# Lesson

#include <iostream>

#include <stdlib.h>

#include <sstream>

#include <stack>

#include <string>

#include <list>

#include <vector>

#include <queue>

using namespace std;

template<class *T*>

class BTree{

    public:

        class *Node*; // forward declaration

    private:

*Node*\* root;

    public:

    BTree(){

        root = nullptr;

    }

    ~BTree(){}

    void insert(*T*\* *arr*, int *size*){

        if(*size* == 0) return;

        root = **new** *Node*(*arr*[0]);

        int i=1;

        queue<*Node*\*> q;

        q.push(root);

        while(!q.empty()){

*Node*\* curr = q.front();

            q.pop();

            if(i<*size*){

                curr->pLeft = **new** *Node*(*arr*[i]);

                q.push(curr->pLeft);

                i++;

            }

            if(i<*size*){

                curr->pRight = **new** *Node*(*arr*[i]);

                q.push(curr->pRight);

                i++;

            }

        }

    }

    void printIn(){

        LNR(root);

    }

    void LNR(*Node*\* *root*){

        if(*root* == nullptr) return;

        LNR(*root*->pLeft);

        cout<<*root*->data<<" ";

        LNR(*root*->pRight);

    }

    void mirror(){

        mirror(root);

    }

    void mirror(*Node*\* *root*){

        if(*root*->pLeft != nullptr && *root*->pRight != nullptr){

*T* temp = *root*->pLeft->data;

*root*->pLeft->data = *root*->pRight->data;

*root*->pRight->data = temp;

            mirror(*root*->pLeft);

            mirror(*root*->pRight);

        }

        return;

    }

    void printR2L(){

        printR2L(root);

    }

    void printR2L(*Node*\* *root*){

        if(*root* == NULL) return;

        queue<*Node*\*> q;

        q.push(*root*);

        while(q.size() != 0){

*Node*\* temp = q.front();

            q.pop();

            cout<<temp->data<<" ";

            if(temp->pRight != NULL) q.push(temp->pRight);

            if(temp->pLeft != NULL) q.push(temp->pLeft);

        }

    }

    class *Node*{

        private:

*T* data;

*Node*\* pLeft;

*Node*\* pRight;

            friend class BTree;

        public:

        Node(*T* *data*, *Node*\* *pLeft*=nullptr, *Node*\* *pRight*=nullptr){

*this*->data = *data*;

*this*->pLeft = *pLeft*;

*this*->pRight = *pRight*;

        }

    };

};

int main()

{

    //system("CLS");

    //printR2L -> stack

    BTree<int> bt;

    int arr[5] = {10,1,3,6,2};

    bt.insert(arr, 5);

    bt.printR2L();

    return 0;

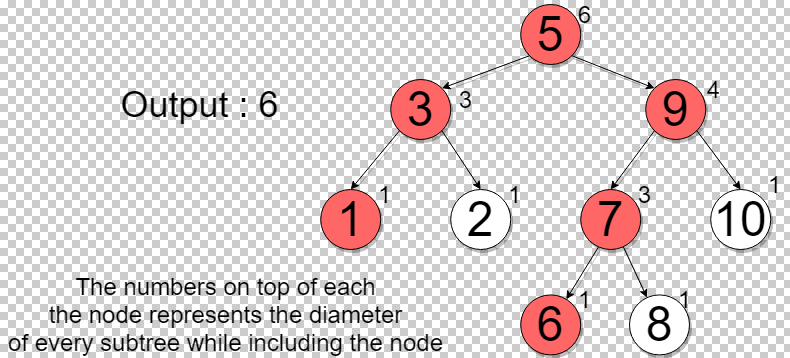
}

# Lab Binary Tree

## Q1

Given a Binary tree, the task is to calculate the diameter of the tree. The diameter of the tree is the maximum distance between two node on the tree.

For example:



Your task is to complete the **diameter** function. You can write one ore more function to help you complete this task.

#include <iostream>  
#include <string>  
#include <sstream>  
**using** **namespace** std;  
#define SEPARATOR "#<ab@17943918#@>#"  
**template**<**class** K, **class** V>  
**class** BinaryTree  
{  
**public**:  
    **class** Node;  
**private**:  
    Node\* root;  
**public**:  
    BinaryTree() : root(**nullptr**) {}  
    ~BinaryTree()  
    {  
        // You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.  
    }  
    **class** Node  
    {  
    **private**:  
        K key;  
        V value;  
        Node\* pLeft, \* pRight;  
        **friend** **class** BinaryTree<K, V>;  
    **public**:  
        Node(K key, V value) : key(key), value(value), pLeft(**NULL**), pRight(**NULL**) {}  
        ~Node() {}  
    };  
    **void** addNode(string posFromRoot, K key, V value)  
    {  
        **if** (posFromRoot == "")  
        {  
            **this**->root = **new** Node(key, value);  
            **return**;  
        }  
        Node\* walker = **this**->root;  
        **int** l = (**int**)posFromRoot.length();  
        **for** (**int** i = 0; i < l - 1; i++)  
        {  
            **if** (!walker)  
                **return**;  
            **if** (posFromRoot[i] == 'L')  
                walker = walker->pLeft;  
            **if** (posFromRoot[i] == 'R')  
                walker = walker->pRight;  
        }  
        **if** (posFromRoot[l - 1] == 'L')  
            walker->pLeft = **new** Node(key, value);  
        **if** (posFromRoot[l - 1] == 'R')  
            walker->pRight = **new** Node(key, value);  
    }

    //Helping functions  
    **int** getDiameter(){  
        **//TODO**  
    }  
};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("",1, 4);  binaryTree.addNode("L",2, 6);  binaryTree.addNode("R",3, 9);  binaryTree.addNode("LL",4, 10);  cout << binaryTree.getDiameter(); | 4 |

//Helping functions  
    **int** getDiameter(){  
        **//TODO**  
    }

//Helping functions

    /\*

    The diameter of the binary tree

    is the longest path distance

    between two end nodes(leaf nodes) in the tree.\*/

    int max(int *v1*, int *v2*){

        if(*v1* > *v2*) return *v1*; else return *v2*;

    }

    int countH(*Node*\* *root*){

        int x,y;

        if(*root* != NULL){

            x = countH(*root*->pLeft);

            y = countH(*root*->pRight);

            if(x>y) return x+1;

            else return y+1;

        }

        return 0;

    }

    int dia(*Node* \**root*){

        if(*root* != NULL){

            int lHeight = countH(*root*->pLeft);

            int rHeight = countH(*root*->pRight);

            int lDia = dia(*root*->pLeft);

            int rDia = dia(*root*->pRight);

            return max(lHeight+rHeight+1,max(lDia,rDia));

        }

        return 0;

    }

    int getDiameter(){

        //TODO

        return dia(root);

    }

## Q2

Given a Binary tree, the task is to traverse all the nodes of the tree using Breadth First Search algorithm and print the order of visited nodes (has no blank space at the end)

#include<iostream>

#include<string>

#include<queue>

using namespace std;

template<class K, class V>

class BinaryTree

{

public:

class Node;

private:

Node \*root;

public:

BinaryTree() : root(nullptr) {}

~BinaryTree()

{

// You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.

}

class Node

{

private:

K key;

V value;

Node \*pLeft, \*pRight;

friend class BinaryTree<K, V>;

public:

Node(K key, V value) : key(key), value(value), pLeft(NULL), pRight(NULL) {}

~Node() {}

};

void addNode(string posFromRoot, K key, V value)

{

if(posFromRoot == "")

{

this->root = new Node(key, value);

return;

}

Node\* walker = this->root;

int l = posFromRoot.length();

for (int i = 0; i < l-1; i++)

{

if (!walker)

return;

if (posFromRoot[i] == 'L')

walker = walker->pLeft;

if (posFromRoot[i] == 'R')

walker = walker->pRight;

}

if(posFromRoot[l-1] == 'L')

walker->pLeft = new Node(key, value);

if(posFromRoot[l-1] == 'R')

walker->pRight = new Node(key, value);

}

// STUDENT ANSWER BEGIN

// STUDENT ANSWER END

};

You can define other functions to help you.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("",2, 4); // Add to root  binaryTree.addNode("L",3, 6); // Add to root's left node  binaryTree.addNode("R",5, 9); // Add to root's right node  binaryTree.BFS(); | 4 6 9 |

// STUDENT ANSWER BEGIN

// You can define other functions here to help you.

void BFS()

{

}

// STUDENT ANSWER END

// STUDENT ANSWER BEGIN

    // You can define other functions here to help you.

    void levelorder(*Node* \* *root*){

        queue<*Node*\*> q;

        cout<<*root*->value;

        q.push(*root*);

        while(!q.empty()){

*Node*\* tmp = q.front(); q.pop();

            if(tmp->pLeft){

                cout<<" "<<tmp->pLeft->value;

                q.push(tmp->pLeft);

            }

            if(tmp->pRight){

                cout<<" "<<tmp->pRight->value;

                q.push(tmp->pRight);

            }

        }

    }

    void BFS()

    {

        levelorder(root);

    }

    // STUDENT ANSWER END

## Q3

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (non-negative integer), left and right are the pointers to the left node and right node of it, respectively.

**Request:** Implement function:

int distinctParities(BTNode\* root);

Where root is the root node of given binary tree (this tree has between 0 and 100000 elements). This function returns the number of **P-nodes** the binary tree has.

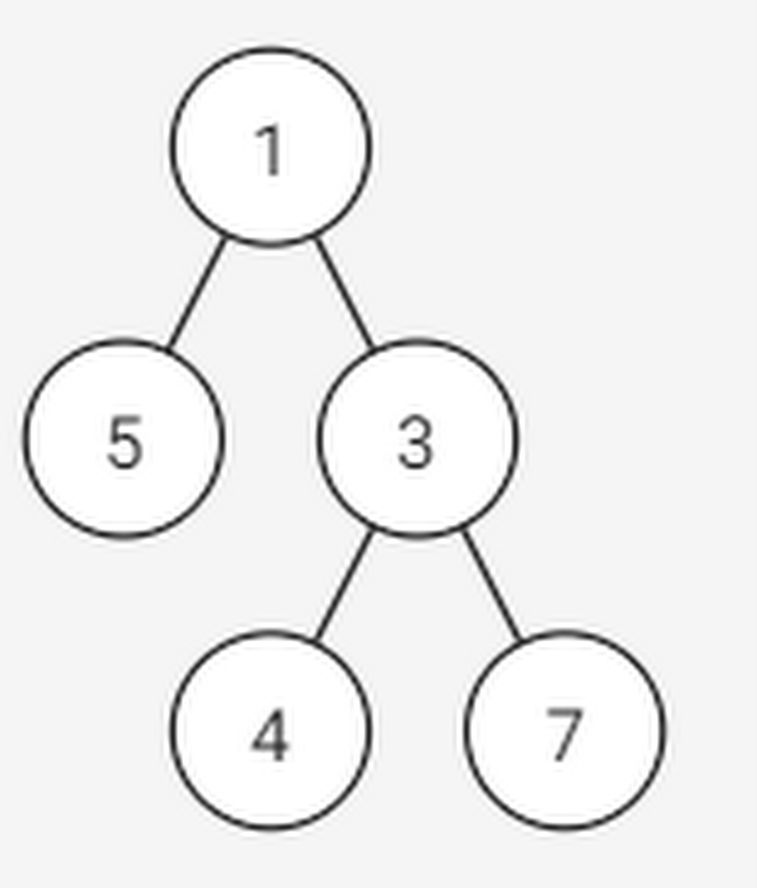
**More information:** A node is called as **P-node** if it satisfies these following rules:

- It has exactly 2 children.

- The sum of a subtree of this node is even, while the sum of the other subtree of this node is odd.

Example:

Given a binary tree in the following:



The number of **P-nodes** of this binary tree is 2, they are nodes 1 and 3.

*Note: In this exercise, the libraries iostream, stack, queue, utility and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2};  int value[] = {1,5,3,4,7};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << distinctParities(root); | 2 |
| int arr[] = {-1,0,0,1,2,2,3,3,7,1};  int value[] = {13,11,22,19,2,23,26,16,22,23};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << distinctParities(root); | 1 |

int distinctParities(BTNode\* root) {

}

int sum(*BTNode*\* *root*) {

    if(*root* != NULL){

        int x = sum(*root*->left);

        int y = sum(*root*->right);

        return x + y + *root*->val;

    }

    return 0;

}

int distinctParities(*BTNode*\* *root*) {

    if(*root* != NULL) {

        int x = distinctParities(*root*->left);

        int y = distinctParities(*root*->right);

        bool evenOdd = (sum(*root*->left)%2 == 0 && sum(*root*->right)%2 != 0)

        || (sum(*root*->left)%2 != 0 && sum(*root*->right)%2 == 0);

        if(*root*->left && *root*->right && evenOdd)

            return x + y + 1;

        else return x + y;

    }

    return 0;

}

## Q4

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (non-negative integer), left and right are the pointers to the left node and right node of it, respectively.

**Request:** Implement function:

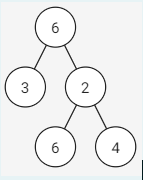
int greatAncestor(BTNode\* root);

Where root is the root node of given binary tree (this tree has between 1 and 100000 elements). This function returns the number of **great ancestor** nodes this binary tree has.

More information: A node of the binary tree is called as **great ancestor** node if its value is greater than or equal to the values of all of its descendants. Each leaf node is also a **great ancestor** node.

Example:

Given a binary tree in the following:



All of **great ancestors** nodes this binary tree has are 6, 3, 6, 4, therefore, return 4.

*Note: In this exercise, the libraries iostream, stack, queue, utility and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2};  int value[] = {6,3,2,6,4};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(arr[0]), value);  cout << greatAncestor(root); | 4 |
| int arr[] = {-1,0,0,2,3,3,4,5,6,4};  int value[] = {596,796,2168,148,1444,651,2279,1749,233,2008};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(arr[0]), value);  cout << greatAncestor(root); | 5 |

int greatAncestor(BTNode\* root) {

}

int max(int *v1*, int *v2*){

    if(*v1* > *v2*) return *v1*; else return *v2*;

}

int max(*BTNode*\* *root*){

    if(*root* != NULL){

        int x = max(*root*->left);

        int y = max(*root*->right);

        return max(*root*->val, max(x,y));

    }

    return 0;

}

int greatAncestor(*BTNode*\* *root*) {

    if(*root* != NULL) {

        int x = greatAncestor(*root*->left);

        int y = greatAncestor(*root*->right);

        if(*root*->val >= max(*root*->left) && *root*->val >= max(*root*->right))

            return x+y+1;

        else return x+y;

    }

    return 0;

}

## Q5 – dung mot phan

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (non-negative integer), left and right are the pointers to the left node and right node of it, respectively.

**Request:** Implement function:

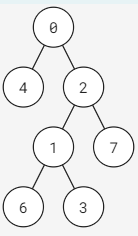
int largestDiff(BTNode\* root);

Where root is the root node of given binary tree (this tree has between 2 and 100000 elements). This function returns the largest absolute difference between any node and its descendants.

**More information:** A node is also the descendant of itself.

Example:

Given a binary tree in the following:



The largest absolute difference is between node 0 and node 7, therefore, return 7.

*Note: In this exercise, the libraries iostream, stack, queue, utility and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2,3,3};  int value[] = {0,4,2,1,7,6,3};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << largestDiff(root) << "\n"; | 7 |
| int arr[] = {-1,0};  int value[] = {1,0};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << largestDiff(root) << "\n"; | 1 |

int largestDiff(BTNode\* root) {

}

void findMM(*BTNode*\* *root*, int& *max*, int& *min*){

    //base case

    if(!root) return;

    if(root->val < min) min = root->val;

    if(root->val > max) max = root->val;

    findMM(root->left,max,min);

    findMM(root->right,max,min);

}

int largestDiff(*BTNode*\* *root*){

    int max = root->val;

    int min = root->val;

    findMM(root,max,min);

    return max-min;

}

### Code tham khảo- không hiểu 2 cái code khác gì nhau

void smallestandbiggest(BTNode \*&root,int small,int big,int&max){

if(root==nullptr) return ;

if(root->val<=small){

small=root->val;

}

if(root->val>=big){

big=root->val;

}

if(max<big-small) max=big-small;

smallestandbiggest(root->left,small,big,max);

smallestandbiggest(root->right,small,big,max);

}

int largestDiff(BTNode\* root) {

int s=root->val;

int b=root->val;

int m=0;

smallestandbiggest(root,s,b,m);

return m;

}

## Q6

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (non-negative integer), left and right are the pointers to the left node and right node of it, respectively.

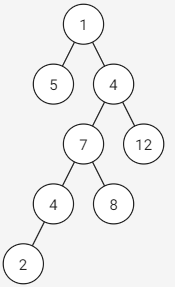
**Request:** Implement function:

int longestPathSum(BTNode\* root);

Where root is the root node of given binary tree (this tree has between 1 and 100000 elements). This function returns the sum of the largest path from the root node to a leaf node. If there are more than one equally long paths, return the larger sum.

Example:

Given a binary tree in the following:



The longest path from the root node to the leaf node is 1-4-7-4-2, so return the sum of this path, is 18.

*Note: In this exercise, the libraries iostream, utility, queue, stack and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2,3,3,5};  int value[] = {1,5,4,7,12,4,8,2};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << longestPathSum(root); | 18 |
| int arr[] = {-1,0,1,0,1,4,5,3,7,3};  int value[] = {6,12,23,20,20,20,3,9,13,15};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << longestPathSum(root); | 61 |

int longestPathSum(BTNode\* root) {

}

int max(int *v1*, int *v2*){

    if(*v1* < *v2*) return *v2*; else return *v1*;

}

int countH(*BTNode*\* *root*){

    int x,y;

    if(*root* != NULL){

        x = countH(*root*->left);

        y = countH(*root*->right);

        if(x>y) return x+1;

        else return y+1;

    }

    return 0;

}

int longestPathSum(*BTNode*\* *root*){

    if(*root* != NULL){

        int x = countH(*root*->left);

        int y = countH(*root*->right);

        if(x>y) return longestPathSum(*root*->left) + *root*->val;

        else if(x<y) return longestPathSum(*root*->right) + *root*->val;

        else return max(longestPathSum(*root*->left),longestPathSum(*root*->right)) + *root*->val;

    }

    return 0;

}

## Q7

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (non-negative integer), left and right are the pointers to the left node and right node of it, respectively.

**Request:** Implement function:

int lowestAncestor(BTNode\* root, int a, int b);

Where root is the root node of given binary tree (this tree has between 2 and 100000 elements). This function returns the **lowest ancestor** node's val of node a and node b in this binary tree (assume a and b always exist in the given binary tree).

**More information:**

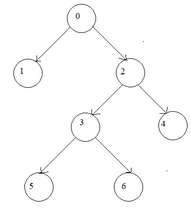
- A node is called as the **lowest ancestor** node of node a and node b if node a and node b are its descendants.

- A node is also the descendant of itself.

- On the given binary tree, each node's val is distinguish from the others' val

Example:

Given a binary tree in the following:



- The **lowest ancestor** of node 4 and node 5 is node 2.

*Note: In this exercise, the libraries iostream, stack, queue, utility and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2,3,3};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr) / sizeof(int), NULL);  cout << lowestAncestor(root, 4, 5); | 2 |
| int arr[] = {-1,0,1,1,0,4,4,2,5,6};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr) / sizeof(int), NULL);  cout << lowestAncestor(root, 4, 9); | 4 |

int lowestAncestor(BTNode\* root, int a, int b) {

}

bool search(*BTNode*\**root*,int& *a*){

    if(root==nullptr) return 0;

    if(root->val==a) return 1;

    bool check= search(root->left,a);

    if(check==1) return 1;

    check= search(root->right,a);

    return check;

}

int lowestAncestor(*BTNode*\* *root*, int *a*, int *b*) {

    if((search(root->left,a) && search(root->left,b)))

    {

        return lowestAncestor( root->left, a, b );

    }

    else if(search(root->right,a) && search(root->right,b))

    {

        return lowestAncestor( root->right, a, b );

    }

    else return root->val;

}

## Q8

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (non-negative integer), left and right are the pointers to the left node and right node of it, respectively.

**Request:** Implement function:

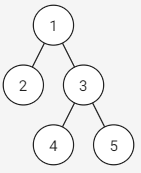
int maximizeProduct(BTNode\* root);

Where root is the root node of given binary tree (this tree has between 2 and 100000 elements). This function returns the largest P which can be gotten after deleting an edge of this tree.

**More information:** Split the binary tree into two subtrees by deleting an edge of it. Take the sum of each subtree, P is the product of these sums.

Example:

Given a binary tree in the following:



Cut the edge between nodes 3 and 5, the P we have is (1+2+3+4)\*5=50 - it is the largest P.

*Note: In this exercise, the libraries iostream, stack, queue, utility and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2};  int value[] = {1,2,3,4,5};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(arr[0]), value);  cout << maximizeProduct(root); | 50 |
| int arr[] = {-1,0,0,1,2,1,4,4,3,3};  int value[] = {4,4,5,5,5,4,0,1,3,3};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(arr[0]), value);  cout << maximizeProduct(root); | 285 |

int maximizeProduct(BTNode\* root) {}

int max(int *a*, int *b*){

    if(*a*<*b*)    return *b*;    else return *a*;

}

int sum(*BTNode*\* *root*){

    if(root){

        int x = sum(root->left);

        int y = sum(root->right);

        return root->val + x + y;

    }

    return 0;

}

void largestP(int& *maxP*, *BTNode*\* *root*, int *sumall*){

    if(!root) return;

    else{

        maxP=

        max(

            max(

                sum(root->left)\*(sumall - sum(root->left))

                ,sum(root->right)\*(sumall - sum(root->right)))

            ,maxP);

        largestP(maxP,root->right,sumall);

        largestP(maxP,root->left,sumall);

    }

}

int maximizeProduct(*BTNode*\* *root*) {

    int maxP=0;

    int sumall=sum(root);

    largestP(maxP,root,sumall);

    return maxP;

}

## Q9 – dung mot phan

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (non-negative integer), left and right are the pointers to the left node and right node of it, respectively.

**Request:** Implement function:

int secondDeepest(BTNode\* root);

Where root is the root node of given binary tree (this tree has between 2 and 100000 elements). This function returns the depth of the second deepest leaf/leaves of the tree (if there is no leaf satisfying, return -1).

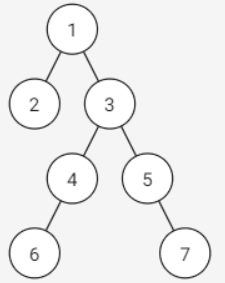
**More information:**

- The root has a depth of 0.

- In a binary tree, the second deepest leaf's/leaves' depth is smaller than the deepest leaf/leaves's depth and higher than the others' depth.

Example:

Given a binary tree in the following:



The second deepest leaf is node 2, the depth of node 2 is 1; therefore, the function returns 1.

*Note: In this exercise, the libraries iostream, stack, queue, utility and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2,3,4};  int value[] = {1,2,3,4,5,6,7};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << secondDeepest(root); | 1 |
| int arr[] = {-1,0,1,2,3,4,5,6,7,8};  int value[] = {1,2,3,4,5,6,7,8,9,10};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << secondDeepest(root); | -1 |

int secondDeepest(BTNode\* root) {

}

bool leaf(*BTNode*\* *node*){

    if(*node*->left || *node*->right) return false;

    else return true;

}

void insertST(*BTNode*\* *node*,stack<*BTNode*\*>& *st*){

    if(*node* != NULL){

        if(leaf(*node*)) *st*.push(*node*);

        else{

            insertST(*node*->left,*st*);

            insertST(*node*->right,*st*);

        }

    }

}

int depth(*BTNode* \**root*, *BTNode* \**leaf*, bool& *existed*){

    if(*root* != NULL){

        if(*root* == *leaf*) {

*existed* = true;

            return 0;

        }

        if(*root*->left){

            int x = depth(*root*->left,*leaf*,*existed*) + 1;

            if(*existed*) return x;

        }

        if(*root*->right){

            int y = depth(*root*->right,*leaf*,*existed*) + 1;

            if(*existed*) return y;

        }

    }

    return 0;

}

int secondDeepest(*BTNode*\* *root*) {

    stack<*BTNode*\*> st;

    insertST(*root*, st);

    if(st.size() >= 2){

        bool temp = false;

        int res = depth(*root*,st.top(),temp);

        st.pop();

        int tmp = 0;

        while(st.size()){

            temp = false;

            tmp = depth(*root*,st.top(),temp);

            st.pop();

            if(tmp != res) {

                break;

            }

        }

        if(tmp != res && tmp != 0) return tmp;

    }

    return -1;

}

## Q10 – dung mot phan

Class **BTNode** is used to store a node in binary tree, described on the following:

class BTNode {

public:

int val;

BTNode \*left;

BTNode \*right;

BTNode() {

this->left = this->right = NULL;

}

BTNode(int val) {

this->val = val;

this->left = this->right = NULL;

}

BTNode(int val, BTNode\*& left, BTNode\*& right) {

this->val = val;

this->left = left;

this->right = right;

}

};

Where val is the value of node (integer, in segment [0,9]), left and right are the pointers to the left node and right node of it, respectively.

**Request:** Implement function:

int sumDigitPath(BTNode\* root);

Where root is the root node of given binary tree (this tree has between 2 and 100000 elements). This function returns the sum of all **digit path** numbers of this binary tree (the result may be large, so you must use mod 27022001 before returning).

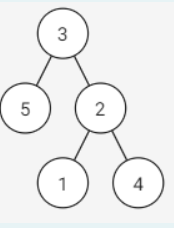
**More information:**

- A path is called as **digit path** if it is a path from the root node to the leaf node of the binary tree.

- Each **digit path** represents a number in order, each node's val of this path is a digit of this number, while root's val is the first digit.

Example:

Given a binary tree in the following:



All of the **digit paths** are 3-5, 3-2-1, 3-2-4; and the number reprensted by them are 35, 321, 324, respectively. The sum of them (after mod 27022001) is 680.

*Note: In this exercise, the libraries iostream, queue, stack, utility and using namespace std are used. You can write helper functions; however, you are not allowed to use other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {-1,0,0,2,2};  int value[] = {3,5,2,1,4};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << sumDigitPath(root); | 680 |
| int arr[] = {-1,0,0};  int value[] = {1,2,3};  BTNode\* root = BTNode::createTree(arr, arr + sizeof(arr)/sizeof(int), value);  cout << sumDigitPath(root); | 25 |

int sumDigitPath(BTNode\* root) {

}

void total(*BTNode*\* *root*, int *sum*, int& *res*){

    if(*root* != NULL){

        if(*root*->left || *root*->right){

            total(*root*->left,*sum*\*10 +*root*->val,*res*);

            total(*root*->right,*sum*\*10 +*root*->val,*res*);

        }

        else{

            //leaf node

*res* += *sum*\*10 + *root*->val;

        }

    }

}

int sumDigitPath(*BTNode*\* *root*) {

    int sum = 0;

    int res = 0;

    total(*root*,sum,res);

    return res%27022001;

}

## Q11

Given a Binary tree, the task is to count the number of nodes with two children

#include<iostream>

#include<string>

using namespace std;

template<class K, class V>

class BinaryTree

{

public:

class Node;

private:

Node \*root;

public:

BinaryTree() : root(nullptr) {}

~BinaryTree()

{

// You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.

}

class Node

{

private:

K key;

V value;

Node \*pLeft, \*pRight;

friend class BinaryTree<K, V>;

public:

Node(K key, V value) : key(key), value(value), pLeft(NULL), pRight(NULL) {}

~Node() {}

};

void addNode(string posFromRoot, K key, V value)

{

if(posFromRoot == "")

{

this->root = new Node(key, value);

return;

}

Node\* walker = this->root;

int l = posFromRoot.length();

for (int i = 0; i < l-1; i++)

{

if (!walker)

return;

if (posFromRoot[i] == 'L')

walker = walker->pLeft;

if (posFromRoot[i] == 'R')

walker = walker->pRight;

}

if(posFromRoot[l-1] == 'L')

walker->pLeft = new Node(key, value);

if(posFromRoot[l-1] == 'R')

walker->pRight = new Node(key, value);

}

// STUDENT ANSWER BEGIN

// STUDENT ANSWER END

};

You can define other functions to help you.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("",2, 4); // Add to root  binaryTree.addNode("L",3, 6); // Add to root's left node  binaryTree.addNode("R",5, 9); // Add to root's right node  cout << binaryTree.countTwoChildrenNode(); | 1 |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("",2, 4);  binaryTree.addNode("L",3, 6);  binaryTree.addNode("R",5, 9);  binaryTree.addNode("LL",4, 10);  binaryTree.addNode("LR",6, 2);  cout << binaryTree.countTwoChildrenNode(); | 2 |

// STUDENT ANSWER BEGIN

// You can define other functions here to help you.

int countTwoChildrenNode(){}

// STUDENT ANSWER END

return 0; //error: control reaches end of non-void function

    /\*

    explain: If control reaches the end of a function and no return is encountered,

    GCC assumes a return with no return value.

    However, for this, the function requires a return value.

    At the end of the function,

    add a return statement that returns a suitable return value,

    even if control never reaches there.

    \*/

// STUDENT ANSWER BEGIN

    // You can define other functions here to help you.

    int count(*Node* \* *root*){

        int x,y;

        if(*root* != NULL){

            x = count(*root*->pLeft);

            y = count(*root*->pRight);

            if(*root*->pLeft && *root*->pRight)

                return x + y+ 1;

                else return x+y;

        }

        return 0;

    }

    int countTwoChildrenNode(){

        return count(root);

    }

    // STUDENT ANSWER END

## Q12

Given class **BinaryTree**, you need to finish methods **getHeight()**, **preOrder()**, **inOrder()**, **postOrder()**.

#include <iostream>  
#include <string>  
#include <algorithm>  
#include <sstream>  
using namespace std;

template<class K, class V>  
class BinaryTree  
{  
public:  
    class Node;  
private:  
    Node\* root;  
public:  
    BinaryTree() : root(nullptr) {}  
    ~BinaryTree()  
    {  
        // You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.  
    }  
    class Node  
    {  
    private:  
        K key;  
        V value;  
        Node\* pLeft, \* pRight;  
        friend class BinaryTree<K, V>;  
    public:  
        Node(K key, V value) : key(key), value(value), pLeft(NULL), pRight(NULL) {}  
        ~Node() {}  
    };  
    void addNode(string posFromRoot, K key, V value)  
    {  
        if (posFromRoot == "")  
        {  
            this->root = new Node(key, value);  
            return;  
        }  
        Node\* walker = this->root;  
        int l = posFromRoot.length();  
        for (int i = 0; i < l - 1; i++)  
        {  
            if (!walker)  
                return;  
            if (posFromRoot[i] == 'L')  
                walker = walker->pLeft;  
            if (posFromRoot[i] == 'R')  
                walker = walker->pRight;  
        }  
        if (posFromRoot[l - 1] == 'L')  
            walker->pLeft = new Node(key, value);  
        if (posFromRoot[l - 1] == 'R')  
            walker->pRight = new Node(key, value);  
    }  
    // STUDENT ANSWER BEGIN   
          
    // STUDENT ANSWER END  
};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("", 2, 4); // Add to root  binaryTree.addNode("L", 3, 6); // Add to root's left node  binaryTree.addNode("R", 5, 9); // Add to root's right node  cout << binaryTree.getHeight() << endl;  cout << binaryTree.preOrder() << endl;  cout << binaryTree.inOrder() << endl;  cout << binaryTree.postOrder() << endl; | 2  4 6 9  6 4 9  6 9 4 |

// STUDENT ANSWER BEGIN

// You can define other functions here to help you.

int getHeight() {

// TODO: return height of the binary tree.

}

string preOrder() {

// TODO: return the sequence of values of nodes in pre-order.

}

string inOrder() {

// TODO: return the sequence of values of nodes in in-order.

}

string postOrder() {

// TODO: return the sequence of values of nodes in post-order.

}

// STUDENT ANSWER END

// STUDENT ANSWER BEGIN

    // You can define other functions here to help you.

    int countH(*Node*\* *root*){

        int x,y;

        if(*root* != NULL){

            x = countH(*root*->pLeft);

            y = countH(*root*->pRight);

            if(x>y) return x+1;

            else return y+1;

        }

        return 0;

    }

    int getHeight() {

        // TODO: return height of the binary tree.

        return countH(root);

    }

*string* pre(*Node* \**root*) {

        if(*root* != NULL){

            return to\_string(*root*->value) + " " + pre(*root*->pLeft) + pre(*root*->pRight);

        }

        return "";

    }

*string* preOrder() {

        // TODO: return the sequence of values of nodes in pre-order.

        return pre(root);

    }

*string* in(*Node* \**root*) {

        if(*root* != NULL){

            return in(*root*->pLeft) + to\_string(*root*->value) + " " + in(*root*->pRight);

        }

        return "";

    }

*string* inOrder() {

        // TODO: return the sequence of values of nodes in in-order.

        return in(root);

    }

*string* post(*Node* \**root*) {

        if(*root* != NULL){

            return post(*root*->pLeft) + post(*root*->pRight) + to\_string(*root*->value) + " ";

        }

        return "";

    }

*string* postOrder() {

        // TODO: return the sequence of values of nodes in post-order.

        return post(root);

    }

    // STUDENT ANSWER END

## Q13

Given a Binary tree, the task is to calculate the sum of leaf nodes. (Leaf nodes are nodes which have no children)

#include<iostream>

#include<string>

using namespace std;

template<class K, class V>

class BinaryTree

{

public:

class Node;

private:

Node \*root;

public:

BinaryTree() : root(nullptr) {}

~BinaryTree()

{

// You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.

}

class Node

{

private:

K key;

V value;

Node \*pLeft, \*pRight;

friend class BinaryTree<K, V>;

public:

Node(K key, V value) : key(key), value(value), pLeft(NULL), pRight(NULL) {}

~Node() {}

};

void addNode(string posFromRoot, K key, V value)

{

if(posFromRoot == "")

{

this->root = new Node(key, value);

return;

}

Node\* walker = this->root;

int l = posFromRoot.length();

for (int i = 0; i < l-1; i++)

{

if (!walker)

return;

if (posFromRoot[i] == 'L')

walker = walker->pLeft;

if (posFromRoot[i] == 'R')

walker = walker->pRight;

}

if(posFromRoot[l-1] == 'L')

walker->pLeft = new Node(key, value);

if(posFromRoot[l-1] == 'R')

walker->pRight = new Node(key, value);

}

//Helping functions

int sumOfLeafs(){

//TODO

}

};

You can write other functions to achieve this task.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("", 2, 4);  cout << binaryTree.sumOfLeafs(); | 4 |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("", 2, 4);  binaryTree.addNode("L", 3, 6);  binaryTree.addNode("R", 5, 9);  cout << binaryTree.sumOfLeafs(); | 15 |

//Helping functions

int sumOfLeafs(){

//TODO

}

//Helping functions

    int sumLeaf(*Node* \**root*){

        int x,y;

        if(*root* != NULL){

            x = sumLeaf(*root*->pLeft);

            y = sumLeaf(*root*->pRight);

            if(*root*->pLeft == NULL && *root*->pRight == NULL)

                return x +y +*root*->value;

            else return x+y;

        }

        return 0;

    }

    int sumOfLeafs(){

        //TODO

        Return sumLeaf(root);

    }