Câu hỏi 1

Chính xác

Chấm điểm của 1,00

In this question, you have to perform **rotate nodes** on AVL tree. Note that:

- When adding a node which has the same value as parent node, add it in the **right sub tree**.

Your task is to implement function: **rotateRight, rotateLeft**. You could define one or more functions to achieve this task.

```
#include <iostream>
#include <math.h>
#include <queue>
using namespace std;
#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue
{
    LH = -1,
    EH = 0,
    RH = 1
};

void printNSpace(int n)
{
    for (int i = 0; i < n - 1; i++)
        cout << " ";
}

void printInteger(int &n)
{
    cout << n << " ";
}

template<class T>
class AVLTree
{
public:
    class Node;
private:
    Node *root;
protected:
    int getHeightRec(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh = this->getHeightRec(node->pLeft);
        int rh = this->getHeightRec(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){}
    int getHeight()
    {
        return this->getHeightRec(this->root);
    }
    void printTreeStructure()
    {
        int height = this->getHeight();
        if (this->root == NULL)
        {
            cout << "NULL\n";
            return;
        }
        queue<Node *> q;
        q.push(root);
        Node *temp;
        int count = 0;
        int maxNode = 1;
        int level = 0;
        int space = pow(2, height);
        printNSpace(space / 2);
        while (!q.empty())
        {
            temp = q.front();
            q.pop();
            if (temp == NULL)
            {
```

```
                cout << " ";
                q.push(NULL);
                q.push(NULL);
            }
            else
            {
                cout << temp->data;
                q.push(temp->pLeft);
                q.push(temp->pRight);
            }
            printNSpace(space);
            count++;
            if (count == maxNode)
            {
                cout << endl;
                count = 0;
                maxNode *= 2;
                level++;
                space /= 2;
                printNSpace(space / 2);
            }
            if (level == height)
                return;
        }
    }

    void insert(const T &value);
```

```
int getBalance(Node*subroot){
    if(!subroot) return 0;
    return getHeightRec(subroot->pLeft)- getHeightRec(subroot->pRight);
}
```

```
Node* rotateLeft(Node* subroot)
```

```
{
```

```
    //TODO: Rotate and return new root after rotate
```

```
};
```

```
Node* rotateRight(Node* subroot)
```

```
{
```

```
    //TODO: Rotate and return new root after rotate
```

```
};
```

```
    class Node
    {
    private:
        T data;
        Node *pLeft, *pRight;
        BalanceValue balance;
        friend class AVLTree<T>;

    public:
        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}
        ~Node() {}
    };
};
```

For example:

Test	Result

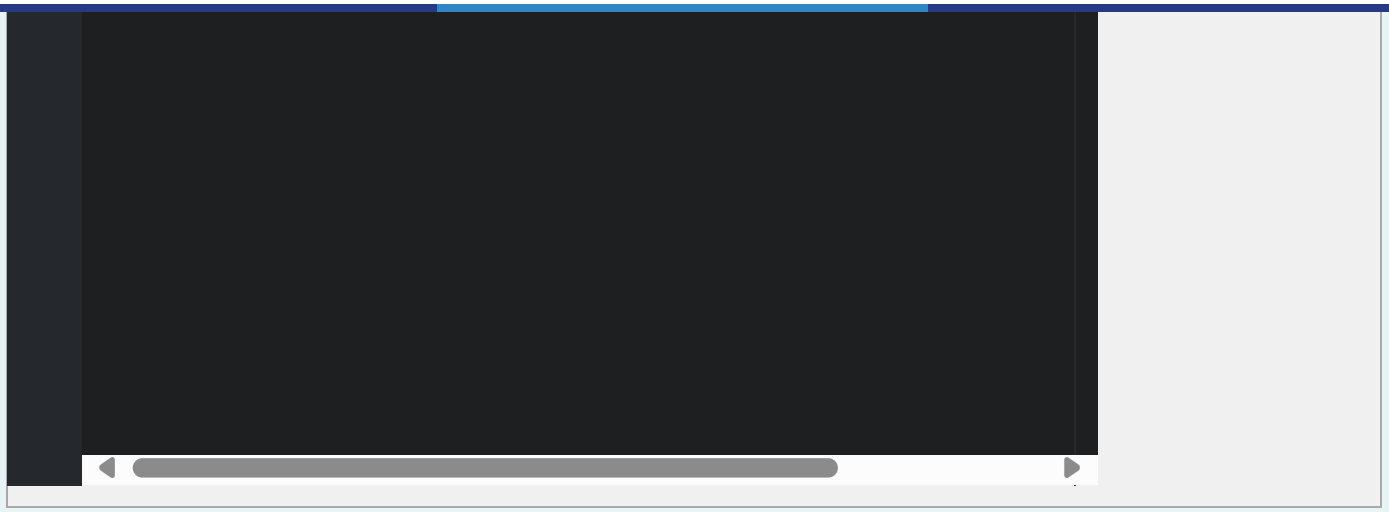
`// Test rotateLeft`
`AVLTree<int> avl;`
`avl.insert(0);`
`avl.insert(1);`
`cout << "After inserting 0, 1. Tree:" << endl;`
`avl.printTreeStructure();`
`avl.insert(2);`
`cout << endl << "After inserting 2, perform 'rotateLeft'. Tree:" << endl;`
`avl.printTreeStructure();`
	`After inserting 0, 1. Tree:`
	` 0`
	`  1`
	
	`After inserting 2, perform 'rotateLeft'. Tree:`
	` 1`
	`0 2`

`// Test rotateRight`
`AVLTree<int> avl;`
`avl.insert(10);`
`avl.insert(9);`
`cout << "After inserting 10, 9. Tree:" << endl;`
`avl.printTreeStructure();`
`avl.insert(8);`
`cout << endl << "After inserting 8, perform 'rotateRight'. Tree:" << endl;`
`avl.printTreeStructure();`
	`After inserting 10, 9. Tree:`
	` 10`
	`9`
	
	`After inserting 8, perform 'rotateRight'. Tree:`
	` 9`
	`8 10`

Answer: (penalty regime: 0 %)

Reset answer

```
Node* rotateRight(Node* root) {
    Node* newRoot = root->pLeft;
    Node* transferSubtree = newRoot->pRight;

    // Perform rotation
    newRoot->pRight = root;
    root->pLeft = transferSubtree;

    // Update balance factor
    root->balance = BalanceValue(max(getHeightRec(root->pLeft), getHeightRec(root->
    newRoot->balance = BalanceValue(max(getHeightRec(newRoot->pLeft), getHeightRec(

    // Return new root
    return newRoot;
}

Node* rotateLeft(Node* root) {
    Node* newRoot = root->pRight;
    Node* transferSubtree = newRoot->pLeft;

    // Perform rotation
    newRoot->pLeft = root;
    root->pRight = transferSubtree;

    // Update balance factor
    root->balance = BalanceValue(max(getHeightRec(root->pLeft), getHeightRec(root->
    newRoot->balance = BalanceValue(max(getHeightRec(newRoot->pLeft), getHeightRec(

    // Return new root
    return newRoot;
}

```



Precheck Kiểm tra

	Test	Expected	Got	

✔	`// Test rotateLeft`
	`AVLTree<int> avl;`
	`avl.insert(0);`
	`avl.insert(1);`
	`cout << "After inserting 0, 1. Tree:" << endl;`
	`avl.printTreeStructure();`
	`avl.insert(2);`
	`cout << endl << "After inserting 2, perform 'rotateLeft'. Tree:" << endl;`
	`avl.printTreeStructure();`
		`After inserting 0, 1. Tree:`
		` 0`
		`  1`
		
		`After inserting 2, perform 'rotateLeft'. Tree:`
		` 1`
		`0 2`
			`After inserting 0, 1. Tree:`
			` 0`
			`  1`
			
			`After inserting 2, perform 'rotateLeft'. Tree:`
			` 1`
			`0 2`
				✔

✔	`// Test rotateRight`
	`AVLTree<int> avl;`
	`avl.insert(10);`
	`avl.insert(9);`
	`cout << "After inserting 10, 9. Tree:" << endl;`
	`avl.printTreeStructure();`
	`avl.insert(8);`
	`cout << endl << "After inserting 8, perform 'rotateRight'. Tree:" << endl;`
	`avl.printTreeStructure();`
		`After inserting 10, 9. Tree:`
		` 10`
		`9`
		
		`After inserting 8, perform 'rotateRight'. Tree:`
		` 9`
		`8 10`
			`After inserting 10, 9. Tree:`
			` 10`
			`9`
			
			`After inserting 8, perform 'rotateRight'. Tree:`
			` 9`
			`8 10`
				✔

Passed all tests! ✔

BKSI

LIÊN HỆ

268 Lý Thường Kiệt, P.14, Q.10, TP.HCM

(028) 38 651 670 - (028) 38 647 256 (Ext: 5258, 5234)

elearning@hcmut.edu.vn

Câu hỏi 2

Chính xác

Chấm điểm của 1,00

In this question, you have to perform **add** on AVL tree. Note that:

- When adding a node which has the same value as parent node, add it in the **right sub tree**.

Your task is to implement function: **insert**. The function should cover at least these cases:

+ Balanced tree

+ Left of left unbalanced tree

+ Right of left unbalanced tree

You could define one or more functions to achieve this task.

```
#include <iostream>
#include <math.h>
#include <queue>
using namespace std;
#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue
{
    LH = -1,
    EH = 0,
    RH = 1
};

void printNSpace(int n)
{
    for (int i = 0; i < n - 1; i++)
        cout << " ";
}

void printInteger(int &n)
{
    cout << n << " ";
}

template<class T>
class AVLTree
{
public:
    class Node;
private:
    Node *root;
protected:
    int getHeightRec(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh = this->getHeightRec(node->pLeft);
        int rh = this->getHeightRec(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){}
    int getHeight()
    {
        return this->getHeightRec(this->root);
    }
    void printTreeStructure()
    {
        int height = this->getHeight();
        if (this->root == NULL)
        {
            cout << "NULL\n";
            return;
        }
        queue<Node *> q;
        q.push(root);
        Node *temp;
        int count = 0;
        int maxNode = 1;
        int level = 0;
        int space = pow(2, height);
        printNSpace(space / 2);
        while (!q.empty())
        {
            temp = q.front();
            q.pop();
            if (temp == NULL)
            {
```

```
                cout << " ";
                q.push(NULL);
                q.push(NULL);
            }
            else
            {
                cout << temp->data;
                q.push(temp->pLeft);
                q.push(temp->pRight);
            }
            printNSpace(space);
            count++;
            if (count == maxNode)
            {
                cout << endl;
                count = 0;
                maxNode *= 2;
                level++;
                space /= 2;
                printNSpace(space / 2);
            }
            if (level == height)
                return;
        }
    }

    void insert(const T &value)
    {
        //TODO
    }

    class Node
    {
    private:
        T data;
        Node *pLeft, *pRight;
        BalanceValue balance;
        friend class AVLTree<T>;

    public:
        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}
        ~Node() {}
    };
};
```

For example:

Test	Result

`AVLTree<int> avl;`
`for (int i = 0; i >= -10; i--){`
`  avl.insert(i);`
`}`
`avl.printTreeStructure();`
	`      -3`
	`  -7      -1`
	`-9  -5  -2  0`
	`-10 -8 -6 -4`

`AVLTree<int> avlTree;`
`avlTree.insert(5);`
`avlTree.insert(7);`
`avlTree.insert(6);`
`avlTree.printTreeStructure();`
	`6`
	`5 7`

Answer: (penalty regime: 0 %)

Reset answer

```
1  // Function to perform a left rotation on a node
2  Node* rotateLeft(Node* node) {
3      Node* temp = node->pRight; // Temporary node for swapping
4      node->pRight = temp->pLeft; // The left child of the temporary node becomes
```

```
    temp->pLeft = node; // The original node becomes the left child of the tempo
    return temp; // The temporary node is now the parent node
}

// Function to perform a right rotation on a node
Node* rotateRight(Node* node) {
    Node* temp = node->pLeft; // Temporary node for swapping
    node->pLeft = temp->pRight; // The right child of the temporary node becomes
    temp->pRight = node; // The original node becomes the right child of the tem
    return temp; // The temporary node is now the parent node
}

// Function to get the balance factor of a node
int getBalance(Node* node) {
    if (node == NULL)
        return 0; // If the node is null, its balance factor is 0
    return getHeightRec(node->pLeft) - getHeightRec(node->pRight); // The balanc
}

// Recursive function to insert a node and balance the tree
Node* insertRec(Node* node, T value) {
    if (node == NULL)
        return new Node(value); // If the node is null, a new node is created
    if (value < node->data)
        node->pLeft = insertRec(node->pLeft, value); // If the value is less tha
    else if (value > node->data)
        node->pRight = insertRec(node->pRight, value); // If the value is greate
    else
        return node; // If the value is equal to the node's data, the node is re

    int balance = getBalance(node); // The balance factor of the node is calcula

    // If the node is unbalanced and the value is less than the node's left chil
    if (balance > 1 && value < node->pLeft->data)
        return rotateRight(node);

    // If the node is unbalanced and the value is greater than the node's right
    if (balance < -1 && value > node->pRight->data)
        return rotateLeft(node);

    // If the node is unbalanced, the value is greater than the node's left chil
    if (balance > 1 && value > node->pLeft->data) {
        node->pLeft = rotateLeft(node->pLeft);
        return rotateRight(node);
    }

    // If the node is unbalanced, the value is less than the node's right child'
    if (balance < -1 && value < node->pRight->data) {
        node->pRight = rotateRight(node->pRight);
        return rotateLeft(node);
    }

    return node; // The node is returned
}

// Function to insert a node into the tree
void insert(const T &value) {
```

Precheck Kiểm tra

	Test	Expected	Got	

✔	`AVLTree<int> avl;`
	`for (int i = 0; i >= -10; i--){`
	`  avl.insert(i);`
	`}`
	`avl.printTreeStructure();`
		-3
		-7 -1
		-9 -5 -2 0
		-10 -8 -6 -4
			-3
			-7 -1
			-9 -5 -2 0
			-10 -8 -6 -4
				✔

	Test	Expected	Got	

✔	`AVLTree<int> avlTree;`
	`avlTree.insert(5);`
	`avlTree.insert(7);`
	`avlTree.insert(6);`
	`avlTree.printTreeStructure();`
		`6`
		`5 7`
			`6`
			`5 7`
				✔

Passed all tests! ✔

Câu hỏi 3

Chính xác

Chấm điểm của 1,00

In this question, you have to perform add on AVL tree. Note that:

> When adding a node which has the same value as parent node, add it in the right sub tree.

Your task is to implement function: `insert`. The function should cover at least these cases:

- Balanced tree
- Right of right unbalanced tree
- Left of right unbalanced tree

You could define one or more functions to achieve this task.

```
#include <iostream>
#include <math.h>
#include <queue>
using namespace std;
#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue
{
    LH = -1,
    EH = 0,
    RH = 1
};

void printNSpace(int n)
{
    for (int i = 0; i < n - 1; i++)
        cout << " ";
}

void printInteger(int &n)
{
    cout << n << " ";
}

template<class T>
class AVLTree
{
public:
    class Node;
private:
    Node *root;
protected:
    int getHeightRec(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh = this->getHeightRec(node->pLeft);
        int rh = this->getHeightRec(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){}
    int getHeight()
    {
        return this->getHeightRec(this->root);
    }
    void printTreeStructure()
    {
        int height = this->getHeight();
        if (this->root == NULL)
        {
            cout << "NULL\n";
            return;
        }
        queue<Node *> q;
        q.push(root);
        Node *temp;
        int count = 0;
        int maxNode = 1;
        int level = 0;
        int space = pow(2, height);
        printNSpace(space / 2);
        while (!q.empty())
        {
            temp = q.front();
            q.pop();
```

```
            if (temp == NULL)
            {
                cout << " ";
                q.push(NULL);
                q.push(NULL);
            }
            else
            {
                cout << temp->data;
                q.push(temp->pLeft);
                q.push(temp->pRight);
            }
            printNSpace(space);
            count++;
            if (count == maxNode)
            {
                cout << endl;
                count = 0;
                maxNode *= 2;
                level++;
                space /= 2;
                printNSpace(space / 2);
            }
            if (level == height)
                return;
        }
    }

    void insert(const T &value)
    {
        //TODO
    }

    class Node
    {
    private:
        T data;
        Node *pLeft, *pRight;
        BalanceValue balance;
        friend class AVLTree<T>;

    public:
        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}
        ~Node() {}
    };
};
```

For example:

Test	Result

`AVLTree<int> avl;`
`int nums[] = {3, 1, 6, 2, 4, 8, 5, 7, 9};`
`for (int i = 0; i < 9; i++){`
`  avl.insert(nums[i]);`
`}`
`avl.printTreeStructure();`
	`  3`
	`1  6`
	`2 4 8`
	`5 7 9`

`AVLTree<int> avl;`
`int nums[] = {6, 8, 3, 5, 7, 9, 1, 2, 4};`
`for (int i = 0; i < 9; i++){`
`  avl.insert(nums[i]);`
`}`
`avl.printTreeStructure();`
	`  6`
	`3  8`
	`1 5 7 9`
	`2 4`

Answer: (penalty regime: 0 %)

Reset answer

```
void rotateLeft(Node *&root){
    Node *tempPtr = root->pRight;
    root->pRight = tempPtr->pLeft;
    tempPtr->pLeft = root;
    root = tempPtr;
}

void rotateRight(Node *&root){
    Node *tempPtr = root->pLeft;
    root->pLeft = tempPtr->pRight;
    tempPtr->pRight = root;
    root = tempPtr;
}

int getHeight(Node *node){
    if(node == NULL)
        return 0;
    int lh = getHeight(node->pLeft);
    int rh = getHeight(node->pRight);
    return (lh > rh ? lh : rh) + 1;
}

void balance(Node *&root){
    int balanceFactor = getHeight(root->pLeft) - getHeight(root->pRight);
    if(balanceFactor > 1){
        if(getHeight(root->pLeft->pLeft) >= getHeight(root->pLeft->pRight))
            rotateRight(root);
        else{
            rotateLeft(root->pLeft);
            rotateRight(root);
        }
    }
    else if(balanceFactor < -1){
        if(getHeight(root->pRight->pRight) >= getHeight(root->pRight->pLeft))
            rotateLeft(root);
        else{
            rotateRight(root->pRight);
            rotateLeft(root);
        }
    }
}

void insertRec(Node *&root, const T &value){
    if(root == NULL)
        root = new Node(value);
    else if(value < root->data)
        insertRec(root->pLeft, value);
    else
        insertRec(root->pRight, value);
    balance(root);
}

void insert(const T &value){
    insertRec(this->root, value);
}
```

Precheck Kiểm tra

	Test	Expected	Got	

✔	AVLTree<int> avl;
	int nums[] = {3, 1, 6, 2, 4, 8, 5, 7, 9};
	for (int i = 0; i < 9; i++){
	avl.insert(nums[i]);
	}
	avl.printTreeStructure();
		3
		1 6
		2 4 8
		5 7 9
			3
			1 6
			2 4 8
			5 7 9
				✔

✔	AVLTree<int> avl;
	int nums[] = {6, 8, 3, 5, 7, 9, 1, 2, 4};
	for (int i = 0; i < 9; i++){
	avl.insert(nums[i]);
	}
	avl.printTreeStructure();
		6
		3 8
		1 5 7 9
		2 4
			6
			3 8
			1 5 7 9
			2 4
				✔

Passed all tests! ✔

Câu hỏi 4

Chính xác

Chấm điểm của 1,00

In this question, you have to perform **add** on AVL tree. Note that:

- When adding a node which has the same value as parent node, add it in the **right sub tree**.

Your task is to implement function: **insert**. You could define one or more functions to achieve this task.

```
#include <iostream>
#include <math.h>
#include <queue>
using namespace std;
#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue
{
    LH = -1,
    EH = 0,
    RH = 1
};

void printNSpace(int n)
{
    for (int i = 0; i < n - 1; i++)
        cout << " ";
}

void printInteger(int &n)
{
    cout << n << " ";
}

template<class T>
class AVLTree
{
public:
    class Node;
private:
    Node *root;
protected:
    int getHeightRec(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh = this->getHeightRec(node->pLeft);
        int rh = this->getHeightRec(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){}
    int getHeight()
    {
        return this->getHeightRec(this->root);
    }
    void printTreeStructure()
    {
        int height = this->getHeight();
        if (this->root == NULL)
        {
            cout << "NULL\n";
            return;
        }
        queue<Node *> q;
        q.push(root);
        Node *temp;
        int count = 0;
        int maxNode = 1;
        int level = 0;
        int space = pow(2, height);
        printNSpace(space / 2);
        while (!q.empty())
        {
            temp = q.front();
            q.pop();
            if (temp == NULL)
            {
```

```
                cout << " ";
                q.push(NULL);
                q.push(NULL);
            }
            else
            {
                cout << temp->data;
                q.push(temp->pLeft);
                q.push(temp->pRight);
            }
            printNSpace(space);
            count++;
            if (count == maxNode)
            {
                cout << endl;
                count = 0;
                maxNode *= 2;
                level++;
                space /= 2;
                printNSpace(space / 2);
            }
            if (level == height)
                return;
        }
    }

    void insert(const T &value)
    {
        //TODO
    }

    class Node
    {
    private:
        T data;
        Node *pLeft, *pRight;
        BalanceValue balance;
        friend class AVLTree<T>;

    public:
        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}
        ~Node() {}
    };
};
```

For example:

Test	Result

`AVLTree<int> avl;`
`for (int i = 0; i < 9; i++){`
`  avl.insert(i);`
`}`
`avl.printTreeStructure();`
	`3`
	`1 5`
	`0 2 4 7`
	`6 8`

`AVLTree<int> avl;`
`for (int i = 10; i >= 0; i--){`
`  avl.insert(i);`
`}`
`avl.printTreeStructure();`
	`7`
	`3 9`
	`1 5 8 10`
	`0 2 4 6`

Answer: (penalty regime: 0 %)

Reset answer

```
1  // Hàm xoay trái tại một nút
2  Node* rotateLeft(Node* node) {
3      Node* temp = node->pRight; // Lưu nút con phải của nút hiện tại
4      node->pRight = temp->pLeft; // Gán nút con phải của nút hiện tại bằng nút con t
```

```
    temp->pLeft = node; // Gán nút con trái của nút temp bằng nút hiện tại
    return temp; // Trả về nút temp
}

// Hàm xoay phải tại một nút
Node* rotateRight(Node* node) {
    Node* temp = node->pLeft; // Lưu nút con trái của nút hiện tại
    node->pLeft = temp->pRight; // Gán nút con trái của nút hiện tại bằng nút con p
    temp->pRight = node; // Gán nút con phải của nút temp bằng nút hiện tại
    return temp; // Trả về nút temp
}

// Hàm cân bằng cây AVL
Node* balance(Node* node) {
    int balance = getHeightRec(node->pLeft) - getHeightRec(node->pRight); // Tính c
    if (balance > 1) { // Nếu chỉ số cân bằng > 1
        if (getHeightRec(node->pLeft->pLeft) >= getHeightRec(node->pLeft->pRight))
            node = rotateRight(node); // Xoay phải tại nút hiện tại
        else {
            node->pLeft = rotateLeft(node->pLeft); // Xoay trái tại nút con trái
            node = rotateRight(node); // Xoay phải tại nút hiện tại
        }
    }
    else if (balance < -1) { // Nếu chỉ số cân bằng < -1
        if (getHeightRec(node->pRight->pRight) >= getHeightRec(node->pRight->pLeft)
            node = rotateLeft(node); // Xoay trái tại nút hiện tại
        else {
            node->pRight = rotateRight(node->pRight); // Xoay phải tại nút con phải
            node = rotateLeft(node); // Xoay trái tại nút hiện tại
        }
    }
    return node; // Trả về nút sau khi đã cân bằng
}

// Hàm thêm một nút vào cây AVL
Node* insertRec(Node* node, const T& value) {
    if (node == NULL) // Nếu nút hiện tại rỗng
        return new Node(value); // Tạo một nút mới với giá trị cho trước
    else if (value < node->data) // Nếu giá trị nhỏ hơn giá trị tại nút hiện tại
        node->pLeft = insertRec(node->pLeft, value); // Thêm nút vào cây con trái
    else // Nếu giá trị lớn hơn hoặc bằng giá trị tại nút hiện tại
        node->pRight = insertRec(node->pRight, value); // Thêm nút vào cây con phải
    return balance(node); // Cân bằng cây AVL sau khi thêm nút
}

// Hàm thêm một giá trị vào cây AVL
void insert(const T& value) {
    root = insertRec(root, value); // Thêm giá trị vào cây AVL và cập nhật nút gốc
}
```

Precheck Kiểm tra

	Test	Expected	Got	

✔	`AVLTree<int> avl;`
	`for (int i = 0; i < 9; i++){`
	`  avl.insert(i);`
	`}`
	`avl.printTreeStructure();`
		3
		1 5
		0 2 4 7
		6 8
			3
			1 5
			0 2 4 7
			6 8
				✔

	Test	Expected	Got	

✔	AVLTree<int> avl;
	for (int i = 10; i >= 0; i--){
	\tavl.insert(i);
	}
	avl.printTreeStructure();
		7
		3 9
		1 5 8 10
		0 2 4 6
			7
			3 9
			1 5 8 10
			0 2 4 6
				✔

Passed all tests! ✔

Câu hỏi 5

Chính xác

Chấm điểm của 1,00

In this question, you have to perform **delete in AVL tree - balanced, L-L, R-L, E-L**. Note that:

- Provided **insert** function already.

Your task is to implement function: **remove** to perform re-balancing (balanced, left of left, right of left, equal of left). You could define one or more functions to achieve this task.

```
#include <iostream>
#include <math.h>
#include <queue>
using namespace std;
#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue
{
    LH = -1,
    EH = 0,
    RH = 1
};

void printNSpace(int n)
{
    for (int i = 0; i < n - 1; i++)
        cout << " ";
}

void printInteger(int &n)
{
    cout << n << " ";
}

template<class T>
class AVLTree
{
public:
    class Node;
private:
    Node *root;
protected:
    int getHeightRec(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh = this->getHeightRec(node->pLeft);
        int rh = this->getHeightRec(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){}
    int getHeight()
    {
        return this->getHeightRec(this->root);
    }
    void printTreeStructure()
    {
        int height = this->getHeight();
        if (this->root == NULL)
        {
            cout << "NULL\n";
            return;
        }
        queue<Node *> q;
        q.push(root);
        Node *temp;
        int count = 0;
        int maxNode = 1;
        int level = 0;
        int space = pow(2, height);
        printNSpace(space / 2);
        while (!q.empty())
        {
            temp = q.front();
            q.pop();
```

```
            if (temp == NULL)
            {
                cout << " ";
                q.push(NULL);
                q.push(NULL);
            }
            else
            {
                cout << temp->data;
                q.push(temp->pLeft);
                q.push(temp->pRight);
            }
            printNSpace(space);
            count++;
            if (count == maxNode)
            {
                cout << endl;
                count = 0;
                maxNode *= 2;
                level++;
                space /= 2;
                printNSpace(space / 2);
            }
            if (level == height)
                return;
        }
    }

    void remove(const T &value)
    {
        //TODO
    }

    class Node
    {
    private:
        T data;
        Node *pLeft, *pRight;
        BalanceValue balance;
        friend class AVLTree<T>;

    public:
        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}
        ~Node() {}
    };
};
```

For example:

Test	Result

`AVLTree<int> avl;`
`int arr[] = {10, 5, 15, 7};`
`for (int i = 0; i < 4; i++)`
`{`
`  avl.insert(arr[i]);`
`}`
`avl.remove(15);`
`avl.printTreeStructure();`
	7
	5 10

Test	Result

AVLTree<int> avl;
int arr[] = {10, 5, 15, 3};
for (int i = 0; i < 4; i++)
{
  avl.insert(arr[i]);
}
avl.remove(15);
avl.printTreeStructure();
	5
	3 10

Answer: (penalty regime: 0 %)

Reset answer

```
Node* rotateRight1(Node* a) {
    //TODO: Rotate and return new root after rotate
    Node *b=a->pLeft;
    Node *c=b->pRight;
    a->pLeft=c;
    b->pRight=a;
    return b;
}

Node* rotateLeft1(Node* a) {
    //TODO: Rotate and return new root after rotate
    Node *b=a->pRight;
    Node *c=b->pLeft;
    a->pRight=c;
    b->pLeft=a;
    return b;
}
int getHeightRec1(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh =getHeightRec1(node->pLeft);
        int rh =getHeightRec1(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
int getBalance1(Node*subroot){
    if(!subroot) return 0;
    return getHeightRec1(subroot->pLeft)- getHeightRec1(subroot->pRight);
}
Node * minValueNode(Node* node)
{
    Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->pRight != NULL)
        current = current->pRight;

    return current;
}
Node* deleteNode(Node* root, const T & key)
{

    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == NULL)
        return root;

    // If the key to be deleted is smaller
    // than the root's key, then it lies
    // in left subtree
    if ( key < root->data )
        root->pLeft = deleteNode(root->pLeft, key);

    // If the key to be deleted is greater
    // than the root's key, then it lies
    // in right subtree
    else if( key > root->data )
```

```
57          root->pRight = deleteNode(root->pRight, key);
58
59      // if key is same as root's key, then
60      // This is the node to be deleted
61      else
62      {
```

Precheck Kiểm tra

	Test	Expected	Got	

✔	AVLTree<int> avl;
	int arr[] = {10, 5, 15, 7};
	for (int i = 0; i < 4; i++)
	{
	  avl.insert(arr[i]);
	}
	avl.remove(15);
	avl.printTreeStructure();
		7
		5 10
			7
			5 10
				✔

✔	AVLTree<int> avl;
	int arr[] = {10, 5, 15, 3};
	for (int i = 0; i < 4; i++)
	{
	  avl.insert(arr[i]);
	}
	avl.remove(15);
	avl.printTreeStructure();
		5
		3 10
			5
			3 10
				✔

✔	AVLTree<int> avl;
	int arr[] = {10,52,98,32,68,92,40,13,42,63,99,100};
	for (int i = 0; i < 12; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(52);
	avl.printTreeStructure();
		42
		32 92
		10 40 68 99
		13 63 98 100
			42
			32 92
			10 40 68 99
			13 63 98 100
				✔

✔	AVLTree<int> avl;
	int arr[] = {20,10,40,5,7,42,2};
	for (int i = 0; i < 7; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(20);
	avl.printTreeStructure();
		10
		5 40
		2 7 42
			10
			5 40
			2 7 42
				✔

✔	AVLTree<int> avl;
	int arr[] = {20,10,40,5,7,42,2};
	for (int i = 0; i < 7; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(10);
	avl.printTreeStructure();
		20
		5 40
		2 7 42
			20
			5 40
			2 7 42
				✔

✔	AVLTree<int> avl;
	int arr[] = {20,10,40,5,7,42,2,6};
	for (int i = 0; i < 8; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(10);
	avl.printTreeStructure();
		20
		5 40
		2 7 42
		6
			20
			5 40
			2 7 42
			6
				✔

	Test	Expected	Got	

✔	AVLTree<int> avl;
	int arr[] = {20,10,40,5,7,42,2,6};
	for (int i = 0; i < 8; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(2);
	avl.remove(10);
	avl.printTreeStructure();
		20
		6 40
		5 7 42
			20
			6 40
			5 7 42
				✔

✔	AVLTree<int> avl;
	int arr[] = {20,10,40,5,7,42,2,6,15};
	for (int i = 0; i < 9; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(6);
	avl.remove(42);
	avl.printTreeStructure();
		7
		5 20
		2 10 40
		15
			7
			5 20
			2 10 40
			15
				✔

✔	AVLTree<int> avl;
	int arr[] = {20,10,40,5,7,42,2,6,15};
	for (int i = 0; i < 9; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(40);
	avl.printTreeStructure();
		7
		5 20
		2 6 10 42
		15
			7
			5 20
			2 6 10 42
			15
				✔

✔	AVLTree<int> avl;
	int arr[] = {20,10,40,5,7,42,2,6,15};
	for (int i = 0; i < 9; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(40);
	avl.remove(20);
	avl.remove(6);
	avl.remove(10);
	avl.remove(42);
	avl.remove(15);
	avl.printTreeStructure();
		5
		2 7
			5
			2 7
				✔

Passed all tests! ✔

(028) 38 651 670 - (028) 38 647 256 (Ext: 5258, 5234)

elearning@hcmut.edu.vn

Câu hỏi 6

Chính xác

Chấm điểm của 1,00

In this question, you have to perform **delete on AVL tree**. Note that:

- Provided **insert** function already.

Your task is to implement two functions: **remove**. You could define one or more functions to achieve this task.

```
#include <iostream>
#include <math.h>
#include <queue>
using namespace std;
#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue
{
    LH = -1,
    EH = 0,
    RH = 1
};

void printNSpace(int n)
{
    for (int i = 0; i < n - 1; i++)
        cout << " ";
}

void printInteger(int &n)
{
    cout << n << " ";
}

template<class T>
class AVLTree
{
public:
    class Node;
private:
    Node *root;
protected:
    int getHeightRec(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh = this->getHeightRec(node->pLeft);
        int rh = this->getHeightRec(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){}
    int getHeight()
    {
        return this->getHeightRec(this->root);
    }
    void printTreeStructure()
    {
        int height = this->getHeight();
        if (this->root == NULL)
        {
            cout << "NULL\n";
            return;
        }
        queue<Node *> q;
        q.push(root);
        Node *temp;
        int count = 0;
        int maxNode = 1;
        int level = 0;
        int space = pow(2, height);
        printNSpace(space / 2);
        while (!q.empty())
        {
            temp = q.front();
            q.pop();
            if (temp == NULL)
            {
```

```
                cout << " ";
                q.push(NULL);
                q.push(NULL);
            }
            else
            {
                cout << temp->data;
                q.push(temp->pLeft);
                q.push(temp->pRight);
            }
            printNSpace(space);
            count++;
            if (count == maxNode)
            {
                cout << endl;
                count = 0;
                maxNode *= 2;
                level++;
                space /= 2;
                printNSpace(space / 2);
            }
            if (level == height)
                return;
        }
    }

    void remove(const T &value)
    {
        //TODO
    }

    class Node
    {
    private:
        T data;
        Node *pLeft, *pRight;
        BalanceValue balance;
        friend class AVLTree<T>;

    public:
        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}
        ~Node() {}
    };
};
```

For example:

Test	Result

`AVLTree<int> avl;`
`int arr[] = {10,52,98,32,68,92,40,13,42,63};`
`for (int i = 0; i < 10; i++){`
`  avl.insert(arr[i]);`
`}`
`avl.remove(10);`
`avl.printTreeStructure();`
	`  52`
	`32  92`
	`13 40 68 98`
	`  42 63`

`AVLTree<int> avl;`
`int arr[] = {10,52,98,32,68,92,40,13,42,63,99,100};`
`for (int i = 0; i < 12; i++){`
`  avl.insert(arr[i]);`
`}`
`avl.remove(13);`
`avl.printTreeStructure();`
	`  52`
	`32  92`
	`10 40 68 99`
	`  42 63  98 100`

Answer: (penalty regime: 0 %)

Reset answer

```
Node* rotateRight1(Node* a) {
    //TODO: Rotate and return new root after rotate
    Node *b=a->pLeft;
    Node *c=b->pRight;
    a->pLeft=c;
    b->pRight=a;
    return b;
}

Node* rotateLeft1(Node* a) {
    //TODO: Rotate and return new root after rotate
    Node *b=a->pRight;
    Node *c=b->pLeft;
    a->pRight=c;
    b->pLeft=a;
    return b;
}
int getHeightRec1(Node *node)
    {
        if (node == NULL)
            return 0;
        int lh =getHeightRec1(node->pLeft);
        int rh =getHeightRec1(node->pRight);
        return (lh > rh ? lh : rh) + 1;
    }
int getBalance1(Node*subroot){
    if(!subroot) return 0;
    return getHeightRec1(subroot->pLeft)- getHeightRec1(subroot->pRight);
}
Node * minValueNode(Node* node)
{
    Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->pRight != NULL)
        current = current->pRight;

    return current;
}
Node* deleteNode(Node* root, const T & key)
{

    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == NULL)
        return root;

    // If the key to be deleted is smaller
    // than the root's key, then it lies
    // in left subtree
    if ( key < root->data )
        root->pLeft = deleteNode(root->pLeft, key);

    // If the key to be deleted is greater
    // than the root's key, then it lies
    // in right subtree
    else if( key > root->data )
        root->pRight = deleteNode(root->pRight, key);

    // if key is same as root's key, then
    // This is the node to be deleted
    else
    {
```

Precheck Kiểm tra

	Test	Expected	Got	

✔	AVLTree<int> avl;
	int arr[] = {10,52,98,32,68,92,40,13,42,63};
	for (int i = 0; i < 10; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(10);
	avl.printTreeStructure();
		52
		32 92
		13 40 68 98
		42 63
			52
			32 92
			13 40 68 98
			42 63
				✔

✔	AVLTree<int> avl;
	int arr[] = {10,52,98,32,68,92,40,13,42,63,99,100};
	for (int i = 0; i < 12; i++){
	\tavl.insert(arr[i]);
	}
	avl.remove(13);
	avl.printTreeStructure();
		52
		32 92
		10 40 68 99
		42 63 98 100
			52
			32 92
			10 40 68 99
			42 63 98 100
				✔

Passed all tests! ✔

Câu hỏi 7

Chính xác

Chấm điểm của 1,00

In this question, you have to search and print inorder on **AVL tree**. You have o implement functions: **search** and **printInorder** to complete the task. Note that:

- When the tree is null, don't print anything.

- There's a whitespace at the end when print the tree inorder in case the tree is not null.

- When tree contains value, search return true.

```
#include <iostream>
#include <queue>
using namespace std;
#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue
{
    LH = -1,
    EH = 0,
    RH = 1
};

template<class T>
class AVLTree
{
public:
    class Node;
private:
    Node *root;
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){}

    void printInorder(){
        //TODO
    }

    bool search(const T &value){
        //TODO
    }

    class Node
    {
    private:
        T data;
        Node *pLeft, *pRight;
        BalanceValue balance;
        friend class AVLTree<T>;

    public:
        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}
        ~Node() {}
    };
};
```

For example:

Test	Result

AVLTree<int> avl;
int arr[] = {10,52,98,32,68,92,40,13,42,63,99,100};
for (int i = 0; i < 12; i++){
avl.insert(arr[i]);
}
avl.printInorder();
cout << endl;
cout << avl.search(10);
	10 13 32 40 42 52 63 68 92 98 99 100
	1

Answer: (penalty regime: 0 %)

```
// Hàm in cây theo thứ tự giữa (inorder)
void printInorder(Node* node){
    // Nếu node rỗng, không làm gì cả
    if(node == nullptr)
        return;
    // In cây con bên trái
    printInorder(node->pLeft);
    // In dữ liệu của node hiện tại
    cout << node->data << " ";
    // In cây con bên phải
    printInorder(node->pRight);
}

// Hàm in cây theo thứ tự giữa (inorder)
void printInorder(){
    // Gọi hàm in cây con bắt đầu từ gốc
    printInorder(root);
}

// Hàm tìm kiếm giá trị trong cây
bool search(const T &value, Node* node){
    // Nếu node rỗng, trả về false
    if(node == nullptr)
        return false;
    // Nếu giá trị bằng dữ liệu của node, trả về true
    else if(value == node->data)
        return true;
    // Nếu giá trị nhỏ hơn dữ liệu của node, tìm kiếm trong cây con bên trái
    else if(value < node->data)
        return search(value, node->pLeft);
    // Nếu giá trị lớn hơn dữ liệu của node, tìm kiếm trong cây con bên phải
    else
        return search(value, node->pRight);
}

// Hàm tìm kiếm giá trị trong cây
bool search(const T &value){
    // Gọi hàm tìm kiếm bắt đầu từ gốc
    return search(value, root);
}

```

Precheck Kiểm tra

	Test	Expected	Got	

✔	AVLTree<int> avl;
	int arr[] =
	{10,52,98,32,68,92,40,13,42,63,99,100};
	for (int i = 0; i < 12; i++){
	\tavl.insert(arr[i]);
	}
	avl.printInorder();
	cout << endl;
	cout << avl.search(10);
		10 13 32 40 42 52 63 68 92 98
		99 100
		1
			10 13 32 40 42 52 63 68 92 98
			99 100
			1
				✔

Passed all tests! ✔