浙江工业大学

C++程序设计课程设计
实验报告

2019/2020(2)

实验题目 用户登录系统模拟

学生姓名 张坤

学生学号 201906061726

学生班级 软件工程 1903 班

任课教师 毛国红

提交日期 2021.1.3

计算机科学与技术学院

用户登录系统模拟 实验报告

一、 大型实验的内容

用户登录系统的模拟

【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；

【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。

【考核要求】要求程序能正常运行，全面完成题目要求。

二、 运行环境

用户登录系统在 Qt 6.0 平台下开发，操作系统：Windows10。

硬件环境：

处理器：Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.60 GHz

内存：8.00GB

系统类型：64 位操作系统

三、 实验课题分析（主要的模块功能、流程图）

3.1 用户登录系统的主要功能

用户登录系统主要功能为：账号登录,账号注册,账号密码更改,账号注销。详细的系统功能结构为图 1 所示。

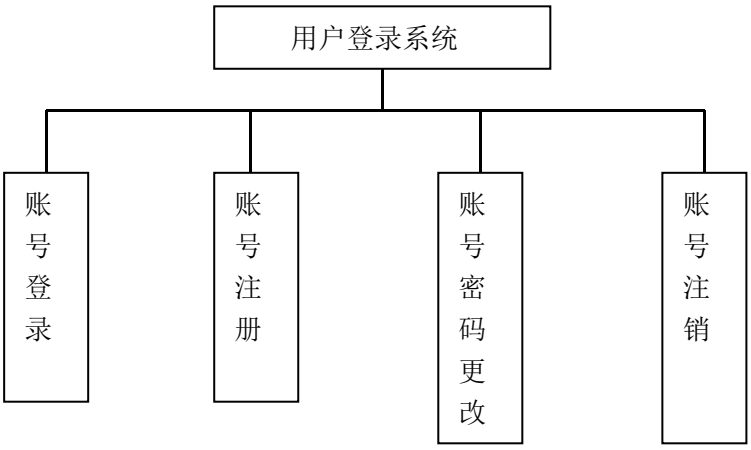


图 1 系统结构图

系统各模块的功能具体描述为：

1、 **账号登录模块:**点击登录按钮,读取输入的账号密码文本框,1)文本框未输入则弹窗提示输入文本 2)有输入则在 AVL 树内查找是否有相同项.有则提示登录成功,否则登录失败.

2、 **账号注册模块:**点击注册按钮,读取输入的账号密码文本框,1)文本框未输入则弹窗提示输入文本 2)有输入则在 AVL 树内添加节点并且进行平衡操作,提示注册成功

3、 **账号密码更改模块:**点击修改按钮，读取输入的账号密码文本框,1)文本框未输入则弹窗提示输入文本 2)修改密码账号密码需要登录才可以,未登录则会弹窗提示,否则会提示修改成功

4、 **账号注销模块:** 点击注销按钮，读取输入的账号密码文本框,1)文本框未输入则弹窗提示输入文本 2)注销账号同样需要先登录账号才可以，未登录则会弹窗提示,否则会提示注销成功

3.2 系统分析及设计

系统涉及对象有三个基本类：AVL 节点,AVL 模板树,UI 界面

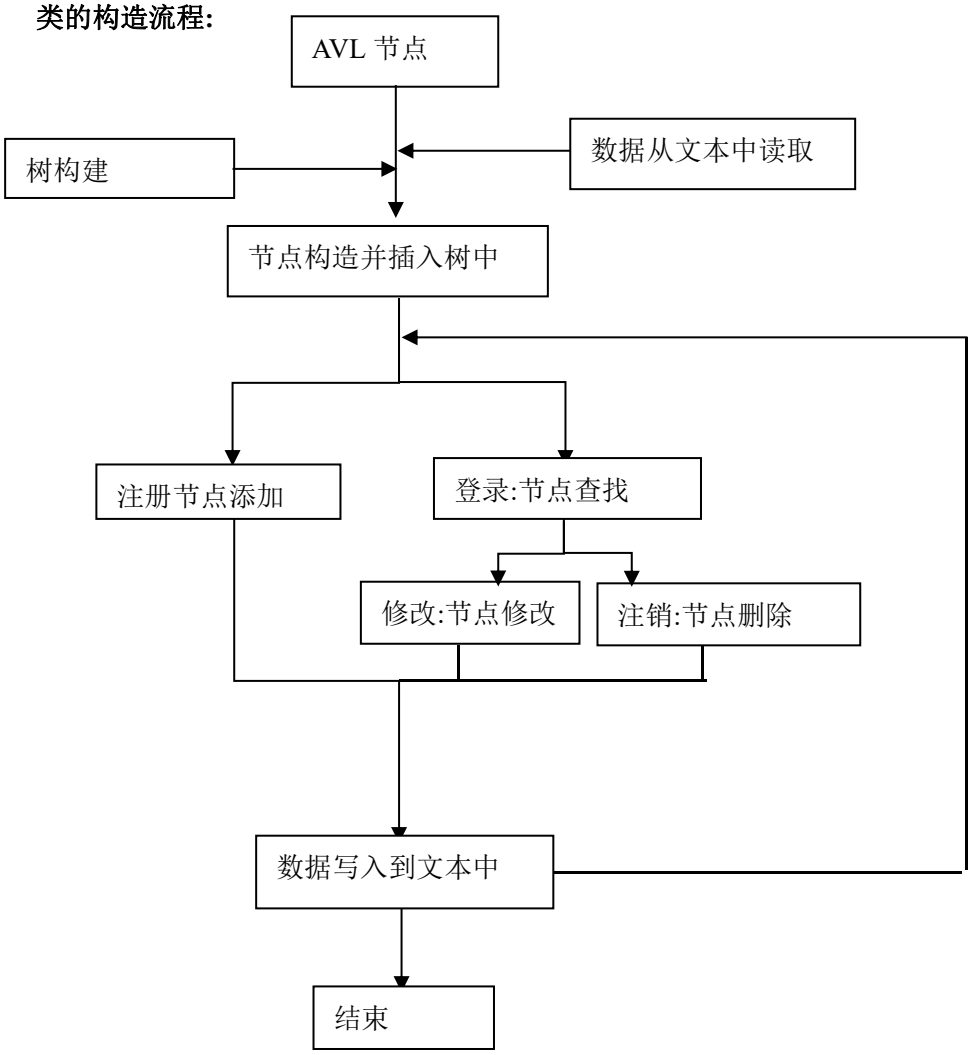
表 1 人员类涉及的操作

对象	涉及的对象操作	
AVL 节点	转换为 String 串	
	重新设置节点的成员变量值	
AVL 模板树	四种平衡操作	
	插入删除	
	重新设置节点值	
	文本读取和写入,即账号密码的保存	
UI 界面	登录按钮	账号密码查询
	注册按钮	账号密码添加
	注销按钮	账号密码删除
	修改按钮	账号密码修改

可以采用面向对象的方式实现图书管理系统，根据不同的功能和用处，其对象分为 AVL 模板类,账号密码类,UI 界面类。

用文本文件进行数据的保存，需要保存的数据主要包括账号,密码.实现所有的文本操作相关的功能。

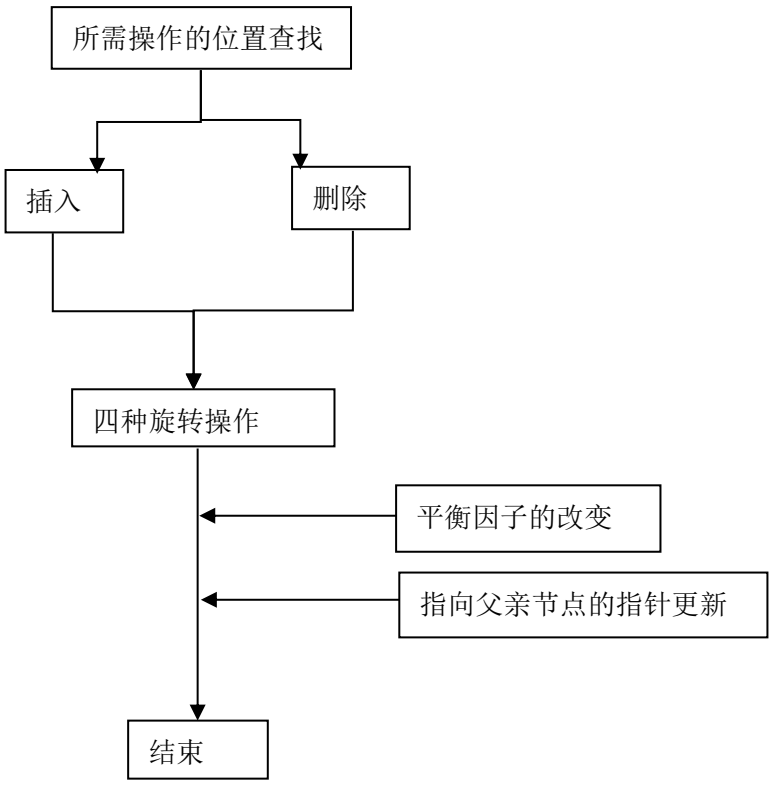
类的构造流程:



类的构造其实不难,只要分析其主要需求就可以

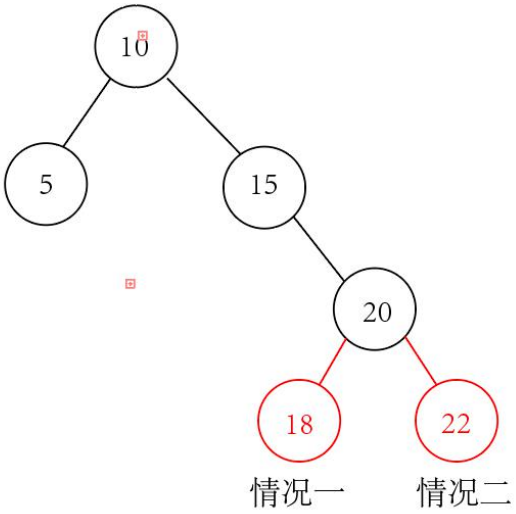
系统核心代码和难点分析:

AVl 树的插入,删除及其所涉及的四种旋转操作和平衡因子改变

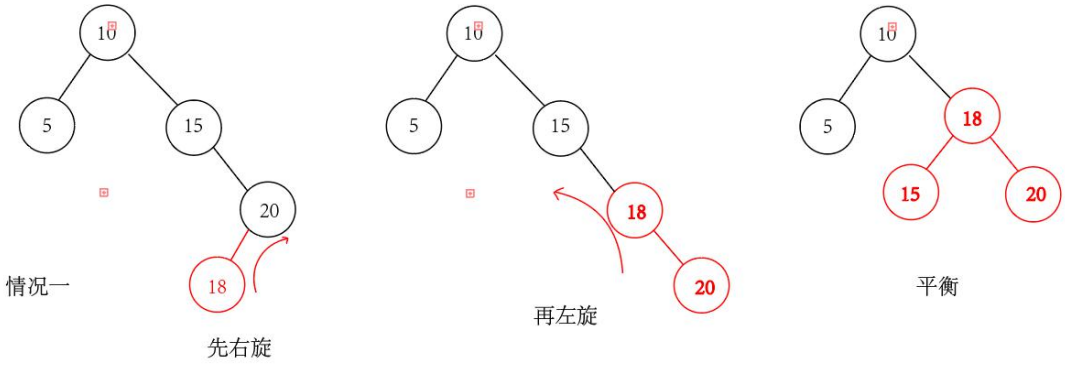


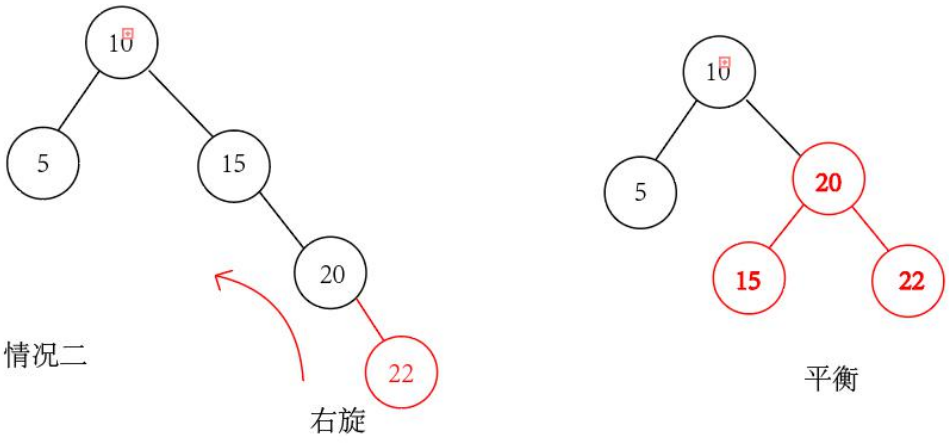
核心代码实现算法分析:

1. **树插入节点怎么实现**:首先必须指出 AVL 树其实是一棵 BST 树,那么它的插入算法也应该是和 BST 树相同,这一步的实现并不难.其次,插入节点后,如何根据不同的情况进行操作.因为根据不同情况分别有不操作,左旋,右旋,左右旋和右左旋,这里我是根据平衡因子的改变来进行左右旋的操作.那么问题就转化成了平衡因子的计算和左右旋的实现.插入操作导致树不平衡分情况有两种:



对应着两种情况需要进行左右旋的操作





当插入节点在左子树的情况不再赘述

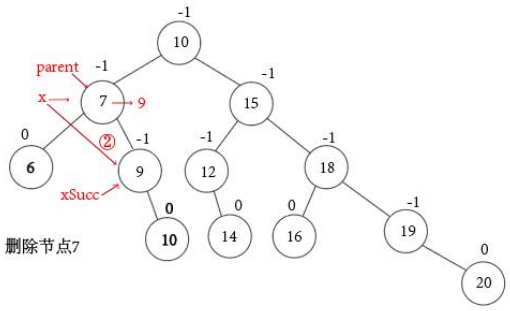
具体的左右旋操作参考下文.

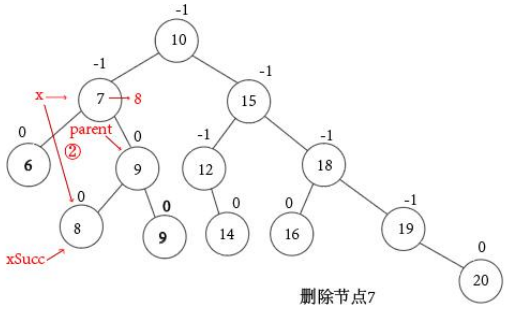
后面需要进行平衡因子的更新,具体参考下文**平衡因子的更新**

2. **树的删除操作怎么实现**:不难看出,删除操作虽然是插入操作的逆序,但是根据删除节点的情况不同也有不一样的操作.

首先对应 BST 树的删除

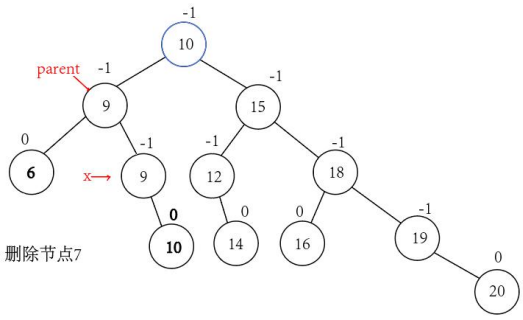
1) 删除节点有两个子节点

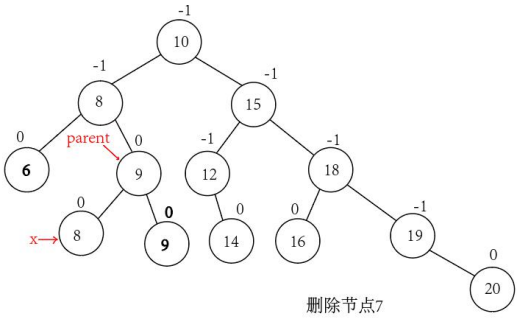




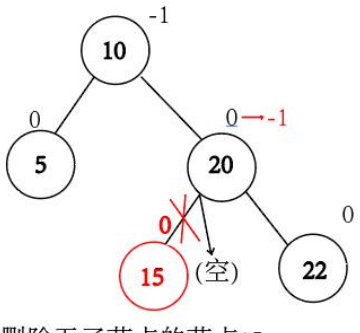
先通过循环让x定位到被删除节点
再另设xSucc指向x的右子树,parent指向x
如果xSucc有左子树,再遍历令xSucc指向左子树的最底层,parent指向xSucc的父节点
再将xSucc的data赋值给x的data
最后x指向xSucc指向的节点

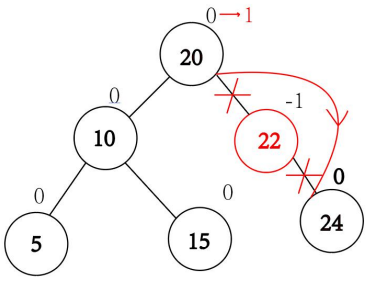
现在情况就转化为 2)了,也就是删除有不超过一个子节点的节点 x





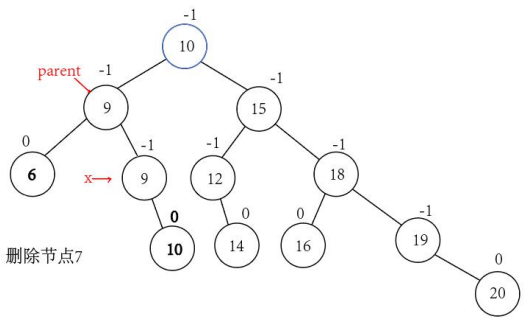
2)删除节点有不超过一个的子节点

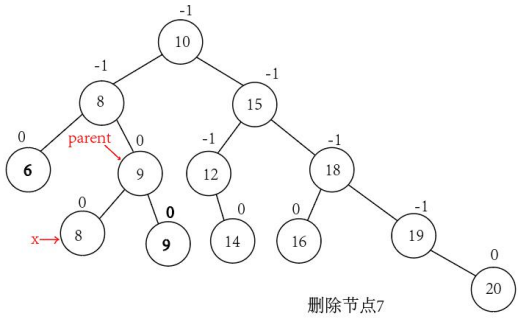




这里有一个陷阱,很容易以为到这里就结束了,但其实不是,这种情况的删除可能会导致更上面的树的失衡,所以需要向上遍历进行平衡因子更新和左右旋操作,具体情况参考下

文**平衡因子的更新**

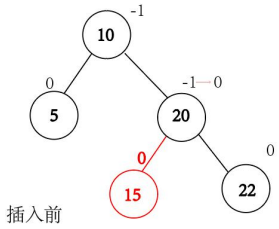




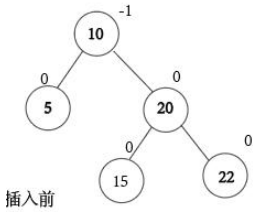
3. **平衡因子的更新:**核心思想就是利用每个节点的一个成员变量:指向父亲的指针.利用循环,顺着树往上遍历,对每一步实行平衡因子的更新,直到遇到根节点或者遇到 break 的情况.与此同时,每个循环过程中也需要根据平衡因子的不同而进行左右旋操作.

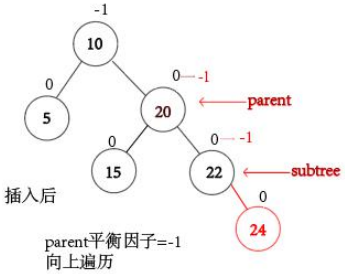
插入函数有三种情况,需要判断 Parent 平衡因子

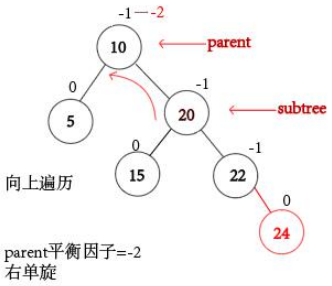
1) 平衡因子=0,表明插入后树平衡,break 即可.



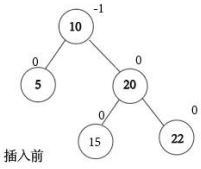
2)平衡因子=+-1,这符合 AVL 树定义,只需要向上遍历即可.

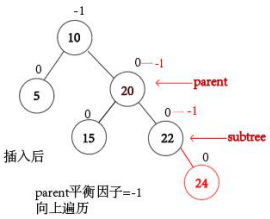


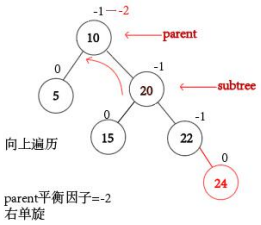


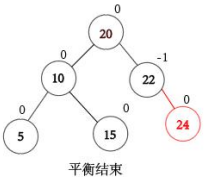


3)平衡因子=+-2,此即需要进行左右旋操作



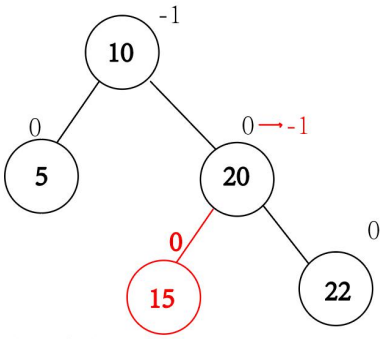






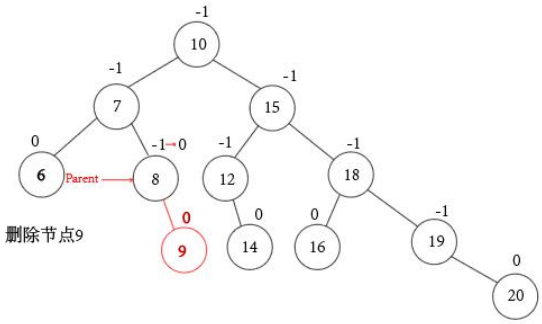
删除函数也有三种情况,但是稍与插入函数不同,需要判断 Parent 的平衡因子

1) 平衡因子=+-1,表明原树平衡因子为 0,不需要操作,break 即可.

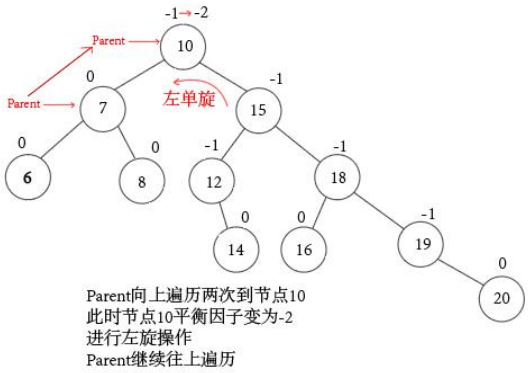


删除节点15

2) 平衡因子=0,需要往前遍历检查父节点.

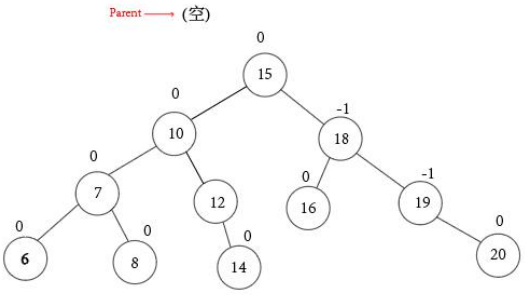


删除节点9



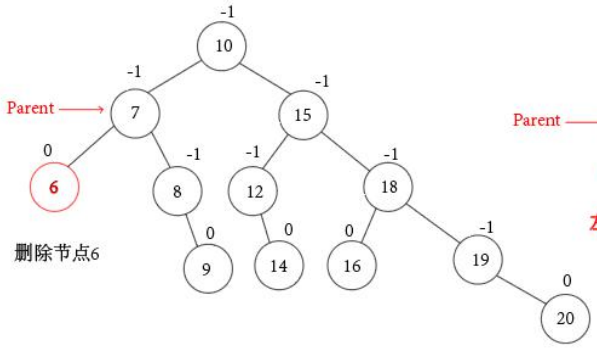
Parent向上遍历两次到节点10
此时节点10平衡因子变为-2
进行左旋操作
Parent继续往上遍历

next

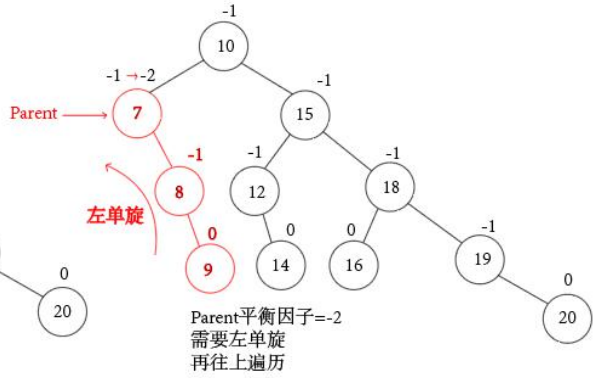


因为Parent空,所以不需要继续遍历
循环结束

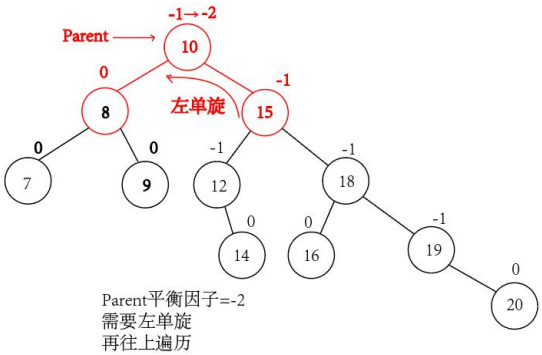
3) 平衡因子=+-2,需要进行左右旋操作,并往上遍历检查父节点

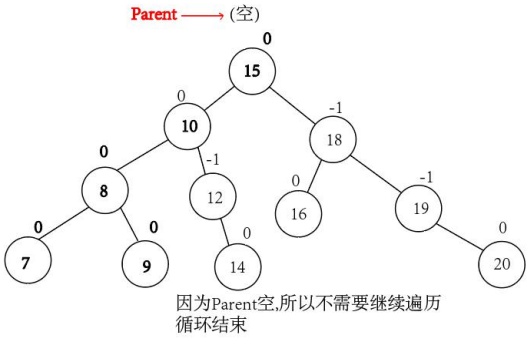


删除节点6



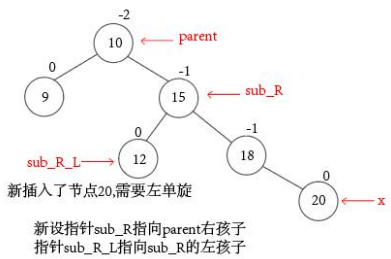
Parent平衡因子=-2
需要左单旋
再往上遍历

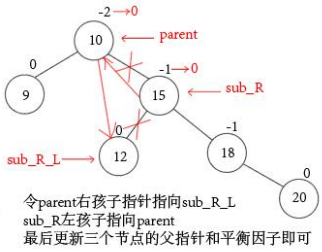


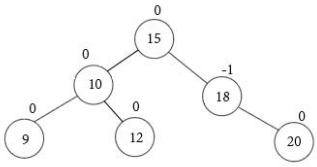


4. **左右旋的实现**:首选需要考虑指针的指向变化,这一点其实不难操作.但是由于每个节点有指向父亲的指针,这里还需要进行该指针指向的更新.最后就是平衡结束,平衡因子的更新.

1) 左单旋: 当插入项位于最近的平衡因子为-2 的祖先节点的右孩子的右子树时,左单旋.



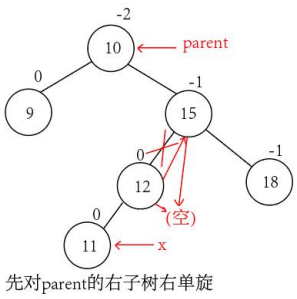


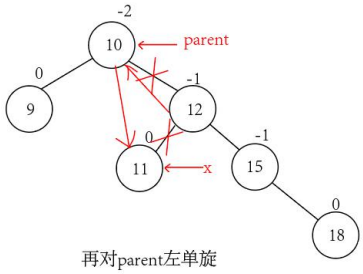


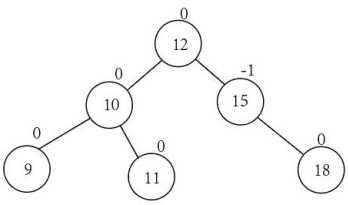
当 sub_R_L 为空时,只要令 parent 右孩子指向空即可,详细情况可看右左旋的例子

2) 右单旋与左单旋为镜像关系,不再赘述

3) 右左旋:当插入项 x 位于最近的平衡因子为-2 的祖先节点 parent 的右孩子的左子树时,左右旋.只需先对 parent 的右子树右单旋,再对 parent 左单旋







4) 左右旋与右左旋为镜像关系,不再赘述

3.3 系统的实现

（1）类的编写

系统工程名为：loginSystem。包含了 AVLnode 类（节点类），AVLtree 类（AVL 模板树）,UI 类三个基本类，由于系统中只需一种使用对象即账号密码，所以直接在 AVLnode 类里储存账号密码,用 AVL 树构造数据结构存储形式.用 UI 类做前端开发.其中最为核心的是 AVLtree 类的声明和定义.

AVLtree 类结构声明如下：

```
template <class Data>
class AVLtree
{
private:
    class node {
    public:
        Data data;
        node* left, * right, * parent;
        short int balance;//平衡因子
        node(Data d) { data = d; balance = 0; left = right = parent
= nullptr; }
    };
    node* root;
    int getHeight(node* theRoot);//求高度
    void inorderAux(node* theRoot,QTextStream txtOutput);//中序
遍历
    void LL(node* parent);//左单旋
    void RR(node* parent);//右单旋
    void LR(node* parent);//左右旋
    void RL(node* parent);//右左旋

public:
    AVLtree() { root = nullptr; };
    bool insert(Data item);//插入节点
    Data search(Data item);//查找节点
    void inOrderWrite(QString path);//层次遍历写入文件
    bool remove(Data item);//删除节点
    bool reSetData(Data item);//重新设置节点值
    void readTxt(QString path);//读取文件
};
```

（2）核心代码

系统用 AVL 树组织数据,其最为关键的核心代码就是 AVL 树的插入和删除,以及其后的四种平衡操作;

1. 插入

```
    //此处省略了查找函数
    //重新测试平衡因子并且只需左右旋等等
    while (Parent) {
        if (p == Parent->left) {
            Parent->balance++;
        }
        else if (p == Parent->right) {
            Parent->balance--;
        }

        if (!Parent->balance) {//插入后树平衡
            break;
        }
        else if (Parent->balance == 1 || Parent->balance == -1) {
            p = p->parent;
            Parent = p->parent;
        }
        else {//balance=2||-2
            int Banlance = (Parent->balance > 0 ? 1 : -1);
            if (Banlance == p->balance) {//相同即只需单旋
                if (Banlance == 1)//节点在左边,需要右旋
                    RR(Parent);
                else LL(Parent);
            }
            else {//不同则需双旋
                if (Banlance == 1)//加入的节点在左边
                    LR(Parent);
                else RL(Parent);
            }
            break;
        }
    }
    return true;
}
```

2. 删除

删除节点后的平衡就四种情况,其中两种情况平衡因子为 0,+-1,只要向上遍历就可,而+-2 则需平衡操作.

```
//平衡因子=+-2,需要进行四种操作
        else {
            node* temp;
             if(parent->balance==2){//对应左子树高
                 temp=parent->left;
                 if(temp->balance==0){
                     RR(parent);
                     parent->balance=-1;//右旋后,改变平
衡因子
                     parent=parent->parent;//更新 parent
节点指向
                     parent->balance=1;
                     break;
                 }
                 else if(temp->balance==1){
                     RR(parent);
                     parent=parent->parent;
                 }
                 else {
                     LL(parent);
                     parent=parent->parent;
                 }
             }
             else {//对应右子树高
                 temp=parent->right;
                 if(temp->balance==0){//右子树平衡
                      LL(parent);
                      parent->balance=-1;//左旋后,改变
平衡因子
                      parent=parent->parent;//更新
parent 节点指向
                      parent->balance=1;
                      break;
                 }
                 else if(temp->balance==-1){
                     LL(parent);
                     parent=parent->parent;
                 }
                 else {
                     RL(parent);
                     parent=parent->parent;
                 }
             }
             //更新 parent 指向,向上遍历
             subtree=parent;
             parent=parent->parent;
```

3. 四种平衡操作:只要实现一个单旋的操作即可实现四种.
单旋操作如下

```
template<class Data>
void AVLtree<Data>::LL(node* parent) {
    node* subR = parent->right;
    node* subR_L = subR->left;
    parent->right = subR_L;
    if (subR_L) {//subR_L 非空
        subR_L->parent = parent;
    }
    subR->left = parent;
    //改变 parent 的父亲节点,Pparent 是 parent
的父节点
    node* Pparent = parent->parent;
    parent->parent = subR;
    subR->parent = Pparent;

    //修改 parent 的父亲节点指向
    if (!Pparent) {//即改变了根节点
        root = subR;
    }
    else if (Pparent->left == parent) {
        Pparent->left = subR;
    }
    else {
        Pparent->right = subR;
    }
    subR->balance = 0;
    parent->balance = 0;
}
```

随即可实现左右旋和右左旋:

```
template<class Data>
void AVLtree<Data>::LR(node* parent) {
    LL(parent->left);
    RR(parent);
}
```

（3）交互界面以及登录菜单的实现

系统运行开始的界面如图 5 所示：

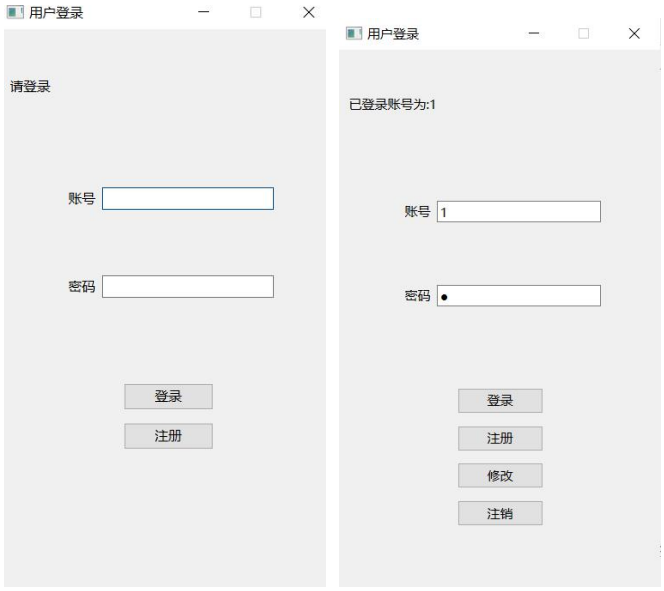


图 5　开始界面

上面的分别对应未登录和已登录的情况. 其涉及的用户交互操作较为简单, 这里不再赘述

四、　实验调试、测试、运行记录及分析

技术难点分析：

- **难点 1:左右旋之后,每个节点中有个指针指向其父节点,怎么调整他们的指向.**
- **难点 2:插入和删除操作怎么设置节点平衡因子的值,遍历还是循环?**
- **难点 3:调整平衡因子是单独写一个函数用高度来遍历每个节点并设置还是在插入和删除操作的末尾位置再平衡好**
- **难点 4:文本写入的遍历顺序那个好?一般四种遍历顺序都可以但是考虑到文件的读取,所以应该还是要考虑层次遍历更节省时间**

调试错误分析:

- **错误 1: Qt 编译器的错误,层次遍历的函数无论如何修改,编译之后会一直输出一个结果,经过多番调试,发现是编译器的问题,编译器并没有编译修改后的函数.**
- **错误 2:左右旋之后,所牵涉到的节点,其父节点指针的更新,处处卡 bug,单步调试调了好久,才写出正确的代码.**
- **错误 3:插入删除操作有许多技术难点和 bug 需要克服 1)平衡因子更新,要正确的处理平衡因子,还是应该用一个节点指针,顺着树的树干转移上去,一边转移一边平衡.这一步没做好将直接影响其他的操作,bug 频出.2)左右旋到底什么时候操作,分别对应哪几种情况.**

五、 优劣点分析:

一、优点:

1. 采用了模板类,可以实现不同的节点,比如包括账号密码昵称用户姓名等等的账户信息
2. 可视化界面,方便用户操作

二、缺点:

1. 节点类需要实现一些函数如 toString(),reset(),重载<,>,==等

六、 实验总结

本来设计了 AVL 模板树的,在代码中也确确实实实现了,但是发现需要节点实现几个函数才可以,比如重载操作符<,>,==实现 toString(),reSet()函数,这个我是首先想到了 java 的接口类,比如 java 的排序接口需要实现 compareTo 函数才可以排序,那么有没有可能在 C++中也实现这样的功能呢.于是我想到或许可以用通过继承虚基类的方法实现,在虚基类中添加 String toString()=0 等函数来实现这个功能,为方便后续的模板类的使用.然而这样我感觉反而是画蛇添足,因为用户在使用 AVL 模板树的时候并不会主动去实现节点的这几个函数或是去继承.而我在看到 Java 这样一段代码的时候反而有了新的思路:或许 C++也可以实现这样的功能.不过令人遗憾的是 C++并没有这样的功能

```
template<class T extend TYPE>
public class tree{……}
```

原本我是想做更简单的题比如大整数的实现,因为之前的编程经历使我对链表很熟悉,对树反而感觉更陌生.但是我深思熟虑后还是选择做 AVL 树,一方面,这不仅仅是有利于加深我对树的理解,更有利于我去理解数据结构更加核心的内容:数据的组织形式.链表和树其实从本质上来说就是数据和指针的合体,只不过是指针数量的多寡而已.另一方面,这也是对我编程能力的一个挑战,也是对我算法能力的一个挑战.学编程打代码其实很考验一个人的知识储备和水平,也更考验人的耐心和恒心.

七、 附录:源代码