**项目说明文档**

**数据结构课程设计**

**——家谱管理系统**

作 者 姓 名： 翟晨昊

学 号： 1952216

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 -](#_Toc59393543)

[1.1 背景分析 - 4 -](#_Toc59393544)

[1.2 功能分析 - 4 -](#_Toc59393545)

[2 设计 - 5 -](#_Toc59393546)

[2.1 数据结构设计 - 5 -](#_Toc59393547)

[2.2 类结构设计 - 5 -](#_Toc59393548)

[2.3 成员与操作设计 - 5 -](#_Toc59393549)

[2.4 系统设计 - 14 -](#_Toc59393550)

[3 实现 - 15 -](#_Toc59393551)

[3.1 初始化家谱功能的实现 - 15 -](#_Toc59393552)

[3.1.1 初始化家谱功能流程图 - 15 -](#_Toc59393553)

[3.1.2 初始化家谱功能核心代码 - 16 -](#_Toc59393554)

[3.1.3 初始化家谱功能截屏示例 - 17 -](#_Toc59393555)

[3.2 完善家谱功能的实现 - 18 -](#_Toc59393556)

[3.2.1 完善家谱功能流程图 - 18 -](#_Toc59393557)

[3.2.2 完善家谱功能核心代码 - 19 -](#_Toc59393558)

[3.2.3 完善家谱功能截屏示例 - 22 -](#_Toc59393559)

[3.3 添加家庭成员功能的实现 - 23 -](#_Toc59393560)

[3.3.1 添加家庭成员功能流程图 - 23 -](#_Toc59393561)

[3.3.2 添加家庭成员功能核心代码 - 24 -](#_Toc59393562)

[3.3.3 添加家庭成员功能截屏示例 - 26 -](#_Toc59393563)

[3.4 解散局部家庭功能的实现 - 27 -](#_Toc59393564)

[3.4.1 解散局部家庭功能流程图 - 27 -](#_Toc59393565)

[3.4.2 解散局部家庭功能核心代码 - 28 -](#_Toc59393566)

[3.4.3 解散局部家庭功能截屏示例 - 30 -](#_Toc59393567)

[3.5 更改成员姓名功能的实现 - 31 -](#_Toc59393568)

[3.5.1 更改成员姓名功能流程图 - 31 -](#_Toc59393569)

[3.5.2 更改成员姓名功能核心代码 - 32 -](#_Toc59393570)

[3.5.3 更改成员姓名功能截屏示例 - 33 -](#_Toc59393571)

[3.6 总体功能的实现 - 34 -](#_Toc59393572)

[3.6.1 总体功能流程图 - 34 -](#_Toc59393573)

[3.6.2 总体功能核心代码 - 35 -](#_Toc59393574)

[3.6.3 总体功能截屏示例 - 36 -](#_Toc59393575)

[4 测试 - 37 -](#_Toc59393576)

[4.1 功能测试 - 37 -](#_Toc59393577)

[4.1.1 初始化家谱功能测试 - 37 -](#_Toc59393578)

[4.1.2 完善家谱功能测试 - 38 -](#_Toc59393579)

[4.1.3 添加家庭成员功能测试 - 39 -](#_Toc59393580)

[4.1.4 解散局部家庭功能测试 - 40 -](#_Toc59393581)

[4.1.5 更改成员姓名功能测试 - 41 -](#_Toc59393582)

[4.2 边界测试 - 42 -](#_Toc59393583)

[4.2.1 完善家谱时输入添加儿女个数为零 - 42 -](#_Toc59393584)

[4.2.2 解散局部家庭时解散第一代祖先家庭 - 43 -](#_Toc59393585)

[4.2.3 家谱中只有祖先时进行解散家庭操作 - 44 -](#_Toc59393586)

[4.2.4 对已经建立家庭的人再次建立家庭 - 45 -](#_Toc59393587)

[4.3 出错测试 - 46 -](#_Toc59393588)

[4.3.1 完善家谱时输入的姓名不存在 - 46 -](#_Toc59393589)

[4.3.2 完善家谱时新添加儿女个数不合法 - 47 -](#_Toc59393590)

[4.3.3 完善家谱时新添加儿女姓名在家谱中已存在 - 48 -](#_Toc59393591)

[4.3.4 完善家谱时新添加儿女姓名存在重复 - 49 -](#_Toc59393592)

[4.3.5 完善家谱时新添加儿女姓名过多 - 50 -](#_Toc59393593)

[4.3.6 添加家庭成员时输入的姓名不存在 - 51 -](#_Toc59393594)

[4.3.7 添加家庭成员时新添加儿女姓名在家谱中已存在 - 52 -](#_Toc59393595)

[4.3.8 添加家庭成员时新添加儿女姓名输入多个 - 53 -](#_Toc59393596)

[4.3.9 解散局部家庭时输入的姓名不存在 - 54 -](#_Toc59393597)

[4.3.10 更改成员姓名时输入的原姓名不存在 - 55 -](#_Toc59393598)

[4.3.11 更改成员姓名时输入的新姓名在家谱中已存在 - 56 -](#_Toc59393599)

[4.3.12 输入操作数不合法 - 57 -](#_Toc59393600)

# 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，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，而多叉树在查找方面性能不是很出色，为了能更快速的进行查找，系统中还增加了一棵平衡二叉搜索树——AVL树，其中保存了多叉树的每一个结点，当系统需要找到某一结点时，通过AVL树来进行快速搜索，搜索到后，AVL树结点中保存的多叉树结点就是要找的目标结点。

## 2.2 类结构设计

首先，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（FamilyTree类）和一个AVL树（AVLTree类），以及它们的结点类：家族树结点类（FamilyTreeNode类）与AVL树结点类（AVLTreeNode类）。将两棵树都设置为对应结点类的友元，使得树可以访问结点类。同时，由于添加子女的时候可能会一次添加多个，但是添加的时候需要将它们输入完，比对是否有重名后再保存进家谱，因此设计了Vector向量类来保存输入数据。除此之外，本系统还设置了家谱类（Genealogy类），给用户提供完善家谱，添加家庭成员，解散家庭，更改成员姓名等功能的接口。为了使Vector类与树更具有泛用性，本系统将Vector类，FamilyTree类，AVLTree类以及它们的结点类都设计为了模板类。

## 2.3 成员与操作设计

**向量类（Vector）：**

template <typename ElementType> class Vector {

public:

    ~Vector();

    Vector();

    ElementType& operator[](const int x);

    const ElementType& operator[](const int x)const;

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

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

    bool isFull();

    bool isEmpty();

    void pushBack(const ElementType& temp);

    void popBack();

    void clear();

    int getSize();

    void reSize(int newSize);

private:

    void extendSize();

    int size;

    int maxSize;

    ElementType\* myVector;

};

**私有成员：**

int size;//Vector中已经存储的元素数量

int maxSize;//Vector中已经申请的空间大小，元素数量如果超过要再次申请

ElementType\* myVector;//存储的元素序列的起始地址

**私有操作：**

void extendSize();

//在Vector申请的空间已经被占满时再次申请空间，每次扩充至maxsize\*2+1是因为刚开始的maxsize为0

**公有操作：**

Vector();

//构造函数，初始化指针并将size与maxSize置为0

~Vector();

//析构函数，调用clear()函数来删除元素，释放内存

ElementType& operator[](const int x);

//重载[]运算符，返回ElementType&类型

const ElementType& operator[](const int x)const;

//重载[]运算符，返回const ElementType&类型

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

//复制构造函数，将一个Vector复制给另一个Vector

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

//重载=运算符，可以将一个Vector赋给另一个Vector

bool isFull();

//判断是否Vector中申请的内存已经被占满

bool isEmpty();

//判断Vector是否为空

void pushBack(const ElementType& temp);

//在Vector末尾添加一个元素

void popBack();

//删除Vector最末尾的元素

void clear();

//删除Vector中的所有元素并释放内存

int getSize();

//返回Vector已经存储的元素数量

void reSize(int newSize);

//重设Vector的大小，若比之前小，则会抛弃多余的元素

**AVL树结点类（AVLTreeNode）：**

template <typename Key, typename ElementType>

class AVLTreeNode {

public:

    friend class AVLTree<Key, ElementType>;

    AVLTreeNode() = default;

    AVLTreeNode(Key inputKey, ElementType inputData) :

        dataKey(inputKey), data(inputData) {}

    ~AVLTreeNode() = default;

private:

    Key dataKey;

    ElementType data;

    int height = 0;

    AVLTreeNode<Key, ElementType>\* left = nullptr;

    AVLTreeNode<Key, ElementType>\* right = nullptr;

};

**私有成员：**

Key dataKey;//AVL树结点的关键码，作为搜索依据

ElementType data;//AVL树结点中保存的数据信息

int height;//AVL树结点当前所处的高度，用于保持平衡

AVLTreeNode<Key, ElementType>\* left;//指针域，指向当前结点的左子女

AVLTreeNode<Key, ElementType>\* right;//指针域，指向当前结点的右子女

**公有操作：**

friend class AVLTree<Key, ElementType>;

//将AVLTree声明为友元

AVLTreeNode() = default;

//默认构造函数

AVLTreeNode(Key inputKey, ElementType inputData);

//含参构造函数

~AVLTreeNode() = default;

//默认析构函数

**AVL树类（AVLTree）：**

template <typename Key, typename ElementType>

class AVLTree {

public:

    friend class Genealogy;

    AVLTree() :root(nullptr), size(0) {}

    ~AVLTree();

    ElementType getData(AVLTreeNode<Key, ElementType>\* ptr);

    void makeEmpty();

    AVLTreeNode<Key, ElementType>\* find(Key findK);

    AVLTreeNode<Key, ElementType>\* insert(Key insertK, ElementType insertData);

    bool remove(Key removeK);

    void change(Key changeOldK, Key changeNewK);

private:

int  max(int x, int y);

    int  getHeight(AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* findPriorNode(AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* findNextNode(AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* rotateL(AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* rotateR(AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* rotateLR(AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* rotateRL(AVLTreeNode<Key, ElementType>\* ptr);

    void makeEmpty(AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* find(Key findK, AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* insert(Key insertK, ElementType insertData, AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* remove(Key removeK, AVLTreeNode<Key, ElementType>\* ptr);

    AVLTreeNode<Key, ElementType>\* root;

    int size;

};

**私有成员：**

int size;//AVL树中的结点个数

AVLTreeNode<Key, ElementType>\* root;//AVL树的根节点

**私有操作：**

int max(int x, int y);

//返回x，y中的最大值，用来修改结点的高度

int getHeight(AVLTreeNode<Key, ElementType>\* ptr);

//返回结点ptr的高度

AVLTreeNode<Key, ElementType>\* findPriorNode(AVLTreeNode<Key, ElementType>\* ptr);

//找到在中序遍历下ptr前面的结点

AVLTreeNode<Key, ElementType>\* findNextNode(AVLTreeNode<Key, ElementType>\* ptr);

//找到在中序遍历下ptr后面的结点

AVLTreeNode<Key, ElementType>\* rotateL(AVLTreeNode<Key, ElementType>\* ptr);

//左单旋转

AVLTreeNode<Key, ElementType>\* rotateR(AVLTreeNode<Key, ElementType>\* ptr);

//右单旋转

AVLTreeNode<Key, ElementType>\* rotateLR(AVLTreeNode<Key, ElementType>\* ptr);

//先左后右双旋转

AVLTreeNode<Key, ElementType>\* rotateRL(AVLTreeNode<Key, ElementType>\* ptr);

//先右后左双旋转

void makeEmpty(AVLTreeNode<Key, ElementType>\* ptr);

//将ptr结点及ptr结点的所有子女结点全部删除并释放内存

AVLTreeNode<Key, ElementType>\* find(Key findK, AVLTreeNode<Key, ElementType>\* ptr);

//找到以结点ptr为根节点的子树中关键码为findK的结点

AVLTreeNode<Key, ElementType>\* insert(Key insertK, ElementType insertData, AVLTreeNode<Key, ElementType>\* ptr);

//将关键码为insertK，数据信息为insertData的结点插入以结点ptr为根节点的子树中

AVLTreeNode<Key, ElementType>\* remove(Key removeK, AVLTreeNode<Key, ElementType>\* ptr);

//将以结点ptr为根节点的子树中关键码为removeK的结点删除

**公有操作：**

friend class Genealogy;

//将Genealogy声明为友元

AVLTree();

//无参构造函数

~AVLTree();

//析构函数，通过调用makeEmpty()函数来删除元素，释放内存

ElementType getData(AVLTreeNode<Key, ElementType>\* ptr);

//获得结点ptr中保存的数据信息

void makeEmpty();

//将AVL树中的所有结点删除并释放内存

AVLTreeNode<Key, ElementType>\* find(Key findK);

//找到AVL树中关键码为findK的结点

AVLTreeNode<Key, ElementType>\* insert(Key insertK, ElementType insertData);

//将关键码为insertK，数据信息为insertData的结点插入AVL树中

bool remove(Key removeK);

//将AVL树中关键码为removeK的结点删除

void change(Key changeOldK, Key changeNewK);

//将AVL树中关键码为changeOldK的结点的关键码改为changeNewK

**家族树结点类（FamilyTreeNode）**：

template<typename ElementType>

class FamilyTreeNode {

public:

    friend class Genealogy;

    friend class FamilyTree<ElementType>;

    FamilyTreeNode() = default;

    FamilyTreeNode(const ElementType& inputData) :data(inputData) {}

    ~FamilyTreeNode() = default;

private:

    ElementType data;

    FamilyTreeNode<ElementType>\* firstChild = nullptr;

    FamilyTreeNode<ElementType>\* nextSibling = nullptr;

    FamilyTreeNode<ElementType>\* parent = nullptr;

};

**私有成员：**

ElementType data;//结点中保存的成员信息

FamilyTreeNode<ElementType>\* firstChild;//该结点的首子女

FamilyTreeNode<ElementType>\* nextSibling;//该结点的下一个兄弟FamilyTreeNode<ElementType>\* parent;//该结点的父亲

**公有操作：**

friend class Genealogy;

//将Genealogy声明为友元

friend class FamilyTree<ElementType>;

//将FamilyTree声明为友元

FamilyTreeNode() = default;

//默认构造函数

FamilyTreeNode(const ElementType& inputData);

//含参构造函数

~FamilyTreeNode() = default;

//默认析构函数

**家族树类（FamilyTree）**：

template<typename ElementType>

class FamilyTree {

public:

    friend class Genealogy;

    FamilyTree() :root(nullptr) {}

    ~FamilyTree();

    FamilyTreeNode<ElementType>\* getLastChild(FamilyTreeNode<ElementType>\* ptr);

    void makeEmpty(FamilyTreeNode<ElementType>\* ptr);

    void clear(Vector<ElementType>& vec, FamilyTreeNode<ElementType>\* ptr);

private:

    FamilyTreeNode<ElementType>\* root;

};

**私有成员：**

FamilyTreeNode<ElementType>\* root;//家族树的根结点

**公有操作：**

friend class Genealogy;

//将Genealogy声明为友元

FamilyTree();

//无参构造函数

~FamilyTree();

//析构函数，通过调用makeEmpty()函数来删除元素，释放内存

FamilyTreeNode<ElementType>\* getLastChild (FamilyTreeNode <ElementType>\* ptr);

//返回结点ptr的最后一个子女

void makeEmpty(FamilyTreeNode<ElementType>\* ptr);

//将ptr结点及ptr结点的所有子女结点全部删除并释放内存

void clear(Vector<ElementType>& vec, FamilyTreeNode<ElementType>\* ptr);

//将ptr结点及ptr结点的所有子女结点全部删除并释放内存，同时用vec保存所有被删除结点中的成员信息

**家谱类（Genealogy）**：

class Genealogy

{

public:

    void menu();

    void init();

    void build();

    void add();

    void destroy();

    void change();

    void show(FamilyTreeNode<string>\* parent);

private:

    bool checkChild(Vector<string>& child);

    bool checkChild(string& child);

    void addNewChild(FamilyTreeNode<string>\* parent, string& child);

    void addNewChild(FamilyTreeNode<string>\* parent, Vector<string>& child);

    FamilyTree<string> family;

    AVLTree<string, FamilyTreeNode<string>\*> familyAVLTree;

};

**私有成员：**

FamilyTree<string> family;//家族树，用来保存家族成员的信息与相互关系

AVLTree<string, FamilyTreeNode<string>\*> familyAVLTree;

//AVL树，用来通过姓名快速查找家族树中该成员所在的结点

**私有操作：**

bool checkChild(Vector<string>& child);

//检查输入的子女姓名是否重复或者已经在家谱中存在

bool checkChild(string& child);

//检查输入的子女姓名是否已经在家谱中存在

void addNewChild(FamilyTreeNode<string>\* parent, string& child);

//向结点parent添加一个子女

void addNewChild(FamilyTreeNode<string>\* parent, Vector<string>& child);

//向结点parent添加多个子女

**公有操作：**

void menu();

//用户操作菜单

void init();

//初始化家谱

void build();

//建立家庭

void add();

//添加子女

void destroy();

//解散家庭

void change();

//改变成员姓名

void show(FamilyTreeNode<string>\* parent);

//展示结点parent的子女

## 2.4 系统设计

系统在使用时，首先要输入祖先姓名来初始化家谱，随后就可以根据菜单，输入对应的操作码来执行相应的操作。功能有完善家谱，添加家庭成员，解散家庭，更改成员姓名，退出系统等等。

# 3 实现

## 3.1 初始化家谱功能的实现

### 3.1.1 初始化家谱功能流程图



### 3.1.2 初始化家谱功能核心代码

**Genealogy类中：**

void Genealogy::init()

{

    cout << "首先建立一个家谱！" << endl;

    cout << "请输入祖先的姓名：";

    string ancestor;

    cin >> ancestor;

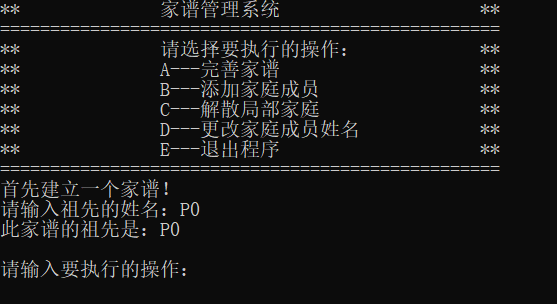
    family.root = new FamilyTreeNode<string>(ancestor);

    familyAVLTree.insert(ancestor, family.root);

    cout << "此家谱的祖先是：" << ancestor << endl;

}

### 3.1.3 初始化家谱功能截屏示例



## 3.2 完善家谱功能的实现

### 3.2.1 完善家谱功能流程图



### 3.2.2 完善家谱功能核心代码

**Genealogy类中：**

void Genealogy::build()

{

    int childNum = 0;

    string parentName;

    Vector<string> child;

    AVLTreeNode<string, FamilyTreeNode<string>\*>\* parent = nullptr;

    cout << "请输入要建立家庭的人的姓名：";

    cin >> parentName;

    parent = familyAVLTree.find(parentName);

    while (parent == nullptr || familyAVLTree.getData(parent)->firstChild != nullptr)

    {

        if (parent == nullptr)

        {

            cout << "此人不在家谱中！" << endl;

        }

        else

        {

            cout << "此人已经建立家庭！" << endl;

        }

        cout << "请重新输入：";

        cin >> parentName;

        parent = familyAVLTree.find(parentName);

    }

    cout << "请输入" << parentName << "的儿女的人数：";

    cin >> childNum;

    while (cin.fail() || childNum <= 0)

    {

        cout << "请输入一个正整数！" << endl;

        cout << "请重新输入：";

        cin.clear();

        cin.ignore(numeric\_limits<streamsize>::max(), '\n');

        cin >> childNum;

    }

    cin.clear();

    cin.ignore(numeric\_limits<streamsize>::max(), '\n');

    cout << "请依次输入" << parentName << "的儿女的姓名：";

    child.reSize(childNum);

    for (int i = 0; i < childNum; i++)

    {

        cin >> child[i];

    }

    while (!checkChild(child))

    {

        cin.clear();

        cin.ignore(numeric\_limits<streamsize>::max(), '\n');

        cout << "请重新依次输入" << parentName << "的儿女的姓名：";

        for (int i = 0; i < childNum; i++)

        {

            cin >> child[i];

        }

    }

    FamilyTreeNode<string>\* pParent = familyAVLTree.getData(parent);

    addNewChild(pParent, child);

    show(pParent);

}

bool Genealogy::checkChild(Vector<string>& child)

{

    for (int i = 0; i < child.getSize() - 1; i++)

    {

        for (int j = i + 1; j < child.getSize(); j++)

        {

            if (child[i] == child[j])

            {

                cout << "输入的儿女中存在姓名相同的人！" << endl;

                return false;

            }

        }

    }

    for (int i = 0; i < child.getSize(); i++)

    {

        if (familyAVLTree.find(child[i]) != nullptr)

        {

            cout << " 存在儿女姓名与家谱中已存在人姓名相同！" << endl;

            return false;

        }

    }

    return true;

}

void Genealogy::addNewChild(FamilyTreeNode<string>\* parent, Vector<string>& child)

{

    if (parent == nullptr || child.isEmpty())

    {

        return;

    }

    FamilyTreeNode<string>\* pChild = new FamilyTreeNode<string>(child[0]);

    familyAVLTree.insert(child[0], pChild);

    parent->firstChild = pChild;

    pChild->parent = parent;

    for (int i = 1; i < child.getSize(); i++)

    {

        pChild->nextSibling = new FamilyTreeNode<string>(child[i]);

        pChild->nextSibling->parent = parent;

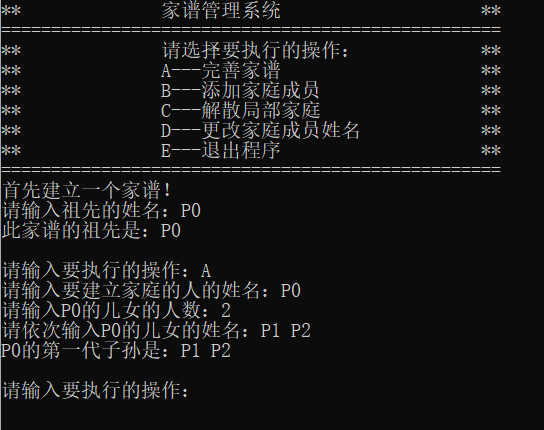
        pChild = pChild->nextSibling;

        familyAVLTree.insert(child[i], pChild);

    }

}

### 3.2.3 完善家谱功能截屏示例



## 3.3 添加家庭成员功能的实现

### 3.3.1 添加家庭成员功能流程图



### 3.3.2 添加家庭成员功能核心代码

**Genealogy类中：**

void Genealogy::add()

{

    string parentName;

    string addChild;

    AVLTreeNode<string, FamilyTreeNode<string>\*>\* parent = nullptr;

    cout << "请输入要添加儿子（或女儿）的人的姓名：";

    cin >> parentName;

    parent = familyAVLTree.find(parentName);

    while (parent == nullptr)

    {

        cout << "此人不在家谱中！" << endl;

        cout << "请重新输入：";

        cin >> parentName;

        parent = familyAVLTree.find(parentName);

    }

    cout << "请输入" << parentName << "新添加儿子（或女儿）的姓名：";

    cin >> addChild;

    while (!checkChild(addChild))

    {

        cout << "请重新输入：";

        cin >> addChild;

    }

    FamilyTreeNode<string>\* pParent = familyAVLTree.getData(parent);

    addNewChild(pParent, addChild);

    show(pParent);

}

bool Genealogy::checkChild(string& child)

{

    if (familyAVLTree.find(child) != nullptr)

    {

        cout << "新的儿女姓名已在家谱中已经存在！" << endl;

        return false;

    }

    return true;

}

void Genealogy::addNewChild(FamilyTreeNode<string>\* parent, string& child)

{

    if (parent == nullptr)

    {

        return;

    }

    FamilyTreeNode<string>\* pChild = new FamilyTreeNode<string>(child);

    familyAVLTree.insert(child, pChild);

    FamilyTreeNode<string>\* lastChild = family.getLastChild(parent);

    if (lastChild == nullptr)

    {

        parent->firstChild = pChild;

    }

    else

    {

        lastChild->nextSibling = pChild;

    }

    pChild->parent = parent;

}

**FamilyTree类中：**

template<typename ElementType>

FamilyTreeNode<ElementType>\* FamilyTree<ElementType>::getLastChild(FamilyTreeNode<ElementType>\* ptr)

{

    FamilyTreeNode<ElementType>\* lastSon = ptr->firstChild;

    if (lastSon == nullptr)

    {

        return nullptr;

    }

    else

    {

        while (lastSon->nextSibling != nullptr)

        {

            lastSon = lastSon->nextSibling;

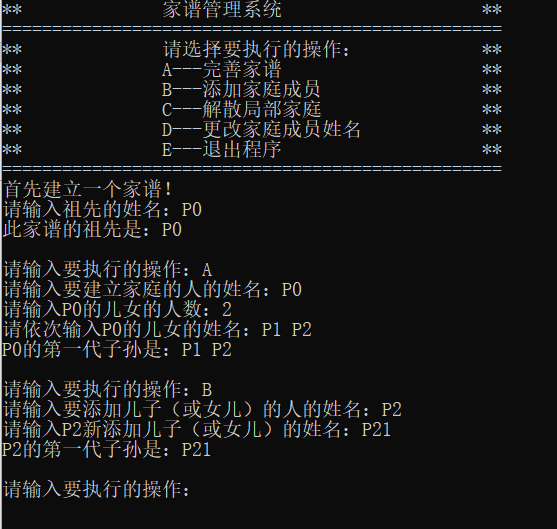
        }

        return lastSon;

    }

}

### 3.3.3 添加家庭成员功能截屏示例



## 3.4 解散局部家庭功能的实现

### 3.4.1 解散局部家庭功能流程图



### 3.4.2 解散局部家庭功能核心代码

**Genealogy类中：**

void Genealogy::destroy()

{

    string ancestorName;

    Vector<string> deleteNode;

AVLTreeNode<string, FamilyTreeNode<string>\*>\* ancestor = nullptr;

if (familyAVLTree.size == 1)

    {

        cout << "家谱中只有第一代祖先！无法解散家庭" << endl;

        return;

    }

    cout << "请输入要解散家庭的人的姓名：";

    cin >> ancestorName;

    ancestor = familyAVLTree.find(ancestorName);

    while (ancestor == nullptr || familyAVLTree.getData(ancestor)->data == family.root->data)

    {

        if (ancestor == nullptr)

        {

            cout << "此人不在家谱中！" << endl;

        }

        else

        {

            cout << "不能解散第一代祖先的家庭！" << endl;

        }

        cout << "请重新输入：";

        cin >> ancestorName;

        ancestor = familyAVLTree.find(ancestorName);

    }

    FamilyTreeNode<string>\* pAncestor = familyAVLTree.getData(ancestor);

    cout << "要解散家庭的人是：" << pAncestor->data << endl;

    show(pAncestor);

    if (pAncestor->parent != nullptr)

    {

        if (pAncestor->parent->firstChild == pAncestor)

        {

            pAncestor->parent->firstChild = pAncestor->nextSibling;

        }

        else

        {

            FamilyTreeNode<string>\* current = pAncestor->parent->firstChild;

            while (current->nextSibling != pAncestor)

            {

                current = current->nextSibling;

            }

            current->nextSibling = pAncestor->nextSibling;

        }

    }

    deleteNode.pushBack(pAncestor->data);

    family.clear(deleteNode, pAncestor->firstChild);

    delete pAncestor;

    pAncestor = nullptr;

    for (int i = 0; i < deleteNode.getSize(); i++)

    {

        familyAVLTree.remove(deleteNode[i]);

    }

}

**FamilyTree类中：**

template<typename ElementType>

void FamilyTree<ElementType>::clear(Vector<ElementType>& vec, FamilyTreeNode<ElementType>\* ptr)

{

    if (ptr == nullptr)

    {

        return;

    }

    clear(vec, ptr->firstChild);

    clear(vec, ptr->nextSibling);

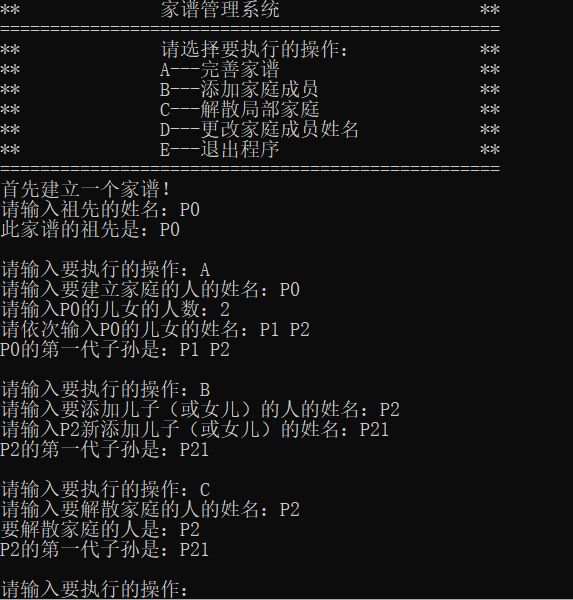
    vec.pushBack(ptr->data);

    delete ptr;

    ptr = nullptr;

}

### 3.4.3 解散局部家庭功能截屏示例



## 3.5 更改成员姓名功能的实现

### 3.5.1 更改成员姓名功能流程图



### 3.5.2 更改成员姓名功能核心代码

**Genealogy类中：**

void Genealogy::change()

{

    string oldName, newName;

    AVLTreeNode<string, FamilyTreeNode<string>\*>\* oldPeople = nullptr;

    AVLTreeNode<string, FamilyTreeNode<string>\*>\* newPeople = nullptr;

    cout << "请输入要更改姓名的人的目前姓名：";

    cin >> oldName;

    oldPeople = familyAVLTree.find(oldName);

    while (oldPeople == nullptr)

    {

        cout << "此人不在家谱中！" << endl;

        cout << "请重新输入：";

        cin >> oldName;

        oldPeople = familyAVLTree.find(oldName);

    }

    FamilyTreeNode<string>\* pPeople = familyAVLTree.getData(oldPeople);

    cout << "请输入更改后的姓名：";

    cin >> newName;

    newPeople = familyAVLTree.find(newName);

    while (newPeople != nullptr)

    {

        cout << "此姓名已存在于家谱中！" << endl;

        cout << "请重新输入：";

        cin >> newName;

        newPeople = familyAVLTree.find(newName);

    }

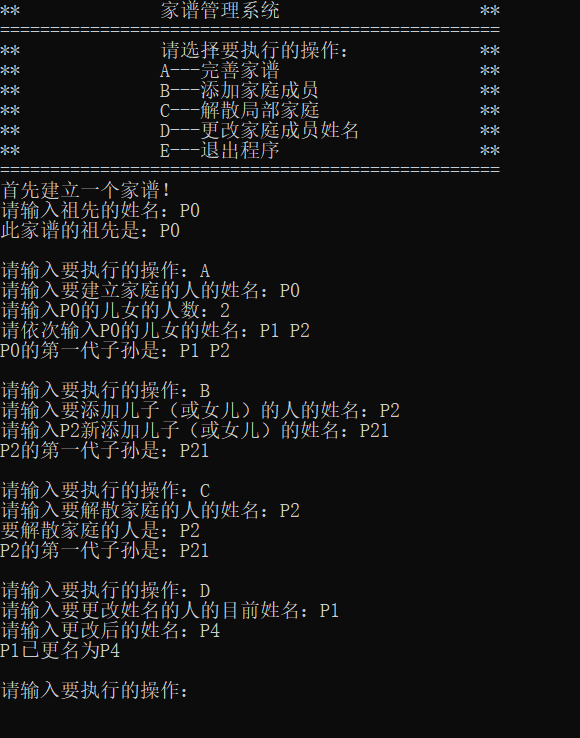
    familyAVLTree.change(oldName, newName);

    pPeople->data = newName;

    cout << oldName << "已更名为" << newName << endl;

}

### 3.5.3 更改成员姓名功能截屏示例



## 3.6 总体功能的实现

### 3.6.1 总体功能流程图



### 3.6.2 总体功能核心代码

int main(void)

{

    Genealogy MyGenealogy;

    MyGenealogy.menu();

    MyGenealogy.init();

    string operation;

    while (true)

    {

        cin.clear();

        cin.ignore(numeric\_limits<streamsize>::max(), '\n');

        cout << endl << "请输入要执行的操作：";

        cin >> operation;

        switch (operation[0])//string类型不能用于switch

        {

        case 'A':

            MyGenealogy.build();

            break;

        case 'B':

            MyGenealogy.add();

            break;

        case 'C':

            MyGenealogy.destroy();

            break;

        case 'D':

            MyGenealogy.change();

            break;

        case 'E':

            cout << "成功退出系统！" << endl;

            return 0;

        default:

            cout << "操作数输入不正确，请重新输入!" << endl;

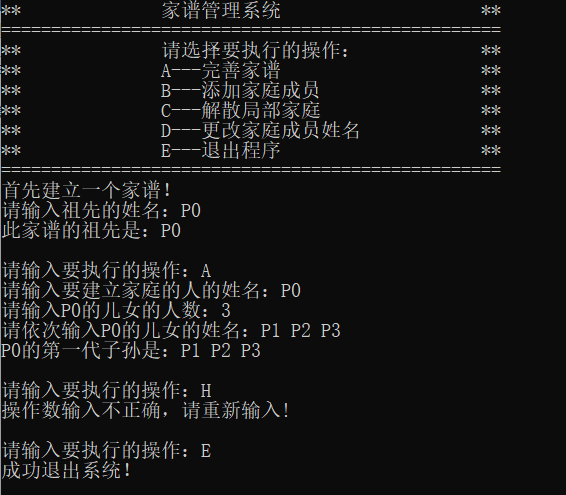
        }

    }

    return 0;

}

### 3.6.3 总体功能截屏示例



# 4 测试

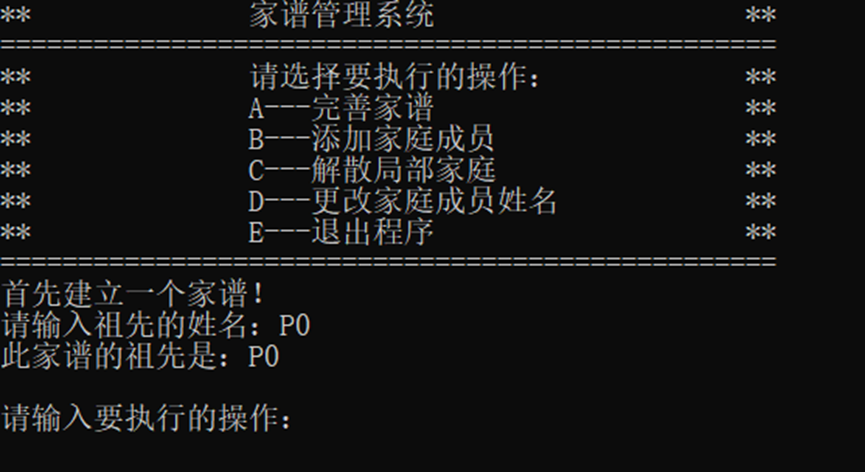
## 4.1 功能测试

### 4.1.1 初始化家谱功能测试

**测试用例**：P0

**预期结果**：程序正常运行不崩溃，输出“此家谱的祖先为：P0”

**实验结果：**



### 4.1.2 完善家谱功能测试

**测试用例：**

P0

A

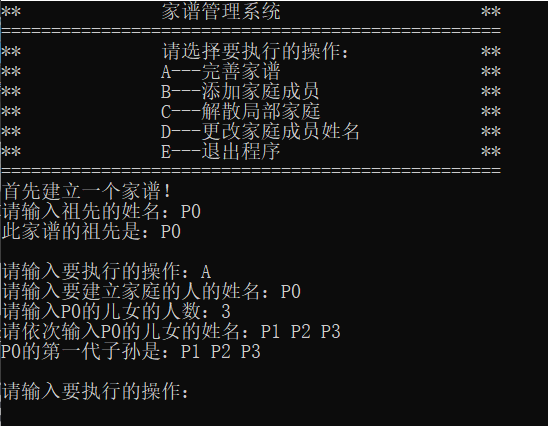
P0

3

P1 P2 P3

**预期结果**：程序正常运行不崩溃,输出“P0的第一代子孙是：P1 P2 P3”

**实验结果：**



### 4.1.3 添加家庭成员功能测试

**测试用例：**

P0

A

P0

3

P1 P2 P3

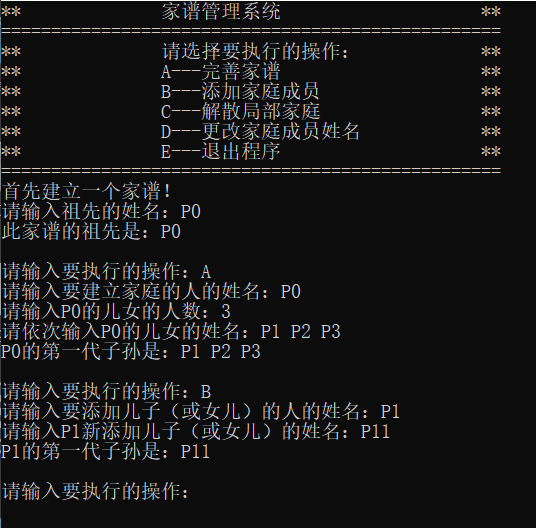
B

P1

P11

**预期结果**：程序正常运行不崩溃,输出“P1的第一代子孙是：P11”

**实验结果：**



### 4.1.4 解散局部家庭功能测试

**测试用例：**

P0

A

P0

3

P1 P2 P3

B

P1

P11

C

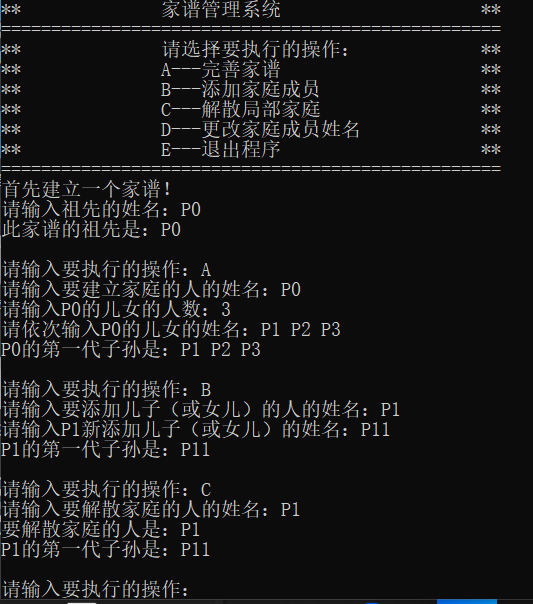
P1

**预期结果：**

程序正常运行不崩溃。

输出“要解散家庭的人是：P1 P1的第一代子孙是：P11”

**实验结果：**



### 4.1.5 更改成员姓名功能测试

**测试用例：**

P0

A

P0

3

P1 P2 P3

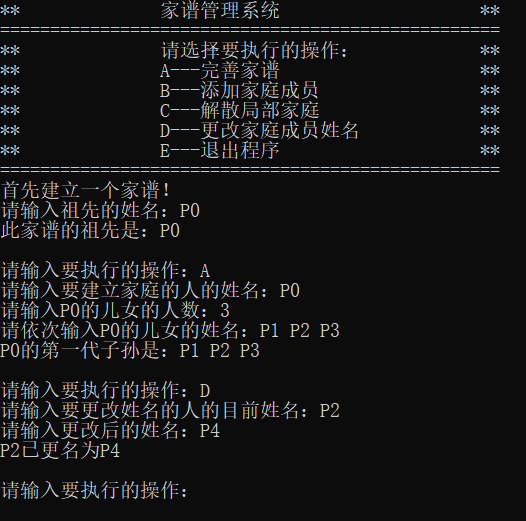
D

P2

P4

**预期结果：**程序正常运行不崩溃,输出“P2已更名为P4”

**实验结果：**



## 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

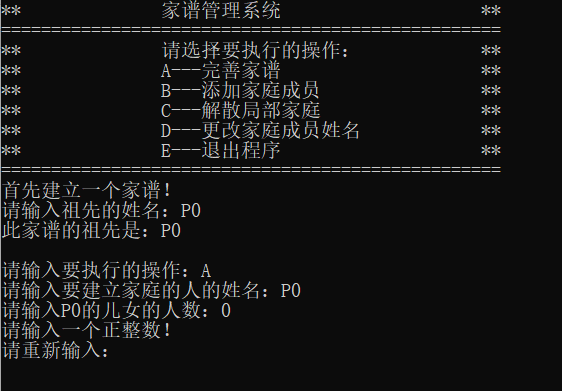
A

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 解散局部家庭时解散第一代祖先家庭

**测试用例：**

P0

A

P0

3

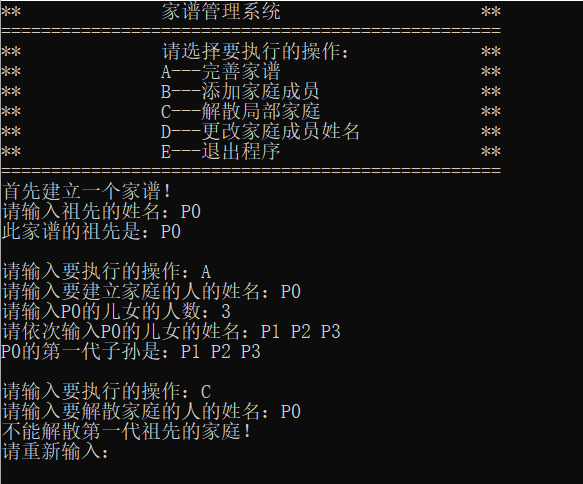
P1 P2 P3

C

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：**



### 4.2.3 家谱中只有祖先时进行解散家庭操作

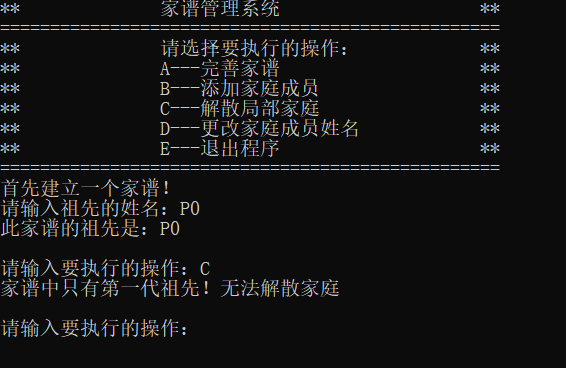
**测试用例：**

P0

C

**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：**



### 4.2.4 对已经建立家庭的人再次建立家庭

**测试用例：**

P0

A

P0

3

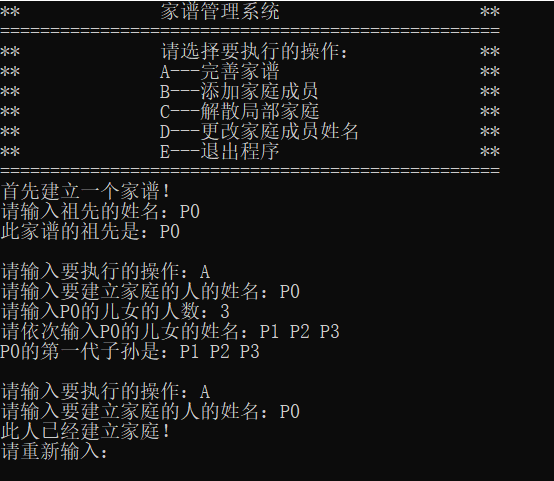
P1 P2 P3

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：**

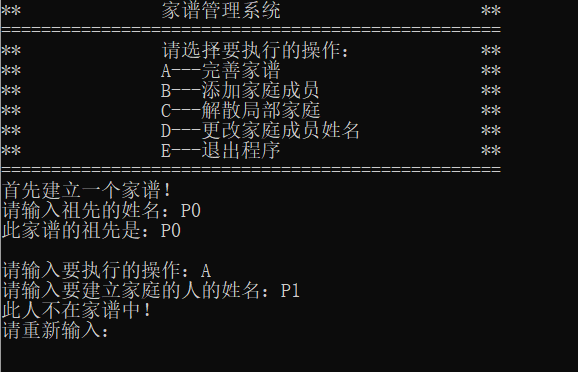
P0

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 完善家谱时新添加儿女个数不合法

**测试用例：**

P0

A

P0

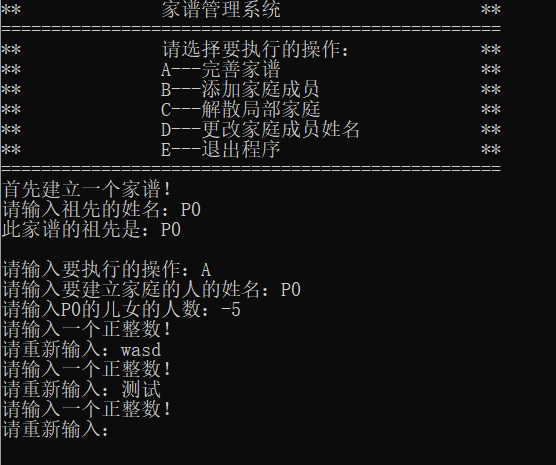
-5

wasd

测试

**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 完善家谱时新添加儿女姓名在家谱中已存在

**测试用例：**

P0

A

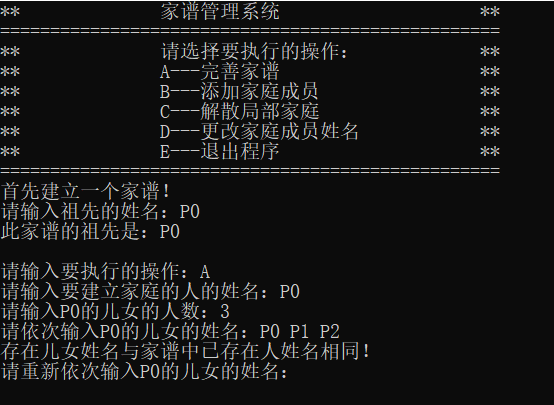
P0

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：**



### 4.3.4 完善家谱时新添加儿女姓名存在重复

**测试用例：**

P0

A

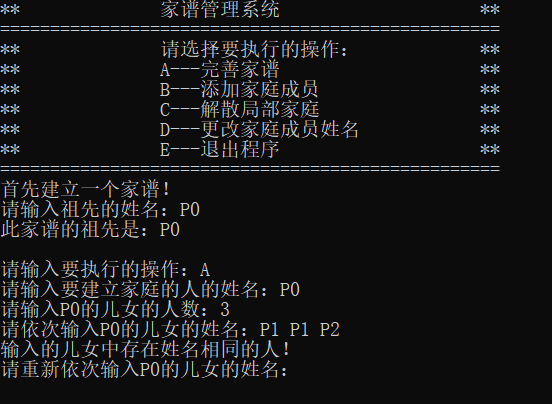
P0

3

P1 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：**



### 4.3.5 完善家谱时新添加儿女姓名过多

**测试用例：**

P0

A

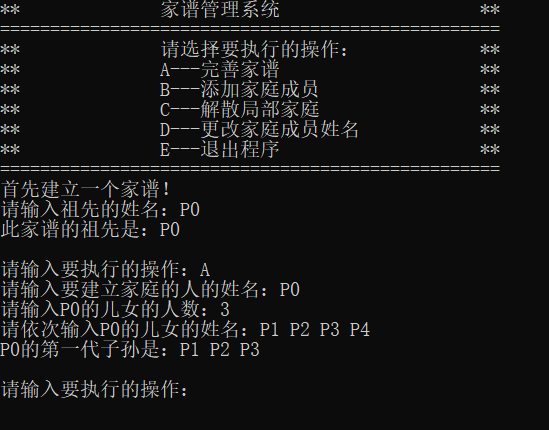
P0

3

P1 P2 P3 P4

**预期结果：**程序忽略多余的姓名，程序正常运行不崩溃。

**实验结果：**



### 4.3.6 添加家庭成员时输入的姓名不存在

**测试用例：**

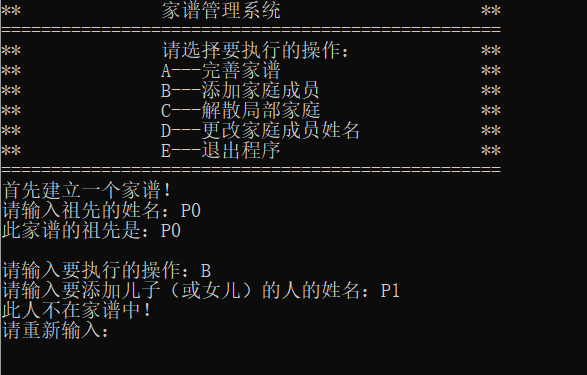
P0

B

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：**



### 4.3.7 添加家庭成员时新添加儿女姓名在家谱中已存在

**测试用例：**

P0

A

P0

3

P1 P2 P3

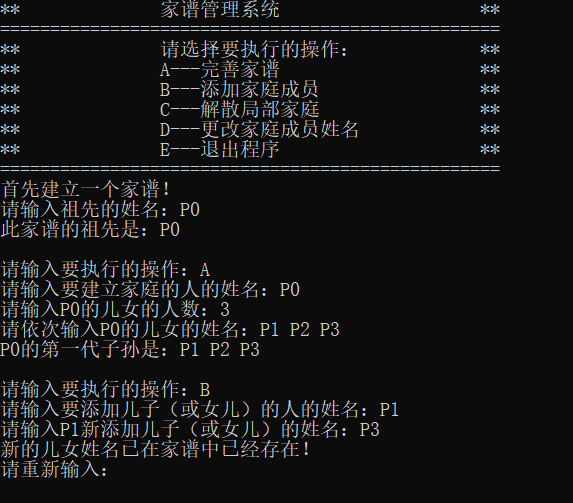
B

P1

P3

**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：**



### 4.3.8 添加家庭成员时新添加儿女姓名输入多个

**测试用例：**

P0

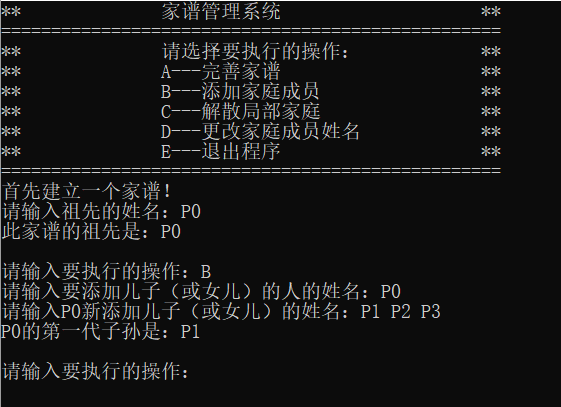
B

P0

P1 P2 P3

**预期结果：**程序只保留第一个，忽略之后的姓名，程序正常运行不崩溃。

**实验结果：**



### 4.3.9 解散局部家庭时输入的姓名不存在

**测试用例：**

P0

A

P0

3

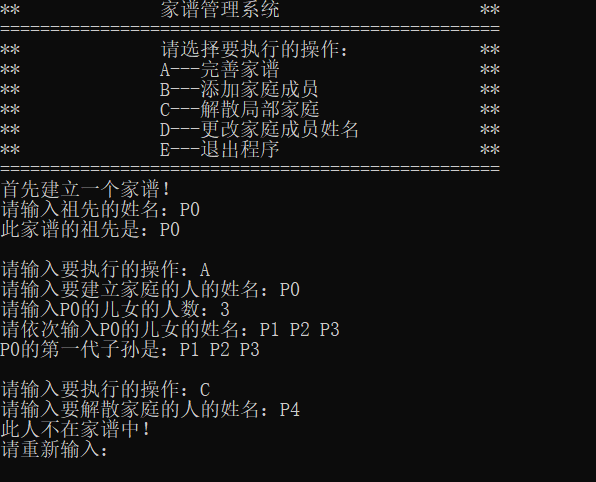
P1 P2 P3

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：**



### 4.3.10 更改成员姓名时输入的原姓名不存在

**测试用例：**

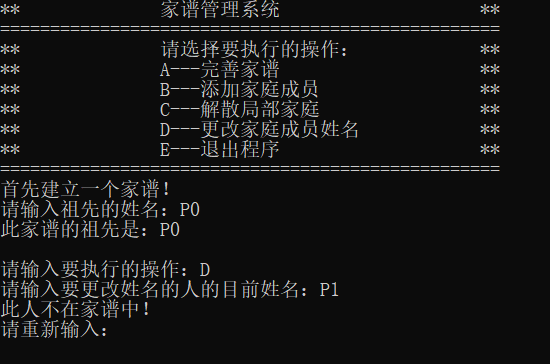
P0

D

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：**



### 4.3.11 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

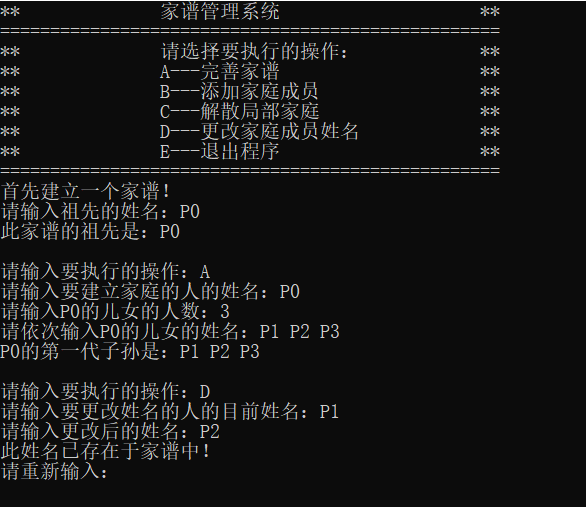
D

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：**

****

### 4.3.12 输入操作数不合法

**测试用例：**

P0

F

1

测试

**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：**

