# 第六次作业

平衡二叉树的插入、遍历、删除操作

## 思路（各模块功能设计）

我通过设置一个AVL树类，然后在类中设置插入、删除、查找、遍历等函数，然后在主函数中调用这些函数，实现平衡二叉树的插入、删除、查找等操作。

### 插入操作

1.首先判断树是否为空，如果为空，新建结点，数据为插入的数据，高度为0，左孩子和右孩子为空。  
2.如果插入的数据小于结点的数据，递归插入左孩子，如果左孩子的高度减去右孩子的高度等于2，如果插入的数据小于左孩子的数据，左单旋转，否则左双旋转。  
3.如果插入的数据大于结点的数据，递归插入右孩子，如果右孩子的高度减去左孩子的高度等于2，如果插入的数据大于右孩子的数据，右单旋转，否则右双旋转。  
4.更新结点的高度。

### 删除操作

1.首先判断树是否为空，如果为空，返回空。  
2.如果删除的数据小于结点的数据，递归删除左孩子，如果右孩子的高度减去左孩子的高度等于2，如果右孩子的左孩子的高度大于右孩子的右孩子的高度，右双旋转，否则右单旋转。  
3.如果删除的数据大于结点的数据，递归删除右孩子，如果左孩子的高度减去右孩子的高度等于2，如果左孩子的右孩子的高度大于左孩子的左孩子的高度，左双旋转，否则左单旋转。  
4.如果左孩子和右孩子都不为空，如果左孩子的高度大于右孩子的高度，查找左孩子的最大值，结点的数据等于最大值的数据，递归删除左孩子，否则查找右孩子的最小值，结点的数据等于最小值的数据，递归删除右孩子。  
5.如果左孩子和右孩子有一个为空，如果左孩子不为空，结点等于左孩子，否则等于右孩子。  
6.更新结点的高度。

### 查找操作

1.首先判断树是否为空，如果为空，返回false。  
2.如果查找的数据小于结点的数据，递归查找左孩子，如果查找的数据大于结点的数据，递归查找右孩子，否则返回true。

### 遍历操作

1.中序遍历：递归遍历左孩子，输出结点的数据，递归遍历右孩子。

## 代码

#include<bits/stdc++.h>  
using namespace std;  
  
// 设置一个AVL类  
class AVLtree {  
 // 私有成员属性  
private:  
 // AVL树的结点  
 struct AVLnode {  
 int data; // 数据  
 int height; // 高度  
 AVLnode\* left; // 左孩子  
 AVLnode\* right; // 右孩子  
 };  
 AVLnode\* root; // 根结点  
  
 // 私有成员函数  
 // 获取结点的高度  
 int height(AVLnode\* node) {  
 return node == NULL ? -1 : node->height;  
 }  
  
 // 左单旋转  
 AVLnode\* singleRotateLeft(AVLnode\* k2) {  
 AVLnode\* k1 = k2->left;//k1是k2的左孩子  
 k2->left = k1->right;//k2的左孩子是k1的右孩子  
 k1->right = k2;//k1的右孩子是k2  
 k2->height = max(height(k2->left), height(k2->right)) + 1;//更新k2的高度  
 k1->height = max(height(k1->left), k2->height) + 1;//更新k1的高度  
 return k1;  
 }  
  
 // 右单旋转  
 AVLnode\* singleRotateRight(AVLnode\* k1) {  
 AVLnode\* k2 = k1->right;//k2是k1的右孩子  
 k1->right = k2->left;//k1的右孩子是k2的左孩子  
 k2->left = k1;//k2的左孩子是k1  
 k1->height = max(height(k1->left), height(k1->right)) + 1;//更新k1的高度  
 k2->height = max(height(k2->right), k1->height) + 1;//更新k2的高度  
 return k2;  
 }  
  
 // 左双旋转  
 AVLnode\* doubleRotateLeft(AVLnode\* k3) {  
 k3->left = singleRotateRight(k3->left);//k3的左孩子是k3的左孩子右孩子的右孩子  
 return singleRotateLeft(k3);//返回k3的左孩子  
 }  
  
 // 右双旋转  
 AVLnode\* doubleRotateRight(AVLnode\* k1) {  
 k1->right = singleRotateLeft(k1->right);//k1的右孩子是k1的右孩子左孩子的左孩子  
 return singleRotateRight(k1);//返回k1的右孩子  
 }  
  
 // 查找最大值  
 AVLnode\* findMax(AVLnode\* node) {  
 if (node == NULL) {//如果结点为空，返回空  
 return NULL;  
 }  
 if (node->right == NULL) {//如果结点的右孩子为空，返回结点  
 return node;  
 }  
 return findMax(node->right);//递归查找右孩子  
 }  
  
 // 查找最大值  
 AVLnode\* findMin(AVLnode\* node) {  
 if (node == NULL) {//如果结点为空，返回空  
 return NULL;  
 }  
 if (node->left == NULL) {//如果结点的左孩子为空，返回结点  
 return node;  
 }  
 return findMin(node->left);//递归查找左孩子  
 }  
  
public:  
 // 构造函数  
 AVLtree() {  
 root = NULL;  
 }  
  
 // 设置函数进行插入  
 void insert(int data) {  
 root = insert(root, data);//插入结点  
 }  
  
 // 设置函数进行删除  
 void remove(int data) {  
 root = remove(root, data);  
 }  
  
 // 设置函数进行查找  
 bool search(int data) {  
 return search(root, data);  
 }  
  
 AVLnode\* insert(AVLnode\*& node, int data) {  
 if (node == NULL) {//如果结点为空，新建结点  
 node = new AVLnode;  
 node->data = data;  
 node->height = 0;  
 node->left = NULL;  
 node->right = NULL;  
 }  
 else if (data < node->data) {//如果插入的数据小于结点的数据  
 insert(node->left, data);//递归插入左孩子  
 if (height(node->left) - height(node->right) == 2) {//如果左孩子的高度减去右孩子的高度等于2  
 if (data < node->left->data) {//如果插入的数据小于左孩子的数据  
 node = singleRotateLeft(node);//左单旋转  
 }  
 else {//否则  
 node = doubleRotateLeft(node);  
 }  
 }  
 }  
 else if (data > node->data) {  
 insert(node->right, data);//递归插入右孩子  
 if (height(node->right) - height(node->left) == 2) {// 如果右孩子的高度减去左孩子的高度等于2  
 if (data > node->right->data) {// 如果插入的数据大于右孩子的数据  
 node = singleRotateRight(node);//右单旋转  
 }  
 else {  
 node = doubleRotateRight(node);//右双旋转  
 }  
 }  
 }  
 node->height = max(height(node->left), height(node->right)) + 1;//更新结点的高度  
 return node;  
 }  
  
 AVLnode\* remove(AVLnode\*& node, int data) {//删除结点  
 if (node == NULL) {  
 return NULL;  
 }  
 if (data < node->data) {//如果删除的数据小于结点的数据  
 remove(node->left, data);//递归删除左孩子  
 if (height(node->right) - height(node->left) == 2) {//如果右孩子的高度减去左孩子的高度等于2  
 if (height(node->right->left) > height(node->right->right)) {//如果右孩子的左孩子的高度大于右孩子的右孩子的高度  
 node = doubleRotateRight(node);//右双旋转  
 }  
 else {  
 node = singleRotateRight(node);//右单旋转  
 }  
 }  
 }  
 else if (data > node->data) {//如果删除的数据大于结点的数据  
 remove(node->right, data);//递归删除右孩子  
 if (height(node->left) - height(node->right) == 2) {//如果左孩子的高度减去右孩子的高度等于2  
 if (height(node->left->right) > height(node->left->left)) {//如果左孩子的右孩子的高度大于左孩子的左孩子的高度  
 node = doubleRotateLeft(node);//左双旋转  
 }  
 else {//否则  
 node = singleRotateLeft(node);//左单旋转  
 }  
 }  
 }  
 else {  
 if (node->left != NULL && node->right != NULL) {//如果左孩子和右孩子都不为空  
 if (height(node->left) > height(node->right)) {//如果左孩子的高度大于右孩子的高度  
 AVLnode\* max = findMax(node->left);//查找左孩子的最大值  
 node->data = max->data;//结点的数据等于最大值的数据  
 remove(node->left, max->data);//递归删除左孩子  
 }  
 else {  
 AVLnode\* min = findMin(node->right);//查找右孩子的最小值  
 node->data = min->data;//结点的数据等于最小值的数据  
 remove(node->right, min->data);//递归删除右孩子  
 }  
 }  
 else {  
 AVLnode\* old = node;  
 node = (node->left != NULL) ? node->left : node->right;//如果左孩子不为空，结点等于左孩子，否则等于右孩子  
 delete old;  
 }  
 }  
 if (node != NULL) {  
 node->height = max(height(node->left), height(node->right)) + 1;//更新结点的高度  
 }  
 return node;  
 }  
  
 bool search(AVLnode\* node, int data) {  
 if (node == NULL) {  
 return false;  
 }  
 if (data < node->data) {  
 return search(node->left, data);//递归查找左孩子  
 }  
 else if (data > node->data) {  
 return search(node->right, data);//递归查找右孩子  
 }  
 else {  
 return true;  
 }  
 }  
  
 // 中序遍历  
 void inorder() {  
 inorder(root);  
 }  
  
 static void inorder(AVLnode\* node) {  
 if (node == NULL) {  
 return;  
 }  
 inorder(node->left);  
 cout << node->data << " ";  
 inorder(node->right);  
 }  
  
 // 层序遍历  
 void levelorder() {  
 levelorder(root);  
 }  
  
 static void levelorder(AVLnode\* node) {  
 queue<AVLnode\*> q;  
 q.push(node);  
 while (!q.empty()) {  
 AVLnode\* temp = q.front();  
 q.pop();  
 cout << temp->data << " ";  
 if (temp->left != NULL) {  
 q.push(temp->left);  
 }  
 if (temp->right != NULL) {  
 q.push(temp->right);  
 }  
 }  
 }  
 //把这个树按照树的形式打印出来  
 void print\_tree()  
 {  
 print\_tree(root,0);  
 }  
 void print\_tree(AVLnode\* node,int depth)//递归打印树  
 {  
 if(node==NULL)  
 {  
 return;  
 }  
 print\_tree(node->right,depth+1);//先打印右子树  
 for(int i=0;i<depth;i++)//打印空格  
 {  
 cout<<" ";//打印空格  
 }  
 cout<<node->data<<endl;//打印结点  
 print\_tree(node->left,depth+1);//打印左子树  
 }  
};  
  
int main() {  
 AVLtree tree;  
 tree.insert(1);  
 tree.insert(2);  
 tree.insert(3);  
 tree.insert(4);  
 tree.insert(5);  
 tree.insert(6);  
 tree.insert(7);  
  
 // 遍历  
 tree.inorder();  
 cout << endl;  
  
 // 层序遍历  
 tree.levelorder();  
 cout << endl;  
  
 // 删除  
 tree.remove(3);  
  
 // 遍历  
 tree.inorder();  
 cout << endl;  
  
 // 层序遍历  
 tree.levelorder();  
 cout << endl;  
   
 //打印树  
 tree.print\_tree();  
 // 查找  
 cout << tree.search(5) << endl;  
  
 return 0;  
}

## 输出结果

