项目说明文档

数据结构课程设计

——家谱管理系统

作 者 姓 名：杨鑫

学　　号：1950787

指 导 教 师：张颖

学院、 专业：软件学院 软件工程

同济大学

Tongji University

目录

1 分析

1.1 背景分析

家谱，又称族谱、宗谱等。是一种以表谱形式，记载一个家族的世系繁衍及重要人物事迹的书。家谱是一种特殊的文献，就其内容而言，是中华文明史中具有平民特色的文献，记载的是同宗共祖血缘集团世系人物和事迹等方面情况的历史图籍。家谱属珍贵的人文资料，对于历史学、民俗学、人口学、社会学和经济学的深入研究，均有其不可替代的独特功能。

以往的家谱都是用纸张来进行记载的，这种记载方式不仅修改起来比较麻烦，而且还很容易丢失，复制起来也十分不方便。如果制作一个家谱管理系统，用计算机来进行管理，那么如果需要修改家谱的话，只需要按操作输入修改信息，计算机就可以自动进行修改，十分便捷。同时家谱保存在计算机中，也较为安全，不易丢失。因此，开发一个基于计算机操作的家谱管理系统是十分有必要的。

1.2 功能分析

作为一个家谱管理系统，首先要能够初始化输入家族的祖先，这样才能进行之后的操作。其次，家谱系统还需要能够不断添加新的家庭成员，完善整个家谱。如果家庭内部出现离异等情况，还需要将离异的家庭解散掉。最后，家庭内部如果有人改名，系统应当也可以对应修改此人的名字。为了方便用户操作，家谱管理系统可以设置菜单栏来指引用户。

综上所述，该家谱管理系统需要有初始化，完善家谱，添加家庭成员，解散家庭，更改成员姓名，展示家谱信息，退出系统的功能。

2 设计

2.1 数据结构设计

如上功能分析所述，该家谱管理系统要求大量增加，删除操作。增加操作要求在一个父亲下添加多名子女，删除操作要求将目标结点以及目标的所有子女全部删除。家谱的结构类似于一种树的结构。增加时在一个树枝上加上叶子，删除时将树枝和上面的叶子一起砍掉。因此决定采用树结构来储存家谱。由于每个父结点的子女个数不确定，因此使用多叉树数据结构，并采用长子女-兄弟链表表示法来存储。同时，每次增加，删除，修改都需要找到目标结点后执行操作，而普通的树性能较差，需要的时间复杂度为 O(n),所以还增加了一棵 AVL 树，用于保存树中的节点，每次操作时只需通过 AVL 树找到目标节点然后执行操作即可，时间复杂度降为 O(log n)，大大提高了性能。

2.2 类结构设计

为了能够正常的储存和查找家谱信息，该系统中设计了一个家族树类（GenealogyTree）和一个 AVL 树类（AVLTree），以及它们的结点结构体：家族树结点（GenealogyTreeNode）与 AVL 树结点（AVLNode）。同时，在这些类提供的一些操作里面，为了实现一定的功能，还使用到了栈和队列。因此程序中还有节点结构体（LinkedNode），栈类（LinkedStack），队列类（LinkedQueue）。此外，程序还有一个家谱类（Genealogy），包含了一个家谱树和一个 AVL 树，同时整合了所有的操作，便于用户调用。为了使 LinkedStack,LinkedQueue 类与 GenealogyTree,AVLTree 树类更具有泛用性，本系统将它们以及其结点类都设计为了模板类。

2.3 成员与操作设计

节点（LinkedNode）

```
1.  T data;
2.  LinkNode <T>* link;
3.  LinkNode(LinkNode<T>* ptr = NULL) : link(ptr) {}; // 构造函数
4.  LinkNode(const T& tem, LinkNode<T>* ptr = NULL) : data(tem), link
    (ptr) {}; // 构造函数
```

栈（LinkedStack）

私有成员：

```
1.  LinkNode<T>* top;  // 栈顶元素
```

公有操作：

```
1.  LinkedStack() : top(NULL) {} // 构造函数
2.  ~LinkedStack() { makeEmpty(); } // 析构函数
```

```
3.   void Push(const T& x); // 入栈
4.   bool Pop(T& x); // 出栈
5.   bool getTop(T& x) const; // 得到栈顶元素
6.   bool IsEmpty() const { return top == NULL; } // 判断是否栈空
7.   int getSize() const; // 返回栈中元素个数
8.   void makeEmpty(); // 栈置空
```

队列（LinkedQueue）

私有成员：

```
1.   LinkNode<T>* front, * rear;
```

公有操作：

```
1.   LinkedQueue() : front(NULL), rear(NULL) {} // 构造函数
2.   ~LinkedQueue() { makeEmpty(); } // 析构函数
3.   bool EnQueue(const T& x); // 进入队列
4.   bool DeQueue(T& x); // 出队列
5.   bool getFront(T& x) const; // 返回 front
6.   void makeEmpty(); // 置空
7.   bool IsEmpty() const { return front == NULL; } // 判断队列是否为
     空
8.   int getSize() const; // 返回队列元素个数
9.
10.   // 友元函数， 实现队列输出
11.  friend ostream& operator << (ostream& os, const LinkedQueue<T>& Q)
     ;
```

家谱树节点（GenealogyTreeNode）

```
1.   T data;
2.   GenealogyTreeNode<T>* pre, *son, *brother; // 分别指向上一节点（父亲
     或者哥哥），儿子，弟弟
3.   GenealogyTreeNode() : pre(NULL), son(NULL), brother(NULL) {} //
     构造函数
4.   GenealogyTreeNode(T& newData, GenealogyTreeNode<T>* p = NULL, Gen
     ealogyTreeNode<T>* s = NULL, GenealogyTreeNode<T>* b = NULL) : data
     (newData), pre(p), son(s), brother(b) {} // 构造函数
5.   ~GenealogyTreeNode() {} // 析构函数
6.   bool operator < (const GenealogyTreeNode<T>& GTN) { return this->
     data < GTN.data; } // 重载小于符号
7.   bool operator > (const GenealogyTreeNode<T>& GTN) { return this->
     data > GTN.data; } // 重载大于符号
8.   bool operator == (const GenealogyTreeNode<T>& GTN) { return this-
     >data == GTN.data; } // 重载 == 符号
```

家谱树（GenealogyTree）

私有成员：

```
GenealogyTreeNode<T>* root;
void makeEmpty(GenealogyTreeNode<T>*& ptr); // 置空
```

公有操作：

```
GenealogyTree() : root(NULL) {}  // 构造函数
~GenealogyTree() { makeEmpty(root); } // 析构函数
bool IsEmpty() { return root == NULL; } // 检测是否为空
void SetRoot(GenealogyTreeNode<T>*& R); // 设置根节点
bool Insert(GenealogyTreeNode<T>*& d, GenealogyTreeNode<T>*& fa); // 插入节点
bool Remove(GenealogyTreeNode<T>*& d); // 删除节点
void Show(); //打印家谱树
```

AVL 树节点（AVLNode）

```
K key;
int bf;
AVLNode<K>* left, *right;
AVLNode() : bf(0), left(NULL), right(NULL) {} // 构造函数
AVLNode(K newKey, AVLNode<K>* newLeft = NULL, AVLNode<K>* newRight = NULL) : key(newKey), bf(0), left(newLeft), right(newRight) {} // 构造函数
~AVLNode() {} // 析构函数

// 三个重载比较运算符的函数
bool operator < (const AVLNode<K>& AN) { return this->key < AN.key; }
bool operator > (const AVLNode<K>& AN) { return this->key > AN.key; }
bool operator == (const AVLNode<K>& AN) { return this->key == AN.key; }
```

AVL 树（AVLTree）

私有成员：

```
AVLNode<K>* root;
bool Insert(AVLNode<K>*& ptr, K& d); // 插入
bool Remove(AVLNode<K>*& ptr, K& d); // 删除
void RotateL(AVLNode<K>*& ptr); // 左单旋转
void RotateR(AVLNode<K>*& ptr); // 右单旋转
void RotateLR(AVLNode<K>*& ptr); // 先左后右旋转
void RotateRL(AVLNode<K>*& ptr); // 先右后左旋转
AVLNode<K>* Search(K& d, AVLNode<K>*& ptr); // 查找函数
void makeEmpty(AVLNode<K>*& ptr); // 置空
```

公有操作：

```
AVLTree() : root(NULL) {} // 构造函数
~AVLTree() { makeEmpty(root); } // 析构函数
AVLNode<K>* Search(K& d) { return Search(d, root); } // 查找
bool Insert(K& d) { return Insert(root, d); } // 插入
bool Remove(K& d, bool tag); // 删除
bool Update(K& curData, K& newData); // 更新
void SetRoot(K& d); // 设置根节点
```

家谱类（Genealogy）

私有成员：

```
GenealogyTree<string>* Tree; // 家谱树， 用于存放所有成员信息， 并体现他们的辈分关系
AVLTree<GenealogyTreeNode<string>*>* AVLGenealogy; // AVL 家谱树， 以平衡二叉排序树的形式存放所有成员信息， 可以减少操作的时间复杂度
```

公有操作：

```
Genealogy(); // 构造函数
~Genealogy(); // 析构函数
void Complete(); // 完善家庭
void AddMember(); // 添加成员
void DeleteFamily(); // 解散子家庭
void SearchMember(); // 寻找成员
void ChangeMember(); // 修改成员
void ShowGenealogy(); // 打印家谱
void Loop(); // 主循环函数
```

2.4 系统设计

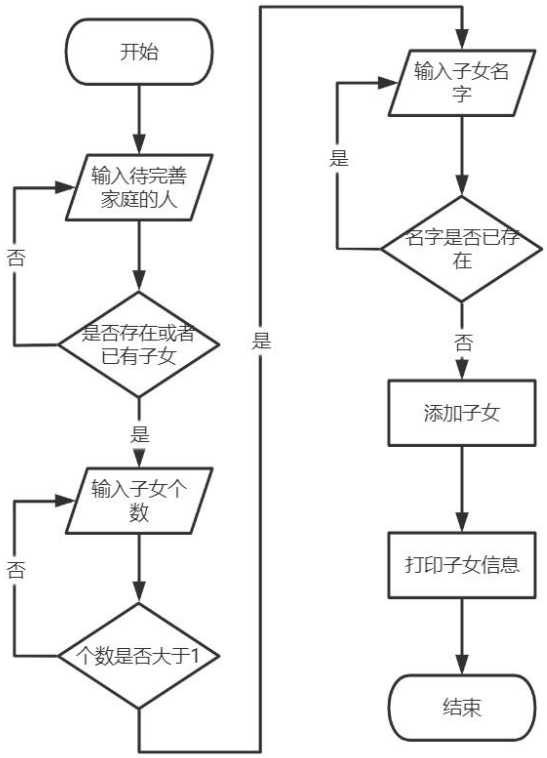
程序开始时，由用户输入家谱的祖先的名字后建立一个初始的家谱，然后不断的从用户处接受指令并完成相应的操作（完善家庭，添加子女，解散家庭，查找家庭成员，展示家谱信息等）。每次操作会先通过 AVL 树进行处理，找到目标节点之后再由普通的家谱树处理，这样子可以大大节省花费的时间，有利于提高程序的性能。

程序兼容了 windows 和 LINUX 平台，在双平台下均可以正常运行。

3 实现

3.1 完善家庭功能的实现

3.1.1 完善家庭功能的流程图

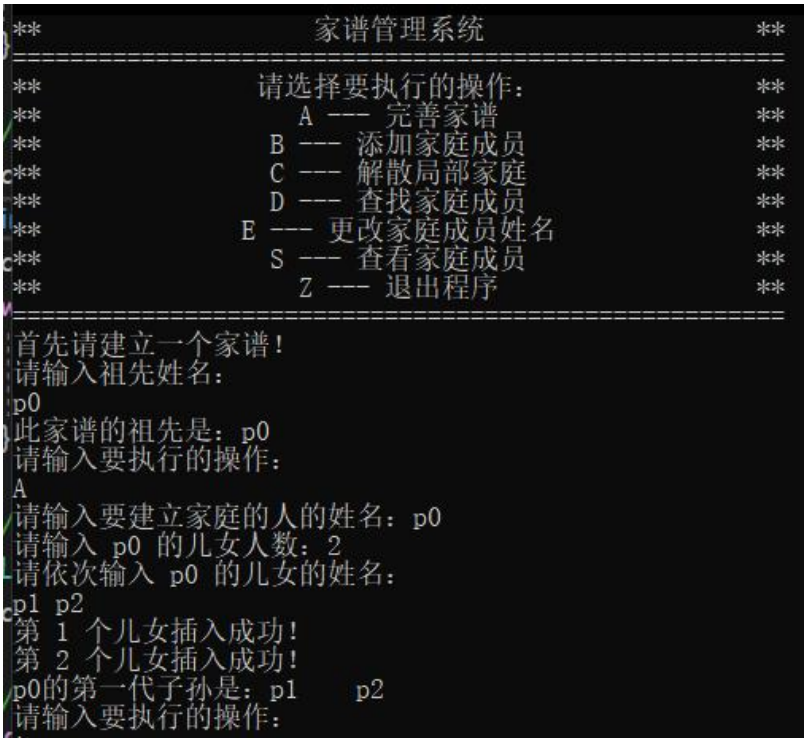


3.1.2 完善家庭功能的核心代码

```
1.   // 用一个栈来存储本次输入的子女信息
2.   LinkedQueue<string>* nameQueue = new LinkedQueue<string>;
3.   cout << "请依次输入 " << name << " 的儿女的姓名： " << endl;
4.
5.   // 依次插入
6.   for (int i = 0; i < nums; i++) {
7.       string temName;
8.       cin >> temName;
9.        GenealogyTreeNode<string>* inTN = new GenealogyTreeNode<strin
   g>(temName);
10.
11.      // 尝试一次插入
12.      if (AVLGenealogy->Insert(inTN)) {
13.          // 插入成功， 将新节点与其父节点链接起来， 并入队列
```

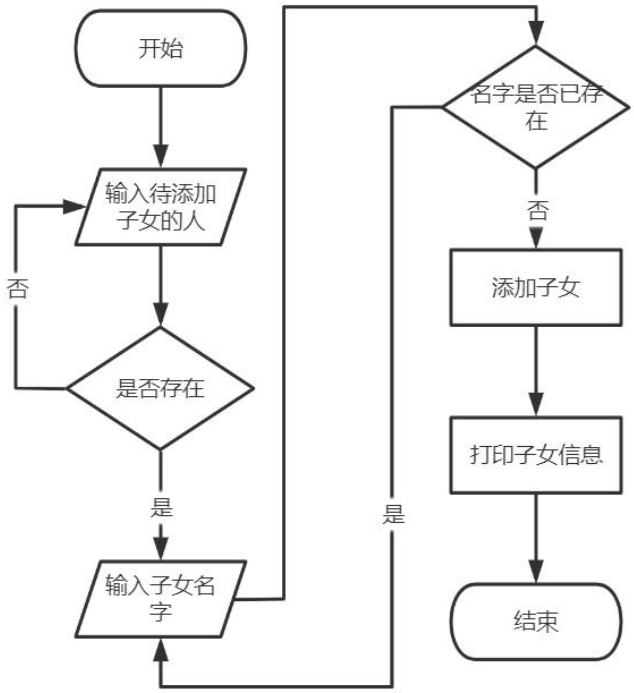
```
            Tree->Insert(inTN, temTN);
            nameQueue->EnQueue(temName);
            cout << "第 " << i + 1 << " 个儿女插入成功！" << endl;
        }
        else {
            // 插入失败， 重新输入
            cout << "第 " << i + 1 << " 个儿女插入失败！请重新输入该子女及其以后的子女姓名：";
            i--;
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }
    }

    // 利用队列打印本次插入的子女信息
    cout << name << "的第一代子孙是：";
    cout << *nameQueue << endl;
    delete nameQueue;
```

3.1.3 完善家庭功能的截屏示例



3.2 添加成员功能的实现

3.2.1 添加成员功能的流程图



3.2.2 添加成员功能的核心代码

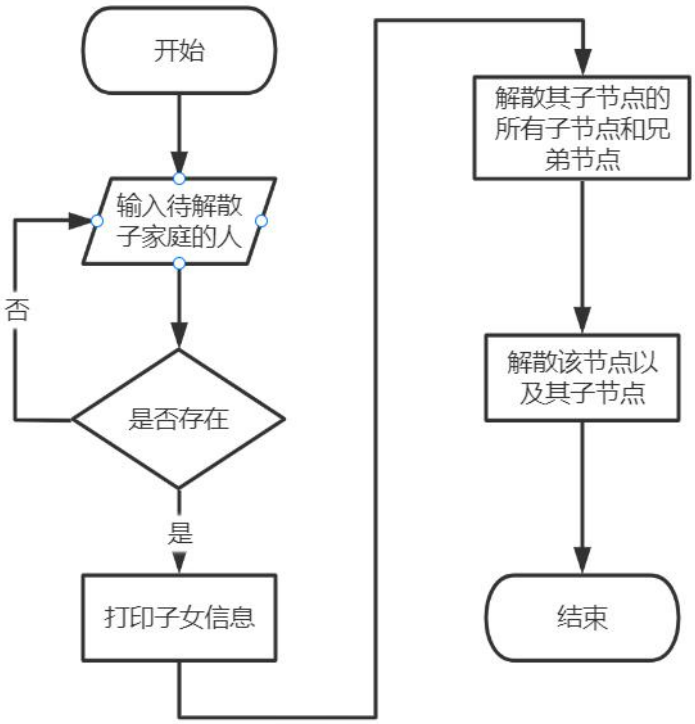
```
// 若其父节点原本无儿子， 该节点成为其父节点长子
if (fa->son == NULL) {
    fa->son = d;
    d->pre = fa;
}
else {
    // 否则， 找到其最后一个兄弟节点， 然后链接在后面
    GenealogyTreeNode<T>* pr = fa->son;
    while (pr->brother != NULL) {
        pr = pr->brother;
    }
    pr->brother = d;
    d->pre = pr;
}
return true;
```

3.2.3 添加成员功能的截屏示例

```
**                    家谱管理系统                    **
=======================================================
**                  请选择要执行的操作：              **
**                    A --- 完善家谱                  **
**                    B --- 添加家庭成员              **
**                    C --- 解散局部家庭              **
**                    D --- 查找家庭成员              **
**                  E --- 更改家庭成员姓名            **
**                    S --- 查看家庭成员              **
**                    Z --- 退出程序                  **
=======================================================
首先请建立一个家谱！
请输入祖先姓名：
p0
此家谱的祖先是：p0
请输入要执行的操作：
B
请输入要添加儿子（或女儿）的人的姓名：p0
请输入 p0 新添加的儿子（或女儿）的姓名：
p1
插入成功！
p0的第一代子孙是：p1
请输入要执行的操作：
```

3.3 解散子家庭功能的实现

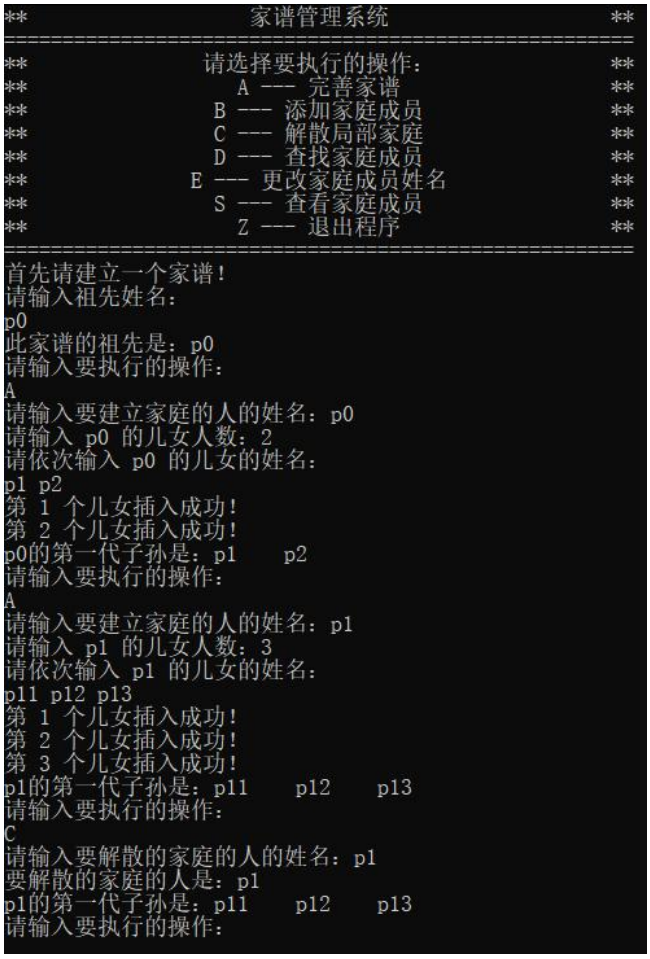
3.3.1 解散子家庭功能的流程图



3.3.2 解散子家庭功能的核心代码

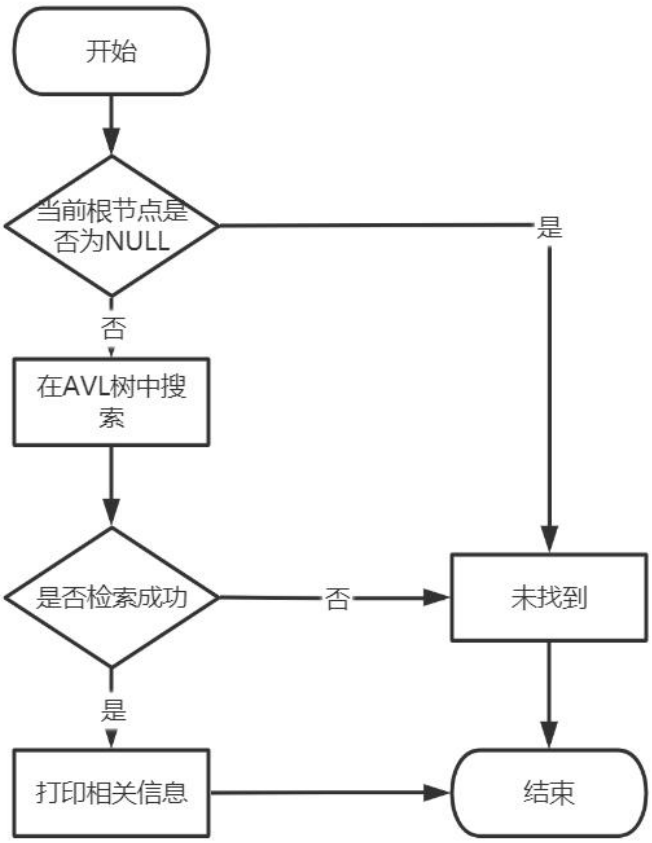
```
if (d == NULL) return false;

// 删除子树
if (d->son != NULL) {
    Remove(d->son, tag);
}

// 若 tag 为 true， 则删除后继兄弟节点
if (tag && d->brother != NULL) {
    Remove(d->brother, tag);
}
Remove(root, d); // 删除该节点
```

3.3.3 解散子家庭功能的截屏示例



3.4 寻找成员功能的实现

3.4.1 寻找成员功能的流程图

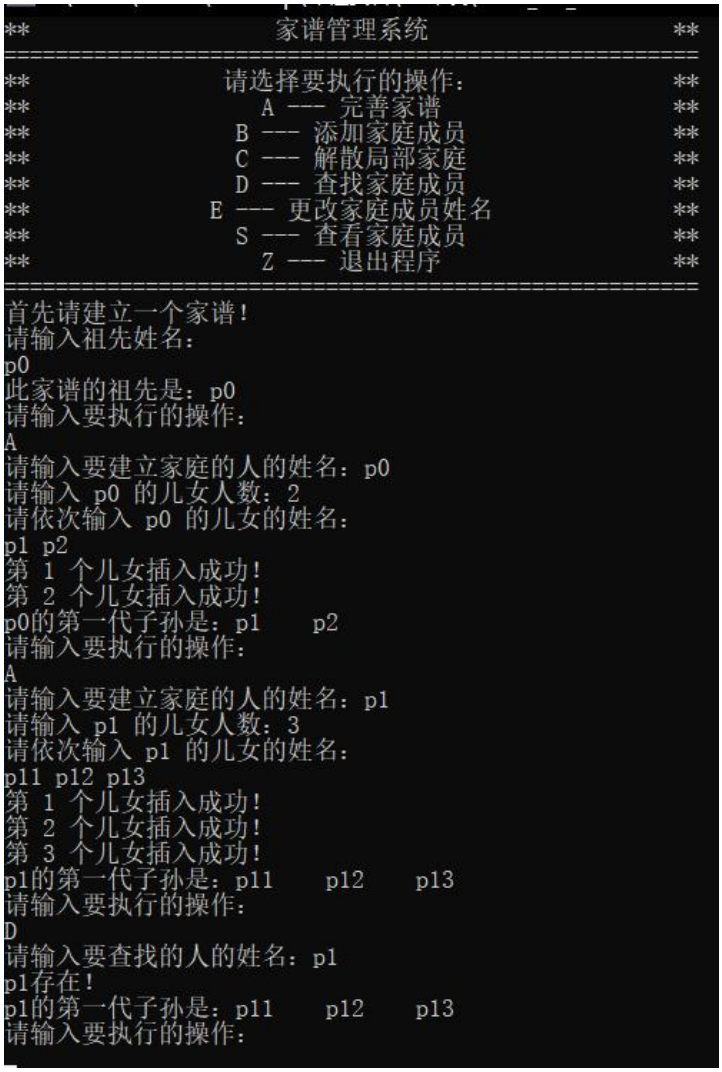


3.4.2 寻找成员功能的核心代码

```
1.   cout << "请输入要查找的人的姓名：";
2.   string name;
3.   cin >> name;
4.   GenealogyTreeNode<string>* temTN = new GenealogyTreeNode<string>(
   name);
5.   AVLNode< GenealogyTreeNode<string>*>* temAN = NULL;
6.
7.   // 在 AVL 树中查找
8.    // 查找失败
9.   if ((temAN = AVLGenealogy->Search(temTN)) == NULL) {
10.      cout << "该成员不存在！" << endl;
11.      delete temTN;
12.      return;
13.  }
14.
```

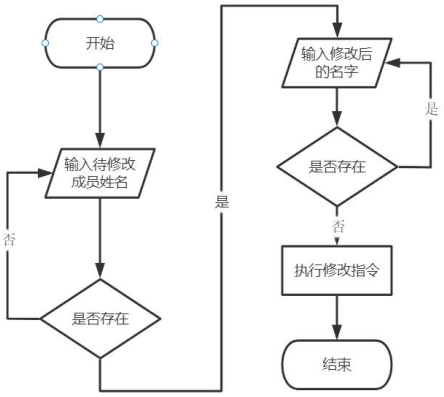
```
15.  delete temTN;
16.  temTN = temAN->key;
17.
18.  // 查找成功， 打印相关信息
19.  cout << name << "存在！" << endl;
20.  cout << name << "的第一代子孙是：";
21.  GenealogyTreeNode<string>* p = temTN->son;
22.  while (p != NULL) {
23.      cout << p->data << "    ";
24.      p = p->brother;
25.  }
26.  cout << endl;
```

3.4.3 寻找成员功能的截屏示例



3.5 修改成员功能的实现

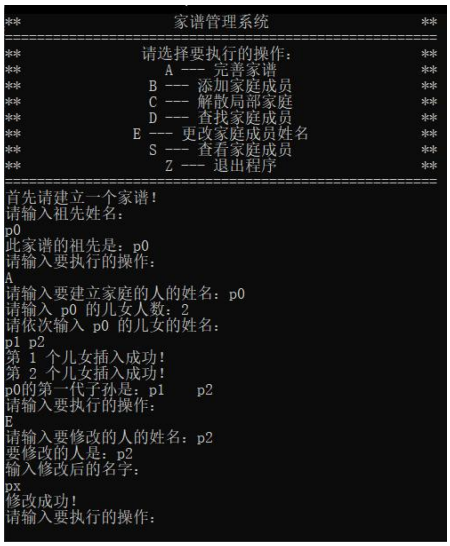
3.5.1 修改成员功能的流程图



3.5.2 修改成员功能的核心代码

```
if (Search(newData, root) != NULL) {
    return false;
}

// 先移除原节点， 修改值后再次插入， 确保仍然是符合二叉排序树的
Remove(root, curData);
curData->data = newData->data;
Insert(curData);
return true;
```

3.5.3 修改成员功能的截屏示例



4 测试

4.1 功能测试

4.1.1 完善家庭功能测试

截屏示例：见 3.1.3

4.1.2 添加成员功能测试

截屏示例：见 3.2.3

4.1.3 解散子家庭功能测试

截屏示例：见 3.3.3

4.1.4 寻找成员功能测试

截屏示例：见 3.4.3

4.1.5 修改成员功能测试

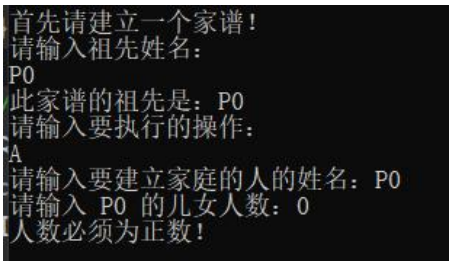
截屏示例：见 3.5.3

经过多次测试，基本功能均正确执行，无异常情况出现。

4.2 边界测试

4.2.1 完善家谱时输入添加儿女个数为零

测试用例：
P0
A
P0
0
预期结果：程序给出提示信息，程序正常运行不崩溃。
实验结果：



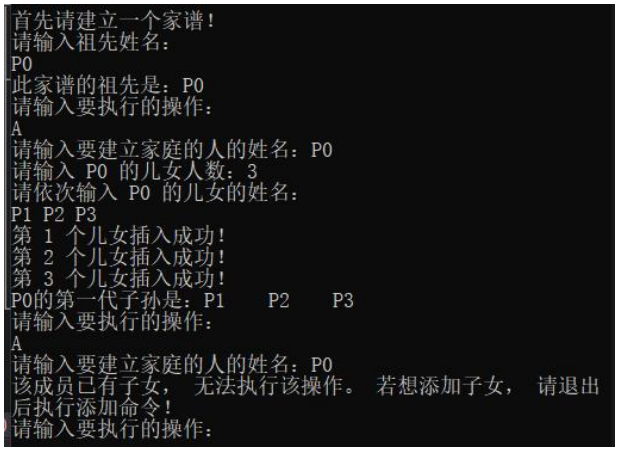
4.2.2 对已经建立家庭的人再次建立家庭

测试用例：

P0
A
P0
3
P1 P2 P3
A
P0

预期结果：程序给出提示信息，程序正常运行不崩溃。

实验结果：



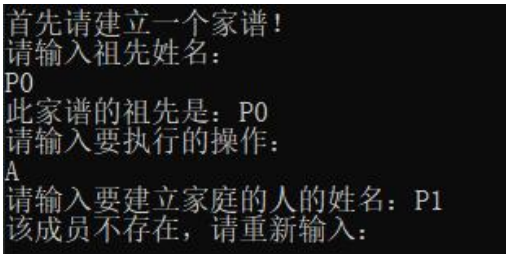
4.3 出错测试

4.3.1 完善家谱或添加家庭成员时输入的姓名不存在

测试用例：

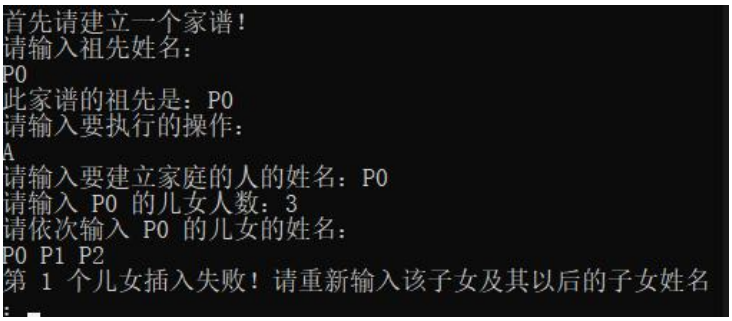
P0
A
P1

预期结果：程序给出提示信息，程序正常运行不崩溃。
实验结果：



4.3.2 完善家谱或添加子女时新添加儿女姓名在家谱中已存在

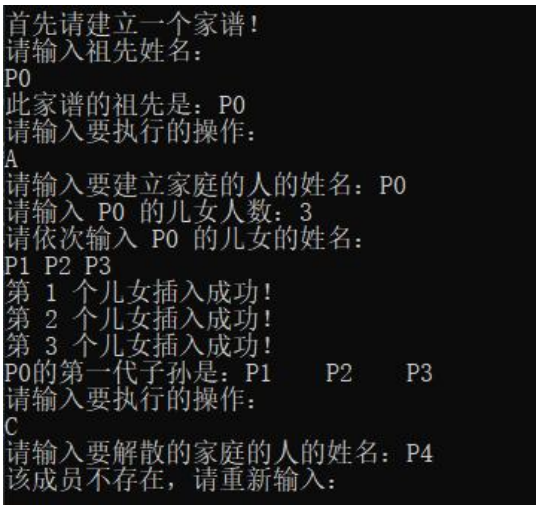
测试用例：
P0
A
P0
3
P0 P1 P2
预期结果：程序给出提示信息，程序正常运行不崩溃。
实验结果：



4.3.3 解散局部家庭时或更改成员姓名时输入的姓名不存在

测试用例：
P0
A
P0
3
P1 P2 P3
C
P4
预期结果：程序给出提示信息，程序正常运行不崩溃。

实验结果：



4.3.4 更改成员姓名时输入的新姓名在家谱中已存在

测试用例：

P0

A

P0

3

P1 P2 P3

E

P1

P2

预期结果：程序给出提示信息，程序正常运行不崩溃。

实验结果：