

int t;

string tc;

getline(cin,tc);

t=stoi(tc);

while(t--)

{

string s;

getline(cin,s);

Node\* root = buildTree(s);

printSpiral(root);

cout << endl;

}

return 0;

}

// } Driver Code Ends

/\* 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;

Node(int x){

data = x;

left = right = NULL;

}

}; \*/

int flag;

int height(Node\* root)

{

if(root==NULL)

return 0;

int hl=height(root->left);

int hr=height(root->right);

return max(hr,hl)+1;

}

void perlevel(Node\* root,int h,int flag)

{

if(root==NULL)

return;

if(h==1)

{

cout<<root->data<<" ";

return;

}

if(!flag)

{

perlevel(root->left,h-1,flag);

perlevel(root->right,h-1,flag);

// flag=!flag;

}

if(flag)

{

perlevel(root->right,h-1,flag);

perlevel(root->left,h-1,flag);

// flag=!flag;

}

return;

}

void printSpiral(Node \*root)

{

if(root==NULL)

return;

int h=height(root);

flag=1;

for(int i=1;i<=h;i++)

{

perlevel(root,i,flag);

flag=!flag;

}

return;

//Your code here

}

6. **Connect Nodes at Same Level**Given a binary tree, connect the nodes that are at same level. You'll be given an addition **nextRight**pointer for the same.

**Initially**, all the **nextRight**pointers point to **garbage**values. **Your function** should set these pointers to point next right for each node.  
       10                       10 ------> NULL  
      / \                       /      \  
     3   5       =>     3 ------> 5 --------> NULL  
    / \     \               /  \           \  
   4   1   2          4 --> 1 -----> 2 -------> NULL

**Example 1:**

**Input:**

3

  / \

  1 2

**Output:**

3 1 2

1 3 2

**Explanation:**The connected tree is

        3 ------> NULL

     /    \

   1-----> 2 ------ NULL

**Example 2:**

**Input:**

10

  / \

  20 30

  / \

40 60

**Output:**

10 20 30 40 60

40 20 60 10 30

**Explanation:**The connected tree is

         10 ----------> NULL

      /     \

     20 ------> 30 -------> NULL

  /    \

 40 ----> 60 ----------> NULL

**Your Task:**  
You don't have to take input. Complete the function **connect()**that takes **root**as parameter and connects the nodes at same level. The printing is done by the driver code.

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

**Constraints:**  
1 <= Number of nodes <= 100  
1 <= Data of a node <= 1000

Code:

#include <bits/stdc++.h>

using namespace std;

// Tree Node

struct Node

{

int data;

Node\* left;

Node\* right;

Node\* nextRight;

};

// Utility function to create a new Tree Node

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

temp->nextRight = NULL;

return temp;

}

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

void connect(struct Node \*p);

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

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

void printSpecial(Node \*root)

{

if (root == NULL)

return;

Node\* next\_root=NULL;

while (root != NULL)

{

cout<< root->data<<" ";

if( root->left && (!next\_root) )

next\_root = root->left;

else if( root->right && (!next\_root) )

next\_root = root->right;

root = root->nextRight;

}

printSpecial(next\_root);

}

void inorder(Node \*root)

{

if (root == NULL)

return;

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

/\* Driver program to test size function\*/

int main()

{

int t;

scanf("%d\n", &t);

while (t--)

{

string s;

getline(cin, s);

Node\* root = buildTree(s);

connect(root);

printSpecial(root);

cout<<endl;

inorder(root);

cout<<endl;

}

return 0;

}

// } Driver Code Ends

/\* struct Node

{

int data;

Node \*left, \*right;

Node \*nextRight; // This has garbage value in input trees

}; \*/

// Should set the nextRight for all nodes

void connect(Node \*root)

{

queue<Node\*> q;

q.push(root);

while(!q.empty())

{

int size=q.size();

while(size--)

{

Node\* p=q.front();

q.pop();

if(size)

p->nextRight=q.front();

if(p->left!=NULL)

q.push(p->left);

if(p->right!=NULL)

q.push(p->right);

}

}

}

7. **Lowest Common Ancestor 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

Code:

#include <bits/stdc++.h>

using namespace std;

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

Node\* root = new Node(stoi(ip[0]));

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

Node\* LCA(Node \* root , int l , int h);

int main()

{

int t;

scanf("%d ",&t);

while(t--)

{

string s;

int l , h;

getline(cin,s);

scanf("%d ",&l);

scanf("%d ",&h);

Node\* root = buildTree(s);

cout<<LCA(root , l , h)->data<<endl;

}

return 1;

}// } Driver Code Ends

/\*The structure of a BST Node is as follows:

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

\*/

// Returns the LCA of the nodes with values n1 and n2

// in the BST rooted at 'root'

Node\* LCA(Node \*root, int n1, int n2)

{

while(root!=NULL)

{

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

root=root->left;

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

root=root->right;

else

break;

}

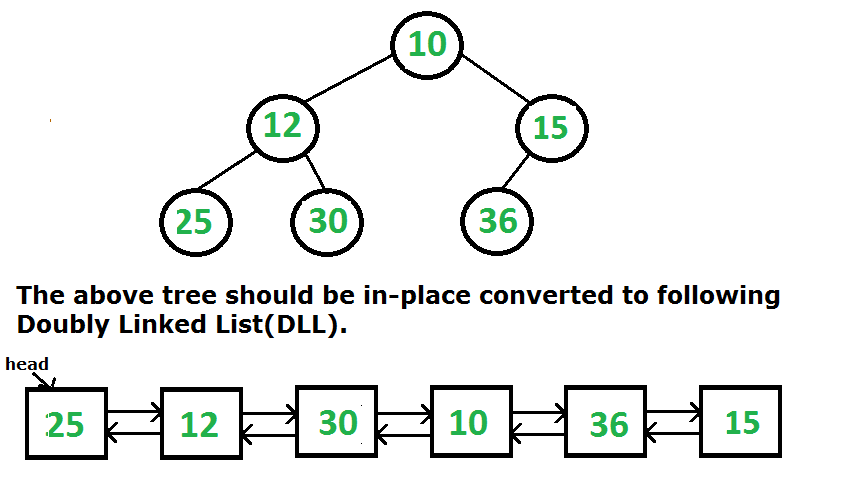
return root;

//Your code here

}

8. **Binary Tree to DLL**

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (leftmost node in BT) must be the head node of the DLL.



**Example 1:**

**Input:**

      1

   /  \

  3    2

**Output:**

3 1 2

2 1 3

**Explanation:** DLL would be 3<=>1<=>2

**Example 2:**

**Input:**

       10

     /   \

20   30

  /   \

 40   60

**Output:**

40 20 60 10 30

30 10 60 20 40

**Explanation:**  DLL would be

40<=>20<=>60<=>10<=>30.

**Your Task:**  
You don't have to take input. Complete the function **bToDLL()**that takes **root**node of the tree as a parameter and returns the head of DLL . The driver code prints the DLL both ways.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(H).  
**Note:**H is the height of the tree and this space is used implicitly for recursion stack.

**Constraints:**  
1 <= Number of nodes <= 1000  
1 <= Data of a node <= 1000

Code:

#include <bits/stdc++.h>

using namespace std;

// Tree Node

struct Node

{

int data;

Node\* left;

Node\* right;

};

// Utility function to create a new Tree Node

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

Node\* bToDLL(Node \*root);

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

void printList(Node \*node)

{

Node \*prev = NULL;

while (node!=NULL)

{

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

prev = node;

node = node->right;

}

cout << endl;

while (prev!=NULL)

{

cout << prev->data << " ";

prev = prev->left;

}

cout << endl;

}

void inorder(Node \*root)

{

if (root != NULL)

{

inorder(root->left);

cout << root->data;

inorder(root->right);

}

}

/\* Driver program to test size function\*/

int main()

{

int t;

cin >> t;

getchar();

while (t--)

{

string inp;

getline(cin, inp);

Node \*root = buildTree(inp);

Node \*head = bToDLL(root);

printList(head);

}

return 0;

}

// } Driver Code Ends

/\* Structure for tree and linked list

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

Node(int x){

data = x;

left = right = NULL;

}

};

\*/

// This function should return head to the DLL

Node \*p;

Node \*h;

void fun(Node \*root)

{

if(root==NULL)

return ;

fun(root->left);

if(p==NULL)

{

h=root;

}

else

{

p->right=root;

root->left=p;

}

p=root;

fun(root->right);

}

Node \* bToDLL(Node \*root)

{

// your code here

h=NULL;

p=NULL;

fun(root);

return h;

}

9. **Determine if Two Trees are Identical**

Given two binary trees, the task is to find if both of them are identical or not.

**Example 1:**

**Input:**

1 1

  / \ / \

  2 3 2 3

**Output:** Yes

**Explanation:** There are two trees both

having 3 nodes and 2 edges, both trees

are identical having the root as 1,

left child of 1 is 2 and right child

of 1 is 3.

**Example 2:**

**Input:**

1 1

  / \ / \

 2 3 3 2

**Output:** No

**Explanation:** There are two trees both

having 3 nodes and 2 edges, but both

trees are not identical.

**Your task:**  
Since this is a functional problem you don't have to worry about input, you just have to complete the function **isIdentical()** that takes two roots as parameters and returns true or false. The printing is done by the driver code.  
  
**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 <= Number of nodes <= 1000  
1 <=Data of a node <= 1000

Code:

#include <bits/stdc++.h>

using namespace std;

struct Node

{

int data;

struct Node \*left;

struct Node \*right;

Node(int x){

data = x;

left = NULL;

right = NULL;

}

};

bool isIdentical(Node\* a, Node\* b);

// Function to Build Tree

Node \*buildTree(string str) {

// Corner Case

if (str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for (string str; iss >> str;)

ip.push\_back(str);

// Create the root of the tree

Node \*root = new Node(stoi(ip[0]));

// Push the root to the queue

queue<Node \*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while (!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node \*currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if (currVal != "N") {

// Create the left child for the current node

currNode->left = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if (i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if (currVal != "N") {

// Create the right child for the current node

currNode->right = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

int main() {

int tc;

scanf("%d ", &tc);

while (tc--) {

string str, str1;

getline(cin, str);

Node \*rootA = buildTree(str);

getline(cin, str1);

Node \*rootB = buildTree(str1);

if (isIdentical(rootA, rootB)) {

cout << "Yes\n";

} else {

cout << "No\n";

}

}

return 0;

}// } Driver Code Ends

/\* A binary tree node

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

Node(int x){

data = x;

left = right = NULL;

}

};

\*/

/\* Should return true if trees with roots as r1 and

r2 are identical \*/

bool isIdentical(Node \*r1, Node \*r2)

{

//Your Code here

if(r1==NULL&&r2==NULL)

{

return true;

}

if(r1==NULL&&r2!=NULL)

{

return false;

}

if(r1!=NULL&&r2==NULL)

{

return false;

}

if (r1->data!=r2->data)

{

return false;

}

return isIdentical(r1->right,r2->right)&&isIdentical(r1->left,r2->left);

}

10. **Symmetric Tree**

Given a Binary Tree. Check whether it is Symmetric or not, i.e. whether the binary tree is a **Mirror image of itself** or not.

**Example 1:**

**Input:**

5

/ \

1 1

/ \

2 2

**Output:** True

**Explanation:** Tree is mirror image of

itself i.e. tree is symmetric

**Example 2:**

**Input:**

5

/ \

10 10

/ \ \

20 20 30

**Output:** False

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **isMirror()** which takes the root of the Binary Tree as its input and returns True if the given Binary Tree is a same as the Mirror image of itself. Else, it returns False.

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

**Constraints:**  
1<=Number of nodes<=100

Code;

#include<bits/stdc++.h>

using namespace std;

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

Node\* root = new Node(stoi(ip[0]));

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

bool isSymmetric(struct Node\* root);

int main()

{

int t;

scanf("%d ",&t);

while(t--)

{

string s;

getline(cin,s);

Node\* root = buildTree(s);

if(isSymmetric(root))

cout<<"True"<<endl;

else

cout<<"False"<<endl;

}

return 0;

}

// } Driver Code Ends

/\*

Structure of the node of the tree is as

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

\*/

vector<int> v;

void inorder(Node\* root)

{

if(root==NULL)

return;

inorder(root->left);

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

inorder(root->right);

return;

}

// return true/false denoting whether the tree is Symmetric or not

bool isSame(struct Node\* r1,struct Node\* r2){

if(!r1 && !r2)

return true;

if(!r1 || !r2)

return false;

return r1->data==r2->data && isSame(r1->left,r2->right) && isSame(r1->right,r2->left);

}

bool isSymmetric(struct Node\* root)

{

return isSame(root,root);

}

11. **Height of Binary Tree**

Given a binary tree, find its height.

​​**Example 1:**

**Input:**

1

   /  \

  2   3

**Output:** 2

**Example 2:**

**Input:**

2

  \

  1

  /

3

**Output:** 3

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **height()**that takes root Node of the Tree as input and returns the Height of the Tree. If the Tree is empty, return 0.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= Number of nodes <= 105  
1 <= Data of a node <= 105

Code:

#include <bits/stdc++.h>

using namespace std;

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

Node\* root = new Node(stoi(ip[0]));

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

int height(struct Node\* node);

int main()

{

int t;

scanf("%d ",&t);

while(t--)

{

string s;

getline(cin,s);

Node\* root = buildTree(s);

cout<<height(root)<<endl;

}

return 0;

}

// } Driver Code Ends

/\* Tree node structure used in the program

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};\*/

// return the Height of the given Binary Tree

int height(Node\* root)

{

if(root==NULL)

return 0;

int lh=height(root->left);

int rh=height(root->right);

return (max(rh,lh)+1);

// Your code here

}

12. **Maximum Path Sum between 2 Leaf Nodes**

Given a binary tree in which each node element contains a number. Find the maximum possible sum from one leaf node to another.

**Example 1:**

**Input :**

3

/ \

4 5

/ \

-10 4

**Output:** 16

**Explanation :**

Maximum Sum lies between leaf node 4 and 5.

4 + 4 + 3 + 5 = 16.

**Example 2:**

**Input :**

-15

/ \

5 6

/ \ / \

-8 1 3 9

/ \ \

2 -3 0

/ \

4 -1

/

10

**Output :** 27

**Explanation:**

The maximum possible sum from one leaf node

to another is (3 + 6 + 9 + 0 + -1 + 10 = 27)

**Your Task:**  
You dont need to read input or print anything. Complete the function **maxPathSum()**which takes root node as input parameter and returns the maximum sum between 2 leaf nodes.

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

**Constraints:**  
1 ≤ N ≤ 10^4

Code:

#include <bits/stdc++.h>

using namespace std;

// Tree Node

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

// Function to Build Tree

Node \*buildTree(string str) {

// Corner Case

if (str.length() == 0 || str[0] == 'N') return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for (string str; iss >> str;) ip.push\_back(str);

// Create the root of the tree

Node \*root = new Node(stoi(ip[0]));

// Push the root to the queue

queue<Node \*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while (!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node \*currNode = queue.front();

queue.pop();

// Get the current Node's value from the string

string currVal = ip[i];

// If the left child is not null

if (currVal != "N") {

// Create the left child for the current Node

currNode->left = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if (i >= ip.size()) break;

currVal = ip[i];

// If the right child is not null

if (currVal != "N") {

// Create the right child for the current Node

currNode->right = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

int maxPathSum(Node \*);

int main() {

int tc;

scanf("%d ", &tc);

while (tc--) {

string treeString;

getline(cin, treeString);

Node \*root = buildTree(treeString);

cout << maxPathSum(root) << "\n";

}

return 0;

}// } Driver Code Ends

/\*

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

Node(int x){

data = x;

left = right = NULL;

}

};

\*/

#define ll long long

ll ans;

ll find(Node\*root)

{

if(root==NULL)

return INT\_MIN;

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

return root->data;

ll x=find(root->left);

ll y=find(root->right);

ans=max(ans,x+y+root->data);

return max(x,y)+root->data;

}

int maxPathSum(Node\* root)

{

ans=INT\_MIN;

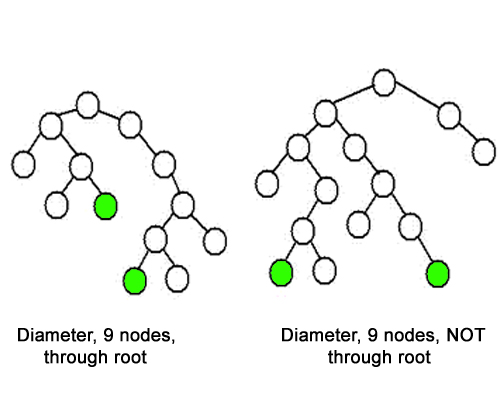
find(root);

return ans;

}

13. **Diameter of Binary Tree**

Given a Binary Tree,**find diameter of it**.  
The diameter of a tree is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).

[](http://geeksforgeeks.org/wp-content/uploads/tree_diameter.GIF)

**Example 1:**

**Input:**

     1

    /  \

   2    3

**Output:** 3

**Example 2:**

**Input:**

         10

       /   \

     20    30

   /   \

   40   60

**Output:** 4

**Your Task:**  
You need to **complete**the **function diameter()**that takes **node**as **parameter**and **returns**the **diameter**.  
  
**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 <= Number of nodes <= 10000  
1 <= Data of a node <= 1000

Code:

#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;

};

Node\* newNode(int val) {

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

Node\* buildTree(string str) {

// Corner Case

if (str.length() == 0 || str[0] == 'N') return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for (string str; iss >> str;) ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while (!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if (currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if (i >= ip.size()) break;

currVal = ip[i];

// If the right child is not null

if (currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

/\* Function to get diameter of a binary tree \*/

int diameter(struct Node\* tree);

/\* Driver program to test size function\*/

int main() {

int t;

scanf("%d\n", &t);

while (t--) {

string s;

getline(cin, s);

Node\* root = buildTree(s);

cout << diameter(root) << endl;

}

return 0;

}

// } Driver Code Ends

/\* Tree node structure used in the program

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

Node(int x){

data = x;

left = right = NULL;

}

}; \*/

class pair1{

public:

int height;

int diameter;

};

pair1 fastdiameter(Node\* root)

{

pair1 p;

if(root==NULL)

{

p.height=0;

p.diameter=0;

return p;

}

pair1 left=fastdiameter(root->left);

pair1 right=fastdiameter(root->right);

p.height=max(left.height,right.height)+1;

p.diameter=max(left.height+right.height+1,max(left.diameter,right.diameter));

return p;

}

int diameter(Node\* root)

{

pair1 p=fastdiameter(root);

return p.diameter;

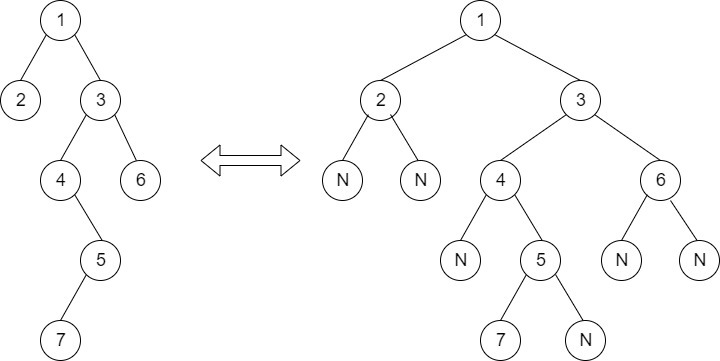
}

14. **Count Leaves in Binary Tree**

Given a Binary Tree of size **N** , You have to count leaves in it. For example, there are two leaves in following tree

        1  
     /      \  
   10      39  
  /  
5

**Input:**  
First line of input contains the number of test cases **T**. For each test case, there will be only a **single** line of input which is a **string** representing the tree as described below:

1. The values in the string are in the order of **level order** traversal of the tree where, numbers denote node values, and a character “N” denotes **NULL** child.
2. For example:  
     
   For the above tree, the string will be: 1 2 3 N N 4 6 N 5 N N 7 N

**Output:**  
For each test case print the count of leaves.  
**Your Task:**  
You don't have to take input. Complete the function **countLeaves()**that takes **root**node of given treeas parameter and **returns**the count of leaves in tree . The **printing**is done by the **driver**code.  
**Constraints:**  
1<= T <= 30  
1<= N <= 104  
**Example:  
Input:**  
2  
3 4 2   
4 8 10 7 N 5 1 3   
**Output:**  
2  
3

Code:

#include <bits/stdc++.h>

using namespace std;

struct Node

{

int data;

struct Node \*left;

struct Node \*right;

};

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

int countLeaves(struct Node\* root);

int main()

{

int t;

scanf("%d ",&t);

while(t--)

{

string s;

getline(cin,s);

Node\* root = buildTree(s);

cout<< countLeaves(root)<<endl;

}

return 0;

}

// } Driver Code Ends

//User function Template for C++

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

and a pointer to right child

struct Node

{

int data;

Node\* left;

Node\* right;

}; \*/

/\* Should return count of leaves. For example, return

value should be 2 for following tree.

10

/ \

20 30 \*/

int countLeaves(Node\* root)

{

if(root==NULL)

return 0;

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

return 1;

return (countLeaves(root->left)+countLeaves(root->right));

// Your code here

}

15. **Check for Balanced Tree**

Given a binary tree, find if it is height balanced or not.   
A tree is height balanced if difference between heights of left and right subtrees is **not more than one** for all nodes of tree.

**A height balanced tree**  
        1  
     /     \  
   10      39  
  /  
5

**An unbalanced tree**  
        1  
     /      
   10     
  /  
5

**Example 1:**

**Input:**

      1

   /

   2

   \

    3

**Output:** 0

**Explanation:** The max difference in height

of left subtree and right subtree is 2,

which is greater than 1. Hence unbalanced

**Example 2:**

**Input:**

       10

    /   \

   20   30

  /   \

40   60

**Output:** 1

**Explanation:** The max difference in height

of left subtree and right subtree is 1.

Hence balanced.

**Your Task:**  
You don't need to take input. Just complete the function**isBalanced()**that takes root **node**as parameter and returns **true,**if the tree is balanced else returns **false**.

**Constraints:**  
1 <= Number of nodes <= 105  
0 <= Data of a node <= 106

**Expected time complexity:**O(N)  
**Expected auxiliary space:**O(h) , where h = height of tree

Code:

#include <bits/stdc++.h>

using namespace std;

struct Node

{

int data;

struct Node \*left;

struct Node \*right;

};

// Utility function to create a new Tree Node

Node\* newNode(int val)

{

Node\* temp = new Node;

temp->data = val;

temp->left = NULL;

temp->right = NULL;

return temp;

}

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after spliting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

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

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = newNode(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

bool isBalanced(Node \*root);

int main()

{

int t;

scanf("%d ",&t);

while(t--)

{

string s;

getline(cin,s);

Node\* root = buildTree(s);

cout << isBalanced(root) << endl;

}

return 1;

}// } Driver Code Ends

/\* A binary tree node structure

struct Node

{

int data;

struct Node\* left;

struct Node\* right;

Node(int x){

data = x;

left = right = NULL;

}

};

\*/

// This function should return tree if passed tree

// is balanced, else false.

int height(Node\* node)

{

if (!node)

return 0;

else

{

int l=height(node->left);

int r=height(node->right);

if (l>r) return(l+1);

else return(r+1);

}

}

bool isBalanced(Node \*root)

{ if(!root) return true;

return (isBalanced(root->right)&&isBalanced(root->left)&&abs(height(root->left)-height(root->right))<=1);

}