**项目说明文档**

**数据结构课程设计**

**——家谱管理系统**

作 者 姓 名 刘淑仪

学 号 2251730

指 导 教 师 张 颖

学 院 专 业 软件学院 软件工程



二〇二三 年 十 月 二十五 日

目录

1 项目分析 1

1.1 项目背景分析 1

1.2 项目功能分析 1

2 项目设计 2

2.1 数据结构设计 2

2.2 类结构设计 2

2.3 成员与操作设计 3

2.4 系统设计 6

3 功能设计与项目实现 7

3.1 初始化家谱功能的实现 7

3.1.1 初始化家谱功能实现思路 7

3.1.2 初始化家谱功能核心代码 7

3.1.3 初始化家谱功能实现示例 8

3.2 完善家谱功能的实现 8

3.2.1 完善家谱功能实现思路 8

3.2.2 完善家谱功能核心代码 8

3.2.3 完善家谱功能实现示例 11

3.3 添加家庭成员功能的实现 12

3.3.1 添加家庭成员功能实现思路 12

3.3.2 添加家庭成员功能核心代码 12

3.3.3 添加家庭成员功能实现示例 13

3.4 解散局部家庭功能的实现 13

3.4.1 解散局部家庭功能实现思路 13

3.4.2 解散局部家庭功能核心代码 14

3.4.3 解散局部家庭功能实现示例 14

3.5 修改家庭成员信息功能的主要实现 15

3.5.1 修改家庭成员信息功能实现思路 15

3.5.2 修改家庭成员信息功能核心代码 15

3.5.3 修改家庭成员信息功能实现示例 16

3.6 Tree.h中主要功能的实现 16

3.6.1 Tree.h头文件中主要包含的类设计 16

3.6.2 Tree.h头文件核心代码 16

4 项目测试 16

4.1 功能测试 16

4.1.1 基本功能测试 16

4.1.2 测试重复名字输入 17

4.1.3 测试解散不存在的家庭 18

4.1.4 测试解散祖先 19

4.1.5 测试修改的名字重复 19

4.2 Linux环境测试 20

5 集成开发环境与编译运行环境 21

# 项目分析

## 项目背景分析

随着时间的推移，传统的家谱记录方式—手写或印刷—逐渐不能满足现代社会的需求。数字化、网络化的信息管理方式为家谱的保存、更新和查询提供了新的可能性。当前，家谱在历史学、民俗学、人口学、社会学和经济学研究中的应用愈发广泛，成为了连接个人与家族、家族与社会的重要桥梁。

本项目通过实现家谱管理的数字化，不仅响应了信息技术发展的趋势，也为传承和弘扬家族文化提供了新的途径。数字化家谱管理系统可以更有效地管理大量数据，便于家族成员之间的沟通和信息共享，同时也有助于家族历史的长期保存。

## 项目功能分析

本家谱管理系统项目的核心功能包括家族成员信息的建立、查找、插入、修改和删除。这些功能基于之前提供的源代码进行分析和扩展。

其主要功能的实现主要依赖于

1、树结构建立和遍历算法：使用二叉树（MyBinaryTree类）来表示家族成员关系。每个节点（TreeNode）代表一个家庭成员。实现了前序、中序、后序和层次遍历。这些遍历方法用于访问树中的每个节点，以实现不同的功能，如打印家谱、查找特定成员等。

前序遍历 (preOrder)：用于快速访问和显示整个家族树的结构。

中序遍历 (inOrder)：主要用于排序或搜索操作。

后序遍历 (postOrder)：可用于删除或释放树节点。

层次遍历 (levelOrder)：适用于展示家谱的层级结构。

2、成员插入算法：

左插入 (leftInsertNode)：将新成员作为当前节点的第一个孩子插入，用于添加家族成员的直系后代。

右插入 (rightInsertNode)：将新成员作为当前节点的兄弟节点插入，用于在同一代中添加新成员。

3、成员删除算法：

查找节点 (findNode)：基于成员名称，遍历家族树以找到特定的成员节点。

修改节点 (modifyNode)：更改家族树中特定成员的信息，如姓名更改。

4、数据完整性和验证算法：重复性检查 (IsDuplicate)：确保添加的家族成员在家谱中是唯一的，避免重复。

# 项目设计

## 数据结构设计

本家谱管理系统项目通过精心设计的数据结构和算法，实现了一个功能全面的家族信息管理工具。核心数据结构包括二叉树（MyBinaryTree类）和队列（LinkedQueue类），它们共同构成了家族树的基础框架。每个家族成员由TreeNode节点表示，其中包含个人信息（例如姓名），并通过指针与其他成员相连，形成家族树的结构。

项目的关键功能涵盖了家族成员的添加、查找、修改、删除等操作，通过这些功能，用户可以构建和维护详尽的家族树。家族树的遍历方法（如前序、中序、后序和层次遍历）提供了不同的视角来观察和分析家族关系，使用户能够以多种方式访问和展示家族信息。此外，项目中还考虑了数据的完整性和一致性，如通过重复性检查来确保家族成员的唯一性。

## 类结构设计

首先在Tree.h中，主要核心是TreeNode 类和MyBinaryTree 类。前者代表家谱中的一个成员，是家族树的基本构建单元。它包含成员信息（如姓名）和指向家族树中其他成员（第一个孩子和下一个兄弟）的指针。后者作为整个家谱的管理中心，维护家族树的结构和操作。其拥有一个根节点，代表家族的始祖。提供了一系列方法来添加、删除、查找和遍历家族成员，使得家族树的维护和管理变得直观且灵活。辅助类有QueueNode类和LinkedQueue类助于遍历的开展。

然后在本项目的主要cpp(genealogy management system.cpp)中，创建GenealogyManagement类来存储MyBinaryTree<People> familyTree（用于存储和操作整个家族树）作为私有成员，公有成员则为实现本系统基本功能的各函数。同时TreeNode<People> 类用于表示家族树中的一个节点，即一个家族成员。功能是存储成员信息和指向家族树中其他成员的指针。MyBinaryTree<People> 类用于管理整个家族树。功能是提供家族树的建立、遍历、成员操作等功能。

## 成员与操作设计

Tree.h

// 定义队列便于进行层次序遍历

// 链表节点定义

template<typename Type>

class QueueNode {

public:

TreeNode<Type>\* data;

QueueNode<Type>\* next;

QueueNode(TreeNode<Type>\* node) : data(node), next(NULL) {}

};

// 链队定义

template<typename Type>

class LinkedQueue {

private:

QueueNode<Type>\* front; // 队列头指针

QueueNode<Type>\* rear; // 队列尾指针

public:

LinkedQueue() : front(NULL), rear(NULL) {}

~LinkedQueue() {

while (!isEmpty()) { dequeue(); }

}

bool isEmpty() const { return front == NULL; }

void enqueue(TreeNode<Type>\* node) {

QueueNode<Type>\* newNode = new QueueNode<Type>(node);

if (rear == NULL) {

front = rear = newNode;

}

else {

rear->next = newNode;

rear = newNode;

}

}

TreeNode<Type>\* dequeue() {

if (isEmpty()) {

return NULL;

}

QueueNode<Type>\* temp = front;

TreeNode<Type>\* node = front->data;

front = front->next;

if (front == NULL) {

rear = NULL;

}

delete temp;

return node;

}

TreeNode<Type>\* getFront() const {

return isEmpty() ? NULL : front->data;

}

};

/\* 构建二叉树 \*/

// 孩子兄弟法定义结点

template<typename Type>

class TreeNode {

public:

Type value;

TreeNode\* firstChild, \* nextSibling;

TreeNode() : firstChild(NULL), nextSibling(NULL) {}

TreeNode(Type item, TreeNode<Type>\* l = NULL, TreeNode<Type>\* r = NULL) : value(item), firstChild(l), nextSibling(r) {}

template<typename T>

friend std::ostream& operator<<(std::ostream& os, const TreeNode<T>& node);

};

// 定义二叉树

template<typename Type>

class MyBinaryTree {

protected:

TreeNode<Type>\* root; // 根节点

TreeNode<Type>\* copy(const TreeNode<Type>\* subTree);

void outputNode(TreeNode<Type>\* node) { if (node != NULL) std::cout << node->value << "->"; }

bool containsRecursive(TreeNode<Type>\* node, const Type& data) {

if (node == nullptr) { return false; }

if (node->value == data) { return true; }

return containsRecursive(node->firstChild, data) || containsRecursive(node->nextSibling, data);

}

public:

MyBinaryTree() : root(NULL) {} // 构造函数

MyBinaryTree(Type& item); // 构造函数

MyBinaryTree(MyBinaryTree<Type>& other) { root = copy(other.root); } // 复制构造函数

~MyBinaryTree() { destroy(root); } // 析构函数

void setRoot(TreeNode<Type>\* newRoot) { root = newRoot; }

void destroy(TreeNode<Type>\*& subTree);

bool isEmpty(void) { return root == NULL; } // 判断二叉树是否为空

// 得到二叉树的高度

int getHeight(TreeNode<Type>\* subTree) { return (subTree == NULL) ? 0 : (std::max(getHeight(subTree->firstChild), getHeight(subTree->nextSibling)) + 1); }

// 得到二叉树的结点数

int getSize(TreeNode<Type>\* subTree) { return (subTree == NULL) ? 0 : (getSize(subTree->firstChild) + getSize(subTree->nextSibling) + 1); }

TreeNode<Type>\* getRoot(void) const { return root; }

TreeNode<Type>\* getParent(TreeNode<Type>\* current, TreeNode<Type>\* subTree);

TreeNode<Type>\* getLeftChild(TreeNode<Type>\* current) { return current == NULL ? NULL : current->firstChild; }

TreeNode<Type>\* getRightChild(TreeNode<Type>\* current) { return current == NULL ? NULL : current->nextSibling; }

TreeNode<Type>\* findNode(const Type& item, TreeNode<Type>\* subTree); // 获取指定值的节点

// 遍历二叉树

void preOrder(TreeNode<Type>\* subTree); // 前序遍历

void inOrder(TreeNode<Type>\* subTree); // 中序遍历

void postOrder(TreeNode<Type>\* subTree); // 后序遍历

void levelOrder(TreeNode<Type>\* subTree); // 层次序遍历

// 插入二叉树

bool leftInsertNode(TreeNode<Type>\* current, const Type& item);

bool rightInsertNode(TreeNode<Type>\* current, const Type& item);

bool modifyNode(const Type& oldItem, const Type& newItem, TreeNode<Type>\* subTree);

bool insertAsFirstChild(TreeNode<Type>\* current, const Type& item);

void deleteChildren(TreeNode<Type>\* node);

MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>& other);

};

在主Cpp里：

// 存储家庭成员的信息

struct People {

char name[MAX\_SIZE]; // 姓名

People() = default;

People(const char\* buf) {

strncpy(name, buf, MAX\_SIZE);

name[MAX\_SIZE - 1] = '\0';

}

// 判断两个成员是否是同一个人

bool operator ==(const People& buf) {return strcmp(name,buf.name) == 0;}

};

// 建立家谱管理系统

typedef class genealogyManagement {

private:

MyBinaryTree<People> familyTree;

public:

void initgenealogy();

void operation();

void completeTree(); // 完善家谱

void addMembers(); // 添加家庭成员

void dissolveFamily(); // 解散局部家庭成员

void changeName(); // 更改家庭成员姓名

bool IsDuplicate(char chiName[maxChildrenNum][MAX\_SIZE]);// 检测建立家庭成员是否重复

void outputChildren(People parent);

void menu();

People inputParents(bool opt);

void Exit() {

std::cout << "按任意键继续...\n";

// 清除输入缓冲区

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

// 等待用户按键

std::cin.get();

}

}GM;

## 系统设计

1. 模块化设计：数据结构模块：TreeNode和MyBinaryTree类构成了家族树的核心数据结构。TreeNode代表家族树中的每一个节点，即一个家庭成员，而MyBinaryTree则提供了管理和操作这些节点的方法，如添加、删除、查找和遍历成员。业务逻辑模块：GenealogyManagement类作为系统的中枢，负责处理用户输入，执行相应的家谱管理操作，如初始化家谱、添加新成员、修改成员信息等。
2. 数据结构的适用性：使用二叉树结构来模拟家族关系图谱，这种数据结构非常适合于家谱管理系统的需求。它允许有效地表现和管理家族成员之间的复杂关系，如父子、兄弟关系，同时也便于进行家族成员的遍历和更新。
3. 封装性和可维护性： 数据和方法都被封装在类中，极大地减少了全局状态的使用。这种封装不仅提高了代码的可读性，还增强了系统的可维护性。每个类和方法都有明确的职责，使得代码更加模块化，便于理解和维护。

# 功能设计与项目实现

## 初始化家谱功能的实现

### 初始化家谱功能实现思路

家谱的初始化首先需要建立一棵树，创建好根节点。根据提示信息输入祖先。在输入祖先的过程中会进行错误处理，至此家谱初始化完毕，可以进行后续操作。

### 初始化家谱功能核心代码

// 建立家谱 确定祖先

void genealogyManagement::initgenealogy()

{

char name[MAX\_SIZE];

std::cout << "\n首先建立家谱;\n";

std::cout << ">>>请输入祖先姓名：";

std::cin.getline(name, MAX\_SIZE);

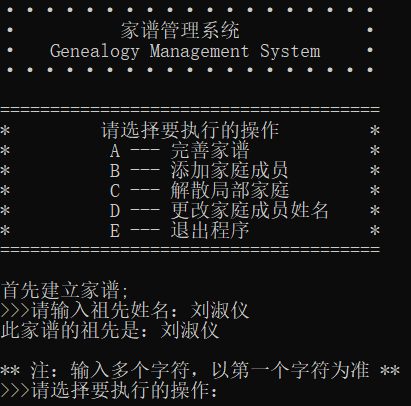
People ancestor(name);

familyTree = MyBinaryTree<People>(ancestor);

std::cout << "此家谱的祖先是：" << familyTree.getRoot()->value << std::endl;

}

### 初始化家谱功能实现示例



## 完善家谱功能的实现

### 完善家谱功能实现思路

完善家谱的主要目的是给祖先输入儿女个数使得最初的家谱得以建立。即先输入儿女人数，再依次输入儿女的姓名。采用二叉树数据结构来存储家谱，是因为用孩子兄弟结点来将原本的树转化成了二叉树便于处理。

在输入儿女人数和姓名时，均作了错误处理。首先，儿女人数不能超过3人（响应国家政策）；其次，儿女的名字与家谱的任何一个人不得重名；最后，输入的儿女名字的个数应该与之前输入的儿女人数匹配，多则舍弃多出来的名字。

### 完善家谱功能核心代码

// 输入父母

People genealogyManagement::inputParents(bool opt)

{

char name[MAX\_SIZE];

People newParents;

/\*

\* 输入要建立家庭/添加儿女的人的姓名

\* 错误判断：该人是否在家族里

\*/

while (true) {

std::cin.getline(name, MAX\_SIZE);

newParents = People(name);

if (familyTree.findNode(newParents, familyTree.getRoot()) == NULL) {

std::cout << "该人不在本家族中！\n";

std::cout << ">>>请重新输入：\n";

}

else if (opt && familyTree.findNode(newParents, familyTree.getRoot())->firstChild != NULL) {

std::cout << newParents.name << "已建立家庭！\n";

std::cout << ">>>请重新输入：\n";

}

else

break;

}

return newParents;

}

// 完善家谱

void genealogyManagement::completeTree()

{

int childrenNum = 0;

char children[MAX\_SIZE];

char chiName[maxChildrenNum][MAX\_SIZE];

People newMember;

People newChild;

// 输入要建立家庭的人的名字

newMember = inputParents(1);

/\*

\* 输入儿女人数

\* 错误判断：儿女人数需要大于0且小于3

\*/

std::cout << ">>>请输入" << newMember.name << "的儿女人数：";

while (true) {

std::cin >> childrenNum;

if (std::cin.fail() || childrenNum <= 0 || childrenNum > maxChildrenNum) {

std::cin.clear();

std::cin.ignore(65536, '\n');

std::cout << "输入数据有误，请重新输入\n";

}

else {

if (std::cin.get() == '\n')

break;

else {

std::cin.clear();

std::cin.ignore(65536, '\n');

std::cout << "输入数据有误，请重新输入\n";

}

}

}

/\*

\* 输入儿女名字

\* 错误判断：超过儿女人数后的名字会被截断；不能重名

\*/

std::cout << "\*\* 注：输入多个名字，只按孩子个数读取 \*\*\n";

std::cout << ">>>请依次输入" << newMember.name << "的儿女的姓名:";

while (true) {

std::cin.getline(children, MAX\_SIZE);

// 将读取到的内容提取

int chiIndex = 0; // 当前正在处理的chiName的索引

int charIndex = 0; // 当前chiName子字符串的字符索引

for (int i = 0; children[i] != '\0' && chiIndex < childrenNum; ++i) {

if (children[i] == ' ') {

if (charIndex > 0) { // 确保不是连续的空格

chiName[chiIndex][charIndex] = '\0'; // 终止当前字符串

chiIndex++; // 移动到下一个字符串

charIndex = 0; // 重置字符索引

}

}

else {

chiName[chiIndex][charIndex++] = children[i]; // 复制字符

}

}

// 处理最后一个字符串

if (charIndex > 0 && chiIndex < childrenNum) { chiName[chiIndex][charIndex] = '\0'; }

if (IsDuplicate(chiName))

break;

else {

std::cout << "孩子不可以重名，也不可以和祖上的人名字一样！\n";

std::cout << ">>>请重新输入所有孩子的名字：";

continue;

}

}

// 将其插入进家族树里

TreeNode<People>\* parentNode = familyTree.findNode(newMember, familyTree.getRoot());

for (int i = 0; i < childrenNum; ++i) {

newChild = People(chiName[i]);

if (i == 0) {

if (!familyTree.leftInsertNode(parentNode, newChild)) {

std::cerr << "无法插入: " << newChild.name << std::endl;

}

else

parentNode = parentNode->firstChild;

}

else {

// 其他名字作为右兄弟插入

if (!familyTree.rightInsertNode(parentNode, newChild)) {

std::cerr << "无法插入: " << newChild.name << std::endl;

}

else

parentNode = parentNode->nextSibling;

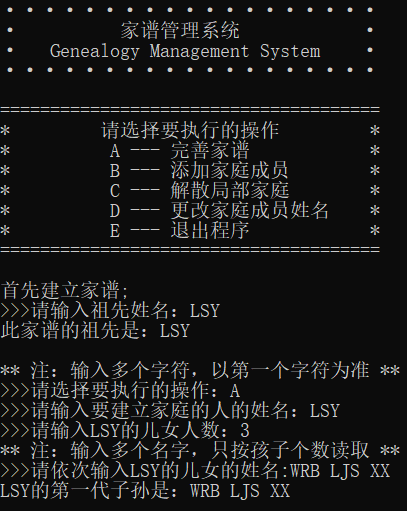
}

}

outputChildren(newMember);

}

### 完善家谱功能实现示例



## 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

这段代码是一个家谱管理系统中的功能，用于向家谱中添加新的成员。其主要步骤包括：首先获取父母的信息，然后在家谱树中定位到父母的节点。接着，系统提示用户输入新孩子的姓名，并检查这个孩子是否已经存在于家谱中。如果孩子不在家谱中，系统则将其作为父母节点的第一个孩子添加到家谱树中。最后，系统显示更新后的父母的所有孩子的信息。这个过程涉及到家谱树的搜索和节点插入操作，确保家谱的信息始终保持最新和准确。

### 3.3.2 添加家庭成员功能核心代码

// 添加家庭成员

void genealogyManagement::addMembers()

{

char childName[MAX\_SIZE];

People parent;

People newChild;

parent = inputParents(0);

TreeNode<People>\* parentNode = familyTree.findNode(parent, familyTree.getRoot());

std::cout << ">>>请输入"<<parent.name<<"新添加的儿/女的姓名[仅一个，多的不予存储]：";

while (true) {

std::cin.getline(childName, MAX\_SIZE);

newChild = People(childName);

if (familyTree.findNode(newChild, familyTree.getRoot())!=NULL) {

std::cout << "该人在本家族中！\n";

std::cout << ">>>请重新输入：\n";

}

else break;

}

if (!familyTree.insertAsFirstChild(parentNode, newChild)) {

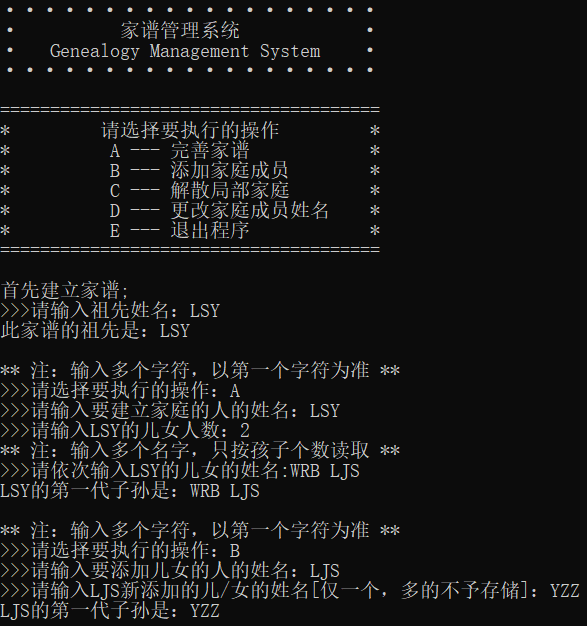
std::cerr << "无法插入: " << newChild.name << std::endl;

}

outputChildren(parent);

}

### 3.3.3 添加家庭成员功能实现示例



## 解散局部家庭功能的实现

### 3.4.1 解散局部家庭功能实现思路

这段代码是家谱管理系统中用于解散局部家庭成员的功能。它首先允许用户输入一个家庭成员的信息，作为要解散家庭的主要成员（父母）。该程序会显示这位成员的名字，并列出其所有子女的信息。随后，代码在家谱树中查找这位成员对应的节点。找到后，程序使用 familyTree.deleteChildren(parentNode) 函数来删除这个节点的所有子节点，即移除这位成员的所有直接后代，从而实现解散该家庭成员的局部家族。这个过程主要涉及到用户输入、家谱树搜索和节点的删除操作，旨在方便地管理和调整家谱树中的家族结构。

### 3.4.2 解散局部家庭功能核心代码

// 解散局部家庭成员

void genealogyManagement::dissolveFamily()

{

People parent;

parent = inputParents(0);

std::cout << "要解散家庭的人是：" << parent.name << std::endl;

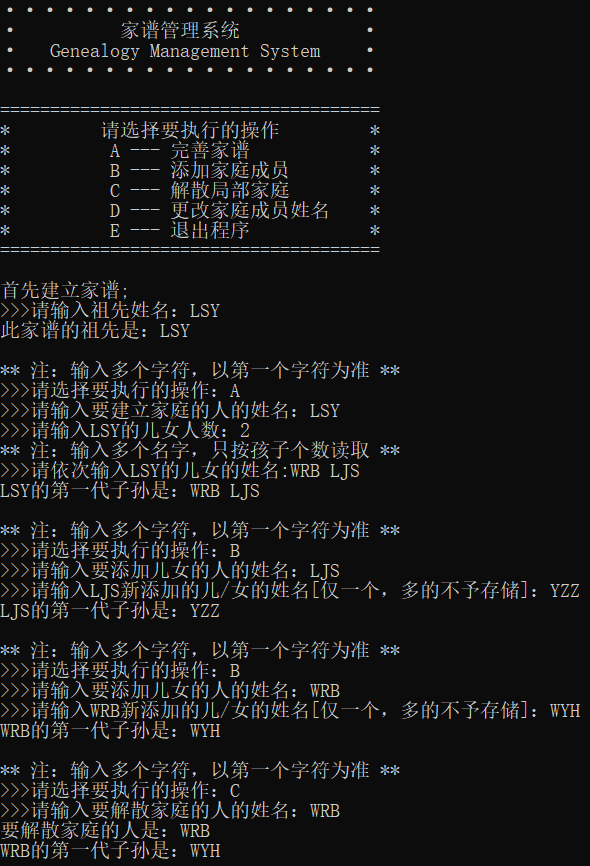
outputChildren(parent);

TreeNode<People>\* parentNode = familyTree.findNode(parent, familyTree.getRoot());

familyTree.deleteChildren(parentNode);

}

### 3.4.3 解散局部家庭功能实现示例



## 修改家庭成员信息功能的主要实现

### 3.5.1 修改家庭成员信息功能实现思路

这段代码是家谱管理系统中用于更改家庭成员姓名的功能。首先，用户通过 inputParents 函数输入要更改姓名的家庭成员（父母）信息。接着，程序提示用户输入新的姓名，并使用循环确保新输入的姓名在家族中是唯一的，即不与家族中其他成员的姓名重复。如果输入的新姓名已经存在于家族中，系统会提示用户重新输入。一旦输入了一个唯一的新姓名，程序使用 familyTree.modifyNode 方法更新家谱树中相应成员的信息，实现姓名的更改。最后，程序输出更名的结果，即显示成员的原名和新名。这个过程涉及用户交互、家谱树的搜索和节点信息的修改，以确保家族成员的姓名可以根据需要灵活更改。

### 修改家庭成员信息功能核心代码

// 更改家庭成员姓名

void genealogyManagement::changeName()

{

char changeName[MAX\_SIZE];

People parent;

People newName;

parent = inputParents(0);

std::cout << ">>请输入更改后的姓名:";

while (true) {

std::cin.getline(changeName, MAX\_SIZE);

newName = People(changeName);

if (familyTree.findNode(newName, familyTree.getRoot()) != NULL) {

std::cout << "该名在本家族中使用过！\n";

std::cout << ">>>请重新输入：\n";

}

else break;

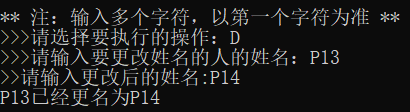
}

familyTree.modifyNode(parent, newName, familyTree.getRoot());

std::cout << parent.name << "已经更名为" << newName.name << std::endl;

}

### 修改家庭成员信息功能实现示例



## Tree.h中主要功能的实现

### 3.6.1 Tree.h头文件中主要包含的类设计

定义了二叉树结点TreeNode，每个节点包含值（value），指向第一个孩子（firstChild）和下一个兄弟（nextSibling）的指针。定义了LinkedQueue作为用于层次遍历二叉树的链队列，包含队列的头（front）和尾（rear）指针，以及相关的队列操作函数（如 enqueue、dequeue）。定义了二叉树的主体MyBinaryTree，表示二叉树的主体，包含根节点指针（root）以及一系列操作二叉树的函数，如 copy、destroy、isEmpty、getHeight、getSize 等。提供树的遍历方法，包括前序、中序、后序和层次序遍历。包含节点的查找、插入和删除操作。

### Tree.h头文件核心代码

详见类设计。

# 项目测试

## 功能测试

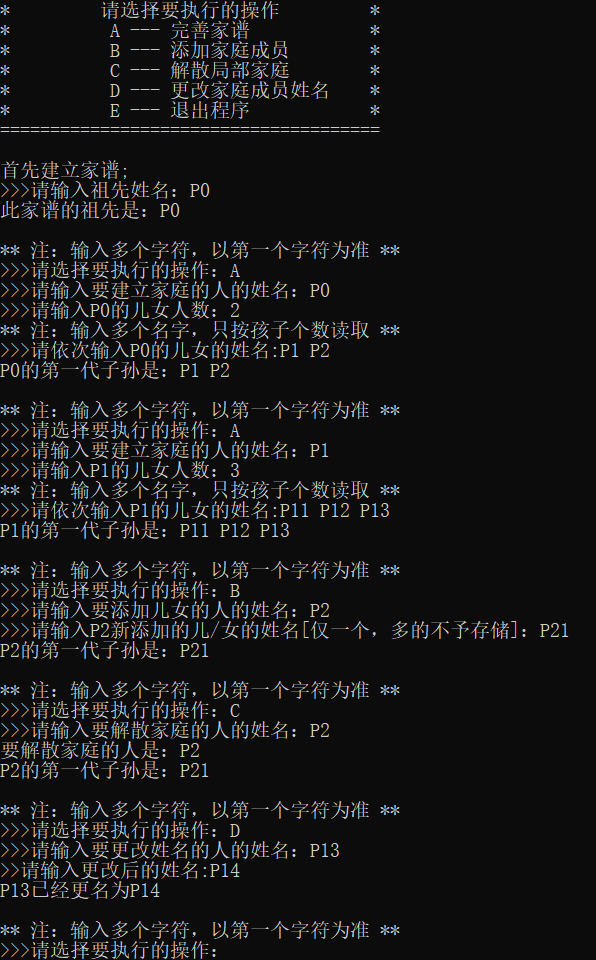
### 基本功能测试

**测试用例：**依照demo

**用例说明：**按照老师给的word进行测试

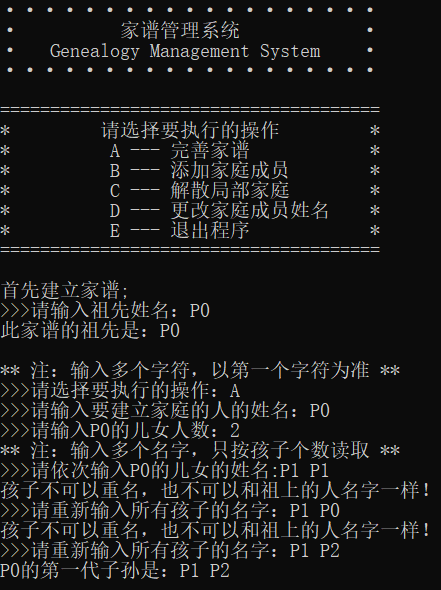
**预期结果：**程序应该正常运行不崩溃，完成每一项任务

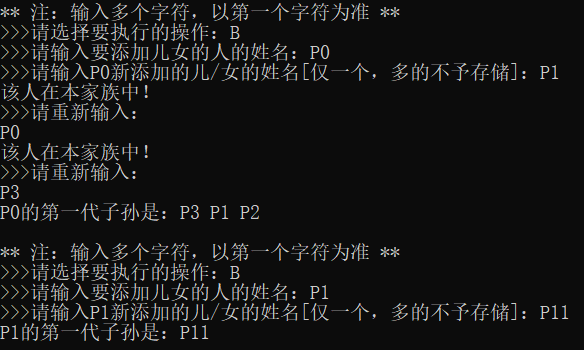
**实验结果：**



### 测试重复名字输入

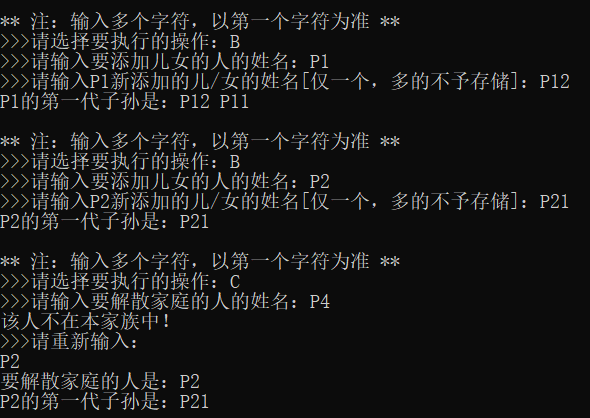
**实验结果：**





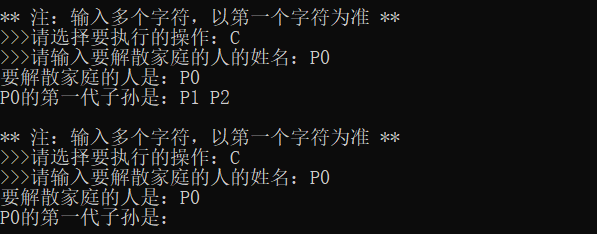
### 测试解散不存在的家庭

**实验结果：**



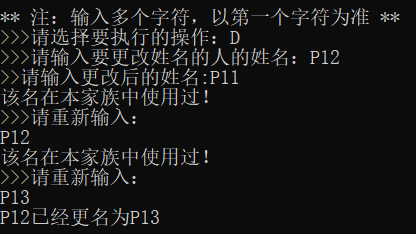
### 测试解散祖先

**实验结果：**



### 测试修改的名字重复

**实验结果：**

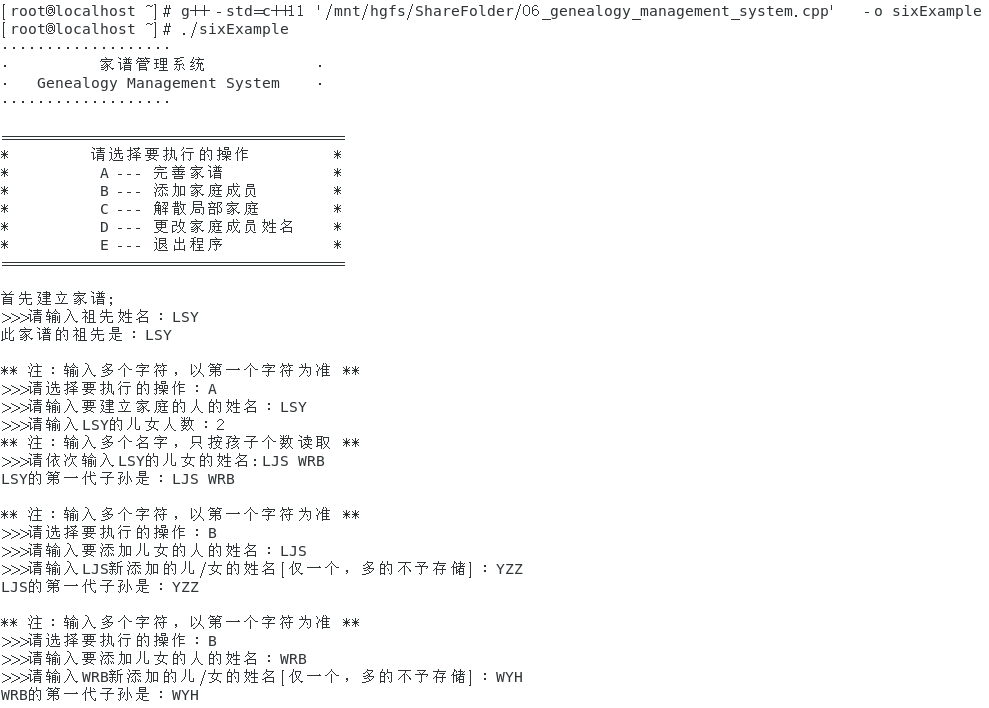


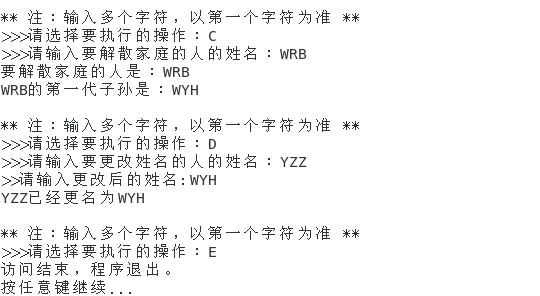
## Linux环境测试

**编译命令：** g++ -std=c++11 '/mnt/hgfs/ShareFolder/06\_genealogy\_management\_system' -o sixExample

**运行命令：** ./sixExample

**实验结果：**





1. **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

Linux系统：CentOS 7 x64