

浙江工业大学

数据结构课程设计 实验报告

2021/2022(1)



实验题目 用户登录系统的模拟

学生姓名 苏源昌

学生学号 202003150722

学生班级 智控 2002

任课教师 刘端阳

提交日期 2021/12/28

计算机科学与技术学院

目录

一、实验题目和要求.....	1
二、运行环境.....	1
三、设计思路.....	2
3.1 系统总体设计.....	2
3.2 系统功能设计.....	2
3.3 类的设计.....	3
(1) 类的关系.....	3
(2) 主要成员函数的编写.....	4
3.4 主程序的设计.....	7
四、调试分析	8
4.1 调试错误分析.....	8
五、测试结果分析.....	17
六、附录：源代码.....	21

用户登录系统的模拟 实验报告

一、 实验题目和要求

【问题描述】在登录服务器系统时，都需要验证用户名和密码，如 telnet 远程登录服务器。用户输入用户名和密码后，服务器程序会首先验证用户信息的合法性。由于用户信息的验证频率很高，系统有必要有效地组织这些用户信息，从而快速查找和验证用户。另外，系统也会经常会添加新用户、删除老用户和更新用户密码等操作，因此，系统必须采用动态结构，在添加、删除或更新后，依然能保证验证过程的快速。请采用相应的数据结构模拟用户登录系统，其功能要求包括用户登录、用户密码更新、用户添加和用户删除等。

【基本要求】

1. 要求自己编程实现二叉树结构及其相关功能，以存储用户信息，不允许使用标准模板类的二叉树结构和函数。同时要求根据二叉树的变化情况，进行相应的平衡操作，即 AVL 平衡树操作，四种平衡操作都必须考虑。测试时，各种情况都需要测试，并附上测试截图；
2. 要求采用类的设计思路，不允许出现类以外的函数定义，但允许友元函数。主函数中只能出现类的成员函数的调用，不允许出现对其它函数的调用。
3. 要求采用多文件方式：.h 文件存储类的声明，.cpp 文件存储类的实现，主函数 main 存储在另外一个单独的 cpp 文件中。如果采用类模板，则类的声明和实现都放在.h 文件中。
4. 不强制要求采用类模板，也不要求采用可视化窗口；要求源程序中有相应注释；
5. 要求测试例子要比较详尽，各种极限情况也要考虑到，测试的输出信息要详细易懂，表明各个功能的执行正确；
6. 建议采用 Visual C++ 6.0 及以上版本进行调试；

【实现提示】

1. 用户信息(即用户名和密码)可以存储在文件中，当程序启动时，从文件中读取所有的用户信息，并建立合适的查找二叉树；
2. 验证过程时，需要根据登录的用户名，检索整个二叉树，找到匹配的用户名，进行验证；更新用户密码时，也需要检索二叉树，找到匹配项后进行更新，同时更新文件中存储的用户密码。
3. 添加用户时，不仅需要在文件中添加，也需要在二叉树中添加相应的节点；删除用户时，也是如此；

二、 运行环境

用户登录系统在 JetBrains Clion 平台下开发

操作系统：Windows 10 家庭中文版。

硬件环境：

处理器：AMD Ryzen 7 4800U with Radeon Graphics 1.80 GHz

内存：16.00GB

系统类型：64 位操作系统

三、 设计思路

3.1 系统总体设计

数据结构：AVL 树

描述：AVL 树，又称为平衡二叉树，是一种二叉排序树，其中每一个结点的左子树和右子树的高度差至多等于 1。

特点：

1. 是一棵二叉排序树。
2. 高度平衡：每个结点的左右子树的高度之差的绝对值（平衡因子）最多为 1。

技术思路：

将用户作为 AVL 树的节点，实现系统添加新用户、删除老用户和更新用户密码等操作，系统采用 AVL 树的动态结构，在添加、删除或更新后，依然能保证验证过程的快速。

3.2 系统功能设计

用户登录系统主要功能为：①用户登录②用户密码更新③用户添加④用户删除

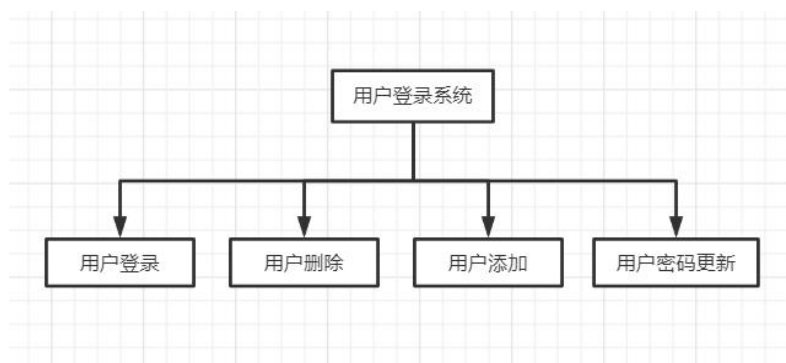


图 1 系统功能结构图

系统各模块的功能具体描述为：

1、用户登录

用户需要填入正确的用户名和密码实现用户登录。

2、用户密码更新

用户登录后，可以选择该功能，输入新密码，实现密码更新。

3、用户添加

用户登录后，可以选择该功能，输入新用户名和该用户密码，实现用户添加。

4、用户删除

用户登录后，可以选择该功能，输入数字 n ($n < \text{总用户数}$)，将删除按照层次遍历中的第 n 个节点用户。

3.3 类的设计

系统涉及对象有两个基本类：Node 类、AVLTree 类。AVLTree 类涉及的功能操作归纳为如下表 1 所示：

表 1 类涉及的操作

类	操作/功能	
AVLTree 类	插入	
	删除	
	查询	
	显示	
	平衡	左旋
		右旋
		左右旋
		右左旋
Node 类	存储用户信息	
	平衡因子	
	左孩子节点	右孩子节点

(1) 类的关系

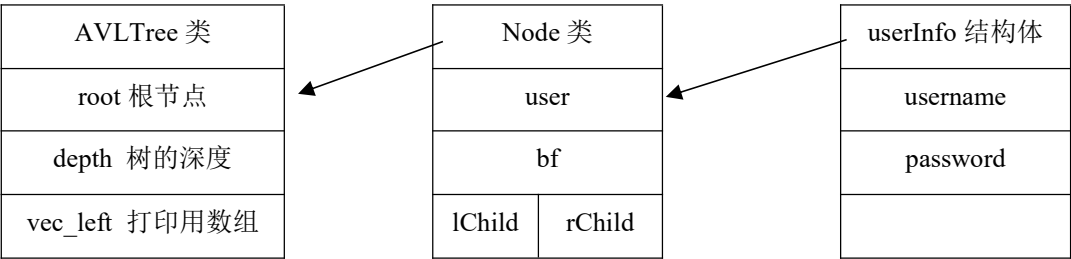


图 2 系统类结构图

AVLTree类的root根节点是Node类的指针,Node类的user成员是userInfo结构体,userInfo结构体存有两个用户信息相关字符串

(2) 主要成员函数的设计

本项目中的重要成员函数全部在 `AVLTree` 类内，所以以下主要成员函数全是 `AVLTree` 内声明的。

要想将文件中用户数据读入并形成树的结构，最主要是用户的添加函数，即将用户一个个添加到原本为空的 AVL 树中。我们知道，平衡二叉树，其实就是在二叉排序树创建的过程中保证他的平衡性，一旦发现有不平衡的情况，马上进行处理，这样就不会出现无法处理的情况。

如何知道我们的树是平衡的？我们需要知道每个结点的 `bf`，一旦有结点出现 `bf` 的绝对值大于 1 的情况，就说明此树是不平衡的。那么，如何寻找到最小不平衡子树成为了关键之关键。

那么接下来先获取结点的 `bf` 值。

获取树的高度和结点的 `bf` 值的函数 `getHeight`

计算一个结点的 `bf` 值是利用其左子树的深度减去右子树的深度。那么，利用递归的方法分别求出左右子树的深度，取其中最大值再加 1(结点本身的高度)，返回给其双亲结点，就可以作为其双亲结点的左子树或者右子树的高度，将他们进行相减，就可以得到 `bf`。当返回到根节点时，返回值就是整个 AVL 树的高度。

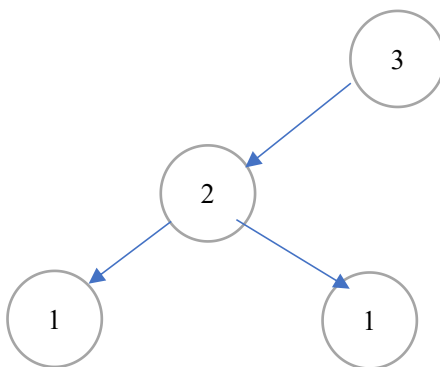


图 3 获取 `bf` 推导图

上图中每个结点内部就是其返回值，通过其左右子树的高度最大值加 1 得出。这是试想其在底部的情况，不过其它情况也可以轻易地类推得出。

知道了 `bf` 值以后，就可以知道树是否平衡了，以此设计结点的添加函数。

结点的添加函数 add

在平衡操作以前，我们首先需要把结点添加到树中。由于 AVL 树是一个二叉排序树，我们只需要根据大小去进行插入即可。

以下是添加函数的流程图

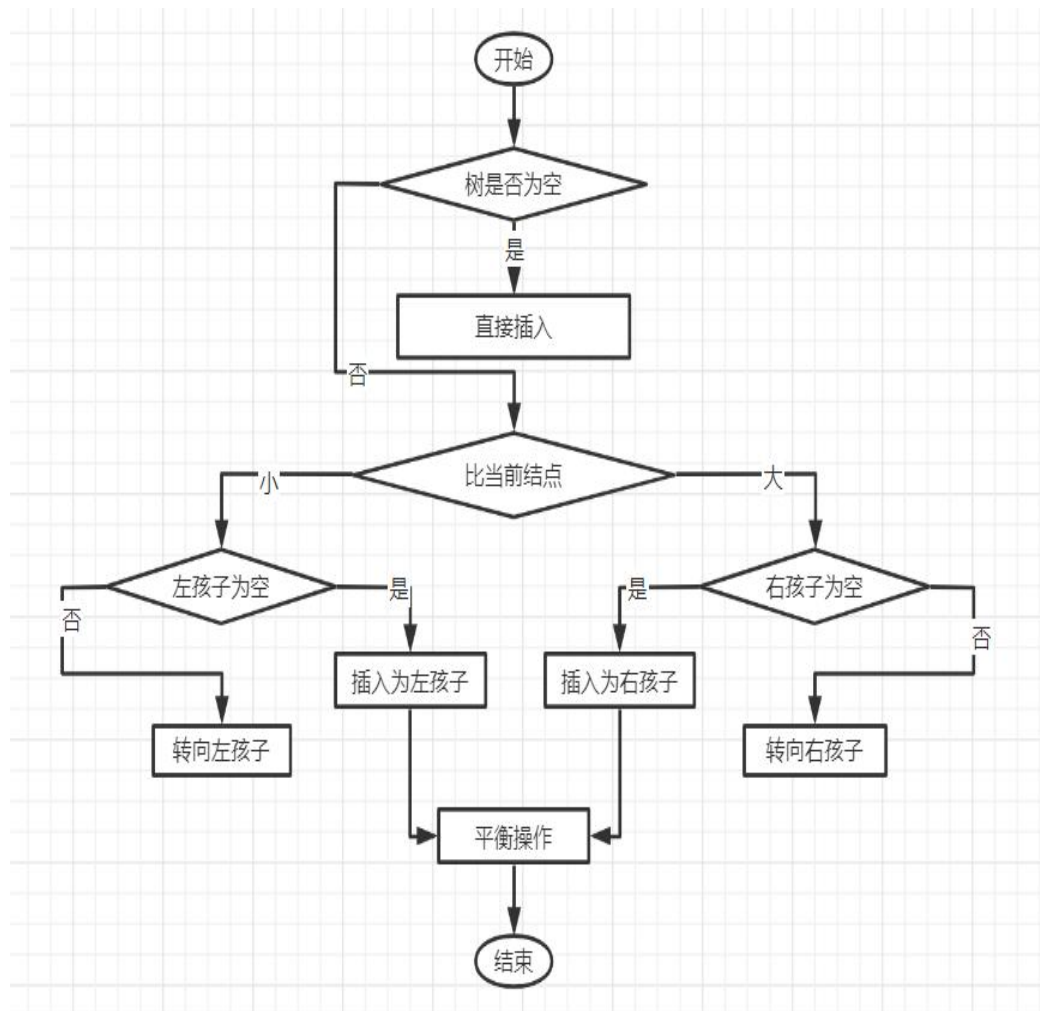


图 4 添加用户流程图

特别注意，每次结点操作之后都要重新获取一遍所有结点的 bf 值，即运行一次 getHeight 函数，后面的删除操作也是一样的。

这里涉及到了平衡操作，接下来设计平衡操作函数。

平衡操作函数 RRotate(右旋)

由于平衡操作具有对称性，下面只介绍右旋的情况，左旋只需反向即可，由理论得知，当最小不平衡子树根结点的平衡因子大于 1 时，进行右旋，小于 1 时就左旋。

当其和他孩子结点的 bf 相反时，需要对孩子结点先进行一次旋转使得符号相同后，再反向旋转一次就能够完成平衡操作。

当最小不平衡子树的孩子结点旋转时，会出现需要将其子树移动到其后继结点左孩子的情况(保证二叉排序树的特性，左子树全体一定都要比其后继结点小)，所以也要纳入考虑。

当操作对象为根节点时，也有其特殊性(parent 指针为空。以及 AVL 树根变化，root 指针需要重新定向)，所以也要单独进行操作

下方是右旋函数的流程图：

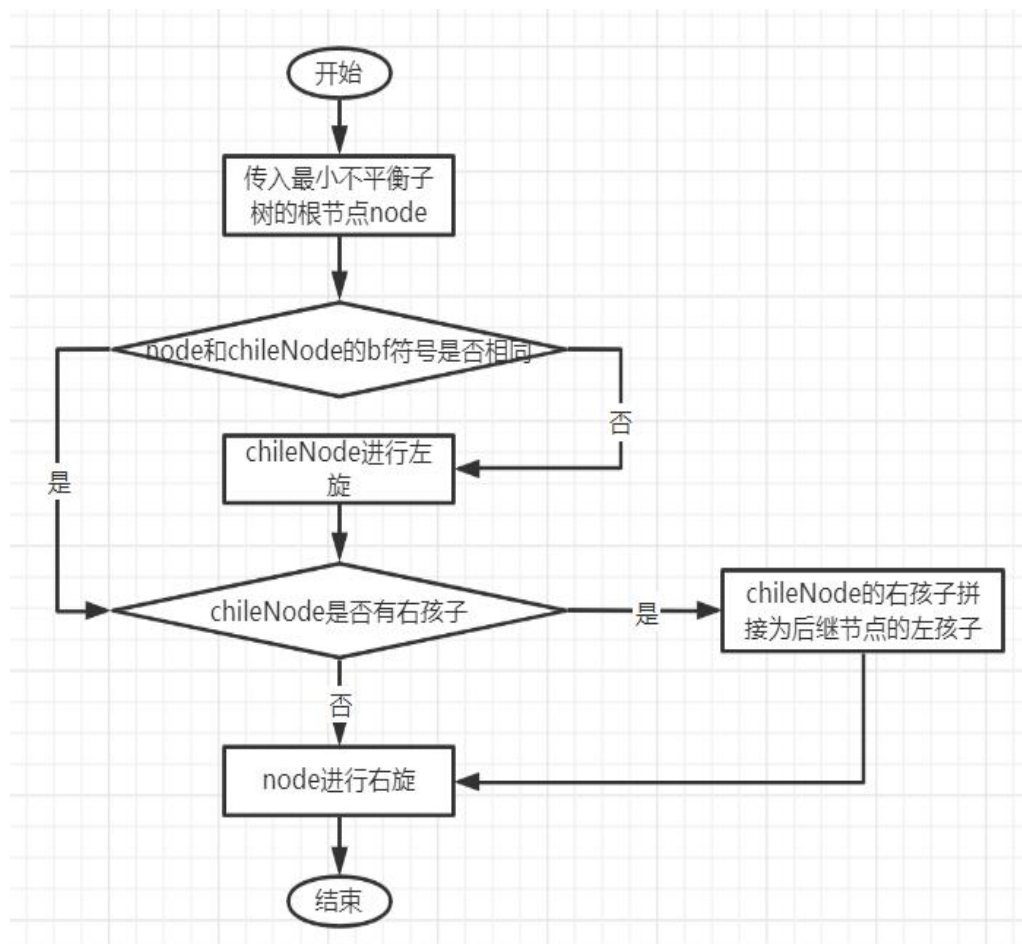


图 5 平衡结点流程图

现在我们可以读取用户信息了，也能够对其进行添加用户的操作。接下来设计删除用户的函数。

用户删除函数 deleteNode

当一个结点被删除时，对目标结点来说有三种情况：没有孩子结点，有一个孩子结点，有两个孩子结点。

没有孩子结点，即当其为叶子结点的时候，这个时候只需要将其删除并将其双亲结点对应指针置空即可。

有一个孩子结点时，将其双亲结点对应指针指向孩子结点即可。

当有两个孩子结点时，和旋转时情况类似：为了保持二叉排序树的特性，我们需要将其左子树的树根拼接接到其后继结点的左孩子的位置。再将其双亲结点对应指针指向右子树根。

接下来是删除之后对树进行平衡的操作。当目标结点只有一个孩子时，结点被删除，其子树的 bf 值是不会更改的(无孩子时更不用说)，只需要在原目标结点位置向上进行平衡即可。

当目标结点有两个孩子时，都是左子树拼接接到其后继结点。根据推导，我们只需要计算目标结点的右子树的 bf 即可：由于原本右子树根的 bf 绝对值一定是小于 1 的。而后继结点相当于拼接接到右子树根的左子树处，相当于给 bf 加上了一个正数，所以右子树根的 bf 一定是越来越大的，所以我们只需要将其旋转到 bf 小于 2 时即可，这时再向上层进行平衡。由于递归的特性，我们只需要在向上层递归的过程中添加平衡的操作即可。

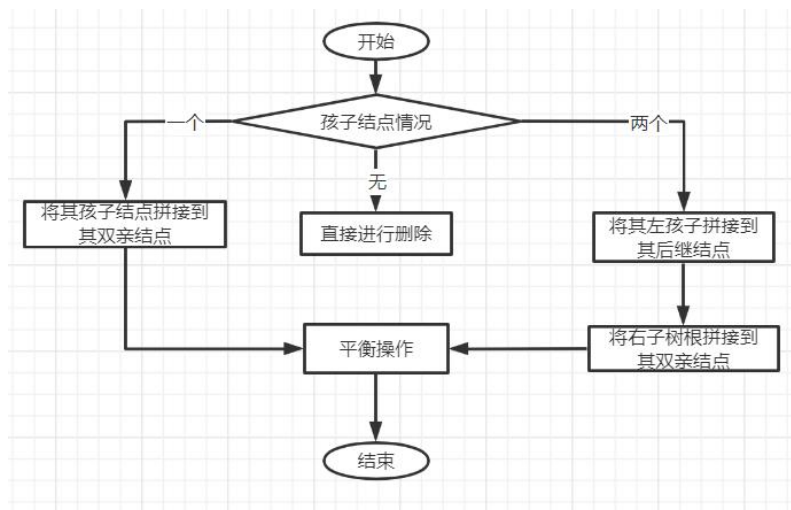


图 6 删除结点流程图

密码的更新，用户的登录，只需要用中序遍历，找到匹配的结点进行判断和操作即可，在此不作展示。

3.4 主程序的设计

用户登录系统主要功能：①用户登录②用户密码更新③用户添加④用户删除

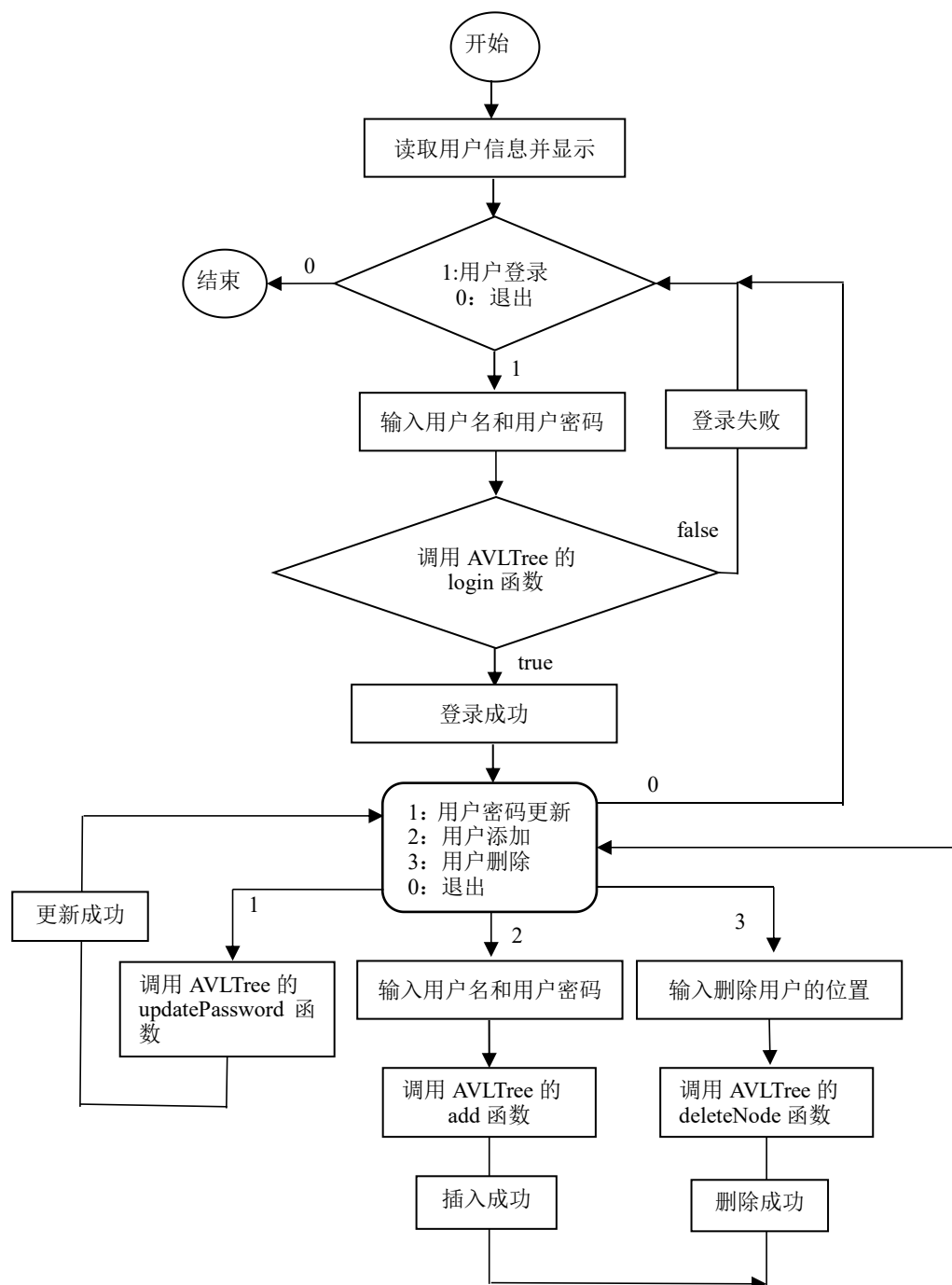


图 7 删除结点流程图

四、 调试错误分析

注：在下面的三个调试过程中，树形输出尚未更改，靠上方的是左子树，下方的是右子树。

● 问题 1:

问题描述：写新结点插入函数时，发现无法输出

修正前的截图：

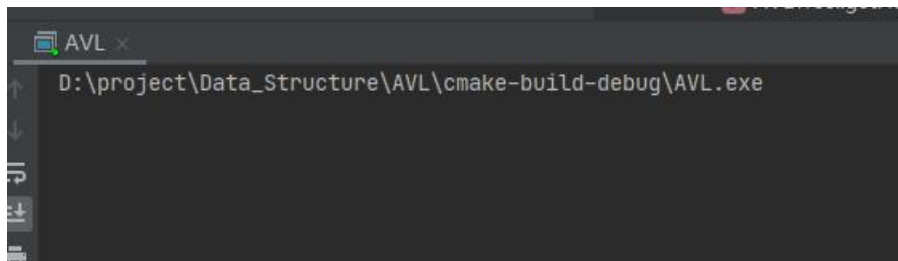


图 8 问题一截图

调试截图：

猜测问题出在对结点进行操作的部分，添加如下断点进行测试

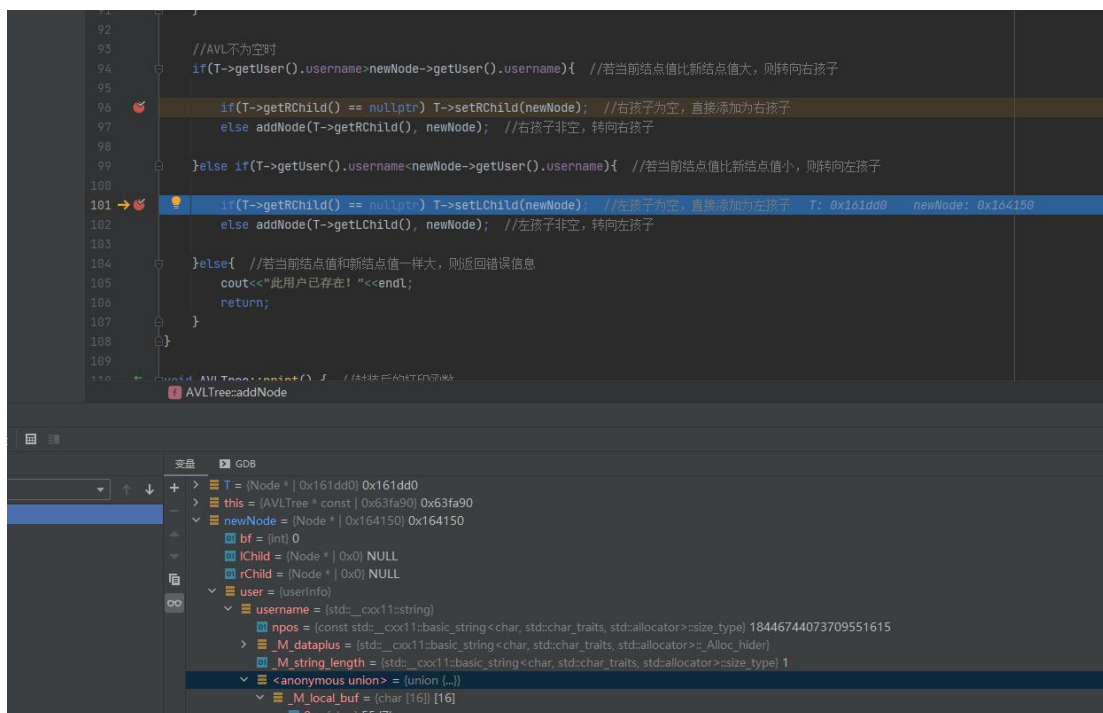


图 9 问题一调试截图

经过调试，发现之前为 2 的结点位置变成了 3，被覆盖了，猜测是结点索引的问题

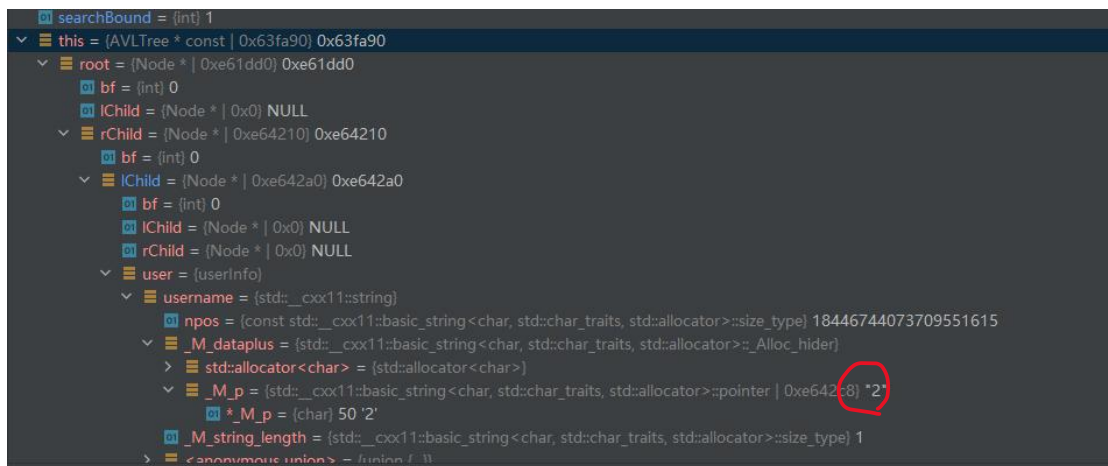


图 10 问题一调试截图

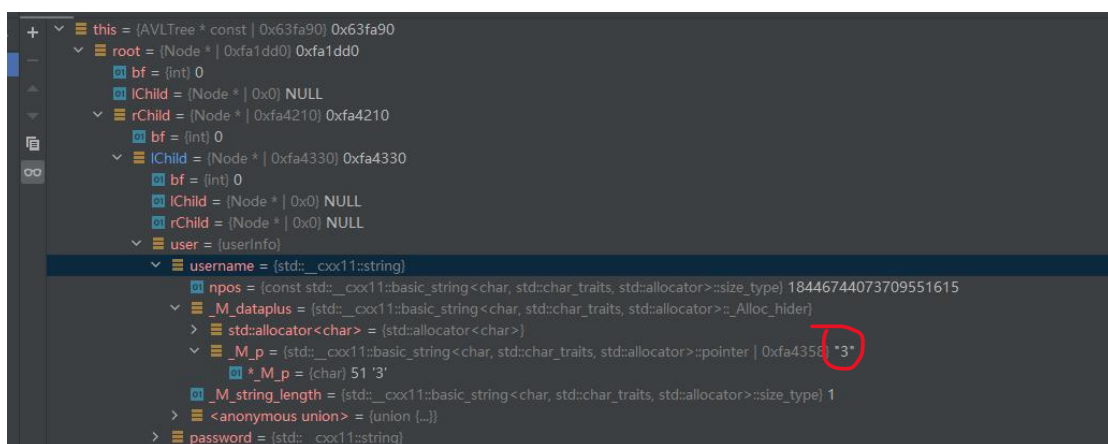


图 11 问题一调试截图

发现是 L 写成了 R



图 12 问题一调试截图

解决方法:
将 R 改成 L

修正后的截图：

```
//AVL不为空时
if(T->getUser().username>newNode->getUser().username){ //若当前结点值比新结点值大，则转向左孩子

    if(T->getLChild() == nullptr) T->setLChild(newNode); //左孩子为空，直接添加为左孩子
    else addNode(T->getLChild(), newNode); //左孩子非空，转向左孩子

}else if(T->getUser().username<newNode->getUser().username){ //若当前结点值比新结点值小，则转向右孩子

    if(T->getRChild() == nullptr) T->setRChild(newNode); //右孩子为空，直接添加为右孩子
    else addNode(T->getRChild(), newNode); //右孩子非空，转向右孩子

}else{ //若当前结点值和新结点值一样大，则返回错误信息
    cout<<"用户\"<<newNode->getUser().username<<"\"已存在! "<<endl;
    delete newNode; //释放空间
    return;
}
```

图 13 问题一调试截图

更改后输出正常

● 问题 2:

问题描述： 删除一个有两个孩子的根结点时，发现输出的树形是错误的

修正前的截图：

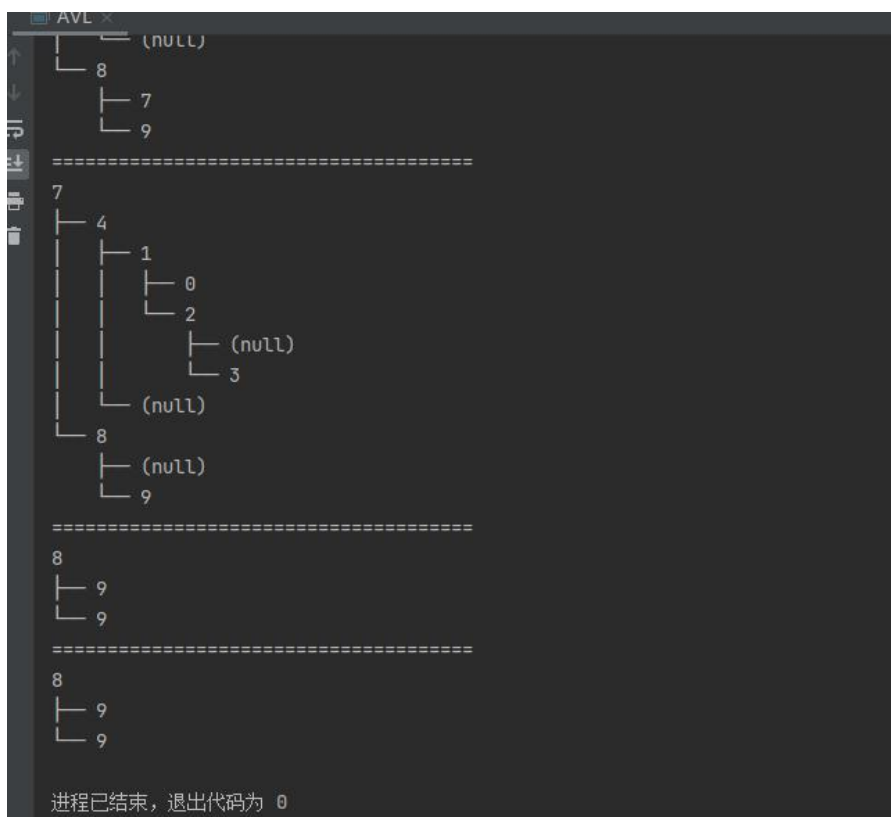


图 14 问题二调试截图

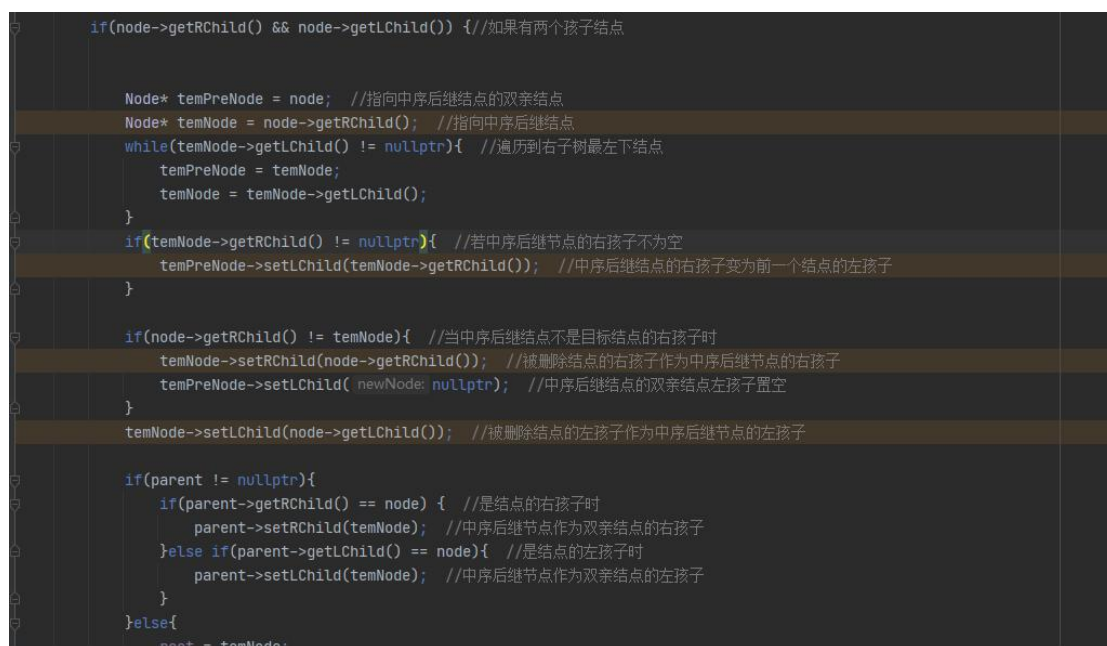


图 15 问题二调试截图

调试截图:

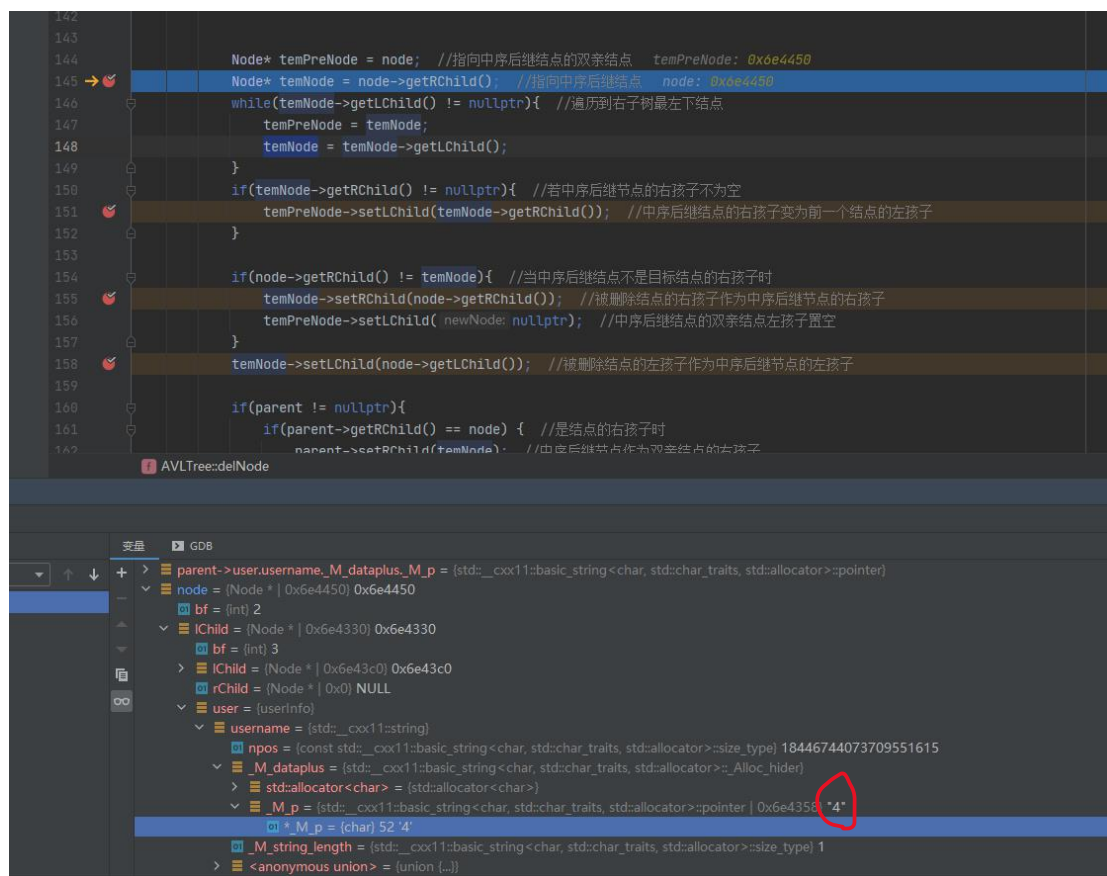


图 16 问题二调试截图

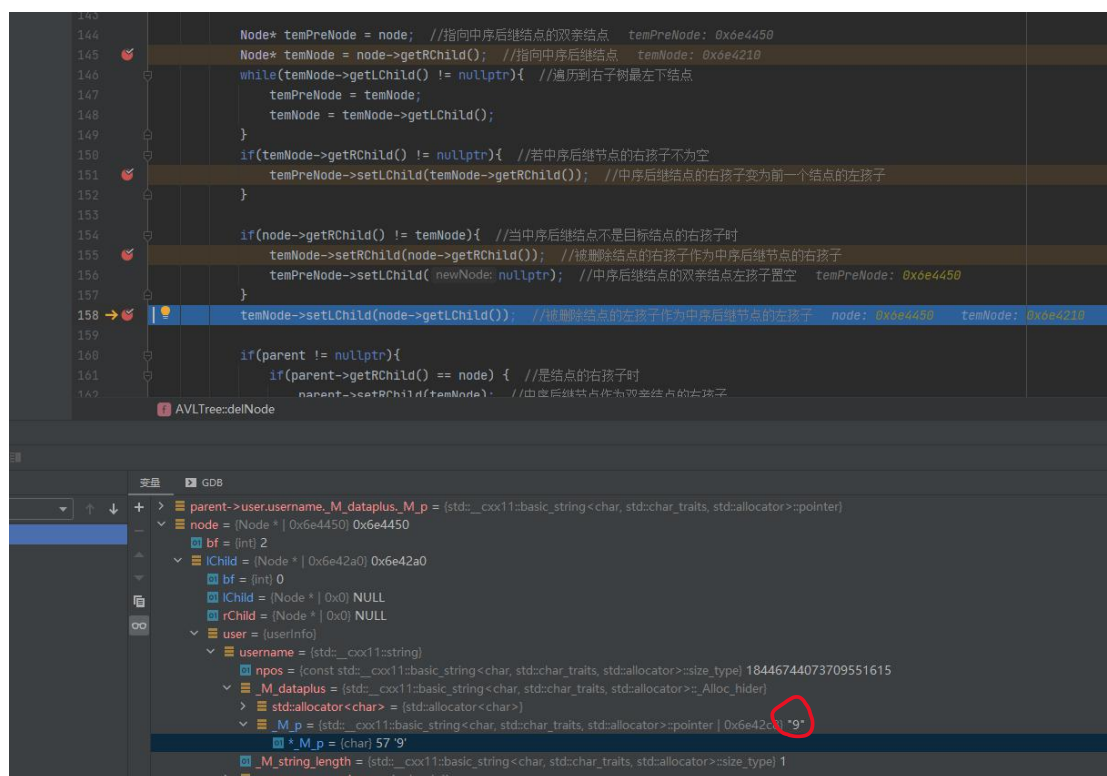


图 17 问题二调试截图

可以看到，一开始根节点的左孩子是 4，但是后来变成了 9，所以是因为判定的逻辑顺序没有设置好从而导致了中序后继节点的右孩子接到了根节点的右孩子，对于中序后继节点是目标结点的右孩子的情况欠考虑，所以我们要先判断其是否为目标结点的右孩子，如果不是，才去对中序后继节点的孩子结点进行操作，否则只需要对其本身进行操作即可。

解决方法：



图 18 问题二调试截图

```
if(node->getRChild() != temNode){ //当中序后继结点不是目标结点的右孩子时
    if(temNode->getRChild() != nullptr){ //若中序后继结点右孩子不为空
        temPreNode->setLChild(temNode->getRChild()); //中序后继结点的右孩子变为前一个结点的左孩子
    }
    temNode->setRChild(node->getRChild()); //被删除结点的右孩子作为中序后继结点的右孩子
    temPreNode->setLChild( newNode: nullptr); //中序后继结点的双亲结点左孩子置空
}
temNode->setLChild(node->getLChild()); //被删除结点的左孩子作为中序后继结点的左孩子
```

图 19 问题二调试截图

将处理中序后继结点的孩子结点的代码段,放进判断中序后继结点是否为目标结点的孩子结点的语句中,先对其位置进行判断,再进行操作。

修正后的截图:

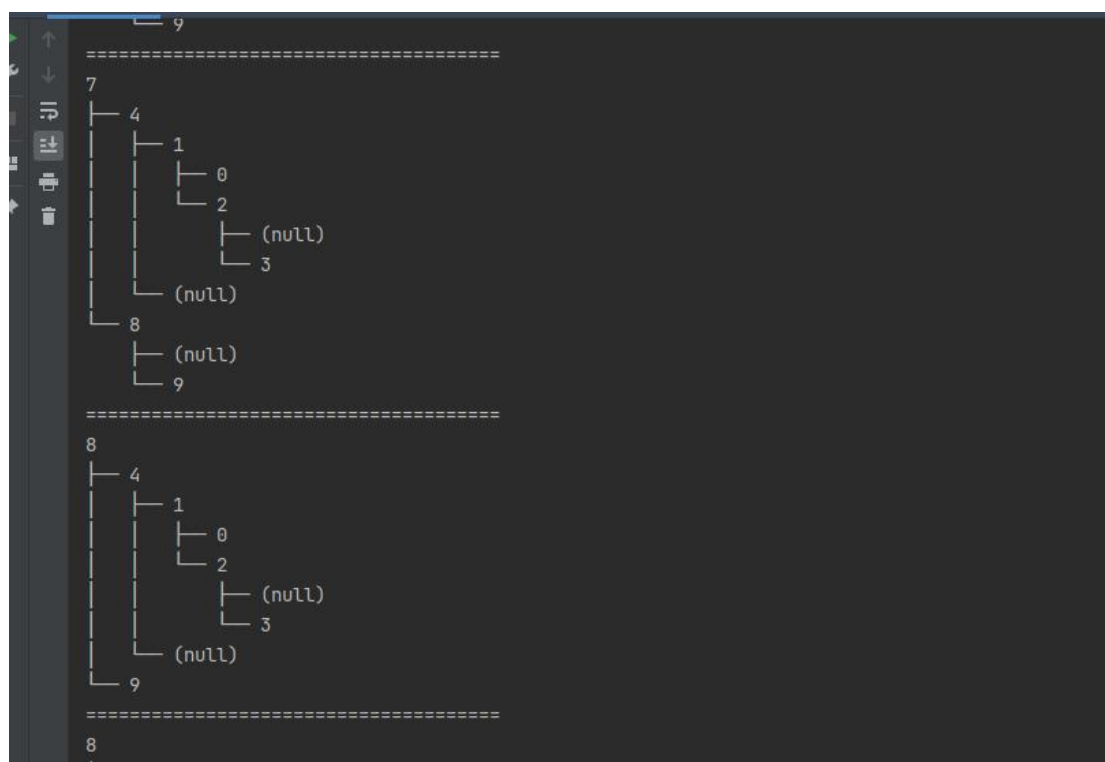


图 20 问题二调试截图

最后输出结果正常

- 问题 3:

问题描述: 在 AVL 进行平衡操作时, 发现值为 2 的结点丢失

修正前的截图:

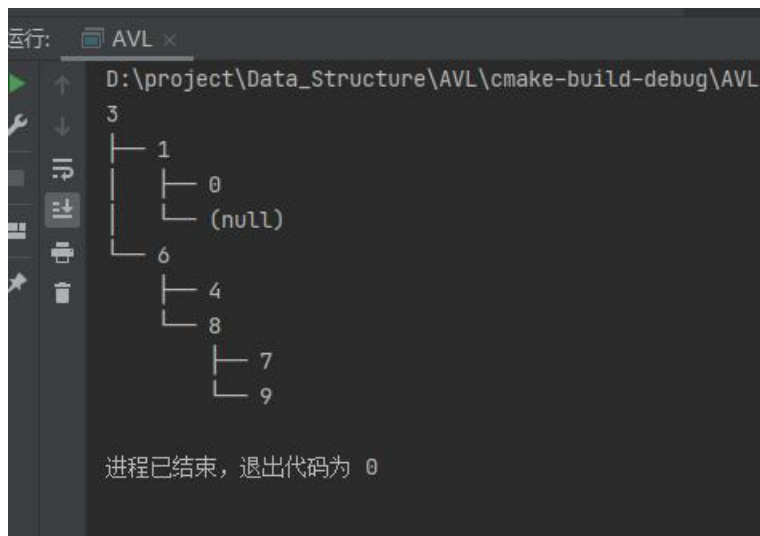


图 21 问题三调试截图

```
void AVLTree::LRotate(Node *node, Node *parent) {
    if(node == root){ //若最小不平衡子树根结点为根节点
        if (node->getRChild()->getBF() > 0) { //如果符号不同, 则子结点先进行右旋
            RRotate(node->getRChild(), parent);
        }
        Node *temNode = root;
        root = root->getRChild();

        if(root->getLChild() == nullptr){ //如果孩子结点没有左孩子
            root->setLChild(temNode);
        }else{ //如果孩子结点有两个孩子结点, 则将孩子结点的左孩子移到双亲结点的右下角
            temNode->setRChild( newNode: root->getLChild());
            root->setLChild(temNode);
        }
        temNode->setRChild( newNode: nullptr);
    } else {
        if (node->getRChild()->getBF() > 0) { //如果符号不同, 则子结点先进行右旋
            RRotate(node->getRChild(), parent);
        }
    }
}
```

图 22 问题三调试截图

调试截图：

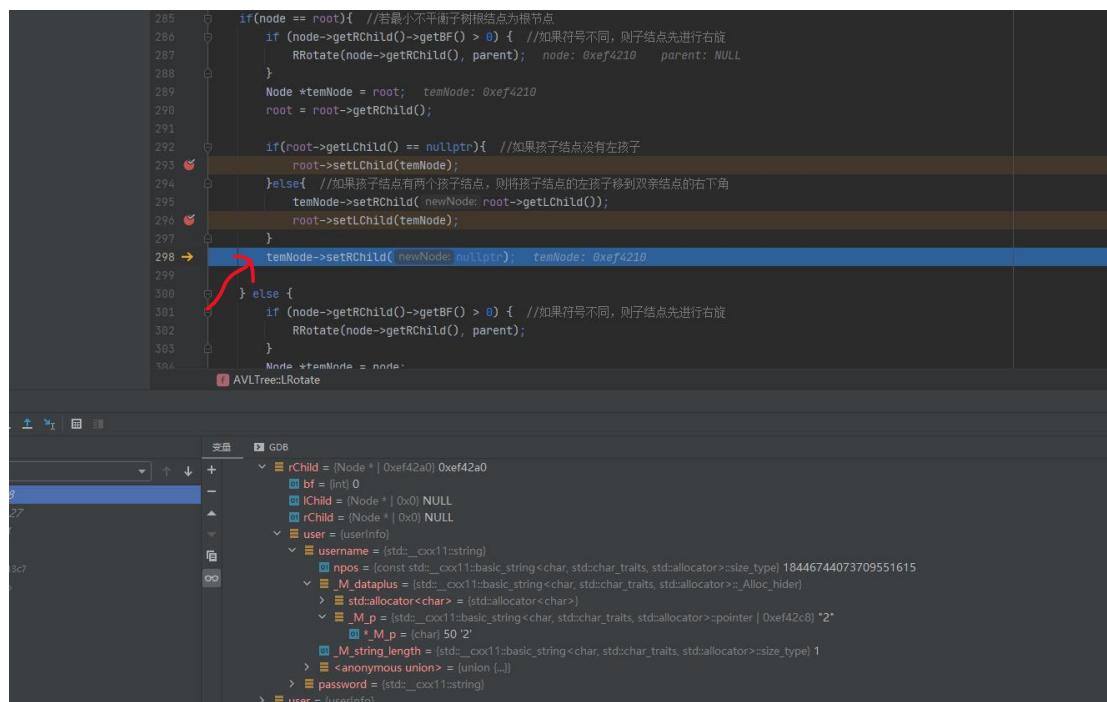


图 23 问题三调试截图

可以看到，当最小不平衡子树进行左旋操作时，其右孩子结点的左孩子结点的值不为空，所以要拼接到左子树的最右下方，拼接到左子树最右下方时是正常的

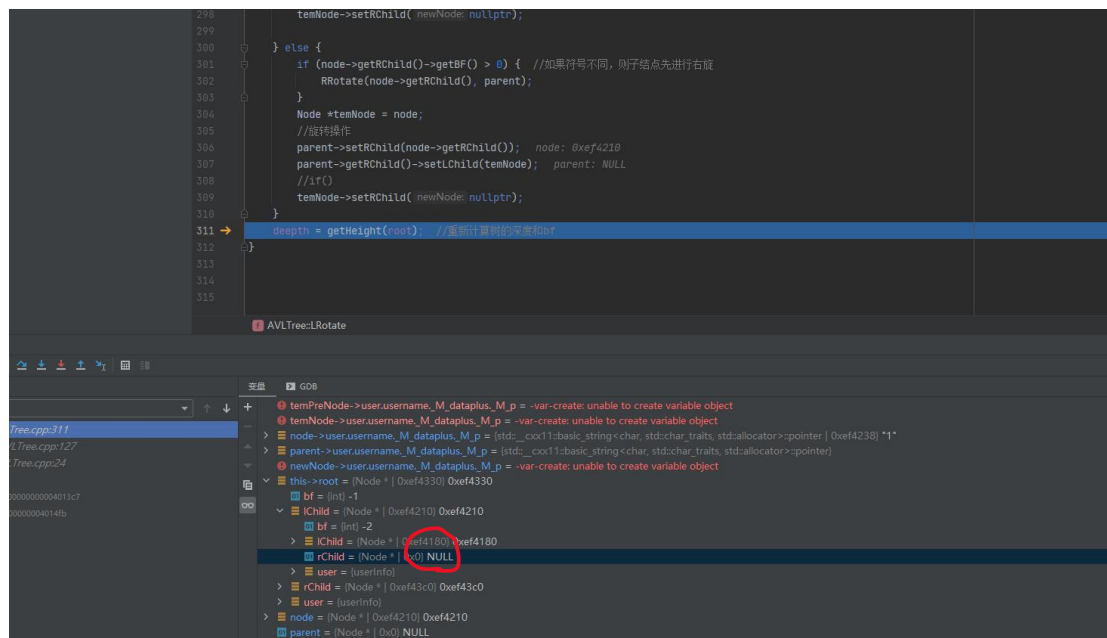


图 24 问题三调试截图

但执行完后一句后变成了 NULL，是由于置最小不平衡子树的右节点为空导致的，这个操作原本应该是在其原状况下右孩子结点的左孩子结点的值不为空情况下进行的，但因为判断条件外，所以导致两种情况都会进行调用。

五、 测试结果分析

登录测试：



图 27 登录测试截图

使用用户名 3，密码为 666 进行登录

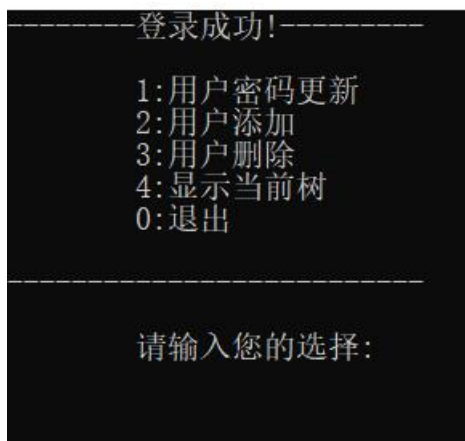


图 28 登录测试截图

登陆成功，显示当前树的结构：



图 29 当前树截图

可见树是平衡的(下方是左子树，上方是右子树)。

下面进行用户添加的操作：

添加用户名为 5，密码为 1234

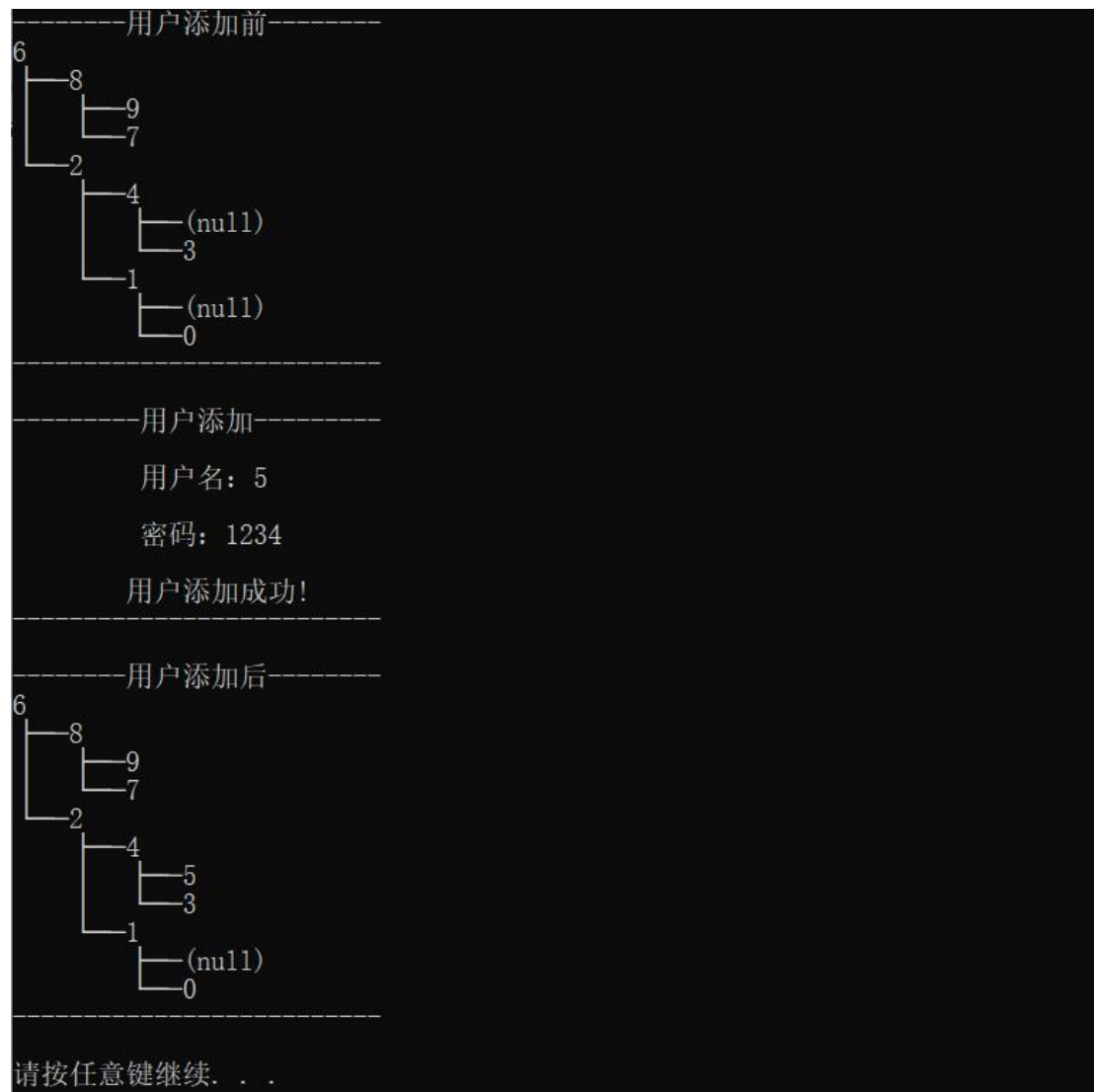


图 30 用户添加截图

添加成功后仍是 AVL 树。

下面进行用户删除操作：

删除用户名为 6



图 31 用户删除截图

可见删除后树还是平衡的。
下面进行密码修改功能测试：

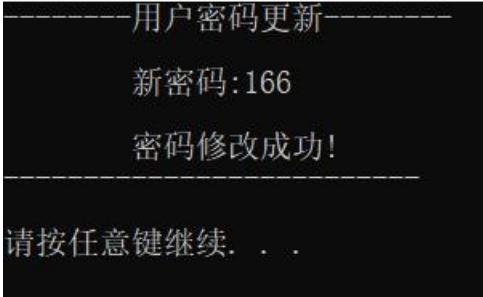
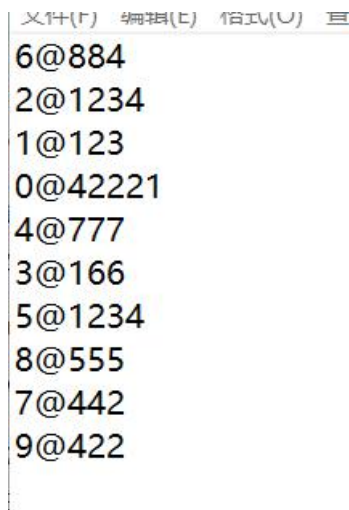


图 32 用户密码修改截图

密码修改成功



文件(F) 编辑(E) 格式(O) 窗

6@884
2@1234
1@123
0@42221
4@777
3@166
5@1234
8@555
7@442
9@422

图 33 文件输出和更改结果图

可见密码修改和用户添加成功输出到文件

经过上面的各功能测试，主函数功能也是正常的，测试完毕

六、 附录：源代码

main.cpp

```
//----main.cpp----//
#include "AVLTree.h"
int main() {
    ifstream inFile;
    ofstream outFile;
    int num = -1, num2 = -1;
    string name, pwd;

    inFile.open("test.txt",ios::in);
    outFile.open("test1.txt");
    AVLTree test(inFile); //读取数据

    while(true){
        system("cls");
        cout << "-----用户登录系统-----\n\n";
        cout << "          1:用户登录\n";
        cout << "          0:退出\n\n";
        cout << "-----\n\n";
        cout << "          请输入您的选择:";
        cin >> num;
        switch (num)
        {
```

```
case 0:
    break;
case 1:
{
    system("cls");
    cout << "-----登录-----\n\n";
    cout << "        用户名: ";
    cin >> name;
    cout << "\n        密码: ";
    cin >> pwd;
    test.login(name, pwd);
    if (test.getSign())
    {
        test.reSign();
        while (1)
        {
            system("cls");
            cout << "-----登录成功!-----\n\n";
            cout << "        1:用户密码更新\n";
            cout << "        2:用户添加\n";
            cout << "        3:用户删除\n";
            cout << "        4:显示当前树\n";
            cout << "        0:退出\n\n";
            cout << "-----\n\n";
            cout << "        请输入您的选择:";
            cin >> num2;
            switch (num2)
            {
                case 0:
                    break;
                case 1:
                {
                    system("cls");
                    cout << "-----用户密码更新-----\n\n";
                    test.updatePassword(name, pwd);
                    if (test.getSign())
                    {
                        test.reSign();
                        cout << "\n        密码修改成功!\n";
                        cout << "-----\n\n";
                    }
                    system("pause");
                    break;
                }
            }
        }
    }
}
```



```
case 2:
{
    system("cls");
    string nName, nPwd;
    cout << "-----用户添加前-----\n";
    test.print();
    cout << "-----\n\n";
    cout << "-----用户添加-----\n\n";
    cout << "          用户名: ";
    cin >> nName;
    cout << "\n          密码: ";
    cin >> nPwd;
    test.add(nName, nPwd);
    if (test.getSign())
    {
        test.reSign();
        cout << "\n          用户添加成功!\n";
        cout << "-----\n\n";
        cout << "-----用户添加后-----\n";
        test.print();
        cout << "-----\n\n";
    }
    system("pause");
    break;
}
case 3:
{
    system("cls");
    string dName;
    cout << "-----用户删除前-----\n";
    test.print();
    cout << "-----\n\n";
    cout << "-----用户删除-----\n\n";
    cout << "          删除的用户名:";
    cin >> dName;
    test.deleteNode(dName);
    if (test.getSign()) {
        test.reSign();
        cout << "\n          用户删除成功!\n\n";
        cout << "-----用户删除后-----\n";
        test.print();
        cout << "-----\n\n";
    }
}
else
```

```
        {
            cout << "\n        用户删除失败!\n\n";
        }
        system("pause");
        break;
    }
    case 4:
        system("cls");
        test.print();
        system("pause");
        break;
    default:
        break;
    }
    if (num2 == 0)
    {
        system("cls");
        break;
    }
}
else
{
    cout << "\n    用户名或密码错误!\n\n";
    system("pause");
}
break;
}
default:
    break;
}
if (num == 0) break;
}
test.save(outFile); //保存数据
outFile.close();
inFile.close();
return 0;
}
```

Node.h

```
//---Node.h---//
#ifndef AVL_NODE_H
```

```
#define AVL_NODE_H
#include <string>
using namespace std;
struct userInfo{ //用户数据结构体
    string username;
    string password;
};

class Node {
private:
    int bf = 0; //结点平衡因子
    Node *lChild = nullptr, *rChild = nullptr; //左右孩子指针
    userInfo user; //用户信息

public:
    Node(string usn, string pwd); //初始化
    userInfo& getUser(); //获取用户信息
    Node* getLChild(); //获取左孩子
    Node* getRChild(); //获取右孩子
    int getBF(); //获取平衡因子
    void setBf(int bf); //设置平衡因子
    void setLChild(Node* newNode); //添加左孩子
    void setRChild(Node* newNode); //添加右孩子
    ~Node(); //释放结点
};

#endif //AVL_NODE_H
```

Node.cpp

```
//----Node.cpp----//
#include "Node.h"

Node::Node(string usn, string pwd) { //初始化用户信息
    user.username = usn;
    user.password = pwd;
}

userInfo& Node::getUser(){
    return user;
}

Node::~~Node() {
```

```
}

Node* Node::getLChild() {
    return lChild;
}

Node* Node::getRChild() {
    return rChild;
}

void Node::setLChild(Node *newNode) {
    lChild = newNode;
}

void Node::setRChild(Node *newNode) {
    rChild = newNode;
}

void Node::setBf(int bf) {
    this->bf = bf;
}

int Node::getBF() {
    return bf;
}
```

AVLTree.h

```
//----AVLTree.h----//
#ifndef AVL_AVLTREE_H
#define AVL_AVLTREE_H
#include <iostream>
#include <fstream>
#include <algorithm>
#include "Node.h"

class AVLTree {
private:
    Node *root = nullptr; //指向树根
    int depth = 0; //二叉树的深度
    bool sign = false; //记录操作是否成功

    // 一个数组，打印用，数组长度不低于二叉树的高度，设为一百
```

```

// 标记当前的节点是父节点的左孩子还是右孩子
int vec_left[100] = {0};

void printAVL(Node *node, int sign = 0); //打印二叉树
void deleteNode(Node *node, string usn, Node *p = nullptr); //删除结点
void add(Node *T, Node *newNode, Node *parent= nullptr); //添加结点
void updatePassword(string usn, string pwd, Node *T); //更新密码
void login(string usn, string pwd, Node *T); //登录
void save(ofstream &outFile, Node *T);

public:
    AVLTree(istream &inFile); //使用文件对象初始化二叉树
    ~AVLTree(); //二叉树的释放

    void save(ofstream &outFile); //将用户信息存入文件
    void print(); //封装后的打印函数

    bool getSign(); //判断操作是否成功
    void reSign(); //重置标志

    int getHeight(Node *T); //获取树的高度
    void deleteNode(string usn); //封装后的删除结点函数
    void add(string usn, string pwd); //封装后的添加结点函数
    void updatePassword(string usn, string pwd); //封装后的更新密码函数
    void login(string usn, string pwd); //封装后的登录函数

    void RRotate(Node *node, Node *parent); //右旋
    void LRotate(Node *node, Node *parent); //左旋
};

#endif //AVL_AVLTREE_H

```

AVLTree.cpp

```

//----AVLTree.cpp----//
#include "AVLTree.h"
AVLTree::AVLTree(istream &inFile) {
    string tempstr; //读取字符串
    int searchBound; //记录字符串分割标志"@"的位置
    Node *temp; //新结点指针

    //用户初始化
    while(inFile.peek() != EOF){ //检测文件是否到底
        //字符处理
    }
}

```

```

        inFile>>tempstr;
        searchBound = tempstr.find('@');

        //新建结点并初始化
        temp = new Node(tempstr.substr(0,searchBound), //获取用户名
                        tempstr.substr(searchBound+1,tempstr.length()-searchBound-1) //
获取密码
                        );

        //在树中插入新结点
        add(root, temp);
    }
    sign = false;
}

AVLTree::~~AVLTree() {

}

void AVLTree::printAVL(Node *node, int sign) {
    if(sign> 0){ //当树的高度大于 0 时
        for(int i = 0; i < sign - 1; i++){
            printf(vec_left[i] ? " | " : " ");
        }
        printf(vec_left[sign-1] ? " |—— " : " —— ");
    }

    if(! node){ //当结点不存在的时候
        printf("(null)\n");
        return;
    }

    //打印当前结点数据
    printf("%s\n", node->getUser().username.c_str());

    //当前结点为叶子结点时返回
    if(!node->getLChild() && !node->getRChild()){
        return;
    }

    vec_left[sign] = 1; //进入当前结点的右孩子
    printAVL(node->getRChild(), sign + 1);
    vec_left[sign] = 0; //进入当前结点的左孩子
    printAVL(node->getLChild(), sign + 1);
}

```

```
}

void AVLTree::print() { //封装后的打印函数
    printAVL(root);
}

void AVLTree::add(Node *T, Node *newNode, Node *parent) { //第一个参数是当前结点地址，
//作递归用，第二个参数是新结点地址
    //AVL 树是空树时，头节点指向加入的结点
    if(root == nullptr) {
        root = newNode;
        //获取树的深度
        depth = getHeight(root);
        sign = true;
        return;
    }

    //AVL 不为空时
    if(T->getUser().username > newNode->getUser().username){ //若当前结点值比新结点值
        大，则转向左孩子

        if(T->getLChild() == nullptr){
            T->setLChild(newNode); //左孩子为空，直接添加为左孩子
            sign = true;
            //获取树的深度
            depth = getHeight(root);
        } else add(T->getLChild(), newNode, T); //左孩子非空，转向左孩子

    } else if(T->getUser().username < newNode->getUser().username){ //若当前结点值比新结
        点值小，则转向右孩子

        if(T->getRChild() == nullptr){
            T->setRChild(newNode); //右孩子为空，直接添加为右孩子
            sign = true;
            //获取树的深度
            depth = getHeight(root);
        } else add(T->getRChild(), newNode, T); //右孩子非空，转向右孩子

    } else { //若当前结点值和新结点值一样大，则返回错误信息
```

```
        cout<<"用户\\"<<newNode->getUser().username<<"\n"已存在！"<<endl;
        delete newNode; //释放空间
        return;
    }

    //平衡操作
    if(T->getBF() > 1){ //找到最小不平衡子树
        RRotate(T, parent); //右旋
    }else if(T->getBF() < -1){
        LRotate(T, parent); //左旋
    }
}

/*
 * 由于获取树的深度时需要得到每个结点左右子树的深度
 * 所以可以顺便计算出当前结点的平衡因子
 */
int AVLTree::getHeight(Node *T) { //使用递归获取树的深度
    if(T == nullptr) return 0;

    int left, right; //记录左右子树深度

    left = getHeight(T->getLChild()); //当前结点左子树深度
    right = getHeight(T->getRChild()); //当前结点右子树深度
    T->setBf(left-right); //计算出当前结点的平衡因子

    return max(left, right)+1; //取左右子树中最深的一个作为返回值
}

void AVLTree::deleteNode(Node *node, string usn, Node *p) {
    if(node == nullptr){
        //cout<<"empty node!"<<endl;
        return;
    }

    if(node->getUser().username == usn){ //若当前结点即为所寻结点

        if(node->getRChild() && node->getLChild()){//如果有两个孩子结点

            Node* temPreNode = node; //指向中序后继结点的双亲结点
            Node* temNode = node->getRChild(); //指向中序后继结点
            while(temNode->getLChild() != nullptr){ //遍历到右子树最左下结点
```



```

        temPreNode = temNode;
        temNode = temNode->getLChild();
    }

    if(node->getRChild() != temNode){ //当中序后继结点不是目标结点的右孩子
时
        if(temNode->getRChild() != nullptr){ //若中序后继节点右孩子不为空
            temPreNode->setLChild(temNode->getRChild()); //中序后继结点的
右孩子变为前一个结点的左孩子
        }
        temNode->setRChild(node->getRChild()); //被删除结点的右孩子作为中
序后继节点的右孩子
        temPreNode->setLChild(nullptr); //中序后继结点的双亲结点左孩子置空
    }
    temNode->setLChild(node->getLChild()); //被删除结点的左孩子作为中序后
继结点的左孩子

    if(p != nullptr){
        if(p->getRChild() == node) { //是结点的右孩子时
            p->setRChild(temNode); //中序后继节点作为双亲结点的右孩子
        } else if(p->getLChild() == node){ //是结点的左孩子时
            p->setLChild(temNode); //中序后继节点作为双亲结点的左孩子
        }
    } else{
        root = temNode;
    }
    delete node; //删除本结点

} else if(node->getRChild() && !node->getLChild()){ //如果当前结点只有右孩子

    if(p == nullptr) { //考虑目标结点为根节点的情形
        root = node->getRChild(); //设置根节点
        delete node; //删除结点
    } else if(p->getLChild() == node){ //若为父节点的左孩子
        p->setLChild(node->getRChild());
        delete node; //删除结点
    } else if(p->getRChild() == node){ //若为父节点的右孩子
        p->setRChild(node->getRChild());
        delete node; //删除结点
    }
}

} else if(!node->getRChild() && node->getLChild()){ //如果当前结点只有左孩子

    if(p == nullptr) { //考虑目标结点为根节点的情形

```

```
        root = node->getLChild(); //设置根节点
        delete node; //删除结点
    }else if(p->getLChild() == node){ //若为父节点的左孩子
        p->setLChild(node->getLChild());
        delete node; //删除结点
    }else if(p->getRChild() == node){ //若为父节点的右孩子
        p->setRChild(node->getLChild());
        delete node; //删除结点
    }
}

} else{ //若为叶子结点，则直接删除

    if(p == nullptr){ //如果目标为孤立的根节点，则直接删除
        delete node;
        root = nullptr;
    }else if(p->getLChild() == node){ //若为父节点的左孩子
        delete node; //删除结点
        p->setLChild(nullptr); //父节点左孩子指针置空
    }else if(p->getRChild() == node){ //若为父节点的右孩子
        delete node; //删除结点
        p->setRChild(nullptr); //父节点右孩子指针置空
    }

}

node = nullptr;
//重新计算高度
depth = getHeight(root);
sign = true;
return;
}

deleteNode(node->getLChild(), usn, node);

//原结点位置平衡操作
while(node->getLChild() != nullptr && node->getLChild()->getBF() > 1) {
    RRotate(node, p); //右旋
}

//在左支删除时的上层平衡操作
if(node->getBF() > 1){ //找到最小不平衡子树
    RRotate(node, p); //右旋
}else if(node->getBF() < -1){
    LRotate(node, p); //左旋
}
```

```
deleteNode(node->getRChild(), usn, node);

//原结点位置平衡操作
while(node->getRChild() != nullptr && node->getRChild()->getBF() > 1) {
    RRotate(node, p); //右旋
}

//在右支删除时的上层平衡操作
if(node->getBF() > 1){ //找到最小不平衡子树
    RRotate(node, p); //右旋
}else if(node->getBF() < -1){
    LRotate(node, p); //左旋
}
}

void AVLTree::deleteNode(string usn) {
    deleteNode(root, usn);
}

void AVLTree::RRotate(Node *node, Node *parent) {
    if(node == root){ //若最小不平衡子树根结点为根节点
        if(node->getLChild()->getBF() < 0){ //如果符号不同，则子结点先进行左旋
            LRotate(node->getLChild(), node);
        }
        Node *temNode = root;
        root = root->getLChild();

        if(root->getRChild() == nullptr){ //如果孩子结点没有右孩子
            root->setRChild(temNode);
            temNode->setLChild(nullptr);
        }else{ //如果孩子结点有右孩子，则将孩子结点的右孩子移到双亲结点的左下角
            temNode->setLChild(root->getRChild());
            root->setRChild(temNode);
        }
    }

    } else{ //若不为根节点
        if(node->getLChild()->getBF() < 0){ //如果符号不同，则子结点先进行左旋
            if(parent->getLChild() == node){ //左边的右旋，双亲结点走左边
                LRotate(node->getLChild(), parent->getLChild());
            }else{ //右边的右旋，双亲结点走右边
                LRotate(node->getLChild(), parent->getRChild());
            }
        }
    }
}
```

```

//旋转操作
Node *temNode = node;
Node *childNode = node->getLChild();

if(parent->getLChild() == node){ //左边的右旋
    parent->setLChild(childNode);
}else{ //右边的右旋
    parent->setRChild(childNode);
}

if(childNode->getRChild() == nullptr){ //如果孩子结点没有右孩子
    childNode->setRChild(temNode);
    temNode->setLChild(nullptr);
}else{ //如果孩子结点有右孩子结点，则将孩子结点的右孩子移到目标结点的左
下角，进行右旋操作
    temNode->setLChild(childNode->getRChild());
    childNode->setRChild(temNode);
}

}
depth = getHeight(root); //重新计算树的深度和 bf
}

void AVLTree::LRotate(Node *node, Node *parent) {
    if(node == root){ //若最小不平衡子树根结点为根节点
        if (node->getRChild()->getBF() > 0) { //如果符号不同，则子结点先进行右旋
            RRotate(node->getRChild(), node);
        }
        Node *temNode = root;
        root = root->getRChild();

        if(root->getLChild() == nullptr){ //如果孩子结点没有左孩子
            root->setLChild(temNode);
            temNode->setRChild(nullptr);
        }else{ //如果孩子结点有两个孩子结点，则将孩子结点的左孩子移到双亲结点的
右下角
            temNode->setRChild(root->getLChild());
            root->setLChild(temNode);
        }

    } else { //若不为根节点
        if (node->getRChild()->getBF() > 0) { //如果符号不同，则子结点先进行右旋
            if(parent->getLChild() == node){ //左边的右旋，双亲结点走左边

```

```
        RRotate(node->getRChild(), parent->getLChild());
    }else{ //右边的左旋，双亲结点走右边
        RRotate(node->getRChild(), parent->getRChild());
    }
}
//旋转操作
Node *temNode = node;
Node *childNode = node->getRChild();

if(parent->getLChild() == node){ //左边的左旋
    parent->setLChild(childNode);
}else{ //右边左旋
    parent->setRChild(childNode);
}

if(childNode->getLChild() == nullptr){ //如果孩子结点没有左孩子
    childNode->setLChild(temNode);
    temNode->setRChild(nullptr);
}else{ //如果孩子结点有两个孩子结点，则将孩子结点的左孩子移到双亲结点的
右下角
    temNode->setRChild(childNode->getLChild());
    childNode->setLChild(temNode);
}

}
depth = getHeight(root); //重新计算树的深度和 bf
}

void AVLTree::add(string usn, string pwd) {
    add(root, new Node(usn, pwd));
}

void AVLTree::updatePassword(string usn, string pwd, Node *T) {
    if(T == nullptr) return;

    updatePassword(usn, pwd, T->getLChild());

    if(usn == T->getUser().username && pwd == T->getUser().password){
        cout << "        新密码:";
        string p;
        cin >> p;
        T->getUser().password = p;
        sign = true;
        return;
    }
}
```

```
    }

    updatePassword(usn, pwd, T->getRChild());
}

void AVLTree::updatePassword(string usn, string pwd) {
    updatePassword(usn, pwd, root);
}

void AVLTree::save(ofstream &outFile) {
    save(outFile, root);
}

void AVLTree::login(string usn, string pwd, Node *T) {
    if(T == nullptr) return;

    login(usn, pwd, T->getLChild());

    if(usn == T->getUser().username && pwd == T->getUser().password){
        sign = true;
    }
    login(usn, pwd, T->getRChild());
}

void AVLTree::login(string usn, string pwd) {
    login(usn, pwd, root);
}

bool AVLTree::getSign() {
    return sign;
}

void AVLTree::reSign() {
    sign = false;
}

void AVLTree::save(ofstream &outFile, Node *T) {
    if(T == nullptr) return;
    if(T == root) outFile<<T->getUser().username<<'@'<<T->getUser().password;
    else outFile<<endl<<T->getUser().username<<'@'<<T->getUser().password;
    save(outFile, T->getLChild());
    save(outFile, T->getRChild());
}
```