

数据结构课程设计

实验报告

**2020/2021(1)**



实验题目 用户登录系统

学生姓名 吕锐

学生学号 201906061609

学生班级 软工02

任课教师 毛国红

提交日期 2021/1/12

**计算机科学与技术学院**

**用户登录系统 实验报告**

1. **大型实验的内容**

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，其功能包括用户登录、用户名和密码更新、用户添加和用户删除。

1. **运行环境**

在Visual Studio 6.0平台下开发，使用了MSVC v142-VS 2019 C++ x64/x86 生成工具，操作系统：Windows 10。

硬件环境：

处理器：Intel(R) Core(TM) i7-9759H CPU @ 2.60GHz 2.59GHz

内存：16.00GB

系统类型：64位操作系统

1. **实验课题分析**

**3.1 用户登录系统的主要功能**

**用户登录系统主要功能为：**用户登录、用户名和密码更新、用户添加和用户删除等工作。详细的系统功能结构为图1所示。

用户登录系统

管理系统

信息修改

用户添加

用户删除

用户登录

图1 系统结构图

**系统各模块的功能具体描述为：**

**1、用户登录模块**

用户信息(即用户名和密码)存储在文件中，输入用户名和密码，在登录界面输入用户名密码并点击“登录”后，从文件中读取所有的用户信息，并建立合适的查找二叉树。若在二叉树中查找到匹配信息，进入用户界面，否则，跳出用户名或密码错误弹框。

1. **信息修改模块**

用户在进入用户界面后可以点击修改用户名密码按钮后进入信息修改模块，输入更改后的用户名密码更新二叉树和文件，删除原信息，再插入修改后的新信息。

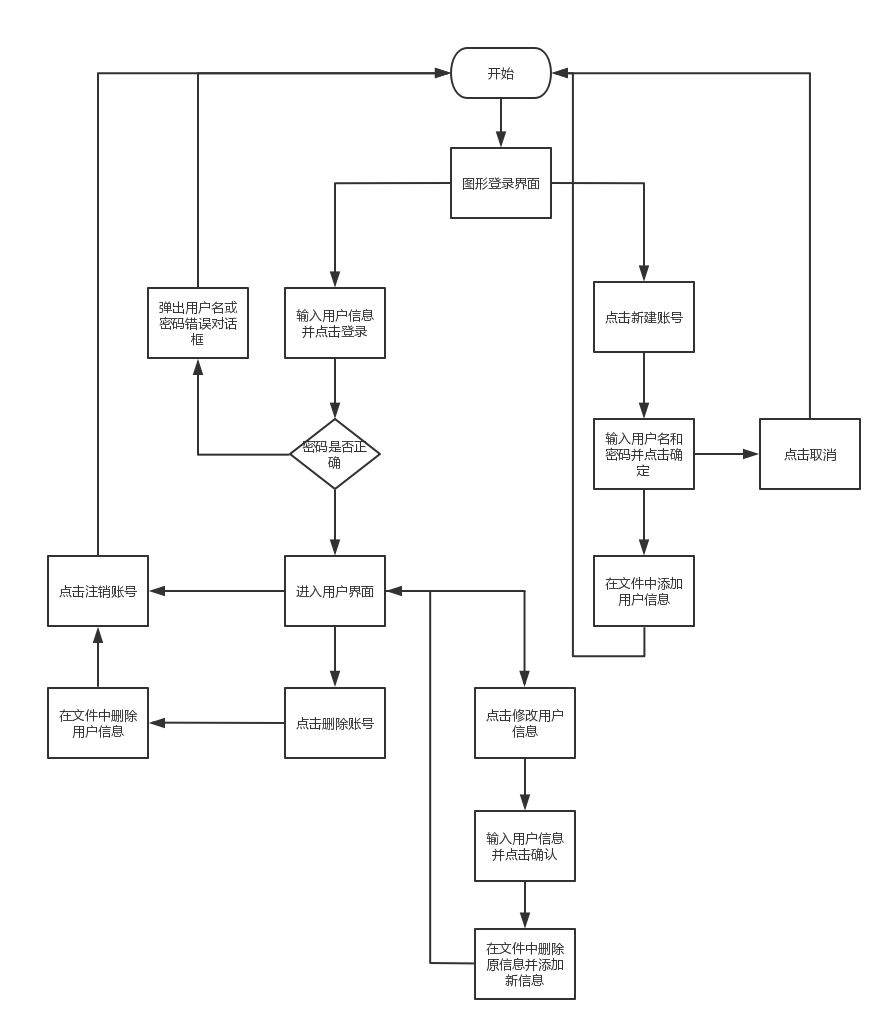
**3、用户添加模块**

在登录界面点击“注册账号”可以新添加用户信息到二叉树中并且更新文件信息。添加完毕后回到登录界面。

**4、用户删除模块**

在登陆成功进入用户界面后，可以点击“删除用户”，在二叉树删除用户信息后更新文件。

**3.2 用户登录系统程序流程**

****

**3.3文件组成**

1. **AVL树结构部分**

|  |  |
| --- | --- |
| **BST.h** | **AVL树的声明和实现** |
| **UserInfo.h** | **用户数据类的声明与实现** |
| **Data.h** | **MFC前段需要使用的数据类型存储和文件的关联** |

**2、MFC窗口部分**

|  |  |
| --- | --- |
| **MFCApplication1Dlg.h/.cpp** | **用户登录窗口** |
| **access.h/.cpp** | **用户窗口** |
| **Newuser.h/.cpp** | **新建用户窗口** |
| **Codewrong.h/cpp** | **密码错误窗口** |
| **Change.h/.cpp** | **修改信息窗口** |

**3.4基本数据结构**

采用二叉平衡查找树（AVL），采用了模板类，以用户信息（UserInfo类）作为比较的关键词进行插入。二叉平衡查找树是在二叉搜索树（BST）的基础上进行了优化，使得树基本达到平衡。定义内部类BinNode来存储AVL树的节点信息。

**1、UserInfo类：**

#include<iostream>

#include<fstream>

#include<string>

using namespace std;

class UserInfo

{

public:

inline string id() const { return myId; }

inline void read(istream& in)

{

in >> myId >> myPassword;

}

inline void display(ostream& out)

{

out << myId << " " << myPassword;

}

inline bool operator==(const UserInfo& user)const

{

return myId == user.myId && myPassword == user.myPassword;

}

inline bool operator<(const UserInfo& user)const

{

return myId < user.myId || myId == user.myId && myPassword < user.myPassword;

}

inline bool operator>(const UserInfo& user)const

{

return myId > user.myId || myId == user.myId && myPassword > user.myPassword;

}

inline void setuser(string x,string y) {

myId = x;

myPassword = y;

}

private:

string myId;

string myPassword;

};

inline istream& operator>>(istream& in, UserInfo& user)

{

user.read(in);

return in;

}

inline ostream& operator<<(ostream& out, UserInfo& user)

{

user.display(out);

return out;

}

inline string wstring2string(wstring wstr)

{

string result;

int len = WideCharToMultiByte(CP\_ACP, 0, wstr.c\_str(), wstr.size(), NULL, 0, NULL, NULL);

char\* buffer = new char[len + 1];

WideCharToMultiByte(CP\_ACP, 0, wstr.c\_str(), wstr.size(), buffer, len, NULL, NULL);

buffer[len] = '\0';

result.append(buffer);

delete[] buffer;

return result;

}

**2、BinNode内部类：**

class BinNode

{

public:

DataType data;

BinNode\* left;

BinNode\* right;

BinNode() :left(0), right(0) {}

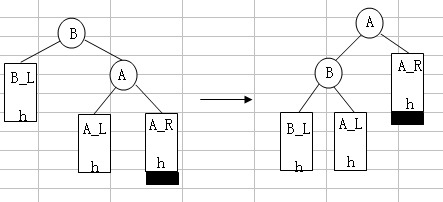
BinNode(DataType item) :data(item), left(0), right(0) {}

};

**3、AVL树实现**

AVL树相对于BST树，多了平衡两字，树都有高度，而AVL树就是要求每一个节点的左子树和右子树的高度差不超过1，这样就能使其尽可能的减小整棵树的高度，使时间复杂度能稳定在O(logN), 但我们不可能去约束用户的输入，因此，引入了四种旋转：左旋、右旋、先右旋再左旋、先左旋后右旋。

左旋：



template <class DataType>

typename BST<DataType>::BinNode\* BST<DataType>::LAdjest(BST<DataType>::BinNode\* subtreeRoot)

{

if (subtreeRoot == nullptr)

return subtreeRoot;

BinNode\* p = subtreeRoot->right;

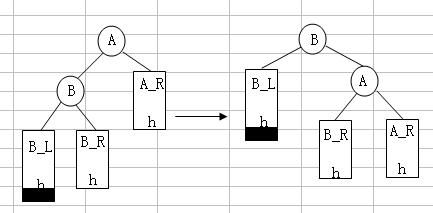
subtreeRoot->right = p->left;

p->left = subtreeRoot;

return p;

}

右旋：

****

template <class DataType>

typename BST<DataType>::BinNode\* BST<DataType>::RAdjest(BST<DataType>::BinNode\* subtreeRoot)

{

if (subtreeRoot == nullptr)

return subtreeRoot;

BinNode\* p = subtreeRoot->left;

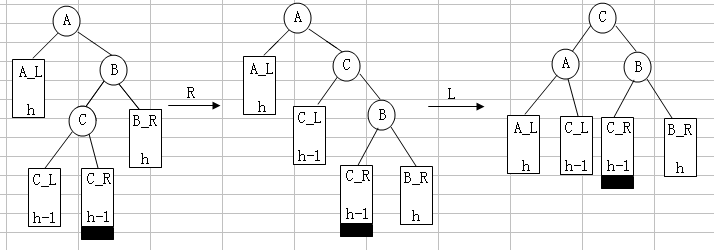
subtreeRoot->left = p->right;

p->right = subtreeRoot;

return p;

}

右左旋：

****

template <class DataType>

typename BST<DataType>::BinNode\* BST<DataType>::RLAdjest(BST<DataType>::BinNode\* subtreeRoot)

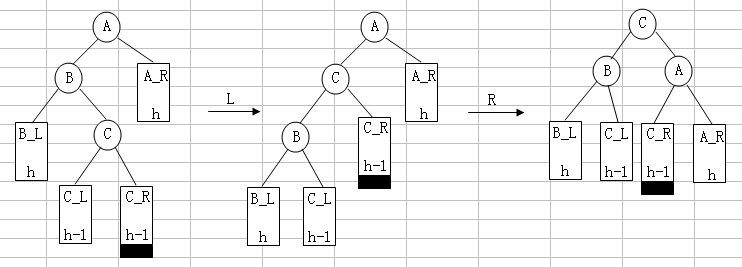
{

subtreeRoot->right = RAdjest(subtreeRoot->right);

return LAdjest(subtreeRoot);

}

左右旋：

****

template <class DataType>

typename BST<DataType>::BinNode\* BST<DataType>::LRAdjest(BST<DataType>::BinNode\* subtreeRoot)

{

subtreeRoot->left = LAdjest(subtreeRoot->left);

return RAdjest(subtreeRoot);

}

求树高函数：

template<typename DataType>

int BST<DataType>::Height(BST<DataType>::BinNode\* p) {

if (p == NULL)

return 0;

int i = Height(p->left);

int j = Height(p->right);

return i > j ? i + 1 : j + 1;

}

调节AVL树平衡：

template <class DataType>

typename BST<DataType>::BinNode\* BST<DataType>::Balance(BST<DataType>::BinNode\* subtreeRoot)

{

int bf = BalanceFactor(subtreeRoot);

if (bf > 1)

{

if (BalanceFactor(subtreeRoot->left) > 0)

subtreeRoot = RAdjest(subtreeRoot);

else

subtreeRoot = LRAdjest(subtreeRoot);

}

else if (bf < -1)

{

if (BalanceFactor(subtreeRoot->right) > 0)

subtreeRoot = RLAdjest(subtreeRoot);

else

subtreeRoot = LAdjest(subtreeRoot);

}

return subtreeRoot;

}

AVL树节点插入：

AVL树的插入过程可以分为两步：

1.按照二叉搜索树的方式插入新节点

2.调整节点的平衡因子

代码：

template<typename DataType>

typename BST<DataType>::BinNode\* BST<DataType>::insert(BinNode\* subtreeRoot, const DataType& item)

{

if (subtreeRoot == NULL) {

subtreeRoot = new BinNode(item);

}

else if (item > subtreeRoot->data)

{

subtreeRoot->right = insert(subtreeRoot->right, item);

subtreeRoot = Balance(subtreeRoot);

}

else if (item < subtreeRoot->data)

{

subtreeRoot->left = insert(subtreeRoot->left, item);

subtreeRoot = Balance(subtreeRoot);

}

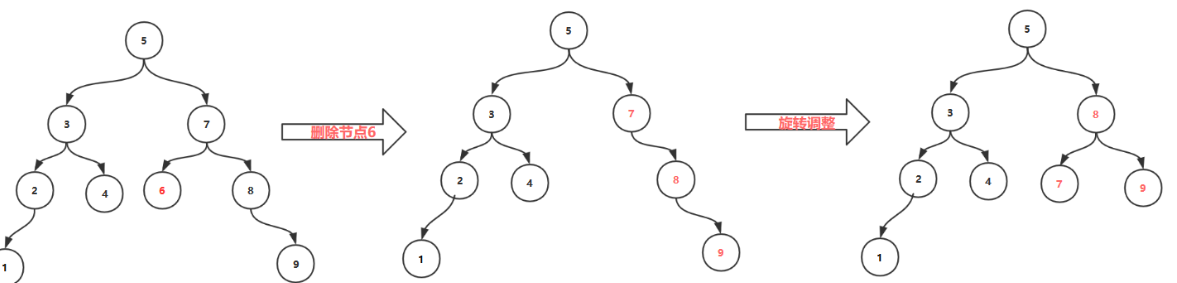
return subtreeRoot;

}

AVL树节点删除：

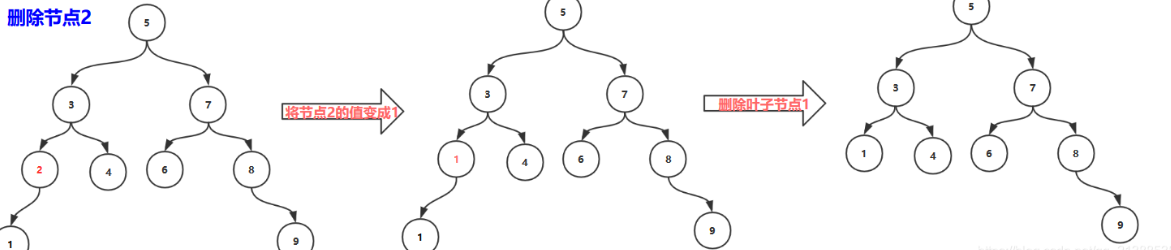
第一种情况：

删除叶子节点6

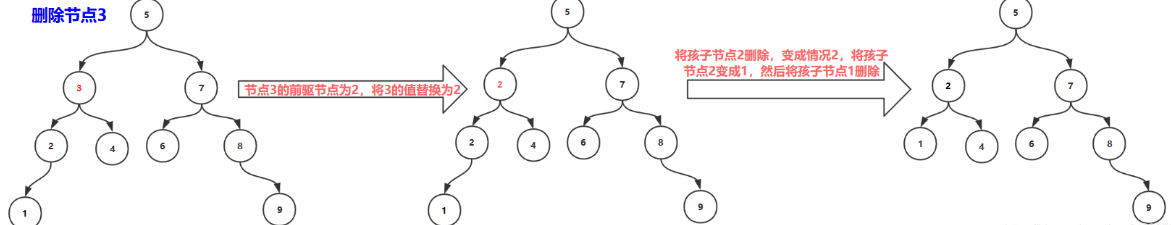
操作：直接删除，然后依次向上调整为AVL树。

第二种情况：

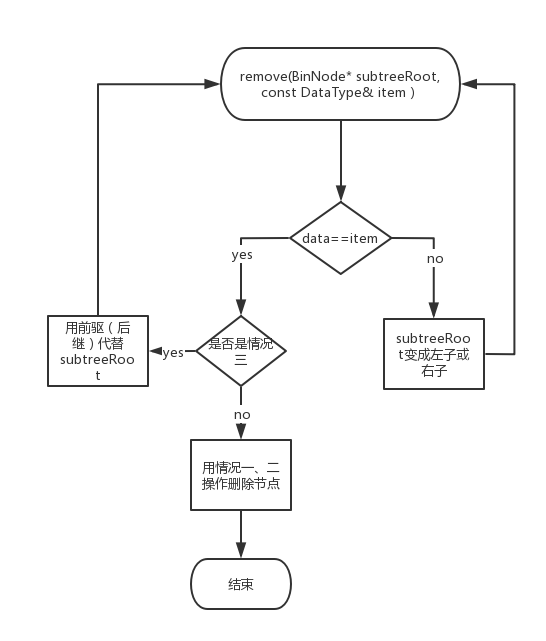
删除非叶子节点，该节点只有左孩子



第三种情况：

删除非叶子节点（节点3），非叶子节点既有左孩子，又有右孩子

删除的递归实现



template<typename DataType>

typename BST<DataType>::BinNode\* BST<DataType>::remove(BinNode\* subtreeRoot, const DataType& item)

{

if (!search(item)) {

return myRoot;

}

if (!myRoot)

return myRoot;

if (subtreeRoot->data == item)

{

if (subtreeRoot->left && subtreeRoot->right) {

if (BalanceFactor(subtreeRoot) > 0) {

BinNode\* p = TreeMax(subtreeRoot->left);

subtreeRoot->data = p->data;

subtreeRoot->left = remove(subtreeRoot->left, p->data);

}

else {

BinNode\* p = TreeMin(subtreeRoot->right);

subtreeRoot->data = p->data;

subtreeRoot->right = remove(subtreeRoot->right, p->data);

}

}

else {

BinNode\* p = subtreeRoot;

subtreeRoot = (subtreeRoot->right) ? (subtreeRoot->right) : (subtreeRoot->left);

delete p;

p = nullptr;

}

}

else if (item < subtreeRoot->data) {

subtreeRoot->left = remove(subtreeRoot->left, item);

if (BalanceFactor(subtreeRoot) < -1)

subtreeRoot = Balance(myRoot);

}

else {

subtreeRoot->right = remove(subtreeRoot->right, item);

if (BalanceFactor(subtreeRoot) > 1)

subtreeRoot = Balance(myRoot);

}

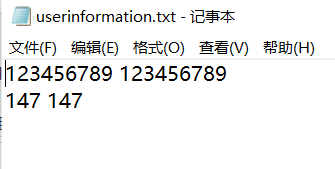
return subtreeRoot;

}

1. **实验调试、测试、运行记录及分析**

主要的测试经过如下：

开始时文件内的数据



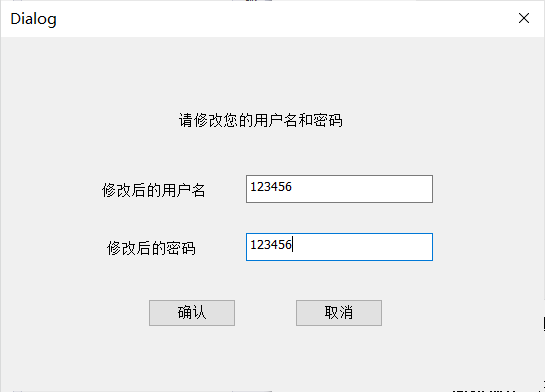
开始程序，输入用户名密码



点击登录进入用户界面



点击修改账号信息并输入修改后的信息，点击确定



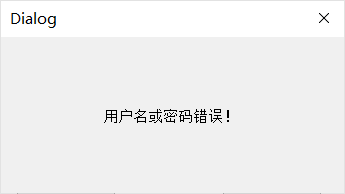
点击注销账号返回用户登录界面，输入另一个用户名密码



点击删除账号



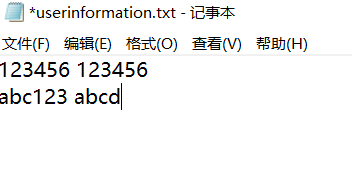
再回去登录该用户时显示错误



点击注册账号并输入用户名密码，点击确定



结束程序后文件的内容



**遇到的问题及解决方法如下：**

* + **问题1：**

**问题描述：**写删除节点的时候很难下手，因为不知道怎么在用三种删除方案删除的时候保持树的平衡。

**解决方法：**使用递归删除节点，仅在树变为1、2情况时删除并调节平衡树。

* + **问题2：**

**问题描述：**写平衡操作时难以计算平衡因子，写了也会让代码一层套一层变的很冗长。

**解决方法：**专门引入算树高的Height函数和算平衡因子的函数BalanceFactor函数，

每次调整时调用。

* + **问题3：**

**问题描述：**MFC一个窗口到另一个窗口变量难以传导。

**解决方法：**专门建了一个文件存变量，发现还是只有部分能传部分不能传，找规律发现只有大写C开头的变量类型和指针能传。

1. **实验总结**

我设计的用户登录系统基本满足任务书的所有要求。

从结构上来讲，用户登录系统使用了AVL树加快了查找、插入删除速度，用链式结构节省了空间开销。还使用了模板加强了灵活性, 可重用性和可扩展性。加之几乎完成了BST常用的所有功能，不仅限于该系统的实现，我相信它在各种需求下只需微调便可以在很多程序中使用。

从人机交互讲，它使用了MFC设计的前端，相比控制台指令更直观的把功能呈现给了用户，多窗口的设计也使得该系统完成度更高。

存在的缺点主要是出于日常接触的商用用户系统软件，我在每次回用户登录窗口的时候都重新从文件读取一次二叉树，好处是用户信息会实时更新（多人同时读一个文件），但是这让数据量大的时候的返回用户登录窗口时会消耗不必要的时间。

通过这次C++的大型实验，我得到以下几点经验：

1. 初步了解算法、结构的思想、原理 整明白抽象结构，想要弄清楚一个算法的实现，首先要大致知道这个算法、结构的原理，这是最简单的一步，也是最基础的一步。

2. 查找各种结构资料 发现许多不同的结构，其实书上列举的只是数据结构的一小部分，而且对于一个算法，也有各种不同的实现方法。有了这些知识储备和实践，我们可以提出问题与同学交流或者解答同学的问题。

3.我们还是要注重看懂弄懂会做书上的代码，毕竟那是最基础也是最重要的了解他们的目的，在此基础上深入的了解算法的实现过程，并且最终实用到自己的设计中，而不是在死记硬背，那样会事倍功半。

4. 坚持上机操作，实践出真知 和所有计算机类的学习一样，数据结构是非常需要动手的一门课程，看书看一天也不如把书上的代码打到电脑上实践一遍，发现错误弄懂，那收获必定匪浅的，上级的很多内容在课设中都可以直接或间接受益。掌握好平时实验知识，课设就已经成功一部分了。

1. **附录：源代码**