

目录

1 实验题目和要求	2
2 设计思路	3
2.1 系统总体设计	3
2.2 系统功能设计	3
2.3 类的设计	5
2.4 主程序的设计	24
3 调试分析	25
3.1 技术难点分析	25
3.2 调试错误分析	25
4 测试结果分析	30
4.1 一般数据测试	30
4.2 边界数据测试	35
4.3 错误数据测试	44
5 附录	48

1 实验题目和要求

实验题目：用户登陆系统的模拟

【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；

2 设计思路

2.1 系统总体设计

2.1.1 系统的技术思路

系统总体设计主要分为三个部分：

(1) 信息节点：用 `UserInfo` 类保存每一个用户的信息，`UserInfo` 类中还包
括上下节点的指针，方便后续处理；

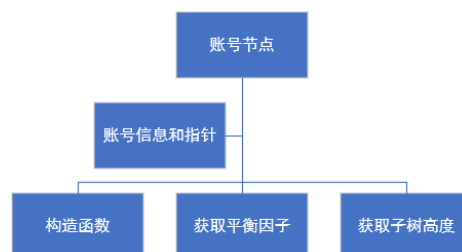


图 1 信息节点示意图

(2) 信息维护：用 AVL 树的数据结构对用户信息库进行存储和维护，用户
信息库保存在 `UserTree` 中。

(3) 主程序：用于和用户的交互，有多种功能。

2.1.2 系统的数据结构

系统采用 AVL 树实现。

AVL 树作为一种具有高度平衡特性的 BST，在查找的效率上有着独特的优越性和稳定性。

2.2 系统功能设计

利用 AVL 树 `UserTree` 实现各种信息的维护功能。程序头文件为 `UserTree.h`，其需要完成的任务可以分成两大类：面向用户的功能和面向组件的功能。面向用户的功能包括登录、注销、修改密码、注册、图形输出。面向组件的功能包括插入、删除、搜索、读取、判空、输出和平衡树专属的平衡操作等。其中平衡树专

属的平衡操作又包括左旋、右旋、左-右旋、右-左旋等 4 个函数。

此外，还有一些辅助函数用来帮助功能的实现。

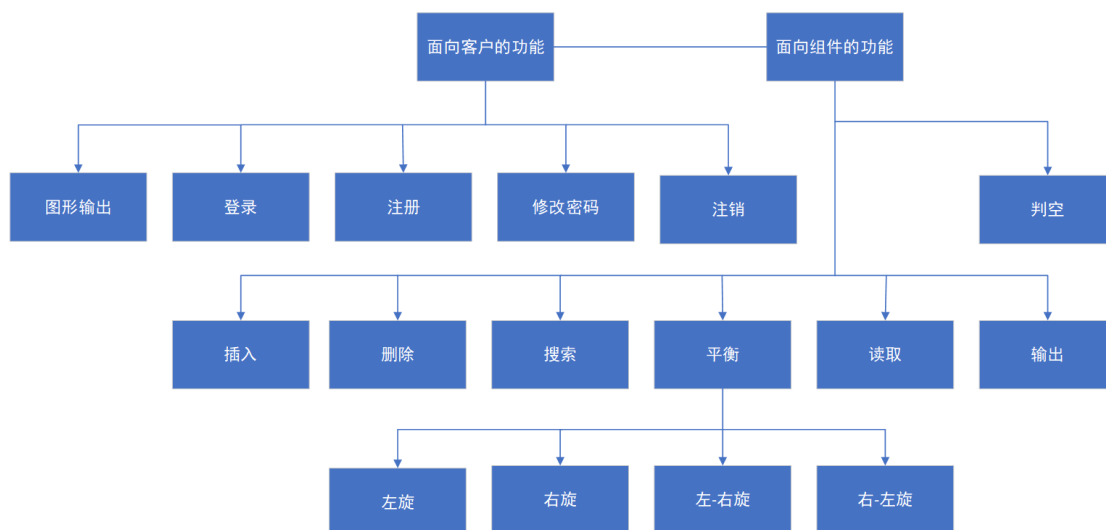


图 2 信息维护示意图

至于用户的交互方面，在主界面，用户可以登录、注册，以及查看当前用户信息库的拓扑结构；而在用户界面，其可以进行账号的管理，操作包括修改密码和删除账户（假设所有用户都具有管理员权限，即都可以对整个系统的所有账户进行管理）。

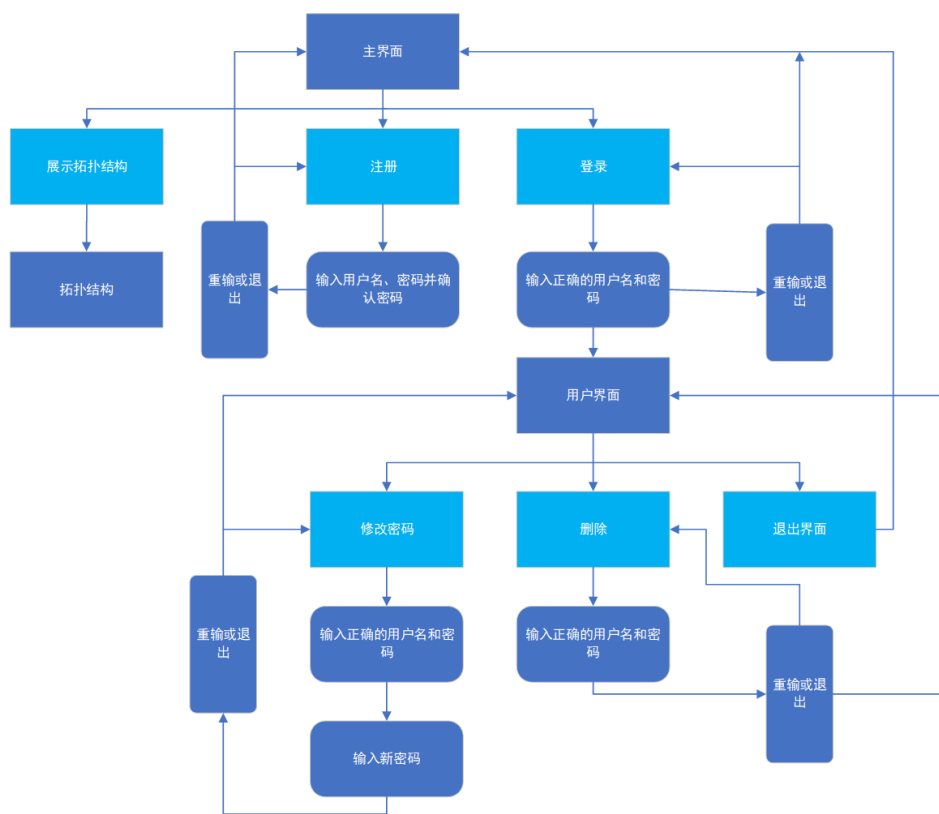


图 3 主程序示意图

2.3 类的设计

2.3.1 UserInfo 类

该类的成员及说明如下：

表 1 UserInfo 类

UserInfo 类		
public 成员		
变量名	变量类型	功能描述
userId	string	用户 id
userPw	string	用户密码
balanceFactor	int	平衡因子
height	int	高度
left	UserInfo*	指向左子女的指针
right	UserInfo*	指向右子女的指针
parent	UserInfo*	指向双亲节点的指针
函数名	函数类型	功能描述
UserInfo()	默认构造函数	默认构造函数
UserInfo(const string& id, const string& password)	重载构造函数	重载构造函数
getBalanceFactor()	int	获取节点的平衡因子
getHeight()	int	获取子树的高度
private 成员		
函数名	函数类型	功能描述
getHeightAux(UserInfo* root)	int	getHeight 的辅助函数

2.3.1.1 获取节点的平衡因子函数 getBalanceFactor

返回当前节点的平衡因子。平衡因子即该节点左子树高度减去右子树高度，其计算左右子树高度的过程需要调用 getHeight()。

该函数流程图如下。

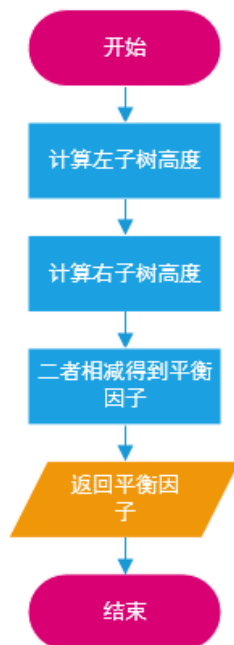


图 4 获取节点的平衡因子函数 getBalanceFactor

2.3.1.2 获取子树的高度函数 getHeight

返回以当前节点为根节点的子树的高度。该函数通过调用辅助函数 getHeightAux(UserInfo* root)实现递归操作，取边界条件当前节点指针为空，此时函数返回值为 0，递归过程中左右子树高度逐次加 1，每一轮递归在两个高度值中取较大的返回，作为以当前节点为根节点的子树的高度。

该函数流程图如下。

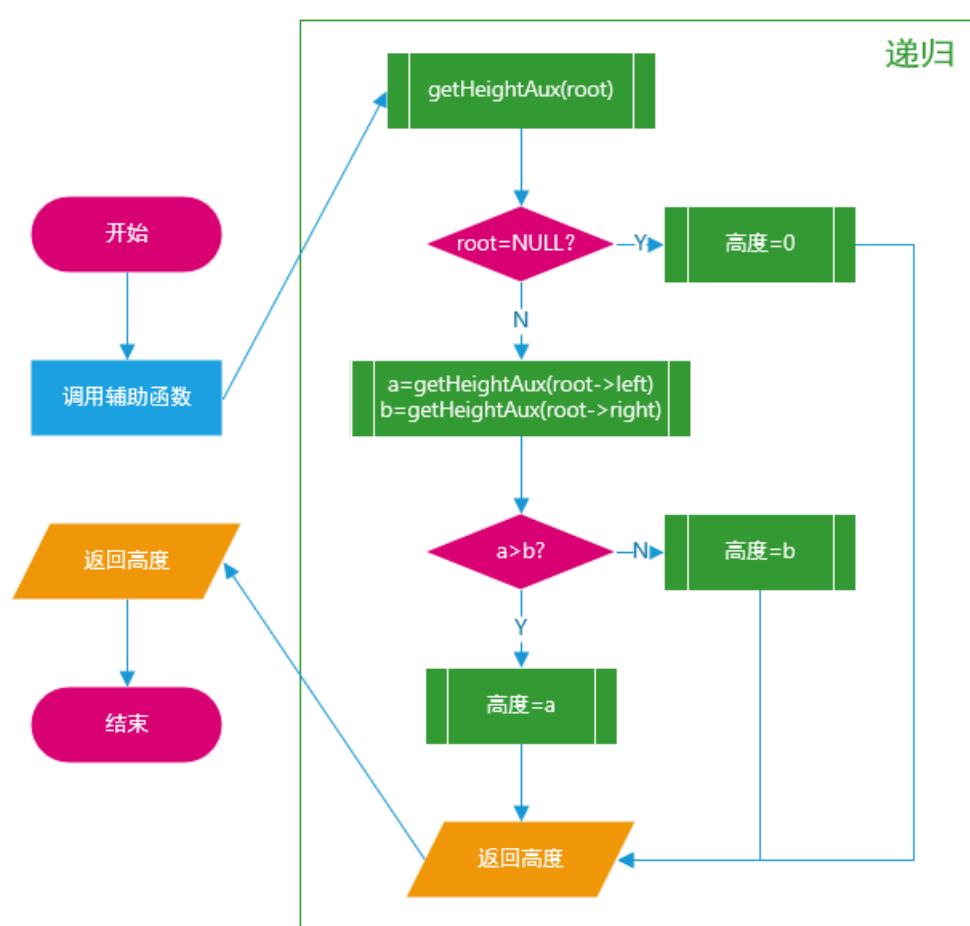


图 5 获取子树的高度函数 getHeight

2.3.2 UserTree 类

该类的成员及说明如下：

表 2 UserTree 类

UserTree 类		
public 成员		
函数名	函数类型	功能描述
UserTree ()	默认构造函数	默认构造函数
UserTree(const string& f)	重载构造函数	重载构造函数
empty ()	bool	判空函数
insert ()	void	插入函数

read()	void	输入函数
write(ostream& out)	void	输出函数
inorder(ostream& out) const	void	中序遍历函数
login()	bool	登录函数
remove()	void	删除函数
search(const string& id, bool& found, UserInfo*& locptr) const	void	查找函数
update()	void	修改密码函数
graph() const	void	图形输出函数
getBF()	void	获取各节点的平衡因子
~UserTree()	析构函数	析构函数
private 成员		
函数名	函数类型	功能描述
inorderAux(ostream& out, UserInfo* subroot) const	void	inorder 的辅助函数
insertAux(const bool& print, const string& id, const string& password)	void	insert 的辅助函数
getBFAux(UserInfo* subroot)	void	getBF 的辅助函数
LL(UserInfo* userB)	void	左旋操作
RR(UserInfo* userB)	void	右旋操作
LR(UserInfo* userC)	void	左-右旋操作
RL(UserInfo* userC)	void	右-左旋操作
destAux(UserInfo* subtreeRoot)	void	析构函数的辅助函数
变量名	变量类型	功能描述
myRoot	UserInfo*	指向根节点
filename	string	存储的文件名
n	int	结点数量

2.3.2.1 插入函数 insert 和读取函数 read

插入和读取函数在实现上非常相似，两者区别仅在于，前者由控制台输入，且包含用户交互，而后者则由文件输入，不包含用户交互。这两个函数共同使用了辅助函数 insertAux。更新文件信息时调用了输出函数 write(ostream& out)，其具体说明见“2.3.2.6 输出函数 write 与中序遍历函数 inorder”。

插入函数的流程图如下。

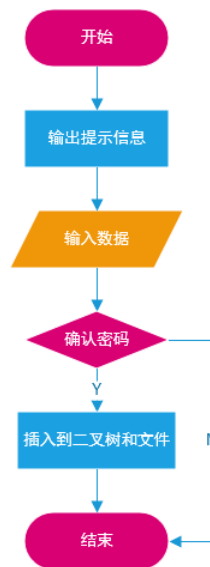


图 6 插入函数 insert

执行插入函数时，先输出与用户交互的提示信息，用户输入数据后，若两次输入的密码一致，则调用 insertAux 将节点插入到二叉树；若不一致，则重新输入或退出。在“插入到二叉树和文件”一步，用户可查看节点拓扑结构，详情请见对辅助函数 insertAux 的说明。

读取函数的流程图如下。

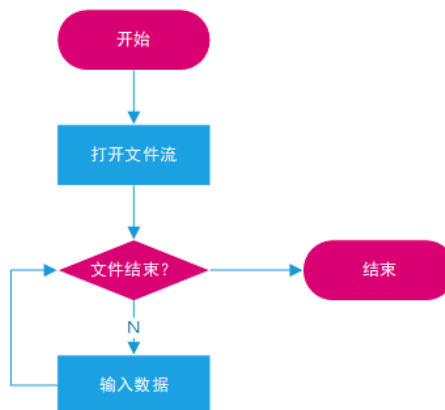


图 7 读取函数 read

执行读取函数时，打开文件流后就循环调用 `insertAux`，将文件数据逐个插入到二叉树。此时默认无提示信息和图像输出。

辅助函数 `insertAux` 的流程图如下。

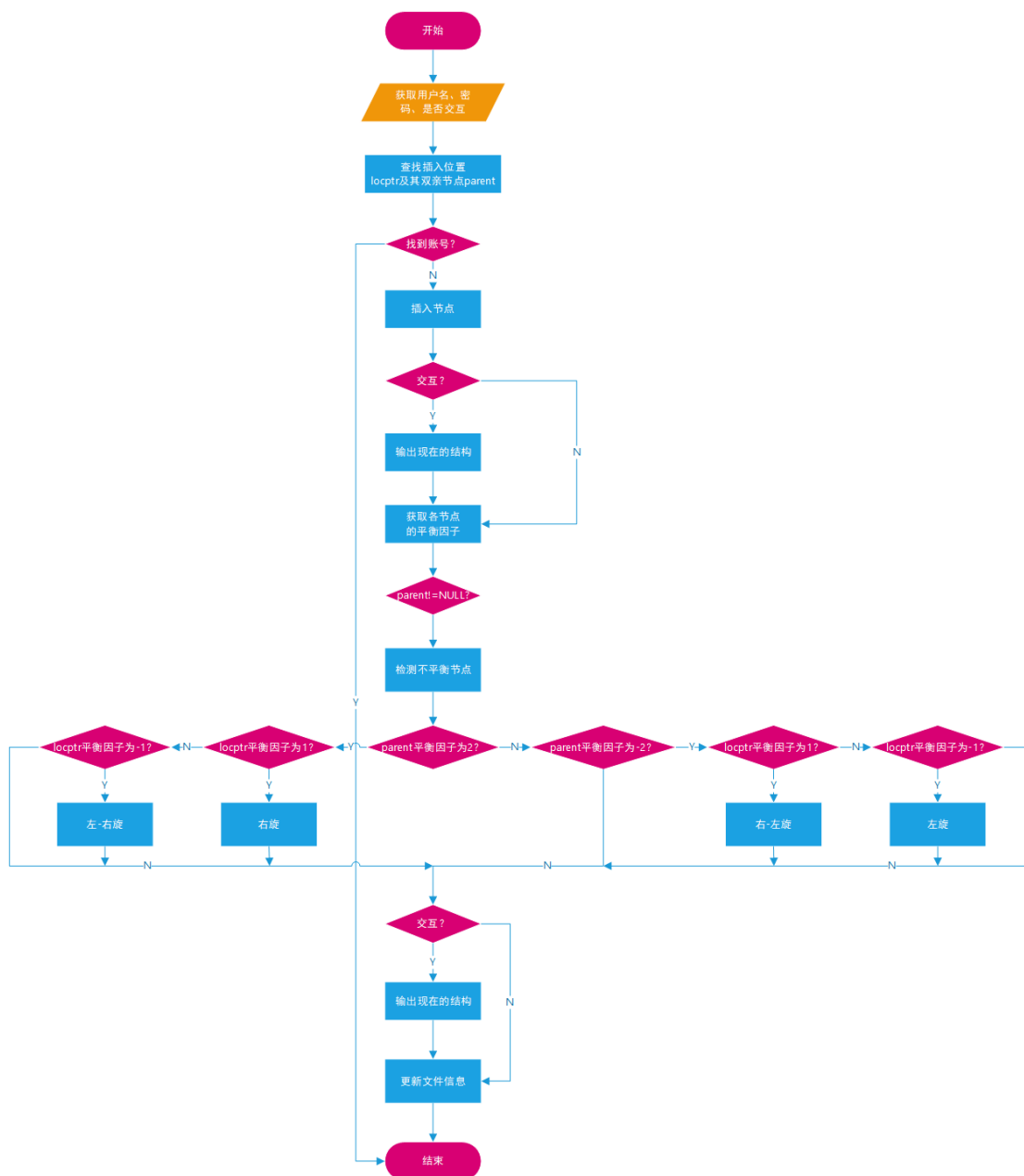


图 8 辅助函数 `insertAux`

执行 `insertAux` 时，会先搜索节点的插入位置，若找到了对应账号，则提示已有账号并结束。若未找到，则插入节点并获取各节点的平衡因子，因为插入位置必为最底层，故可由插入位置向上搜索平衡因子为 ± 2 的不平衡节点，记 `parent` 为首个不平衡节点，`locptr` 为其插入侧的子女节点，则当且仅当 `parent` 和 `locptr` 的平衡因子分别为 ± 2 和 ± 1 时，才进行相应的重平衡操作。在函数执行过程中，

可根据输入的参数确定是否需要与用户交互，方便被以上两函数调用。

对 4 种平衡操作的具体说明见“2.3.2.3 四种平衡操作”。

2.3.2.2 删除函数 `remove`

删除函数 `remove()` 用于删除某个特定账号及其对应的信息节点，并将结果同步到文件。执行删除函数时，会先输出与用户交互的提示信息，用户输入数据后即调用查找函数 `search(const string& id, bool& found, UserInfo*& locptr) const` 查找账号所对应的信息节点，若未找到该账号或账号的用户名密码不匹配，则退出函数；若用户名密码匹配，则开始删除 AVL 树节点和重平衡的操作。最后调用输出函数 `write(ostream& out)` 更新文件信息。在函数执行过程中，用户可选择是否查看节点拓扑结构。

对查找函数的具体说明见“2.3.2.4 查找函数 `search`”。

删除函数的流程图如下。

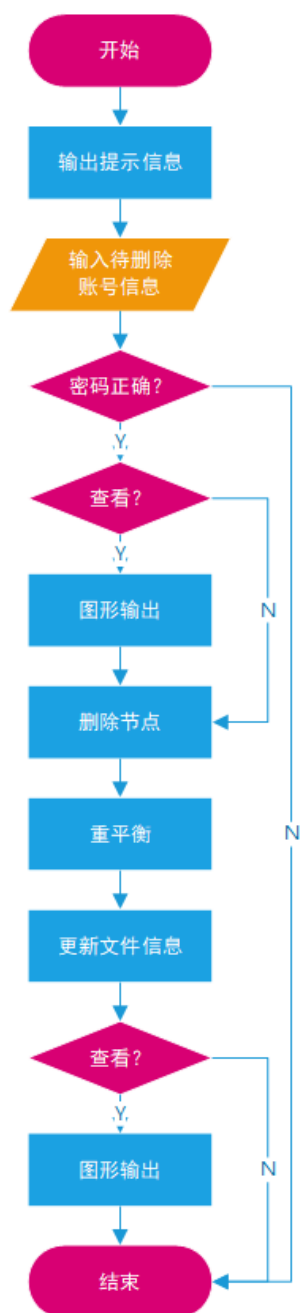


图 9 删除函数 remove

AVL 树的删除操作说明如下：当待删除节点 x 有 2 个子女时，查找 x 的中序后继 $xSucc$ ，再将 $xSucc$ 的内容移至 x ，修改 x 为指向将被删除的后继。由于在 BST 中， $xSucc$ 比 x 左子树任何一个节点都大，比右子树任何一个其他节点（如果存在）都小，故修改并删除 x 后该树仍为 BST，且自此情况将统一于 0 或 1 子女节点的删除。当待删除节点 x 有 1 个子女时，则以 x 的左\右子女（默认左子女，不存在则为右子女）将其替换并删除对应子女； x 为叶节点时，则将 x 直接删除。

删除节点操作的流程图如下。

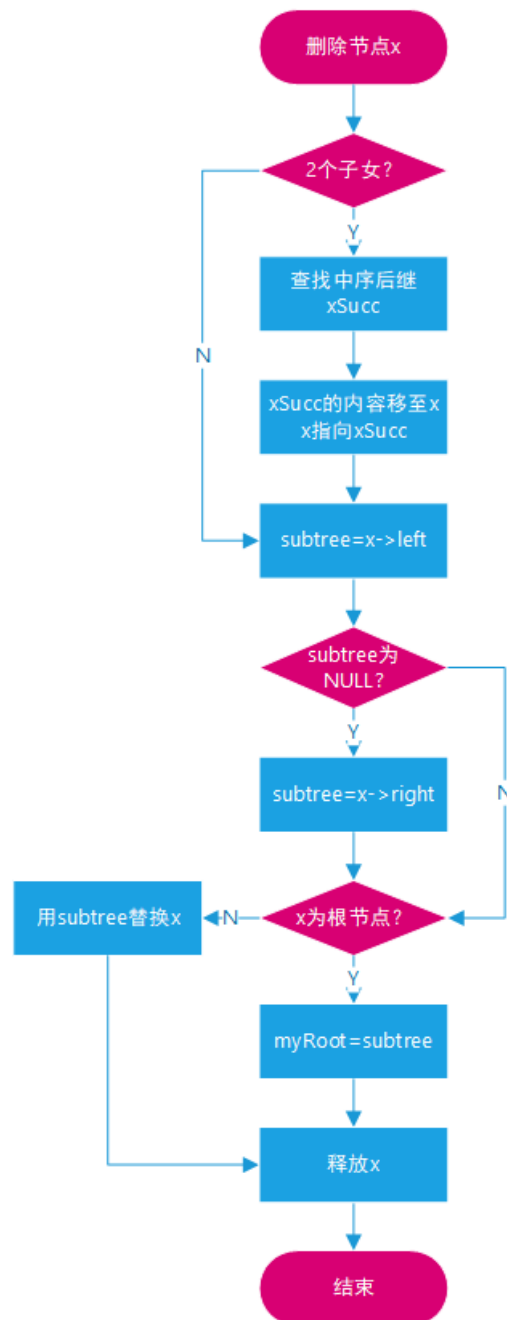


图 10 删除节点操作

删除后的平衡情况较插入更多。不像插入，删除的操作不一定在最底层进行，因此可能会导致不平衡节点的子女节点平衡因子为 0 的情况，此时无论对其进行单旋还是双旋操作都可以使其平衡。

该情况示意图如下。

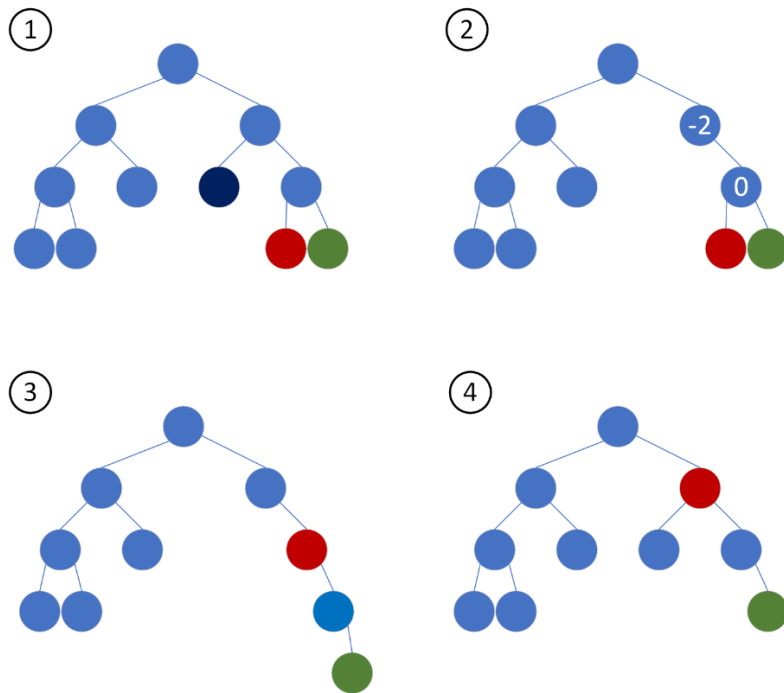


图 11-2—0，采用右-左旋

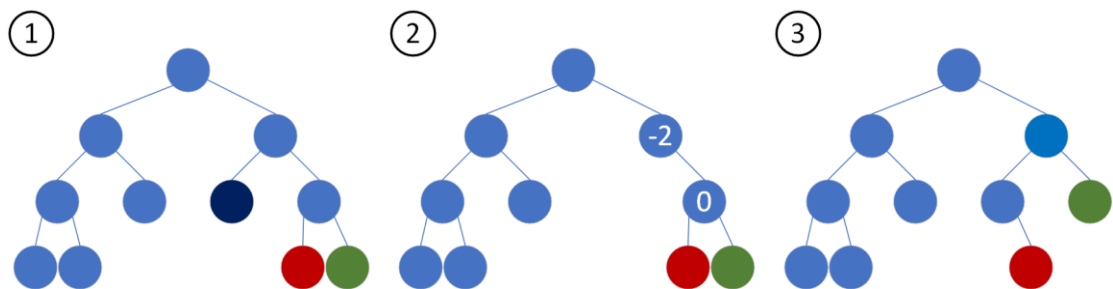


图 12-2—0，采用左旋

2—0 情况类似，故不赘述。为了节省效率，实现中我采用了单旋操作。
重平衡操作的流程图如下。

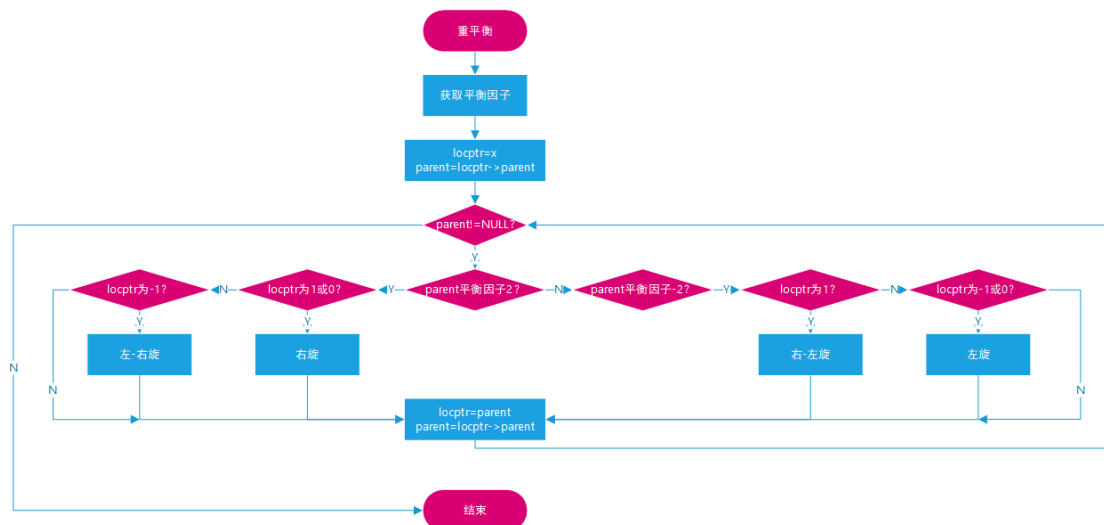


图 13 删除后重平衡

2.3.2.3 四种平衡操作

为了维护 AVL 树，一方面需要检测子树的高度来确定每个节点的平衡因子，另一方面要用到 4 种操作维持平衡状态。

1. 左旋 LL(UserInfo* userB)

A—B 满足 -2—-1 或 -2—0 时使用左旋。

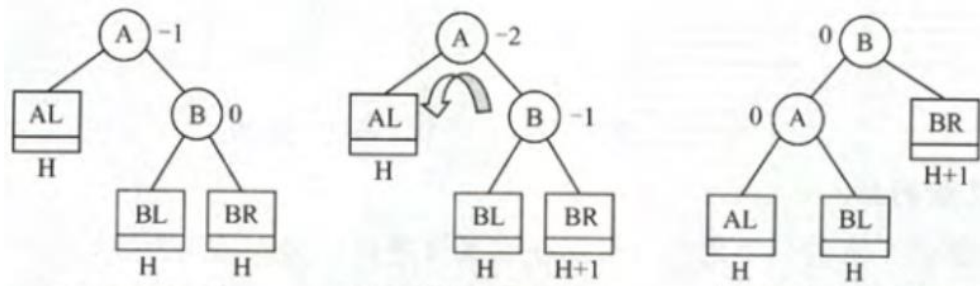


图 14 左旋 LL(UserInfo* userB)

如图，A 处的平衡因子为 -2，为了使 A 处的平衡因子绝对值减小，需要增加该节点处左侧的树，因此将 A 向左移，从而带动 B 移动使得 B 成为了子树的根节点。由于 BST 的大小关系以及出度不能大于 2 性质，B 的左子树成为了 A 的右子树。

代码实现如下。

```
//--- 左旋操作
void UserTree::LL(UserInfo* userB)
{
    UserInfo* userA = userB->parent;
    if (!(userA->right == userB && userB->right != 0)) {
        cerr << "不符合左旋条件\n";
        return;
    }
    bool flag = false;
    if (userA == myRoot)
        flag = true;
    // A的双亲节点对应指针指向B
    if (userA->parent != 0) {
        if (userA->parent->left == userA)
            userA->parent->left = userB;
        else
            userA->parent->right = userB;
    }
    // 修改AB之间的指针
```

```

userB->parent = userA->parent;
userA->parent = userB;
userA->right = userB->left;
if (userB->left)
    userB->left->parent = userA;
userB->left = userA;
// A是根节点，则旋转后根节点指针指向B
if (flag)
    myRoot = userB;
}
    
```

2. 右旋 RR(UserInfo* userB)

A—B 满足 2—1 或 2—0 时使用右旋，原理与左旋一致，故不赘述。

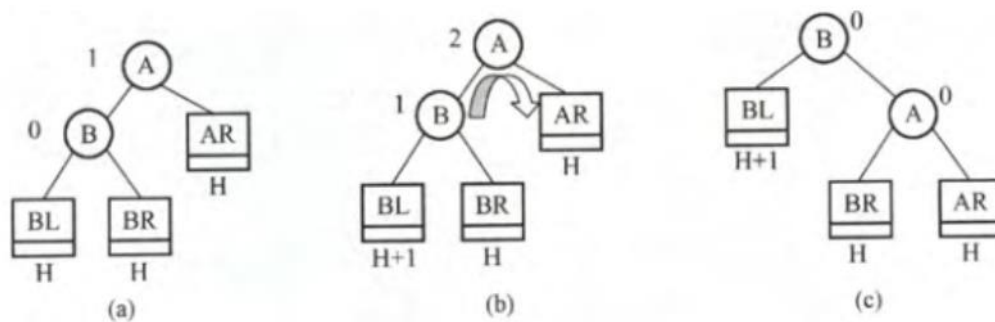


图 15 右旋 RR(UserInfo* userB)

3. 左-右旋 LR(UserInfo* userC)

A—B 满足 2—-1 时使用左-右旋。

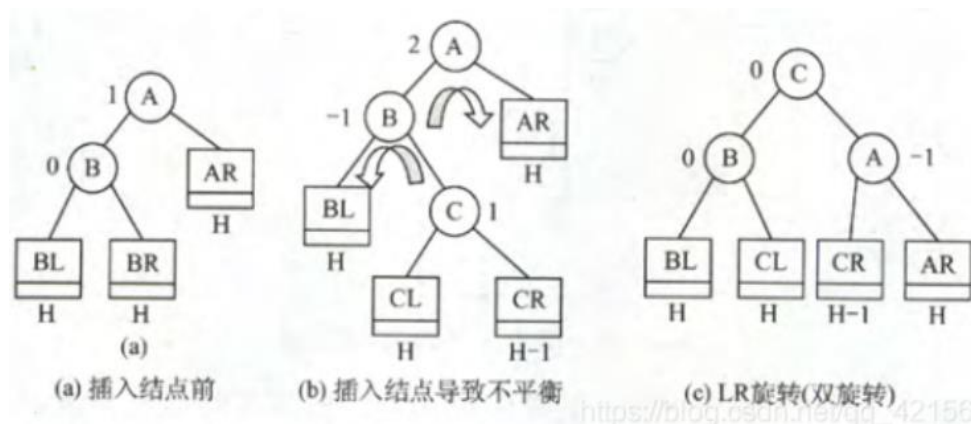


图 16 左-右旋 LR(UserInfo* userC)

进行左-右旋操作时，先将以 B 为根节点的子树左旋，使得 C 成为根节点，

C 的左子树成为 B 的右子树；再将 A 为根节点子树右旋，使得 C 成为根节点，C 的右子树成为 A 的左子树。

代码实现如下。

```
//--- 左-右旋操作
void UserTree::LR(UserInfo* userB)
{
    // 先左旋
    UserInfo* userC = userB->right;
    if (userC == 0) {
        cerr << "不符合左-右旋条件\n";
        return;
    }
    UserInfo* userA = userB->parent;
    if (userA == 0 || userA->left != userB) {
        cerr << "不符合左-右旋条件\n";
        return;
    }
    userC->parent = userA;
    userB->parent = userC;
    userB->right = userC->left;
    if (userC->left)
        userC->left->parent = userB;
    userC->left = userB;
    userA->left = userC;
    // 再右旋
    RR(userC);
}
```

4. 右-左旋 RL(UserInfo* userC)

A—B 满足-2—1 时使用右-左旋，原理与左-右旋一致，故不赘述。

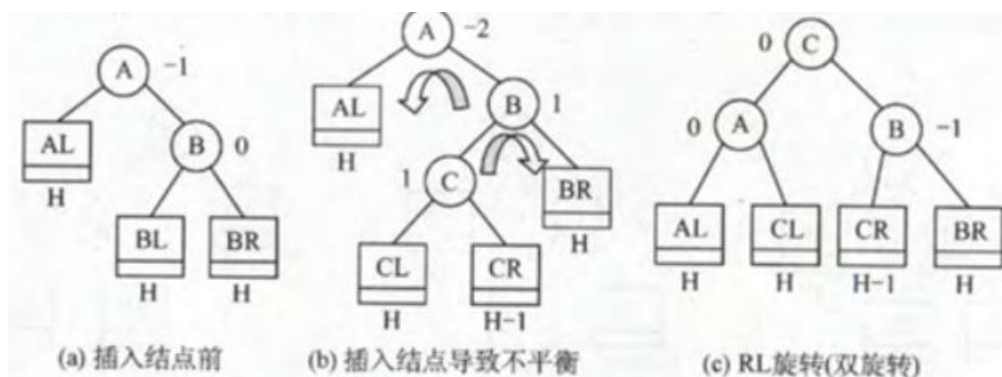


图 17 右-左旋 RL(UserInfo* userC)

2.3.2.4 查找函数 search

查找函数 `search(const string& id, bool& found, UserInfo*& locptr) const` 用于帮助服务于用户的功能函数查找对应的节点。其流程图如下。

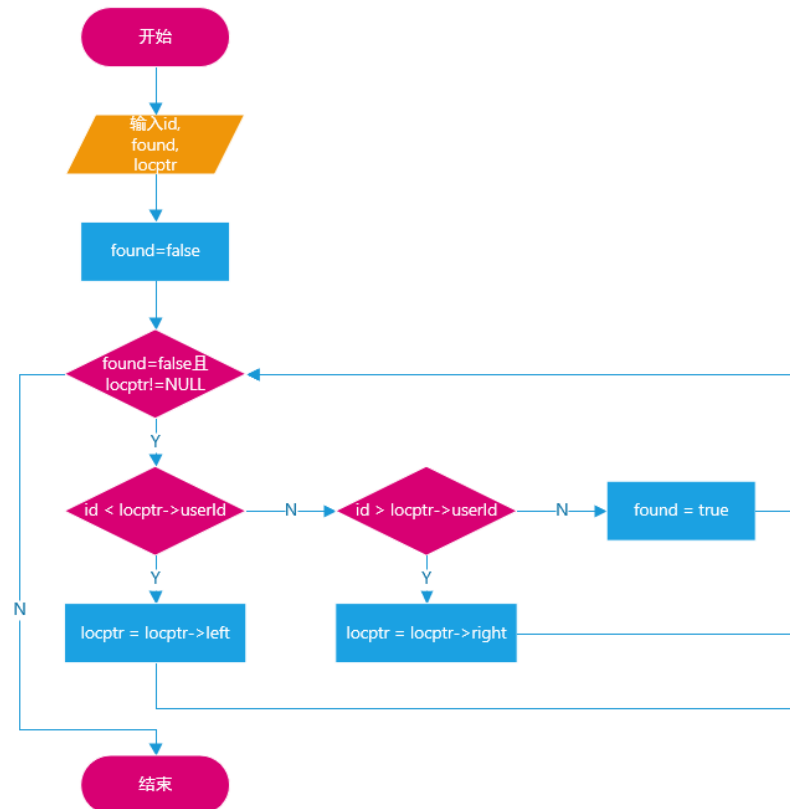


图 18 查找函数 search

2.3.2.5 修改密码函数 update

修改密码函数 `updatee()` 用于修改某个特定账号的密码，并将结果同步到文件。执行修改密码函数时，会先输出与用户交互的提示信息，用户输入数据后即调用查找函数 `search(const string& id, bool& found, UserInfo*& locptr) const` 查找账号所对应的信息节点，若未找到该账号或账号的用户名密码不匹配，则退出函数；若用户名密码匹配，则按用户输入的数据修改密码。最后调用输出函数 `write(ostream& out)` 更新文件信息。

修改密码函数的流程图如下。

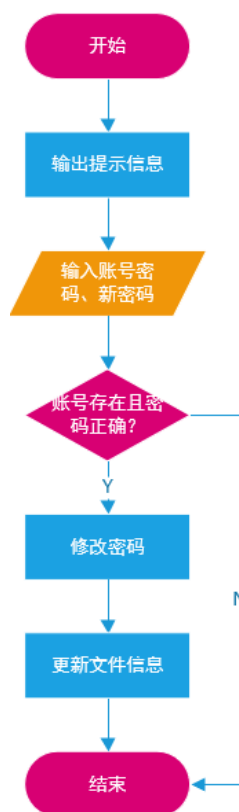


图 19 修改密码函数 update

2.3.2.6 输出函数 write 与中序遍历函数 inorder

输出函数 write 用于中序输出各信息节点的信息。为保证可扩展性（如以不同顺序输出，既能控制台输出又能文件输出等），该函数通过直接调用中序遍历函数实现，故下面直接说明中序遍历函数 inorder。

中序遍历函数通过调用辅助函数 inorderAux(ostream& out, UserInfo* subroot) 实现递归操作，取边界条件当前节点指针为空。该函数在考察到一个节点后，将其暂存，遍历完左子树后，再输出该节点的值，然后遍历右子树。

中序遍历函数的流程图如下。

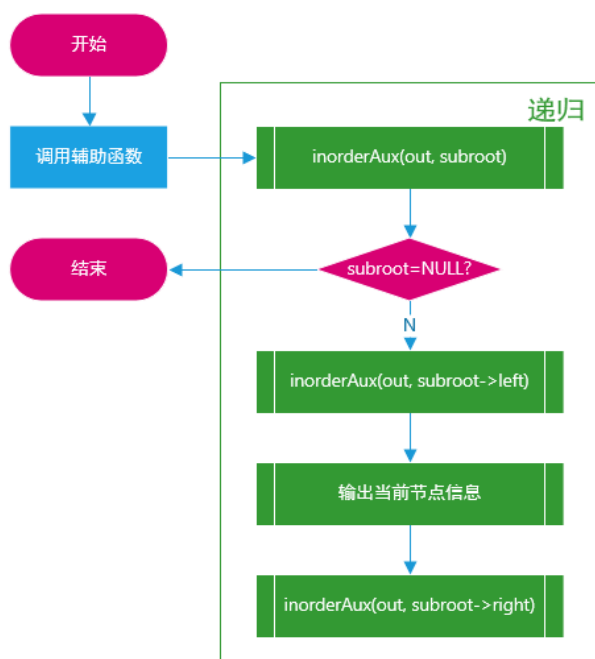


图 20 中序遍历函数 inorder

2.3.2.7 图形输出函数 graph

图形输出函数 **graph** 用于显示信息节点的拓扑结构。为实现层序输出，该函数的实现采用广度优先搜索的思路，将每个节点按拓扑顺序存入队列，再将队列输出，同一层每两个节点间的空格数逐渐减小，达到输出二叉树图像的目的。

实现代码如下。

```

//--- 图形输出函数
void UserTree::graph() const
{
    if (myRoot != 0) {
        queue<UserInfo*> q;
        int h = myRoot->getHeight();           // h >= 1
        int n = 0, depth = 0;
        UserInfo* none = new UserInfo(" ", " "); // 节点值为空格，用于占位

        for (q.push(myRoot); h > 0 && !q.empty(); q.pop()) {
            UserInfo* locptr = q.front();
            int t = 2 * (int(pow(2, h)) - 1);
            int u = t + 4;                       // 包括了每个输出的id所占长度
        }
    }
}

```

```

// 输出节点
cout << setfill(' ') << setw(u) << locptr->userId;
cout << setfill(' ') << setw(t) << ' ';

if (locptr->left)
    q.push(locptr->left);
else
    q.push(none); // 发现无左\右子女，插入空格节点占位
if (locptr->right)
    q.push(locptr->right);
else
    q.push(none); // 发现无左\右子女，插入空格节点占位

// 位于该层最后的节点时
if (++n == int(pow(2, depth))) {
    n = 0;
    ++depth;
    --h;
    cout << endl;
}
}
}
else {
    cout << "没有数据! \n";
}
}
}

```

图形输出函数的流程图如下。

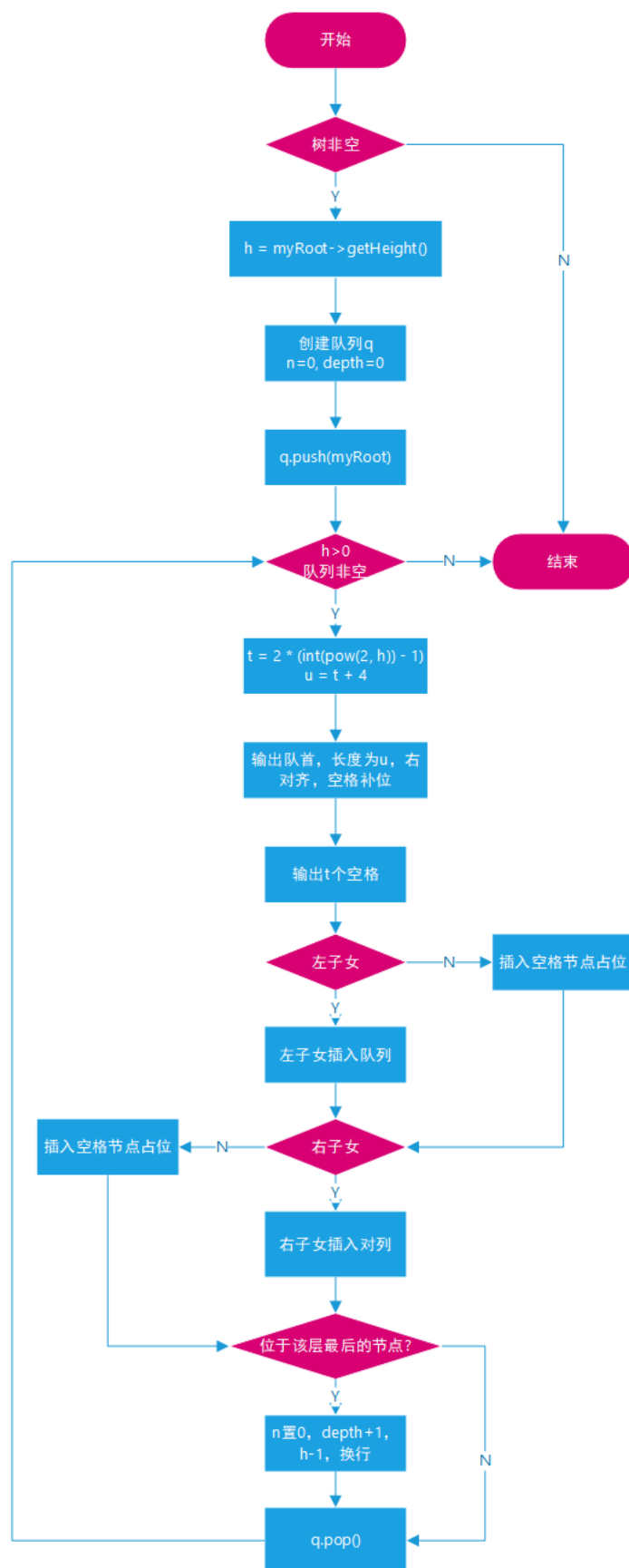


图 21 图形输出函数 graph

2.3.2.8 登录函数 login

登录函数用于从登录界面跳转到用户界面。执行登录函数时，会先输出与用户交互的提示信息，用户输入数据后即调用查找函数 `search(const string& id, bool& found, UserInfo*& locptr) const` 查找账号所对应的信息节点，若未找到该账号，则提示是否注册新用户，进而调用插入函数 `insert` 或退出；若账号的用户名密码不匹配，则提示重输退出；若用户名密码匹配，则进入用户界面。

登录函数的流程图如下。

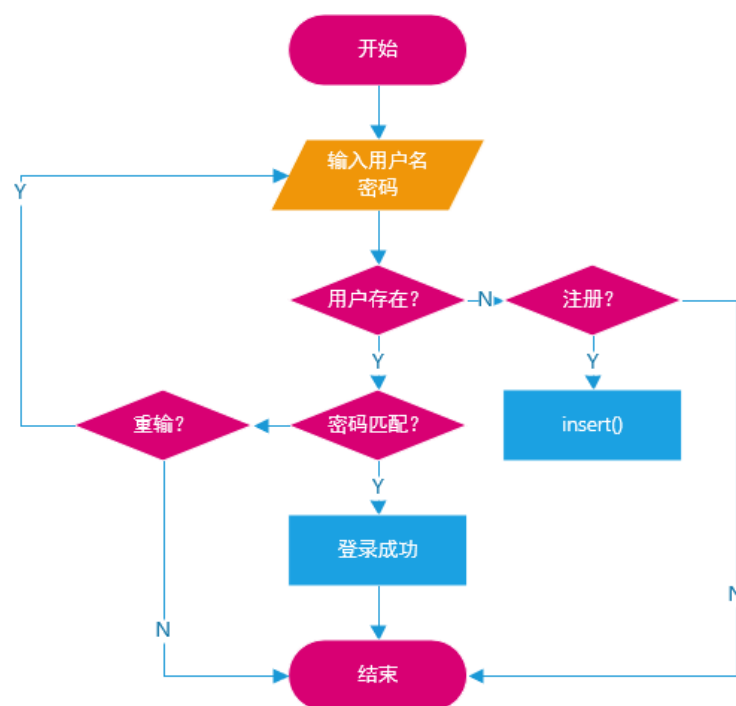


图 22 登录函数 login

2.3.3 类之间的关系

`UserInfo` 类为实现 `UserTree` 类所需的节点类，用来存储 `UserTree` 所建立的 AVL 树各节点的信息，并实现如计算子树高度和节点平衡因子等与节点相关的函数。`UserTree` 类则负责建立起 AVL 树，并进行与树相关的各种操作。

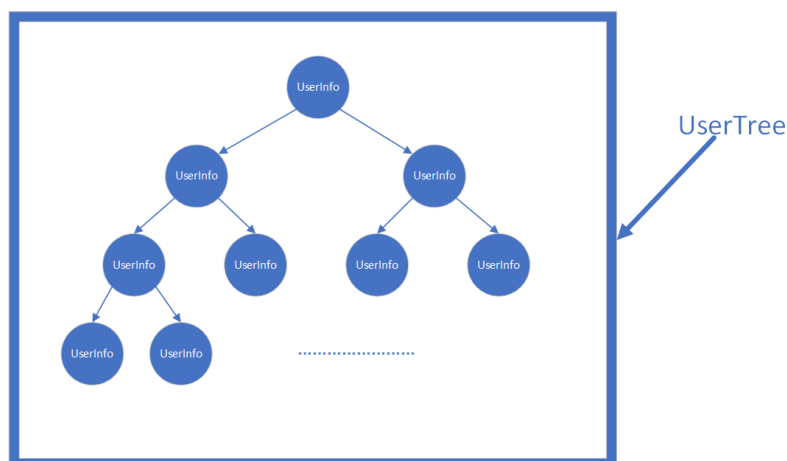


图 23 类关系

2.4 主程序的设计

主程序分为登录界面和用户界面两块，每个界面都有不同的功能。

在登录界面，用户可以输入 1、2、3，对应的功能分别为登录、注册、展示拓扑结构，调用的函数分别为 `login`、`insert`、`graph`。

`login` 用于连接登录界面和用户界面。登录成功后，由登录界面进入用户界面，这时用户可以输入 1、2、3，对应的功能分别为修改密码、删除用户、退出界面，前两个调用的函数分别为 `update` 和 `remove`，若选 3 则回到登录界面。

主程序的示意图如下。

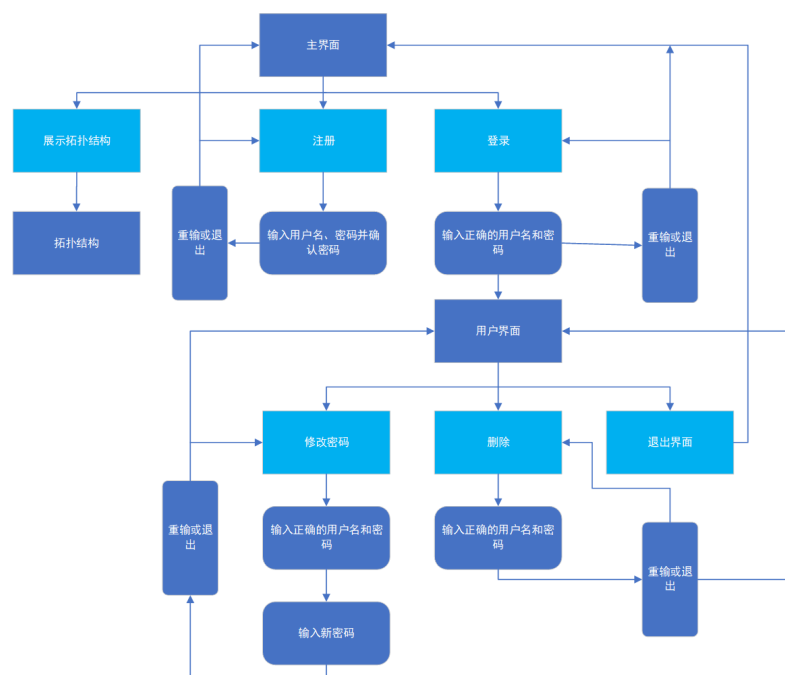


图 24 主程序的示意图

3 调试分析

3.1 技术难点分析

1. 删除节点

删除节点的情况远比插入复杂：不仅需要考虑删除的位置，如叶节点、只有左子女或右子女的节点、有两个子女的节点，还要考虑删除后出现的更多的不平衡情况。这不仅需要我们跳出插入操作的简单的思维方式，想到更多的情况，更需要我们思考如何找到不同情况之间的共性，达到简化程序的目的。

在解决多种删除情况的问题时，可以将待删除节点的中序后继内容移动至待删除节点，使得程序能够在不改变 AVL 树平衡特性和节点大小关系的基础上将情况统一于 0 或 1 子女节点的删除。

在解决多种不平衡情况的问题时，由于单旋和双旋的解决方案对新增的不平衡情况都可适用，于是可以将其与更简单的单旋情况视作一致，节省开支。

2. 图形输出

简单的前序、中序、后序输出都无法实现层序输出的目的，必须借助队列和广度优先搜索的思想才能将同一层的节点数据显示在同一行。

在实现过程中，还将面对缺少节点，输出无法填满一行的情况，解决这个问题可以将队列元素的类型设为 `UserInfo`，在插入队列的过程中每当当前节点没有左或右子女时，就在队列中插入一个数据全为空格，指针全为 `NULL` 的节点，达到占位的目的。

对二叉树的排版，还需要每一层数据间的空格个数都不一样，这个问题可以通过设置随节点数和层数而变化的参数来解决。

3.2 调试错误分析

1. 读入数据失败

发现无法读入数据。程序读完 `txt` 文件后，进行登录操作，发现用户数据读入失败。

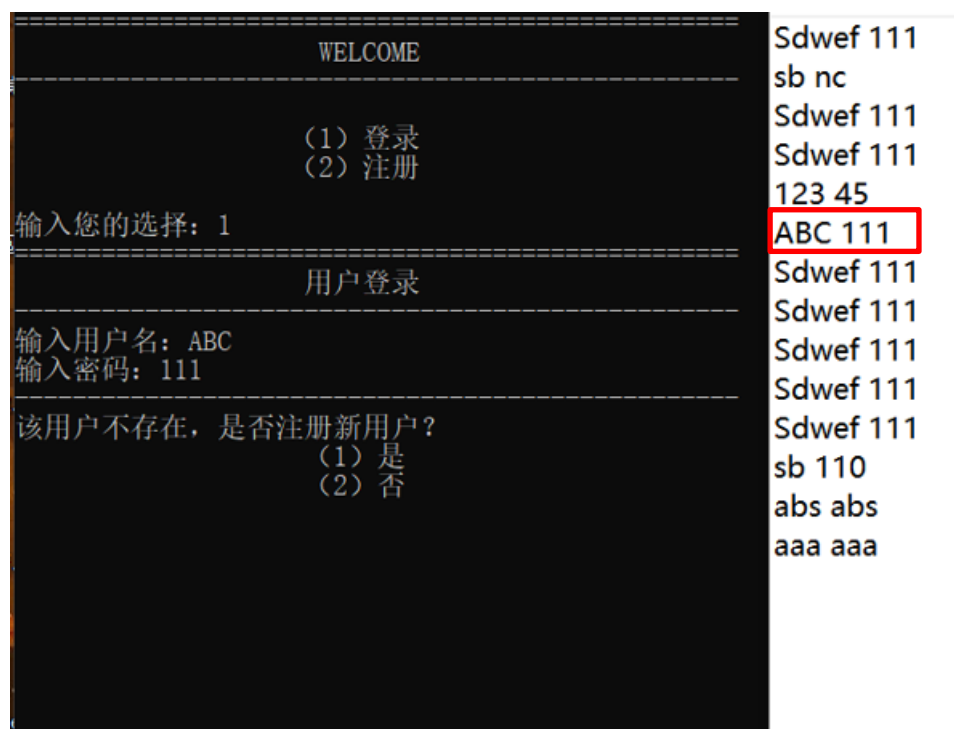


图 25 用户不存在（请忽略我那被恶搞过的数据，我也是最后整理错误时才发现）
如图所示，程序显示用户 ABC 不存在，然而实际上在导入的文件中有。

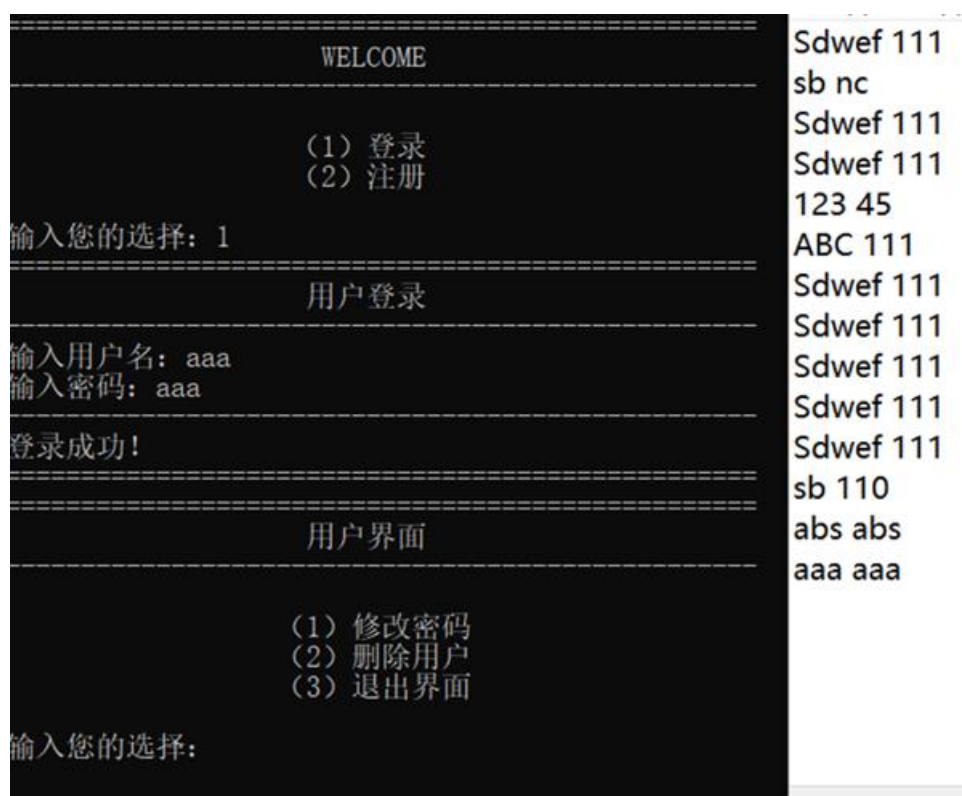


图 26 仅最后的用户有效（请忽略我那被恶搞过的数据，我也是最后整理错误时才发现）
如图所示，研究发现程序似乎只能记录最后一组读入的数据。

调试后发现，问题原因出自 read 函数，其没有在读入时循环插入节点。

=====	Sdwef 111
WELCOME	sb nc
=====	Sdwef 111
(1) 登录	Sdwef 111
(2) 注册	123 45
=====	ABC 111
输入您的选择: 1	Sdwef 111
=====	Sdwef 111
用户登录	Sdwef 111
=====	Sdwef 111
输入用户名: ABC	Sdwef 111
输入密码: 111	Sdwef 111
=====	sb 110
登录成功!	abs abs
=====	aaa aaa
=====	
用户界面	
=====	
(1) 修改密码	
(2) 删除用户	
(3) 退出界面	
=====	
输入您的选择:	

图 27 修改后（请忽略我那被恶搞过的数据，我也是最后整理错误时才发现）

修改 read 函数后，程序读入数据正常。

2. 平衡操作错误

发现平衡操作后输出数据形态发生根本性错误。

```

E:\VS2019-workspace\LoginSystem\Debug>LoginSystem.exe
=====
WELCOME
=====
(1) 登录
(2) 注册
(3) 展示拓扑结构
=====
输入您的选择: 3
=====
1
2
10 3
4
5
6
7
8
9
=====
请按任意键继续. . .
    
```

图 28 平衡操作错误

如图，数据间的组织形式完全不像二叉树。

检查代码后，发现是我为节点定义了 `parent` 指针，而在平衡过程中却没有给 `parent` 指针重新赋值，导致发生严重错误。

```

E:\VS2019-workspace\LoginSystem\Debug>LoginSystem.exe
=====
WELCOME
=====
(1) 登录
(2) 注册
(3) 展示拓扑结构
输入您的选择: 3
=====
4
2 6
1 3 5 8
10 7 9
=====
请按任意键继续. . .
    
```

图 29 修改后每层的节点排列正常

如图，在平衡函数中加上了给 `parent` 指针赋值的代码后，程序输出正常（此时尚未开始进行二叉树的排版）。

3. 二叉树排版错误

发现二叉树输出时对底层节点间的空格数没有把握准确。

```

E:\VS2019-workspace\LoginSystem\Debug>LoginSystem.exe
=====
WELCOME
=====
(1) 登录
(2) 注册
(3) 展示拓扑结构
输入您的选择: 3
=====
              7
             / \
            3   9
           / \ / \
        10 3 5 8  B
       / \ / \ / \
      1 2 4 6 A C
=====
请按任意键继续. . .
    
```

图 30 二叉树排版错误

如图所示，4、6 本应为 5 的左右子女，然而不在 5 的下方；A、C 本应为 B 的左右子女，然而也不在 B 的下方。

调试发现错误原因一方面是空格数问题，另一方面也因为没有考虑有节点无左右子女的情况。解决方法见“3.1 技术难点分析”中的“2. 图形输出”。

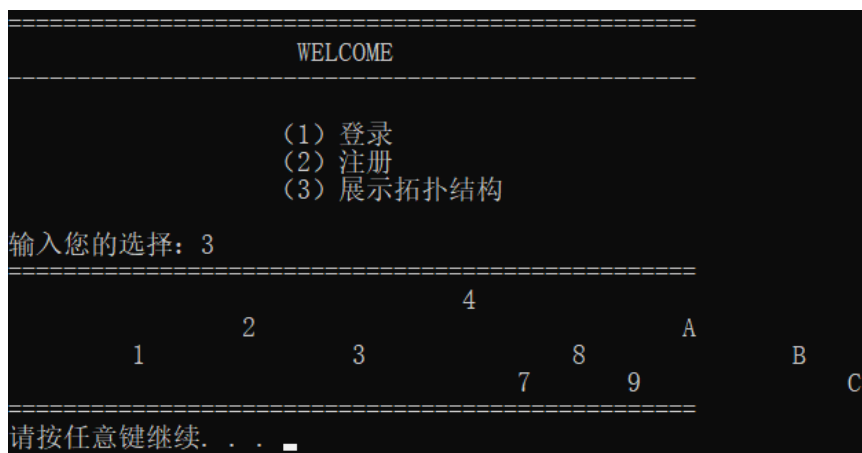


图 31 修改后的二叉树（注：这张是之后补的，数据和比较标准都与之前不同）

如图所示，修改后能顺利应对有空缺的情况。

4. 删除时未考虑新的不平衡情况

发现删除造成的 $\pm 2-0$ 情况无法被平衡。



图 32 未考虑新的不平衡情况

如图所示，删除了节点 8 后，9 的平衡因子为-2，但由于 A1 的平衡因子为 0，

故未被检测出来。



图 33 考虑了新的不平衡情况

如图，在判断条件中加入了 $\pm 2-0$ 的情况后，程序运行正常。

4 测试结果分析

4.1 一般数据测试

1. 进入登录界面



图 34 登录界面

2. 登录



图 35 正常登录

3. 修改密码

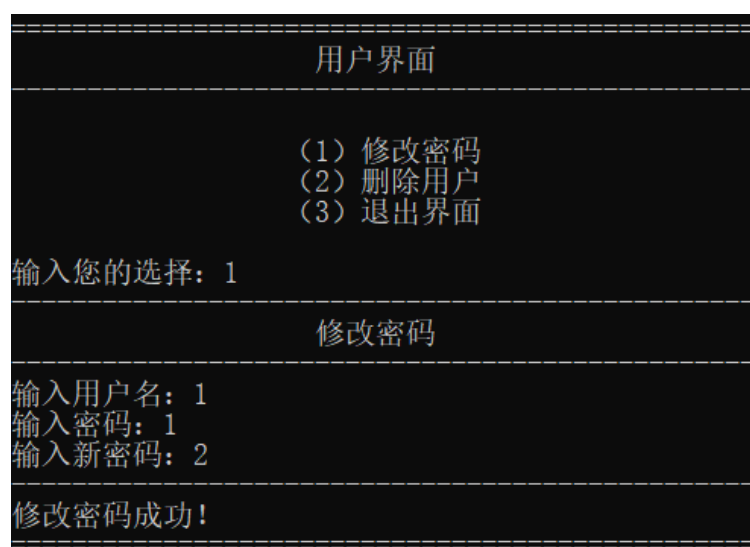


图 36 修改密码成功



图 37 账号 1 登录时密码为 1，现在为 2（第二行）

4. 删除用户

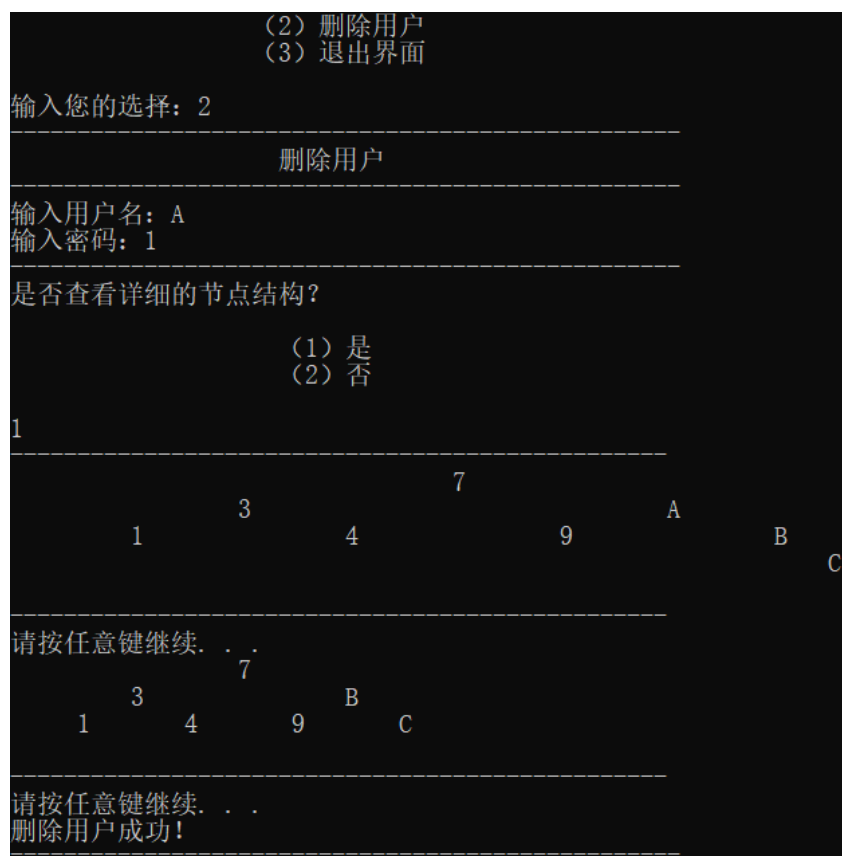



图 38 删除用户，查看结构

 E:\VS2019-workspace\LoginSystem\Debug>LoginSystem.exe

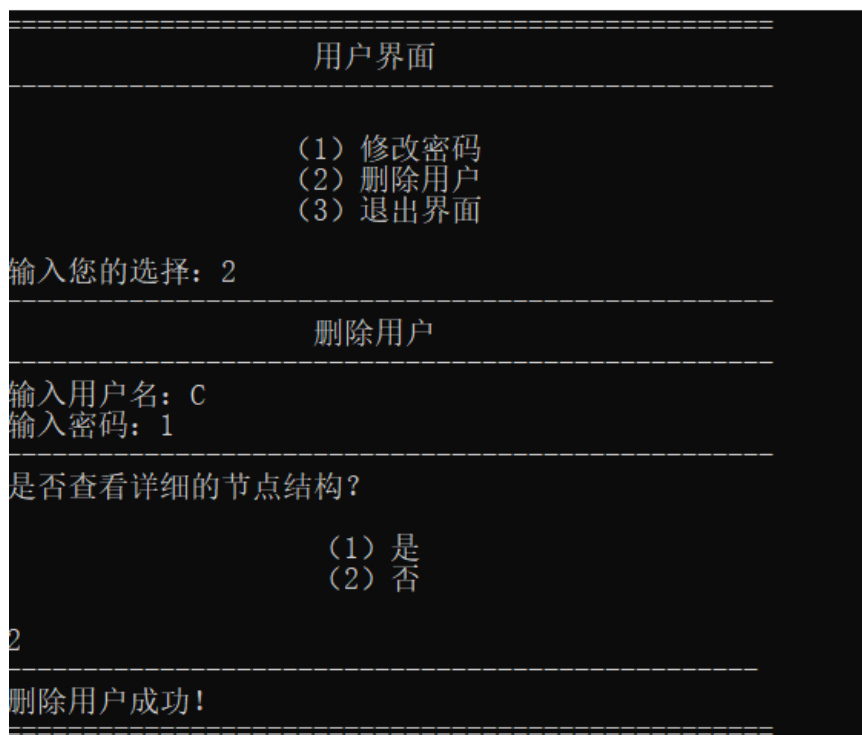


图 39 删除用户，不看结构


```

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
6
1 2
3 1
4 1
7 1
9 1
B 1
    
```

图 40 图 37 中有用户 AC，现在没有

5. 退出界面

```

=====
                        用户界面
=====
                        (1) 修改密码
                        (2) 删除用户
                        (3) 退出界面
输入您的选择: 3
    
```

图 41 选择 3

```

E:\vs2019-workspace\loginsystem\Debug\loginsystem.exe
=====
                        WELCOME
=====
                        (1) 登录
                        (2) 注册
                        (3) 展示拓扑结构
输入您的选择:
    
```

图 42 等待 1000ms 后回到登录界面

6. 注册

```

=====
                        新建用户信息
=====
输入用户名: E
输入密码: 1
确认密码: 1
=====
是否查看详细的节点结构?
                        (1) 是
                        (2) 否
2
=====
新建用户成功，即将返回主界面。
=====
    
```

图 43 注册用户，不看结构

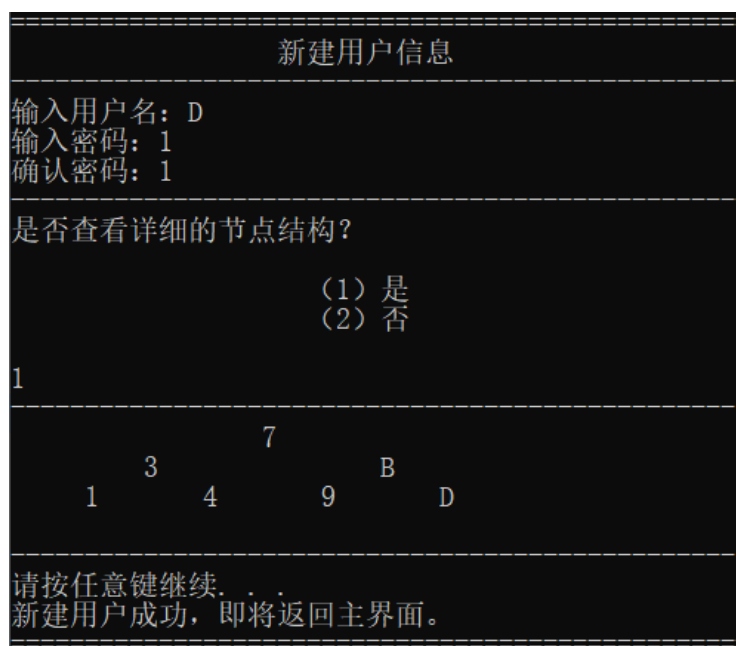


图 44 注册用户，查看结构



图 45 图 40 中无用户 DE，现在有

7. 展示拓扑结构

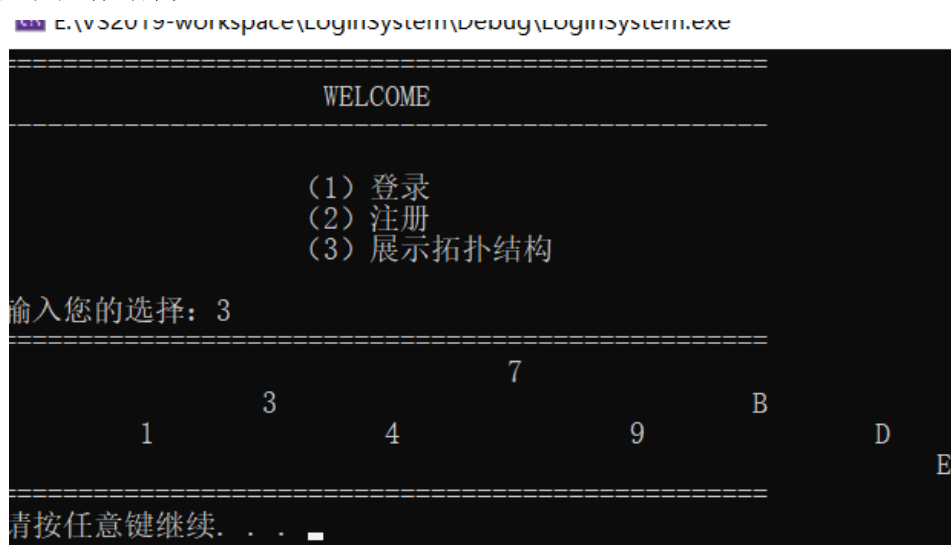


图 46 展示拓扑结构

4.2 边界数据测试

1. 插入函数

a) 左旋

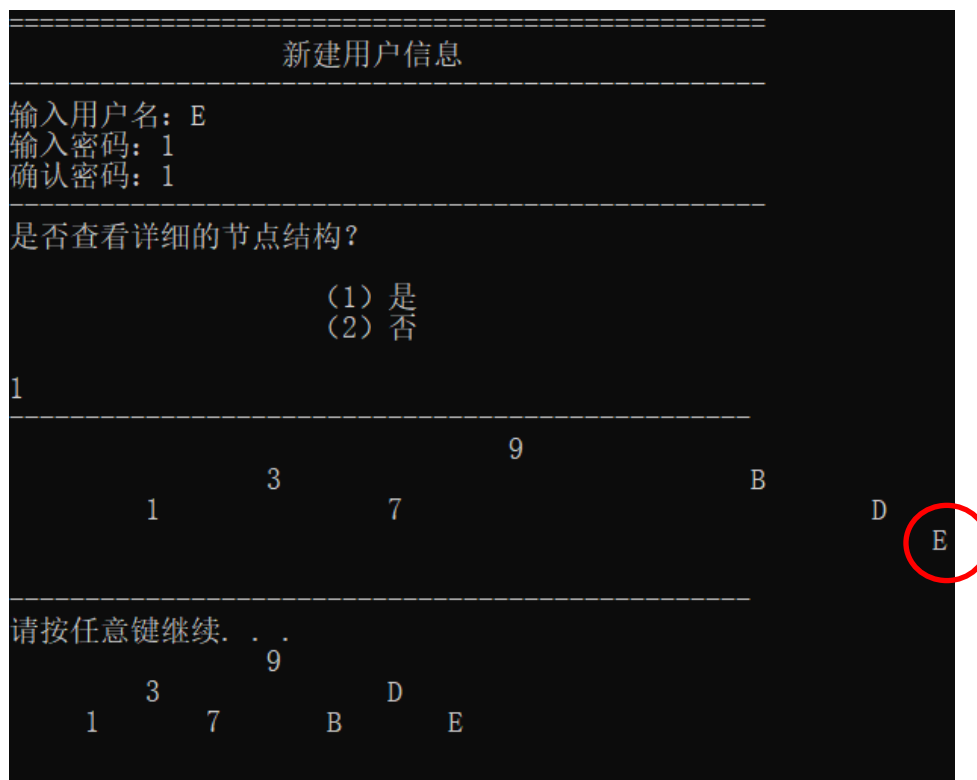


图 47 简单左旋

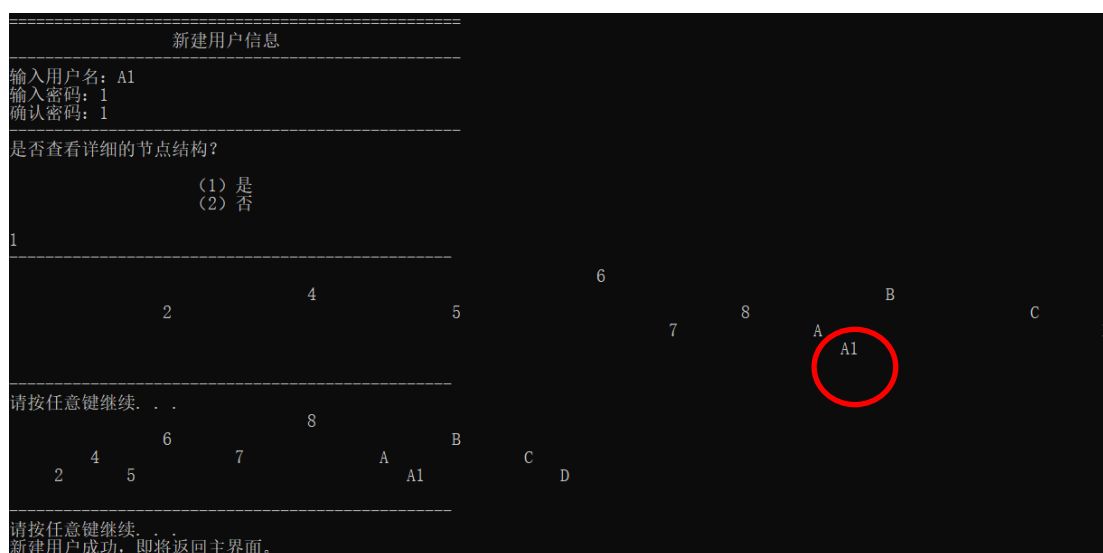


图 48 带左子树的左旋

b) 右旋

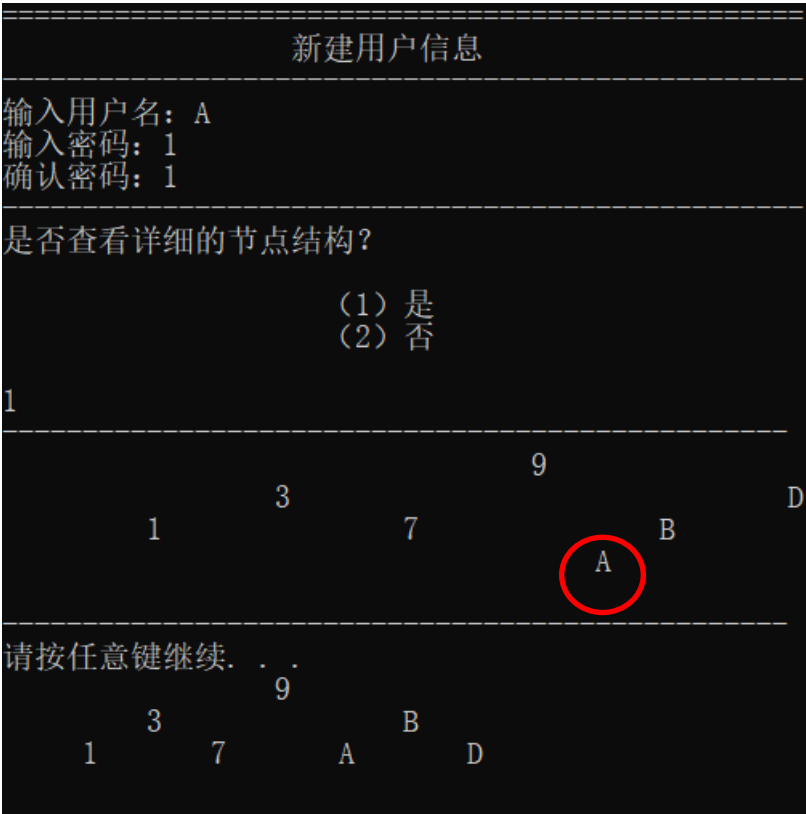


图 49 简单右旋



图 50 带右子树的右旋

c) 左-右旋

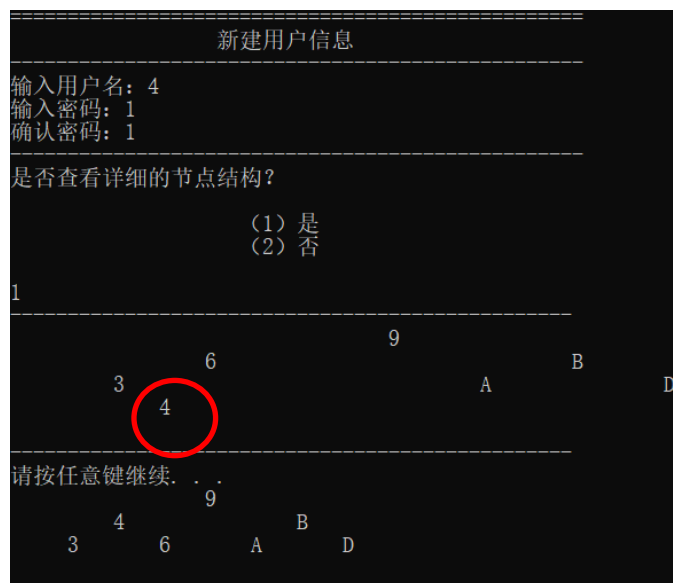


图 51 简单左-右旋

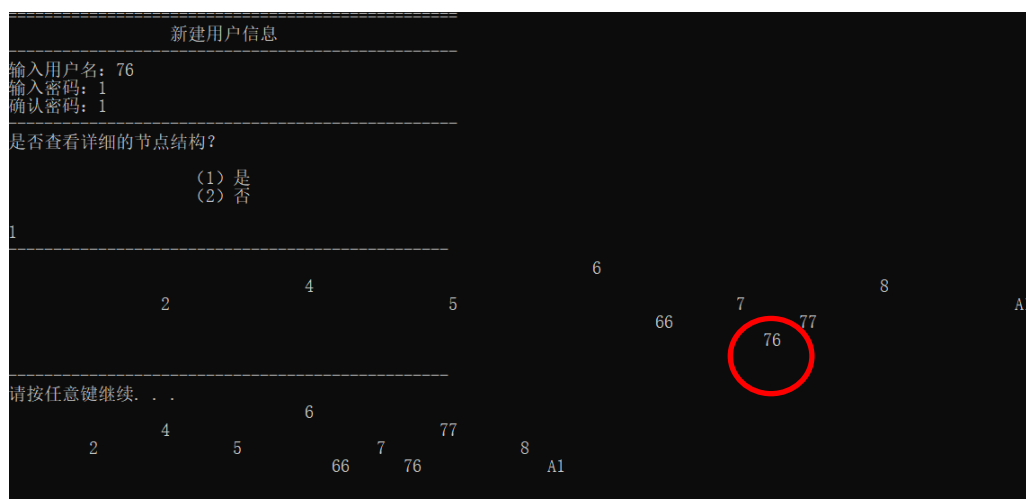


图 52 带左子树的左-右旋

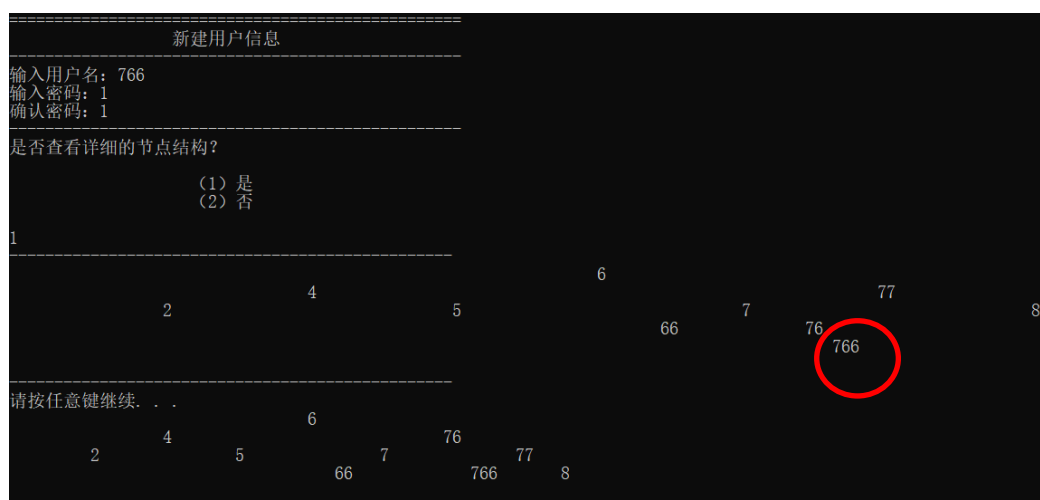


图 53 带右子树的左-右旋

d) 右-左旋

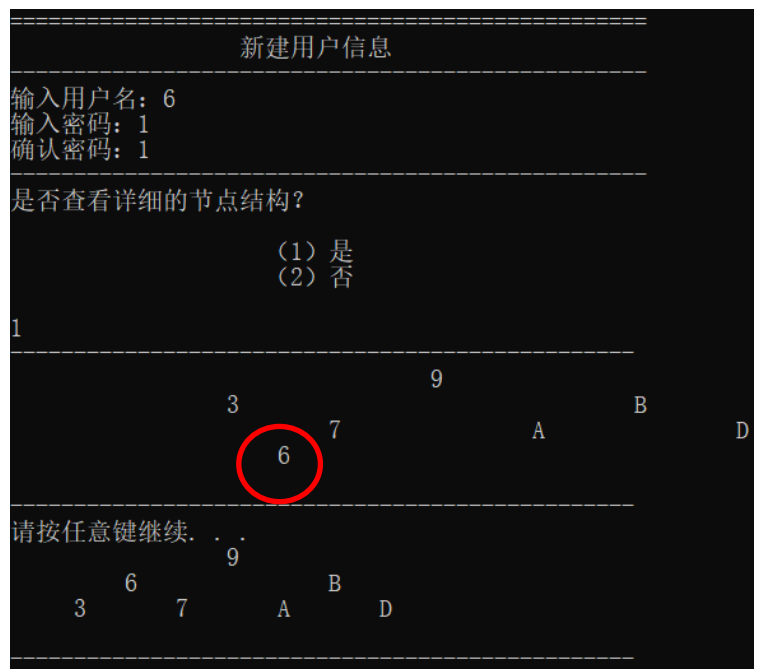


图 54 简单右-左旋

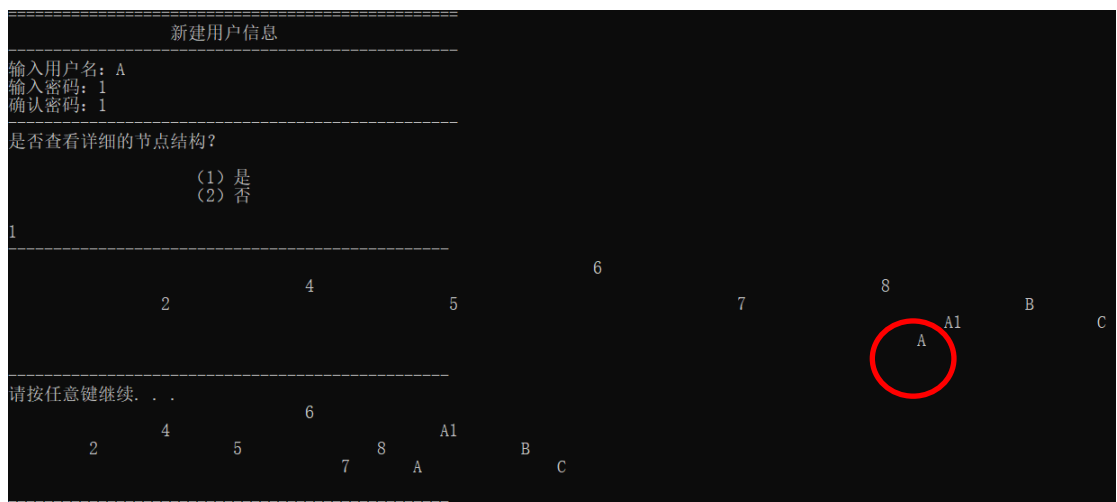


图 55 带左子树的右-左旋

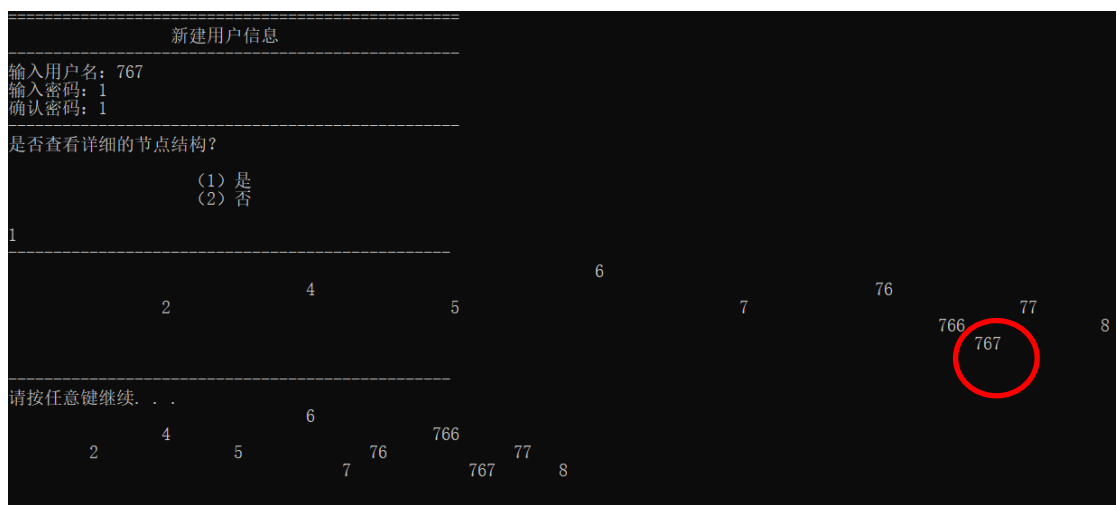


图 56 带右子树的右-左旋

2. 删除函数

a) 删除叶节点

i. 不引起不平衡

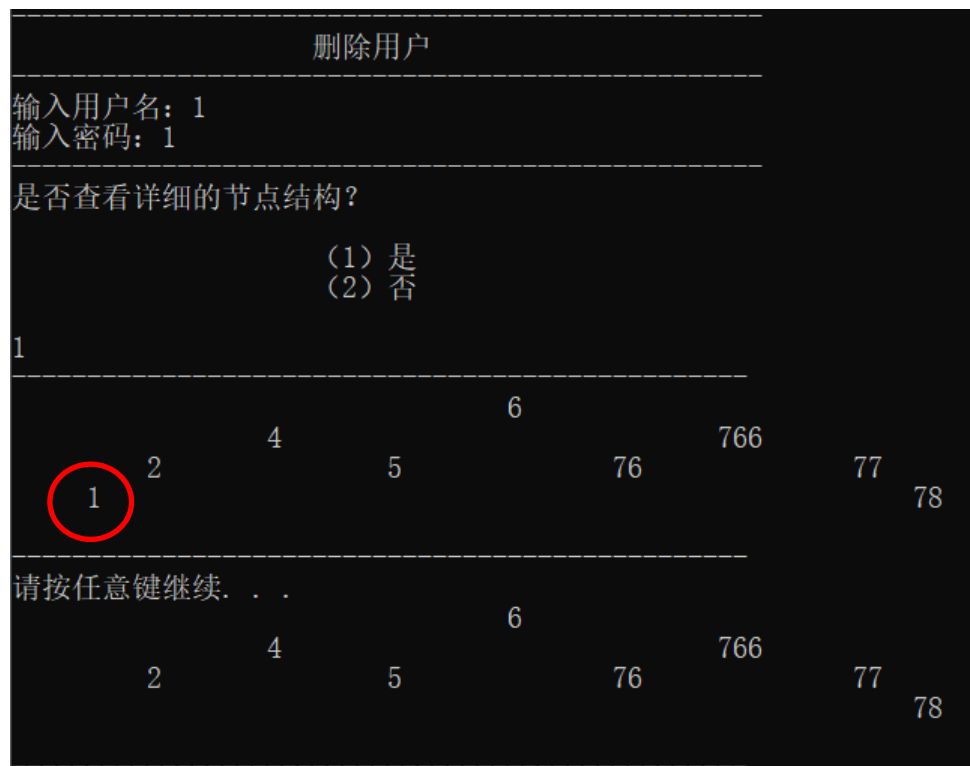


图 57 左

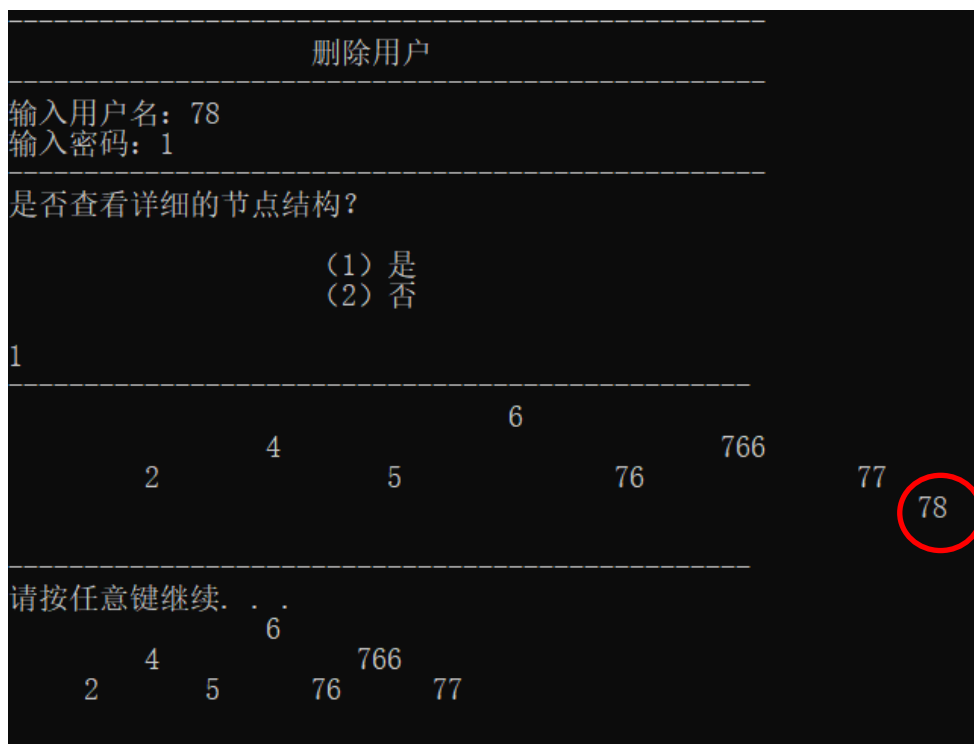


图 58 右

ii. 引起不平衡

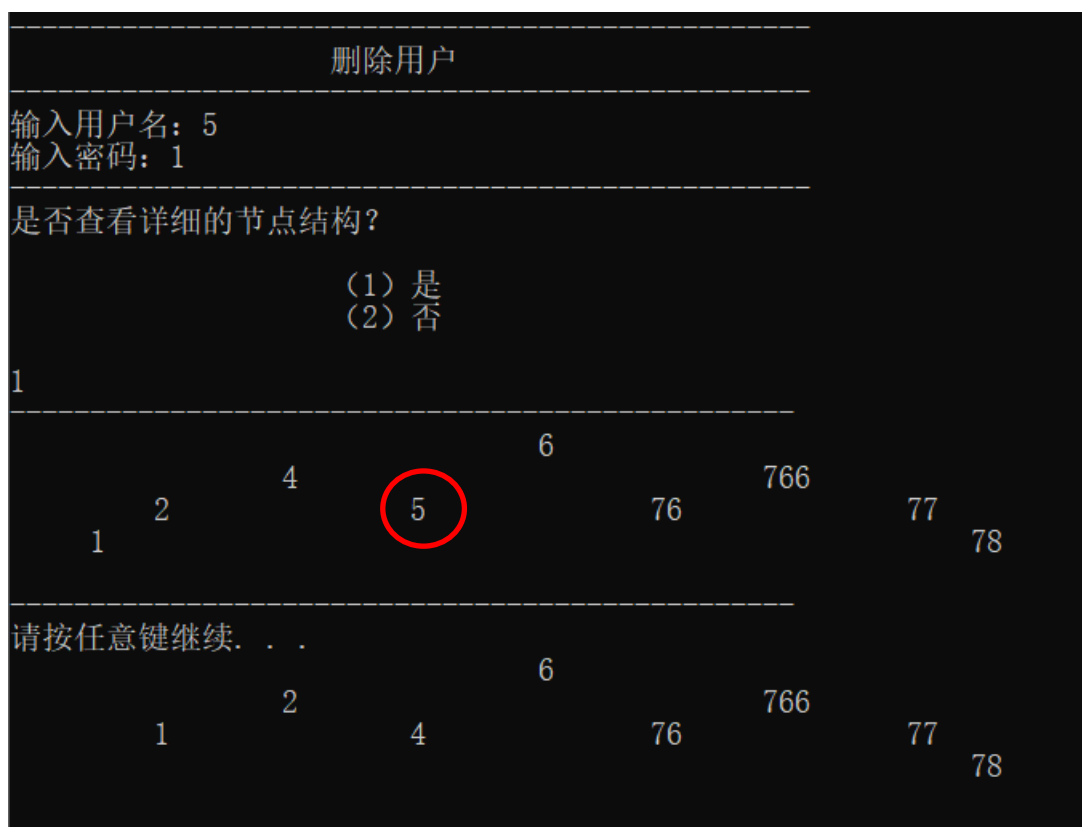


图 59 右

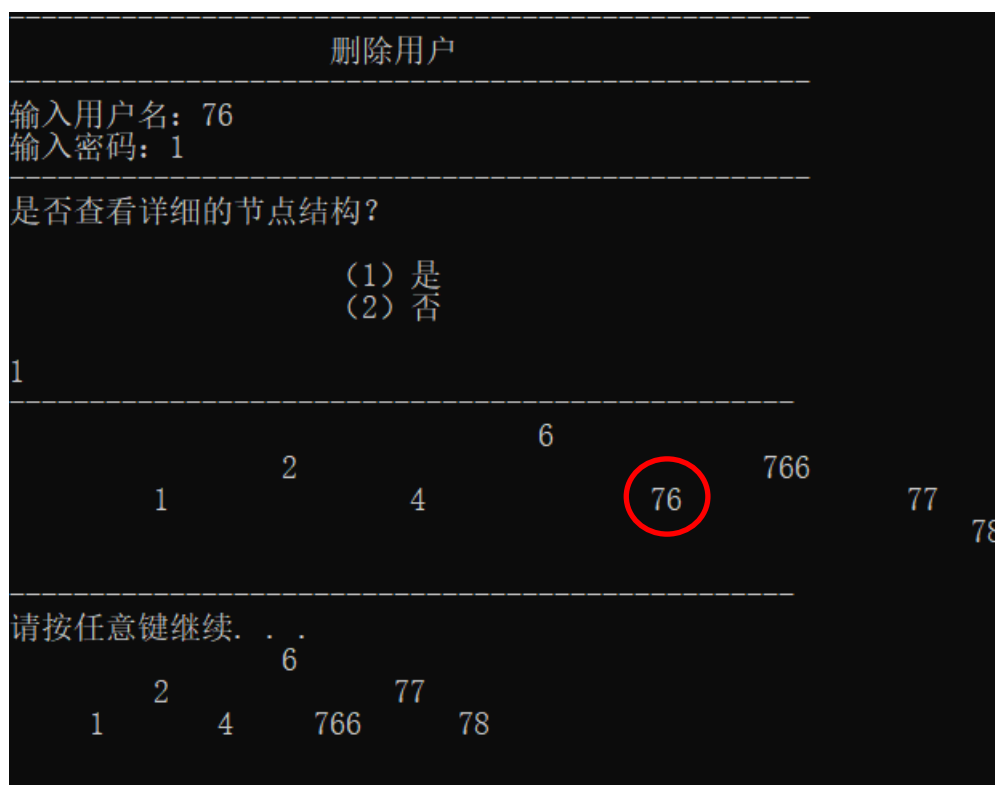


图 60 左

b) 删除 1 子女的非叶节点



图 61 该节点只有左孩子

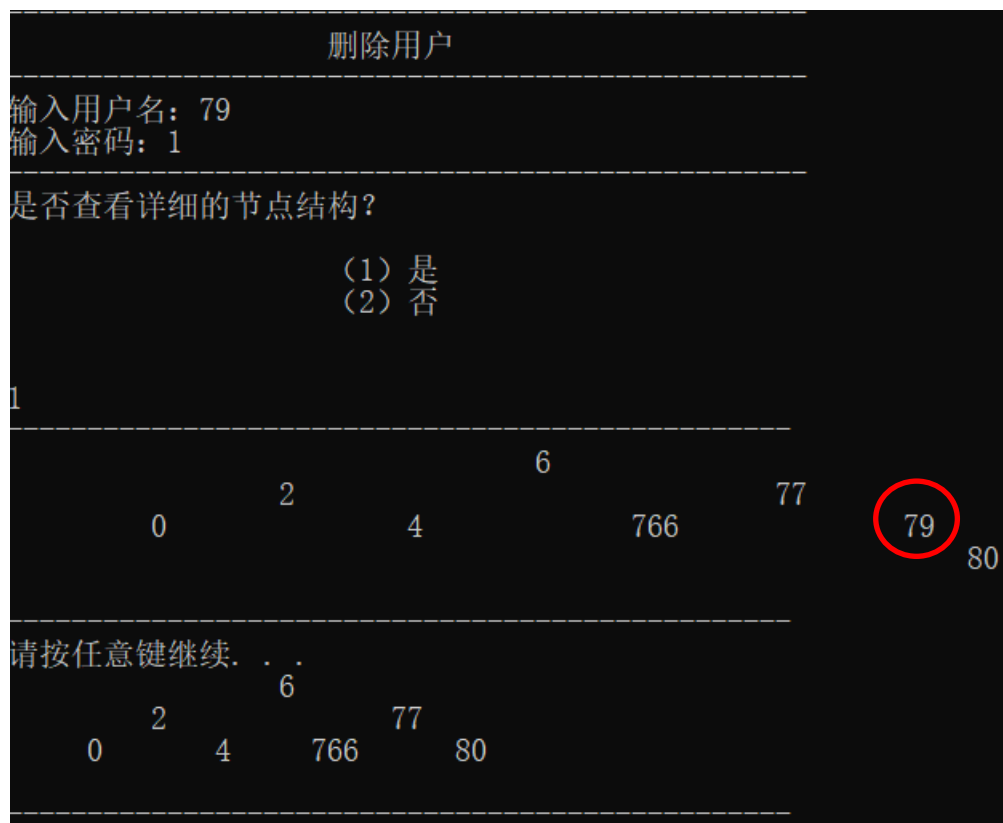


图 62 该节点只有右孩子

c) 删除 2 子女的非叶节点

删除用户

输入用户名: 2
 输入密码: 1

是否查看详细的节点结构?

(1) 是
 (2) 否

1

0 2 4 6 77 80
 766

请按任意键继续. . .

0 4 6 77 80
 766

请按任意键继续. . .

图 63 非根节点

删除用户

输入用户名: 6
 输入密码: 1

是否查看详细的节点结构?

(1) 是
 (2) 否

1

0 4 6 77 80
 766

请按任意键继续. . .

0 4 766 77 80

图 64 删除根节点

d) 删除操作特有的不平衡情况

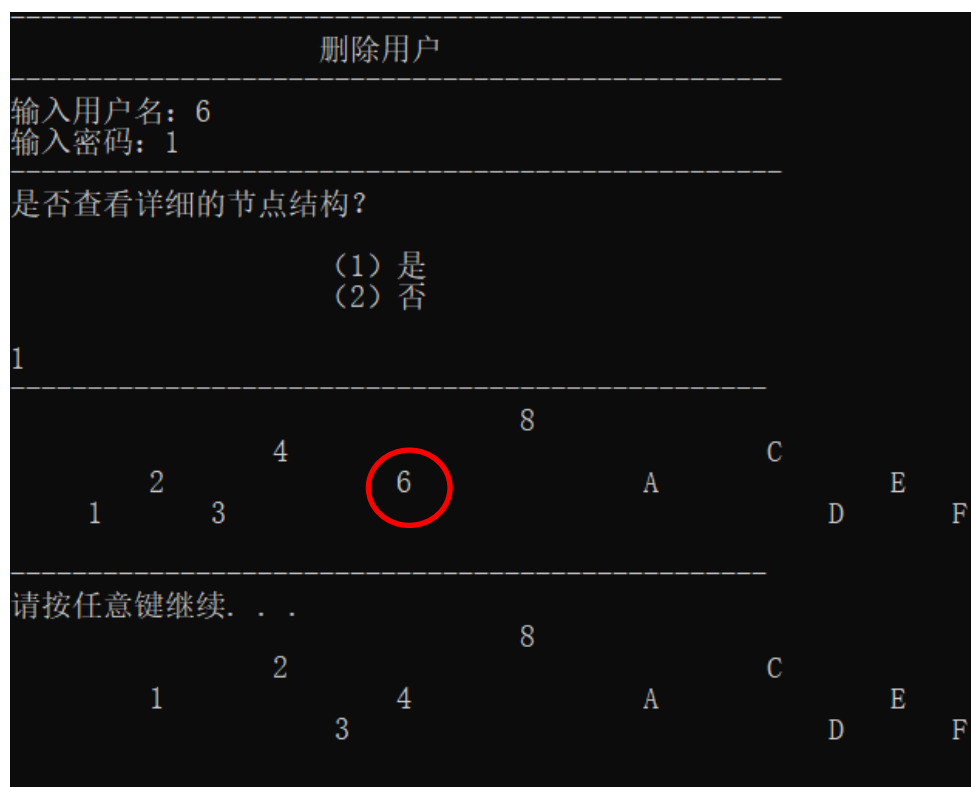


图 65 右

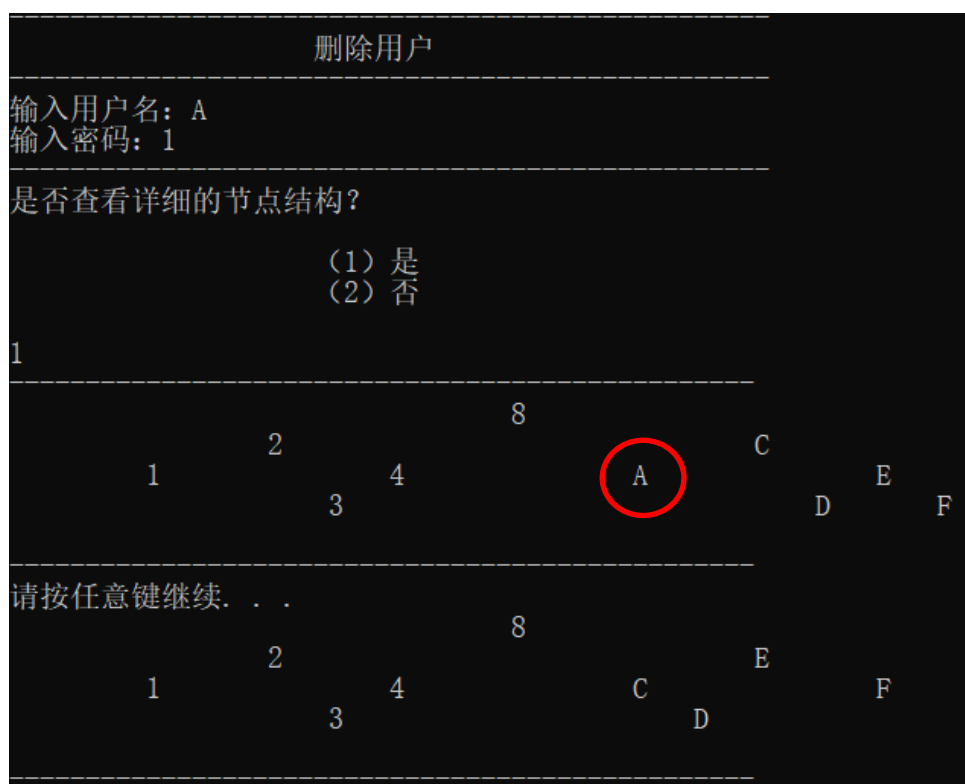


图 66 左

其他不平衡情况与插入基本一致，故不赘述。

4.3 错误数据测试

1. 进入登录界面

```
=====
WELCOME
=====
(1) 登录
(2) 注册
(3) 展示拓扑结构
输入您的选择: 4
=====
输入错误, 请重新输入!
```

图 67 提示错误信息并在 1s 后返回原界面

2. 登录

```
=====
WELCOME
=====
(1) 登录
(2) 注册
(3) 展示拓扑结构
输入您的选择: 1
=====
用户登录
=====
输入用户名: 00
输入密码: 1
=====
该用户不存在, 是否注册新用户?
(1) 是
(2) 否
```

图 68 用户不存在, 则提示注册或退出

```
=====
WELCOME
=====
(1) 登录
(2) 注册
(3) 展示拓扑结构
输入您的选择: 1
=====
用户登录
=====
输入用户名: 1
输入密码: 111
=====
密码错误, 是否需要重新输入?
(1) 是
(2) 否
```

图 69 密码错误, 则提示重输或退出

3. 注册

```
=====
                                WELCOME
=====

                                (1) 登录
                                (2) 注册
                                (3) 展示拓扑结构

输入您的选择: 2
=====
                                新建用户信息
=====

输入用户名: 0
输入密码: 1
确认密码: 2
=====
两次密码不一致, 是否需要重新输入?

                                (1) 是
                                (2) 否
```

图 70 两次输入密码不一致, 则提示重输或退出

```
=====
                                WELCOME
=====

                                (1) 登录
                                (2) 注册
                                (3) 展示拓扑结构

输入您的选择: 2
=====
                                新建用户信息
=====

输入用户名: 1
输入密码: 1
确认密码: 1
=====
该账号已存在, 即将返回主界面。
=====
```

图 71 提示账号已存在并在 1s 后返回原界面

```
=====
                                新建用户信息
=====

输入用户名: 0
输入密码: 1
确认密码: 1
=====
是否查看详细的节点结构?

                                (1) 是
                                (2) 否

3
=====
新建用户成功, 即将返回主界面。
=====
```

图 72 是否查看节点结构这里, 输错即认为选“否”

4. 进入用户界面

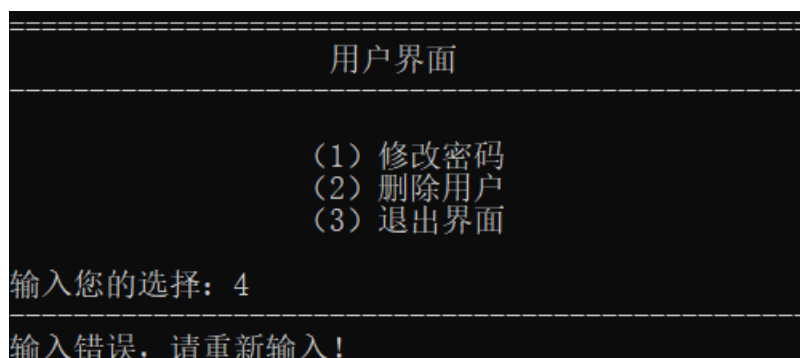


图 73 提示错误信息并在 1s 后返回原界面

5. 修改密码

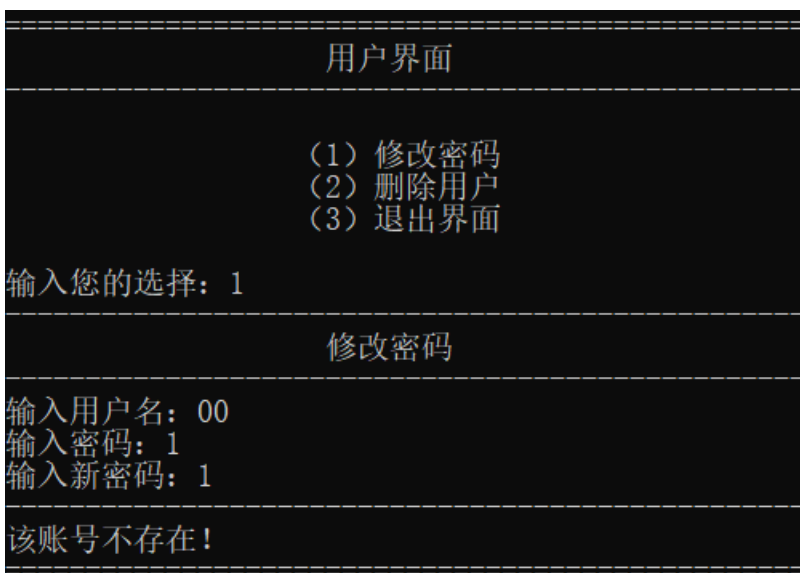


图 74 提示账号不存在并在 1s 后返回原界面

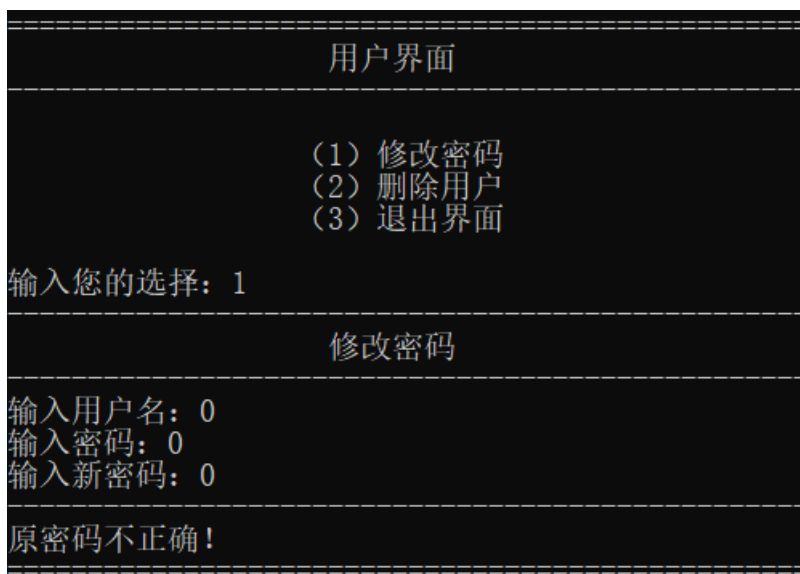


图 75 提示原密码不正确并在 1s 后返回原界面

6. 删除用户

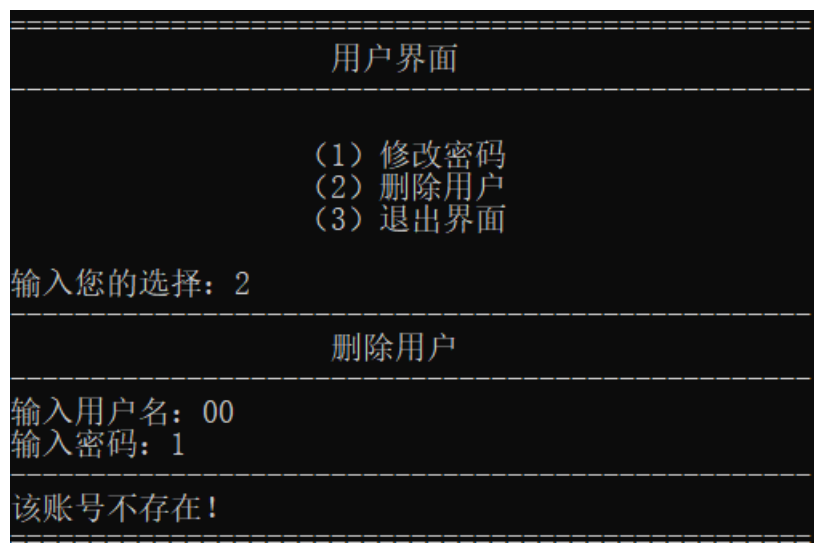


图 76 提示账号不存在并在 1s 后返回原界面

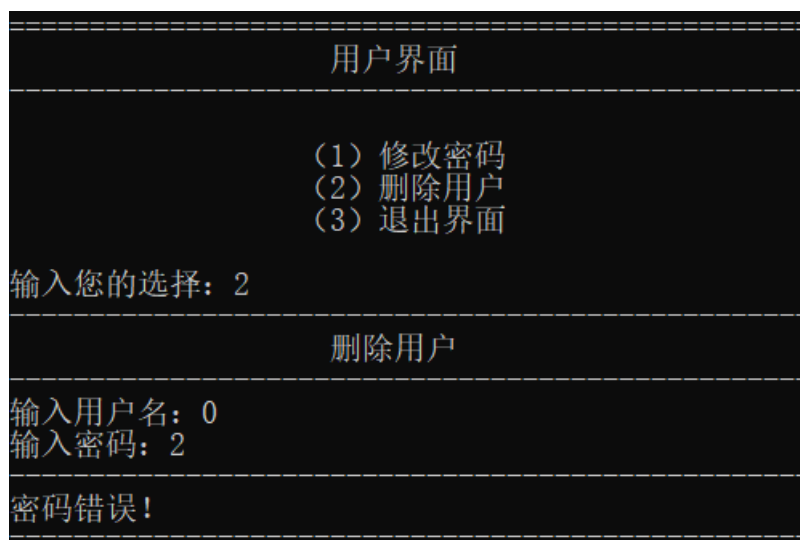


图 77 提示密码错误并在 1s 后返回原界面

5 附录

以下为源代码。

userInfo.h

```
#pragma once
#include<iostream>
#include<string>
using namespace std;

//----- 包含用户信息的节点类 -----
class UserInfo
{
public:
    string userId, userPw;                // 用户id和密码
    int balanceFactor, height;            // 平衡因子和高度
    UserInfo* left, * right, * parent;    // 指向左右子女、双亲节点的指针

    UserInfo() :balanceFactor(0), height(0),
               left(0), right(0), parent(0) {};                // 默认构造函数
    UserInfo(const string& id, const string& password) :
               userId(id), userPw(password), balanceFactor(0), height(0),
               left(0), right(0), parent(0) {};                // 显式构造函数
    int getBalanceFactor();                // 获取节点的平衡因子
    int getHeight();                       // 获取子树的高度
private:
    int getHeightAux(UserInfo* root);      // getHeight的辅助函数
};

//--- 获取子树的高度
inline int UserInfo::getHeight() {
    return getHeightAux(this);
}
```

userTree.h

```
#pragma once
#include"userInfo.h"
#include<iostream>
#include<fstream>
#include<string>
```



```

using namespace std;

//----- 包含所有用户信息的类 -----
class UserTree
{
public:
    //***** 函数成员 *****
    UserTree() :myRoot(0), n(0) {} // 默认构造函数
    UserTree(const string& f) :myRoot(0), filename(f), n(0) {} // 重载构造函数
    bool empty() const { return myRoot == 0; } // 判空函数
    void insert(); // 插入函数
    void read(); // 输入函数
    void write(ostream& out); // 输出函数
    void inorder(ostream& out) const; // 中序遍历函数
    bool login(); // 登录函数
    void remove(); // 删除函数
    void search(const string& id, bool& found,
        UserInfo*& locptr) const; // 查找函数
    void update(); // 修改密码函数
    void graph() const; // 图形输出函数
    void getBF(); // 获取各节点的平衡因子
    ~UserTree(); // 析构函数

private:
    //***** 私有函数成员 *****
    void inorderAux(ostream& out, UserInfo* subroot) const; // inorder的辅助函数
    void insertAux(const bool& print,
        const string& id, const string& password); // insert的辅助函数
    void getBFAux(UserInfo* subroot); // getBF的辅助函数
    void LL(UserInfo* userB); // 左旋操作
    void RR(UserInfo* userB); // 右旋操作
    void LR(UserInfo* userC); // 左-右旋操作
    void RL(UserInfo* userC); // 右-左旋操作
    void destAux(UserInfo* subtreeRoot); // 析构函数的辅助函数

    //***** 数据成员 *****
    UserInfo* myRoot; // 指向根节点
    string filename; // 存储的文件名
    int n; // 结点数量
};

//--- 中序遍历函数
inline void UserTree::inorder(ostream& out) const {
    inorderAux(out, myRoot);
}

```

```

}

//--- 获取各节点的平衡因子
inline void UserTree::getBF() {
    getBFAux(myRoot);
}

//--- 析构函数
inline UserTree::~UserTree()
{
    destAux(myRoot);
}

```

userInfo.cpp

```

#include "userInfo.h"
#include <iostream>
using namespace std;

//--- 获取节点的平衡因子
int UserInfo::getBalanceFactor()
{
    // 平衡因子等于该节点左子树高度减去右子树高度
    balanceFactor = this->left->getHeight() - this->right->getHeight();
    return balanceFactor;
}

//--- getHeight的辅助函数
int UserInfo::getHeightAux(UserInfo* root)
{
    {
        if (root == 0) {
            return 0;
        }
        else {
            int a = getHeightAux(root->left) + 1;           // 递归过程中由0开始逐次加1
            int b = getHeightAux(root->right) + 1;
            height = a > b ? a : b;                          // 每一轮中取较大的作为子树高度
            return height;
        }
    }
}

```

userTree.cpp

```

#include "userTree.h"

```

```

#include<iomanip>
#include<iostream>
#include<fstream>
#include<queue>
#include<vector>
#include<Windows.h>
using namespace std;

class UserInfo;

//--- 插入函数
void UserTree::insert()
{
    string id, password, pw;

    // 输入信息
    while (1) {
        cout << setfill(' ') << setw(31) << "新建用户信息\n";
        cout << "-----\n";
        cout << "输入用户名: ";
        cin >> id;
        cout << "输入密码: ";
        cin >> password;
        cout << "确认密码: ";
        cin >> pw;
        cout << "-----\n";
        // 确认密码
        if (password != pw) {
            cout << "两次密码不一致, 是否需要重新输入? \n\n";
            cout << setfill(' ') << setw(28) << " (1) 是\n";
            cout << setfill(' ') << setw(29) << " (2) 否\n\n";
            int t;
            cin >> t;
            if (t == 2)
                return;
            system("cls");
        }
        else {
            ++n;
            // 第一个参数表示输出提示信息
            insertAux(1, id, password);
            break;
        }
    }
}

```

```

        Sleep(1000);
    }

//--- 输入函数
void UserTree::read()
{
    string id, password;

    // 打开连接到包含合法用户信息的文件的流
    ifstream in(filename.data(), ios::in);
    if (!in.is_open()) {
        cerr << filename << "文件无法打开\n";
        exit(1);
    }

    int i;
    in >> i;
    n = i;
    //读取用户信息
    for (; i > 0; --i) {
        in >> id >> password;
        // 第一个参数表示不输出提示信息
        insertAux(0, id, password);
    }
}

//--- 输出函数
void UserTree::write(ostream& out)
{
    inorder(out); // 中序输出
}

//--- 登录函数
bool UserTree::login()
{
    string id, password;
    bool found;
    UserInfo* locptr = myRoot;

    // 输入信息
    while (1) {
        cout << setfill(' ') << setw(29) << "用户登录\n";
        cout << "-----\n";
    }
}

```

```

        cout << "输入用户名: ";
        cin >> id;
        cout << "输入密码: ";
        cin >> password;
        cout << "-----\n";
        // 查找节点
        search(id, found, locptr);
        if (!found || locptr == 0) {
            cout << "该用户不存在, 是否注册新用户? \n";
            cout << setfill(' ') << setw(28) << " (1) 是\n";
            cout << setfill(' ') << setw(29) << " (2) 否\n\n";
            int t;
            cin >> t;
            cout << "===== \n";
            if (t == 1)
                insert();
            else
                return false;
        }
        else if (locptr->userPw != password) {
            cout << "密码错误, 是否需要重新输入? \n\n";
            cout << setfill(' ') << setw(28) << " (1) 是\n";
            cout << setfill(' ') << setw(29) << " (2) 否\n\n";
            int t;
            cin >> t;
            if (t == 2)
                break;
            system("cls");
        }
        else {
            cout << "登录成功! \n";
            cout << "===== \n";
            return true;
        }
    }
    return false;
}

//--- 删除函数
void UserTree::remove()
{
    string id, password;
    bool draw = false;
    bool found;

```

```

UserInfo* x = myRoot;

// 输入信息
cout << setfill(' ') << setw(29) << "删除用户\n";
cout << "-----\n";
cout << "输入用户名: ";
cin >> id;
cout << "输入密码: ";
cin >> password;
cout << "-----\n";

// 查找 x 的位置
search(id, found, x);
if (!found || x == 0) {
    cout << "该账号不存在! \n";
    cout << "===== \n";
    return;
}
if (password != x->userPw) {
    cout << "密码错误! \n";
    cout << "===== \n";
    return;
}

// 图形输出
cout << "是否查看详细的节点结构? \n\n";
cout << setfill(' ') << setw(28) << " (1) 是\n";
cout << setfill(' ') << setw(29) << " (2) 否\n\n";
int t;
cin >> t;
cout << "-----\n";
t == 1 ? draw = true : draw = false;
if (draw) {
    graph();
    cout << "\n-----\n";
    system("pause");
}

// 该节点有 2 个子女时
if (x->left != 0 && x->right != 0) {
    // 查找 x 的中序后继
    UserInfo* xSucc = x->right;
    while (xSucc->left != 0)                // 下降到左子树
        xSucc = xSucc->left;
}

```

```

    // 将 xSucc 的内容移至 x，修改 x 为指向将被删除的后继
    x->userId = xSucc->userId;
    x->userPw = xSucc->userPw;
    x = xSucc;
    // 自此将情况统一于 0 或 1 子女节点的删除
}
// 该节点有 1 个或没有子女时
UserInfo* subtree = x->left;
if (subtree == 0)
    subtree = x->right;
if (x->parent == 0) // x 为将被删除的根节点
    myRoot = subtree; // 实际上子树本就平衡无需调整
else if (x->parent->left == x) {
    x->parent->left = subtree; // 以 x 的左\右子女将其替换
    if(subtree)
        subtree->parent = x->parent;
}
else {
    x->parent->right = subtree;
    if(subtree)
        subtree->parent = x->parent;
}

// 获取各节点的平衡因子
getBF();

UserInfo* locptr = x;
UserInfo* parent = locptr->parent;
// 重新平衡
while (parent) { // 删除点不为根节点时
    if (parent->balanceFactor == 2) { // 此时一定是右端删除
        // 子节点平衡因子为 0 时，也采用右旋
        if (parent->left &&
            (parent->left->balanceFactor == 1 || parent->left->balanceFactor == 0))
            RR(parent->left); // 右旋
        else if (parent->left && parent->left->balanceFactor == -1)
            LR(parent->left); // 左-右旋
    }
    else if (parent->balanceFactor == -2) { // 此时一定是左端删除
        if (parent->right && parent->right->balanceFactor == 1)
            RL(parent->right); // 右-左旋
        // 子节点平衡因子为 0 时，也采用左旋
        else if (parent->right &&
            (parent->right->balanceFactor == -1 || parent->right->balanceFactor == 0))

```

```

        LL(parent->right);                // 左旋
    }

    // 向上搜索不平衡的节点直到搜索到根
    locptr = parent;
    parent = locptr->parent;
}

// 释放内存
delete x;
--n;

// 打开连接到包含合法用户信息的文件的流
ofstream out(filename.data());
if (!out.is_open()) {
    cerr << filename << "文件无法打开\n";
    exit(1);
}
// 更新文件内容
out << n << endl;
write(out);

// 图形输出
if (draw) {
    graph();
    cout << "\n-----\n";
    system("pause");
}

cout << "删除用户成功! \n";
cout << "=====\n";
}

//--- 修改密码函数
void UserTree::update()
{
    // 输入信息
    string id, password, newpw;
    cout << setfill(' ') << setw(29) << "修改密码\n";
    cout << "-----\n";
    cout << "输入用户名: ";
    cin >> id;
    cout << "输入密码: ";
    cin >> password;
}

```



```

    cout << "输入新密码: ";
    cin >> newpw;
    cout << "-----\n";

    // 查找位置
    bool found;
    UserInfo* x = myRoot;
    search(id, found, x);
    if (!found) {
        cout << "该账号不存在! \n";
        cout << "===== \n";
        return;
    }

    if (password == x->userPw) {
        x->userPw = newpw;

        // 打开连接到包含合法用户信息的文件的流
        ofstream out(filename.data());
        if (!out.is_open()) {
            cerr << filename << "文件无法打开\n";
            exit(1);
        }
        // 更新文件内容
        out << n << endl;
        write(out);

        cout << "修改密码成功! \n";
    }
    else
        cout << "原密码不正确! \n";
    cout << "===== \n";
}

//--- 图形输出函数
void UserTree::graph() const
{
    if (myRoot != 0) {
        queue<UserInfo*> q;
        int h = myRoot->getHeight();           // h >= 1
        int n = 0, depth = 0;
        UserInfo* none = new UserInfo(" ", " "); // 节点值为空格, 用于占位

        for (q.push(myRoot); h > 0 && !q.empty(); q.pop()) {

```

```

        UserInfo* locptr = q.front();
        int t = 2 * (int(pow(2, h)) - 1);
        int u = t + 4;                                // 包括了每个输出的 id 所占长度

        // 输出节点
        cout << setfill(' ') << setw(u) << locptr->userId;
        cout << setfill(' ') << setw(t) << ' ';

        if (locptr->left)
            q.push(locptr->left);
        else
            q.push(none);                             // 发现无左\右子女，插入空格节点占位
        if (locptr->right)
            q.push(locptr->right);
        else
            q.push(none);                             // 发现无左\右子女，插入空格节点占位

        // 位于该层最后的节点时
        if (++n == int(pow(2, depth))) {
            n = 0;
            ++depth;
            --h;
            cout << endl;
        }
    }
}

else {
    cout << "没有数据! \n";
}
}

//--- 查找函数
void UserTree::search(const string& id, bool& found, UserInfo*& locptr) const
{
    found = false;
    while (!found && locptr != 0) {
        if (id < locptr->userId)
            locptr = locptr->left;
        else if (id > locptr->userId)
            locptr = locptr->right;
        else
            found = true;
    }
}

```

```
//--- inorder 的辅助函数
void UserTree::inorderAux(ostream& out, UserInfo* subroot) const
{
    if (subroot != 0) {
        inorderAux(out, subroot->left);
        out << subroot->userId << " " << subroot->userPw << endl;
        inorderAux(out, subroot->right);
    }
}

//--- insert 的辅助函数
void UserTree::insertAux(const bool& print, const string& id, const string& password)
{
    bool found = false;
    bool draw = false;
    UserInfo* locptr = myRoot, * parent = 0;

    // 查找插入位置
    while (!found && locptr != 0) {
        parent = locptr;
        if (id < locptr->userId)
            locptr = locptr->left;
        else if (id > locptr->userId)
            locptr = locptr->right;
        else
            found = true;
    }

    // 插入节点
    if (!found) {
        locptr = new UserInfo(id, password);
        if (parent == 0)
            myRoot = locptr; // 空树
        else if (id < parent->userId) {
            parent->left = locptr;
            locptr->parent = parent;
        }
        else {
            parent->right = locptr;
            locptr->parent = parent;
        }
    }

    // 图形输出
}
```

```

if (print) {
    cout << "是否查看详细的节点结构? \n\n";
    cout << setfill(' ') << setw(28) << "(1) 是\n";
    cout << setfill(' ') << setw(29) << "(2) 否\n\n";
    int t;
    cin >> t;
    cout << "-----\n";
    t == 1 ? draw = true : draw = false;
    if (draw) {
        graph();
        cout << "\n-----\n";
        system("pause");
    }
}

// 获取各节点的平衡因子
getBF();

// 重新平衡
while (parent) {
    // 默认插入前树是平衡的，则平衡因子为 0 时认为到达根或已平衡
    if (parent->balanceFactor == 0)
        break;
    // 向上搜索不平衡的节点
    else if (parent->balanceFactor == 1 || parent->balanceFactor == -1) {
        locptr = parent;
        parent = locptr->parent;
    }
    // 对符合条件的进行旋转
    // 由于是逐个插入，故插入节点上不会存在子节点平衡因子为 0 而父节点为 ±2 的情况
    else {
        if (parent->balanceFactor == 2) { // 此时一定是左端插入
            if (locptr->balanceFactor == 1)
                RR(locptr); // 右旋
            else if (locptr->balanceFactor == -1)
                LR(locptr); // 左-右旋
        }
        else if (parent->balanceFactor == -2) { // 此时一定是右端插入
            if (locptr->balanceFactor == 1)
                RL(locptr); // 右-左旋
            else if (locptr->balanceFactor == -1)
                LL(locptr); // 左旋
        }
    }
    if (print && draw) { // 图形输出

```

```

        graph();
        cout << "\n-----\n";
        system("pause");
    }
    break;
}

}

if (print) {
    // 打开连接到包含合法用户信息的文件的流
    ofstream out(filename.data());
    if (!out.is_open()) {
        cerr << filename << "文件无法打开\n";
        exit(1);
    }
    // 更新文件信息
    out << n << endl;
    write(out);

    cout << "新建用户成功，即将返回主界面。\\n";
    cout << "=====\\n";
}

}

else
{
    if (print) {
        cout << "该账号已存在，即将返回主界面。\\n";
        cout << "=====\\n";
    }
}

}

//--- getBF 的辅助函数
void UserTree::getBFAux(UserInfo* subroot)
{
    // 中序遍历获得每个节点的平衡因子
    if (subroot != 0) {
        getBFAux(subroot->left);
        subroot->getBalanceFactor();
        getBFAux(subroot->right);
    }
}

//--- 左旋操作
void UserTree::LL(UserInfo* userB)
{

```

```

UserInfo* userA = userB->parent;
if (!(userA->right == userB && userB->right != 0)) {
    cerr << "不符合左旋条件\n";
    return;
}

bool flag = false;
if (userA == myRoot)
    flag = true;
// A 的双亲节点对应指针指向 B
if (userA->parent != 0) {
    if (userA->parent->left == userA)
        userA->parent->left = userB;
    else
        userA->parent->right = userB;
}

// 修改 AB 之间的指针
userB->parent = userA->parent;
userA->parent = userB;
userA->right = userB->left;
if (userB->left)
    userB->left->parent = userA;
userB->left = userA;
// A 是根节点，则旋转后根节点指针指向 B
if (flag)
    myRoot = userB;
}

//--- 右旋操作
void UserTree::RR(UserInfo* userB)
{
    UserInfo* userA = userB->parent;
    if (!(userA->left == userB && userB->left != 0)) {
        cerr << "不符合右旋条件\n";
        return;
    }

    bool flag = false;
    if (userA == myRoot)
        flag = true;
// A 的双亲节点对应指针指向 B
if (userA->parent != 0) {
    if (userA->parent->left == userA)
        userA->parent->left = userB;
    else
        userA->parent->right = userB;
}

```

```

    }
    // 修改 AB 之间的指针
    userB->parent = userA->parent;
    userA->parent = userB;
    userA->left = userB->right;
    if (userB->right)
        userB->right->parent = userA;
    userB->right = userA;
    // A 是根节点，则旋转后根节点指针指向 B
    if (flag)
        myRoot = userB;
}

//--- 左-右旋操作
void UserTree::LR(UserInfo* userB)
{
    // 先左旋
    UserInfo* userC = userB->right;
    if (userC == 0) {
        cerr << "不符合左-右旋条件\n";
        return;
    }
    UserInfo* userA = userB->parent;
    if (userA == 0 || userA->left != userB) {
        cerr << "不符合左-右旋条件\n";
        return;
    }
    userC->parent = userA;
    userB->parent = userC;
    userB->right = userC->left;
    if (userC->left)
        userC->left->parent = userB;
    userC->left = userB;
    userA->left = userC;
    // 再右旋
    RR(userC);
}

//--- 右-左旋操作
void UserTree::RL(UserInfo* userB)
{
    // 先右旋
    UserInfo* userC = userB->left;
    if (userC == 0) {

```

```

        cerr << "不符合右-左旋条件\n";
        return;
    }
    UserInfo* userA = userB->parent;
    if (userA == 0 || userA->right != userB) {
        cerr << "不符合右-左旋条件\n";
        return;
    }
    userC->parent = userA;
    userB->parent = userC;
    userB->left = userC->right;
    if (userC->right)
        userC->right->parent = userB;
    userC->right = userB;
    userA->right = userC;
    // 再左旋
    LL(userC);
}

void UserTree::destAux(UserInfo* subroot)
{
    // 后序遍历，依次释放
    if (subroot != 0) {
        destAux(subroot->left);
        destAux(subroot->right);
        delete subroot;
    }
}

```

main.cpp

```

#include "userTree.h"
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
#include <Windows.h>
using namespace std;

int main()
{
    // 创建用户登陆系统的二叉树
    string filename = "userinfo.txt";
    UserTree* tree = new UserTree(filename);
}

```



```

tree->read();

//主界面
while (1) {
    // 登录界面
    cout << "=====\\n";
    cout << setfill(' ') << setw(29) << "WELCOME\\n";
    cout << "-----\\n\\n";
    cout << setfill(' ') << setw(29) << " (1) 登录\\n";
    cout << setfill(' ') << setw(29) << " (2) 注册\\n";
    cout << setfill(' ') << setw(38) << " (3) 展示拓扑结构\\n\\n";
    cout << "输入您的选择: ";
    int choosel;
    cin >> choosel;
    cout << "=====\\n";

    // 用户界面
    if (choosel == 1) {
        bool flag = tree->login();
        if (flag) {
            system("pause");
            system("cls");
            while (1) {
                cout << "=====\\n";
                cout << setfill(' ') << setw(29) << "用户界面\\n";
                cout << "-----\\n\\n";
                cout << setfill(' ') << setw(32) << " (1) 修改密码\\n";
                cout << setfill(' ') << setw(32) << " (2) 删除用户\\n";
                cout << setfill(' ') << setw(33) << " (3) 退出界面\\n\\n";
                cout << "输入您的选择: ";
                int choose2;
                cin >> choose2;
                cout << "-----\\n";

                if (choose2 == 1)
                    tree->update();
                else if (choose2 == 2) {
                    tree->remove();
                }
                else if (choose2 == 3)
                    break;
                else
                    cout << "输入错误, 请重新输入! \\n";
            }
        }
    }
}

```

```

        // 清屏
        Sleep(1000);
        system("cls");
    }
}

else if (choosel == 2)
    tree->insert();
else if (choosel == 3) {
    tree->graph();
    cout << "=====\n";
    system("pause");
}
else
    cout << "输入错误，请重新输入！\n";

// 清屏
Sleep(1000);
system("cls");
}
}

```