项目说明文档

数据结构课程设计

——家谱管理系统

作 者 姓 名：张翔

学　　号：2352985

指 导 教 师：张颖

学院、 专业：计算机科学与技术学院 软件工程

同济大学

Tongji University

二〇二四 年 十二 月 七 日

1 项目分析

1.1 项目背景分析

家谱是一种独特的文献形式，专门以谱表方式记录一个家族的世系繁衍、重要人物事迹及血缘关系。作为中华民族三大文献（国史、地志、族谱）之一，家谱是中国特有的文化瑰宝，具有重要的人文价值。它不仅是历史学、民俗学、人口学、社会学和经济学研究的重要资料，也是中华文化传承的珍贵载体。

本项目旨在开发一个家谱管理系统，用于创建和维护家族的家谱信息。该系统帮助用户记录家族成员、梳理亲缘关系，并保存家族的历史与重要信息。通过这一系统，用户可以追溯家族的血脉渊源，了解祖先事迹，进而促进家族文化的传承与研究，对家族历史的保存具有深远意义。

1.2 项目需求分析

基于以上背景分析，本项目需要实现需求如下：

(1)实现对家族成员信息的完善、添加、解散、更改、统计等功能，确保数据的准确、高效管理；

(2)设计简单直观的控制台界面，使操作便捷、容易上手，适应不同用户的操作习惯；

(3)选择合适的数据结构，以支持对家族成员信息的高效操作，同时考虑信息的关联性和复杂度；

(4)实现异常处理机制，确保系统稳定性和安全性，避免因用户输入错误导致系统崩溃；

(5)设计系统以支持未来的扩展和功能增加，满足不同用户、不同应用场景下的需求。

1.3 项目功能分析

1.3.1 完善家谱功能

允许用户为特定成员添加直系后代。这是建立家谱时的关键功能，它允许用户逐步构建完整的家族树。

1.3.2 添加家庭成员功能

用户可以添加新的家庭成员到现有的家谱中。这对于记录新生儿或通过婚姻等方式加入家族的成员非常重要。

1.3.3 解散家庭成员功能

提供删除特定家庭成员的后代的选项，这对于维护准确和最新的家谱数据很重要。

1.3.4 更改家庭成员功能

允许修改家庭成员的姓名。这一功能在家庭成员改名或者录入错误时非常有用。

1.3.5 统计家庭成员功能

提供统计和查看特定家庭成员及其后代的功能，便于用户了解家族规模和成员构成。

1.3.6 异常处理功能

实现异常处理机制，处理用户可能输入的非法信息，确保系统的稳定性和安全性。

2 项目设计

2.1 数据结构设计

基于项目分析，家谱管理系统的设计中选择使用二叉树结构作为数据结构而不是一般树结构，主要基于以下几个考虑：

(1)简化结构和操作：在家谱管理中，每个家庭成员通常有多个孩子，但使用二叉树可以将这种复杂性简化。第一个孩子作为左孩子，其余孩子作为右孩子的兄弟链表可以简化添加、删除等操作。

(2)时间和空间效率：二叉树的遍历、搜索、插入和删除操作相对一般树来说更加简单明了。二叉树相较于一般树在存储上更加高效。每个节点只需存储两个子节点的链接，减少了存储空间的浪费。对于家谱管理系统，大部分家庭成员的后代数量并不会非常庞大，因此使用二叉树不会造成太多的空间浪费。

(3)易于扩展和修改：二叉树的结构使得对家谱的扩展和修改变得容易。例如，添加新成员或更改现有成员关系时，只需调整有限的节点链接。对于家族成员的插入和删除操作，二叉树提供了较高的灵活性和效率。

(4)便于家族层级展示：使用二叉树可以方便地展示家族成员的层级关系。通过左孩子和右孩子的关系，可以清晰地表达家族成员之间的直系和旁系血缘关系。家谱的图形化展示在使用二叉树时更加直观和易于理解。

2.2 结构体与类设计

2.2.1 MyTreeNode 结构体的设计

2.2.1.1 概述

MyLinkNode 结构体是一个用于构建链表节点的模板结构体。该结构体用于表示链表中的每个节点，其中包括节点存储的数据以及指向下一个节点的指针。

2.2.1.2 类定义

```
template <typename Type>
struct MyTreeNode {
```

```
    Type data;
    MyTreeNode<Type>* left;
    MyTreeNode<Type>* right;
    MyTreeNode(const Type& value) : data(value), left(nullptr), right(nullptr) {}
};
```

2.2.2 MyBinaryTree 类的设计

2.2.2.1 概述

MyBinaryTree 是一个模板类，用于实现二叉树数据结构。它的私有成员变量 root 存储二叉树的根节点指针。构造函数初始化 root 为 nullptr，并提供了一个可以接受元素参数初始化树的构造函数。析构函数调用 destroy 方法销毁树中的所有节点。

该类提供了多个方法来操作和查询二叉树，包括：isEmpty 检查树是否为空；setLeftChild 和 setRightChild 分别设置父节点的左子节点和右子节点；modifyNode 修改指定节点的数据；getSize 计算子树的节点数；getRoot 获取根节点指针；getParent 获取指定节点的父节点；getLeftChild 和 getRightChild 分别获取指定节点的左子节点和右子节点；findNode 查找某个元素所在的节点。

此外，类还提供了遍历方法 preorder、inorder 和 postorder，分别用于前序、中序和后序遍历二叉树。最后，重载了赋值运算符 operator= 和 copyTree 方法，用于复制树的内容。

2.2.2.2 类定义

```
template <typename Type>
class MyBinaryTree {
private:
    MyTreeNode<Type>* root;
public:
    MyBinaryTree() : root(nullptr) {}
    MyBinaryTree(Type& item);
    ~MyBinaryTree() { destroy(root); }
    void destroy(MyTreeNode<Type>* subTree);
    bool isEmpty(void) { return root == nullptr; }
    bool setLeftChild(MyTreeNode<Type>* parent, Type& item);
    bool setRightChild(MyTreeNode<Type>* parent, Type& item);
    bool modifyNode(MyTreeNode<Type>* current, Type& item);
```

```
    int getSize(MyTreeNode<Type>* current) { return (current == nullptr) ? 0 :
(getSize(current->left) + getSize(current->right) + 1); }
    MyTreeNode<Type>* getRoot(void) { return root; }
    MyTreeNode<Type>* getParent(MyTreeNode<Type>* current, MyTreeNode<Type>* subTree);
   MyTreeNode<Type>* getLeftChild(MyTreeNode<Type>* current) { return current == nullptr ?
nullptr : current->leftChild; }
    MyTreeNode<Type>* getRightChild(MyTreeNode<Type>* current) { return current ==
nullptr ? nullptr : current->rightChild; }
    MyTreeNode<Type>* findNode(const Type& item, MyTreeNode<Type>* subTree);
    // 遍历方法
    void preorder(MyTreeNode<Type>* node);
    void inorder(MyTreeNode<Type>* node);
    void postorder(MyTreeNode<Type>* node);
    // 运算符重载
    MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>& other);
    MyTreeNode<Type>* copyTree(MyTreeNode<Type>* subTree);
};
```

2.2.3 PersonInfo 结构体的设计

2.2.3.1 概述

PersonInfo 结构体用于存储个人的基本信息，主要包括姓名字段 name，其长度由常量 NameMaxLength 决定。构造函数初始化姓名为空字符串。该结构体重载了赋值运算符 operator=，确保对象之间的赋值操作时不会发生自我赋值，并正确复制姓名信息。此外，还重载了相等运算符 operator==，用于比较两个 PersonInfo 对象的姓名是否相等。该结构体还声明了两个友元函数 operator>> 和 operator<<，分别用于实现从输入流读取和向输出流写入 PersonInfo 对象，便于输入和输出操作。

2.2.3.2 结构体定义

```
struct PersonInfo
{
    char name[NameMaxLength + 1];
    PersonInfo() { strcpy(name, ""); }
```

```
    PersonInfo& operator=(const PersonInfo& other) { if (this != &other) strcpy(name, other.name); return *this; }
    bool operator==(const PersonInfo& other) { return strcmp(name, other.name) == 0; }
    friend std::istream& operator>>(std::istream& in, PersonInfo& info);
    friend std::ostream& operator<<(std::ostream& out, PersonInfo& info);
};
```

2.2.4 family 类的设计

2.2.4.1 概述

family 类是一个用于管理家谱的类，封装了家谱的基本操作功能。它包含一个私有成员变量 myfamily，表示家谱的数据结构，基于二叉树实现，存储 PersonInfo 类型的成员信息。类的构造函数通过传入祖先信息初始化家谱的根节点。公共成员函数提供了多种功能，包括 selectOptn 用于用户操作选择，completeFamilyMembers 添加一个成员的所有后代，addFamilyMembers 添加单个后代，removeFamilyMembers 解散某成员的家庭，changeFamilyMembers 修改成员的姓名，以及 countFamilyMembers 统计某成员的子孙数量。

2.2.4.2 类定义

```
class family
{
private:
    MyBinaryTree<PersonInfo> myfamily;
public:
    family(PersonInfo& ancestor);
    bool selectOptn();
    void completeFamilyMembers();
    void addFamilyMembers();
    void removeFamilyMembers();
    void changeFamilyMembers();
    void countFamilyMembers();
};
```

2.3 项目主体架构设计

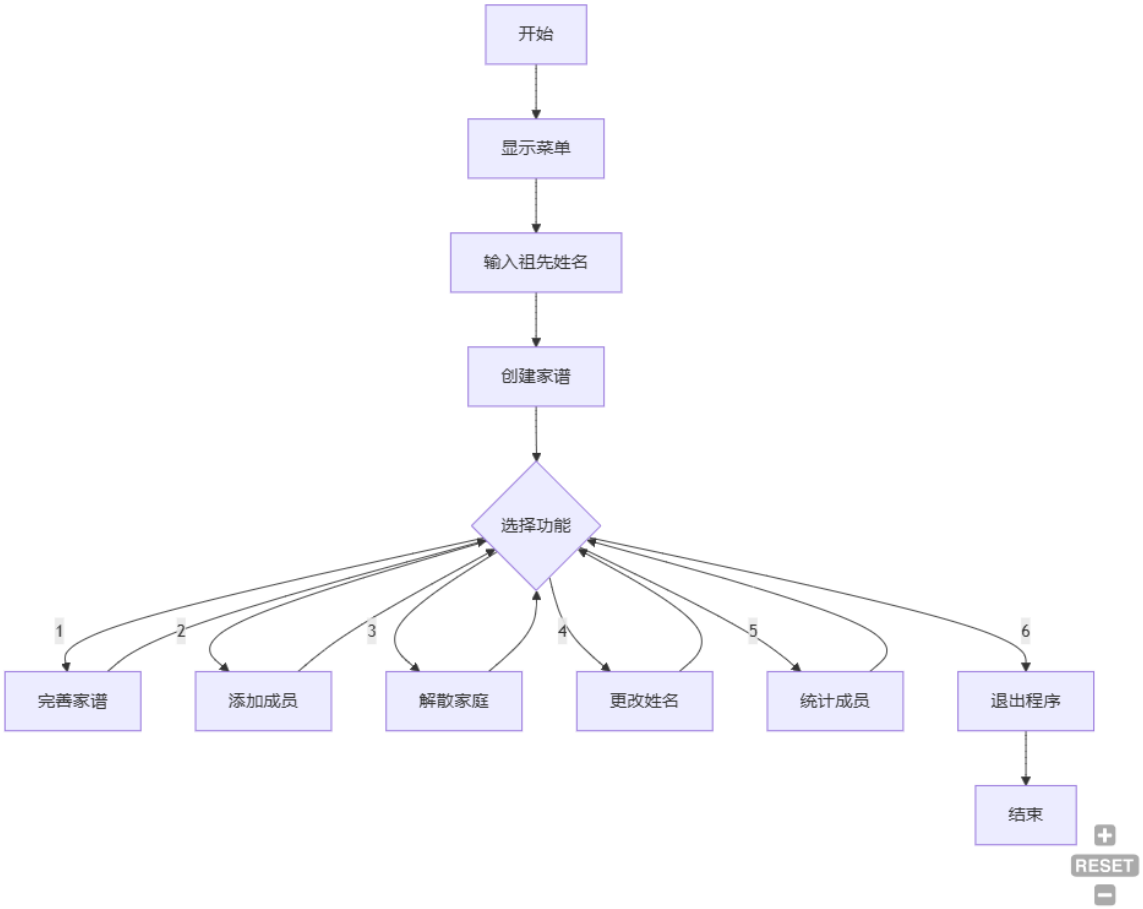


图 2.3.1 项目主体架构设计流程图

3 项目功能实现

3.1 项目主体架构的实现

3.1.1 项目主体架构实现思路

这段代码通过输出菜单界面展示家谱管理系统的功能选项，引导用户输入祖先姓名创建家谱，并初始化 family 类实例来管理家谱树。在一个循环中处理用户选择，通过调 myFamily.selectOptn() 实现具体功能，直至用户选择退出，最后打印提示并结束程序。

3.1.2 项目主体架构核心代码

```
int main()
{
```

```
    std::cout << "+---------------------------------+" << std::endl;
    std::cout << "|           家谱管理系统          |" << std::endl;
    std::cout << "|  Family_Tree_Management_System  |" << std::endl;
    std::cout << "+---------------------------------+" << std::endl;
    std::cout << "|       [1] --- 完善家谱          |" << std::endl;
    std::cout << "|       [2] --- 添加家庭成员      |" << std::endl;
    std::cout << "|       [3] --- 解散局部家庭      |" << std::endl;
    std::cout << "|       [4] --- 更改成员姓名      |" << std::endl;
    std::cout << "|       [5] --- 统计家庭成员      |" << std::endl;
    std::cout << "|       [6] --- 退出程序          |" << std::endl;
    std::cout << "+---------------------------------+" << std::endl << std::endl;
    /* 系统开始提示 */
    std::cout << ">>> 请建立家谱管理系统" << std::endl << std::endl;
    std::cout << "请输入祖先姓名: ";
    PersonInfo ancestor;
    std::cin >> ancestor;
    family myFamily(ancestor);
    std::cout << std::endl << ">>> 家谱已建立，祖先为: " << ancestor.name << std::endl;
    /* 循环处理用户输入 */
    do {
        std::cout << std::endl << ">>> 请选择要使用的功能: ";
    } while (myFamily.selectOptn());
    /* 系统退出提示 */
    std::cout << ">>> 已成功退出家谱管理系统"  << std::endl;
    return 0;
}
```

3.2 完善家谱功能的实现

3.2.1 完善家谱功能实现思路

这个函数实现了为家谱中指定长辈添加后代的功能。

首先，通过用户输入找到对应的长辈节点，若未找到或该长辈已有家庭，则提示错误并退出。

接着，用户输入长辈的儿女数量和姓名，并逐个检查输入的姓名是否已存在于家谱中，若重复则要求重新输入。对于每个有效的儿女，按照长子为左子节点、其他子女为右子节点的规则依次插入到家谱中，最终输出该长辈的所有后代姓名并释放内存。

3.2.2 完善家谱功能核心代码

```
void family::completeFamilyMembers()
{
    std::cout << "请输入要添加后代的长辈的姓名: ";
    PersonInfo parentInfo;
    std::cin >> parentInfo;
    std::cout << std::endl;
    MyTreeNode<PersonInfo>* root = myfamily.getRoot();
    MyTreeNode<PersonInfo>* parent = myfamily.findNode(parentInfo, root);
    if (parent == nullptr) {
        std::cout << ">>> 未在家谱中找到" << parentInfo << "，请确认姓名输入是否正确！"
<< std::endl;
        return;
    }
    if (parent->left != nullptr) {
        std::cout << ">>> " << parentInfo << "已建立家庭" << std::endl;
        return;
    }
    int num = inputInteger(1, 10, "该长辈的儿女数量");
    std::cout << std::endl << ">>> 请依次输入" << parentInfo << "的儿女的姓名" << std::endl
<< std::endl;
    PersonInfo* descendants = new(std::nothrow) PersonInfo[num];
    if (descendants == nullptr) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(-1);
    }
    for (int i = 0; i < num; i++) {
        std::cout << "请输入" << parentInfo << "的第" << i + 1 << "个儿女的姓名: ";
```

```
        std::cin >> descendants[i];
        if (myfamily.findNode(descendants[i], myfamily.getRoot())) {
            std::cout << std::endl << ">>> " << descendants[i--] << "在家谱中已存在" <<
std::endl << std::endl;
            continue;
        }
        if (i == 0) {
            myfamily.setLeftChild(myfamily.findNode(parentInfo, myfamily.getRoot()),
descendants[i]);
        }
        else {
            myfamily.setRightChild(myfamily.findNode(descendants[i - 1],
myfamily.getRoot()), descendants[i]);
        }
    }
    std::cout << std::endl << ">>> " << parentInfo << "的下一代子孙是: ";
    for (int i = 0; i < num; i++)
        std::cout << descendants[i] << (i == num - 1 ? "" : " ");
    delete[]descendants;
    std::cout << std::endl;
}
```

3.3 添加家庭成员功能的实现

3.3.1 添加家庭成员功能实现思路

这个函数实现了为家谱中指定长辈添加单个后代的功能。首先，通过用户输入找到长辈节点，若未找到则提示错误并退出。随后，用户输入后代的姓名，并检查是否已存在于家谱中，若重复则重新输入。对于有效的后代，如果长辈尚无儿女，则将其设置为左子节点；否则，沿着当前儿女的右子节点链查找到最后一个节点，将后代作为新的右子节点添加。最后，输出长辈新增后的所有儿女姓名，并提示添加成功。

3.3.2 添加家庭成员功能核心代码

```
void family::addFamilyMembers()
```

```
{
    std::cout << "请输入要添加后代的长辈的姓名: ";
    PersonInfo parentInfo,childInfo;
    std::cin >> parentInfo;
    std::cout << std::endl;
    MyTreeNode<PersonInfo>* parent = myfamily.findNode(parentInfo, myfamily.getRoot());
    if (parent == nullptr) {
        std::cout << ">>> 未在家谱中找到" << parentInfo << ", 请确认姓名输入是否正确! "
<< std::endl;
        return;
    }
    while (true) {
        std::cout << "请输入要添加的后代的姓名: ";
        std::cin >> childInfo;
        if (myfamily.findNode(childInfo, myfamily.getRoot())) {
            std::cout << std::endl << ">>> " << childInfo << "在家谱中已存在" << std::endl;
            continue;
        }
        break;
    }
    std::cout << std::endl << ">>> 添加成功 " << std::endl << std::endl;
    std::cout << ">>> 添加后" << parentInfo << "的下一代儿女是: ";
    if (parent->left == nullptr) {
        myfamily.setLeftChild(parent, childInfo);
    }
    else {
        MyTreeNode<PersonInfo>* current = parent->left;
        std::cout << current->data << " ";
        while (current->right != nullptr) {
            current = current->right;
            std::cout << current->data << " ";
```

```
        }
        myfamily.setRightChild(current, childInfo);
    }
    std::cout << childInfo << std::endl;
}
```

3.4 解散家庭成员功能的实现

3.4.1 解散家庭成员功能实现思路

这个函数实现了从家谱中解散指定成员家庭的功能。首先，通过用户输入找到目标成员节点，若未找到则提示错误并退出；若该成员没有家庭（左子节点为空），则直接提示无需操作并退出。否则，调用家谱的 destroy 方法递归删除该成员的左子树（即所有后代节点），并将其左子节点指针置为空，完成家庭解散操作。最后，输出解散成功的提示信息。

3.4.2 解散家庭成员功能核心代码

```
void family::removeFamilyMembers()
{
    std::cout << "请输入要解散家庭成员的人的名字: ";
    PersonInfo targetInfo;
    std::cin >> targetInfo;
    std::cout << std::endl;
    MyTreeNode<PersonInfo>* target = myfamily.findNode(targetInfo, myfamily.getRoot());
    if (target == nullptr) {
        std::cout  << ">>> 未在家谱中找到" << targetInfo << "，请确认姓名输入是否正确！" << std::endl;
        return;
    }
    if (target->left == nullptr) {
        std::cout << ">>> " << targetInfo << "无家庭成员" << std::endl;
        return;
    }
    myfamily.destroy(target->left);
    target->left = nullptr;
```

```
    std::cout << ">>> " << targetInfo << "的家庭成员已解散" << std::endl;
}
```

3.5 更改家庭成员姓名功能的实现

3.5.1 更改家庭成员姓名功能实现思路

这个函数实现了家谱中成员姓名的修改功能。首先，通过用户输入找到需要修改名字的成员节点，若未找到则提示错误并退出。接着，用户输入新的姓名，并检查新姓名是否已存在于家谱中，若重复则要求重新输入。对于有效的新姓名，调用 myfamily.modifyNode 方法更新节点信息为新姓名。最后，输出修改成功的提示信息，显示原姓名和更改后的新姓名。

3.5.2 更改家庭成员姓名功能核心代码

```
void family::changeFamilyMembers()
{
    std::cout << "请输入需要修改名字的成员的更改前姓名: ";
    PersonInfo targetInfo,infoAfterChange;
    std::cin >> targetInfo;
    std::cout << std::endl;
    MyTreeNode<PersonInfo>* target = myfamily.findNode(targetInfo, myfamily.getRoot());
    if (target == nullptr) {
        std::cout << ">>> 未在家谱中找到" << targetInfo << ", 请确认姓名输入是否正确！"
<< std::endl;
        return;
    }
    while (true) {
        std::cout << "请输入要更改姓名的人的更改后姓名: ";
        std::cin >> infoAfterChange;
        if (myfamily.findNode(infoAfterChange, myfamily.getRoot())) {
            std::cout << std::endl << ">>> " << infoAfterChange << "在家谱中已存在" <<
std::endl;
            continue;
        }
        break;
```

```
    }
    myfamily.modifyNode(target, infoAfterChange);
    std::cout << std::endl << ">>> " << targetInfo << "已更名为" << infoAfterChange <<
std::endl;
}
```

3.6 统计家庭成员功能的实现

3.6.1 统计家庭成员功能实现思路

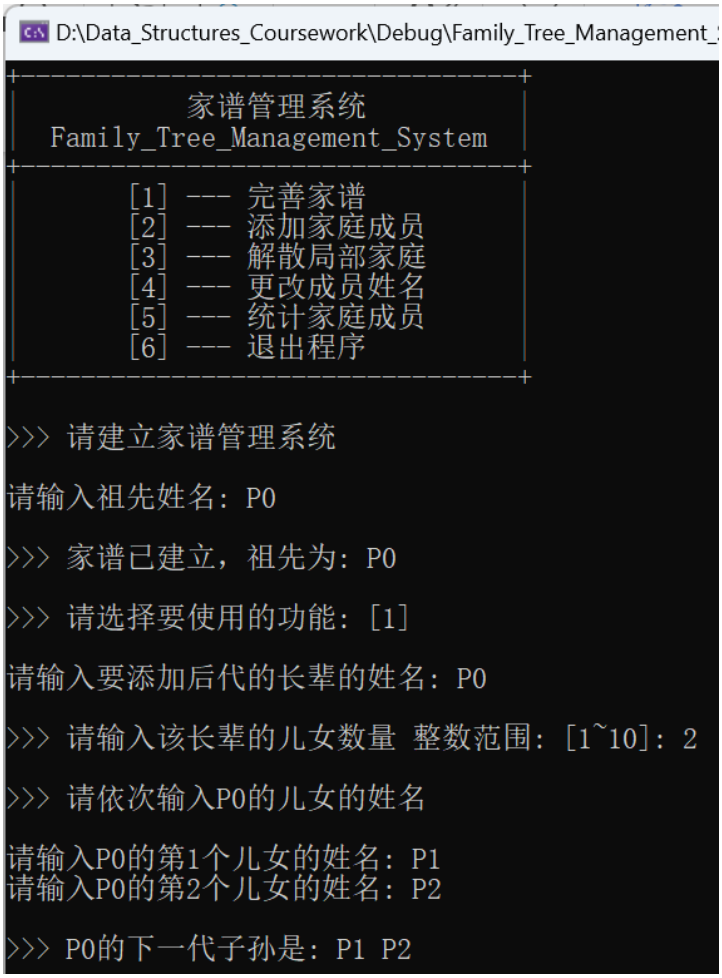
这个函数实现了统计指定家庭成员的子孙数量并输出其名字的功能。首先，通过用户输入找到目标家庭成员的节点，若未找到则提示错误并退出。然后，调用 myfamily.getSize 方法计算该成员左子树(代表其后代)的节点总数,得到其子孙数量。若子孙数量不为零,则调用 myfamily.inorder 方法对左子树进行中序遍历,依次输出其子孙的名字。最后,显示统计结果及子孙列表(如果存在)。

3.6.2 统计家庭成员功能核心代码

```
void family::countFamilyMembers()
{
    std::cout << "请输入要统计的家庭成员的人的姓名: ";
    PersonInfo targetInfo;
    std::cin >> targetInfo;
    std::cout << std::endl;
    MyTreeNode<PersonInfo>* target = myfamily.findNode(targetInfo, myfamily.getRoot());
    if (target == nullptr) {
        std::cout << ">>> 未在家谱中找到" << targetInfo << ", 请确认姓名输入是否正确！"
<< std::endl;
        return;
    }
    int num = myfamily.getSize(target->left);
    std::cout << std::endl << ">>> " << targetInfo << "共有" << num << "个儿女";
    if (num != 0) {
        std::cout << ", 分别是: ";
        myfamily.inorder(target->left);
    }
```

```
    std::cout << std::endl;
}
```

4 项目测试



2 图 4.1 完善家谱功能示例

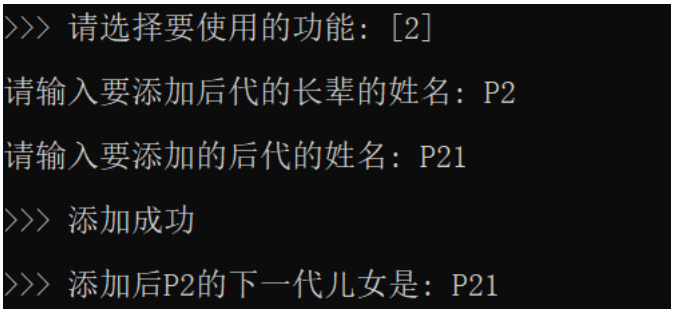


图 4.2 添加家庭成员功能示例

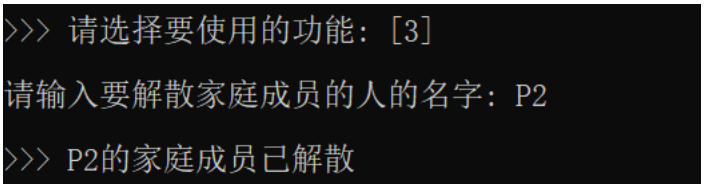


图 4.3 解散局部家庭功能示例

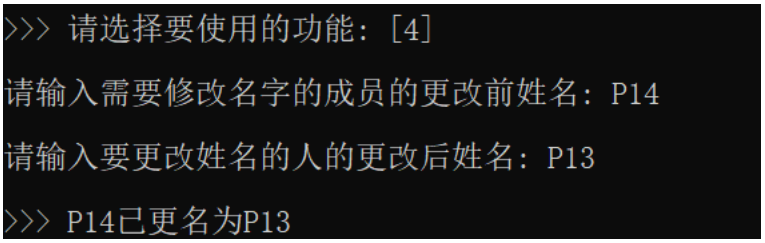


图 4.4 更改家庭成员名称功能示例

5 集成开发环境与编译运行环境

Windows 系统：Windows 11 x64

Windows 集成开发环境：Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境：本项目适用于 x86 架构和 x64 架构