项目说明文档

数据结构课程设计

——家谱管理系统



作 者 姓 名： 刘相成

学 号： 2452207

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

# 1 项目分析

## 背景分析

## 家谱作为记录家族血缘关系与重要人物事迹的特殊图书体裁，是中国珍贵的文化遗产，与国史、地志并称为中华民族的三大文献。它在历史学、民俗学、人口学等多个学科的研究中有着不可替代的作用。本项目旨在模拟家谱管理的核心流程，实现家族成员信息的查看、插入、删除等操作，为用户提供一个便捷、高效的家谱管理工具。

## 1.2 功能分析

1.2.1功能要求

本项目核心是完成家谱成员信息的建立、查找、插入、修改、删除等一系列操作。首先定义家族成员的数据结构，将每个功能封装为对应的成员函数，通过函数调用实现对家谱数据的操作，最后通过主函数搭建用户交互界面，验证各个功能的正确性并输出运行结果。

1.2.2输入要求

用户通过输入指定字符选择对应操作，如输入 A 完善家谱、输入 B 添加成员等。在执行具体操作时，需按照提示输入相关数据，例如初始化家谱时输入祖先姓名，完善家谱时输入家长姓名、子女数量及子女姓名等。输入需符合规范，姓名不含空格且为单个，数字需为正整数等。

1.2.3输出要求

每次执行操作后，系统会输出对应的操作结果。例如初始化家谱后，输出祖先姓名；添加成员后，输出该成员的所有子女姓名；查找亲属后，输出指定代际的祖先或子孙姓名等。同时，提供家谱树状图的显示功能，直观呈现整个家族的层级关系。

1.2.4项目实例



# 2 项目设计

## 2.1 数据结构设计

本家谱管理系统采用树结构作为核心数据结构，用于表示家族成员之间的层级血缘关系。树结构属于非线性数据结构，由节点和边构成，每个节点代表一位家族成员，每个节点可以拥有多个子节点，对应该成员的子女，且每个节点仅有一个父节点，这种结构能够完美契合家族成员间的亲属关系。

该项目数据结构设计的特点和优势如下：

层级关系直观：树结构可以清晰地展现家族成员的父子关系，让家族的繁衍脉络一目了然。

操作灵活便捷：基于树结构可以轻松实现成员的添加、删除等操作，同时能快速更新整个树结构，适应家族成员变化的需求。

递归操作高效：借助递归算法，能够便捷地在树结构中进行成员查找、家谱遍历等操作，简化了代码实现，提高了开发效率。

## 2.2 类结构设计

## 2.2.1 节点表示 (Member 结构体)

## Member 结构体用于表示家谱中的一个成员，相当于树结构中的一个节点。该结构体包含成员姓名、子节点数组指针、当前子节点数量和子节点数组容量等成员变量，完整记录了成员自身信息及其子女相关情况。

## 2.2.2 树结构表示 (Tree 类)

## Tree 类封装了家谱树的所有操作，包含根节点指针作为私有成员，根节点即家谱的祖先。该类提供了初始化家谱、完善家谱、添加成员、解散家庭等一系列公有的成员函数，同时包含递归查找成员、扩展子节点容量等私有辅助函数，实现了对家谱的全方位管理。

## 2.3 成员与函数设计

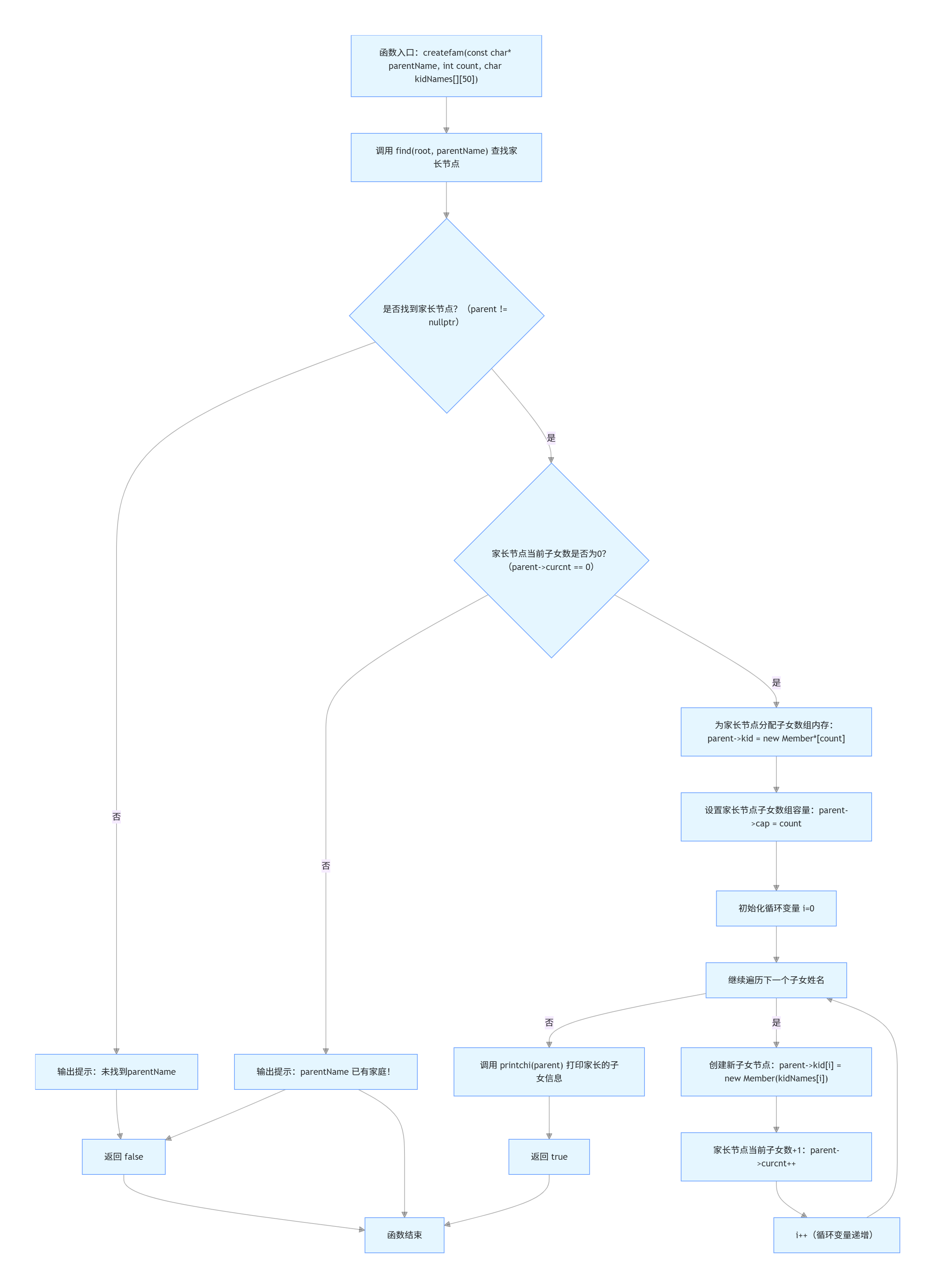
// 成员节点  
struct Member {  
    char name[50];// 成员姓名  
    Member\*\* kid;// 节点数组指针  
    int curcnt;// 当前节点数量  
    int cap;// 节点数组容量  
    Member(const char\* n);// 构造函数  
};  
// 家谱树类  
class Tree {  
private:  
    Member\* root;// 根节点  
    Member\* find(Member\* current, const char\* name);// 查找成员  
    void expand(Member\* parent); // 扩展子节点数组容量  
    void remove(Member\* node);// 释放内存  
    bool findanc(Member\* current, const char\* targetname,Member\*& anc, int& curgen, int targen); // 查找距离己辈第n代祖先  
    void finddes(Member\* member, int curgen, int targen);// 查找第n代子孙  
    void showshu(Member\* node, int level, string prefix);// 显示子树  
​  
public:  
    Tree();// 构造函数  
    ~Tree();// 析构函数  
    bool exists(const char\* name);// 检查姓名是否存在  
    void init(const char\* ancName);// 初始化家谱  
    bool createfam(const char\* parentName, int count,char kidNames[][50]);// 完善家谱  
    bool addone(const char\* parentName, const char\* childName); // 添加单个子女  
    bool dissolve(const char\* memberName);// 解散局部家庭  
    bool change(const char\* oldName, const char\* newName);// 更改家庭成员姓名  
    bool findrel(const char\* name, int ancGen, int descendantGen); // 查找亲属  
    Member\* findpar(Member\* current, const char\* name);// 查找父节点  
    void printchi(Member\* member);// 打印成员及其子孙  
    void show();// 显示家谱图  
};

## 2.4 代码总设计（流程图）

# 3 项目实现

## 3.1 完善家谱功能的实现

### 3.1.1 完善家谱功能流程图

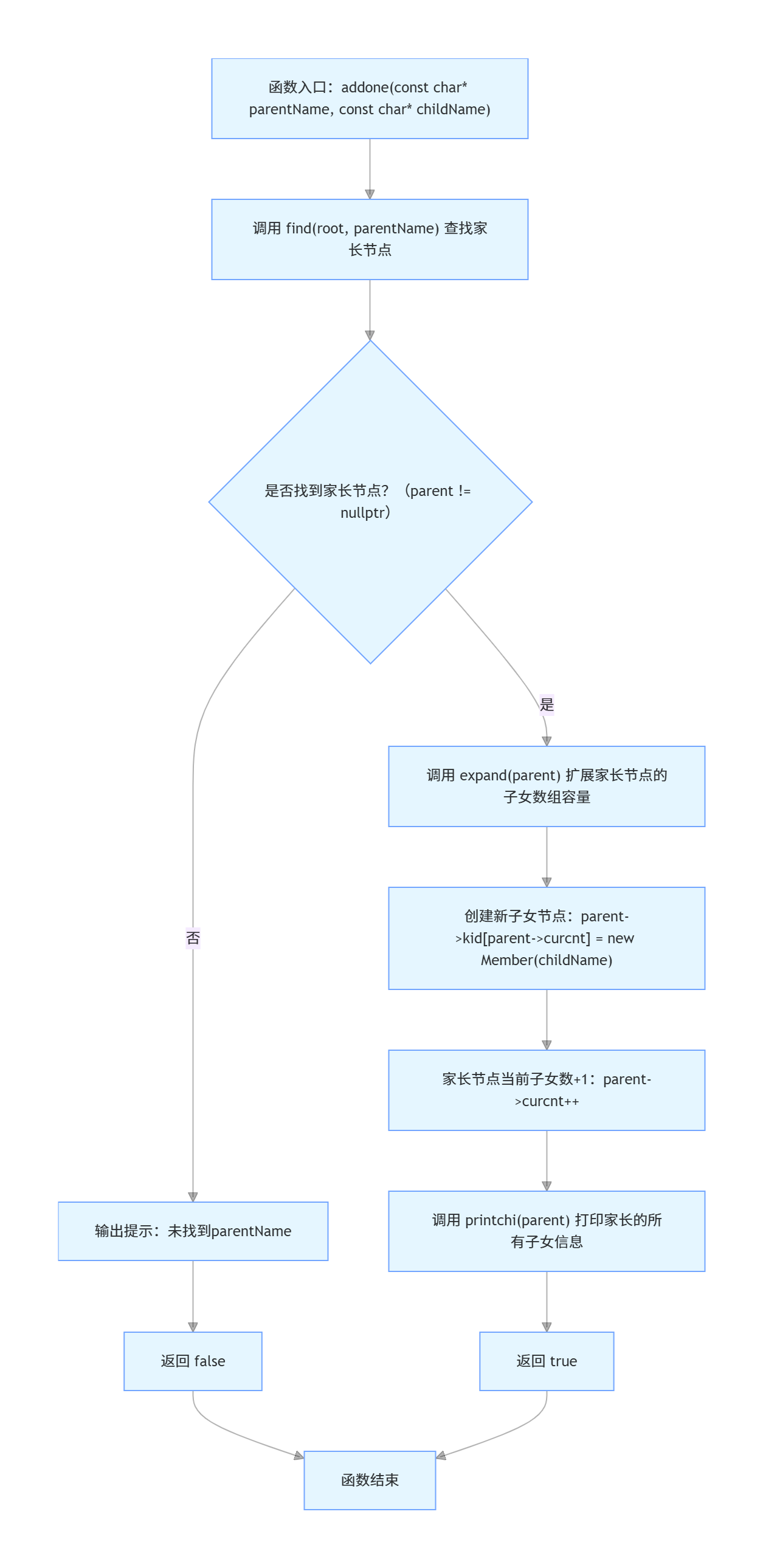


### 3.1.2 完善家谱功能核心代码

// 完善家谱  
bool Tree::createfam(const char\* parentName, int count,  
    char kidNames[][50]) {  
    Member\* parent = find(root, parentName);  
    if (parent == nullptr) {  
        cout << "未找到" << parentName << endl;  
        return false;  
   }  
    if (parent->curcnt > 0) {  
        cout << parentName << " 已有家庭！" << endl;  
        return false;  
   }  
    parent->kid = new Member \* [count];  
    parent->cap = count;  
    for (int i = 0; i < count; i++) {  
        parent->kid[i] = new Member(kidNames[i]);  
        parent->curcnt++;  
   }  
    printchi(parent);  
    return true;  
}

## 3.2 添加单个子女功能的实现

### 3.2.1 添加单个子女功能流程图

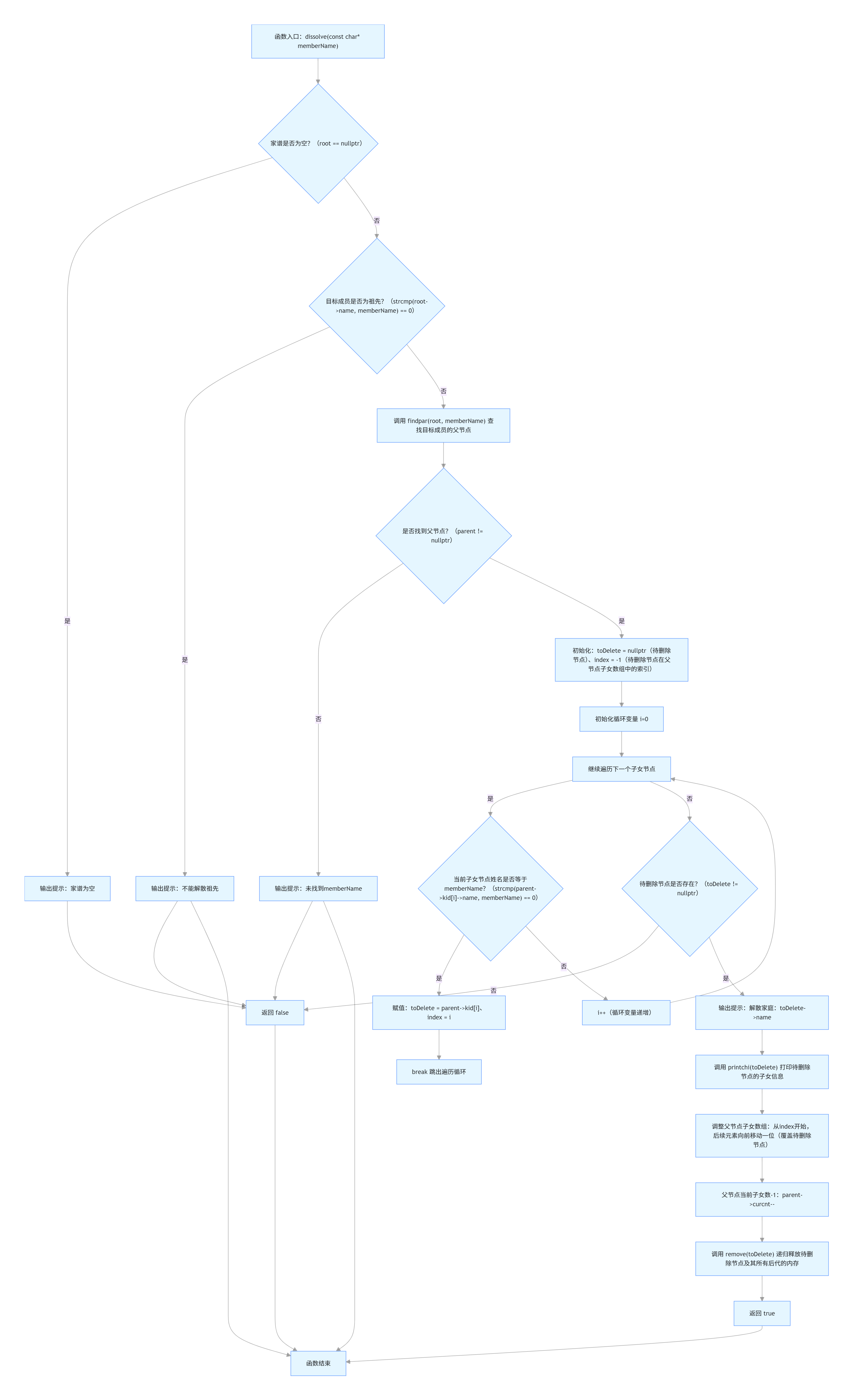


### 3.2.2 添加单个子女功能核心代码

// 给指定成员添加一个子女  
bool Tree::addone(const char\* parentName, const char\* childName) {  
    Member\* parent = find(root, parentName);  
    if (parent == nullptr) {  
        cout << "未找到" << parentName<< endl;  
        return false;  
   }  
    expand(parent);  
    parent->kid[parent->curcnt] = new Member(childName);  
    parent->curcnt++;  
    printchi(parent);  
    return true;  
}

## 3.3 解散局部家庭功能的实现

### 3.3.1 解散局部家庭功能流程图

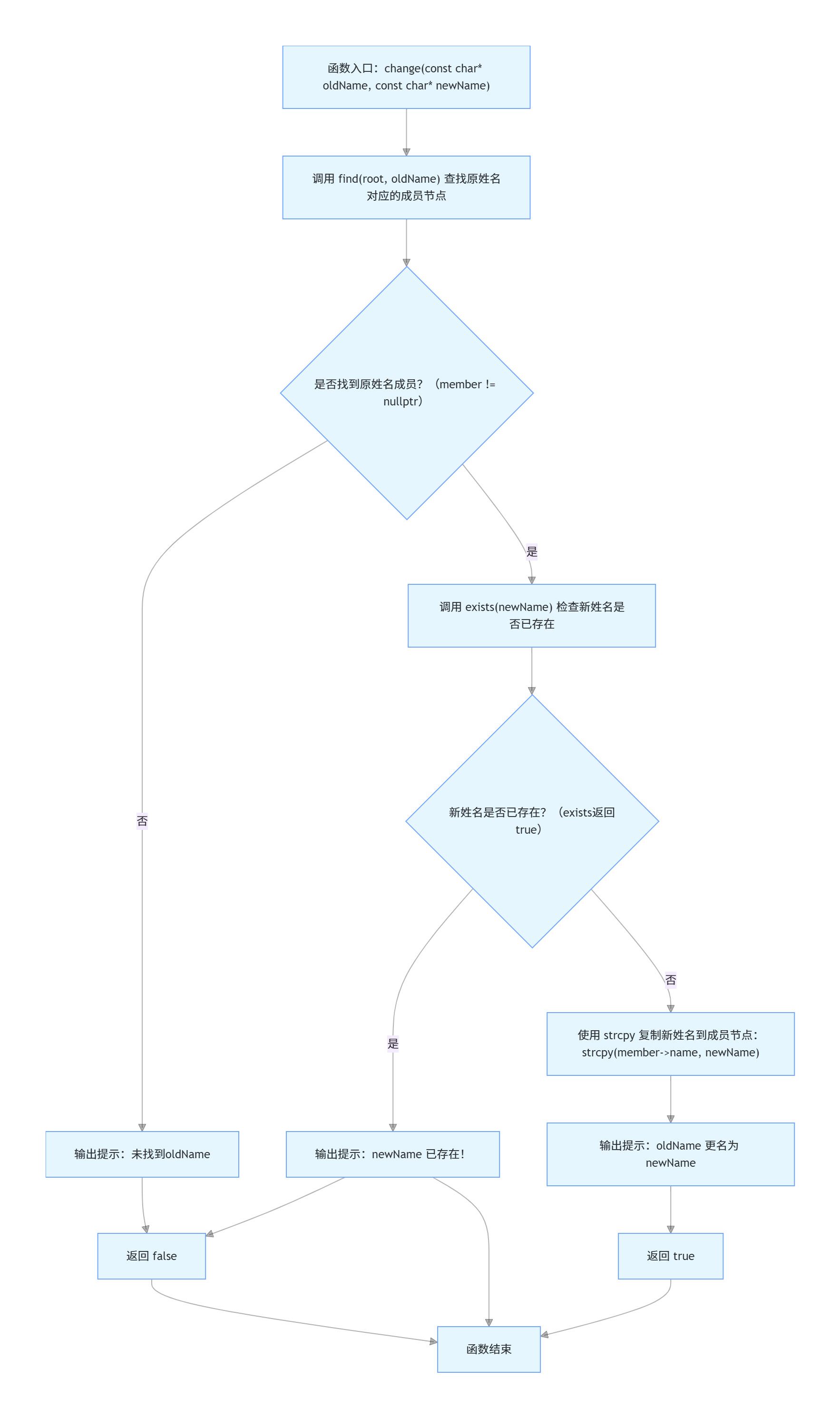


### 3.3.2 解散局部家庭功能核心代码

// 解散局部家庭  
bool Tree::dissolve(const char\* memberName) {  
    if (root == nullptr) {  
        cout << "家谱为空" << endl;  
        return false;  
   }  
    if (strcmp(root->name, memberName) == 0) {  
        cout << "不能解散祖先" << endl;  
        return false;  
   }  
    Member\* parent = findpar(root, memberName);  
    if (parent == nullptr) {  
        cout << "未找到" << memberName << endl;  
        return false;  
   }  
    Member\* toDelete = nullptr;  
    int index = -1;  
    for (int i = 0; i < parent->curcnt; i++) {  
        if (strcmp(parent->kid[i]->name, memberName) == 0) {  
            toDelete = parent->kid[i];  
            index = i;  
            break;  
       }  
   }  
    if (toDelete != nullptr) {  
        cout << "解散家庭：" << toDelete->name << endl;  
        printchi(toDelete);  
​  
        for (int i = index; i < parent->curcnt - 1; i++) {  
            parent->kid[i] = parent->kid[i + 1];  
       }  
        parent->curcnt--;  
        remove(toDelete);  
        return true;  
   }  
    return false;  
}

## 3.4 更改成员姓名功能的实现

### 3.4.1 更改成员姓名功能流程图

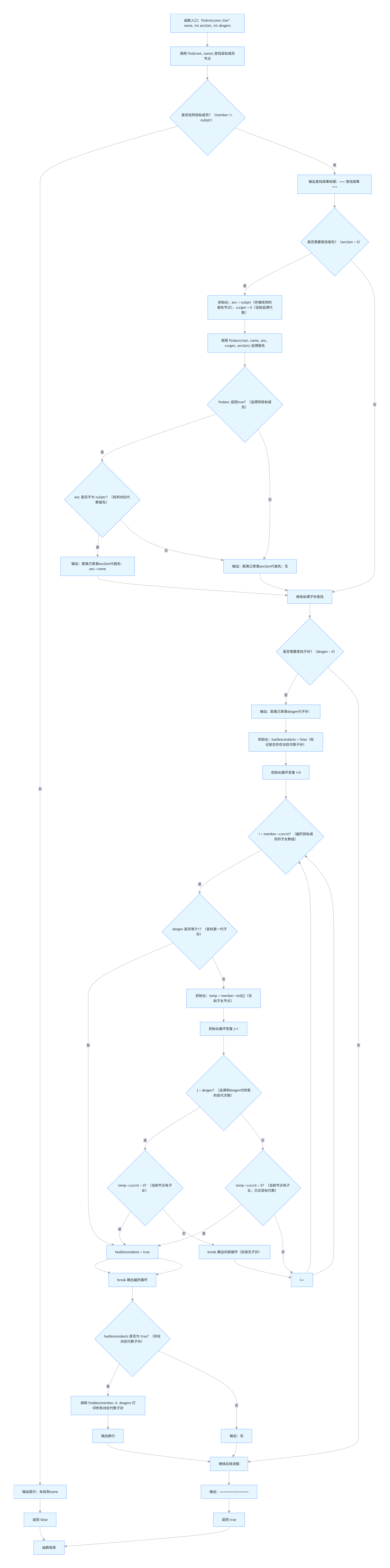


### 3.4.2 更改成员姓名功能核心代码

// 更改成员姓名  
bool Tree::change(const char\* oldName, const char\* newName) {  
    Member\* member = find(root, oldName);  
    if (member == nullptr) {  
        cout << "未找到" << oldName << endl;  
        return false;  
   }  
    if (exists(newName)) {  
        cout << newName << " 已存在！" << endl;  
        return false;  
   }  
    strcpy(member->name, newName);  
    cout << oldName << "更名为" << newName << endl;  
    return true;  
}

## 3.4 查找亲戚功能的实现

### 3.4.1 查找亲戚功能流程图



### 3.4.2 查找亲戚功能核心代码

// 查找亲戚  
bool Tree::findrel(const char\* name, int ancGen, int desgen) {  
    Member\* member = find(root, name);  
    if (member == nullptr) {  
        cout << "未找到" << name  << endl;  
        return false;  
   }  
    cout << "\n=== 查找结果 ===" << endl;  
    if (ancGen > 0) {  
        Member\* anc = nullptr;  
        int curgen = 0;  
        if (findanc(root, name, anc, curgen, ancGen)) {  
            if (anc != nullptr) {  
                cout << "距离己辈第" << ancGen << "代祖先：" << anc->name << endl;  
           }  
            else cout << "距离己辈第" << ancGen << "代祖先：无" << endl;  
​  
       }  
        else cout << "距离己辈第" << ancGen << "代祖先：无" << endl;  
   }  
    if (desgen > 0) {  
        cout << "距离己辈第" << desgen << "代子孙：";  
        bool hasDescendants = false;  
        for (int i = 0; i < member->curcnt; i++) {  
            if (desgen == 1) {  
                hasDescendants = true;  
                break;  
           }  
            else {  
                Member\* temp = member->kid[i];  
                for (int j = 1; j < desgen; j++) {  
                    if (temp->curcnt > 0) {  
                        hasDescendants = true;  
                        break;  
                   }  
                    if (temp->curcnt == 0) break;  
                    temp = temp->kid[0];  
               }  
                if (hasDescendants) break;  
           }  
       }  
        if (hasDescendants) {  
            finddes(member, 0, desgen);  
            cout << endl;  
       }  
        else cout << "无" << endl;  
   }  
    cout << "================" << endl;  
    return true;  
}

# 4 项目测试

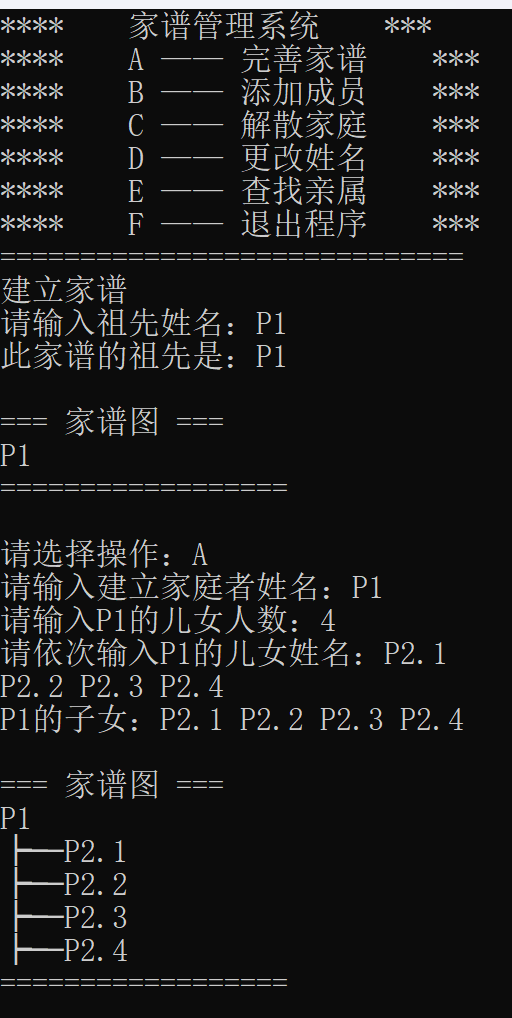
## 4.1 功能测试

### 4.1.1 建立与完善家谱测试

**测试用例**：P1|4 P2.1 P2.2 P2.3 P2.4

**预期结果**：P1有四个孩子

**实验结果**

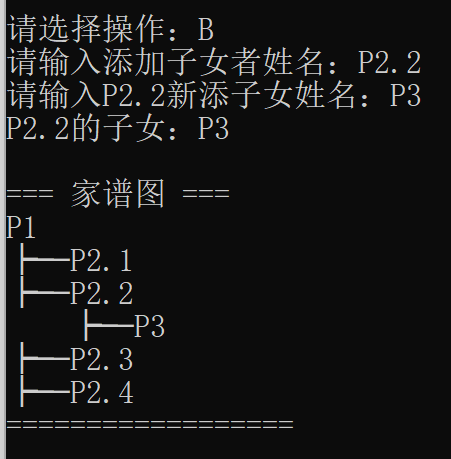


### 4.1.2 添加成员测试

**测试用例：P2.2|P3**

**预期结果：P2.2的孩子是P3**

**实验结果：**

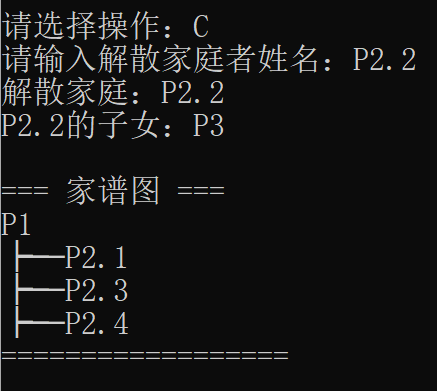


### 4.1.3 解散家庭测试

**测试用例：P2.2**

**预期结果：P2.2与P3消失**

**实验结果：**

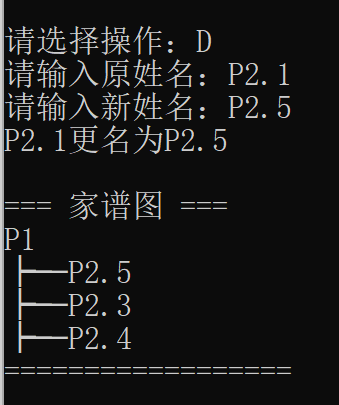


### 4.1.4 更改姓名测试

**测试用例：P2.1->P2.5**

**预期结果：P2.1消失，P2.5出现**

**实验结果：**

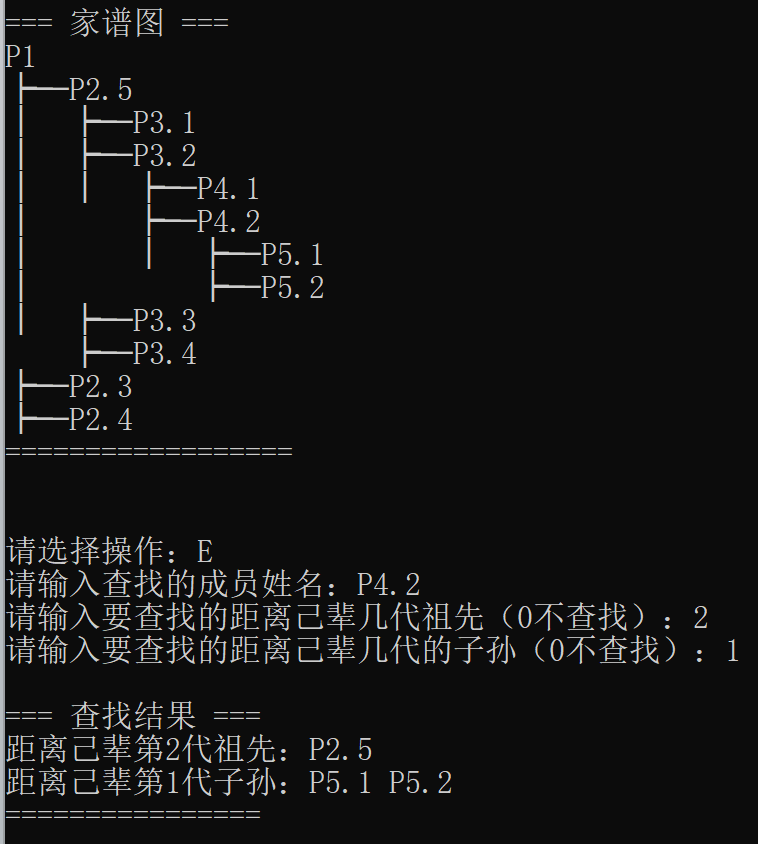
****

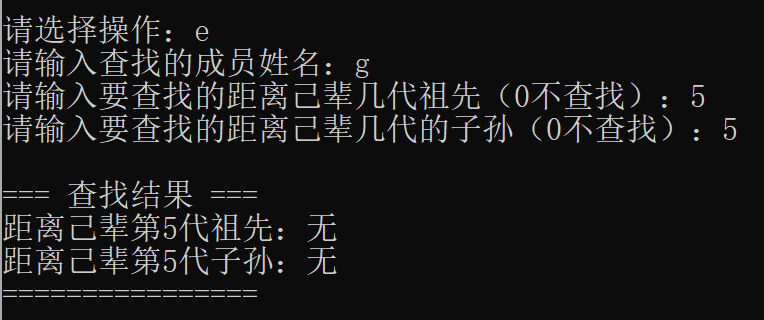
### 4.1.5 查找亲属测试

**测试用例：P4.2|2|1**

**预期结果：P2.5|P5.1 P5.2**

**实验结果：**

****

****

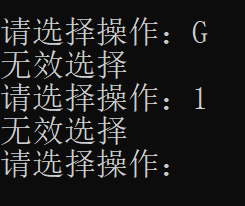
## 4.2 错误测试

### 4.2.1 选择操作不合要求

**测试用例：G|1**

**预期结果：**错误提示，运行正常不崩溃。

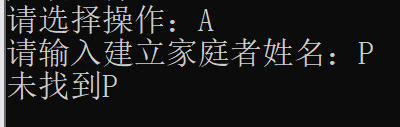
**实验结果：**

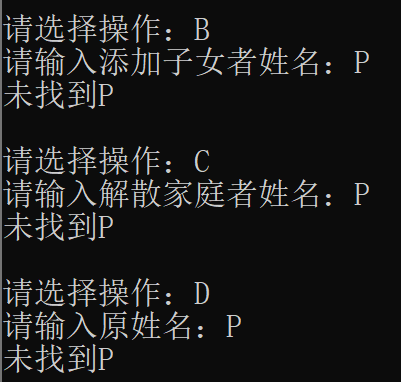


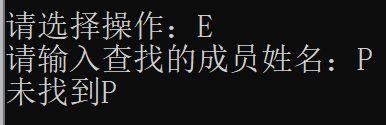
### 4.2.2 未找到成员

**测试用例：P**

**预期结果：**错误提示，程序运行正常不崩溃。**实验结果：**



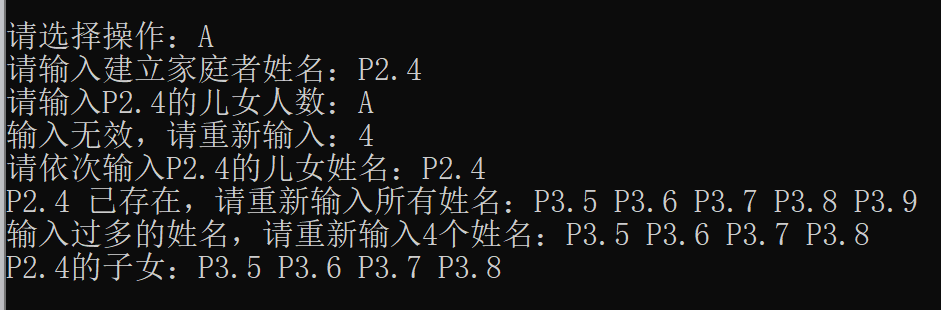


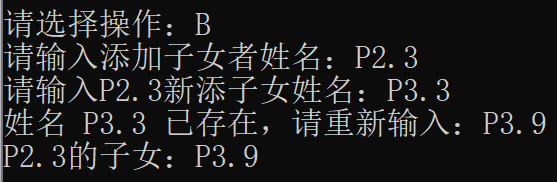


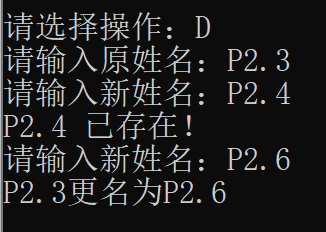
### 4.2.2 人数非法或超范围，输入已存在，人数与输入数量不符

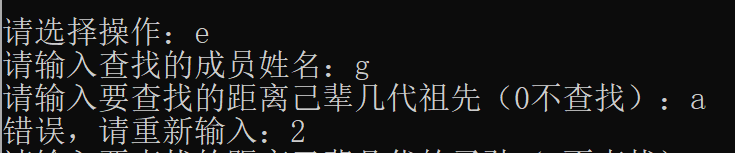
**测试用例：如截图所示**

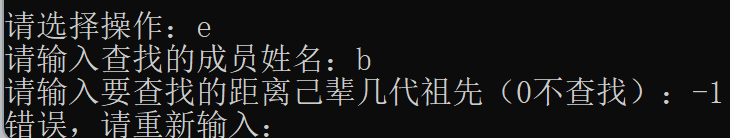
**预期结果：**错误提示，程序运行正常不崩溃。**实验结果：**











# 5 项目心得

完成这个银行业务模拟项目，让我实实在在学到了不少实用的编程知识，也感受到了把课本上的理论用到实际中的乐趣。

一开始我对队列的理解只停留在先进先出 这个概念上，做项目的时候才发现，队列在模拟排队场景里太好用了。顾客按顺序进来，按规则分配到 A、B 窗口，不用手动去调整顺序，入队和出队操作就能把顺序理顺。而且这次用的链式存储比之前学的数组存储方便很多，不用一开始就定好容量，顾客多了就自动加节点，不会浪费内存，也不用担心不够用，这让我明白选对数据结构能省不少事。

这次写模板队列类是个挺大的收获。以前写代码都是针对固定类型，这次用模板后，这个队列不仅能存顾客编号的整数，以后要是想存其他类型的数据也能用，不用再重新写一遍队列的代码。虽然一开始写模板函数的时候经常出错，但慢慢调试下来，终于搞懂了泛型编程的思路，也知道了这种写法能让代码复用性更高。

还有就是输入验证这块，让我知道了写程序不能只考虑正常情况。刚开始写的版本，只要输入字母或者负数，程序就直接崩溃了。后来查了资料，加上了判断输入是否为整数、编号是否为正数的逻辑，还有清空输入缓冲区的操作，这样就算用户输错了，程序也会提示重新输入，不会直接卡住。这让我明白，一个好用的程序不仅要能完成功能，还得考虑用户可能犯的错误。

项目里还有些小细节特别磨人，比如输出的时候不能有多余的空格。一开始我没注意，最后一位顾客后面总带着个空格，后来想到用剩余顾客数来判断，不是最后一位才输出空格，这个问题才解决。这让我觉得，写代码不能粗心，每个小需求都得考虑到，不然程序看着功能实现了，却不符合要求。

调试代码的时候也遇到过不少问题，比如出队后队尾指针没更新，导致后续入队出错，还有队列清空后再操作会报错。后来我就一步步打印关键变量的值，慢慢找到问题所在，最后把这些漏洞都补上了。这个过程虽然麻烦，但让我学会了怎么排查程序里的错误，而不是遇到问题就慌了手脚。

总的来说，这个项目让我把数据结构和 C++ 编程的知识好好巩固了一遍。以前觉得抽象的知识点，通过这次实践都变得很直观。以后再遇到类似的排队、调度问题，我就能更快地想到解决方案，也会更注重程序的实用性和稳定性。