项目说明文档

数据结构课程设计

——**家谱管理系统**

作 者 姓 名： 陈翔飞

学 号： 1851756

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 4](#_Toc27603830)

[1.1 背景分析 4](#_Toc27603831)

[1.2 功能分析 4](#_Toc27603832)

[2 设计 4](#_Toc27603833)

[2.1 数据结构设计 4](#_Toc27603834)

[2.2 类结构设计 4](#_Toc27603835)

[2.3 成员与操作设计 4](#_Toc27603836)

[2.4 系统设计 12](#_Toc27603837)

[3 实现 13](#_Toc27603838)

[3.1 建立家谱功能的实现 13](#_Toc27603839)

[3.1.1 建立家谱流程图 13](#_Toc27603840)

[3.1.2 建立家谱功能核心代码 13](#_Toc27603841)

[3.1.3 建立家谱功能截屏示例 14](#_Toc27603842)

[3.2 建立家庭功能的实现 15](#_Toc27603843)

[3.2.1 建立功能流程图 15](#_Toc27603844)

[3.2.2 建立家庭功能核心代码 16](#_Toc27603845)

[3.2.3 建立家庭功能截屏示例 18](#_Toc27603846)

[3.3 解散家庭功能的实现 20](#_Toc27603847)

[3.3.1 解散家庭功能流程图 20](#_Toc27603848)

[3.3.2 解散家庭功能核心代码 20](#_Toc27603849)

[3.3.3 解散家庭功能截图示例 21](#_Toc27603850)

[3.4 添加子女功能的实现 23](#_Toc27603851)

[3.4.1 添加子女功能流程图 23](#_Toc27603852)

[3.4.2 添加子女功能核心代码 23](#_Toc27603853)

[3.4.3 添加子女功能截屏示例 25](#_Toc27603854)

[3.5 更改姓名功能的实现 26](#_Toc27603855)

[3.5.1 更改姓名功能流程图 26](#_Toc27603856)

[3.5.2 更改姓名核心代码 26](#_Toc27603857)

[3.5.3 更改姓名功能截屏示例 28](#_Toc27603858)

[3.6 查询功能的实现 29](#_Toc27603859)

[3.6.1 查询功能流程图 29](#_Toc27603860)

[3.6.3 查询功能截屏示例 30](#_Toc27603861)

[3.6 总体系统的实现 31](#_Toc27603862)

[3.6.1 总体系统流程图 31](#_Toc27603863)

[3.6.2 总体系统核心代码 31](#_Toc27603864)

[3.6.3 总体系统截屏示例 32](#_Toc27603865)

[4 测试 33](#_Toc27603866)

[4.1 功能测试 33](#_Toc27603867)

[4.1.1 建立家庭功能测试 33](#_Toc27603868)

[4.1.2 添加子女功能测试 35](#_Toc27603869)

[4.1.3 解散家庭功能测试 36](#_Toc27603870)

[4.1.4 修改姓名功能测试 37](#_Toc27603871)

[4.1.5 查询功能功能测试 38](#_Toc27603872)

[4.2 边界测试 39](#_Toc27603873)

[4.2.1 建立家庭时无输入 39](#_Toc27603874)

[4.2.2 解散祖先家庭 40](#_Toc27603875)

[4.2.3 查询的人不存在于家谱中 41](#_Toc27603876)

[4.2.4 查询的人没有任何子女 42](#_Toc27603877)

[4.2.5 更改姓名前后姓名一样 43](#_Toc27603878)

[4.3 出错测试 43](#_Toc27603879)

[4.3.1 子女人数错误 43](#_Toc27603880)

[4.3.2 建立家庭时输入姓名不存在 44](#_Toc27603881)

[4.3.3 建立家庭时子女姓名于已有家庭成员重复 45](#_Toc27603882)

[4.3.4 建立家庭时子女姓名有重复项 46](#_Toc27603883)

[4.3.5 解散家庭的人不存在 47](#_Toc27603884)

[4.3.6 添加子女的人不存在 48](#_Toc27603885)

[4.3.7 添加的子女姓名与原家谱内成员重复 49](#_Toc27603886)

[4.3.8 更改姓名的人不存在 49](#_Toc27603887)

[4.3.9 更改姓名的新姓名与原家谱中成员重复 50](#_Toc27603888)

# 1 分析

## 背景分析

家谱，又称族谱、家乘、祖谱、谱牒、宗谱等，是记载某个姓氏家族子孙世系传承之书，具有区 分家族成员血缘关系亲疏远近的作用，是中国封建宗法制度的产物。随着历史的发展，家谱由官修变为私修，所录内容不断丰富，其作用也不断增加和变化。如今，家谱同各姓氏的郡望、堂号一样，不仅为区别姓氏源流，可作为数典认祖、研究历史、地理、社会、民俗等参考资料，它还 是姓氏文化的重要组成部分。

## 1.2 功能分析

作为一个家谱管理系统，其作用是帮助用户对家谱进行管理，以管理家庭中成员的信息。首先应该有的功能就是建立家谱，其次，家谱管理系统还应该具有查找功能，这是家谱其他功能的基础。然后家谱还应该具有插入、修改、删除的功能，以此来实现对家谱中家庭成员信息的管理。

综上所述，该家谱管理系统需要有建立家谱、查找、插入、删除、修改、退出的功能。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，我们需要频繁地进行数据的插入与删除，并且家谱的结构也是一种树的结构，所以本项目采用树的数据结构来存储家谱，因为家谱中每个结点的子女结点的个数可能不尽相同，所以采用了树的子女链表表示法。

## 2.2 类结构设计

为了保证设计的数据结构的泛用性，本项目选择将MultiNode类和MultiTree类都设计为模板类，MultiNode类储存数据和父子兄弟，MultiTree类储存树根，并提供展示子女、清空等接口。同时为了根据名字快速查找族谱中的人，还设计了平衡二叉树AVLTree。同时为了动态保存输入以进行检查，还设计了Vector类。最后本项目设计了一个族谱类，以将所有接口集合在一起。

## 2.3 成员与操作设计

**多叉树结点类（MultiNode）**

**类定义：**

1. **template**<**typename** ElementType> **class** MultiNode
2. {
3. **friend** **class** MultiTree<ElementType>;
4. **public**:
5. ~MultiNode<ElementType>() {};
6. MultiNode<ElementType>() = **default**;
7. MultiNode<ElementType>(**const** ElementType& E) : Element(E) {};
8. **bool** IsRoot() { **return** Father == nullptr; }
9. **bool** IsFirstSon() { **return** Father != nullptr&& Father->FirstSon == **this**; }
10. MultiNode<ElementType>\* GetLastSon();
11. MultiNode<ElementType>\* FirstSon = nullptr, \*NextBro = nullptr, \*Father = nullptr;
12. ElementType Element;
13. };

**公有成员：**

MultiNode<ElementType> \*FirstSon;//节点的首儿子

MultiNode<ElementType> \*NextBro; //节点的下一兄弟

MultiNode<ElementType> \*Father;//节点的父亲

ElementType Element;//节点储存的数据

**公有操作：**

~MultiNode<ElementType>() {};

//析构函数

MultiNode<ElementType>() = default;

//默认构造函数

MultiNode<ElementType>(const ElementType& E) : Element(E) {};

//带数据参数的构造函数

bool IsRoot() { return Father == nullptr; }

//判断该节点是否为根

bool IsFirstSon() { return Father != nullptr&& Father->FirstSon == this; }

//判断该节点是否为父亲的第一个儿子

**多叉树类（MultiTree）**

**类定义：**

1. **template**<**typename** ElementType> **class** MultiTree
2. {
3. **public**:
4. MultiTree<ElementType>() = **default**;
5. ~MultiTree<ElementType>();
7. **void** DisplayAllChild(ostream& os,MultiNode<ElementType>\* N);
8. **void** Destory(MultiNode<ElementType>\* N);
9. **void** Clear();
10. MultiNode<ElementType>\* Root = nullptr;
11. };

**公有成员：**

MultiNode<ElementType>\* Root；//树根

**公有操作：**

void DisplayAllChild(ostream& os,MultiNode<ElementType>\* N);

//展示节点的所有儿子

void Destory(MultiNode<ElementType>\* N);

//销毁节点N及其所有子孙节点

void Clear();

//清空整棵树

MultiTree<ElementType>() = default;

//默认构造函数

~MultiTree<ElementType>();

//析构函数通过调用Clear()来回收内存

**AVL树结点类（TreeNode）**

**类定义：**

1. **template** <**typename** Key, **typename** Value> **class** TreeNode
2. {
3. **public**:
4. **void** MidVisit(ostream& os);
5. TreeNode<Key, Value>() = **default**;
6. TreeNode<Key, Value>(Key K, Value V);
7. ~TreeNode<Key, Value>();
8. Key ElementKey;
9. Value ElementValue;
10. TreeNode<Key, Value>\* Left = nullptr, \*Right = nullptr;
11. **int** Height = 0;
12. };

**公有成员：**

Key ElementKey；//树节点的键，作为平衡树的关键词

Value ElementValue；//树节点的值，储存需要保存的信息

TreeNode<Key,Value>\* Left；//树节点的左儿子

TreeNode<Key,Value>\* Right；//树节点的右儿子

Int Height；//树节点的高度，用于维护平衡

**公有操作：**

TreeNode<Key, Value>() = default; //默认构造函数

TreeNode<Key, Value>(Key K, Value V); //含键值对参数的构造函数

void MidVisit(ostream& os);//中序遍历该节点并输出值

**AVL树类（AVLTree）**

**类定义：**

1. **template** <**typename** Key = **int**, **typename** Value = **int**> **class** AVLTree
2. {
4. **public**:
5. AVLTree<Key, Value>();
6. AVLTree<Key, Value>(Key K, Value V);
7. ~AVLTree<Key, Value>();
8. TreeNode<Key, Value>\* Change(Key K, Value V);
9. TreeNode<Key, Value>\* Insert(Key K, Value V = Value());
10. TreeNode<Key, Value>\* Find(Key K);
11. TreeNode<Key, Value>\* Delete(Key K);
12. **void** MakeEmpty();
13. **void** DisPlay(ostream& os);
14. **int** GetHeight();
15. TreeNode<Key, Value>\* UncheckChange(TreeNode<Key, Value>\* P, Value V);
16. TreeNode<Key, Value>\* UncheckInsert(Key K, Value V);
17. **int** GetSize();
18. **private**:
19. **int**  Max(**int** x, **int** y);
20. **int** GetHeight(TreeNode<Key, Value>\* P);
21. TreeNode<Key, Value>\* SingleRotateWithLeft(TreeNode<Key, Value>\* K2);
22. TreeNode<Key, Value>\* SingleRotateWithRight(TreeNode<Key, Value>\* K2);
23. TreeNode<Key, Value>\* DoubleRotateWithLeft(TreeNode<Key, Value>\* K3);
24. TreeNode<Key, Value>\* DoubleRotateWithRight(TreeNode<Key, Value>\* K3);
25. TreeNode<Key, Value>\* Find(Key K, TreeNode<Key, Value>\* P);
26. TreeNode<Key, Value>\* FindMin(TreeNode<Key, Value>\* P);
27. TreeNode<Key, Value>\* FindMax(TreeNode<Key, Value>\* P);
28. TreeNode<Key, Value>\* Insert(Key K, Value V, TreeNode<Key, Value>\* P);
29. TreeNode<Key, Value>\* Delete(Key K, TreeNode<Key, Value>\* P);
30. TreeNode<Key, Value>\* MakeEmpty(TreeNode<Key, Value>\* P);
31. **int** Size = 0;
32. TreeNode<Key, Value>\* Root = nullptr;
33. };

**私有成员：**

int Size = 0;//AVL树的大小，即储存键值对数

TreeNode<Key, Value>\* Root ;//AVL树的树根

**私有操作：**

int Max(int x, int y); //返回x，y中的最大值

int GetHeight(TreeNode<Key, Value>\* P); //获取节点P的高度

TreeNode<Key, Value>\* SingleRotateWithLeft(TreeNode<Key, Value>\* K2);

//左单旋，用于维护平衡

TreeNode<Key, Value>\* SingleRotateWithRight(TreeNode<Key, Value>\* K2);

//右单旋，用于维护平衡

TreeNode<Key, Value>\* DoubleRotateWithLeft(TreeNode<Key, Value>\* K3);

//左双旋，用于维护平衡

TreeNode<Key, Value>\* DoubleRotateWithRight(TreeNode<Key, Value>\* K3);

//右双旋，用于维护平衡

TreeNode<Key, Value>\* Find(Key K, TreeNode<Key, Value>\* P);

//以节点P为根，查找有关键词K的节点

TreeNode<Key, Value>\* FindMin(TreeNode<Key, Value>\* P);

//以节点P为根，查找最小节点

TreeNode<Key, Value>\* FindMax(TreeNode<Key, Value>\* P);

//以节点P为根，查找最大节点

TreeNode<Key, Value>\* Insert(Key K, Value V, TreeNode<Key, Value>\* P);

//以节点P为根，插入键值对<K,V>

TreeNode<Key, Value>\* Delete(Key K, TreeNode<Key, Value>\* P);

//以节点P为根，删除有关键词K的节点

TreeNode<Key, Value>\* MakeEmpty(TreeNode<Key, Value>\* P);

//清空节点P以及P的所有子节点并释放内存

**公有操作：**

AVLTree<Key, Value>();

//AVLTree的默认构造函数

AVLTree<Key, Value>(Key K, Value V);

//AVLTree的带键值对参数的构造函数，以参数中的键值对开辟一个树节点并使Root指向该节点

~AVLTree<Key, Value>();

//AVLTree的析构函数，通过调用MakeEmpty()实现对内存的回收

TreeNode<Key, Value>\* Change(Key K, Value V);

//将树中以K为键的节点的值改为V，如果不存在就插入键值对<K,V>

TreeNode<Key, Value>\* Insert(Key K, Value V = Value());

//将键值对<K,V>插入树中，如果键K已经存在则将对应节点中的值改为V

TreeNode<Key, Value>\* Find(Key K);

//返回键为K的树节点指针

TreeNode<Key, Value>\* Delete(Key K);

//删除键为K的树节点

void MakeEmpty();

//清空整个树并释放内存

void DisPlay(ostream& os);

//中序展示整棵树的所有树节点

int GetHeight();

//返回树的高度

TreeNode<Key, Value>\* UncheckChange(TreeNode<Key, Value>\* P, Value V);

//直接将节点P的值改为V

TreeNode<Key, Value>\* UncheckInsert(Key K, Value V);

//不检查K是否存在于树中，直接插入键值对<K,V>，如果插入失败则返回空

int GetSize();

//返回已经储存的键值对数量

**向量类（Vector）**

**类定义：**

1. **template**<**typename** ElementType> **class** Vector
2. {
3. **public**:
4. ~Vector<ElementType>();
5. Vector<ElementType>() = **default**;
6. Vector<ElementType>(**const** Vector<ElementType> & v);
7. Vector<ElementType>& operator = (**const** Vector<ElementType>&v);
8. **void** PushBack(**const** ElementType& t);
9. **void** PopBack();
10. **void** Clear();
11. **int** GetSize() **const**;
12. **void** ReSize(**int** NewSize);
13. ElementType& operator[](**int** Index) **const**;
14. **bool** Empty()**const**;
15. **const** ElementType& Back() **const**;
16. **private**:
17. **void** Extend();
18. **int** Size = 0;
19. **int** Capacity = 0;
20. ElementType\* Array = nullptr;
21. };

**私有成员：**

int Size;//Vector中实际储存的元素数量

int Capacity;//Vector已经申请的空间

ElementType\* Array;//储存的数据的起始地址

**私有操作：**

void Extend();

//扩容函数，当容量不足时调用

**公有操作：**

~Vector<ElementType>();

//析构函数，通过调用Clear()来释放内存

Vector<ElementType>() = default;

//默认构造函数

Vector<ElementType>(const Vector<ElementType> & v);

//拷贝构造函数

Vector<ElementType>& operator = (const Vector<ElementType>&v);

//重载=运算符，使该类支持赋值运算

void PushBack(const ElementType& t);

//向Vector末尾添加一个元素

void PopBack();

//删除末尾的元素

void Clear();

//清空Vector，释放内存

int GetSize() const;

//返回储存元素的数量

void ReSize(int NewSize);

//重设Vector的大小

ElementType& operator[](int Index) const;

//重载[]运算符，使Vector可以像数组一样使用

bool Empty()const;

//判断Vector是否为空

const ElementType& Back() const;

//返回末尾的元素

**族谱类（Genealogy）**

**类定义：**

1. **class** Genealogy
2. {
3. **public**:
5. Genealogy() = **default**;
6. ~Genealogy() = **default**;
7. **void** Menu();
8. **void** Init();
9. **void** Build();
10. **void** Add();
11. **void** Dissolve();
12. **void** Query();
13. **void** Change();
14. **private**:
15. MultiNode<string> \*AddNewSon(MultiNode<string>\* N, **const** string& E);
16. MultiNode<string> \*AddNewSon(MultiNode<string>\* N, **const** Vector<string>& V);
17. **void** Display(MultiNode<string>\* N);
18. **bool** CheckNewChild(**const** Vector<string>& AllChild);
19. **bool** CheckNewChild(**const** string& Child);
20. MultiTree<string> FamilyTree;
21. AVLTree<string, MultiNode<string>\*> FamilyMap;
22. };

**私有成员：**

MultiTree<string> FamilyTree;

// 家族树，储存成员姓名与亲属关系

AVLTree<string, MultiNode<string>\*> FamilyMap;

//家族表，用于通过姓名快速查找家族树中的对应节点

**私有操作：**

MultiNode<string> \*AddNewSon(MultiNode<string>\* N, const string& E);

//向节点N添加单个子女

MultiNode<string> \*AddNewSon(MultiNode<string>\* N, Vector<string>& V);

//向节点N添加多个子女

void Display(MultiNode<string>\* N);

//展示节点N的所有第一代子女

bool CheckNewChild(const Vector<string>& AllChild);

//批量检查子女姓名是否合法

bool CheckNewChild(const string& Child);

//检查单个子女姓名是否合法

**公有操作：**

Genealogy() = default;

//默认构造函数

~Genealogy() = default;

//析构函数

void Menu();

//显示菜单

void Init();

//初始化族谱

void Build();

//建立家庭

void Add();

//增加子女

void Dissolve();

//解散家庭

void Query();

//查询

void Change();

//更改成员姓名

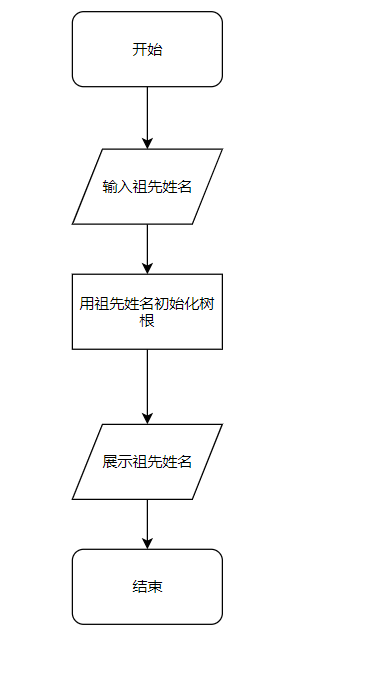
## 2.4 系统设计

系统首先调用Menu ()函数实现对屏幕的初始化，然后调用Init（）初始化族谱，然后根据用户所输入的操作码（Cmd）执行家谱对应的成员函数。

# 3 实现

## 3.1 建立家谱功能的实现

### 3.1.1 建立家谱流程图

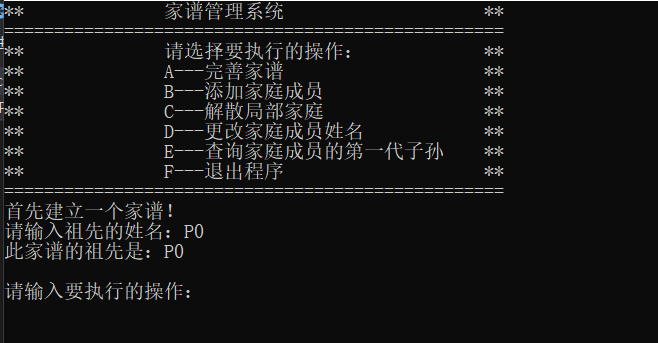


### 3.1.2 建立家谱功能核心代码

**用户界面接口：**

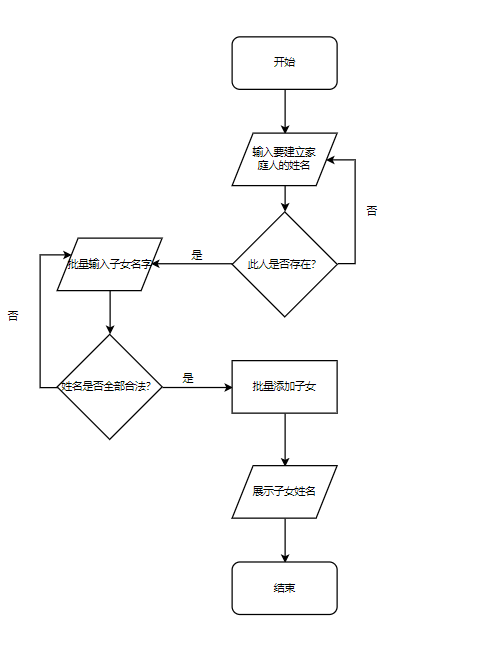
1. **void** Genealogy::Init()
2. {
3. FamilyTree.Clear();
4. cout << "首先建立一个家谱！" << endl;
5. cout << "请输入祖先的姓名：";
6. string s;
7. cin >> s;
8. FamilyTree.Root=**new**  MultiNode<string>(s);
9. FamilyMap.Insert(s, FamilyTree.Root);
10. cout << "此家谱的祖先是：" << s << endl;
11. }

### 3.1.3 建立家谱功能截屏示例



## 3.2 建立家庭功能的实现

### 3.2.1 建立功能流程图



### 3.2.2 建立家庭功能核心代码

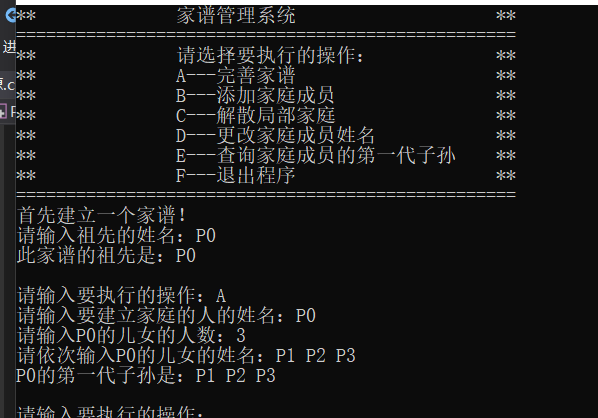
**用户界面接口：**

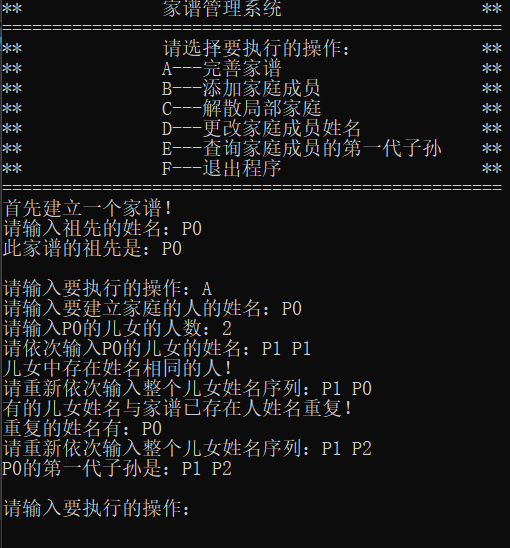
1. **void** Genealogy::Build()
2. {
3. string s;
4. TreeNode<string, MultiNode<string>\*>\* Ancestor;
5. cout << "请输入要建立家庭的人的姓名：" ;
6. cin >> s;
7. **while**((Ancestor=FamilyMap.Find(s)) == nullptr)
8. {
9. cout << "此人不存在！" << endl;
10. cout << "请重新输入：";
11. cin >> s;
12. }
13. MultiNode<string>\* AncestorNode = Ancestor->ElementValue;
14. **int** Num = 0;
15. cout << "请输入" << s << "的儿女的人数：";
16. cin >>  Num;
17. **while** (Num <= 0)
18. {
19. cout << "请输入一个大于0的整数！" << endl;
20. cout << "请重新输入：";
21. cin >> Num;
22. }
23. Vector<string> AllChild;
24. AllChild.ReSize(Num);
25. **int** i;
26. cout << "请依次输入" << s << "的儿女的姓名：";
27. **for** (i = 0; i < Num; ++i)
28. {
29. cin >> AllChild[i];
30. }
31. **while** (CheckNewChild(AllChild) == **false**)
32. {
33. cout << "请重新依次输入整个儿女姓名序列：";
34. **for** (i = 0; i < Num; ++i)
35. {
36. cin >> AllChild[i];
37. }
38. }
39. AddNewSon(AncestorNode, AllChild);
40. Display(AncestorNode);
41. }

**族谱内部接口：**

1. MultiNode<string>\* Genealogy::AddNewSon(MultiNode<string>\* N, **const** Vector<string>& V)
2. {
3. **if** (N == nullptr || V.Empty()) **return** nullptr;
4. MultiNode<string>\* NewNode = **new** MultiNode<string>(V[0]);
5. FamilyMap.Insert(V[0], NewNode);
6. MultiNode<string>\* Last = N->GetLastSon();
7. **if** (Last == nullptr) N->FirstSon = NewNode;
8. **else** Last->NextBro = NewNode;
9. Last = NewNode;
10. Last->Father = N;
11. **int** i;
12. **for** (i = 1; i < V.GetSize(); ++i)
13. {
14. string t = V[i];
16. Last->NextBro =**new** MultiNode<string>(V[i]);
17. Last->NextBro->Father = Last;
18. Last = Last->NextBro;
19. FamilyMap.Insert(V[i], Last);
20. }
21. **return** Last;;
22. }
23. **bool** Genealogy::CheckNewChild(**const** Vector<string>& AllChild)
24. {
25. **int** i;
26. AVLTree<string> NameSet;
27. **for** (i = 0; i < AllChild.GetSize(); ++i)
28. {
29. **if** (NameSet.Find(AllChild[i]) != nullptr)
30. {
31. cout << "儿女中存在姓名相同的人！" << endl;
32. **return** **false**;
33. }
34. NameSet.Insert(AllChild[i]);
35. }
36. **bool** Flag = **true**;
37. **for** (i = 0; i < AllChild.GetSize(); ++i)
38. {
39. **if** (FamilyMap.Find(AllChild[i]) != nullptr)
40. {
41. **if** (Flag == **true**)
42. {
43. cout << "有的儿女姓名与家谱已存在人姓名重复！" << endl;
44. cout << "重复的姓名有：";
45. Flag = **false**;
46. }
47. cout << AllChild[i] << " ";
48. }
49. }
50. **if**(Flag==**false**) cout << endl;
51. **return** Flag;
52. }

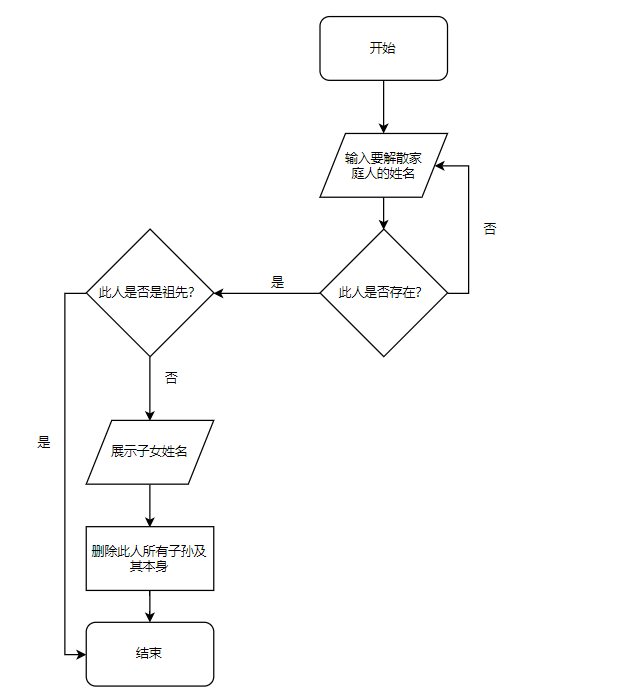
### 3.2.3 建立家庭功能截屏示例





## 3.3 解散家庭功能的实现

### 3.3.1 解散家庭功能流程图



### 3.3.2 解散家庭功能核心代码

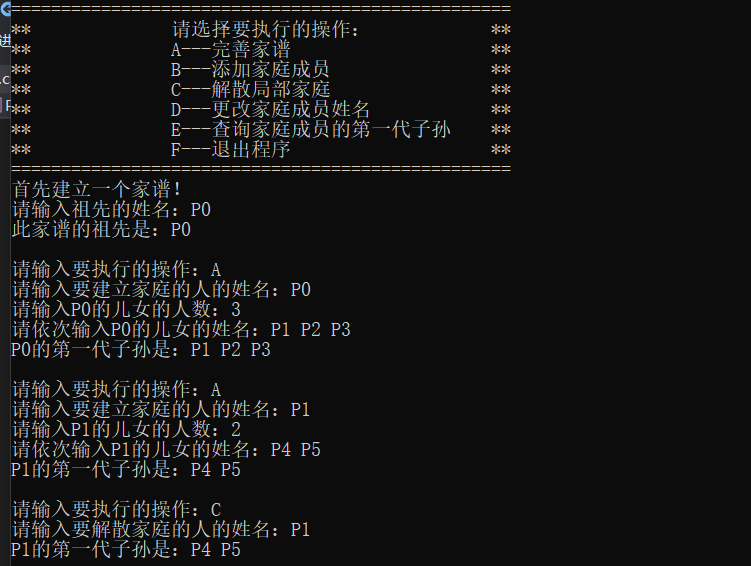
**用户界面接口：**

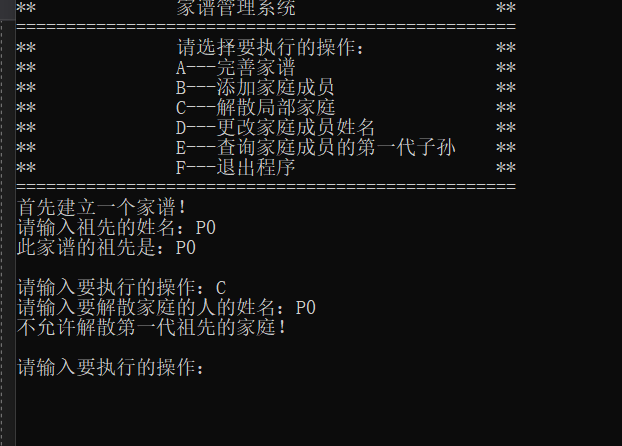
1. **void** Genealogy::Dissolve()
2. {
3. string s;
4. TreeNode<string, MultiNode<string>\*>\* Ancestor;
5. cout << "请输入要解散家庭的人的姓名：";
6. cin >> s;
7. **while** ((Ancestor = FamilyMap.Find(s)) == nullptr)
8. {
9. cout << "此人不存在！" << endl;
10. cout << "请重新输入：";
11. cin >> s;
12. }
13. MultiNode<string>\* AncestorNode = Ancestor->ElementValue;
14. **if** (AncestorNode == FamilyTree.Root)
15. {
16. cout << "不允许解散第一代祖先的家庭！" << endl;
17. **return**;
18. }
19. Display(AncestorNode);
20. **if** (AncestorNode->Father != nullptr)
21. {
22. **if** (AncestorNode->IsFirstSon() == **false**)
23. AncestorNode->Father->NextBro = AncestorNode->NextBro;
24. **else** AncestorNode->Father->FirstSon = AncestorNode->NextBro;
25. }
26. FamilyTree.Destory(AncestorNode->FirstSon);
27. **delete** AncestorNode;
28. FamilyMap.Delete(s);
29. }

**多叉树接口：**

1. **template**<**typename** ElementType>
2. **void** MultiTree<ElementType>::Destory(MultiNode<ElementType>\* N)
3. {
4. **if** (N == nullptr) **return**;
5. Destory(N->FirstSon);
6. Destory(N->NextBro);
7. **delete** N;
8. }

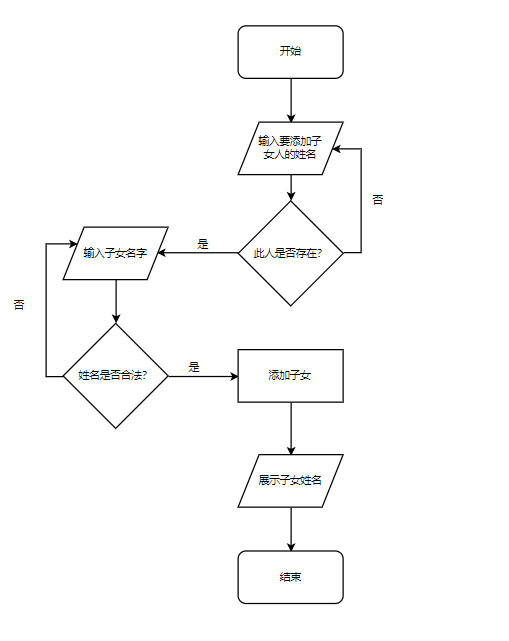
### 3.3.3 解散家庭功能截图示例





## 3.4 添加子女功能的实现

### 3.4.1 添加子女功能流程图



### 3.4.2 添加子女功能核心代码

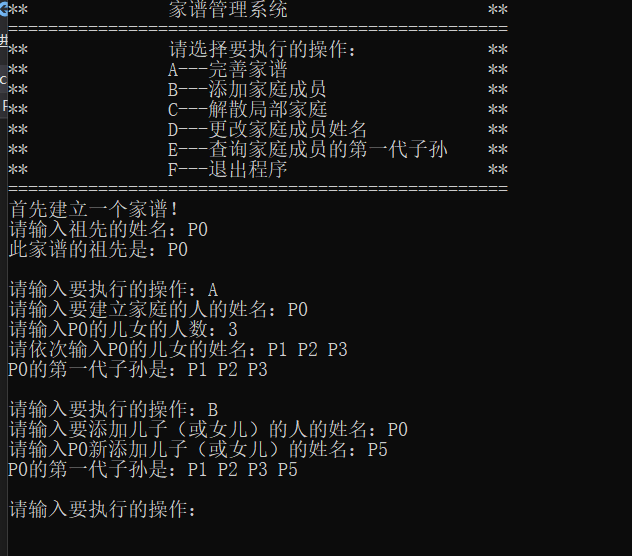
**用户界面接口：**

1. **void** Genealogy::Add()
2. {
3. string s;
4. cout << "请输入要添加儿子（或女儿）的人的姓名：";
5. cin >> s;
6. TreeNode<string, MultiNode<string>\*>\* Ancestor;
7. **while** ((Ancestor = FamilyMap.Find(s)) == nullptr)
8. {
9. cout << "此人不存在！" << endl;
10. cout << "请重新输入：";
11. cin >> s;
12. }
13. MultiNode<string>\* AncestorNode = Ancestor->ElementValue;
15. string Child;
16. cout << "请输入"<<s<<"新添加儿子（或女儿）的姓名：";
17. cin >> Child;
18. **while** (CheckNewChild(Child) == **false**)
19. {
20. cout << "请重新输入：";
21. cin >> Child;
22. }
23. AddNewSon(AncestorNode, Child);
24. Display(AncestorNode);
26. }

**家谱内部接口：**

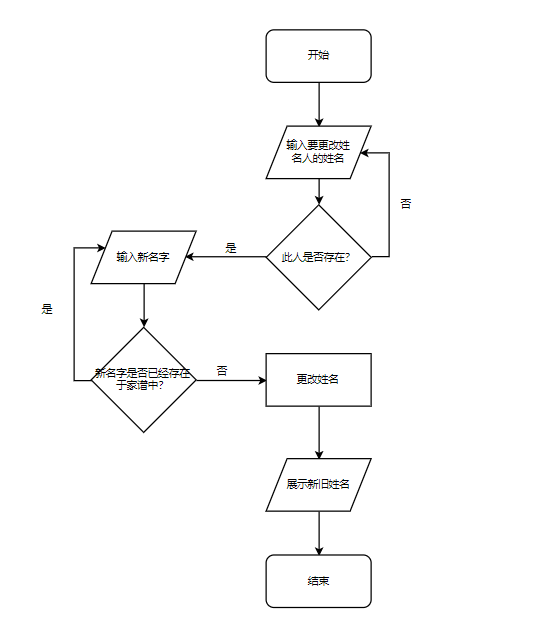
1. MultiNode<string>\* Genealogy::AddNewSon(MultiNode<string>\* N, **const** string & E)
2. {
3. **if** (N == nullptr) **return** nullptr;
4. MultiNode<string>\* NewNode = **new** MultiNode<string>(E);
5. FamilyMap.Insert(E, NewNode);
6. MultiNode<string>\* Last = N->GetLastSon();
7. **if** (Last == nullptr) N->FirstSon = NewNode;
8. **else** Last->NextBro = NewNode;
9. NewNode->Father = N;
10. **return** NewNode;
11. }

### 3.4.3 添加子女功能截屏示例



## 3.5 更改姓名功能的实现

### 3.5.1 更改姓名功能流程图

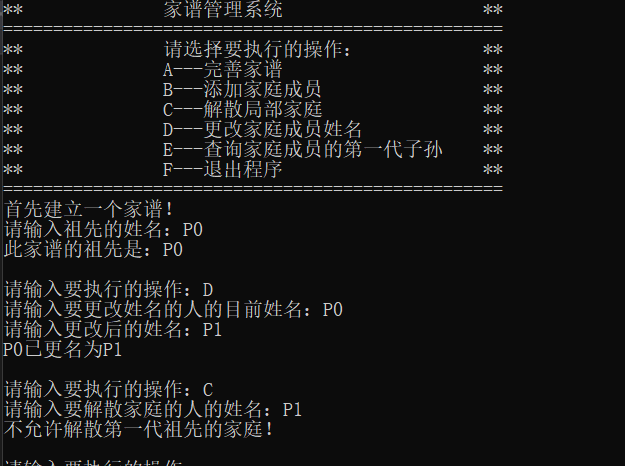


### 3.5.2 更改姓名核心代码

**用户界面接口：**

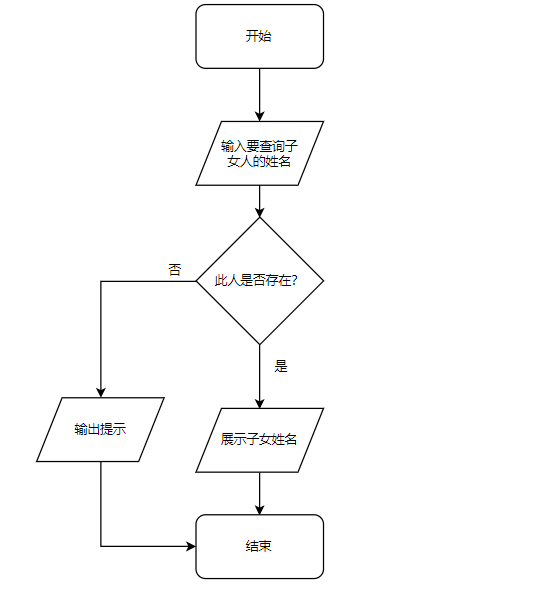
1. **void** Genealogy::Change()
2. {
3. string OldName,NewName;
4. TreeNode<string, MultiNode<string>\*>\* Ancestor;
5. cout << "请输入要更改姓名的人的目前姓名：";
6. cin >> OldName;
7. **while** ((Ancestor = FamilyMap.Find(OldName)) == nullptr)
8. {
9. cout << "此人不存在！" << endl;
10. cout << "请重新输入：";
11. cin >> OldName;
12. }
13. MultiNode<string>\* AncestorNode = Ancestor->ElementValue;
14. FamilyMap.Delete(OldName);
15. cout << "请输入更改后的姓名：";
16. cin >> NewName;
17. **while** ( FamilyMap.Find(NewName) != nullptr)
18. {
19. cout << "此姓名已存在于家谱中！" << endl;
20. cout << "请重新输入：";
21. cin >> NewName;
22. }
23. FamilyMap.Insert(NewName, AncestorNode);
24. AncestorNode->Element = NewName;
25. cout << OldName << "已更名为" << NewName << endl;
26. }

### 3.5.3 更改姓名功能截屏示例



## 3.6 查询功能的实现

### 3.6.1 查询功能流程图

  
3.6.2 查询功能核心代码

**void** Genealogy::Query()

{

    string s;

    TreeNode<string, MultiNode<string>\*>\* Ancestor;

    cout << "请输入要查询第一代子孙的人的姓名：";

    cin >> s;

**if**((Ancestor = FamilyMap.Find(s)) == nullptr)

    {

        cout << "此人不存在！" << endl;

**return**;

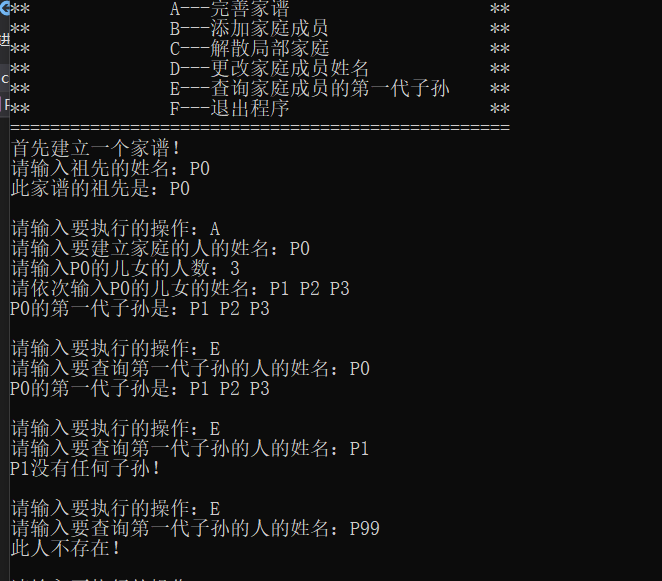
    }

    MultiNode<string>\* AncestorNode = Ancestor->ElementValue;

    Display(AncestorNode);

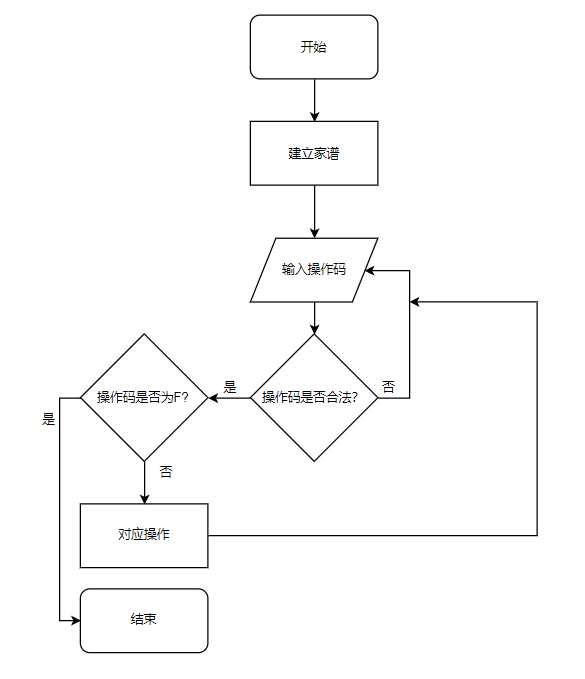
}

### 3.6.3 查询功能截屏示例



## 3.6 总体系统的实现

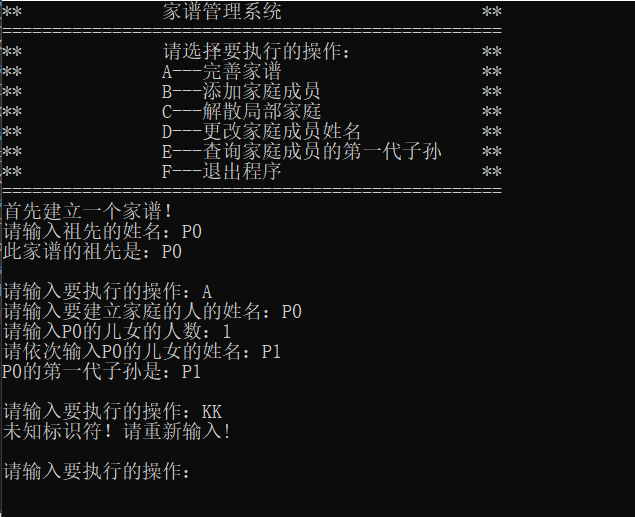
### 3.6.1 总体系统流程图



### 3.6.2 总体系统核心代码

1. **int** main(**void**)
2. {
3. Genealogy MyGenealogy;
4. MyGenealogy.Menu();
5. MyGenealogy.Init();
6. string Cmd;
7. **bool** IsLooping = **true**;
8. **while** (IsLooping)
9. {
10. cout << endl << "请输入要执行的操作：";
11. cin >> Cmd;
12. **switch** (Cmd[0])
13. {
14. **case** 'A':
15. MyGenealogy.Build();
16. **break**;
17. **case** 'B':
18. MyGenealogy.Add();
19. **break**;
20. **case** 'C':
21. MyGenealogy.Dissolve();
22. **break**;
23. **case** 'D':
24. MyGenealogy.Change();
25. **break**;
26. **case** 'E':
27. MyGenealogy.Query();
28. **break**;
29. **case** 'F':
30. IsLooping = **false**;
31. **break**;
32. **default**:
33. cout << "未知标识符！请重新输入!" << endl;
35. }
36. }
37. }

### 3.6.3 总体系统截屏示例



# 4 测试

## 4.1 功能测试

### 4.1.1 建立家庭功能测试

**测试用例**：

A

P0

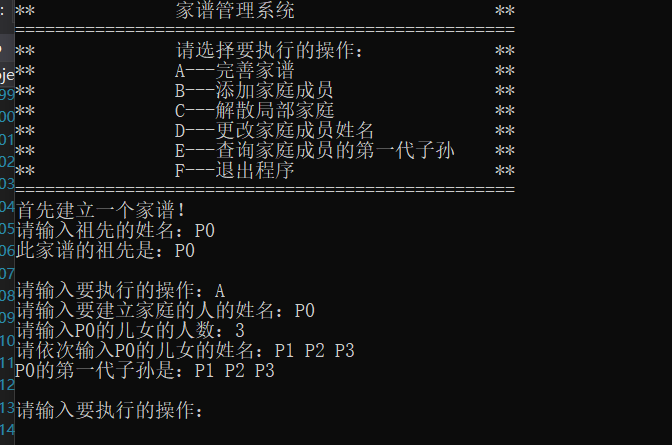
3

P1 P2 P3

**预期结果**：

程序运行正常，输出对应子孙:P1 P2 P3

**实验结果：**



**测试用例**：

A

P0

3

P1 P2 P3

A

P1

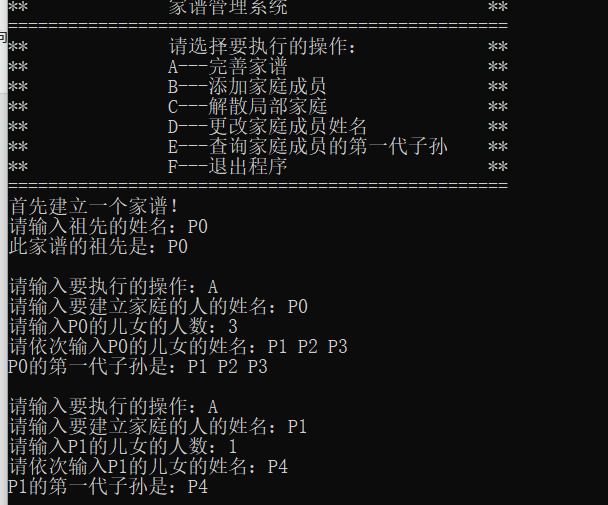
1

P4

**预期结果**：

程序运行正常，输出对应子孙:P4

**实验结果：**



### 4.1.2 添加子女功能测试

**测试用例：**

A

P0

3

P1 P2 P3

B

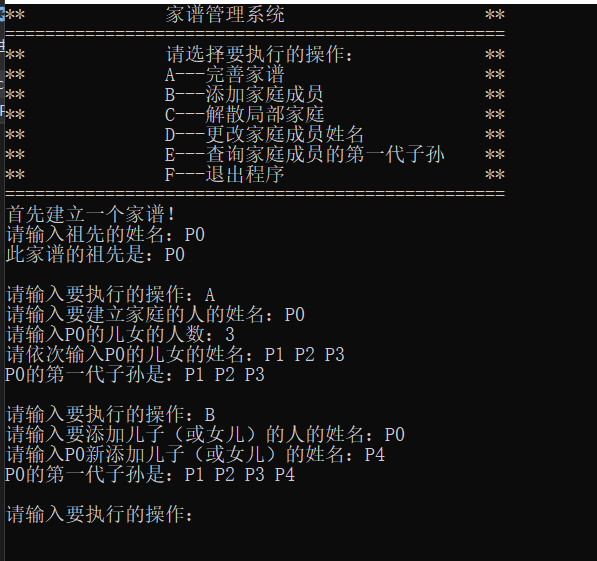
P0

P4

**预期结果：**

程序运行正常，输出对应子孙:P1 P2 P3 P4

**实验结果：**



### 4.1.3 解散家庭功能测试

**测试用例：**

A

P0

3

P1 P2 P3

A

P1

1

P4

C

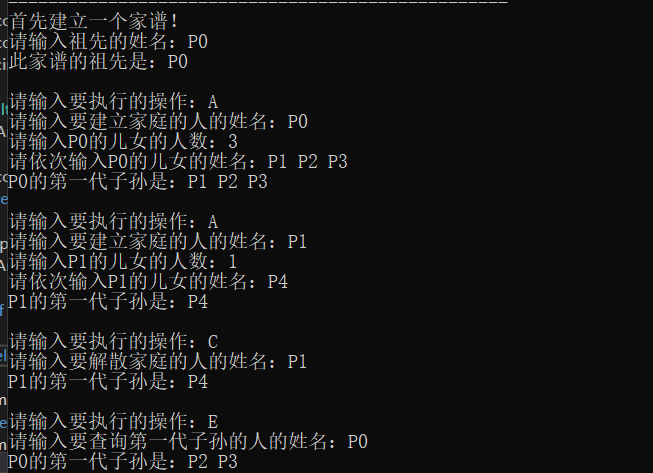
P1

**预期结果：**

程序运行正常，输出对应子孙:P4

P0子孙仅剩P2 P3

**实验结果：**



### 4.1.4 修改姓名功能测试

**测试用例：**

A

P0

3

P1 P2 P3

D

P1

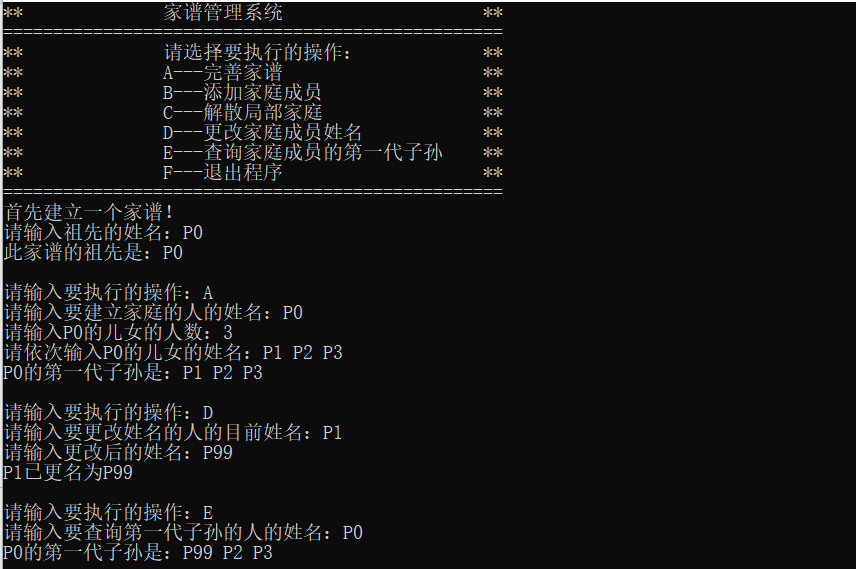
P99

**预期结果：**

程序运行正常，

P0子孙变为P99 P2 P3

**实验结果：**



### 4.1.5 查询功能功能测试

**测试用例：**

A

P0

3

P1 P2 P3

E

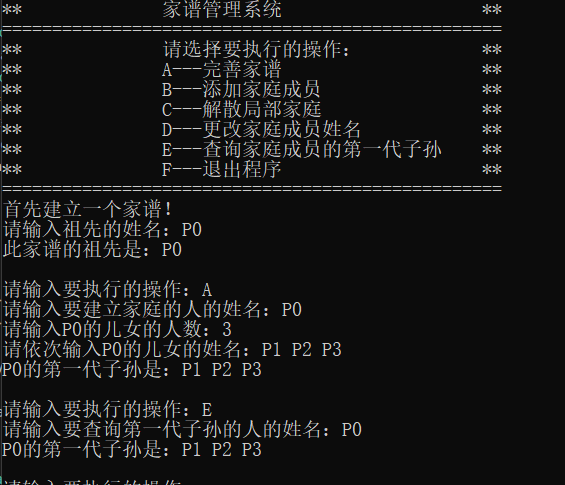
P0

**预期结果：**

程序运行正常，

P0子孙为P1 P2 P3

**实验结果：**



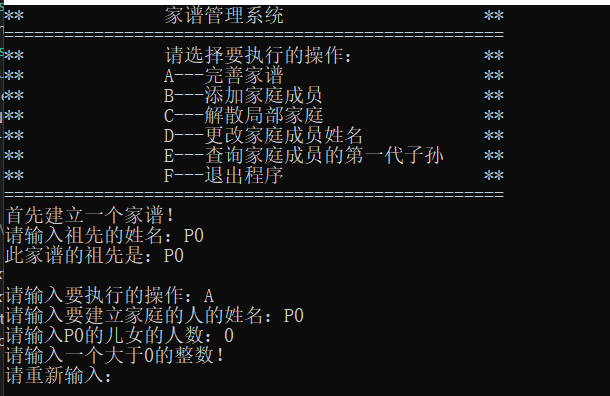
## 4.2 边界测试

### 4.2.1 建立家庭时无输入

**测试用例：**建立家庭时无输入

**预期结果：**给出错误提示，程序运行正常不崩溃。

**实验结果：**

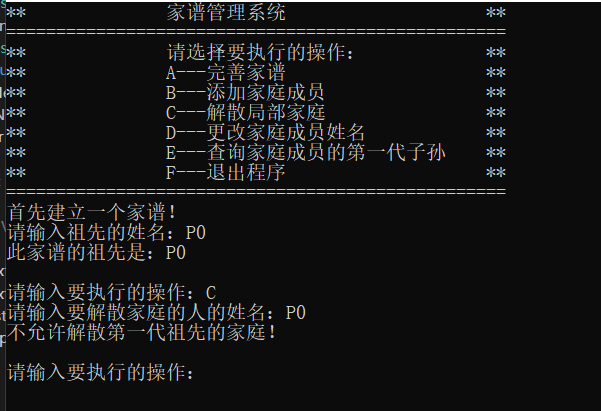


### 4.2.2 解散祖先家庭

**测试用例：**解散祖先家庭

**预期结果：**程序正常运行，不崩溃，给出提示信息。

**实验结果：**

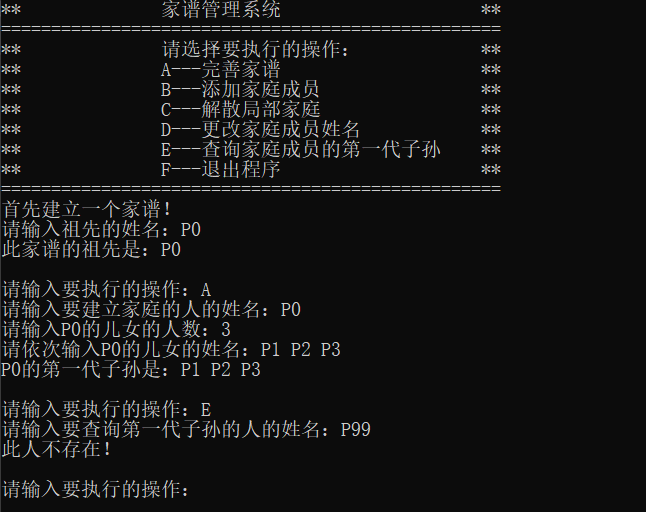


### 4.2.3 查询的人不存在于家谱中

**测试用例：**查询的人不存在于家谱中

**预期结果：**程序给出相应提示，程序正常运行。

**实验结果：**

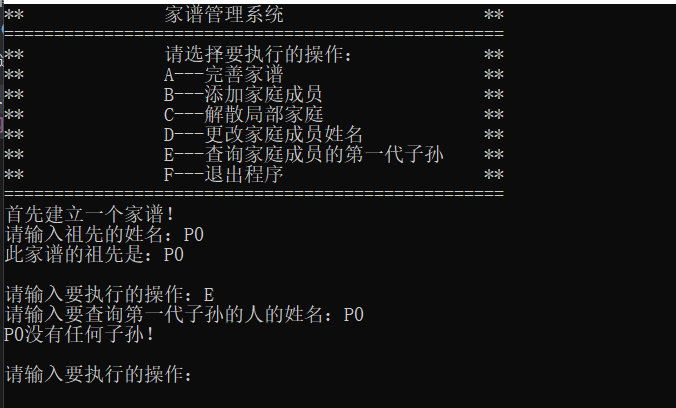


### 4.2.4 查询的人没有任何子女

**测试用例：**查询的人没有任何子女

**预期结果：**程序给出正确信息，程序正常运行。

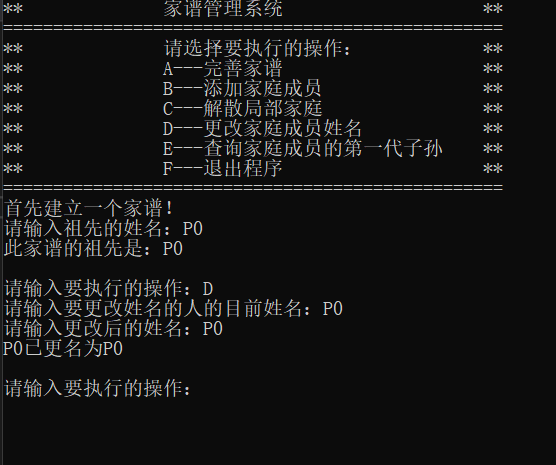
**实验结果：**



### 4.2.5 更改姓名前后姓名一样

**测试用例：**更改姓名前后姓名一样

**预期结果：**程序给出正确信息，程序正常运行。

**实验结果：**

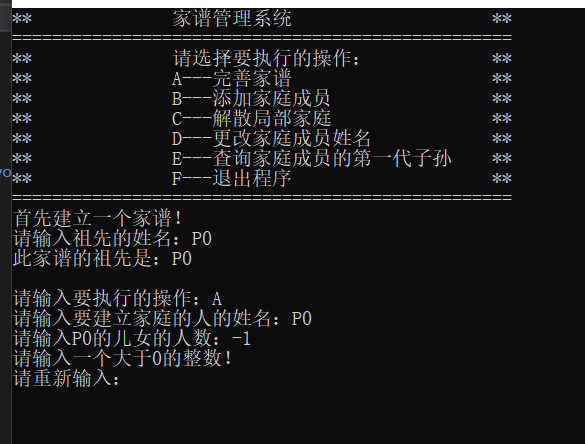
## 4.3 出错测试

### 4.3.1 子女人数错误

**测试用例：**建立家庭时输入子女数为负数

**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：**

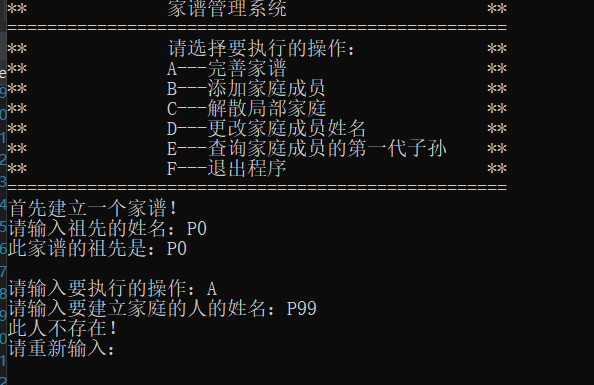


### 4.3.2 建立家庭时输入姓名不存在

**测试用例：**建立家庭时输入姓名不存在

**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：**

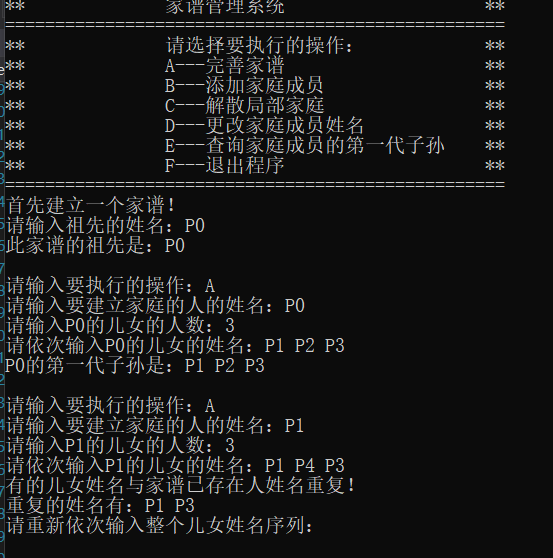


### 4.3.3 建立家庭时子女姓名于已有家庭成员重复

**测试用例：**建立家庭时子女姓名于已有家庭成员重复

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

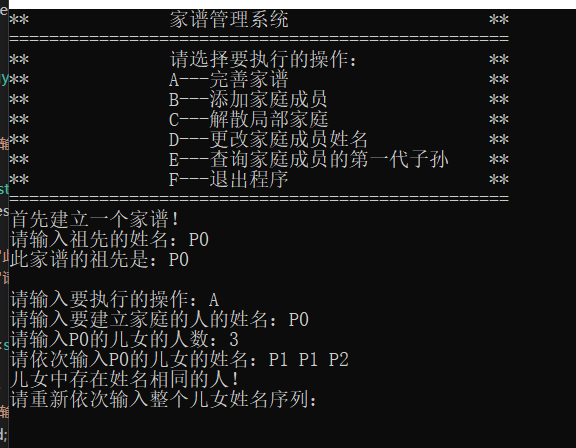


### 4.3.4 建立家庭时子女姓名有重复项

**测试用例：**建立家庭时子女姓名有重复项

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

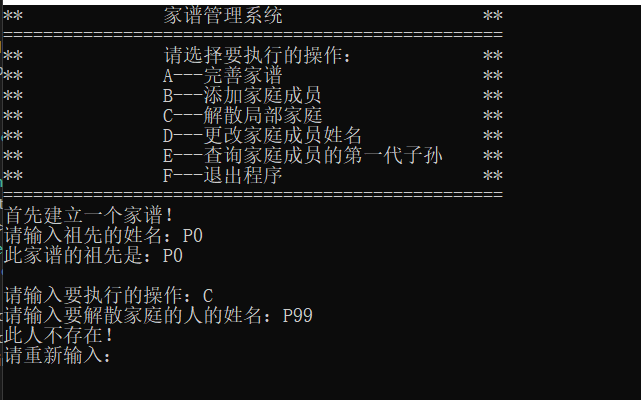


### 4.3.5 解散家庭的人不存在

**测试用例：**解散家庭的人不存在

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

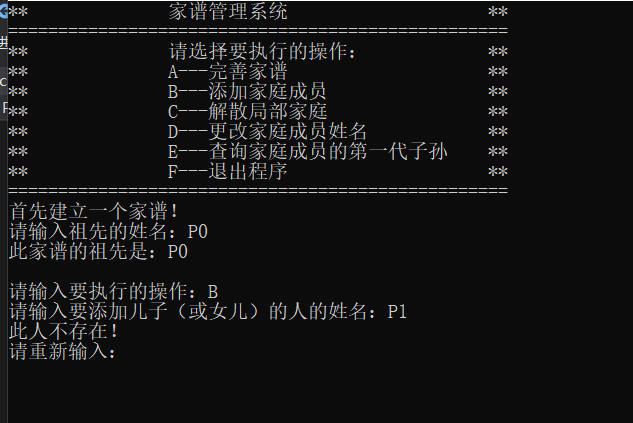


### 4.3.6 添加子女的人不存在

**测试用例：**添加子女的人不存在

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

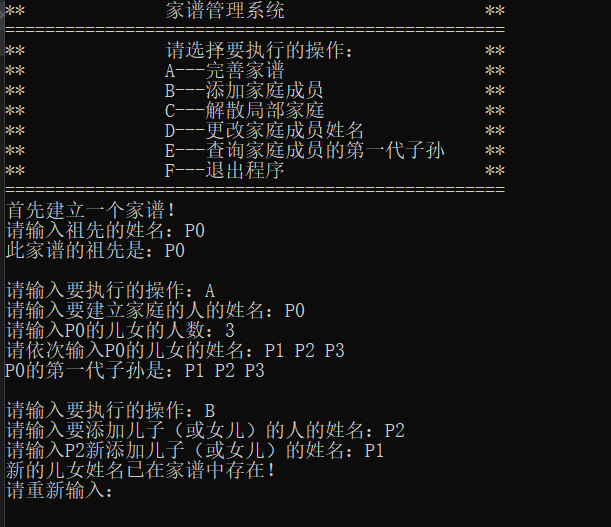


### 4.3.7 添加的子女姓名与原家谱内成员重复

**测试用例：**添加的子女姓名与原家谱内成员重复

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

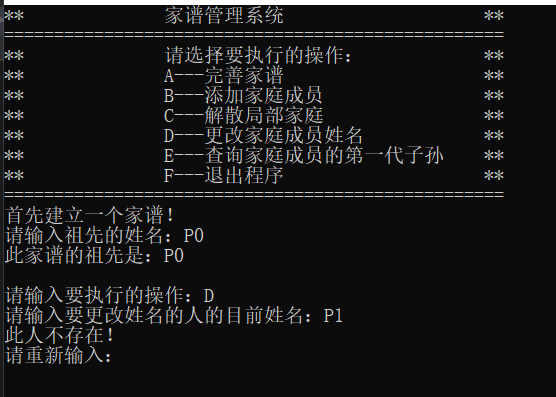


### 4.3.8 更改姓名的人不存在

**测试用例：**更改姓名的人不存在

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**



### 4.3.9 更改姓名的新姓名与原家谱中成员重复

**测试用例：**更改姓名的新姓名与原家谱中成员重复

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**