项目说明文档

数据结构课程设计

——家谱管理系统

作 者 姓 名：陈垲昕

学　　号：[redacted]

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

可通过按住 Ctrl 并单击访问说明文档各个模块：

1. 分析

1.1 背景分析

家谱是一种以表谱形式，记载一个以血缘关系为主体的家族世袭繁衍和重要任务事迹的特殊图书体裁。家谱是中国特有的文化遗产，是中华民族的三大文献（国史，地志，族谱）之一，属于珍贵的人文资料，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。本项目兑对家谱管理进行简单的模拟，以实现查看祖先和子孙个人信息，插入家族成员，删除家族成员的功能。

1.2 功能分析

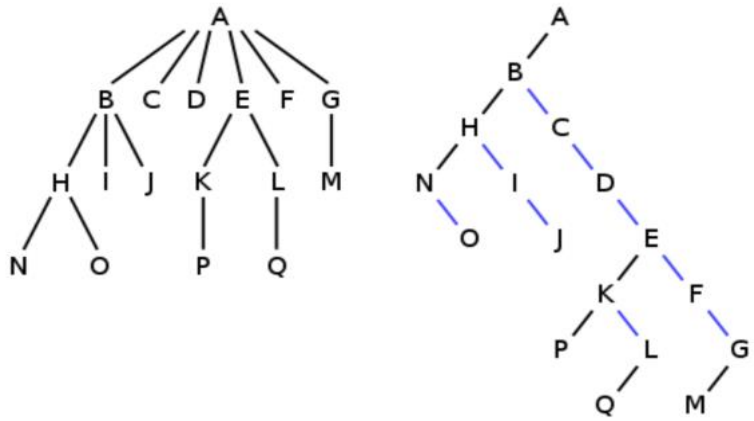
本项目的实质是完成兑家谱成员信息的建立，查找，插入，修改，删除等功能，可以首先定义家族成员数据结构，然后将每个功能作为一个成员函数来完成对数据的操作，最后完成主函数以验证各个函数功能并得到运行结果。

2.设计

2.1 主要数据结构设计

在家谱中，一个家庭成员可能有多个孩子，且他的孩子也可能将生育多个孩子。其中，每一代的父亲和后代形成一对多的关系，但是一个后代不能对应多个父母。这是典型的树结构。 我们使用树数据结构将家族关系存储在家族树中。

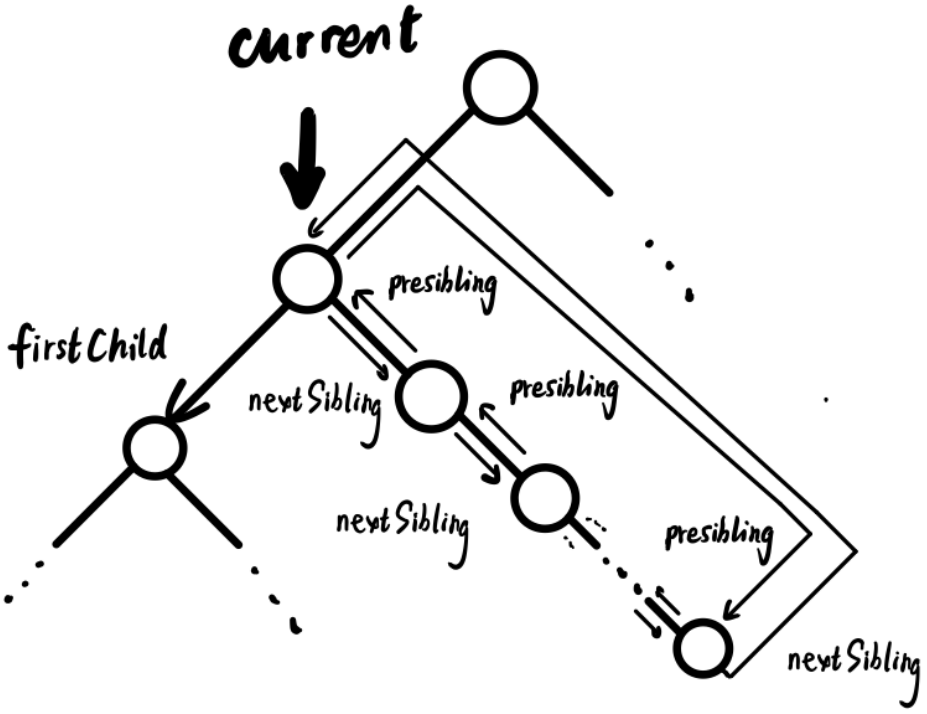
考虑到父节点可能对应多个孩子，此时无法完全确定子节点的数目，使用普通的二叉树的方法将无法呈现一对多的父子关系，故我们我们考虑对家谱树使用“左长子，右兄弟”规则，将多叉树标准化为二叉树：每个父节点的左孩子是它的第一个子节点，右孩子是自己的兄弟节点，该数据结构可完成对家谱树信息的有效存储。



当然，对于维护这样的家谱树，诸如搜索/插入/删除之类的基本操作往往会花费较长的时间，因为要找到合适的位置，我们必须遍历搜索/插入/删除的节点的所有同级节点。

1.树的结点结构体设计

模板类树节点 **TreeNode** 是家谱树的结点单元，数据成员包括自定义类型的 **data** 表示成员的名字，以及三个指针域*** firstChild**（左子节点）、*** nextSibling**（下一个兄弟节点）以及*** preSibling**（前一个兄弟节点），加入*** preSibling** 指针域的设计，使得兄弟节点间形成循环双向链表连接，这样设计主要为了可以在快速定位某父节点的最后一个孩子(通过 **firstChild->preSibling** 的方式得到)。



该模板类的方法成员除了构造函数之外，还定义了对节点所有的第一代子节点遍历并打印的方法。

代码说明

```
template<class T>
struct TreeNode {
    //数据成员
    T data;
    //左子节点，下一个兄弟节点和前一个兄弟节点,兄弟节点间形成循环双向链表连接
    TreeNode<T>* firstChild, * nextSibling, * preSibling;
    //记录孩子的数量，在FamilyTree中更新
    int childNumber;
    //方法成员:构造函数
    TreeNode(T value = 0, TreeNode<T>* fc = nullptr) :
        data(value), firstChild(fc), nextSibling(nullptr),preSibling(nullptr),childNumber(0) {}
    //遍历所有孩子并打印
    void printChildrenName()const;
};
```

2.家谱树类设计

家谱树类 **FamilyTree** 是程序的主体部分，包括两个 string 类型的 **TreeNode***指针，分别表示根节点与当前节点，关键成员见下

```
//成员：根节点与当前节点
TreeNode<string>* _root;
TreeNode<string>* _current;

//设置方法：
inline bool setCurrentAsRoot(); //设置当前节点为子节点，使用于所有的查找函数中

//查找办法:当前节点的第一个子节点与最后一个子节点
TreeNode<string>* currentFirstChild()const { return _current->firstChild; }
TreeNode<string>* currentLastChild()const { return _current->firstChild->preSibling; }

//_current节点的操作：添加孩子，删除孩子（所有子孙）
bool currentAddChildren(vector<string>& childVec);
void currentDeleteChildren(TreeNode<string>* pTempRoot);

//查找并更新当前节点的操作
bool findCurrentFirstChild();        //查找当前节点的长子节点并更新当前节点为长子节点
bool findCurrentNextSibling();       //查找当前节点的下一个兄弟节点并更新当前节点为下一个兄弟节点
bool findPerson(string name);                                    //查找特定名字的家庭成员
bool findPerson(TreeNode<string>* pTempRoot, string name);  //以pTempRoot为根节点查找家庭成员
//......
```

3.核心功能实现

3.1 家谱的建立

主要实现函数:**void FamilyTree::initFamilyTreeWithAncestor()**

通过初始化一个祖先的姓名建立家谱树，设定根节点与当前节点为该新建的以 **ancenstorName** 为数据的祖先节点

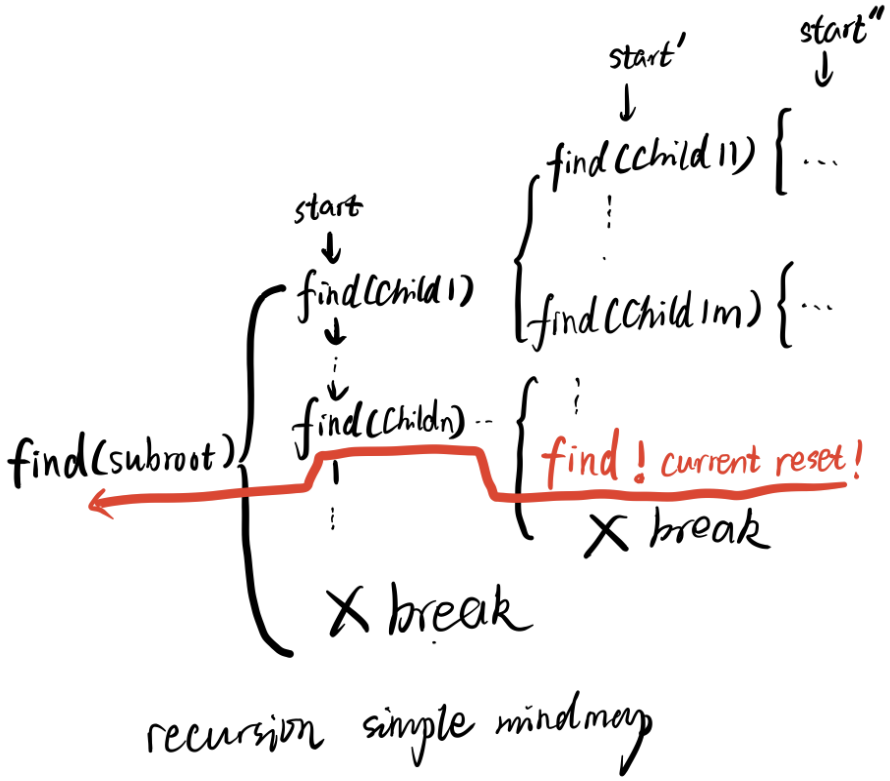
```
void FamilyTree::initFamilyTreeWithAncestor()
{
    string ancestorName;
    std::cout << "首先建立一个家谱" << std::endl;
    std::cout << "请输入祖先的姓名：";
    std::cin >> ancestorName;
    createRoot(ancestorName);
    std::cout << "此家谱的祖先是：" << ancestorName << std::endl;
}

FamilyTree::createRoot(string name)
    { _root = _current = new TreeNode<string>(name); assert(_root); }
```

3.2 查找家庭成员

主要实现函数:**bool FamilyTree::findPerson(TreeNode* pTempRoot, string name)**

此函数采用递归写法：终止条件为 **pTempRoot** 指针的数据域与所要查找的 **name** 相同，并更新当前节点为所找到的结点；在递归主体中，对当前 **pTempRoot** 的所有孩子遍历，以他们自己的根节点递归调用 **findPerson** 函数，并传递是否查找到的结果赋值给布尔变量 **personFinded**，根据找到与否的结果返回 bool 值。

递归说明



代码说明

```
bool FamilyTree::findPerson(TreeNode<string>* pTempRoot, string name)
{
    bool personFinded = false;
    //找到节点，更新当前节点为所找到的结点
    if (pTempRoot->data == name)
    {
        personFinded = true;
        _current = pTempRoot;
    }
    else
    {
        TreeNode<string>* pTemp = pTempRoot->firstChild;
        //遍历，循环终止条件为：搜索完所有的子节点，或者对某个子节点搜索时personFinded=true
        int counter = 0;
        while (pTemp && !(personFinded = findPerson(pTemp, name))&&counter!=pTempRoot->childNumber)
        {
            pTemp = pTemp->nextSibling;
            counter++;
        }

    }

    return personFinded;

}
```

3.3 家谱的完善

主要实现函数:**void FamilyTree::A_updateFamilyTree()**

输入需要添加子女的家庭成员,调用 **findPerson** 查找到对应节点之后，将当前节点将它们的名字存储在一个 string 类型 vector 中，传入 **currentAddChildren()**函数，为当前节点添加所有孩子到节点孩子链表的尾部

代码说明

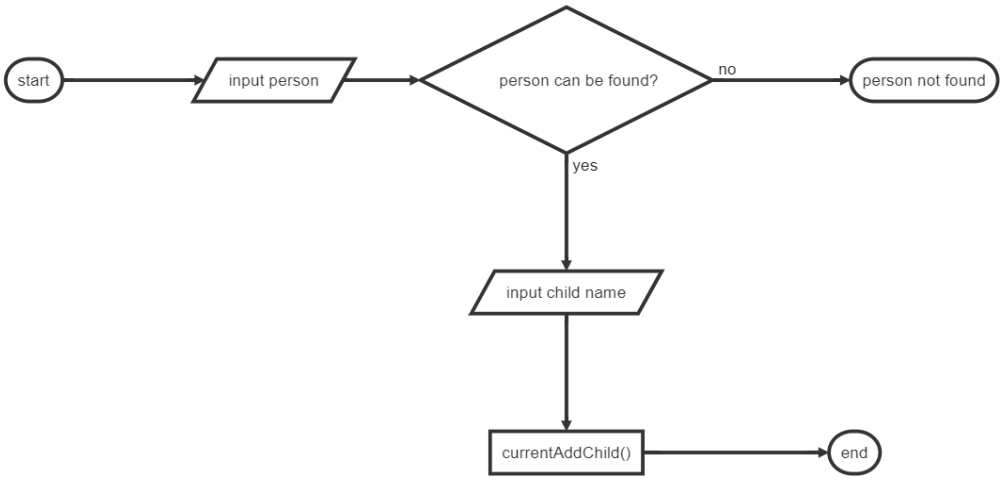
```
//决定遍历时的起始索引：如果current没有孩子，从0开始，有孩子时从1开始
int startIndex = (_current->firstChild) ? 0 : 1;

//current开始没有孩子时，用childrenVec[0]为它添加第一个孩子，此时该孩子的nextSibling与preSibling就是自身
if (!_current->firstChild&&!childrenVec.empty())
{
    TreeNode<string>* newFirstChild = new TreeNode<string>(childrenVec.at(0));
    newFirstChild->nextSibling = newFirstChild->preSibling = newFirstChild;
    _current->firstChild = newFirstChild;
}

//若current开始已经有孩子了，则从childrenVec[0]为它添加第一个孩子;
//若current开始没孩子（执行了上面的代码后），则从childrenVec[1]为它添加第一个孩子;
for (int i = startIndex; i < childrenVec.size(); i++)
{
    TreeNode<string>* newChild = new TreeNode<string>(childrenVec.at(i));
    assert(newChild);
    //直接通过循环链表回退得到当前节点的最后一个子节点，添加在家谱树尾部，并维护双向链表的完整性
    TreeNode<string>* pOriginLast = currentLastChild();
    pOriginLast->nextSibling = newChild;
    newChild->preSibling = pOriginLast;

    currentFirstChild()->preSibling = newChild;
    newChild->nextSibling = currentFirstChild();
}
```

3.4 添加成员功能的实现

主要实现函数:**void FamilyTree::B_addFamilyMember()**

输入需要添加子女的家庭成员,调用 **findPerson** 查找到对应节点后，将所要添加的孩子名传入 **currentAddChildren()**函数添加孩子

流程图说明



代码说明

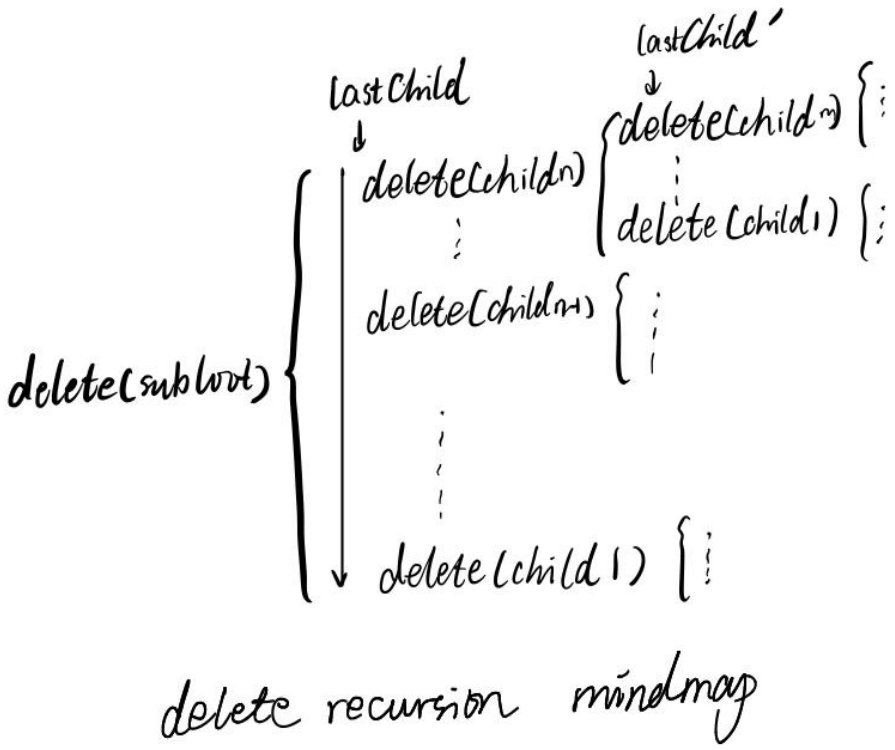
```
if (!findPerson(parentName))
{
    std::cout << "目前家谱中找不到该成员" << std::endl;
    return;
}

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

vector<string> childVec;
std::cin >> childName;
childVec.push_back(childName);
currentAddChildren(childVec);

std::cout << parentName << "的第一代子孙是：";
_current->printChildrenName();
```

3.5 解散家庭成员功能的实现

主要实现函数:**void FamilyTree::C_dismissFamilySubTree**

输入需要添加子女的家庭成员,调用 **findPerson** 查找到对应节点后，先用 **TreeNode** 方法 **printChildrenName**()打印当前节点的第一代子孙（题目要求），后调用 **currentDeleteChildren(TreeNode* pTempRoot)**函数删除：定位到当前节点的最后一个子节点，从后往前遍历，对经过的每个子节点，以子节点本身为 **pTempRoot** 递归调用 **currentDeleteChildren** 函数，直到删除了所有子节点为止。该递归调用删除了**_current** 节点的所有子代节点

递归说明



delete recursion mindmap

代码说明

```
void FamilyTree::currentDeleteChildren(TreeNode<string>* pTempRoot)
{
    if (!pTempRoot||!pTempRoot->firstChild)
    {
        return;
    }
    //定位到最后一个子节点
    TreeNode<string>* pRear = pTempRoot->firstChild->preSibling;

    for (int i = 0; i < pTempRoot->childNumber-1; i++)
    {
        //从后往前遍历，对每个子节点递归调用删除函数
        pTempRoot->firstChild->preSibling = pRear->preSibling;
        currentDeleteChildren(pRear);
        delete pRear;
        pRear = nullptr;
        pRear = pTempRoot->firstChild->preSibling;
    }
    //对最后一个子节点的处理:删除，置父节点的孩子数为0
    pTempRoot->firstChild->preSibling = pRear->preSibling;
    currentDeleteChildren(pRear);
    delete pTempRoot->firstChild;
    pTempRoot->firstChild = nullptr;

    pTempRoot->childNumber = 0;
}
```

3.6 更改家庭成员姓名功能的实现

主要实现函数:**void FamilyTree::D_changePersonName()**

输入需要更改名字的家庭成员,调用 **findPerson** 查找到对应节点后，修改对应成员的名字（代码基本同 3.2，不再赘述）

3.7 打印家谱树

主要实现函数:**void FamilyTree::printFamilyTree(TreeNode* subRoot)const**

此函数采用递归实现：递归终止条件为 **subRoot** 为空，其执行过程为：先打印自身的名字 **subRoot->data** 与**(**，后顺序遍历每个子节点，对每个子节点以其自身为根节点递归调用 **printFamilyTree** 方法，当遍历完所有子节点后再打印**)**

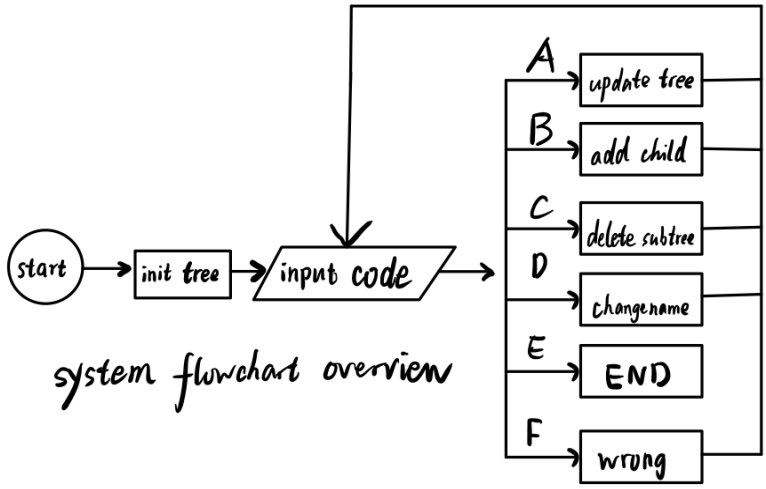
代码说明

```
//节点为空，返回
if (!subRoot)
{
    return;
}
//节点不为空：以“父节点(...所有子节点...)的格式打印树”
std::cout << subRoot->data;
if (TreeNode<string>* p=subRoot->firstChild)
{
    std::cout << " ( ";
    for (int i = 0; i < subRoot->childNumber; i++)
    {
        printFamilyTree(p);
        std::cout << " ";
        p = p->nextSibling; //遍历
    }
    std::cout << ") ";
}
```

3.8 总体系统的实现

先初始化家谱树，再根据循环输入的操作码执行相应的操作

流程说明



代码说明

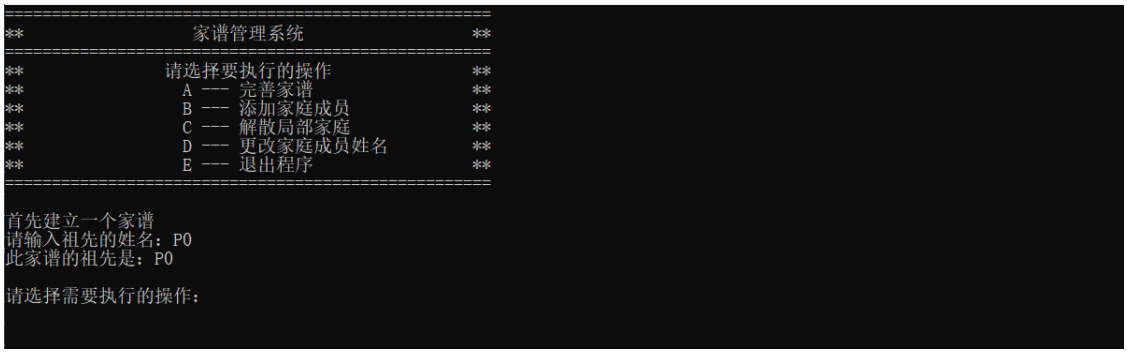
```
initFamilyTreeWithAncestor();
while (true)
{
    std::cout << std::endl;
    std::cout << "请选择需要执行的操作：";
    char operateCode;

    std::cin >> operateCode;
    switch (operateCode)
    {
        case 'A':A_updateFamilyTree(); break;
        case 'B':B_addFamilyMember(); break;
        case 'C':C_dismissFamilySubTree(); break;
        case 'D':D_changePersonName(); break;
        case 'E':std::cout << "程序已退出" << std::endl; return;
        default:std::cout << "无效的操作码，请重新输入" << std::endl; break;
    }
    std::cout << "家谱树简图：";
    printFamilyTree(_root);
    std::cout << std::endl;
}
```

4.测试

4.1 常规功能测试

Task 1 家谱的建立

实际结果：



Task 2 家谱的完善

实际结果：

Task 3 添加家庭成员功能测试

实际结果：

```
家谱树简图：P0 （ P1 P2 P3 ）

请选择需要执行的操作：B
请输入要添加儿子（或女儿）的人的姓名：P1
请输入P1新添加的儿子（或女儿）的姓名：P4
P1的第一代子孙是：P4
家谱树简图：P0 （ P1 （ P4 ） P2 P3 ）

请选择需要执行的操作：B
请输入要添加儿子（或女儿）的人的姓名：P0
请输入P0新添加的儿子（或女儿）的姓名：P6
P0的第一代子孙是：P1 P2 P3 P6
家谱树简图：P0 （ P1 （ P4 ） P2 P3 P6 ）

请选择需要执行的操作：B
请输入要添加儿子（或女儿）的人的姓名：P1
请输入P1新添加的儿子（或女儿）的姓名：P7
P1的第一代子孙是：P4 P7
家谱树简图：P0 （ P1 （ P4 P7 ） P2 P3 P6 ）
```

Task 4 解散局部家庭功能测试

实际结果：

```
家谱树简图：P0 （ P1 （ P4 P7 ） P2 P3 P6 ）

请选择需要执行的操作：C
请输入要解散的家庭的人的姓名：P1
要解散家庭的人是：P1
P1的第一代子孙是：P4 P7
家谱树简图：P0 （ P1 P2 P3 P6 ）

请选择需要执行的操作：C
请输入要解散的家庭的人的姓名：P0
要解散家庭的人是：P0
P0的第一代子孙是：P1 P2 P3 P6
家谱树简图：P0
```

Task5 更改家庭成员姓名功能测试

实际结果：

```
家谱树简图：P2

请选择需要执行的操作：D
请输入要更改姓名的人的目前姓名：P2
请输入更改后的姓名：P3
P2已更名为P3
家谱树简图：P3
```

4.2 边界测试

Task 1 对没有子女的父节点执行删除子节点操作

实际结果：

```
家谱树简图：P3

请选择需要执行的操作：C
请输入要解散的家庭的人的姓名：P3
要解散家庭的人是：P3
P3的第一代子孙是：没有孩子
家谱树简图：P3
```

4.3 错误测试

Task 1 家谱中无执行相应操作的成员

预期结果：提示找不到当前成员

实际结果：

```
家谱树简图：P3 （ P0 P1 P2 （ P4 P5 P6 P7 ） ）

请选择需要执行的操作：A
请输入要建立家庭的人的姓名：P8
目前家谱中找不到该成员
家谱树简图：P3 （ P0 P1 P2 （ P4 P5 P6 P7 ） ）

请选择需要执行的操作：B
请输入要添加儿子（或女儿）的人的姓名：P8
目前家谱中找不到该成员
家谱树简图：P3 （ P0 P1 P2 （ P4 P5 P6 P7 ） ）

请选择需要执行的操作：C
请输入要解散的家庭的人的姓名：P8
目前家谱中找不到该成员
家谱树简图：P3 （ P0 P1 P2 （ P4 P5 P6 P7 ） ）

请选择需要执行的操作：D
请输入要更改姓名的人的目前姓名：P8
目前家谱中找不到该成员
家谱树简图：P3 （ P0 P1 P2 （ P4 P5 P6 P7 ） ）
```

Task 2 完善家谱操作中输入子女数不合法

预期结果：提示需要输入正整数，并要求重新输入

实际结果：

```
家谱树简图：P3 （ P0 P1 P2 （ P4 P5 P6 P7 ） ）

请选择需要执行的操作：A
请输入要建立家庭的人的姓名：P3
请输入P3的儿女个数：-1
需输入正整数，请重新输入:1
请依次输入P3的儿女的姓名：P9
P3的第一代子孙是：P0 P1 P2 P9
家谱树简图：P3 （ P0 P1 P2 （ P4 P5 P6 P7 ） P9 ）
```

Task 3 输入的操作码不合法

预期结果：提示输入错误的操作码，并要求重新输入

实际结果：

```
家谱树简图：P3

请选择需要执行的操作：0
无效的操作码，请重新输入
家谱树简图：P3
```