AVL树：带有平衡条件的二叉查找树。平衡主要是保证二叉查找树的操作（某些可能的插入操作除外，当插入操作破坏了平衡条件）的时间为。

<https://blog.csdn.net/u014665013/article/details/78335433>

AVL树的平衡条件：每个结点的左子树和右子树的高度最多差1.

树的高度：根结点到一片树叶的最长路径的长。

空树（没有任何结点，包括根结点的树）的高度定义为-1。主要是为了方便计算只有一个根结点的树的高度为0。



结点的左子树高度：从结点的左子树计算的结点的高度。

结点的右子树高度：从结点的右子树计算的结点的高度。



结点A的左子树高度为3，右子树高度为2。

结点B的左子树高度为2，右子树高度为1.

结点G的左子树高度为0，右子树高度为0.

结点E的左子树高度为0，右子树高度为1.

树的旋转：AVL树执行了插入操作后，有可能会破坏平衡条件，这时需要对树进行修正，以使其能够重新满足平衡条件。

AVL树插入某个结点导致树的平衡性被破坏，只有插入结点到根结点的路径上的结点的平衡条件可能会被破坏。应该在第1个需要重新平衡的结点（最深的结点，可以理解为离根结点最远的结点）处来平衡AVL树，将这个结点记为α结点，α结点的左右子树的高度差肯定为2。α结点的不平衡总共有以下4种可能引起：

1. 对α的左儿子的左子树进行了一次插入（右单旋转重新平衡）。
2. 对α的右儿子的右子树进行了一次插入（左单旋转重新平衡）。
3. 对α的左儿子的右子树进行了一次插入（先左旋后右旋重新平衡）。
4. 对α的右儿子的左子树进行了一次插入（先右旋后左旋重新平衡）。

例：程序avl\_tree\_test1

// Copyright 2018.刘珅珅

// author：刘珅珅

// AVL自平衡二叉树

#include <algorithm>

#include <iostream>

//#include <functional>

using namespace std;

template<typename T, typename Comparator = less<T>>

class AVLTree

{

public:

AVLTree() : root\_(nullptr)

{

}

~AVLTree()

{

Destroy(root\_);

}

public:

int Height() const // 树的高度

{

return Height(root\_);

}

bool Contains(T value) const

{

return Contains(value, root\_);

}

void Insert(T value)

{

Insert(value, root\_);

}

void Remove(T value)

{

Remove(value, root\_);

}

void PrintTree() const

{

PrintTree(root\_);

}

void PrePrintTree() const

{

PrePrintTree(root\_);

}

void PostPrintTree() const

{

PostPrintTree(root\_);

}

private:

struct AVLTreeNode

{

AVLTreeNode(T value, AVLTreeNode\* left, AVLTreeNode\* right)

: value\_(value), left\_child\_(left), right\_child\_(right)

{

}

T value\_;

int height\_; // 节点高度

AVLTreeNode\* left\_child\_;

AVLTreeNode\* right\_child\_;

};

// 结点的高度

int Height(AVLTreeNode\* node)

{

if (node != nullptr)

{

return node->height\_;

}

return -1; // 空树的高度为-1

}

// 左旋转

// root为最小失衡子树的根结点

// 即导致AVL失衡的插入结点与根结点

// 路径上第1个失衡的结点

// 返回旋转后的子树的根结点

AVLTreeNode\* LeftRotation(AVLTreeNode\* root)

{

cout << "向左旋转" << endl;

// 左旋转

// 将失衡子树的根结点的右孩子作为新的根结点

// 原根结点作为新根结点的左孩子

// 类似于将整个失衡子树向左旋转了一格

AVLTreeNode\* right\_child = root->right\_child\_;

// 新根结点right\_child的左孩子为原根结点的右孩子

// 新根结点的左孩子的值肯定比原根结点的值大

root->right\_child\_ = right\_child->left\_child\_;

right\_child->left\_child\_ = root;

// 参考C++算法学习8-AVL树.docx中结点的高度计算

root->height\_ = max(Height(root->left\_child\_), Height(root->right\_child\_)) + 1;

right\_child->height\_ = max(Height(right\_child->left\_child\_), Height(right\_child->right\_child\_)) + 1;

return right\_child;

}

// 右旋转

// root为最小失衡子树的根结点

// 返回旋转后的子树的根结点

AVLTreeNode\* RightRotation(AVLTreeNode\* root)

{

cout << "向右旋转" << endl;

// 右旋转

// 将失衡子树的根结点的左孩子作为新的根结点

// 原根结点作为新根结点的右孩子

// 类似于将整个失衡子树向右旋转了一格

AVLTreeNode\* left\_child = root->left\_child\_;

// 新根结点left\_child的右孩子的值肯定比原根结点的值小

root->left\_child\_ = left\_child->right\_child\_;

left\_child->right\_child\_ = root;

root->height\_ = max(Height(root->left\_child\_), Height(root->right\_child\_)) + 1;

left\_child->height\_ = max(Height(left\_child->left\_child\_), Height(left\_child->right\_child\_)) + 1;

return left\_child;

}

// 先左旋后右旋

AVLTreeNode\* LeftRightRotation(AVLTreeNode\* root)

{

cout << "先左旋后右旋" << endl;

// 先左旋后右旋是为了解决

// 左子树插入右孩子导致的失衡

// root为失衡子树的根结点

// 先对root的左子树进行左旋，

// 是为了将root子树变成左子树插入左孩子的问题

root->left\_child\_ = LeftRotation(root->left\_child\_);

return RightRotation(root);

}

// 先右旋后左旋

// root为失衡子树的根结点

// 在右子树上插入了左孩子导致失衡

AVLTreeNode\* RightLeftRotation(AVLTreeNode\* root)

{

cout << "先右旋后左旋" << endl;

// 先将失衡子树的根结点的右子树进行右旋转

// 将整个失衡子树变成类似右子树插入右孩子导致失衡的情形

// 然后再用左旋转解决失衡问题

root->right\_child\_ = RightRotation(root->right\_child\_);

return LeftRotation(root);

}

bool Contains(T value, AVLTreeNode\* node) const

{

while (node != nullptr)

{

if (node->value\_ == value)

{

return true;

}

else if (node->value\_ < value)

{

node = node->right\_child\_;

}

else

{

node = node->left\_child\_;

}

}

return false;

}

// 插入指定的元素

AVLTreeNode\* Insert(T value, AVLTreeNode\*& node)

{

if (!Contains(value, node))

{

if (node == nullptr)

{

node = new AVLTreeNode(value, nullptr, nullptr);

}

else if (IsLessThan(node->value\_, value)) // 插入右子树

{

node->right\_child\_ = Insert(value, node->right\_child\_);

// AVL树失衡

if (Height(node->right\_child\_) - Height(node->left\_child\_) == 2)

{

// 右子树插入右孩子

if (IsLessThan(node->right\_child\_->value\_, value))

{

node = LeftRotation(node);

}

else if (IsLessThan(value, node->right\_child\_->value\_))

{

// 右子树插入左孩子

node = RightLeftRotation(node);

}

}

}

else if (IsLessThan(value, node->value\_)) // 插入左子树

{

node->left\_child\_ = Insert(value, node->left\_child\_);

// AVL树失衡

if (Height(node->left\_child\_) - Height(node->right\_child\_) == 2)

{

// 左子树插入左孩子

if (IsLessThan(value, node->left\_child\_->value\_))

{

node = RightRotation(node);

}

else if (IsLessThan(node->left\_child\_->value\_, value))

{

// 左子树插入右孩子

node = LeftRightRotation(node);

}

}

}

// 更新结点的高度

node->height\_ = max(Height(node->left\_child\_), Height(node->right\_child\_)) + 1;

return node;

}

return nullptr;

}

// 删除指定的元素

AVLTreeNode\* Remove(T value, AVLTreeNode\*& node)

{

if (node != nullptr)

{

if (node->value\_ == value) // 找到删除结点

{

// AVL树是二叉查找树

// 左右结点均不为空

if (node->right\_child\_ != nullptr && node->left\_child\_ != nullptr)

{

if (Height(node->left\_child\_) > Height(node->right\_child\_))

{

// 左子树比右子树高

// 查找左子树的最大结点

// 用其值代替node根结点的值

// 然后删除左子树的最大结点

AVLTreeNode\* left\_max\_node = FindMax(node->left\_child\_);

node->value\_ = left\_max\_node->value\_;

node->left\_child\_ = Remove(left\_max\_node->value\_, node->left\_child\_);

}

else

{

// 用右子树的最小结点值代替根结点的值

// 然后删除右子树的最小结点

AVLTreeNode\* right\_min\_node = FindMin(node->right\_child\_);

node->value\_ = right\_min\_node->value\_;

node->right\_child\_ = Remove(value, node->right\_child\_);

}

}

else

{

AVLTreeNode\* temp = node;

if (node->left\_child\_ != nullptr)

{

node = node->left\_child\_;

}

else if (node->right\_child\_ != nullptr)

{

node = node->right\_child\_;

}

delete temp;

return nullptr;

}

}

else if (node->value\_ < value)

{

// 要删除的结点比当前结点大

// 在右子树进行删除

node->right\_child\_ = Remove(value, node->right\_child\_);

// 删除右子树结点导致不平衡

// 左子树的高度大于右子树的高度

if (Height(node->left\_child\_) - Height(node->right\_child\_) == 2)

{

// 相当于在左子树上插入右孩子

// node左孩子的右子树高度大于左子树

if (Height(node->left\_child\_->right\_child\_) >

Height(node->left\_child\_->left\_child\_))

{

node = LeftRightRotation(node);

}

else

{

// 相当于在左子树上插入左孩子

node = RightRotation(node);

}

}

}

else if (node->value\_ > value)

{

// 要删除的结点比当前结点小

node->left\_child\_ = Remove(value, node->left\_child\_);

// 删除左子树导致的不平衡

// 右子树的高度大于左子树

if (Height(node->right\_child\_) - Height(node->left\_child\_) == 2)

{

// node右孩子的左子树的高度大于右子树的高度

// 相当于node右子树上插入左孩子

if (Height(node->right\_child\_->left\_child\_) >

Height(node->right\_child\_->right\_child\_))

{

node = RightLeftRotation(node);

}

else

{

// 相当于node右子树上插入右孩子

node = LeftRotation(node);

}

}

}

return node;

}

return nullptr;

}

AVLTreeNode\* FindMin(AVLTreeNode\* node) const

{

if (node != nullptr)

{

while (node->left\_child\_ != nullptr)

{

node = node->left\_child\_;

}

return node;

}

return nullptr;

}

AVLTreeNode\* FindMax(AVLTreeNode\* node) const

{

if (node != nullptr)

{

while (node->right\_child\_ != nullptr)

{

node = node->right\_child\_;

}

return node;

}

return nullptr;

}

// 销毁AVL树

void Destroy(AVLTreeNode\* node)

{

if (node != nullptr)

{

Destroy(node->left\_child\_);

Destroy(node->right\_child\_);

delete node;

node = nullptr;

}

}

// 中序遍历

void PrintTree(AVLTreeNode\* node) const

{

if (node == nullptr)

{

return;

}

PrintTree(node->left\_child\_);

cout << node->value\_ << ' ';

PrintTree(node->right\_child\_);

}

// 前序遍历

void PrePrintTree(AVLTreeNode\* node) const

{

if (node != nullptr)

{

cout << node->value\_ << ' ';

PrePrintTree(node->left\_child\_);

PrePrintTree(node->right\_child\_);

}

}

// 后序遍历

void PostPrintTree(AVLTreeNode\* node) const

{

if (node == nullptr)

{

return;

}

PostPrintTree(node->left\_child\_);

PostPrintTree(node->right\_child\_);

cout << node->value\_ << ' ';

}

private:

AVLTreeNode\* root\_;

Comparator IsLessThan;

};

int main(int argc, char\* argv[])

{

AVLTree<int> avl\_tree;

avl\_tree.Insert(5);

avl\_tree.Insert(3);

avl\_tree.Insert(7);

avl\_tree.Insert(2);

avl\_tree.Insert(4);

avl\_tree.Insert(6);

avl\_tree.Insert(8);

avl\_tree.Insert(0);

avl\_tree.Insert(1);

avl\_tree.Remove(2);

avl\_tree.PrintTree();

cout << endl;

avl\_tree.PrePrintTree();

cout << endl;

avl\_tree.PostPrintTree();

cout << endl;

return 0;

}