

AVL.h

```
#ifndef WORKFLOWSTUDY_AVL_H
#define WORKFLOWSTUDY_AVL_H
#include <iostream>
#include <string>
#include <vector>
#include <queue>

using namespace std;

template<class T>
struct AVLNode{
    T key;
    int height;
    struct AVLNode<T> * left;
    struct AVLNode<T> * right;
    AVLNode(T tempKey){
        key=tempKey;
        left= nullptr;
        right=nullptr;
        height=1;
    }
};

template<class Key>
inline int getHeight(AVLNode<Key> * node);

/* LL(Y rotates to the right): ( Right Rotation )
 * 根节点的左孩子的左子树添加了新节点
 *
 *      k2                      k1
 *     / \                    / \
 *    k1  Z                  X  k2
 *   / \                  / \
 *  X   Y                  Y  Z
 */
/*
 * Return which the root pointer(at a higher level) should point to
 */
template<class Key>
AVLNode<Key> * LL_Rotate(AVLNode<Key> * k2);

/* RR (Y rotates to the left): (Left Rotation)
 * 根节点的右孩子的右子树添加了新节点
 *
 *      k2                      k1
 *     / \                    / \
 *    X  k1                  k2  Z
 */
```

```

        / \
       Y  Z
    */
    template<class Key>
    AVLNode<Key> * RR_Rotate( AVLNode<Key> * k2);

    /* LR(B rotates to the left, then C rotates to the right):
    * 左孩子的右子树添加了新节点

        k3                k3                k2
        / \              / \              / \
       k1  D            k2  D            k1  k3
        / \            / \            / \ / \
       A  k2          k1  C          A  B C  D
        / \          / \
       B  C          A  B

    */
    /*
    Return which the root pointer should point to
    */
    template<typename Key>
    AVLNode<Key> * LR_Rotate(AVLNode<Key> * k3);

    /* RL(D rotates to the right, then C rotates to the left):
    * 右孩子的左子树添加了新节点

        k3                k3                k2
        / \              / \              / \
       A  k1            A  k2            k3  k1
        / \            / \            / \ / \
       k2  B          C  k1          A  C D  B
        / \          / \          / \
       C  D          D  B

    */
    template<class Key>
    AVLNode<Key> * RL_Rotate(AVLNode<Key> * k3);

    template<class Key>
    AVLNode<Key> * Insert( AVLNode<Key> * root, int key);

    template<class Key>
    AVLNode<Key> * Delete(AVLNode<Key> * root, Key key);

    //template<class Key>
    //void output_impl(AVLNode<Key> * n, bool left, std::string const& indent);
    //
    //template<class Key>
    template<class Key>

```

```

vector<vector<int>> levelOrder(AVLNode<Key> *root);

//void output(AVLNode<Key> * root);

template<class Key>
void InOrder(AVLNode<Key> * root);

template<class Key>
void PrintTree(AVLNode<Key> * root);

//#include "AVL.cpp"

#endif //WORKFLOWSTUDY_AVL_H

```

AVL.cpp

```

#include "AVL.h"

template<class Key>
inline int getHeight(AVLNode<Key> * node)
{
    return (node== nullptr )? 0:node->height;
}

/* LL(Y rotates to the right): ( Right Rotation )
 * 根节点的左孩子的左子树添加了新节点
 *
 *      k2                k1
 *     /  \              /  \
 *    k1   Z            X   k2
 *   /  \              /  \
 *  X    Y            Y    Z
 *
 */
/*
 * Return which the root pointer(at a higher level) should point to
 */
template<class Key>
AVLNode<Key> * LL_Rotate(AVLNode<Key> * k2)
{
    AVLNode<Key> * k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = std::max(getHeight(k2->right), getHeight(k2->left)) + 1;
    k1->height = std::max(getHeight(k1->left), k2->height) + 1;
    return k1;
}

/* RR (Y rotates to the left): (Left Rotation)

```

* 根节点的右孩子的右子树添加了新节点



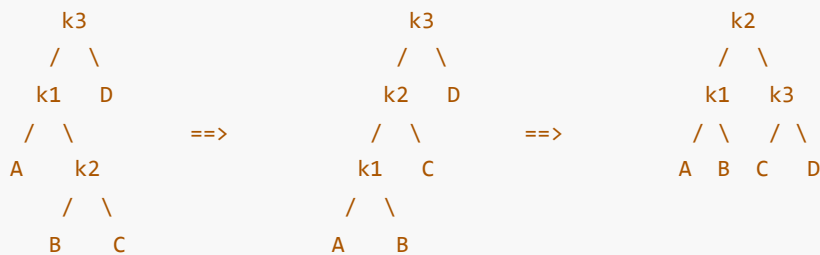
*/

```

template<class Key>
AVLNode<Key> * RR_Rotate( AVLNode<Key> * k2)
{
    AVLNode<Key>* k1 = k2->right;
    k2->right = k1->left;
    k1->left = k2;
    k2->height = std::max(getHeight(k2->left), getHeight(k2->right)) + 1;
    k1->height = std::max(getHeight(k1->right), k2->height) + 1;
    return k1;
}
  
```

/* LR(B rotates to the left, then C rotates to the right):

* 左孩子的右子树添加了新节点



*/

/*

Return which the root pointer should point to

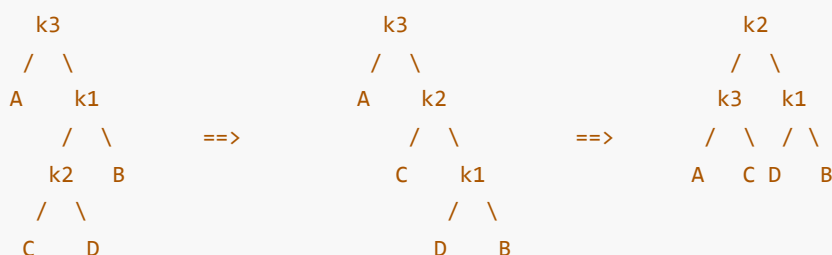
*/

```

template<typename Key>
AVLNode<Key> * LR_Rotate(AVLNode<Key> * k3)
{
    k3->left = RR_Rotate(k3->left);
    return LL_Rotate(k3);
}
  
```

/* RL(D rotates to the right, then C rotates to the left):

* 右孩子的左子树添加了新节点



```

*/
template<class Key>
AVLNode<Key> * RL_Rotate(AVLNode<Key> * k3)
{
    k3->right = LL_Rotate(k3->right);
    return RR_Rotate(k3);
}

template<class Key>
AVLNode<Key> * Insert(AVLNode<Key> * root, int key)
{
    if(root == nullptr)
    {
        root=new AVLNode<Key>(key);
        return root;
    }
    else if(key < root->key)
        root->left = Insert(root->left, key);
    else //key >= root->key
        root->right = Insert(root->right, key);

    root->height = std::max(getHeight(root->left), getHeight(root->right)) + 1;

    if(getHeight(root->left) - getHeight(root->right) == 2)
    {
        // 操作root的左孩子
        if(key < root->left->key)
            // 左孩子的左子树添加了节点 LL
            root = LL_Rotate(root);
        else
            // 左孩子的右子树添加了节点 LR
            root = LR_Rotate(root);
    }
    else if(getHeight(root->right) - getHeight(root->left) == 2)
    {
        // 操作root的右孩子
        if(key < root->right->key)
            // 右孩子的左子树添加了节点 RL
            root = RL_Rotate(root);
        else
            // 操作右孩子的右子树添加了节点 RR
            root = RR_Rotate(root);
    }
    return root;
}

// delete 就是BST的Delete
template<class Key>
AVLNode<Key> * Delete(AVLNode<Key> * root, Key key)
{

```

```

if(root== nullptr)
    return nullptr;
if(key == root->key)
{
    if(root->right == nullptr)
    {
        AVLNode<Key> * temp = root;
        root = root->left;
        delete(temp);
        // delete root 后 return root->left
        return root;

    }else if(root->left== nullptr){
        AVLNode<Key> * temp = root;
        root = root->right;
        delete(temp);
        // delete root 后 return root->right
        return root;
    }
    else{
        // root's left and right all exists

        // 找右子树的最左节点 (右子树的最小节点)
        AVLNode<Key> * temp = root->right;
        while(temp->left!= nullptr) temp = temp->left;

        /* replace the value */
        root->key = temp->key;
        /* Delete the node (successor node) that should be really deleted */
        // 改为删除右子树的最小节点
        root->right = Delete(root->right, temp->key);
    }
}
else if(key < root->key)
    root->left = Delete(root->left, key);
else
    root->right = Delete(root->right, key);

return root;
}

```

```

template<class Key>
vector<vector<int>> levelOrder(AVLNode<Key> *root) {
    vector<vector<int>> res;
    if (root == nullptr) {
        return res;
    }

    queue<AVLNode<Key> *> queueTreeNodees;
    queueTreeNodees.push(root);
}

```

```

while (!queueTreeNodes.empty()) {

    int sz = queueTreeNodes.size();
    vector<int> cens;
    for (int i = 0; i < sz; ++i) {
        AVLNode<Key> *node = queueTreeNodes.front();
        queueTreeNodes.pop();
        cens.push_back(node->key);
        if (node->left != nullptr) {
            queueTreeNodes.push(node->left);
        }
        if (node->right != nullptr) {
            queueTreeNodes.push(node->right);
        }
    }
    res.push_back(cens);

}

return res;

}

template<class Key>
void InOrder(AVLNode<Key> * root)
{
    if(root)
    {
        InOrder(root->left);
        printf("key: %d height: %d ", root->key, root->height);
        if(root->left)
            printf("left child: %d ", root->left->key);
        if(root->right)
            printf("right child: %d ", root->right->key);
        printf("\n");
        InOrder(root->right);
    }
}

template<class Key>
void PrintTree(AVLNode<Key> * root){
    cout << "#####" <<endl;
    auto res= levelOrder(root);
    for (auto c:res) {
        for (auto b :c) {
            cout << "\t"<< b<< "\t";
        }
        cout <<endl;
    }
}

```

```
cout << "#####" << endl;  
  
}
```