### 第20题：平衡二叉树的演示

#### 数据结构：

采用二叉树的链式存储结构。下面是二叉树结点（node）的结构体

struct Node {

int key;

int value;

int height;

Node \*left;

Node \*right;

// 构造函数

Node(int k, int v) : key(k), value(v), height(1), left(nullptr), right(nullptr) {}

};

输出数据时采用层序遍历，使用到了STL中的队列（queue）

源文件里有n行，每行有两个整数，用一个空格间隔，第一个数字为关键字值，第二个为该关键字所对应的值value

#### 算法设计思想：

在主函数里获取源文件里的键值对并建立AVL树，打印输出菜单，并根据用户输入的指令进行相应操作,即查找，插入，删除，输出整个AVL树，退出程序以及提示输入指令错误。并调用相关函数，

每个操作函数又由几个基础的对树操作的基础函数组成;

1. 获取节点的高度函数 int getHeight(Node \*node)
2. 取节点的平衡因子函数 int getBalanceFactor(Node \*node)
3. 更新节点的高度函数 void updateHeight(Node \*node)
4. 左旋转函数 Node\* leftRotate(Node \*node)
5. 右旋转函数 Node\* rightRotate(Node \*node)
6. 寻找右子树最小结点函数 Node\* findMin(Node \*node)

对于AVL树的三个基础操作函数查找，插入及删除函数如下

1. 查找函数Node\* find(Node \*node, int key)：

采用递归的方法，若要查找的关键字比当前结点的关键字大，则找寻当前结点的右子树；若小，则找寻左子树。直至找到关键字值相同的结点或者递归至叶子结点仍未找到关键字值相同的结点

1. 插入函数Node\* insert(Node \*node, int key, int value)

同样采用递归的方法，按照插入二叉查找树结点的方法插入结点，如果key已经存在，则更新节点的值。然后更新节点高度并计算平衡因子，调用相关函数。

如果节点的平衡因子大于1，则需要进行旋转操作。其中如果新插入的节点在节点的左子树的左侧，则进行右旋；如果新插入的节点在节点的左子树的右侧，则先进行左旋再进行右旋。

如果节点的平衡因子小于-1，则需要进行旋转操作。其中如果新插入的节点在节点的右子树的右侧，则进行左旋；如果新插入的节点在节点的右子树的左侧，则先进行右旋再进行左旋。

1. 删除函数Node\* remove(Node \*node, int key)

查找到对应节点，如果节点无子树，删除该结点即可。如果节点仅有一个子树，对应节点的位置由它的孩子结点代替。如果节点既有左子树又有右子树，则找到右子树中的最小节点来替换对应节点。然后更新节点的值。然后更新节点高度并计算平衡因子，调用相关函数。

1. 建立AVL树，即读取文件使用插入函数直至读取文件完毕。
2. 输出AVL树函数，void printTree(Node \*node)：

使用层序遍历输出对应结点的key值和左右孩子结点key值。

1. 将修改后的树覆盖写入源文件函数void printTreeFile(Node\* root,ofstream &fout)

用户决定执行退出操作时调用。

#### 源程序：

#include <iostream>

#include <fstream>

#include <stdlib.h>

#include <iomanip>

#include <queue>

using namespace std;

// 定义AVL树节点

struct Node {

int key;

int value;

int height;

Node \*left;

Node \*right;

// 构造函数

Node(int k, int v) : key(k), value(v), height(1), left(nullptr), right(nullptr) {}

};

// 获取节点的高度

int getHeight(Node \*node) {

if (node == nullptr) return 0;

return node->height;

}

// 获取节点的平衡因子

int getBalanceFactor(Node \*node) {

if (node == nullptr) return 0;

return getHeight(node->left) - getHeight(node->right);

}

// 更新节点的高度

void updateHeight(Node \*node) {

node->height = max(getHeight(node->left), getHeight(node->right)) + 1;

}

// 左旋

Node\* leftRotate(Node \*node) {

Node \*temp = node->right;

node->right = temp->left;

temp->left = node;

updateHeight(node);

updateHeight(temp);

return temp;

}

// 右旋

Node\* rightRotate(Node \*node) {

Node \*temp = node->left;

node->left = temp->right;

temp->right = node;

updateHeight(node);

updateHeight(temp);

return temp;

}

Node\* findMin(Node \*node) {

if (node == nullptr || node->left == nullptr) return node;

return findMin(node->left);

}

// 在AVL树中插入新节点

Node\* insert(Node \*node, int key, int value) {

if (node == nullptr) return new Node(key, value);

if (key < node->key) {

node->left = insert(node->left, key, value);

} else if (key > node->key) {

node->right = insert(node->right, key, value);

} else {

// 如果key已经存在，则更新节点的值

node->value = value;

return node;

}

// 更新节点的高度

updateHeight(node);

// 计算节点的平衡因子

int balanceFactor = getBalanceFactor(node);

// 如果节点的平衡因子大于1，则需要进行旋转操作

if (balanceFactor > 1) {

// 如果新插入的节点在节点的左子树的左侧，则进行右旋

if (key < node->left->key) {

return rightRotate(node);

}

// 如果新插入的节点在节点的左子树的右侧，则先进行左旋再进行右旋

if (key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

}

// 如果节点的平衡因子小于-1，则需要进行旋转操作

if (balanceFactor < -1) {

// 如果新插入的节点在节点的右子树的右侧，则进行左旋

if (key > node->right->key) {

return leftRotate(node);

}

// 如果新插入的节点在节点的右子树的左侧，则先进行右旋再进行左旋

if (key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

}

return node;

}

// 在AVL树中查找节点

Node\* find(Node \*node, int key) {

if (node == nullptr) return nullptr;

if (key < node->key) {

return find(node->left, key);

} else if (key > node->key) {

return find(node->right, key);

} else {

return node;

}

}

// 在AVL树中删除节点

Node\* remove(Node \*node, int key) {

if (node == nullptr) return nullptr;

if (key < node->key) {

node->left = remove(node->left, key);

} else if (key > node->key) {

node->right = remove(node->right, key);

} else {

if (node->left == nullptr && node->right == nullptr) {

delete node;

return nullptr;

} else if (node->left == nullptr) {

Node \*temp = node->right;

delete node;

return temp;

}

else if (node->right == nullptr) {

Node \*temp = node->left;

delete node;

return temp;

} else {

// 如果节点既有左子树又有右子树，则找到右子树中的最小节点来替换当前节点

Node \*successor = findMin(node->right);

node->key = successor->key;

node->value = successor->value;

node->right = remove(node->right, successor->key);

}

}

// 更新节点的高度

updateHeight(node);

// 计算节点的平衡因子

int balanceFactor = getBalanceFactor(node);

// 如果节点的平衡因子大于1，则需要进行旋转操作

if (balanceFactor > 1) {

// 如果节点的左子树的平衡因子大于0，则进行右旋

if (getBalanceFactor(node->left) > 0) {

return rightRotate(node);

}

// 如果节点的左子树的平衡因子小于0，则先进行左旋再进行右旋

if (getBalanceFactor(node->left) < 0) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

}

// 如果节点的平衡因子小于-1，则需要进行旋转操作

if (balanceFactor < -1) {

// 如果节点的右子树的平衡因子小于0，则进行左旋

if (getBalanceFactor(node->right) < 0) {

return leftRotate(node);

}

// 如果节点的右子树的平衡因子大于0，则先进行右旋再进行左旋

if (getBalanceFactor(node->right) > 0) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

}

return node;

}

//打印二叉树

void printTree(Node \*node) {

if (node == nullptr) return;

queue<Node \*> q;

q.push(node);

Node \*temp;

cout<<"key lchild rchild"<<endl;

while (!q.empty()){

temp = q.front();

q.pop();

cout << std::left << setw(4)<<temp->key;

if (temp->left) {

q.push(temp->left);

cout<< std::left << setw(7)<<temp->left->key;

}else {

cout<< std::left << setw(7)<<" ";

}

if (temp->right) {

q.push(temp->right);

cout<< std::left << setw(7)<<temp->right->key<<endl;;

}else {

cout<<endl;

}

}

cout << endl;

}

//存文件

void printTreeFile(Node\* root,ofstream &fout) {

if (root == nullptr) {

return;

}

fout << root->key<<" "<<root->value << endl;

printTreeFile(root->left,fout);

printTreeFile(root->right,fout);

}

int main() {

// 从文件中读取数据

ifstream fin("data.txt");

if (!fin) {

cerr << "打开文件data.txt失败." << endl;

return 1;

}

// 建立AVL树

Node \*root = nullptr;

int key, value;

while (fin >> key >> value) {

root = insert(root, key, value);

}

while (1){

// 菜单

int command;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

cout<<"\*\* 1.查找节点 \*\*"<<endl;

cout<<"\*\* 2.插入节点 \*\*"<<endl;

cout<<"\*\* 3.删除节点 \*\*"<<endl;

cout<<"\*\* 4.输出AVL树\*\*"<<endl;

cout<<"\*\* 5. 退出 \*\*"<<endl;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

cin >> command;

if (command == 1){

// 查找节点

int searchKey;

cout << "请输入一个整数关键字Key进行查找节点: ";

cin >> searchKey;

Node \*node = find(root, searchKey);

if (node != nullptr) {

cout << "Key: " << node->key << ", Value: " << node->value << endl;

} else {

cout << "关键字未找到." << endl;

}

system("pause");

} else if (command == 2){

// 插入节点

int insertKey, insertValue;

cout << "请输入一个整数关键字Key和一个整数Value进行插入节点: ";

cin >> insertKey >> insertValue;

root = insert(root, insertKey, insertValue);

cout << "插入成功."<<endl;

system("pause");

} else if (command == 3){

// 删除节点

int removeKey;

cout << "请输入一个整数关键字Key进行删除节点: ";

cin >> removeKey;

root = remove(root, removeKey);

cout << "删除成功，接下来将输出整棵树观察结果"<<endl;

// 以可验证的方式输出结果

cout << "AVL Tree: " << endl;

printTree(root);

system("pause");

} else if (command == 4){

// 以可验证的方式输出结果

cout << "AVL Tree: " << endl;

printTree(root);

system("pause");

} else if (command == 5){

// 修改后结果存放至data.txt，以覆盖的形式

ofstream fout("data.txt");

if (!fout) {

cerr << "打开文件data.txt失败." << endl;

return 1;

}

printTreeFile(root,fout);

break;

} else {

cout << "输入指令无效，请重新输入"<<endl;

system("pause");

}

//清屏

system("cls");

}

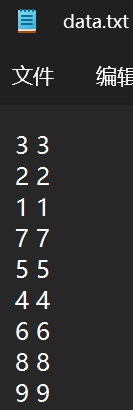
system("pause");

return 0;

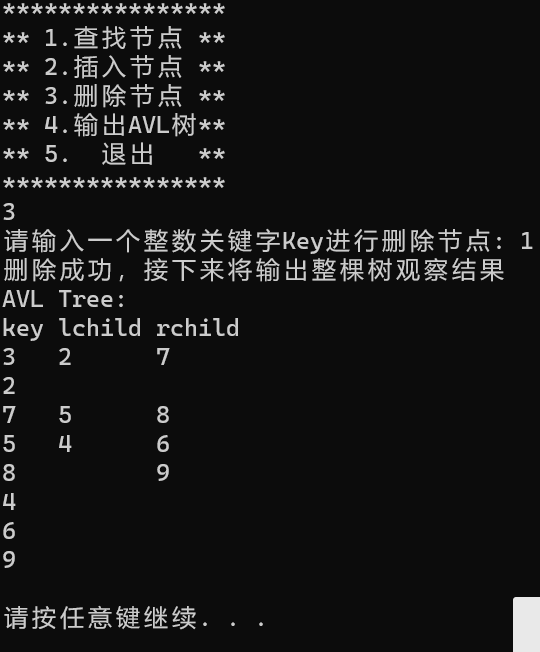
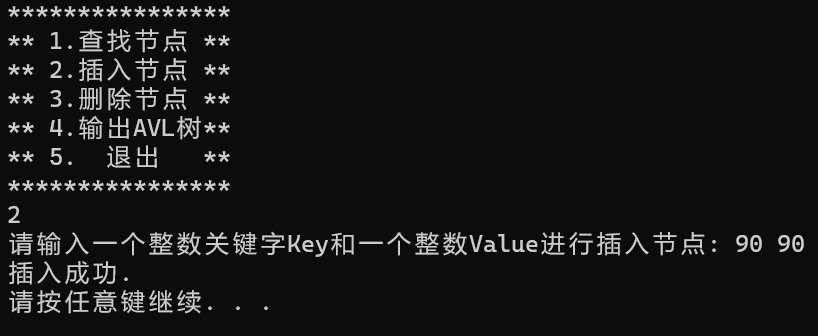
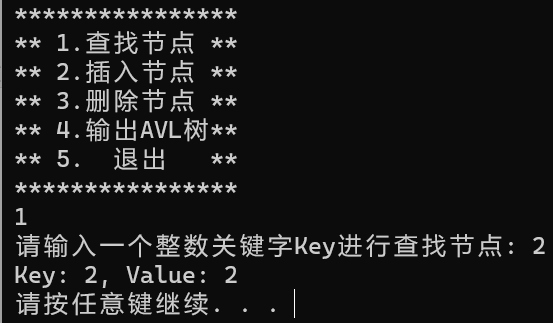
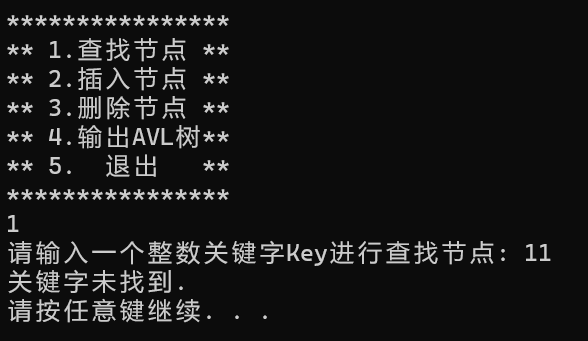
}

#### 测试数据和结果

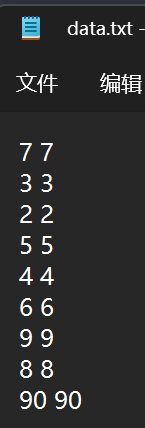
##### 测试数据



##### 结果



##### 测试结果中的源文件



#### 时间复杂度：

* insert(Node \*node, int key, int value)：O(h)
* remove(Node \*node, int key)：O(h)
* find(Node \*node, int key)：O(h)
* 其中，h为AVL树的高度。由于AVL树是一种平衡二叉树，所以它的高度一般不会超过O(logn)，因此这段代码中的各个函数的时间复杂度也就不会超过O(logn)

find(Node \*node, int key)：该函数的时间复杂度为O(h)，其中h为AVL树的高度。在最坏的情况下，每次查找操作都会沿着AVL树的最底层进行搜索，这样就会导致时间复杂度升高。但是由于AVL树是一种平衡二叉树，所以它的高度一般不会超过O(logn)，因此查找操作的时间复杂度也就不会超过O(logn)。

remove(Node \*node, int key)：该函数的时间复杂度为O(h)，其中h为AVL树的高度。在最坏的情况下，每次删除操作都会沿着AVL树的最底层进行搜索，这样就会导致时间复杂度升高。但是由于AVL树是一种平衡二叉树，所以它的高度一般不会超过O(logn)，因此删除操作的时间复杂度也就不会超过O(logn)。

insert(Node \*node, int key, int value)：该函数的时间复杂度为O(h)，其中h为AVL树的高度。在最坏的情况下，每次插入操作都会沿着AVL树的最左侧或最右侧进行搜索，这样就会导致时间复杂度升高。但是由于AVL树是一种平衡二叉树，所以它的高度一般不会超过O(logn)，因此插入操作的时间复杂度也就不会超过O(logn)。

#### 反思：

菜单功能呢写在主函数不方便复用。输出树的信息不够直观可视化。

#### 改进：

仅在退出时才把现在在程序上的树保存至源文件，不执行其他操作就保存，减少程序不必要运行，

#### 该题代码行：

300行