
浙江工业大学

数据结构课程设计报告



学 号 201806061108

姓 名 胡皓睿

班 级 计实 1801

完成日期 2019. 12. 16

目录

1	实验题目与要求.....	2
2	设计思路.....	3
2.1	系统总体设计.....	3
2.2	系统功能设计.....	4
2.2.1	用户登陆验证功能.....	4
2.2.2	添加用户功能.....	4
2.2.3	显示用户功能.....	4
2.2.4	删除用户功能.....	4
2.2.5	修改密码功能.....	4
2.3	类的设计.....	5
2.3.1	AvlTree 类.....	5
2.3.2	Node 类.....	13
2.3.3	ShadowTreeNode 类.....	13
2.3.4	ShadowTreeNodeQueue 类.....	13
2.4	主程序的设计.....	14
3	调试分析.....	15
3.1	技术难点分析.....	15
3.2	调试错误分析.....	15
4	测试结果分析.....	16
4.1	登陆.....	16
4.2	管理员界面.....	16
4.2.1	正向打印树.....	17
4.2.2	添加用户.....	17
4.2.3	删除用户.....	18
4.3	一般用户.....	19
4.3.1	修改密码.....	19
5	技术亮点分析.....	20
5.1	在不同的时候合理使用二叉树不同的遍历方式.....	20
5.2	使用影子树实现二叉树的正向打印.....	21
5.3	使用 GitHub 进行版本控制.....	22
5.4	使用 Microsoft Visio 进行流程图的绘制.....	23
6	附录.....	24
6.1	main.cpp.....	24
6.2	AvlTree.h.....	29
6.3	AvlTree.cpp.....	31
6.4	Node.h.....	38
6.5	Node.cpp.....	39
6.6	ShadowTreeNode.h.....	40
6.7	ShadowTreeNodeQueue.h.....	41
6.8	ShadowTreeNodeQueue.cpp.....	42

1 实验题目与要求

【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；

【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；

【运行结果要求】要求能够实现用户登录验证、添加用户、删除用户和更新用户密码功能，实验报告要求有详细的功能测试截图。

验收要求：1、通过管理员权限，正向打印用户信息的平衡二叉查找树；

2、管理员在删除、插入用户信息后，需重新旋转调整 AVL 树，并正向打印；

3、用户只能看到自己的信息，只能更改密码。

【考核要求】要求程序能正常运行，全面完成题目要求。

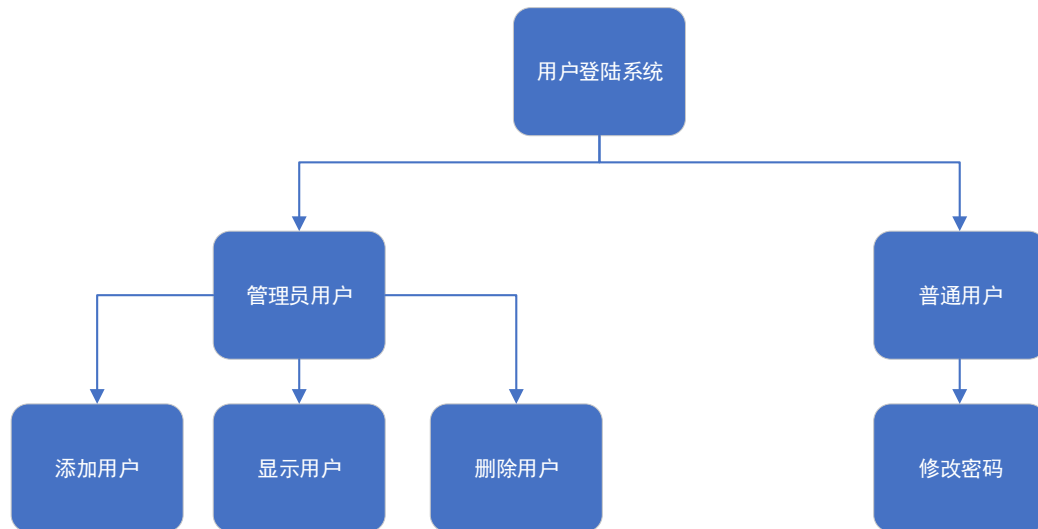
【题目难度】 难，成绩等级高

2 设计思路

2.1 系统总体设计

系统需要实现“用户登陆系统”的功能，由于线性表的时间复杂度为 $O(n)$ ，效率较低，不足以满足大容量高并发的需求，所以使用二叉搜索树的结构进行数据存储。由于二叉存储树可能存在不平衡的问题，影响数据结构的效率，所以我们使用 AVL 树进行存储。

2.2 系统功能设计



2.2.1 用户登陆验证功能

用户输入用户名和密码，系统会在数据库中查找该用户。如果用户不存在或者密码错误，发出相应的提示，否则根据用户所属的类型（管理员/普通用户），完成登陆

2.2.2 添加用户功能

管理员用户有权限添加用户，输入要添加的用户名和密码，如果在数据库中存在同名用户，提示添加失败，否则将用户信息存放到数据库里

2.2.3 显示用户功能

管理员用户有权限显示用户，使用打印命令，可以正向打印出 Avl 树中存储的用户信息。

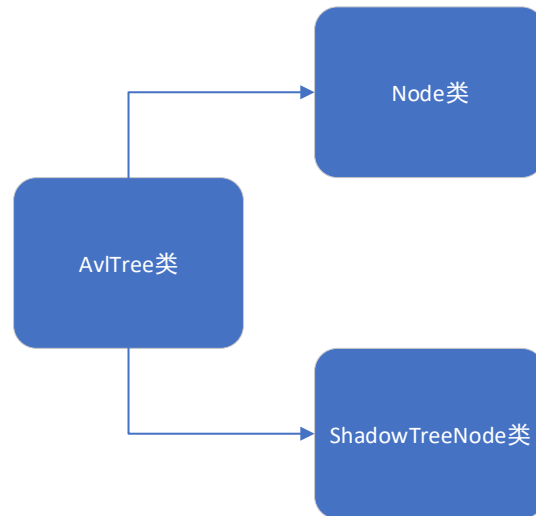
2.2.4 删除用户功能

管理员用户有权限删除用户，输入要删除的用户名称，如果数据库中存在该用户，就将其删除，否则提示错误。

2.2.5 修改密码功能

普通用户可以更改自己的密码

2.3 类的设计



2.3.1 AVLTree 类

AVLTree 类定义了一棵 AVL 树，包含了 Node 节点和 ShadowTreeNode 节点，包含了用影子树正向打印树，旋转节点，添加节点等功能。

2.3.1.1 Save 函数

```
9 void AVLTree::save(Node *tree) {
10     if (tree == nullptr) { //如果根节点是空指针，说明这棵树已经完成了遍历，即可退出
11         return;
12     }
13     fstream out;
14     if (tree == root) { //如果该节点是根节点，说明文件输出刚刚开始，则清除原有文件，否则在文件尾追加
15         out.open( _Filename: "database.dat", ios::out);
16     } else {
17         out.open( _Filename: "database.dat", _Mode: ios::out | ios::app);
18     }
19     out << tree->getName() << " " << tree->getPassword() << endl;
20     out.close();
21     save(tree->left); //递归遍历左子树
22     save(tree->right); //递归遍历右子树
23 }
```

在 save 函数中，我们使用了 fstream 对象来进行文件的输出，使用了中序遍历的方法，先保存中间节点的内容，然后递归保存左右子树。使用中序遍历而非其他遍历的好处是，在读取文件重新构建树的时候，前序遍历保存的文件可以避免树的旋转。

2.3.1.2 AddNode 函数

```

40 Node *AvlTree::addNode(Node *pNode, string &name, string &password) {
41     if (pNode == nullptr) { //找到空的插入点, 创建一个新的Node, 并返回插入点
42         pNode = new Node(&name, &password);
43         return pNode;
44     } else if (name < pNode->getName()) { //要插入左边
45         pNode->left = addNode(pNode->left, &name, &password);
46         if (getHeight(pNode->left) - getHeight(pNode->right) == 2) { //平衡因子为2, 发生了不平衡
47             if (name < pNode->left->getName()) { //添加的节点在左子树的左子树内, 采用右旋操作
48                 pNode = RightSpin(pNode);
49             } else { //添加的节点在左子树的右子树内, 采用先左旋再右旋操作
50                 pNode->left = LeftSpin(pNode->left);
51                 pNode = RightSpin(pNode);
52             }
53         }
54     } else {
55         pNode->right = addNode(pNode->right, &name, &password);
56         if (getHeight(pNode->right) - getHeight(pNode->left) == 2) { //平衡因子为2, 发生了不平衡
57             if (name > pNode->right->getName()) { //添加的节点在右子树的右子树内, 采用左旋操作
58                 pNode = LeftSpin(pNode);
59             } else { //添加的节点在右子树的左子树内, 采用先右旋再左旋操作
60                 pNode->right = RightSpin(pNode->right);
61                 pNode = LeftSpin(pNode);
62             }
63         }
64     }
65     pNode->setHeight(max(getHeight(pNode->left), getHeight(pNode->right)) + 1); //获取完成插入后新的树的高度
66     return pNode;
67 }

```

在插入新的节点时, 我们需要一层一层向下寻找插入点, 进行插入。完成插入后, 如果树失衡了, 要使用 [Spin](#) 函数进行调整。

2.3.1.3 Remove 函数

```

80 Node *AvlTree::remove(Node *root, const string &name) {
81     if (root == nullptr) return nullptr; //如果节点是空的, 说明要删除的东西不存在, 返回空指针
82     if (name < root->getName()) { //如果节点的内容比查找值大, 说明目标节点在左子树, 递归进入
83         root->left = remove(root->left, name);
84         if (getHeight(root->right) - getHeight(root->left) == 2) { //如果平衡因子为2, 说明发生了失衡, 需要旋转
85             Node *right = root->right; //右子树比较深, 说明右子树需要进行旋转
86             if (getHeight(right->left) > getHeight(right->right)) { //如果右子树的左子树比较深, 执行先右旋再左旋
87                 root->right = RightSpin(root->right);
88                 root = LeftSpin(root);
89             } else { //否则直接左旋
90                 root = LeftSpin(root);
91             }
92         }
93     } else if (name > root->getName()) { //如果用户名比当前节点要大, 说明要删除的节点在右子树, 递归进入
94         root->right = remove(root->right, name);
95         if (getHeight(root->left) - getHeight(root->right) == 2) { //如果平衡因子为2, 说明发生了失衡, 需要旋转
96             Node *left = root->left; //左子树比较深, 说明左子树需要进行旋转
97             if (getHeight(left->left) < getHeight(left->right)) { //如果左子树的右子树比较深, 执行先左旋再右旋
98                 root->left = LeftSpin(root->left);
99                 root = RightSpin(root);
100             } else { //否则直接右旋
101                 root = RightSpin(root);
102             }
103         }
104     }
105 }

```

```

104     } else { //既不偏大，也不偏小，显然是找到了删除点
105         if (root->left != nullptr && root->right != nullptr) { //两侧都存在子树
106             if (getHeight(root->left) > getHeight(root->right)) { //如果左子树比较深
107                 Node *toRemove = maxNode(root->left); //删除点的内容会被左子树的最大值替换，所以找到左子树的最大值
108                 root->setName(toRemove->getName()); //将根节点（也就是要删除的节点）的信息替换为左子树的最大节点
109                 root->setPassword(toRemove->getPassword());
110                 root->left = remove(root->left, toRemove->getName()); //删除左子树最大的节点
111             } else {
112                 Node *toRemove = minNode(root->right); //删除点的内容会被右子树的最小值替换，所以找到右子树的最小值
113                 root->setName(toRemove->getName());
114                 root->setPassword(toRemove->getPassword());
115                 root->right = remove(root->right, toRemove->getName());
116             }
117         } else { //如果至少有一侧是空的，那就直接把一棵子树提高
118             Node *tmp = root;
119             if (root->left != nullptr) { //如果左子树存在，就将右节点提高
120                 root = root->left;
121             } else { //否则将左节点提高
122                 root = root->right;
123             }
124             delete tmp; //删除根节点
125         }
126     }
127     if (root != nullptr) {
128         root->setHeight(max(getHeight(root->left), getHeight(root->right)) + 1);
129     }
130     return root;
131 }

```

在执行删除操作的时候，我们也一层一层向下找，找到那个要被删除的节点。然后，视情况，在左子树或右子树找到最大或最小的节点，将其移到根节点的位置，再将原来的节点删除，即实现了删除特定节点的功能。完成删除后，如果树发生了失衡，要调用 [Spin](#) 函数进行调整

2.3.1.4 Print 函数

```

147 void AvlTree::print() {
148     if (root == nullptr) return; //如果根节点不存在，显然是不需要打印的，返回
149     ShadowTreeNode *ShadowTree = nullptr;
150     ShadowTree = ShadowTreeBuild(Tree: root, ShadowTree, TreeRow: 1); //开始时，应该打印在第一行
151     ShadowTreeNodeQueue queue; //建立一个打印队列
152     queue.push(ShadowTree);
153     int MaxRow = 1, MaxColumn = 0;
154     int MaxLength = getMaxLength(pTree: this); //设置内容的最大长度，以对齐树
155     while (!queue.empty()) { //如果打印队列非空
156         ShadowTreeNode *pShadowTreeNode = queue.front(); //取出队列第一项
157         queue.pop();
158         if (pShadowTreeNode->row > MaxRow) { //如果行数超过了最大行数，说明要开始打印下一行
159             cout << endl; //换行
160             MaxRow = pShadowTreeNode->row; //重设最大行数
161             MaxColumn = 0;
162         }
163         string tmpName = pShadowTreeNode->name;
164         while (tmpName.length() < MaxLength) { //检查名字有没有过长，如果太长，就将其截断，否则在后面补上空格
165             tmpName += ' ';
166             if (tmpName.length() < MaxLength) {
167                 tmpName = string(_Ptr: " ").append(tmpName);
168             }
169         }

```



```

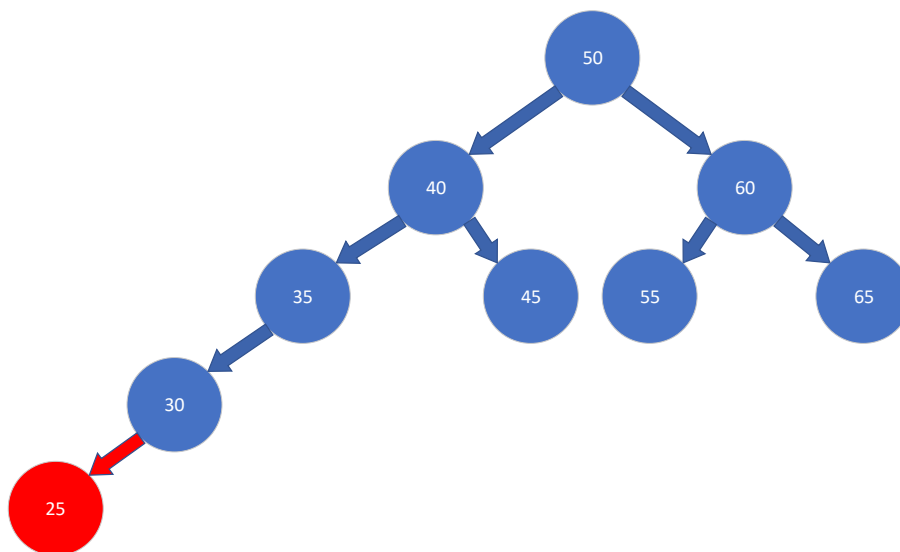
170     for (int i = 1; i < pShadowTreeNode->column - MaxColumn; i++) { //输出前置的空格
171         for (int j = 0; j < MaxLength; j++) cout << ' ';
172     }
173     cout << tmpName;
174     MaxColumn = pShadowTreeNode->column;
175     if (pShadowTreeNode->left != nullptr) queue.push(pShadowTreeNode->left); //如果左子树非空, 将其加入打印队列
176     if (pShadowTreeNode->right != nullptr) queue.push(pShadowTreeNode->right);
177 }
178 cout << endl;
179 }

```

Print 函数使用了影子树实现 Avl 树的正向打印。影子树是通过保存树中每一个节点的横纵坐标信息,来正向打印二叉树的一种数据结构。通过影子树打印,可以获取更好的打印效果。

2.3.1.5 Spin 函数

LeftSpin 函数和 RightSpin 函数互成镜像,所以我只介绍右旋。



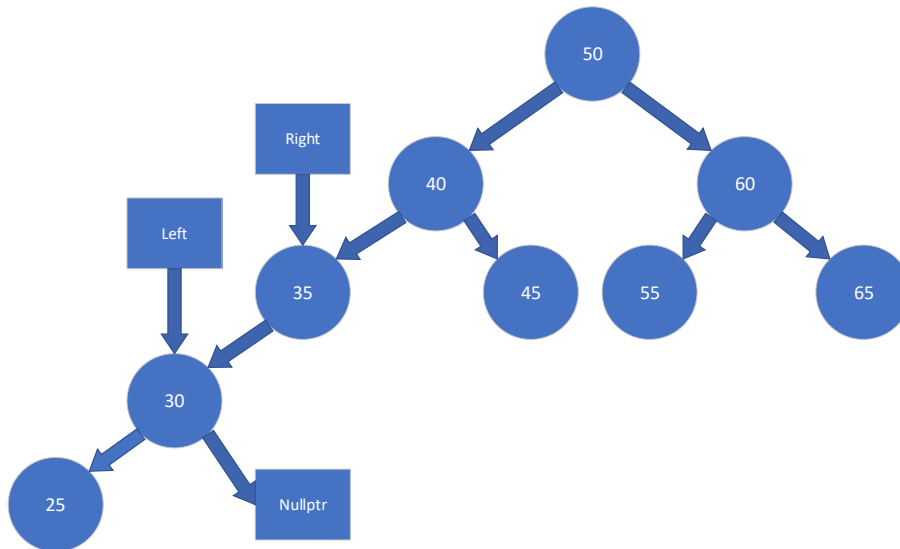
看这棵树,它原本是平衡的。由于节点 25 插入,这棵树变的不平衡。25 被插在 35 节点左子树的左子树上,所以我们只需要进行一次右旋操作。

```

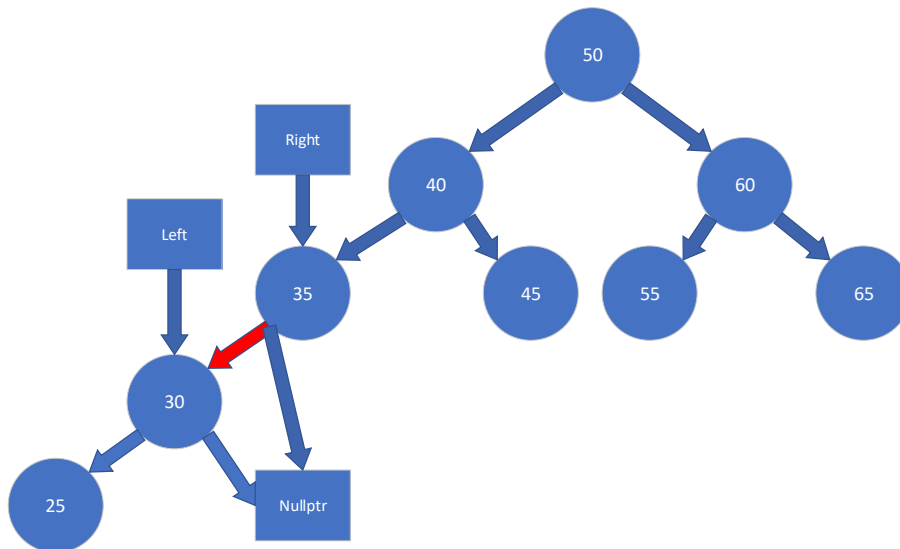
197 Node *AvlTree::RightSpin(Node *root) {
198     Node *left = root->left;
199     root->left = left->right;
200     left->right = root;
201     root->setHeight(max(getHeight(root->left), getHeight(root->right)) + 1); //旋转后, 重设树的高度
202     left->setHeight(max(getHeight(left->left), getHeight(left->right)) + 1);
203     return left;
204 }

```

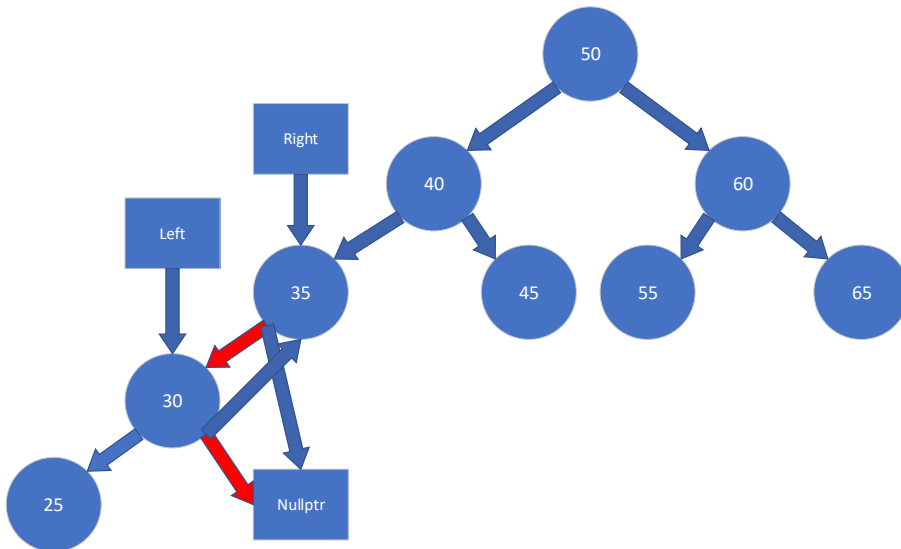
首先，使用 left 指针指向左节点，即 30



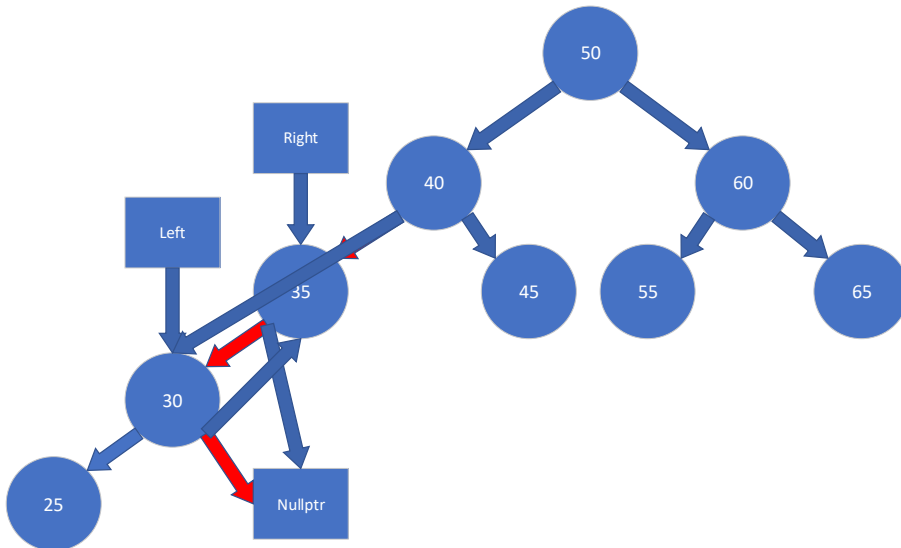
将根节点的左指针指向 30 节点的 right。可以看到，尽管 30 节点对应的右指针是 nullptr，我在这里将其画出，所得结构显得清晰。



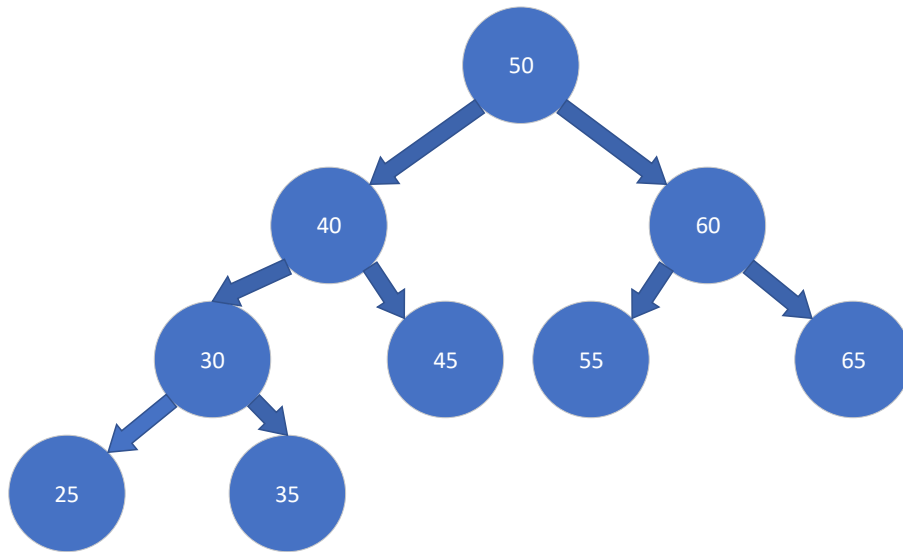
将 left 节点的右指针指向 root



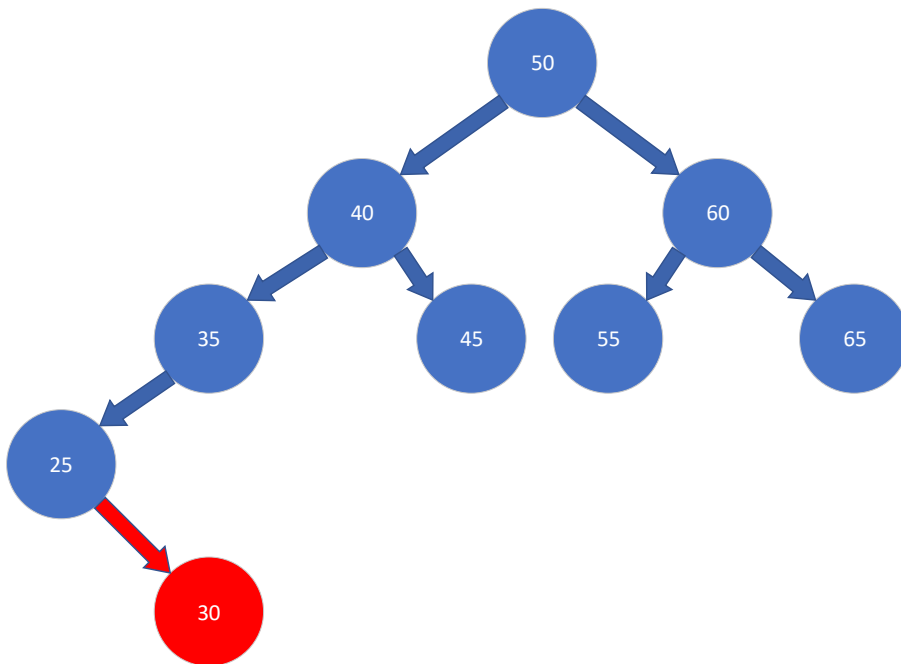
返回左节点的指针，让上层节点对应的指针指向前面提到的 left 节点



整理树的结构，删除已经消失的指针的之前加上的 nullptr

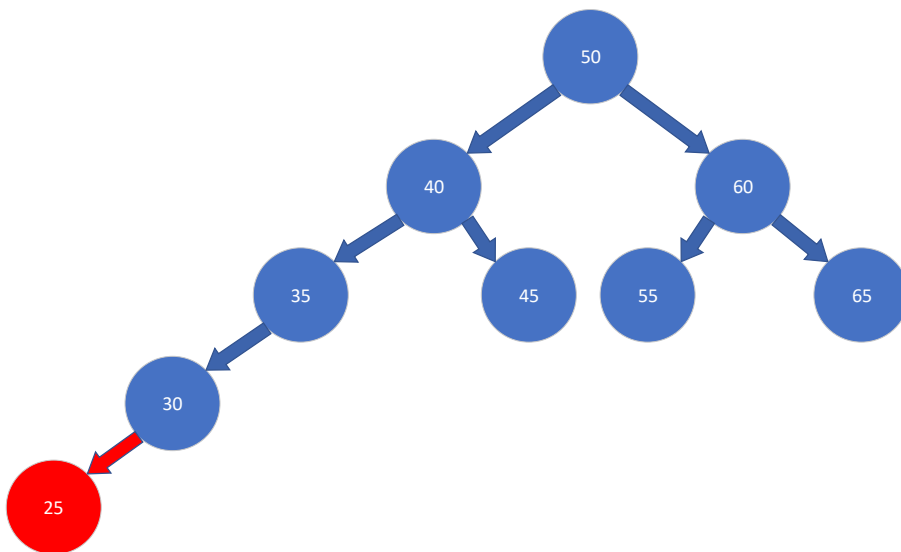


可以看到，我们成功完成了右旋的操作。



看这一棵树，它也是不平衡的，但是它不平衡的原因是在左子树的右子树插入了一个新的节点。对于这种情况，我们要先左旋再右旋。

左旋，指的是对根节点的左子树左旋，经过一次左旋，这棵树就会变成和前面的一样。



然后，对它进行一次右旋操作，即可完成平衡过程。先右旋再左旋的情况也同理。

2.3.2 Node 类

Node 类定义了 Avl 树的节点，包含了 name, password, height 三个字段，分别存储用户名，密码，以及这个节点所处的高度。基于数据访问控制的要求，这三个字段使用了 private 访问权限，所以各自实现了 set 和 get 方法，进行修改和输出。同时，包含了指向两个子节点的 left 和 right 指针，用于实现数据结构。

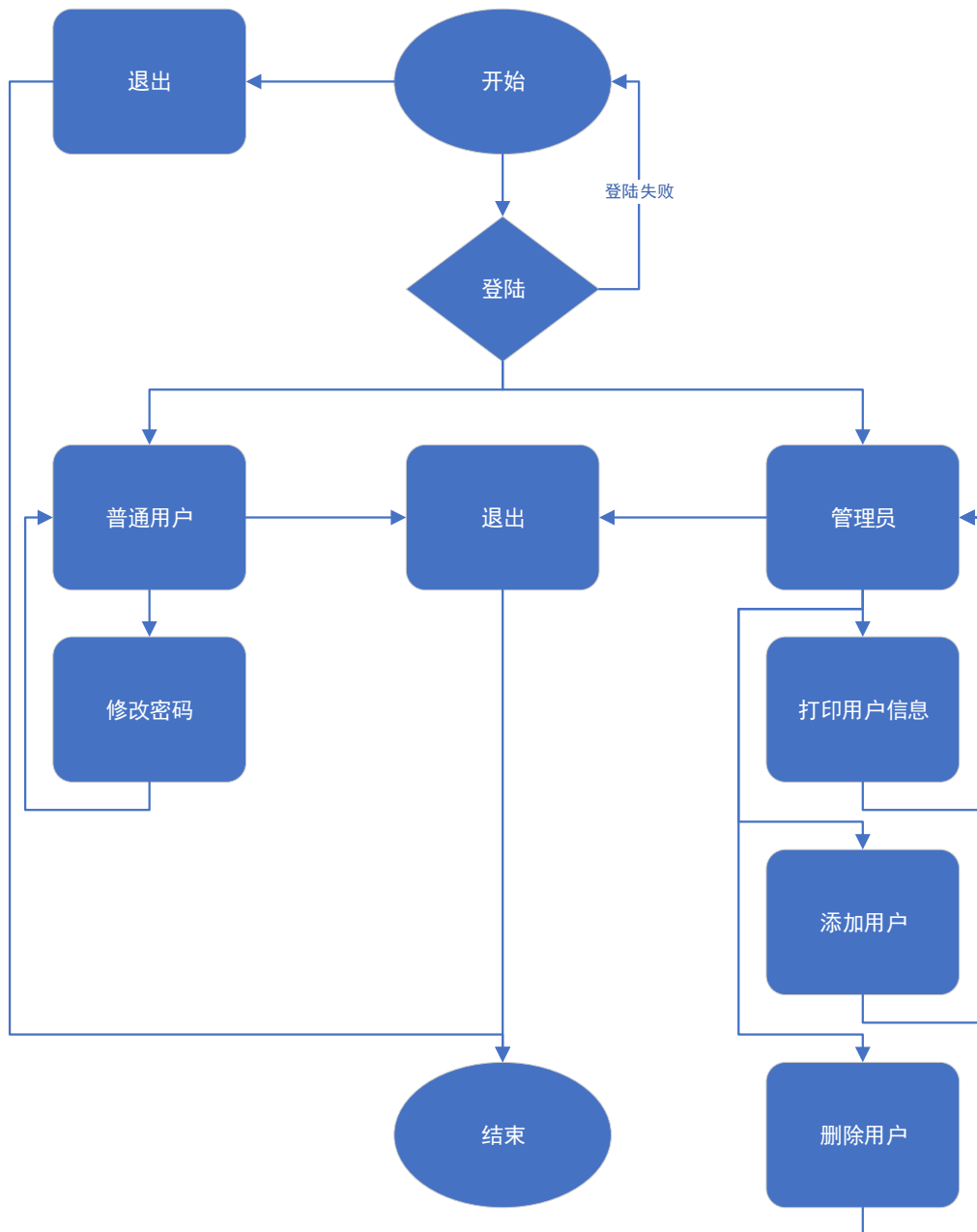
2.3.3 ShadowTreeNode 类

ShadowTreeNode 类定义了影子树节点，存储了每个节点打印时应当存储的位置，从而实现在正向打印时对其的功能。

2.3.4 ShadowTreeNodeQueue 类

ShadowTreeNodeQueue 类用数组实现了 queue 的功能，用于在打印时存放 ShadowTreeNoded 的指针。

2.4 主程序的设计



3 调试分析

3.1 技术难点分析

建立 Avl 树时，要尽量避免树的旋转，减少程序负担。所以，在输出时选择前序遍历或是后续遍历，可以保证重建二叉树时，子节点总是在母节点之后被生成，也就不会发生旋转了。

Avl 树进行打印的时候，如何确定每个结点所处的位置是一个很大的问题，所以，引入了影子树的思想，在打印前，先构建一棵带每个节点横纵坐标的影子树，用于打印。打印时，一层一层进行打印，下一层的内容被放入队列中等待下一次打印。

3.2 调试错误分析

1. 在某些情况下，database.dat 的内容不会及时被修改
SaveFile 函数调用的位置选择不当，导致在某些操作时，文件并没有被及时存储，导致了问题。
2. 由于没有进行内存回收，导致内存泄漏

```
11 void GarbageCollecting(Node *root) {  
12     if (root->left){  
13         GarbageCollecting(root->left);  
14     }  
15     if (root->right){  
16         GarbageCollecting(root->right);  
17     }  
18     delete root;  
19 }
```

添加了如下函数，使用递归遍历树，并在退出程序时调用这个函数，实现内存回收。

4 测试结果分析

4.1 登陆

```
欢迎使用用户登陆管理系统
1. 登陆
2. 退出
请选择你要进行的操作：
```

图 4.1.1

```
请输入账号:admin
请输入密码:123456
```

图 4.1.2

```
请输入账号:test
用户不存在
请按任意键继续. . .
```

图 4.1.3

```
请输入账号:admin
请输入密码:err_pass
密码错误
请按任意键继续. . .
```

图 4.1.4

在启动程序后，会先显示欢迎界面（图 4.1.1），选择登陆，就会进入账号密码输入界面。如果输入正确的账号和密码，即可完成登陆过程。如果用户不存在或者密码错误，就会显示对应的错误信息（如图 4.1.3，4.1.4）

4.2 管理员界面

```
你好, admin
1. 正向打印树
2. 添加用户
3. 删除用户
4. 退出
请选择你要进行的操作：
```

图 4.2.1

如果用户名为“admin”，就会进入管理员界面（图 4.2.1），可以在此界面选择要进行的操作。

4.2.1 正向打印树

```

      hhr
     /  \
  admin  kmp
 /   \  /   \
123   hhr123 papapa ppp
      /   \
    hhr11. test
  
```

请按任意键继续. . .

图 4.2.1.1

选择正向打印树功能后，就会调用影子树的代码，在屏幕上生成一棵正向的树（图 4.2.1.1）。为了让树可以对齐，我强制了用户名的最大长度如果超过该长度，就会将后面的内容变为点号，保证了名字字段等长。

4.2.2 添加用户

```

请输入姓名和密码：
hhr
123456
  
```

图 4.2.2.1

```

请输入姓名和密码：
123
123
用户已存在
请按任意键继续. . .
  
```

图 4.2.2.2

```

请输入姓名和密码：
1234
1234
创建成功
      hhr
     /  \
  1234  kmp
 /   \  /   \
123   hhr123 papapa ppp
      /   \
    hhr11. test
  
```

请按任意键继续. . .

图 4.2.2.3

选择添加用户功能后，会要求你输入用户名及其密码（图 4.2.1.1）。如果已经存在同名用户，会提示你用户已存在（图 4.2.2.2），否则就会将用户插入 Avl 树中，并打印这棵树（图 4.2.2.3）。

4.2.3 删除用户

```
请输入姓名
123_
```

图 4.2.3.1

```
请输入姓名
123
删除成功
      hhr
1234
  admin      hhr123      papapa
           hhr11.      kmp      ppp      test
请按任意键继续. . . _
```

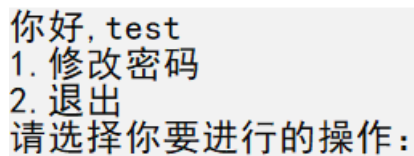
图 4.2.3.2

```
请输入姓名
no
用户不存在
请按任意键继续. . . _
```

图 4.2.3.3

选择删除用户功能后，系统会提示你输入用户名（图 4.2.3.1），如果用户存在，就会将其删除，并打印出操作后的树（图 4.2.3.2），否则提示该用户不存在（图 4.2.3.3）。

4.3 一般用户

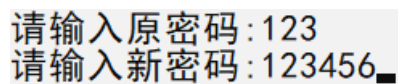


```
你好, test  
1. 修改密码  
2. 退出  
请选择你要进行的操作:
```

图 4.3.1

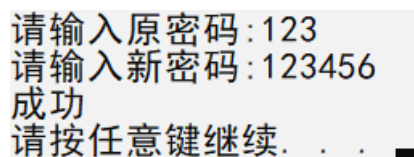
如果用户为一般用户，则只允许修改密码（图 4.3.1）。

4.3.1 修改密码



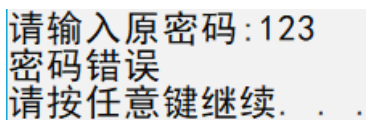
```
请输入原密码:123  
请输入新密码:123456_
```

图 4.3.1.1



```
请输入原密码:123  
请输入新密码:123456  
成功  
请按任意键继续. . .
```

图 4.3.1.2



```
请输入原密码:123  
密码错误  
请按任意键继续. . .
```

图 4.3.1.3

进入修改密码界面后，系统会提示输入原密码（图 4.3.1.1）。如果原密码错误，就会提示密码错误（图 4.3.1.3），否则要求输入新密码，并完成修改（图 4.3.1.2）。

5 技术亮点分析

5.1 在不同的时候合理使用二叉树不同的遍历方式

众所周知，二叉树有三种遍历方式，即前序遍历，中序遍历，以及后续遍历。在我的代码里，考虑了这三种遍历各自的特点，在特定的区块使用了特定的遍历，提高运行效率。

例 1:

```

14 void AvlTree::save(Node *tree) {
15     if (tree == nullptr) { //如果根节点是空指针，说明这棵树已经完成了遍历，即可退出
16         return;
17     }
18     fstream out;
19     if (tree == root) { //如果该节点是根节点，说明文件输出刚刚开始，则清除原有文件，否则在文件尾追加
20         out.open(_Filename: "database.dat", ios::out);
21     } else {
22         out.open(_Filename: "database.dat", _Mode: ios::out | ios::app);
23     }
24     out << tree->getName() << " " << tree->getPassword() << endl;
25     out.close();
26     save(tree->left); //递归遍历左子树
27     save(tree->right); //递归遍历右子树
28 }

```

在保存二叉树时，我使用了前序遍历，这样做的好处是，在重新读取文件时，根节点总是会其子节点之前被读入，生成，就减少了很多旋转树的操作，在不增加其它负担的情况下，极大程度提高了代码的效率。

例 2:

```

11 void GarbageCollecting(Node *root) {
12     if (root->left) {
13         GarbageCollecting(root->left);
14     }
15     if (root->right) {
16         GarbageCollecting(root->right);
17     }
18     cout << "已释放: " << root->getName() << endl;
19     delete root;
20 }

```

在程序退出时用于内存回收的代码，我使用了后续遍历。因为内存回收需要遍历树的每一个节点，如果根节点被删除了，自然就不能再找到其子节点，也就不能进行递归删除的操作，所以使用后续遍历，保证子节点被删除，然后再删除父节点，就实现了内存回收的目的。

5.2 使用影子树实现二叉树的正向打印

相比于一般基于层序遍历的二叉树打印方法，我选择了基于影子树的二叉树打印实现。影子树，即一棵带有节点坐标信息的二叉树。在生成影子树之前，先计算出每个元素应该显示的长度，以确定每一项前缀及后缀的用于对其的空格的数目。同时，为了尽量减少对 STL 模板容器的依赖，我用数组模拟出了一个队列，用于打印队列的存放。

```

216  int AvlTree::getMaxLength(Node *Tree) {
217      int maxLength = Tree->getName().length();
218      if (Tree->left) {
219          maxLength = max(maxLength, getMaxLength(Tree->left));
220      }
221      if (Tree->right) {
222          maxLength = max(maxLength, getMaxLength(Tree->right));
223      }
224      return maxLength;
225  }
226
227  int AvlTree::getMaxLength(AvlTree *pTree) {
228      return getMaxLength(pTree->root);
229  }

```

用于获取最大内容长度的代码

```

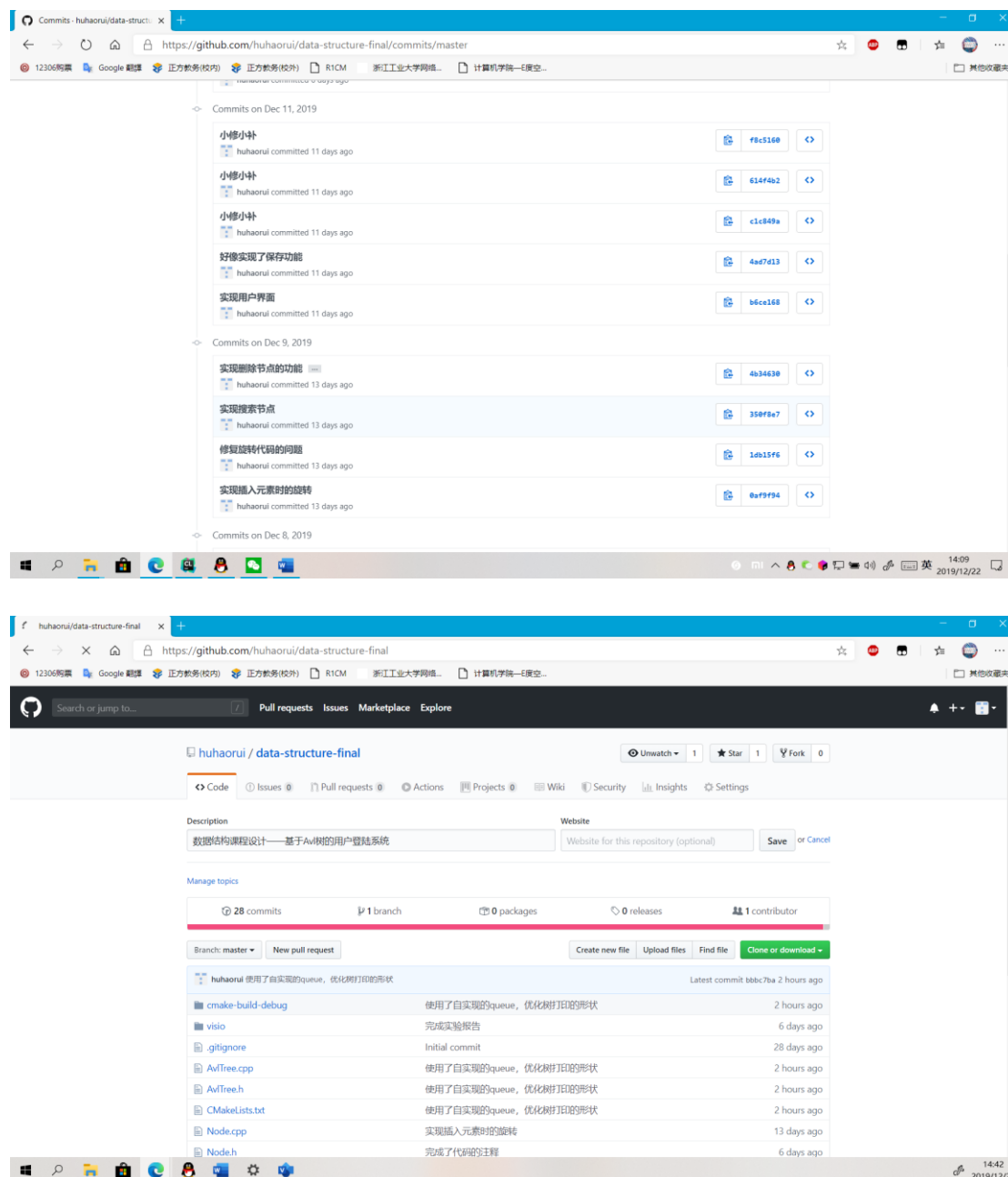
if (tmpName.length() < MaxLength) {
    tmpName = string(_Ptr: " ").append(tmpName);
}

```

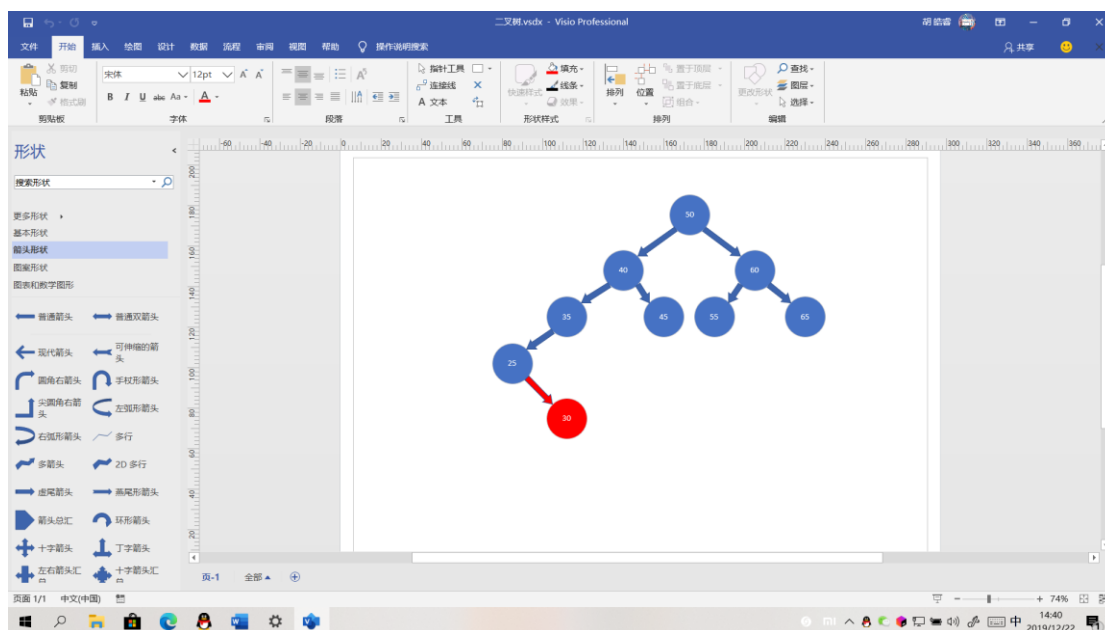
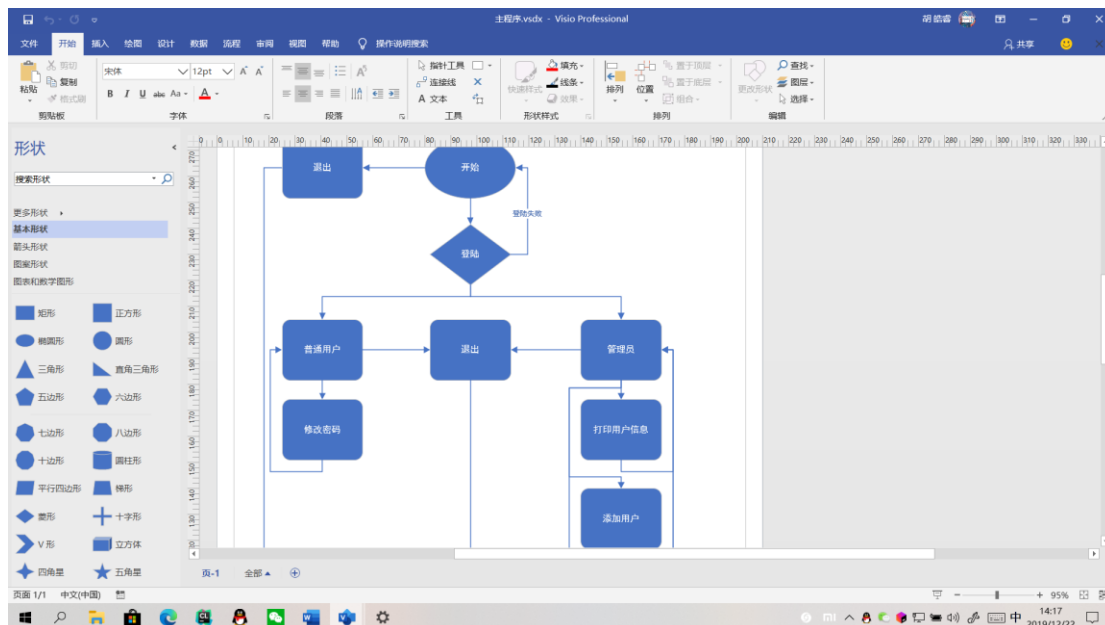
这三行代码，基于 Clang-Tidy 的要求，使用了有参构造函数生成的 string 对象，及其 append 方法，避免了形如 `tmpName = " " + tmpName;` 的代码，由于多进行了一次 string 对象的构造与析构导致的性能损失，提高了程序的性能。

5.3 使用 GitHub 进行版本控制

为了方便进行版本控制，以及证明代码原创性的需求，我使用了 GitHub 进行代码的托管
<https://github.com/huhaorui/data-structure-final>
 每一次对代码的修改都可以在该页面看见。



5.4 使用 Microsoft Visio 进行流程图的绘制



Microsoft Visio 是一款出色的流程图绘制软件，我使用它绘制了课程设计报告的绘图。Microsoft Visio 提供了大量预置的模板，在几乎所有行业上都可以使用它绘制需要的图表，而且为未来的修改保存了空间。

6 附录

6.1 main.cpp

```
//
// Created by HHR on 2019/11/27.
//

#include <iostream>
#include <fstream>
#include "AvlTree.h"

using namespace std;

void GarbageCollecting(Node *root) {
    if (root->left) {
        GarbageCollecting(root->left);
    }
    if (root->right) {
        GarbageCollecting(root->right);
    }
    cout << "已释放: " << root->getName() << endl;
    delete root;
}

void readFile(AvlTree *root) { //从文件中读取 Avl 树的信息
    string name, password;
    ifstream infile;
    infile.open("database.dat", ios::in);
    while (infile >> name >> password) {
        root->addUser(name, password);
    }
    infile.close();
}

void SaveFile(AvlTree *root) { //将 Avl 树保存到文件
    root->save(root->root);
}

void PasswordReset(Node *user) { //修改密码功能
    string password;
    cout << "请输入原密码:";
```

```

    cin >> password;
    if (password != user->getPassword()) {
        cout << "密码错误" << endl;
        system("pause");
        return;
    }
    cout << "请输入新密码:";
    cin >> password;
    user->setPassword(password);
    cout << "成功" << endl;
    system("pause");
}

void AddUser(AvlTree *tree) { //添加用户功能
    string name, password;
    cout << "请输入姓名和密码:" << endl;
    cin >> name >> password;
    if (AvlTree::SearchNode(name, tree->root) != nullptr) {
        cout << "用户已存在" << endl;
        return;
    }
    tree->addUser(name, password);
    cout << "创建成功" << endl;
    tree->print();
}

void DeleteUser(AvlTree *tree) { //删除用户功能
    string name;
    cout << "请输入姓名" << endl;
    cin >> name;
    if (AvlTree::SearchNode(name, tree->root) == nullptr) {
        cout << "用户不存在" << endl;
        return;
    }
    if (name == "admin") {
        cout << "该用户不能被删除" << endl;
        return;
    }
    tree->removeUser(name);
    cout << "删除成功" << endl;
    tree->print();
}

```

```

void adminView(AvlTree *tree) { //管理员用户的界面
    while (true) {
        system("cls");
        cout << "你好,admin" << endl;
        cout << "1.正向打印树" << endl;
        cout << "2.添加用户" << endl;
        cout << "3.删除用户" << endl;
        cout << "4.修改密码" << endl;
        cout << "5.退出" << endl;
        cout << "请选择你要进行的操作: ";
        string op;
        cin >> op;
        switch (op[0]) {
            case '1':
                system("cls");
                tree->print();
                system("pause");
                break;
            case '2':
                system("cls");
                AddUser(tree);
                system("pause");
                break;
            case '3':
                system("cls");
                DeleteUser(tree);
                system("pause");
                break;
            case '4':
                system("cls");
                PasswordReset(AvlTree::SearchNode("admin",
tree->root));
                break;
            case '5':
                break;
            default:
                cout << "错误" << endl;
        }
        if (op[0] == '5') break;
        SaveFile(tree);
    }
}

```

```

void userView(AvlTree *tree, Node *user) {//普通用户的界面
    while (true) {
        system("cls");
        cout << "你好," << user->getName() << endl;
        cout << "1.修改密码" << endl;
        cout << "2.退出" << endl;
        cout << "请选择你要进行的操作: ";
        string op;
        cin >> op;
        if (op[0] == '1') {
            system("cls");
            PasswordReset(user);
            SaveFile(tree);
        } else if (op[0] == '2') {
            cout << "再见" << endl;
            break;
        }
    }
}

void loginSuccessful(AvlTree *tree, Node *user) {//登陆成功
后, 根据用户类别, 跳转到对应的主界面
    if (user->getName() == "admin") {
        adminView(tree);
    } else {
        userView(tree, user);
    }
}

void login(AvlTree *tree) {//登陆界面
    string name, password;
    cout << "请输入账号:";
    cin >> name;
    Node *user = AvlTree::SearchNode(name, tree->root);
    if (user == nullptr) {
        cout << "用户不存在" << endl;
        system("pause");
        return;
    }
    cout << "请输入密码:";
    cin >> password;
    if (user->getPassword() == password) {
        cout << "登陆成功" << endl;
    }
}

```

```

        loginSuccessful(tree, user);
    } else {
        cout << "密码错误" << endl;
        system("pause");
    }
}

int main() {
    auto *tree = new AvlTree;
    readFile(tree);
    while (true) {
        string op;
        system("cls");
        cout << "欢迎使用用户登陆管理系统" << endl;
        cout << "1.登陆" << endl;
        cout << "2.退出" << endl;
        cout << "请选择你要进行的操作: ";
        cin >> op;
        system("cls");
        switch (op[0]) {
            case '1':
                login(tree);
                break;
            case '2':
                cout << "正在进行内存回收, 请稍后\n";
                SaveFile(tree);
                GarbageCollecting(tree->root);
                exit(0);
            default:
                cout << "错误" << endl;
                system("pause");
        }
    }
}

```

6.2 AvlTree.h

```
//
// Created by HHR on 2019/11/27.
//
#ifndef DATA_STRUCTURE_FINAL_AVLTREE_H
#define DATA_STRUCTURE_FINAL_AVLTREE_H

#include "Node.h"
#include "ShadowTreeNode.h"

class AvlTree {
public:
    Node *root;

    AvlTree() { //默认构造函数，初始化 root 指针，防止出现悬挂指针的现象
        root = nullptr;
    }

    static int getHeight(Node *tree) { //静态成员函数，用于获得树的高度
        if (tree != nullptr) return tree->getHeight();
        return -1; //如果节点不存在，那么它的高度是-1
    }

    void addUser(string &, string &); //添加用户功能
    void print(); //使用影子树正向打印树
    static Node *addNode(Node *, string &, string &); //静态成员函数，用于添加节点
    static ShadowTreeNode *ShadowTreeBuild(Node *Tree, ShadowTreeNode *ShadowTree, int TreeRow); //静态成员函数，用于生成影子树
    static Node *LeftSpin(Node *); //静态成员函数，将二叉树左旋
    static Node *RightSpin(Node *); //静态成员函数，将二叉树右旋
    static Node *SearchNode(const string &, Node *); //静态成员函数，搜索用户名对应的节点，返回指向它的指针
    static Node *remove(Node *, const string &); //静态成员函数，删除一个节点
    static Node *maxNode(Node *); //静态成员函数，返回左子树的最大节点，在删除节点时被调用
    static Node *minNode(Node *); //静态成员函数，返回右子树的最小节点，在删除节点时被调用
};
```

```

void removeUser(string &); //删除用户名对应的用户
void save(Node *); //保存树到文件
static int getMaxLength(AvlTree *pTree);

static int getMaxLength(Node *Tree);
};

#endif //DATA_STRUCTURE_FINAL_AVL_TREE_H

```

6.3 AvlTree.cpp

```
//
// Created by HHR on 2019/11/27.
//
#include "AvlTree.h"
#include "ShadowTreeNode.h"
#include "ShadowTreeNodeQueue.h"
#include <cmath>
#include <fstream>

int max(int a, int b) {
    return a > b ? a : b;
}

void AvlTree::save(Node *tree) {
    if (tree == nullptr) {//如果根节点是空指针，说明这棵子树已经完成了遍历，即可退出
        return;
    }
    fstream out;
    if (tree == root) {//如果该节点是根节点，说明文件输出刚刚开始，则清除原有文件，否则在文件尾追加
        out.open("database.dat", ios::out);
    } else {
        out.open("database.dat", ios::out | ios::app);
    }
    out << tree->getName() << " " << tree->getPassword() << endl;
    out.close();
    save(tree->left);//递归遍历左子树
    save(tree->right);//递归遍历右子树
}

void AvlTree::addUser(string &name, string &password) {
    if (SearchNode(name, root) != nullptr) return;//如果用户已经存在，则不添加
    root = addNode(root, name, password);//调用 addNode 函数添加节点
}

void AvlTree::removeUser(string &name) {
    if (SearchNode(name, root) == nullptr) return;//如果用户
```


不存在，则不删除

```
    root = remove(root, name); //调用 remove 函数删除节点
}
```

```
Node *AvlTree::addNode(Node *pNode, string &name, string
&password) {
    if (pNode == nullptr) { //找到空的插入点，创建一个新的
Node，并返回插入点
        pNode = new Node(name, password);
        return pNode;
    } else if (name < pNode->getName()) { //要插入左边
        pNode->left = addNode(pNode->left, name, password);
        if (getHeight(pNode->left) - getHeight(pNode->right)
== 2) { //平衡因子为 2，发生了不平衡
            if (name < pNode->left->getName()) { //添加的节点在
左子树的左子树内，采用右旋操作
                pNode = RightSpin(pNode);
            } else { //添加的节点在左子树的右子树内，采用先左旋再右
旋操作
                pNode->left = LeftSpin(pNode->left);
                pNode = RightSpin(pNode);
            }
        }
    } else {
        pNode->right = addNode(pNode->right, name,
password);
        if (getHeight(pNode->right) - getHeight(pNode->left)
== 2) { //平衡因子为 2，发生了不平衡
            if (name > pNode->right->getName()) { //添加的节点
在右子树的右子树内，采用左旋操作
                pNode = LeftSpin(pNode);
            } else { //添加的节点在右子树的左子树内，采用先右旋再左
旋操作
                pNode->right = RightSpin(pNode->right);
                pNode = LeftSpin(pNode);
            }
        }
    }
    pNode->setHeight(max(getHeight(pNode->left),
getHeight(pNode->right)) + 1); //获取完成插入后新的树的高度
    return pNode;
}
```

```
Node *AvlTree::SearchNode(const string &name, Node *root) {
    if (root == nullptr) return nullptr; //如果节点为空, 说明找不到需要的节点, 返回空指针
    if (root->getName() == name) { //如果节点内容与查找值相同, 返回找到的节点指针
        return root;
    } else if (root->getName() > name) { //如果节点的内容比查找值大, 说明目标节点在左子树, 递归进入查找
        return SearchNode(name, root->left);
    } else { //否则去右子树查找
        return SearchNode(name, root->right);
    }
}
```

```
Node *AvlTree::remove(Node *root, const string &name) {
    if (root == nullptr) return nullptr; //如果节点是空的, 说明要删除的东西不存在, 返回空指针
    if (name < root->getName()) { //如果节点的内容比查找值大, 说明目标节点在左子树, 递归进入
        root->left = remove(root->left, name);
        if (getHeight(root->right) - getHeight(root->left) == 2) { //如果平衡因子为 2, 说明发生了失衡, 需要旋转
            Node *right = root->right; //右子树比较深, 说明右子树需要进行旋转
            if (getHeight(right->left) > getHeight(right->right)) { //如果右子树的左子树比较深, 执行先右旋再左旋
                root->right = RightSpin(root->right);
                root = LeftSpin(root);
            } else { //否则直接左旋
                root = LeftSpin(root);
            }
        }
    } else if (name > root->getName()) { //如果用户名比当前节点要大, 说明要删除的节点在右子树, 递归进入
        root->right = remove(root->right, name);
        if (getHeight(root->left) - getHeight(root->right) == 2) { //如果平衡因子为 2, 说明发生了失衡, 需要旋转
            Node *left = root->left; //左子树比较深, 说明左子树需要进行旋转
            if (getHeight(left->left) < getHeight(left->right)) { //如果左子树的右子树比较深, 执行先左旋再右旋

```

```

        root->left = LeftSpin(root->left);
        root = RightSpin(root);
    } else { //否则直接右旋
        root = RightSpin(root);
    }
}
} else { //既不偏大，也不偏小，显然是找到了删除点
    if (root->left != nullptr && root->right != nullptr)
    { //两侧都存在子树
        if (getHeight(root->left) >
            getHeight(root->right)) { //如果左子树比较深
            Node *toRemove = maxNode(root->left); //删除点
            的内容会被左子树的最大值替换，所以找到左子树的最大值
            root->setName(toRemove->getName()); //将根节点
            (也就是要删除的节点)的信息替换为左子树的最大节点
            root->setPassword(toRemove->getPassword());
            root->left = remove(root->left,
            toRemove->getName()); //删除左子树最大的节点
        } else {
            Node *toRemove = minNode(root->right); //删除点
            的内容会被右子树的最小值替换，所以找到右子树的最小值
            root->setName(toRemove->getName());
            root->setPassword(toRemove->getPassword());
            root->right = remove(root->right,
            toRemove->getName());
        }
    } else { //如果至少有一侧是空的，那就直接把一棵子树提高
        Node *tmp = root;
        if (root->left != nullptr) { //如果左子树存在，就将右
            节点提高
            root = root->left;
        } else { //否则将左节点提高
            root = root->right;
        }
        delete tmp; //删除根节点
    }
}
if (root != nullptr) {
    root->setHeight(max(getHeight(root->left),
    getHeight(root->right)) + 1);
}
return root;
}

```

```

Node *AvlTree::maxNode(Node *root) {
    if (root->right != nullptr) { //如果右子树存在, 说明最大的节点在右子树内
        return maxNode(root->right); //递归查找右子树
    }
    return root; //如果右子树不存在, 说明该节点即为最大的节点
}

Node *AvlTree::minNode(Node *root) {
    if (root->left != nullptr) { //如果左子树存在, 说明最小的节点在左子树内
        return minNode(root->left); //递归查找左子树
    }
    return root; //如果左子树不存在, 说明该节点即为最大的节点
}

void AvlTree::print() {
    if (root == nullptr) return; //如果根节点不存在, 显然是不需要打印的, 返回
    ShadowTreeNode *ShadowTree = nullptr;
    ShadowTree = ShadowTreeBuild(root, ShadowTree, 1); //开始时, 应该打印在第一行
    ShadowTreeNodeQueue queue; //建立一个打印队列
    queue.push(ShadowTree);
    int MaxRow = 1, MaxColumn = 0;
    int MaxLength = getMaxLength(this); //设置内容的最大长度, 以对齐树
    while (!queue.empty()) { //如果打印队列非空
        ShadowTreeNode *pShadowTreeNode = queue.front(); //取出队列第一项
        queue.pop();
        if (pShadowTreeNode->row > MaxRow) { //如果行数超过了最大行数, 说明要开始打印下一行
            cout << endl; //换行
            MaxRow = pShadowTreeNode->row; //重设最大行数
            MaxColumn = 0;
        }
        string tmpName = pShadowTreeNode->name;
        while (tmpName.length() < MaxLength) { //检查名字有没有过长, 如果太长, 就将其截断, 否则在后面补上空格
            tmpName += ' ';
            if (tmpName.length() < MaxLength) {

```

```

        tmpName = string(" ").append(tmpName);
    }
}
for (int i = 1; i < pShadowTreeNode->column -
MaxColumn; i++) { //输出前置的空格
    for (int j = 0; j < MaxLength; j++) cout << ' ';
}
cout << tmpName;
MaxColumn = pShadowTreeNode->column;
if (pShadowTreeNode->left != nullptr)
queue.push(pShadowTreeNode->left); //如果左子树非空, 将其加入打印队列
if (pShadowTreeNode->right != nullptr)
queue.push(pShadowTreeNode->right);
}
cout << endl;
}

ShadowTreeNode *AvlTree::ShadowTreeBuild(Node *Tree,
ShadowTreeNode *ShadowTree, int TreeRow) {
    if (Tree != nullptr) {
        static int TreeColumn; //这个变量需要 static, 保证不变
        if (TreeRow == 1) {
            TreeColumn = 1; //第一行只有一列
        }
        ShadowTree = new ShadowTreeNode();
        ShadowTree->left = ShadowTreeBuild(Tree->left,
ShadowTree->left, TreeRow + 1); //递归建立左子树
        ShadowTree->column = TreeColumn++; //设置影子树节点的内
容
        ShadowTree->row = TreeRow;
        ShadowTree->name = Tree->getName();
        ShadowTree->right = ShadowTreeBuild(Tree->right,
ShadowTree->right, TreeRow + 1); //递归建立右子树
    }
    return ShadowTree;
}

Node *AvlTree::RightSpin(Node *root) {
    Node *left = root->left;
    root->left = left->right;
    left->right = root;
    root->setHeight(max(getHeight(root->left),

```

```

getHeight(root->right)) + 1); //旋转后, 重设树的高度
    left->setHeight(max(getHeight(left->left),
getHeight(left->right)) + 1);
    return left;
}

```

```

Node *AvlTree::LeftSpin(Node *root) {
    Node *right = root->right;
    root->right = right->left;
    right->left = root;
    root->setHeight(max(getHeight(root->left),
getHeight(root->right)) + 1); //旋转后, 重设树的高度
    right->setHeight(max(getHeight(right->left),
getHeight(right->right)) + 1);
    return right;
}

```

```

int AvlTree::getMaxLength(Node *Tree) {
    int maxLength = Tree->getName().length();
    if (Tree->left) {
        maxLength = max(maxLength,
getMaxLength(Tree->left));
    }
    if (Tree->right) {
        maxLength = max(maxLength,
getMaxLength(Tree->right));
    }
    return maxLength;
}

```

```

int AvlTree::getMaxLength(AvlTree *pTree) {
    return getMaxLength(pTree->root);
}

```

6.4 Node.h

```
//
// Created by HHR on 2019/11/27.
//
#ifndef DATA_STRUCTURE_FINAL_NODE_H
#define DATA_STRUCTURE_FINAL_NODE_H

#include <iostream>

using namespace std;

class Node {
public:
    Node(string &, string &);

    Node *left = nullptr;
    Node *right = nullptr;

    string getName() {
        return name;
    }

    string getPassword() {
        return password;
    }

    int getHeight() {
        return height;
    }

    void setHeight(int);

    void setName(basic_string<char> _name) {
        name = std::move(_name);
        //std::move 为 C++11 标准新增加的功能，可以在传输 string 类
        //型遍历时，避免不必要的析构操作，提高代码效率
        //https://www.cnblogs.com/SZxiaochun/p/8017349.html
    }

    void setPassword(basic_string<char> _password) {
        password = std::move(_password);
        //std::move 为 C++11 标准新增加的功能，可以在传输 string 类
```

型遍历时，避免不必要的析构操作，提高代码效率

[//https://www.cnblogs.com/SZxiaochun/p/8017349.html](https://www.cnblogs.com/SZxiaochun/p/8017349.html)

```
}
```

```
private:
```

```
    string name;
```

```
    string password;
```

```
    int height; //节点高度
```

```
};
```

```
#endif //DATA_STRUCTURE_FINAL_NODE_H
```

6.5 Node.cpp

```
//
```

```
// Created by HHR on 2019/11/27.
```

```
//
```

```
#include "Node.h"
```

```
Node::Node(string &name, string &password) {
```

```
    this->name = name;
```

```
    this->password = password;
```

```
    height = 0;
```

```
}
```

```
void Node::setHeight(int i) {
```

```
    this->height = i;
```

```
}
```


6.6 ShadowTreeNode.h

```
//
// Created by HHR on 2019/11/30.
//

#ifndef DATA_STRUCTURE_FINAL_SHADOWTREENODE_H
#define DATA_STRUCTURE_FINAL_SHADOWTREENODE_H

#include <iostream>

using namespace std;

class ShadowTreeNode {
public:
    int row = 0;
    int column = 0;
    string name = "";
    ShadowTreeNode *left = nullptr;
    ShadowTreeNode *right = nullptr;
};

#endif //DATA_STRUCTURE_FINAL_SHADOWTREENODE_H
```

6.7 ShadowTreeNodeQueue.h

```
//
// Created by HHR on 2019/12/19.
//
#include "ShadowTreeNode.h"

#ifndef DATA_STRUCTURE_FINAL_SHADOWTREENODEQUEUE_H
#define DATA_STRUCTURE_FINAL_SHADOWTREENODEQUEUE_H

class ShadowTreeNodeQueue {
public:
    ShadowTreeNode *Node[1000] = {nullptr};
    int begin = 0, end = -1; //begin 是队列头, end 是队列尾, 当
    begin > end 时, 说明队列为空

    bool empty() {
        return begin > end;
    }

    void push(ShadowTreeNode *node);

    void pop() {
        begin++;
    }

    ShadowTreeNode *front();
};

#endif //DATA_STRUCTURE_FINAL_SHADOWTREENODEQUEUE_H
```

6.8 ShadowTreeNodeQueue.cpp

```
//  
// Created by HHR on 2019/12/19.  
//  
  
#include "ShadowTreeNodeQueue.h"  
  
void ShadowTreeNodeQueue::push(ShadowTreeNode *node) {  
    Node[++end] = node;  
}  
  
ShadowTreeNode *ShadowTreeNodeQueue::front() {  
    return Node[begin];  
}
```