浙江工业大学

数据结构课程设计

2017/2018(1)

实验题目：用户登录系统的模拟

学生姓名：杨振华

学生学号：201626811225

学生班级：软件工程 1603

任课教师：张晖

计算机科学与技术学院、软件学院

目录

一、 实验题目及要求

1. 实验题目

用户登录系统的模拟。

2. 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。

3. 基本要求

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。

（1） 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。

（2） 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。

（3） 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。

（4） 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。

（5） 要求采用 Visual C++ 6.0 及以上版本进行调试。

二、 设计思路

1. 系统总体设计

本题要求有效地组织信息，使用一定的数据结构从而高效地完成数据的增加、查询、更新、删除等工作（即所谓的 CRUD 操作）。在这里使用 AVL 树这种数据结构存储数据是适合、恰当的。

AVL 树是一种自平衡查找树，其特点为根据四种旋转操作，使其中任何节点的两个子树的高度最大差别为 1。得益于这种优良特性，AVL 树的查找、插入和删除在平均和最坏情况下的时间复杂度都是$O(\log n)$，远远优于线性结构。

确定数据结构之后，完成本题只需要实现 AVL 树这种数据结构，并实现一定的界面调用数据结构的 API，完成题中的要求即可。相对于传统的控制台界面，由于图形用户界面具有对用户友好、表现力强等的优点，本题复杂的树图形和其旋转操作适宜采用图形用户界面进行展示。这里最终采用了 C++编写的 Qt 框架进行图形用户界面的编程。

2. 系统功能设计

如题中所述，系统的主体功能包括从文件中读取用户信息、保存用户信息到文件、用户登录、添加新用户、删除老用户、更新用户密码等。功能的具体描述如下：

（1） 从文件读取用户信息：首先判断被读取的文件是否存在，若不存在则提示文件不存在。若文件存在，按行读入每个用户的用户名和密码，存入 AVL 树（若读取之前 AVL 树不为空，析构并新建一棵空树）。若每条信息均完整，操作完成后提示操作成功，否则提示操作成功但部分数据不完整。最后重新渲染 AVL 树视图。

（2） 保存用户信息到文件：先判断文件路径是否合法，不合法则给出相应提示，合法则新建空白文件，并层次遍历 AVL 树，将节点信息按行存入文件，提示操作成功。

（3） 用户登录：用户输入用户名和密码，根据用户名在 AVL 树中查找用户的密码。若查找到的密码和用户输入的匹配，提示登录成功，否则提示相应的错误。

（4） 添加新用户：用户输入用户名和密码，先根据用户名在 AVL 树中查找用户。用户不存在则插入节点，提示添加成功，否则提示用户已存在，添加失败。最后重新渲染 AVL 树视图。

（5） 删除老用户：用户输入用户名，在 AVL 树中查找用户。用户不存在则提示删除失败，否则删除用户并提示删除成功。最后重新渲染 AVL 树视图。

（6） 更新用户密码：用户输入用户名和新密码，根据用户名在 AVL 树中查找用户。用户不存在则插入节点，提示添加成功提示更新失败，否则更新密码并提示更新成功。

略的功能流程图如图 1。

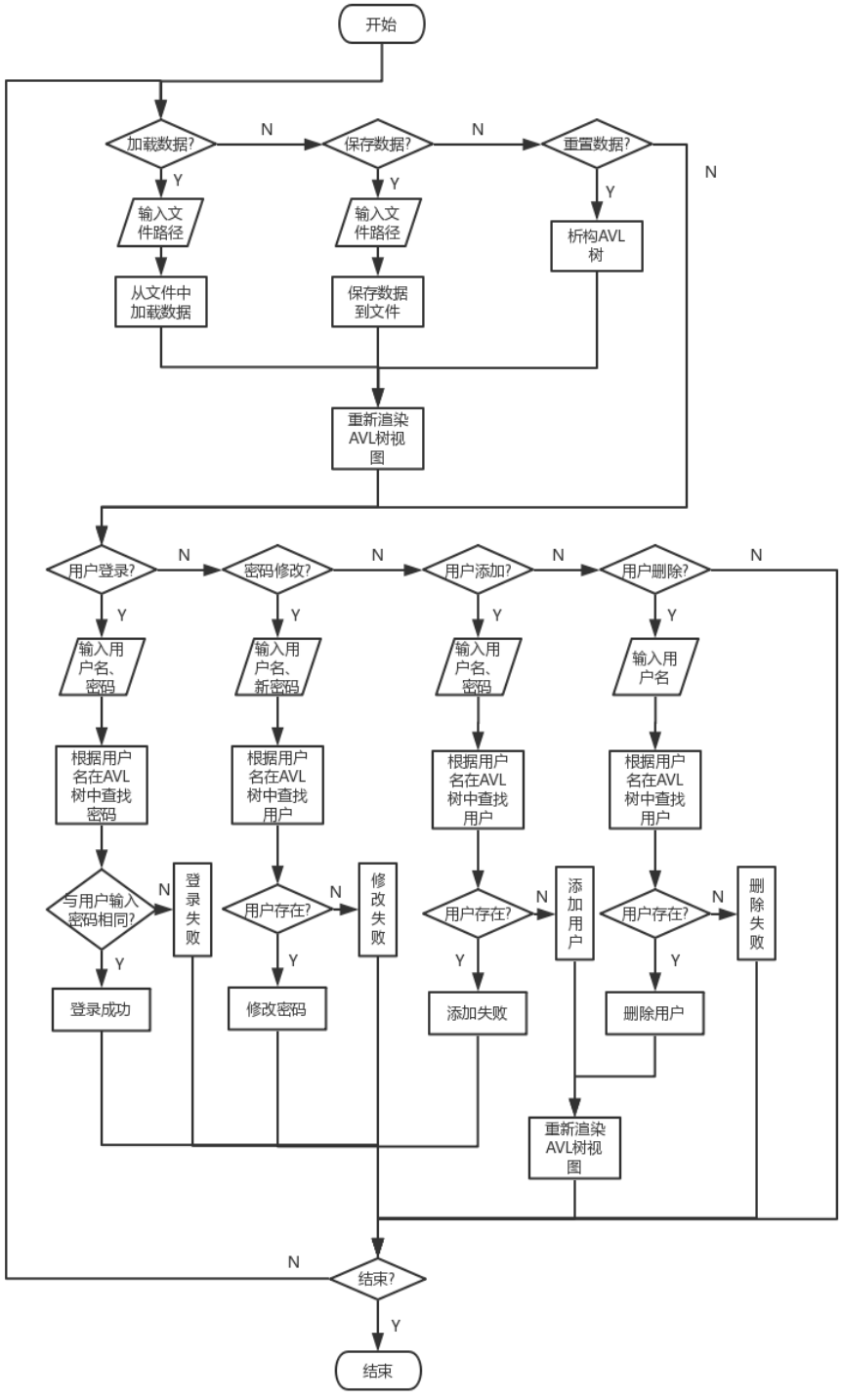

图 1 系统功能流程图

3. 类的设计

程序主要包括四个类和一个枚举类型。其中 Node 模板类是 AVL 树的节点类，AVLTree 类为 AVL 树的主类，用于实现各种 AVL 树相关操作，两个继承于 QWidget（Qt GUI 框架的窗体基类）的类 MainWidget、TreeViewer 分别用于实现用户操

作界面和 AVL 树展示界面。枚举类型 OperationState 用于指示操作状态（成功、失败）。UML 图由图 2 给出。

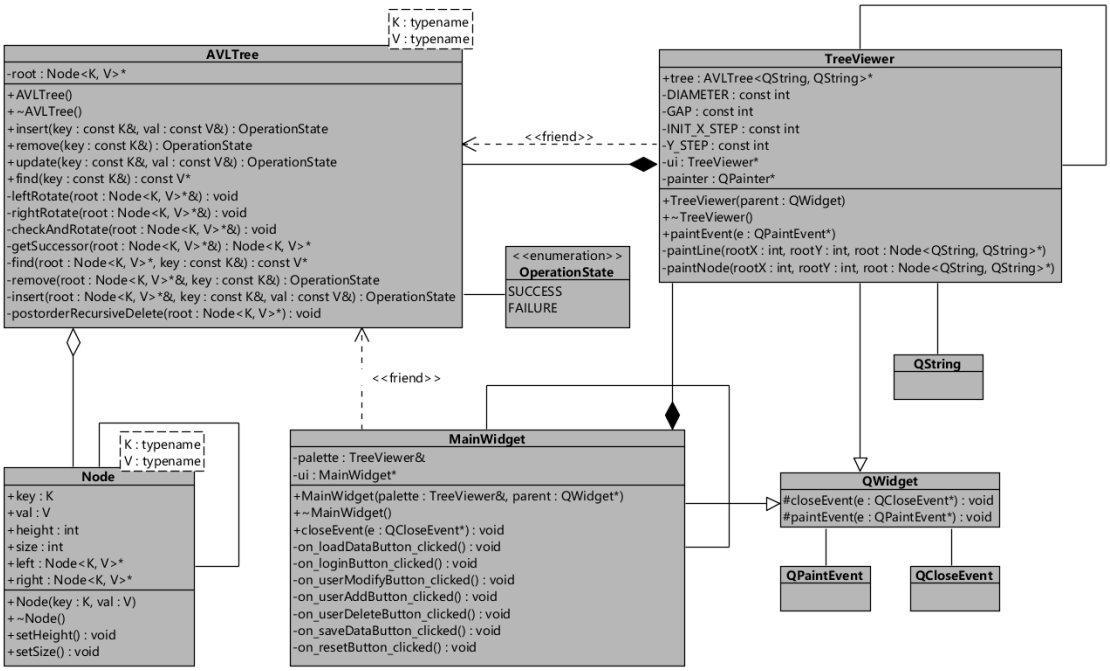

图 2 程序中类的 UML 图

由于程序重点在于 AVL 树相关操作，在此将对 Node 类和 AVLTree 类进行展开分析。

Node 类中的重点函数是 setHeight()，用于在对节点进行操作之后更新节点的高度（保存在节点之中）。易知某节点高度等于其左右子树高度最大值加一。具体代码如下：

```
template <typename K, typename V>
void Node<K, V>::setHeight()
{
    height = 1 + std::max(left == nullptr ? 0 : left->height,
                          right == nullptr ? 0 : right->height);
}
```

AVLTree 类较复杂，包含操作较多。由于 AVL 树操作过程中大量用到递归操作，但递归函数不能直接暴露给用户，这里采取了递归函数私有化，对递归函数同时封装一个非递归版本的函数给用户使用的方法，比如：

```
//真正操作的版本,private
OperationState insert(Node<K, V>*& root, const K& key, const V& val);


//暴露给用户的封装版本,public
OperationState insert(const K& key, const V& val);
```

这里在分析时将只分析递归版本。

由于 AVL 树的操作中大量用到旋转操作，因此将旋转操作单独封装起来实现了 DRY（Don’t Repeat Yourself）的编程原则。尽管旋转操作有四种（分别为

LL、RR、LR、RL）,但每种操作均由左旋和右旋两种基本操作构成。首先实现这两种基本操作。

左旋：

```
template <typename K, typename V>
void AVLTree<K, V>::leftRotate(Node<K, V>*& root)
{
    Node<K, V>* previousRoot = root;
    Node<K, V>* tmp = root->right->left;
    root->right->left = root;
    root = root->right;
    root->left->right = tmp;
    previousRoot->setHeight();  //每次旋转之后重新计算节点高度,下同
    previousRoot->setSize();
    root->setHeight();
    root->setSize();
}
```

右旋：

```
template <typename K, typename V>
void AVLTree<K, V>::rightRotate(Node<K, V>*& root)
{
    Node<K, V>* previousRoot = root;
    Node<K, V>* tmp = root->left->right;
    root->left->right = root;
    root = root->left;
    root->right->left = tmp;
    previousRoot->setHeight();
    previousRoot->setSize();
    root->setHeight();
    root->setSize();
}
```

当对原来平衡的 AVL 树的节点进行增、删等操作后，就要从操作位置往树根回溯，逐一检查路径中的节点是否平衡，若不平衡就要进行相关旋转操作来使 AVL 树重新回到平衡状态。图 3 给出了检查节点是否平衡并进行平衡操作的流程。

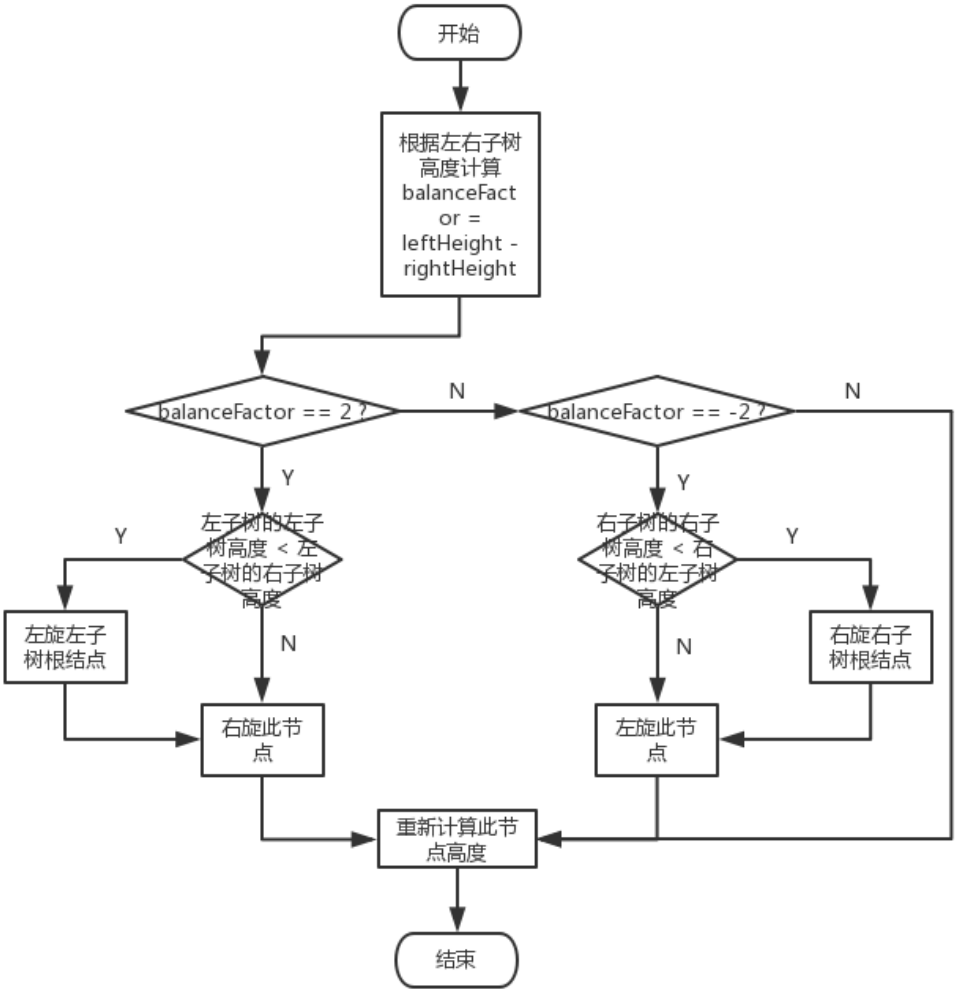

图 3 检查节点平衡状态、进行平衡操作流程图

有了基础的旋转操作，就可以对 AVL 树进行相关的 CRUD 操作并且保持树平衡了。查找操作只需要简单递归查找即可，而修改操作基于查找进行，均不在此赘述。插入操作与查找类似，需要注意的是插入后更新相关节点的高度。代码如下：

```
template <typename K, typename V>
OperationState AVLTree<K, V>::insert(Node<K, V>*& root, const K& key,
                                const V& val)
{
    if (root == nullptr) {
        root = new Node<K, V>(key, val);
    } else if ((root->key > key
              && insert(root->left, key, val) == FAILURE)
              || (root->key < key
              && insert(root->right, key, val) == FAILURE)
```

```
            || (root-> key == key)) {
        return FAILURE;
    }

    root->setSize();
    root->setHeight();
    checkAndRotate(root);  //调平，下同

    return SUCCESS;
}
```

删操作也需要注意调平 AVL 树，除此之外，还需要考虑多种情况。尤其需要注意的是，在被删除节点的左右子树均不为空的情况下，需要使用其左子树的最大节点或其右子树的最小节点代替该节点。为了实现这种操作，在这里特地封装了一个函数，用于删除右子树的最小节点并将其返回，代码如下：

```
template <typename K, typename V>
Node<K, V>* AVLTree<K, V>::getSuccessor(Node<K, V>*& root)
{
    Node<K, V>* tmp = root;
    if(root->left == nullptr) {
        root = tmp->right;
        tmp->right = nullptr;
    } else {
        tmp = getSuccessor(root->left);
    }

    if(root != nullptr) {
        root->setHeight();
        root->setSize();
    }

    return tmp;
}
```

体的删除节点的代码：

```
template <typename K, typename V>
OperationState AVLTree<K, V>::remove(Node<K, V>*& root, const K& key)
{
    if (root == nullptr) {
        return FAILURE;
    }

    if ((root->key > key && remove(root->left, key) == FAILURE)
```

```
        || (root->key < key && remove(root->right, key) == FAILURE)) {
        return FAILURE;
    } else if(root->key == key){
        if (root->left == nullptr && root->right == nullptr) {  //叶子
            delete root;
            root = nullptr;
        } else if (root->left == nullptr) {  //只有右子树
            Node<K, V>* tmp = root->right;
            delete root;
            root = tmp;
        } else if (root->right == nullptr) {  //只有左子树
            Node<K, V>* tmp = root->left;
            delete root;
            root = tmp;
        } else {  //有左、右子树
            Node<K, V>* successor = getSuccessor(root->right);
            Node<K, V>* leftBackup = root->left;
            Node<K, V>* rightBackup = root->right;

            delete root;
            root = successor;
            root->left = leftBackup;
            root->right = rightBackup;
        }
    }

    if(root != nullptr) {
        root->setSize();
        root->setHeight();
        checkAndRotate(root);
    }

    return SUCCESS;
}
```

4. 主程序的设计

由于采用了图形用户界面，此处简要分析主界面的结构和功能实现。

程序由主界面和 AVL 树视图界面构成。主界面主要包含文件名编辑器、加载数据按钮、保存数据按钮、重置 AVL 树视图按钮、用户名编辑器、密码编辑器、用户登录按钮、密码修改按钮、用户添加按钮、用户删除按钮、操作状态标签。AVL 树视图为空窗体，用于绘制 AVL 树的图形。图 4 和图 5 分别给出了两个窗体的界面。

图 4 主窗体

图 5 AVL 树视图窗体

由于主窗体对象内含有 AVL 树视图对象的指针，而 AVL 树视图对象内含有 AVLTree 对象的指针，当主窗体上相应按钮被按下，只需要调用 AVLTree 对象内的相关函数并调用 AVL 树视图对象的 update()函数（来源于对 QWidget 内纯虚函数的实现，用于重绘画面）即可。接下来相应操作就会完成，同时 AVL 树视图中的画面会得到更新。具体功能已在系统功能中进行分析。

三、 调试分析

1. 技术难点分析

（1） AVL 树旋转等操作复杂，涉及指针操作多，实现的难度大。

（2） 使用图形界面绘制 AVL 树时为了避免树枝之间互相交叉，需要合适的节点间距离。考虑到二叉树本身的性质，在这里，层与层之间的节点间距离变化采用指数增长的方式较为恰当。

2. 调试错误分析

（1） 编码错误，在实现查找函数时返回了临时变量的引用，代码如图 6。

```
template <typename K, typename V>
V* const& AVLTree<K, V>::find(Node<K, V>* const& root, const K& key)
{
    if (root == nullptr) {
        return nullptr;
    }

    if (root->key == key) {
        return &(root->val);
    } else if (root->key > key) {
        return find(root->left, key);
    } else {
        return find(root->right, key);
    }
}
```

图 6 返回临时变量引用的代码

该代码在运行时产生的错误如图 7。

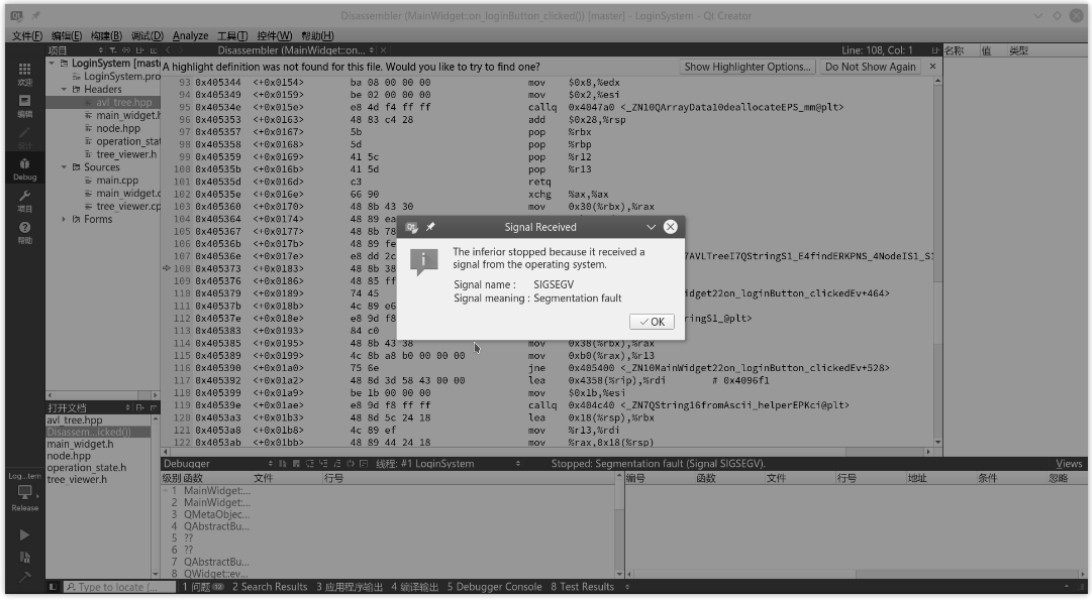

图 7 返回临时变量引用的函数在运行时报错

后修复了此问题，返回值改为非引用，正确代码如图 8，正常运行结果如图 9。

```
template <typename K, typename V>
const V* AVLTree<K, V>::find(Node<K, V>* const& root, const K& key)
{
    if (root == nullptr) {
        return nullptr;
    }

    if (root->key == key) {
        return &(root->val);  //返回临时变量，因而返回值不能为引用
    } else if (root->key > key) {
        return find(root->left, key);
    } else {
        return find(root->right, key);
    }
}
```

图 8 修复返回值为引用的问题的代码

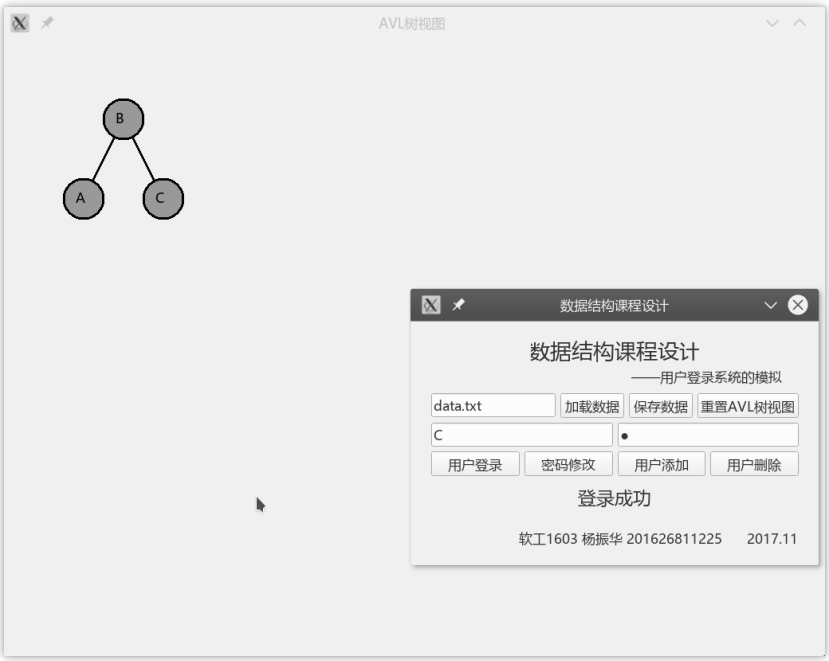

图 9 查找功能正常运行

（2） 逻辑错误，未备份指针。在前面的分析中已经提到，在删除左右子树均非空的节点时需要取右子树最左边的节点代替该节点。在代替操作之前，应当对原节点的左右子树的指针进行备份。起初，这一点被忽略了，造成了如图 11 的错误（图 10 为删除之前的状态）。

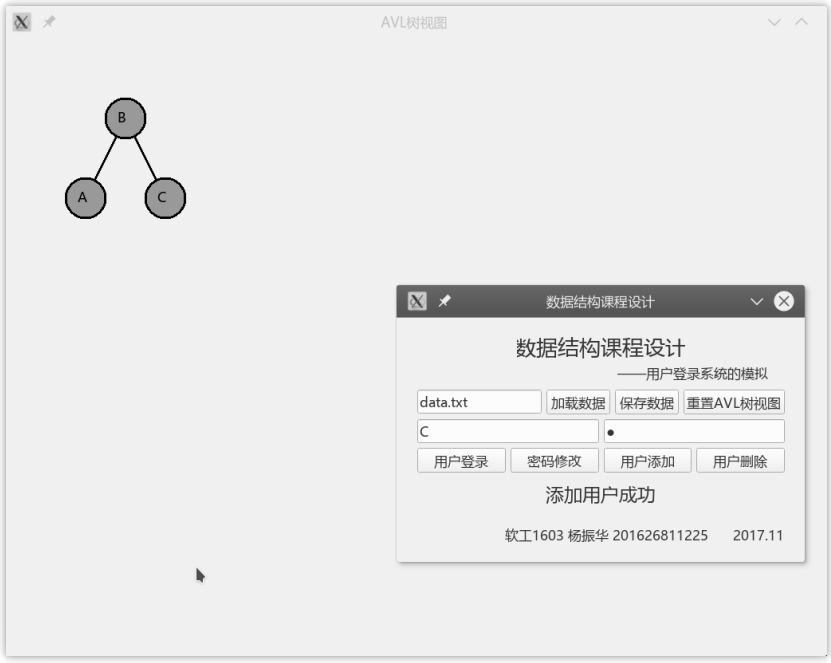

图 10 删除 B 节点之前

图 11 删除 B 节点，逻辑错误

后对两个指针进行备份，并赋值给新节点的相应指针域，代码如图 12，正确的运行结果如图 13。

```
} else {  //有左、右子树
    Node<K, V>* successor = getSuccessor(root->right);
    Node<K, V>* leftBackup = root->left;
    Node<K, V>* rightBackup = root->right;

    delete root;
    root = successor;
    root->left = leftBackup;
    root->right = rightBackup;
}
```

图 12 备份左右子树的指针代码

图 13 删除 B 节点，正确运行结果

（3） 逻辑错误，未验证保存数据到文件时用户输入的文件名有效，错误的运行结果如图 14。

图 14 用户输入的文件名非法导致错误

修复此问题，只需验证文件打开正确即可。代码如图 15，正确运行结果如图 15。

```
if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
    ui->operationStatusLabel->setText("文件打开失败");
    return;
}
```

图 15 验证文件打开方式正确的代码

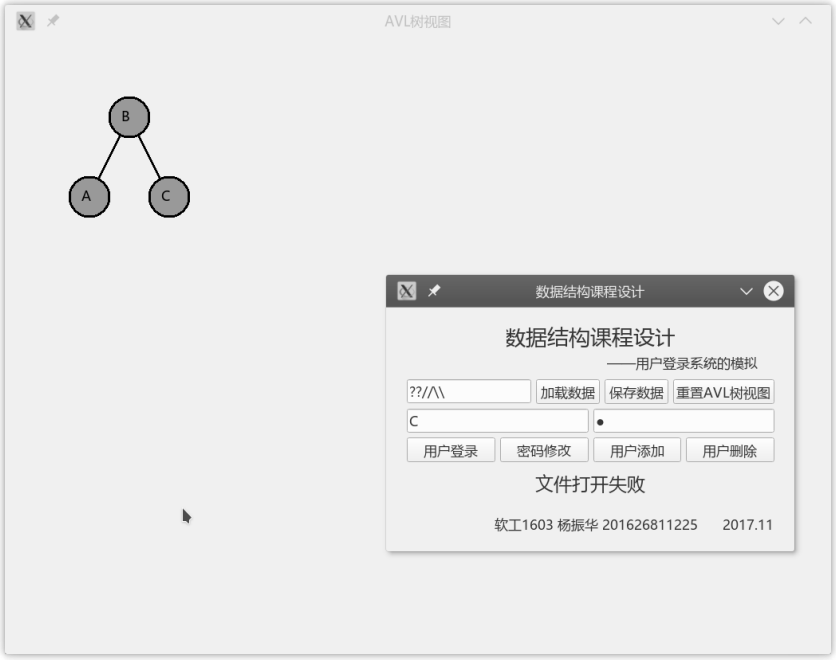

图 16 对非法文件名提示文件打开失败

四、 测试结果分析

1. 文件保存测试

功的测试结果如图 17、18。

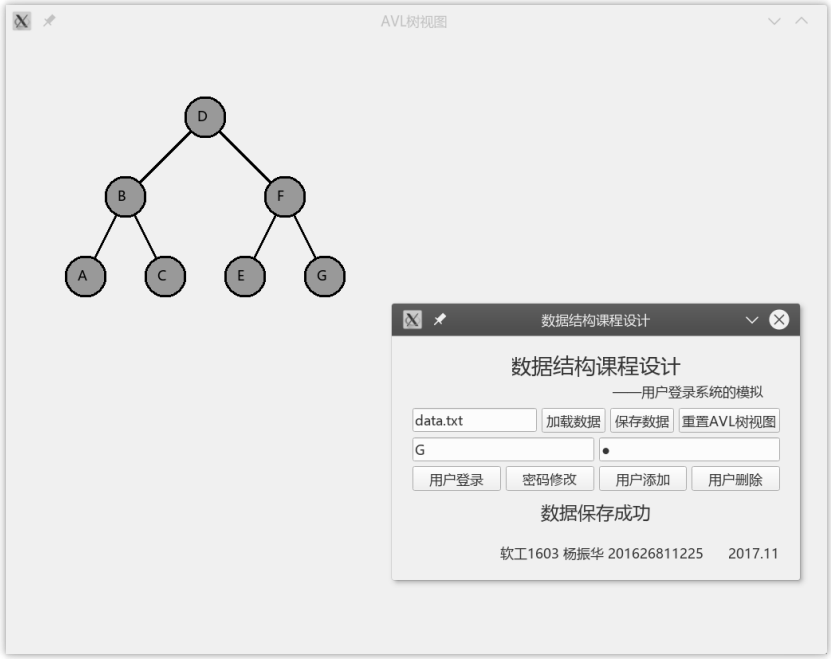

图 17 保存成功

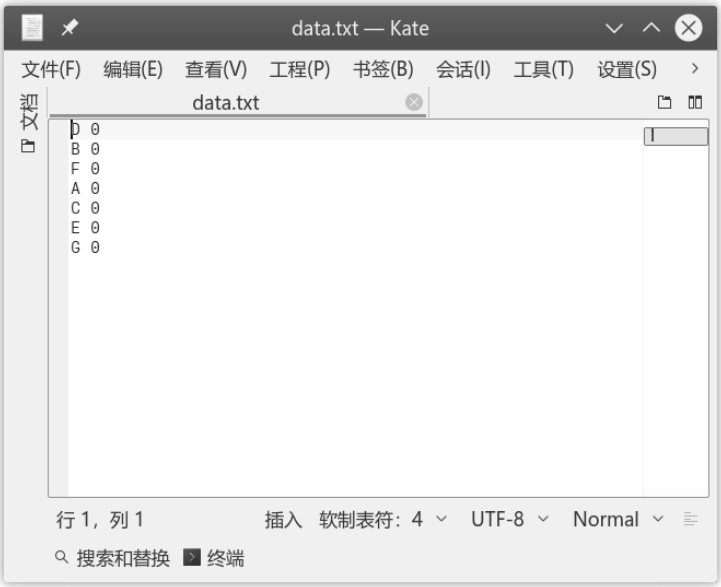

图 18 保存成的文件

若当前 AVL 树为空，则不会保存，并提示相关信息，如图 19。

图 19 无数据可供保存

2. 重置 AVL 树视图测试

图 20，析构整棵 AVL 树。

图 20 重置 AVL 树视图成功

3. 加载数据测试

加载之前保存的 data.txt，如图 21。

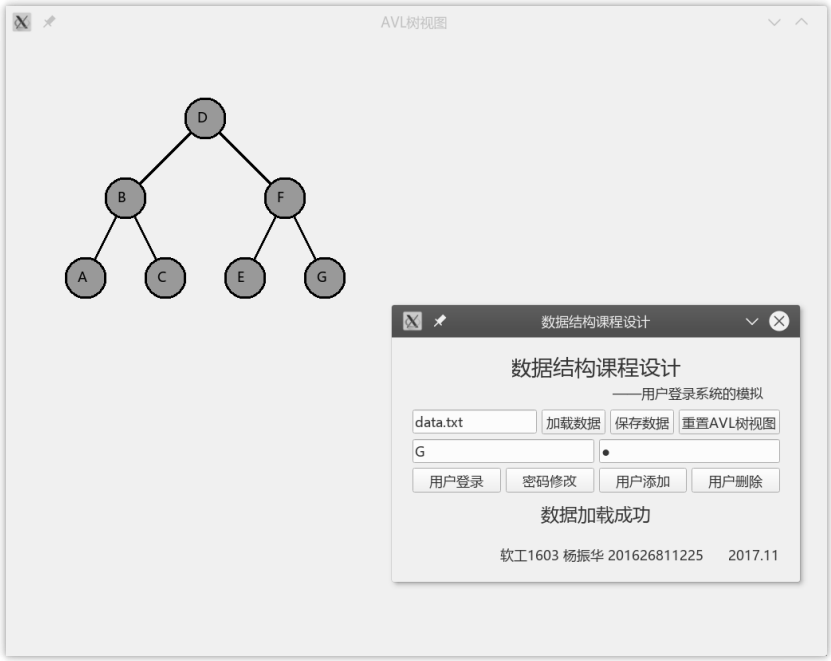

图 21 数据加载成功

若这里输入的文件不存在，将会提示文件不存在，如图 22。

图 22 文件打开失败

4. 用户添加测试

这里将展开四种旋转方式（密码均为 0）。
LL（图 23、图 24）。

图 23 LL 旋转测试，插入 C 之前

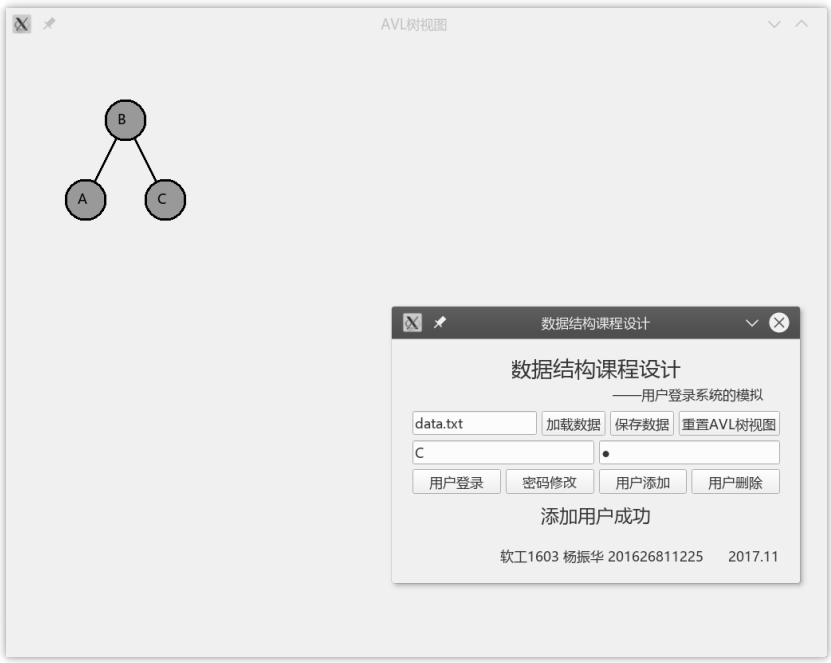

图 24 LL 旋转测试，插入 C 之后

RR（图 25、图 26）。

图 25 RR 旋转测试，插入 A 之前

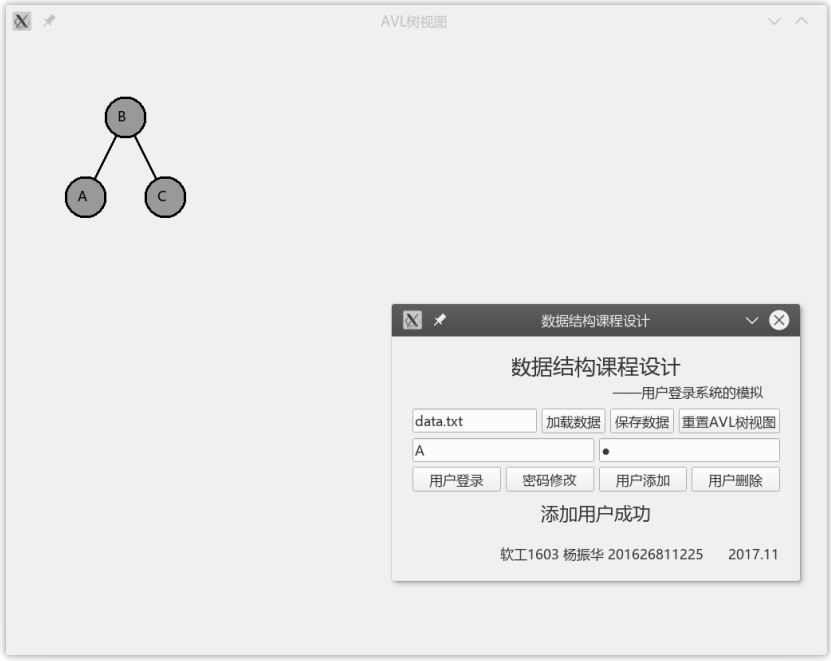

图 26 RR 旋转测试，插入 A 之后

LR（图 27、图 28）。

图 27 LR 旋转测试，插入 B 之前

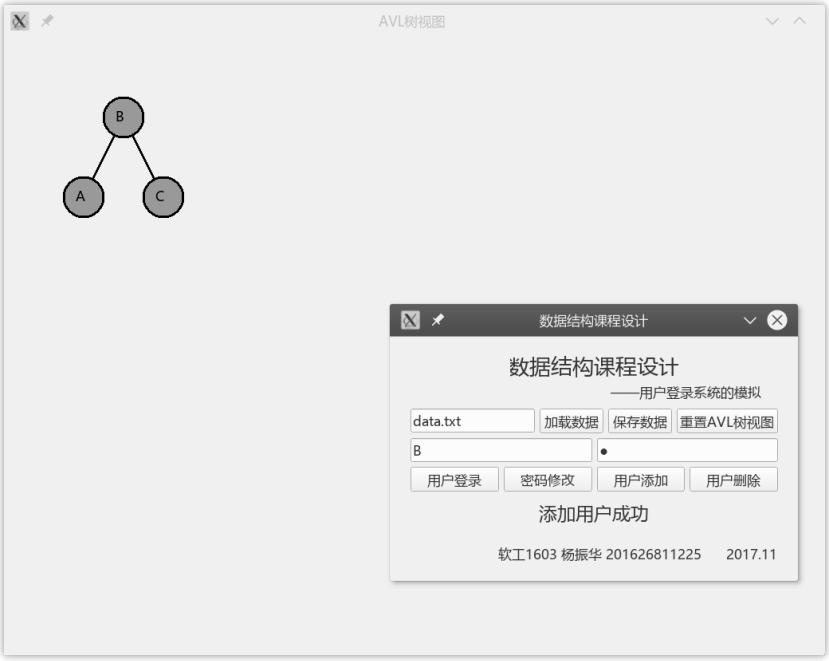

图 28 LR 旋转测试，插入 B 之后

RL（图 29、图 30）。

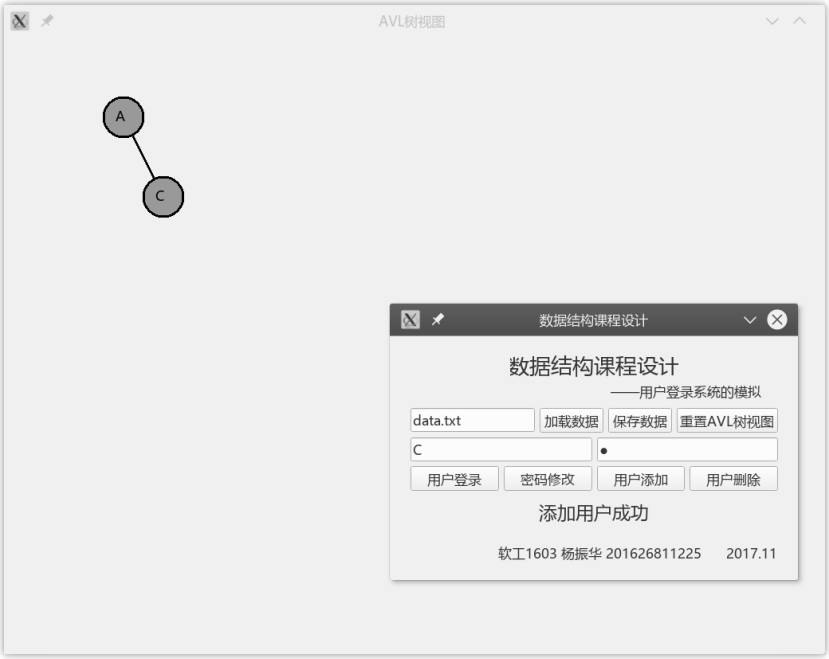

图 29 RL 旋转测试，插入 B 之前

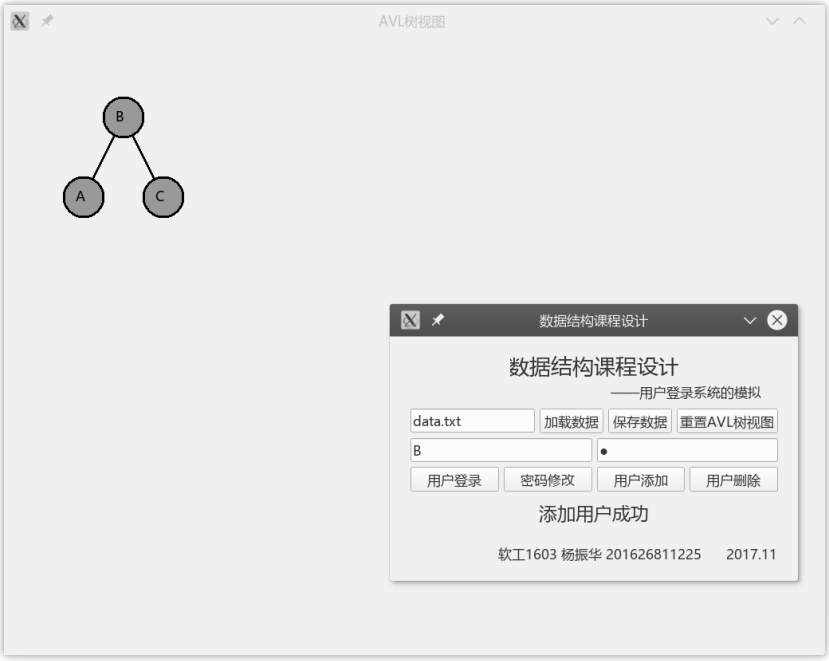

图 30 RL 旋转测试，插入 B 之后

除此之外，若试图添加的用户原来就在 AVL 树中，将会添加失败并提示错误信息（图 31）。

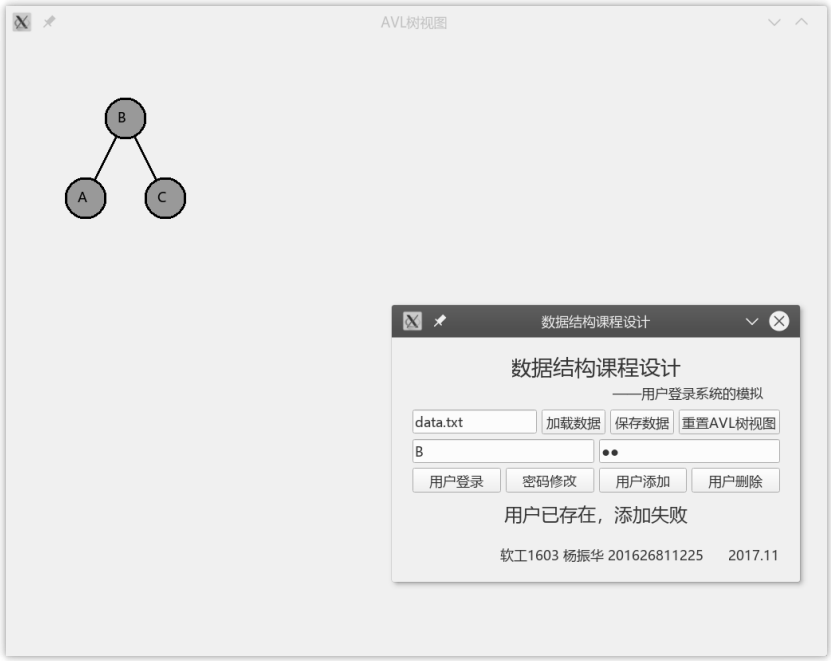

图 31 试图添加已存在的用户

5. 用户删除测试

若试图删除的用户不存在于 AVL 树中，则删除失败并提示错误信息（图 32）。

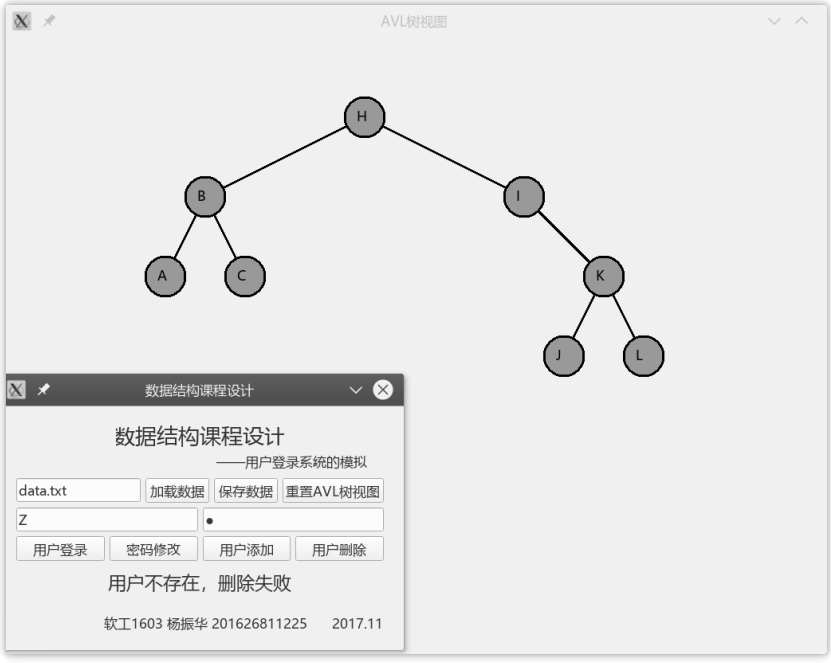

图 32 删除失败

若用户存在于 AVL 树中，则删除成功，此处展开为三种情况。
设删除之前的 AVL 树如图 33。

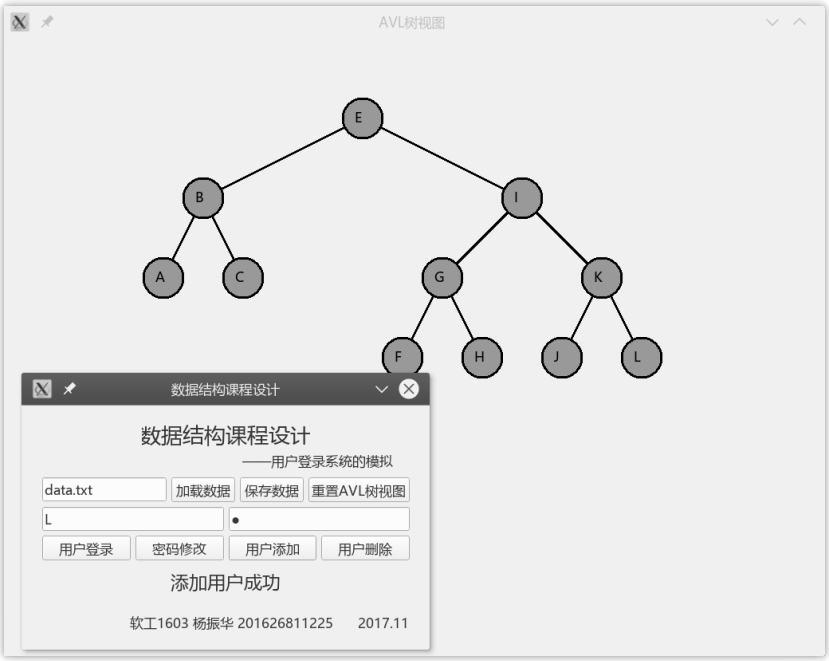

图 33 删除节点之前

删除叶子节点（图 34）。

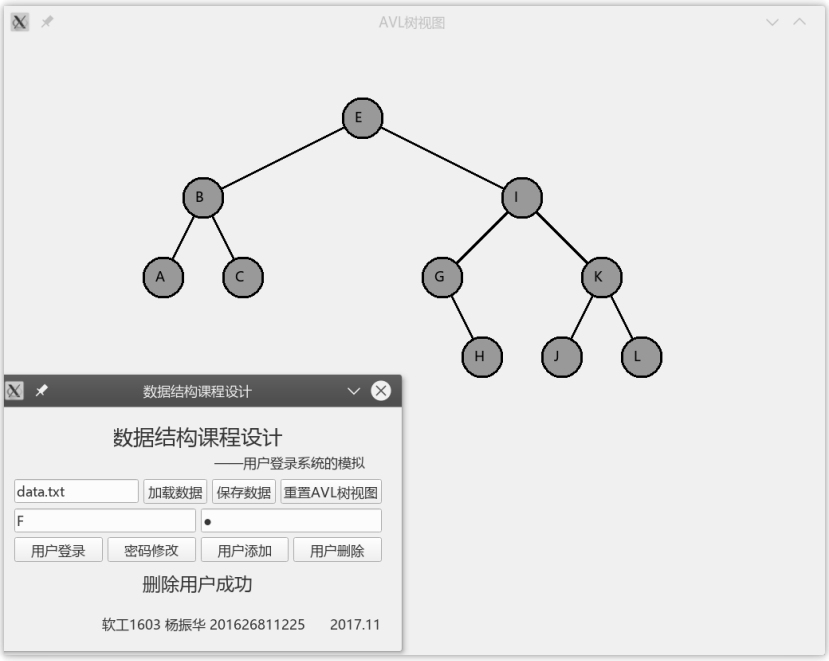

图 34 删除叶子结点 F

在上一步基础上删除有一个儿子的节点（图 35）。

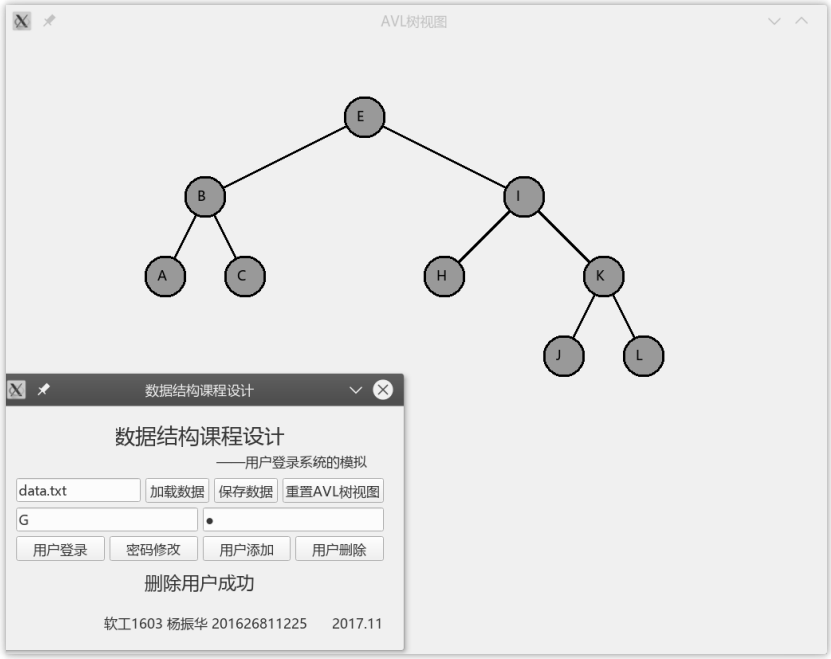

图 35 删除有一个儿子的节点 G

在上一步的基础上删除有两个儿子的节点（图 36）。

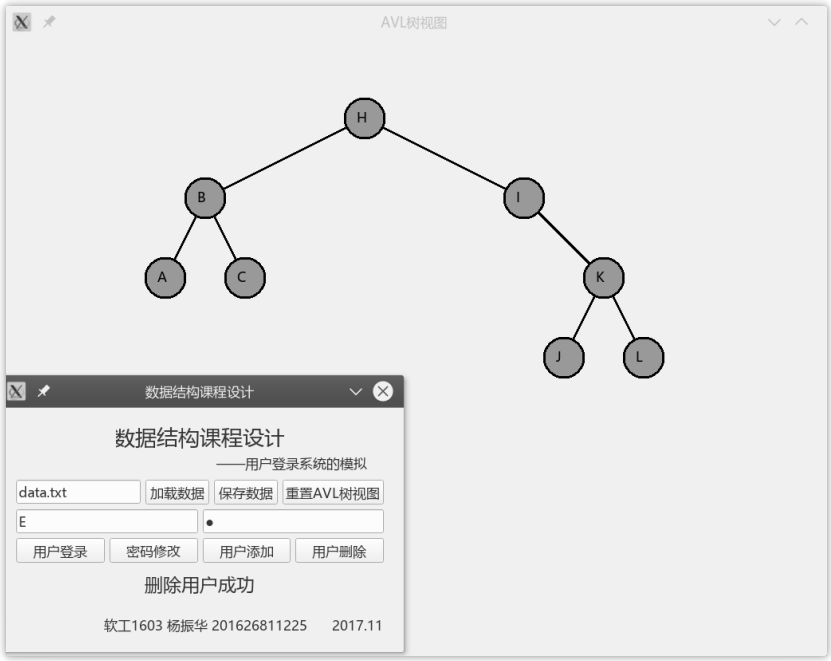

图 36 删除有两个儿子的节点 E

6. 密码修改测试

用户存在于 AVL 树中，修改成功（图 37）。

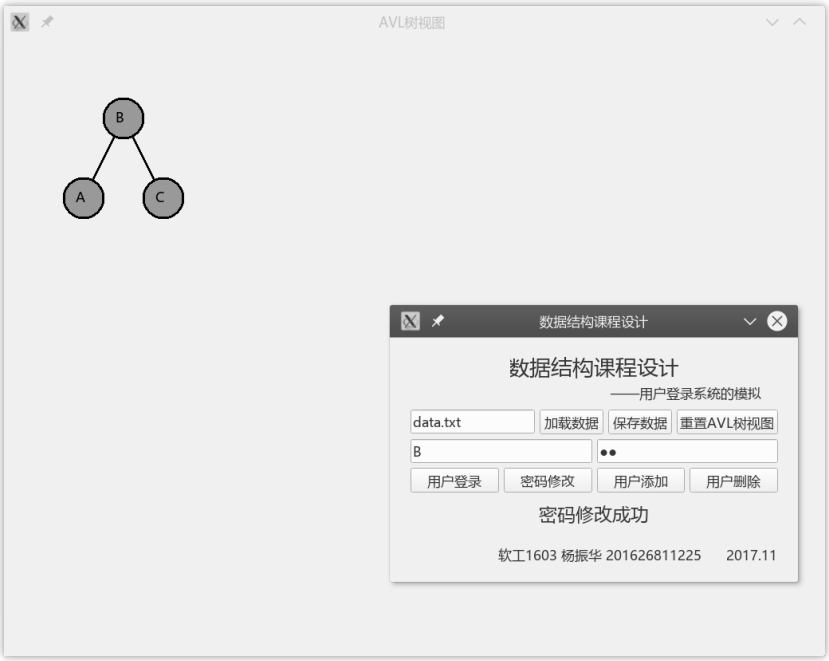

图 37 修改密码成功

否则，修改失败，提示错误信息（图 38）。

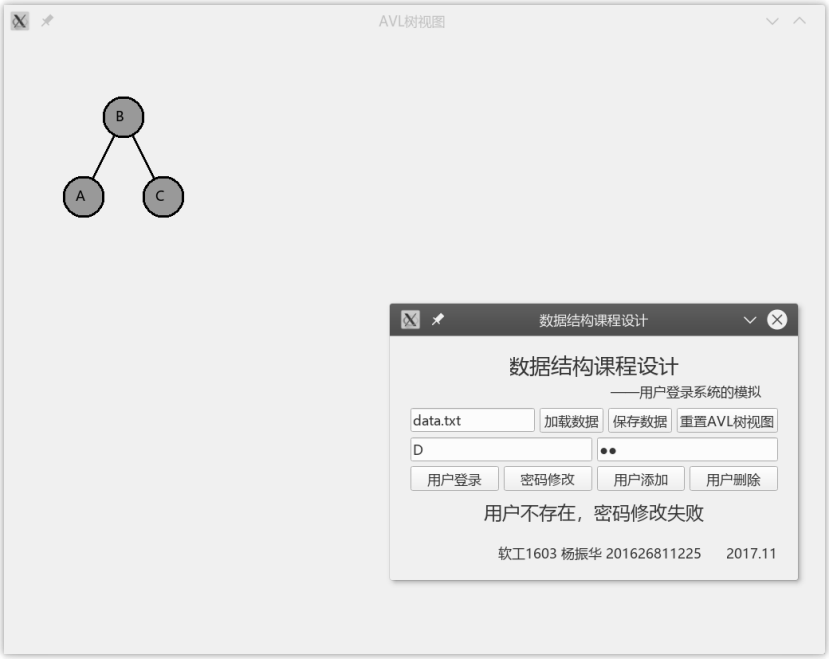

图 38 修改密码失败

7. 用户登录测试

用户存在于 AVL 树中且密码正确，则登录成功（图 39）。

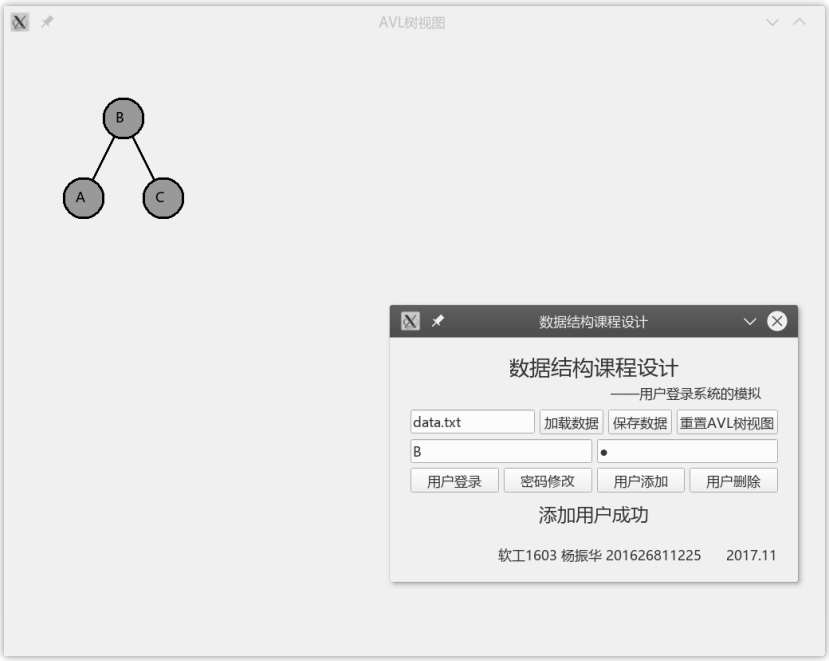

图 39 登录成功

若密码错误，则登录失败，并提示相关错误信息（图 40）。

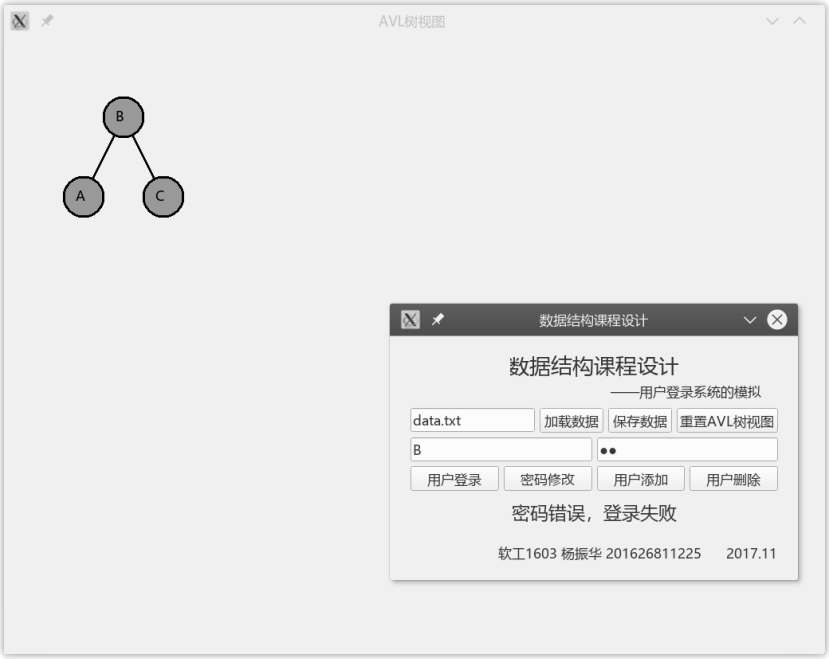

图 40 密码错误，登录失败

若用户不存在，则登录失败，并提示相关错误信息（图 41）。

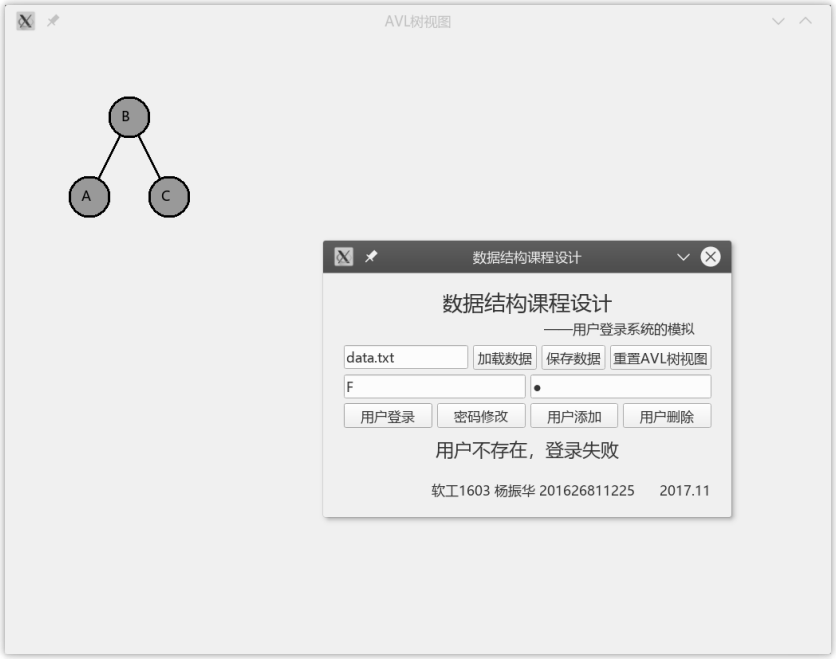

图 41 用户不存在，登录失败

五、 附录

详细的源代码见附件。