Trạng thái	Đã xong
Bắt đầu vào lúc	Thứ Ba, 8 tháng 4 2025, 3:48 PM
Kết thúc lúc	Thứ Ba, 8 tháng 4 2025, 3:50 PM
Thời gian thực hiện	1 phút 57 giây
Điểm	7,00/7,00
Điểm	**10,00** trên 10,00 (**100**%)

Câu hỏi 1

Đúng

Đạt điểm 1,00 trên 1,00

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

Answer: (penalty regime: 5, 10, 15, ... %)

Reset answer

```
// STUDENT ANSWER BEGIN
// You can define other functions here to help you.

// Hàm đệ quy để đếm số nút có hai con từ một nút cụ thể
int countTwoChildrenNodeRecursive(Node* node) {
    // Nếu nút hiện tại là NULL, trả về 0
    if (node == nullptr) return 0;

    // Khởi tạo biến đếm
    int count = 0;

    // Kiểm tra nút hiện tại có hai con không
    if (node->pLeft != nullptr && node->pRight != nullptr) {
        count = 1; // Tăng biến đếm lên 1 nếu nút hiện tại có hai con
    }

    // Đệ quy đếm các nút có hai con ở cây con bên trái
    count += countTwoChildrenNodeRecursive(node->pLeft);

    // Đệ quy đếm các nút có hai con ở cây con bên phải
    count += countTwoChildrenNodeRecursive(node->pRight);

    // Trả về tổng số nút có hai con
    return count;
}

// Hàm chính để đếm số nút có hai con trong toàn bộ cây
int countTwoChildrenNode() {
    // Gọi hàm đệ quy từ nút gốc
    return countTwoChildrenNodeRecursive(this->root);
}
// STUDENT ANSWER END
```

	Test	Expected	Got	

✓	`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	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	2	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **2**

Đúng

Đạt điểm 1,00 trên 1,00

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

Answer: (penalty regime: 5, 10, 15, ... %)

Reset answer

```cpp
// Hàm đệ quy để tính chiều cao của cây từ một nút cụ thể
int getHeightRecursive(Node* node) {
    // Trường hợp cơ bản: nếu nút hiện tại là NULL, trả về 0
    // (để khi cộng 1 thì chiều cao là 1 cho lá gốc)
    if (node == nullptr) return 0;

    // Tính chiều cao của cây con bên trái và bên phải
    int leftHeight = getHeightRecursive(node->pLeft);
    int rightHeight = getHeightRecursive(node->pRight);

    // Chiều cao của cây là chiều cao lớn nhất giữa cây con trái và phải, cộng thêm 1
    return 1 + max(leftHeight, rightHeight);
}
// Hàm chính để trả về chiều cao của cây
int getHeight() {
    return getHeightRecursive(this->root);
}

// Hàm đệ quy để duyệt cây theo thứ tự trước (pre-order): gốc -> trái -> phải
void preOrderRecursive(Node* node, string& result) {
    // Nếu nút hiện tại là NULL, không làm gì cả
    if (node == nullptr) return;

    // Thêm giá trị của nút hiện tại vào kết quả
    // Nếu kết quả không rỗng, thêm dấu cách trước giá trị mới
    if (!result.empty()) result += " ";
    result += to_string(node->value);
    // Duyệt cây con bên trái
    preOrderRecursive(node->pLeft, result);
    // Duyệt cây con bên phải
    preOrderRecursive(node->pRight, result);
}
// Hàm chính để trả về chuỗi các giá trị theo thứ tự trước
string preOrder() {
    string result = "";
    preOrderRecursive(this->root, result);
    return result;
}

// Hàm đệ quy để duyệt cây theo thứ tự giữa (in-order): trái -> gốc -> phải
void inOrderRecursive(Node* node, string& result) {
    // Nếu nút hiện tại là NULL, không làm gì cả
    if (node == nullptr) return;

    // Duyệt cây con bên trái
    inOrderRecursive(node->pLeft, result);
    // Thêm giá trị của nút hiện tại vào kết quả
    // Nếu kết quả không rỗng, thêm dấu cách trước giá trị mới
    if (!result.empty()) result += " ";
    result += to_string(node->value);
    // Duyệt cây con bên phải
    inOrderRecursive(node->pRight, result);
```

	Test	Expected	Got	

✓	`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
			2
			4 6 9
			6 4 9
			6 9 4
				✓

✓	`BinaryTree<int, int> binaryTree;`
	`binaryTree.addNode("", 2, 4);`
	
	`cout << binaryTree.getHeight() << endl;`
	`cout << binaryTree.preOrder() << endl;`
	`cout << binaryTree.inOrder() << endl;`
	`cout << binaryTree.postOrder() << endl;`
		1
		4
		4
		4
			1
			4
			4
			4
				✓

✓	`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.getHeight() << endl;`
	`cout << binaryTree.preOrder() << endl;`
	`cout << binaryTree.inOrder() << endl;`
	`cout << binaryTree.postOrder() << endl;`
		3
		4 6 10 2 9
		10 6 2 4 9
		10 2 6 9 4
			3
			4 6 10 2 9
			10 6 2 4 9
			10 2 6 9 4
				✓

✓	`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("RL", 6, 2);`
	
	`cout << binaryTree.getHeight() << endl;`
	`cout << binaryTree.preOrder() << endl;`
	`cout << binaryTree.inOrder() << endl;`
	`cout << binaryTree.postOrder() << endl;`
		3
		4 6 10 9 2
		10 6 4 2 9
		10 6 2 9 4
			3
			4 6 10 9 2
			10 6 4 2 9
			10 6 2 9 4
				✓

✓	`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("LLL",6, 2);`
	`binaryTree.addNode("LLLR",7, 7);`
	
	`cout << binaryTree.getHeight() << endl;`
	`cout << binaryTree.preOrder() << endl;`
	`cout << binaryTree.inOrder() << endl;`
	`cout << binaryTree.postOrder() << endl;`
		5
		4 6 10 2 7 9
		2 7 10 6 4 9
		7 2 10 6 9 4
			5
			4 6 10 2 7 9
			2 7 10 6 4 9
			7 2 10 6 9 4
				✓

✓	`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("LLL",6, 2);`
	`binaryTree.addNode("LLLR",7, 7);`
	`binaryTree.addNode("RR",8, 30);`
	`binaryTree.addNode("RL",9, 307);`
	
	`cout << binaryTree.getHeight() << endl;`
	`cout << binaryTree.preOrder() << endl;`
	`cout << binaryTree.inOrder() << endl;`
	`cout << binaryTree.postOrder() << endl;`
		5
		4 6 10 2 7 9 307 30
		2 7 10 6 4 307 9 30
		7 2 10 6 307 30 9 4
			5
			4 6 10 2 7 9 307 30
			2 7 10 6 4 307 9 30
			7 2 10 6 307 30 9 4
				✓

	Test	Expected	Got	

✓	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, -3);
	binaryTree.addNode("LLL",7, 2);
	binaryTree.addNode("LLLR",8, 7);
	binaryTree.addNode("RR",9, 30);
	binaryTree.addNode("RL",10, 307);
	
	cout << binaryTree.getHeight() << endl;
	cout << binaryTree.preOrder() << endl;
	cout << binaryTree.inOrder() << endl;
	cout << binaryTree.postOrder() << endl;
		5
		4 6 10 2 7 -3 9 307 30
		2 7 10 6 -3 4 307 9 30
		7 2 10 -3 6 307 30 9 4
			5
			4 6 10 2 7 -3 9 307 30
			2 7 10 6 -3 4 307 9 30
			7 2 10 -3 6 307 30 9 4
				✓

✓	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, -3);
	binaryTree.addNode("LLL",7, 2);
	binaryTree.addNode("LLLR",8, 7);
	binaryTree.addNode("RR",9, 30);
	binaryTree.addNode("RL",10, 307);
	binaryTree.addNode("RLL",11, 2000);
	binaryTree.addNode("RLR",12, 2000);
	
	cout << binaryTree.getHeight() << endl;
	cout << binaryTree.preOrder() << endl;
	cout << binaryTree.inOrder() << endl;
	cout << binaryTree.postOrder() << endl;
		5
		4 6 10 2 7 -3 9 307 2000
		2000 30
		2 7 10 6 -3 4 2000 307
		2000 9 30
		7 2 10 -3 6 2000 2000 307
		30 9 4
			5
			4 6 10 2 7 -3 9 307 2000
			2000 30
			2 7 10 6 -3 4 2000 307
			2000 9 30
			7 2 10 -3 6 2000 2000
			307 30 9 4
				✓

✓	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, -3);
	binaryTree.addNode("LLL",7, 2);
	binaryTree.addNode("LLLR",8, 7);
	binaryTree.addNode("RR",9, 30);
	binaryTree.addNode("RL",10, 307);
	binaryTree.addNode("RLL",11, 2000);
	
	cout << binaryTree.getHeight() << endl;
	cout << binaryTree.preOrder() << endl;
	cout << binaryTree.inOrder() << endl;
	cout << binaryTree.postOrder() << endl;
		5
		4 6 10 2 7 -3 9 307 2000
		30
		2 7 10 6 -3 4 2000 307 9
		30
		7 2 10 -3 6 2000 307 30 9
		4
			5
			4 6 10 2 7 -3 9 307 2000
			30
			2 7 10 6 -3 4 2000 307 9
			30
			7 2 10 -3 6 2000 307 30
			9 4
				✓

✓	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, -3);
	binaryTree.addNode("LLL",7, 2);
	binaryTree.addNode("LLLR",8, 7);
	binaryTree.addNode("RR",9, 30);
	binaryTree.addNode("RL",10, 307);
	binaryTree.addNode("RLL",11, 2000);
	binaryTree.addNode("RLLL",11, 2000);
	
	cout << binaryTree.getHeight() << endl;
	cout << binaryTree.preOrder() << endl;
	cout << binaryTree.inOrder() << endl;
	cout << binaryTree.postOrder() << endl;
		5
		4 6 10 2 7 -3 9 307 2000
		2000 30
		2 7 10 6 -3 4 2000 2000
		307 9 30
		7 2 10 -3 6 2000 2000 307
		30 9 4
			5
			4 6 10 2 7 -3 9 307 2000
			2000 30
			2 7 10 6 -3 4 2000 2000
			307 9 30
			7 2 10 -3 6 2000 2000
			307 30 9 4
				✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **3**

Đúng

Đạt điểm 1,00 trên 1,00

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

Answer: (penalty regime: 0 %)

Reset answer

```
/*
Hàm sumOfLeafs() sẽ duyệt cây theo phương pháp đệ quy dạng duyệt sâu (Depth-First Search - DFS).
Cụ thể, quá trình duyệt diễn ra như sau:
- Bắt đầu từ nút gốc (root)
- Đối với mỗi nút đang xét:
Kiểm tra xem nút có phải là NULL không. Nếu là NULL, trả về 0.
Kiểm tra xem nút có phải là nút lá không (không có con trái và phải).
- Nếu là nút lá, trả về giá trị của nút đó.
- Nếu không phải nút lá, hàm sẽ đệ quy xuống cả cây con bên trái và cây con bên phải.
Cộng kết quả từ cả hai cây con lại với nhau và trả về tổng đó.
*/

//Hàm tính tổng giá trị của tất cả các "nút lá" trong cây nhị phân.
int sumOfLeafsHelper(Node* node) {
    // Trường hợp cơ sở: nếu nút hiện tại là NULL, trả về 0
    if (node == nullptr) return 0;
    // Kiểm tra xem nút hiện tại có phải là lá không (không có con trái và con phải)
    if (node->pLeft == nullptr && node->pRight == nullptr) {
        return node->value;  // Trả về giá trị của nút lá
    }

    // Nếu không phải lá, đệ quy tính tổng của các nút lá ở cây con trái và phải
    return sumOfLeafsHelper(node->pLeft) + sumOfLeafsHelper(node->pRight);
}

int sumOfLeafs() {
    // Gọi hàm đệ quy helper từ nút gốc
    return sumOfLeafsHelper(this->root);
}
```

	Test	Expected	Got	

✓	`BinaryTree<int, int> binaryTree;`
	`binaryTree.addNode("", 2, 4);`
	`cout << binaryTree.sumOfLeafs();`
		4	4	✓

✓	`BinaryTree<int, int> binaryTree;`
	`binaryTree.addNode("", 2, 4);`
	`binaryTree.addNode("L", 3, 6);`
	`binaryTree.addNode("R", 5, 9);`
	`cout << binaryTree.sumOfLeafs();`
		15	15	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **4**

Đúng

Đạt điểm 1,00 trên 1,00

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.

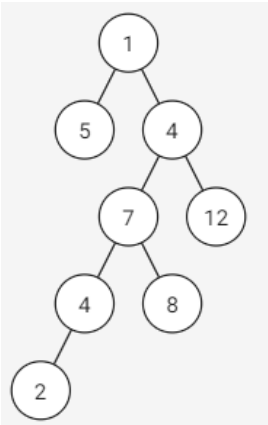
Request: Implement function:

```
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`.

Explanation of function ***`createTree`****: The function has three parameters. The first two parameters take in an array containing the parent of each Node of the binary tree, and the third parameter takes in an array representing the respective values of the Nodes. After processing, the function will construct the binary tree and return the address of the root Node. Note that the root Node is designated with a parent value of -1.*

Example*:*

```
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);
```

`arr[0]=-1` means the Node containing the value `value[0]=3` will be the root Node. Also, since `arr[1]=arr[2]=0`, it implies that the Nodes containing the values `value[1]=5` and `value[2]=2` will have the Node containing the value `value[0]=3` as their parent. Lastly, since `arr[3]=arr[4]=2`, it means the Nodes containing the

values `value[3]=1` and `value[4]=4` will have the Node with the value `value[2]=2` as their parent. Final tree of this example are shown in the figure above.
Note that whichever Node appears first in the `arr` sequence will be the left Node, and the TestCase always ensures that the resulting tree is a binary tree.

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

Answer: (penalty regime: 0, 0, 0, 5, 10, ... %)

Reset answer

```
int longestPathSum(BTNode* root) {
    // Nếu cây rỗng, trả về 0
    if (root == NULL) return 0;

    // Biến để lưu trữ độ dài đường đi dài nhất và tổng giá trị của nó
    int maxLength = -1;  // Độ dài path dài nhất Tổng số node
    int maxSum = 0;      // Tổng giá trị path dài nhất của tất cả cách Node

    // Sử dụng hàng đợi để duyệt cây theo chiều rộng (BFS)
    // Mỗi phần tử trong hàng đợi lưu: {node, chiều dài path tới node, tổng giá trị path tới node}
    queue<pair<BTNode*, pair<int, int>>> q;
    q.push({root, {1, root->val}});  // Bắt đầu từ root với chiều dài 1 và tổng bằng giá trị của root

    while (!q.empty()) {
        // Mỗi phần tử trong hàng đợi chứa ba thông tin:

        BTNode* node = q.front().first;          // Một con trỏ đến nút hiện tại
        int length = q.front().second.first;    // Độ dài của đường đi từ gốc đến nút hiện tại
        int sum = q.front().second.second;      // Tổng giá trị của tất cả các nút trên đường đi đó
        q.pop();

        // Kiểm tra nếu là nút lá (không có con trái và phải)
        // Khi gặp một nút lá (không có con trái và con phải),
        // so sánh độ dài đường đi với độ dài đường đi dài nhất đã tìm thấy:
        if (node->left == NULL && node->right == NULL) {
            // Cập nhật maxLength và maxSum tương ứng nếu tìm thấy
            //  đường đi hiện tại dài hơn
            //  đường đi hiện tại có cùng độ dài với đường đi dài nhất,nhưng có tổng giá trị lớn hơn
            if (length > maxLength || (length == maxLength && sum > maxSum)) {
                maxLength = length;
                maxSum = sum;
            }
        }
        // Thêm con trái vào hàng đợi nếu tồn tại
        if (node->left != NULL) {
            q.push({node->left, {length + 1, sum + node->left->val}});
        }
        // Thêm con phải vào hàng đợi nếu tồn tại
        if (node->right != NULL) {
            q.push({node->right, {length + 1, sum + node->right->val}});
        }
    }
    // Cuối cùng, hàm trả về tổng giá trị của đường đi dài nhất
    //từ gốc đến nút lá (hoặc nếu có nhiều đường đi cùng độ dài thì trả về đường đi có tổng lớn nhất).
    return maxSum;
}
/*
Độ phức tạp:
```

```
49 Thời gian: O(n) - mỗi nút trong cây được duyệt qua đúng một lần
50 Không gian: O(w) - với w là chiều rộng lớn nhất của cây (số lượng nút tối đa ở một tầng)
51 */
```

	Test	Expected	Got	

✓	`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	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	61	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **5**

Đúng

Đạt điểm 1,00 trên 1,00

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

Answer: (penalty regime: 0 %)

Reset answer

```
// STUDENT ANSWER BEGIN
//Breadth First Search algorithm
void BFS() {
    // Kiểm tra nếu cây rỗng (root là nullptr), không thực hiện gì cả.
    if (this->root == nullptr) return;
    // Tạo một hàng đợi (`queue`) và thêm nút gốc (root) vào hàng đợi ban đầu.
    queue<Node*> q;
    q.push(this->root);

    // Biến để theo dõi xem đã in ra phần tử đầu tiên chưa
    // Sử dụng biến `isFirst` để theo dõi khi nào cần in dấu cách giữa các giá trị nút.
    // Điều này giúp đảm bảo không có dấu cách thừa ở cuối chuỗi kết quả.
    bool isFirst = true;

    // Thực hiện vòng lặp chính của BFS:
    // Duyệt qua từng nút trong cây
    while (!q.empty()) {
        // Lấy nút hiện tại từ đầu hàng đợi
        Node* current = q.front();
        q.pop();

        // In ra giá trị của nút hiện tại
        // Nếu là nút đầu tiên, không cần in dấu cách phía trước
        if (isFirst) {
            cout << current->value;
            isFirst = false;
        } else {
            cout << " " << current->value;
        }

        // Thêm các nút con (trái trước, phải sau) vào cuối hàng đợi nếu chúng tồn tại
        // Đẩy con trái vào hàng đợi nếu có
        if (current->pLeft != nullptr) {
            q.push(current->pLeft);
        }
        // Đẩy con phải vào hàng đợi nếu có
        if (current->pRight != nullptr) {
            q.push(current->pRight);
        }

    }
}
// STUDENT ANSWER END
/*
- Thuật toán này đảm bảo duyệt các nút theo từng tầng,
từ trên xuống dưới, từ trái sang phải - đây chính là nguyên tắc của thuật toán BFS.
- Độ phức tạp thời gian của thuật toán là O(n) với n là số lượng nút trong cây,
vì mỗi nút được thêm và lấy ra khỏi hàng đợi đúng một lần.
*/
```

	Test	Expected	Got	

✓	`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	4 6 9	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **6**

Đúng

Đạt điểm 1,00 trên 1,00

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).

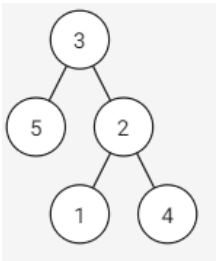
More information:

- 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`.

Explanation of function `createTree`: The function has three parameters. The first two parameters take in an array containing the parent of each Node of the binary tree, and the third parameter takes in an array representing the respective values of the Nodes. After processing, the function will construct the binary tree and return the address of the root Node. Note that the root Node is designated with a parent value of -1.

Example:

```
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);
```

`arr[0]=-1` means the Node containing the value `value[0]=3` will be the root Node. Also, since `arr[1]=arr[2]=0`, it implies that the Nodes containing the values `value[1]=5` and `value[2]=2` will have the Node containing the value `value[0]=3` as their parent. Lastly, since `arr[3]=arr[4]=2`, it means the Nodes containing the values `value[3]=1` and `value[4]=4` will have the Node with the value `value[2]=2` as their parent. Final tree of this example are shown in the figure above.
Note that whichever Node appears first in the `arr` sequence will be the left Node, and the TestCase always ensures that the resulting tree is a binary tree.

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

Answer: (penalty regime: 0 %)

Reset answer

```
// Hàm trợ giúp tính tổng các đường đi một cách đệ quy
void calculatePathSum(BTNode* node, long long currentNumber, long long& totalSum, const int MOD) {
  // Nếu nút là NULL, trả về ngay (điều kiện dừng)
    if (node == NULL) return;

    // Cập nhật số hiện tại cho đường đi này (nhân với 10 và cộng thêm chữ số hiện tại) xe VD
    // Cập nhật số hiện tại: currentNumber = (currentNumber * 10 + node->val) % MOD
    // Nhân với 10 để dịch chữ số sang trái
    // Cộng thêm giá trị của nút hiện tại vào hàng đơn vị
    // Lấy dư theo MOD để tránh tràn số
    currentNumber = (currentNumber * 10 + node->val) % MOD;

    // Kiểm tra xem đây có phải là nút lá không, Nếu đây là nút lá thì:
    if (node->left == NULL && node->right == NULL) {
        // Đã đến cuối đường đi, thêm số vào tổng
        totalSum = (totalSum + currentNumber) % MOD;
        return;
    }
    // Tiếp tục duyệt xuống cây, Nếu không phải nút lá, tiếp tục đệ quy:
    calculatePathSum(node->left, currentNumber, totalSum, MOD);
    calculatePathSum(node->right, currentNumber, totalSum, MOD);
}
int sumDigitPath(BTNode* root) {
    if (root == NULL) return 0;

    const int MOD = 27022001;
    long long totalSum = 0;
    // Bắt đầu tính toán đệ quy từ gốc với số ban đầu là 0
    calculatePathSum(root, 0, totalSum, MOD);
    return totalSum;
}
/*
1.Ý tưởng tổng quát:
- Dùng thuật toán duyệt cây theo chiều sâu (DFS) để đi từ gốc đến các nút lá
- Khi đi xuống từng nút, ta xây dựng số bằng cách nhân số hiện tại với 10
và cộng thêm giá trị của nút hiện tại
- Khi đến nút lá, ta có một số hoàn chỉnh và cộng nó vào tổng kết quả
2. Hàm calculatePathSum:
- node: Con trỏ đến nút hiện tại đang xét
- currentNumber: Số được tạo ra từ đường đi từ gốc đến nút hiện tại
- totalSum: Tổng của tất cả các số từ các đường đi (truyền bằng tham chiếu)
- MOD: Số chia lấy dư để tránh tràn số
3. Hàm sumDigitPath:
Kiểm tra nếu cây rỗng (root == NULL), trả về 0
Khởi tạo giá trị MOD = 27022001 (theo yêu cầu bài toán)
```

```
46 Khởi tạo tổng = 0
47 Gọi hàm đệ quy để tính toán: calculatePathSum(root, 0, totalSum, MOD)
48 Trả về kết quả tổng
49 */
```

	Test	Expected	Got	
✓	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	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	25	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **7**

Đúng

Đạt điểm 1,00 trên 1,00

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:

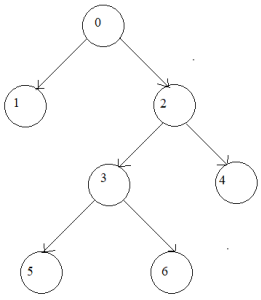
- A node is called as the **lowest ancestor** node of node `a` and node `b` if node `a` and node `b` are its descendants.

- 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`.

Explanation of function `createTree`: The function has three parameters. The first two parameters take in an array containing the parent of each Node of the binary tree, and the third parameter takes in an array representing the respective values of the Nodes. After processing, the function will construct the binary tree and return the address of the root Node. Note that the root Node is designated with a parent value of -1.

Example:

```
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);
```

`arr[0]=-1` means the Node containing the value `value[0]=3` will be the root Node. Also, since `arr[1]=arr[2]=0`, it implies that the Nodes containing the values `value[1]=5` and `value[2]=2` will have the Node containing the value `value[0]=3` as their parent. Lastly, since `arr[3]=arr[4]=2`, it means the Nodes containing the values `value[3]=1` and `value[4]=4` will have the Node with the value `value[2]=2` as their parent. Final tree of this example are shown in the figure above.
Note that whichever Node appears first in the `arr` sequence will be the left Node, and the TestCase always ensures that the resulting tree is a binary tree.

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

Answer: (penalty regime: 0 %)

Reset answer

```
// Hàm trợ giúp để tìm một nút có giá trị cụ thể VAL trong cây nhị phân
BTNode* findNode(BTNode* root, int val) {
    if (root == NULL) return NULL;      // Nếu root là NULL, trả về NULL (không tìm thấy)
    if (root->val == val) return root;  // Nếu giá trị của nút hiện tại bằng val, trả về nút hiện tại

    // Đệ quy tìm kiếm trong cây con trái
    BTNode* leftResult = findNode(root->left, val);
    if (leftResult != NULL) return leftResult;  // Nếu tìm thấy trong cây con trái, trả về kết quả đó

    return findNode(root->right, val);  // Nếu không, đệ quy tìm kiếm trong cây con phải và trả về kết quả
}

// Hàm trợ giúp để Tìm tổ tiên chung thấp nhất của hai nút nodeA và nodeB.
// Thuật toán: Dùng phương pháp đệ quy để tìm kiếm từ gốc xuống.
BTNode* findLCA(BTNode* root, BTNode* nodeA, BTNode* nodeB) {
    // Nếu root là NULL, trả về NULL (không có tổ tiên chung)
    if (root == NULL) return NULL;

    // Nếu root trùng với nodeA hoặc nodeB, trả về root (vì một nút cũng là tổ tiên của chính nó)
    if (root == nodeA || root == nodeB) return root;

    // Tìm kiếm a và b trong cây con trái và cây con phải
    // Đệ quy tìm kiếm tổ tiên chung trong cây con trái (leftLCA)
    BTNode* leftLCA = findLCA(root->left, nodeA, nodeB);
    // Đệ quy tìm kiếm tổ tiên chung trong cây con phải (rightLCA)
    BTNode* rightLCA = findLCA(root->right, nodeA, nodeB);

    // Nếu cả leftLCA và rightLCA đều không NULL, có nghĩa là một nút
    // nằm trong cây con trái và nút kia nằm trong cây con phải,
    // do đó nút hiện tại (root) chính là tổ tiên chung thấp nhất
    if (leftLCA && rightLCA) return root;

    // Nếu không, kiểm tra xem LCA nằm trong cây con trái hay cây con phải
    // Nếu không, trả về nút không NULL trong hai kết quả trên (nếu có)
    return (leftLCA != NULL) ? leftLCA : rightLCA;
}

// Tìm và trả về giá trị của tổ tiên chung thấp nhất của hai nút có giá trị a và b.
int lowestAncestor(BTNode* root, int a, int b) {
    // Đầu tiên, tìm con trỏ đến hai nút có giá trị a và b bằng cách gọi hàm findNode
    BTNode* nodeA = findNode(root, a);
    BTNode* nodeB = findNode(root, b);

    // Sau đó, tìm tổ tiên chung thấp nhất của hai nút này bằng cách gọi hàm findLCA
    BTNode* lca = findLCA(root, nodeA, nodeB);

```

```
47      // Tra ve gia tri cua to tien chung thap nhat
48      return lca->val;
49  }
```

	Test	Expected	Got	
✓	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	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	4	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.