数据结构课程设计项目说明文档

——家谱管理系统

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

****

目 录

[1 项目分析 1](#_Toc495668153)

[1.1 项目背景 1](#_Toc495668154)

[1.2 项目要求 1](#_Toc495668154)

[1.3 项目需求分析 2](#_Toc495668155)

[2 项目设计 3](#_Toc495668156)

[2.1 数据结构设计 3](#_Toc495668157)

[2.2 类设计 3](#_Toc495668158)

2.2.1 家族成员类（FamilyMem） 3

2.2.2 家族树类（FamilyTree） 3

2.2.3 向量类（Vector） 5

2.2.4 字符串类（String） 7

2.3 算法设计 9

2.3.1 算法思路 9

2.3.2 性能评估 10

2.3.3 一般操作过程 10

·流程图表示 10

·完善家谱代码实现 11

·添加家庭成员代码实现 12

·解散局部家庭代码实现 13

·更改家庭成员姓名代码实现 13

·打印全家福代码实现 14

·查找家庭成员代码实现 14

·展示局部家庭代码实现 15

2.3.4 项目主体部分 16

·流程图表示 16

·代码实现 17

[3 项目测试 1](#_Toc495668161)9

[3.1 建立家谱 1](#_Toc495668162)9

[3.2 完善家谱 1](#_Toc495668166)9

[3.3 完成多个家庭建立 2](#_Toc495668170)0

[3.4 添加家庭成员 2](#_Toc495668174)0

[3.5 解散局部家庭 2](#_Toc495668178)1

[3.6 更改家庭成员姓名 2](#_Toc495668182)1

[3.7 查找家庭成员（找到一个） 21](#_Toc495668182)

3.8 查找家庭成员（找到多个） 22

3.9 展示局部家庭 22

3.10 保存族谱 23

3.11 从多个查找结果中选择一个的情况 23

3.12 退出程序 24

3.13 多代单传的情况 24

3.14 一代多子女的情况 24

**1.项目分析**

1.1 项目背景

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 项目要求

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。

项目示例：



1.3 项目需求分析

对于实现家谱管理系统的程序，需考虑以下需求：

**·正确性**

程序应当能够按照要求对完善家谱、添加子女等功能进行实现，没有错误

**·健壮性**

程序应当对相关信息，如家庭成员姓名、祖先姓名等的错误输入进行处理，也要能对特殊情况（如对应的家庭成员不存在）进行合适的处理。

**·高效性**

程序涉及到的操作有多种，并且使用都会非常频繁，这就要求要在尽可能短的时间内处理大量的数据，完成对应的操作。

**·交互性**

程序要以系统的形式出现，涉及到菜单选择、界面切换等功能，需要给用户提供足够的提示与说明，使用方式要友好。

**2.项目设计**

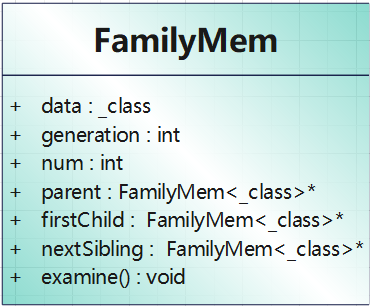
2.1 数据结构设计

由项目分析可以得出，本项目中因为每个家庭成员不仅可能有多个子女，也可能有多个兄弟，所以需要实现一个家族树，也就是多叉树，以及相应的插入、删除操作。

2.2 类设计

**2.2.1 家族成员类（FamilyMem）**

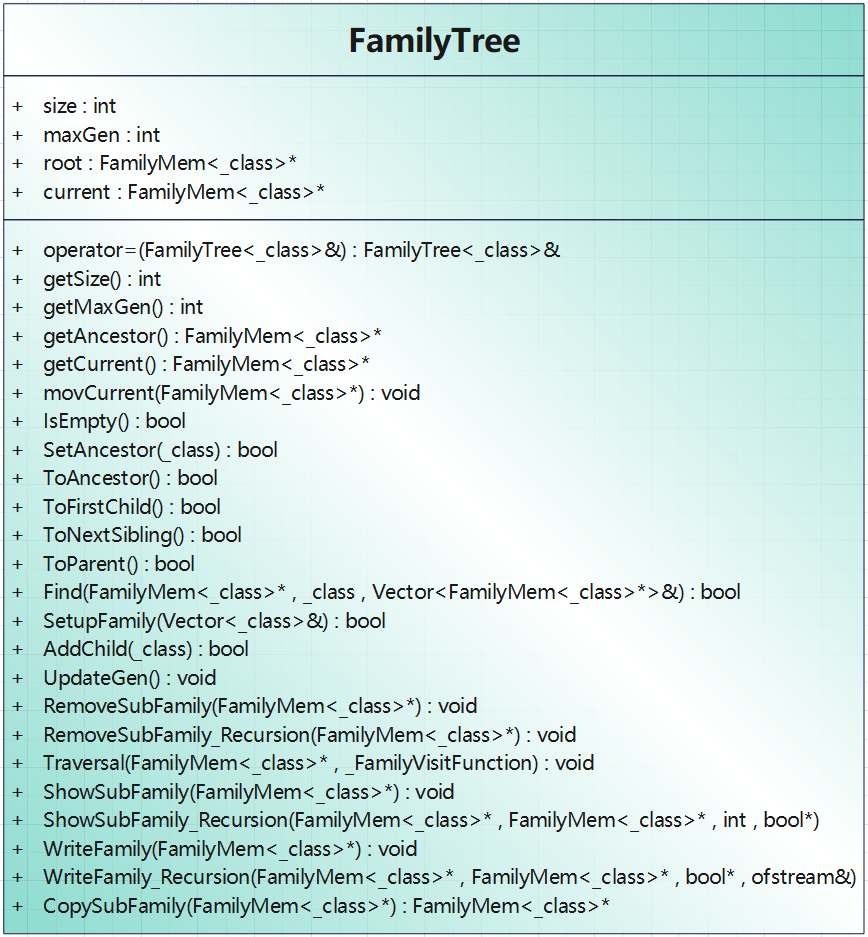
家族成员类（FamilyMem）存储了一个家庭成员较为详细的信息，包括世代、编号、父结点指针、长子指针、兄弟指针等，这是为了在出现同名时可以帮助用户分辨，但相应的存储效率也降低了。其UML图如下：



**2.2.2 家族树类（FamilyTree）**

家族树类（FamilyTree）采用了左长子右兄弟的多叉树结构，符合实际中的家庭情况。其记录有总人数、最大世代，同时还有一个内置的指针current，方便在给成员添加多个子女时做一个位置记录，无需再从根结点来到当前位置，类当中封装了current的各类操作，如移动到当前结点的父结点等。家族树类还支持成员的添加、局部家庭的解散、局部家庭的展示、保存族谱等操作，基本可以满足需要。其中查找过程使用了向量类（Vector），目的是应对有多个人名重复的情况。

其UML图如下：



其中，主要函数如下：

//重载函数：复制

FamilyTree<\_class>& operator=(const FamilyTree<\_class>& family);

//取当前家族人数

int getSize()const;

//取当前家族最大代数

int getMaxGen()const;

//取祖先位置

FamilyMem<\_class>\* getAncestor()const;

//取当前成员位置

FamilyMem<\_class>\* getCurrent()const;

//更改当前成员

void movCurrent(FamilyMem<\_class>\* dst);

//判断家族是否为空

bool IsEmpty()const;

//建立祖先

bool SetAncestor(const \_class ancestor);

//在树中搜索含有target的结点（仅以数据域为标准），将所有查找结果压入results中

bool Find(FamilyMem<\_class>\* root, const \_class target, Vector<FamilyMem<\_class>\*>& results)const;  //BFS

//若当前成员无子女，为其添加若干个子女

bool SetupFamily(Vector<\_class>& children);

//若当前成员有子女，为其添加一个子女

bool AddChild(\_class child);

//更新家族的最大代数

void UpdateGen();  //BFS

//解散以root为根的家庭，完成后root为当前成员

void RemoveSubFamily(FamilyMem<\_class>\* root);

//展示局部家庭（以根结点为祖先，世代为相对世代）

void ShowSubFamily(FamilyMem<\_class>\* root)const;

//将族谱写入文件

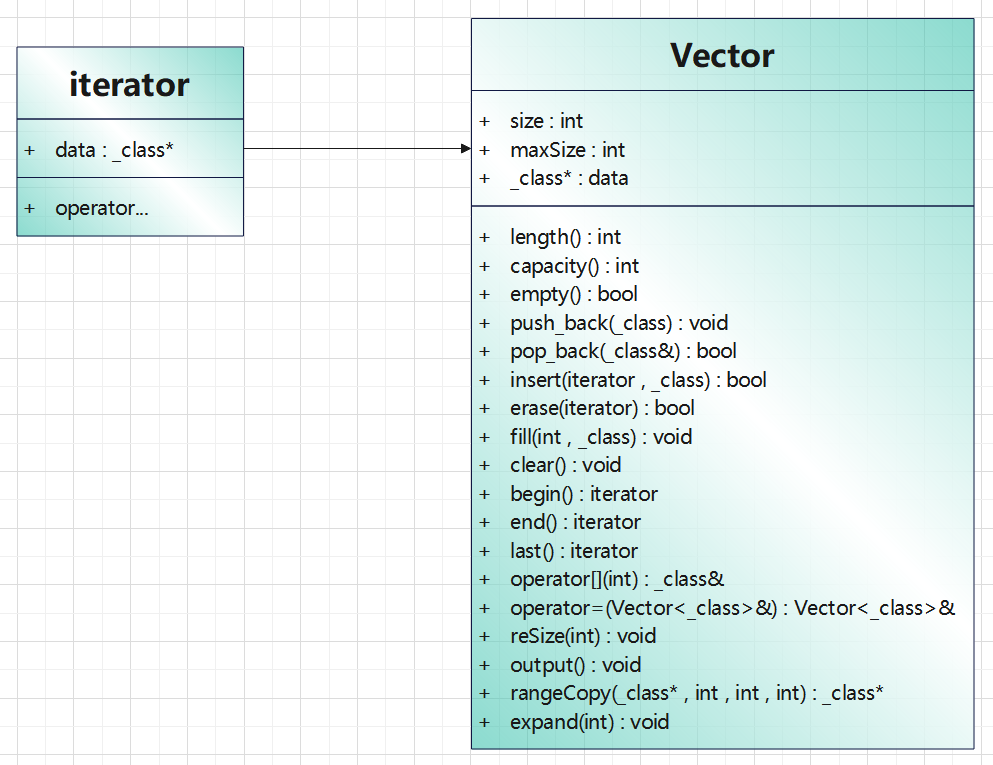
void WriteFamily(FamilyMem<\_class>\* root)const;

**2.2.3 向量类（Vector）**

向量类（Vector）是一种容器，其本质上通过动态数组来实现，使得元素按照线性顺序排列，同时可以通过下标快速访问，也提供了在末尾进行插入、删除元素等操作。

另一种常用的向量类操作是对其的遍历，在此添加了迭代器类（iterator）和一些运算符重载，以便对数组的遍历等操作。由于这个类名在标准库中已有实现，且此处的迭代器类为Vector类特有，故使用了嵌套类的形式，使得iterator只能在Vector类内访问。

该类和其内部的iterator类的UML图如下所示：



其中，主要函数如下：

//返回容器内部数组的长度

int length()const;

//返回容器最大容量

int capacity()const;

//判断容器是否为空

bool empty()const;

//向容器末尾压入一个元素

void push\_back(const \_class item);

//弹出容器末尾元素

bool pop\_back(\_class& x);

//在迭代器位置place上插入元素item，此位置及其后续元素后移一位

bool insert(const Vector<\_class>::iterator place, const \_class item);

//删除迭代器位置place上的元素

bool erase(const Vector<\_class>::iterator place);

//指定容器数组长度，并用元素item进行填充

void fill(const int sz, const \_class item);

//将容器置为空

void clear();

//返回容器的起始位置

inline Vector<\_class>::iterator begin();

inline const Vector<\_class>::iterator begin()const;

//返回容器末尾的后一个位置

inline Vector<\_class>::iterator end();

inline const Vector<\_class>::iterator end()const;

//返回容器的末尾位置

inline Vector<\_class>::iterator last();

inline const Vector<\_class>::iterator last()const;

//重载函数：下标访问

\_class& operator[](const int pos)const;

//重载函数：赋值

Vector<\_class>& operator=(const Vector<\_class>& v);

//调整容器的数组长度

void reSize(const int sz);

//输出容器内元素的值

void output()const;

//从data中的[low,high)区域复制一段长为sz的数据

\_class\* rangeCopy(\_class\* data, const int low, const int high, int sz = 0);

//扩大数组空间

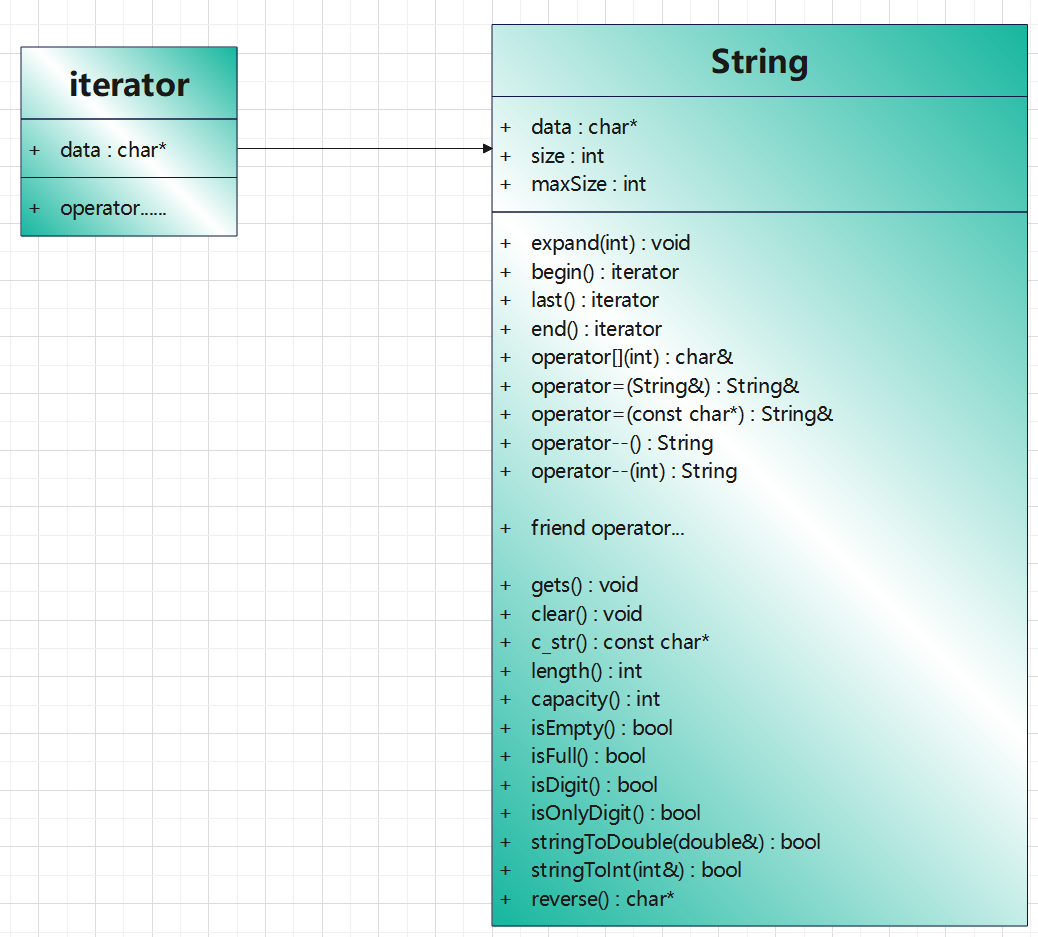
void expand(const int \_timesOfExpandingDefaultSize);

**2.2.4 字符串类（String）**

字符串本质上是通过字符数组实现的，通过动态开辟一个数组存放各个字符。由于字符串的添加、比较、赋值等操作使用得十分频繁，故该类重载了许多运算符的操作，使得这些操作使用起来更易上手。

另一种常用的字符串操作是对其的遍历，在此添加了迭代器类（iterator）和一些运算符重载，以便对字符串的遍历等操作。由于这个类名在标准库中已有实现，且此处的迭代器类为String类特有，故使用了嵌套类的形式，使得iterator只能在String类内访问。

该类和其内部的iterator类的UML图如下所示：



其中，主要函数如下：

//增加数组空间大小

void expand(const int \_timesOfExpandingDefaultSize);

//返回字符串的起始位置

inline String::iterator begin();

inline const String::iterator begin()const;

//返回字符串末尾的后一个位置

inline String::iterator end();

inline const String::iterator end()const;

//返回字符串的末尾位置

inline String::iterator last();

inline const String::iterator last()const;

//重载函数：下标访问

char& operator[](const int pos)const;

//重载函数：复制（从String对象复制）

String& operator=(const String& str);

//重载函数：复制（从const char\*变量复制）

String& operator=(const char\* str);

//输入字符串（支持空格输入）

void gets();

//将字符串置为空

void clear();

//返回字符串首地址（以const char\*形式返回）

const char\* c\_str()const;

//返回字符串长度

int length()const;

//返回字符串最大容量

int capacity()const;

//判断字符串是否为空

bool isEmpty()const;

//判断字符串是否已满

bool isFull()const;

//判断字符串是否表示一个数值

bool isDigit()const;

//判断字符串是否仅有数字

bool isOnlyDigit()const;

//将字符串转成双精度数

bool stringToDouble(double& d)const;

//将字符串转成整型数

bool stringToInt(int& i)const;

//将字符串反转

char\* reverse();

2.3 算法设计

**2.3.1 算法思路**

程序支持的操作较多，但大多都需要成员的名字再进行操作。故每次输入成员名字后，都执行一次对家族树的遍历（本程序用广度优先遍历），每找到一个符合条件的成员，就将指向该结点的指针存入Vector中，再由用户手动选择最终结果，此举可以应对多人同名的情况。若仅找到一个结果则不需要选择。

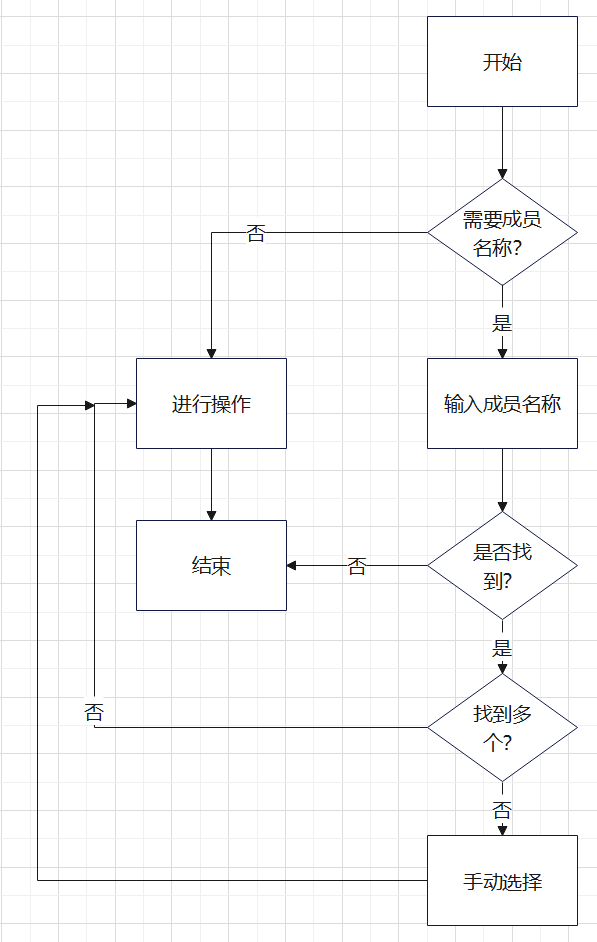
展示局部家庭时，程序使用thisAncestor作为相对祖先，gen\_gap作为世代数差值，这两个量可以使得展示出的局部家庭中thisAncestor在第0代的位置。同时程序使用一个stillSibling数组，记录某一代是否仍有兄弟未被输出，再根据该数组调整输出格式，使得呈现出来的结构更加清晰直观。打印全家福是thisAncestor是绝对祖先的情况，保存族谱是将输出重定向到指定文件的情况。

**2.3.2 性能评估**

设每次操作前家族已有人数为n，多数操作都需要遍历查找或遍历输出，时间复杂度均为O(n)（完善家谱除外，设新加人数为m，复杂度为O(m+n)）。

**2.3.3 一般操作过程**

·流程图表示

****

·完善家谱代码实现

void family\_setup(FamilyTree<String>& Family)

{

    String parent;  /\*成员姓名\*/

    String child;   /\*子女姓名\*/

    Vector<FamilyMem<String>\*> results;  /\*查找结果\*/

    FamilyMem<String>\* Parent;  /\*指向该成员的指针\*/

    cout << "请输入要建立家庭的人的姓名：";

    cin >> parent;

    cin.ignore(INT\_MAX, '\n');

    if (!Family.Find(Family.getAncestor(), parent, results)) {  /\*查找成员\*/

        cout << "成员不存在！" << endl;

        pause();

        return;

    }

    else

        Parent = results\_check(results);  /\*选择成员\*/

    if (Parent->firstChild) {

        cout << "当前成员已有子女！" << endl;

        pause();

        return;

    }

    cout << "请依次输入" << parent << "的儿女的姓名（以#结尾）：";

    Vector<String> children;

    while (1) {

        cin >> child;

        if (child == "#") {  /\*输入#结束，省去输入数量的步骤\*/

            cin.ignore(INT\_MAX, '\n');

            break;

        }

        else

            children.push\_back(child);

    }

    if (children.empty()) {

        cout << "无添加子女数据！" << endl;

        pause();

        return;

    }

    Family.movCurrent(Parent);     /\*移动指针\*/

    Family.SetupFamily(children);  /\*添加子女\*/

    Family.ShowSubFamily(Parent);  /\*展示局部家庭\*/

    pause();

}

·添加家庭成员代码实现

void family\_add(FamilyTree<String>& Family)

{

    String parent;  /\*成员姓名\*/

    String child;   /\*子女姓名\*/

    Vector<FamilyMem<String>\*> results;  /\*查找结果\*/

    FamilyMem<String>\* Parent;  /\*指向该成员的指针\*/

    cout << "请输入要添加儿子（或女儿）的人的姓名：";

    cin >> parent;

    cin.ignore(INT\_MAX, '\n');

    if (!Family.Find(Family.getAncestor(), parent, results)) {  /\*查找成员\*/

        cout << "成员不存在！" << endl;

        pause();

        return;

    }

    else

        Parent = results\_check(results);  /\*选择成员\*/

    if (Parent->firstChild == NULL) {

        cout << "当前成员未有子女！" << endl;

        pause();

        return;

    }

    cout << "请输入" << parent << "新添加的儿子（或女儿）的姓名，或输入#取消：";

    cin >> child;

    cin.ignore(INT\_MAX, '\n');

    if (child == "#") {  /\*输入#结束\*/

        cout << "取消成功" << endl;

        pause();

        return;

    }

    Family.movCurrent(Parent);     /\*移动指针\*/

    Family.AddChild(child);     /\*添加子女\*/

    Family.ShowSubFamily(Parent);  /\*展示局部家庭\*/

    pause();

}

·解散局部家庭代码实现

void family\_remove(FamilyTree<String>& Family)

{

    String parent;  /\*成员姓名\*/

    Vector<FamilyMem<String>\*> results;  /\*查找结果\*/

    FamilyMem<String>\* Parent;  /\*指向该成员的指针\*/

    cout << "请输入要解散家庭的人的姓名：";

    cin >> parent;

    cin.ignore(INT\_MAX, '\n');

    if (!Family.Find(Family.getAncestor(), parent, results)) {  /\*查找成员\*/

        cout << "成员不存在！" << endl;

        pause();

        return;

    }

    else

        Parent = results\_check(results);  /\*选择成员\*/

    Family.movCurrent(Parent);  /\*移动指针\*/

    cout << parent << "的家庭已解散，原家庭情况为：" << endl;  /\*展示局部家庭\*/

    Family.ShowSubFamily(Parent);

    Family.RemoveSubFamily(Family.getCurrent());  /\*删除局部家庭\*/

    pause();

}

·更改家庭成员姓名代码实现

void family\_revise(FamilyTree<String>& Family)

{

    String member, newName;  /\*成员姓名，新姓名\*/

    Vector<FamilyMem<String>\*> results;  /\*查找结果\*/

    FamilyMem<String>\* Member;  /\*指向该成员的指针\*/

    cout << "请输入要更改姓名的人的目前姓名：";

    cin >> member;

    cin.ignore(INT\_MAX, '\n');

    if (!Family.Find(Family.getAncestor(), member, results)) {  /\*查找成员\*/

        cout << "成员不存在！" << endl;

        pause();

        return;

    }

    else

        Member = results\_check(results);  /\*选择成员\*/

    cout << "请输入更改后的姓名：";

    cin >> newName;

    cin.ignore(INT\_MAX, '\n');

    Member->data = newName;

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

    pause();

}

·打印全家福代码实现

void family\_show(FamilyTree<String>& Family)

{

    Family.ShowSubFamily(Family.getAncestor());

    pause();

}

·查找家庭成员代码实现

void family\_search(FamilyTree<String>& Family)

{

    String member;  /\*成员姓名\*/

    Vector<FamilyMem<String>\*> results;  /\*查找结果\*/

    cout << "请输入要查找的家庭成员的姓名：";

    cin >> member;

    cin.ignore(INT\_MAX, '\n');

    if (!Family.Find(Family.getAncestor(), member, results)) {  /\*查找成员\*/

        cout << "成员不存在！" << endl;

        pause();

        return;

    }

    if (results.length() == 1) {  /\*仅搜索到一个结果\*/

        cout << "查找成功，信息如下：" << endl << endl;

        results[0]->examine();

    }

    else {  /\*搜索到多个结果\*/

        cout << "查找到多个结果..." << endl << endl;

        Vector<FamilyMem<String>\*>::iterator iter = results.begin();

        while (iter != results.end()) {

            cout << "结果" << iter - results.begin() + 1 << "：" << endl;

            (\*iter)->examine();

            cout << endl;

            iter++;

        }

    }

    pause();

}

·展示局部家庭代码实现

void family\_show\_branch(FamilyTree<String>& Family)

{

    String ancestor;  /\*祖先姓名\*/

    Vector<FamilyMem<String>\*> results;  /\*查找结果\*/

    FamilyMem<String>\* Ancestor;  /\*指向当前祖先的指针\*/

    cout << "请输入要展示的局部家庭的祖先姓名：";

    cin >> ancestor;

    cin.ignore(INT\_MAX, '\n');

    if (!Family.Find(Family.getAncestor(), ancestor, results)) {  /\*查找成员\*/

        cout << "成员不存在！" << endl;

        pause();

        return;

    }

    else

        Ancestor = results\_check(results);  /\*选择成员\*/

    cout << "该局部家庭如下：" << endl;

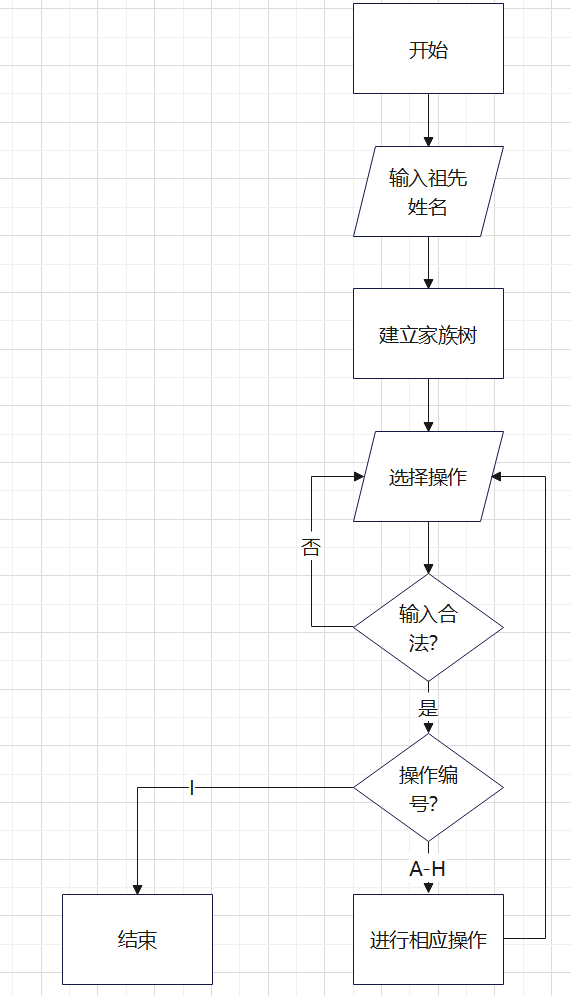
    Family.ShowSubFamily(Ancestor);

    pause();

}

**2.3.4 项目主体部分**

·流程图表示：



·代码实现：

int main()

{

    String ancestor;

    bool if\_on = true;

    print\_head();

    cout << "首先建立一个家谱！\n"

        << "请输入祖先的姓名：";

    cin >> ancestor;

    cin.ignore(INT\_MAX, '\n');

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

    FamilyTree<String> Family(ancestor);

    while (if\_on) {

        /\*显示实时人数和世代\*/

        cout << "\n当前情况：人数" << Family.getSize() << "人，共" << Family.getMaxGen() << "代" << endl;

        cout << "\n请选择要执行的操作：";

        char selection;

        while (1) {

            selection = getchar();

            cin.ignore(INT\_MAX, '\n');

            if ((selection >= 'A' && selection <= 'I') || (selection >= 'a' && selection <= 'i'))

                break;

        }

        switch (selection) {

        case 'A':

        case 'a':

            family\_setup(Family);

            break;

        case 'B':

        case 'b':

            family\_add(Family);

            break;

        case 'C':

        case 'c':

            family\_remove(Family);

            break;

        case 'D':

        case 'd':

            family\_revise(Family);

            break;

        case 'E':

        case 'e':

            family\_show(Family);

            break;

        case 'F':

        case 'f':

            family\_search(Family);

            break;

        case 'G':

        case 'g':

            family\_show\_branch(Family);

            break;

        case 'H':

        case 'h':

            Family.WriteFamily(Family.getAncestor());

            break;

        case 'I':

        case 'i':

            if\_on = false;

            break;

        }

        print\_head();

    }

    cout << "\n家谱管理系统已关闭！" << endl;

    return 0;

}

**3.项目测试**

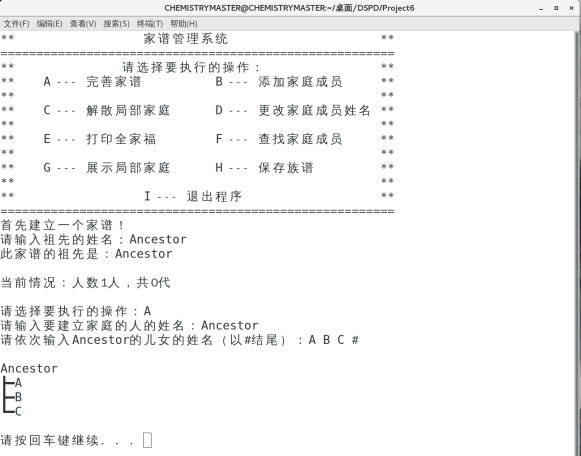
3.1 建立家谱

测试结果：



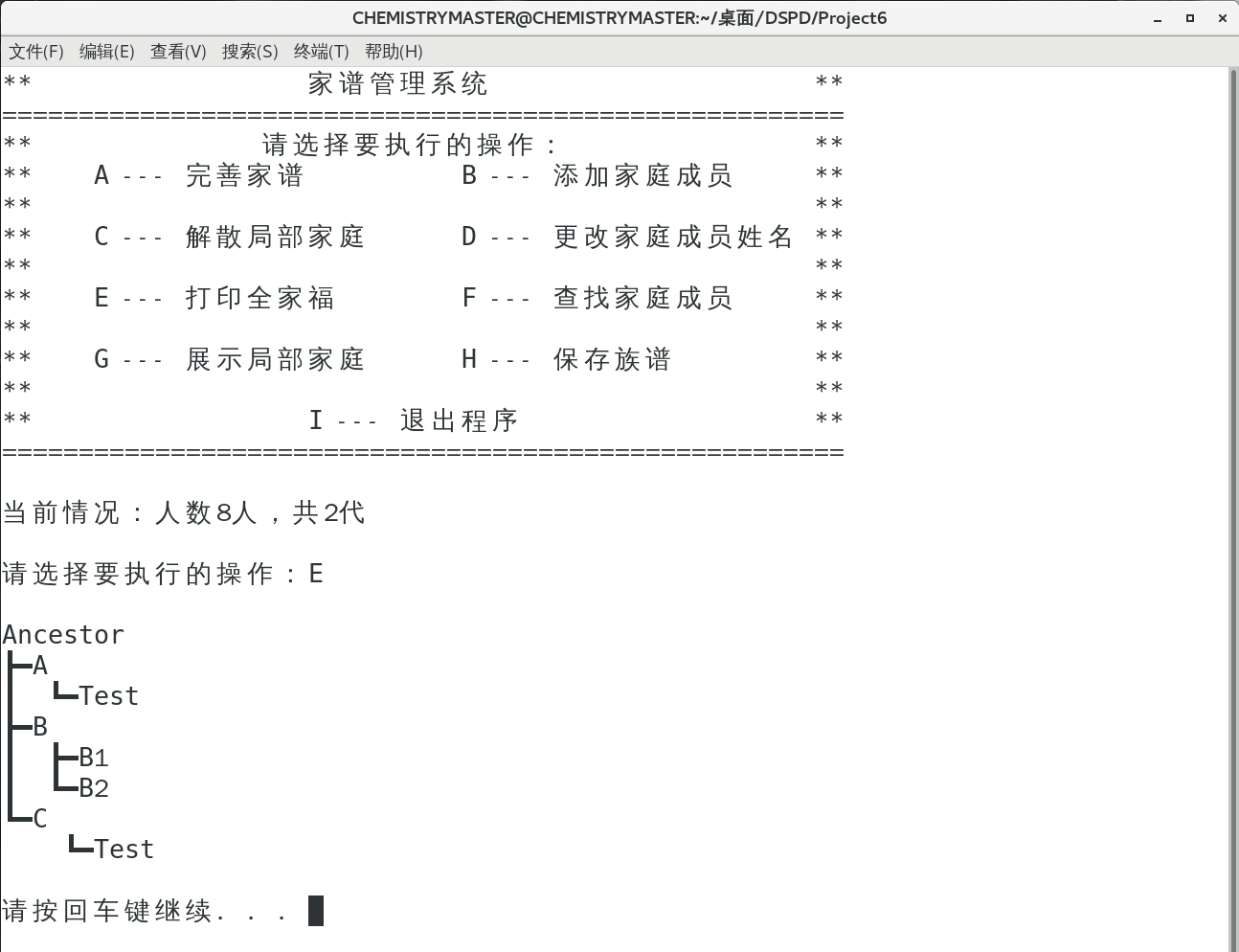
3.2 完善家谱

测试结果：



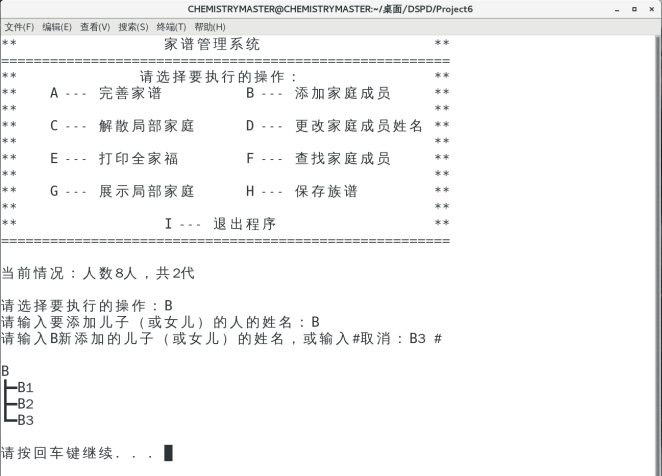
3.3 完成多个家庭建立

测试结果：



3.4 添加家庭成员

测试结果：



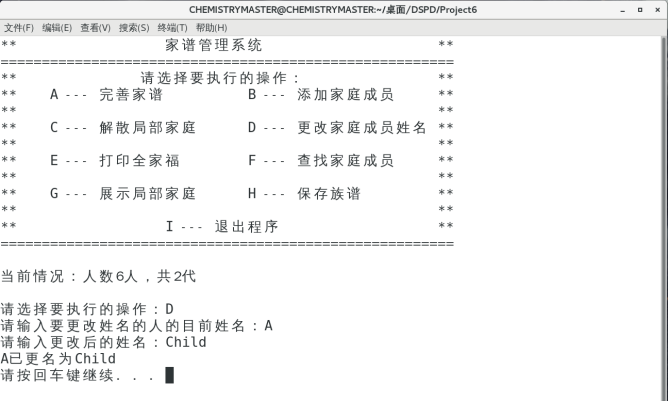
3.5 解散局部家庭

测试结果：



3.6 更改家庭成员姓名

测试结果：



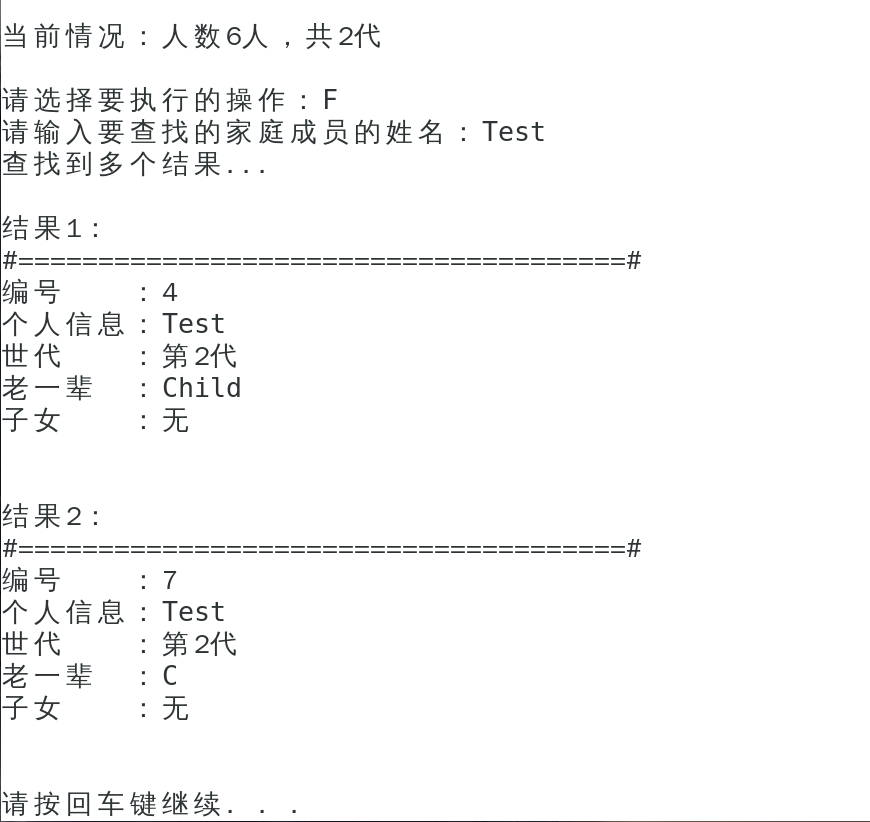
3.7 查找家庭成员（找到一个）

测试结果：



3.8 查找家庭成员（找到多个）

测试结果：



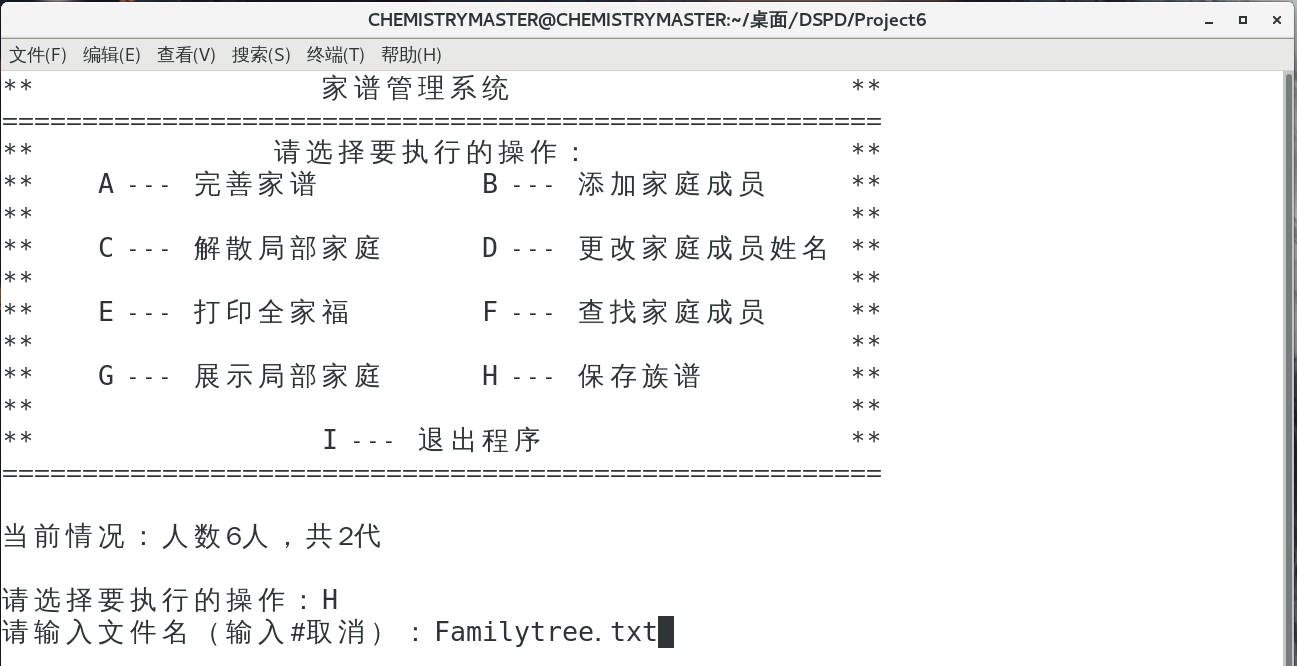
3.9 展示局部家庭

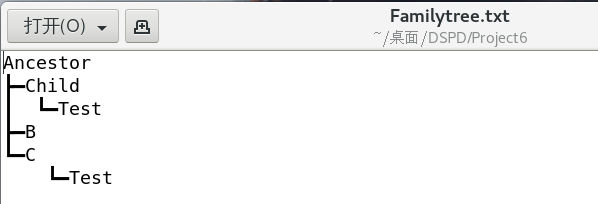
测试结果：



3.10 保存族谱

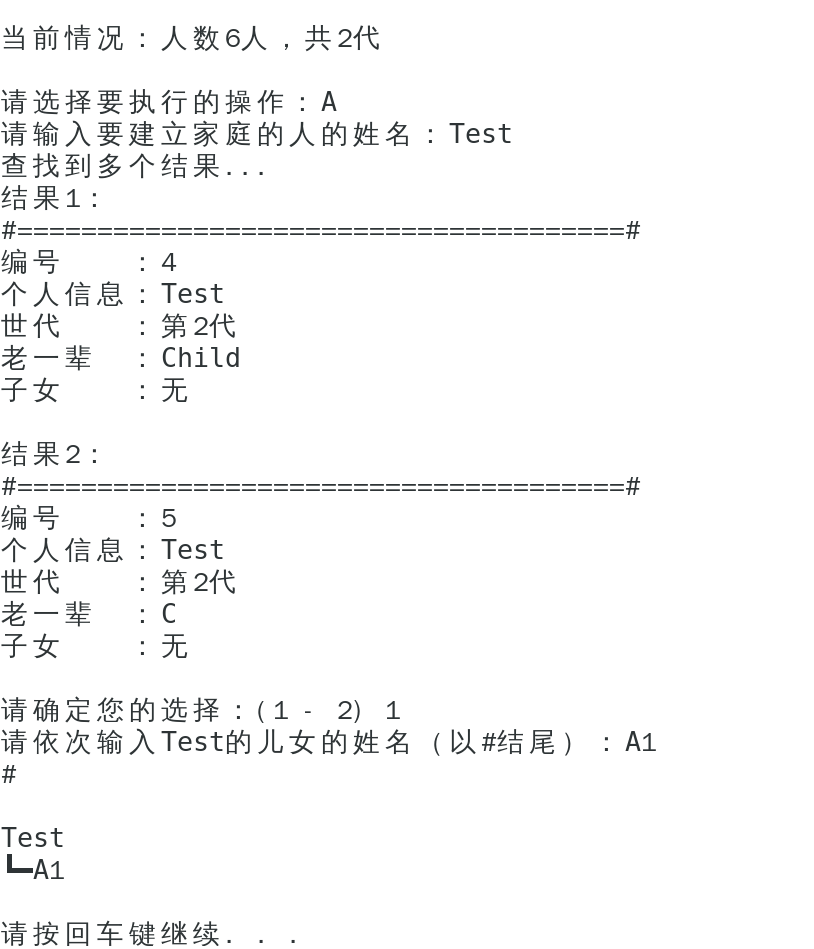
测试结果：





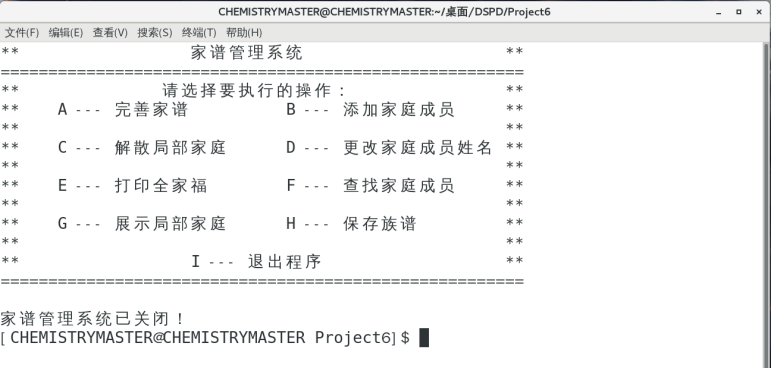
3.11 从多个查找结果中选择一个的情况

测试结果：



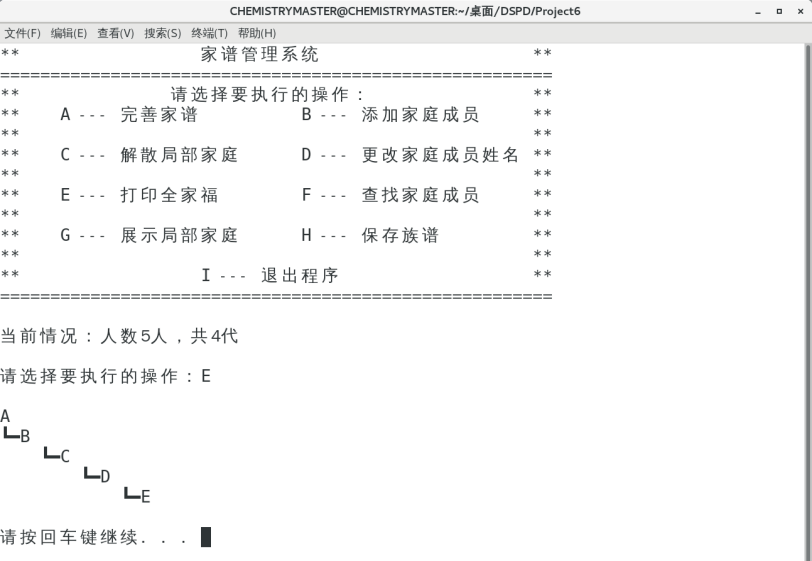
3.12 退出程序

测试结果：



3.13 多代单传的情况

测试结果：



3.14 一代多子女的情况

测试结果：

