项目说明文档

数据结构课程设计

——家谱管理系统

作 者 姓 名： 杨滕超

学 号： 2151298

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

[1 项目概述](#_Toc495668153) 4

[1.1 项目背景](#_Toc495668154) 4

[1.2 项目功能](#_Toc495668155) 4

[1.3 项目分析](#_Toc495668154) 4

[2 项目设计](#_Toc495668156) 4

[2.1 总体设计](#_Toc495668157) 4

[2.2 类设计](#_Toc495668158) 5

[2.2.1 Vector类](#_Toc495668163) 5

[2.2.2 String类](#_Toc495668163) 6

[2.2.3 hashtable类](#_Toc495668163) 7

[2.2.4 Queue类](#_Toc495668163) 7

[2.2.5 familyTreeNode类](#_Toc495668163) 8

[2.2.4 familyTree类](#_Toc495668163) 8

[3 项目实现](#_Toc495668161) 10

[3.1 总体实现](#_Toc495668162) 10

[3.1.1 总体实现流程图](#_Toc495668167) 10

[3.1.2 总体实现思路](#_Toc495668167) 11

[3.1.3 总体实现主要代码](#_Toc495668167) 11

[3.2 初始化家谱实现 1](#_Toc495668166)2

[3.2.1 初始化家谱实现流程图 1](#_Toc495668167)2

[3.2.2 初始化家谱实现思路](#_Toc495668167) 12

[3.2.3 初始化家谱实现主要代码](#_Toc495668168) 12

[3.3 完善家谱实现](#_Toc495668162) 14

[3.3.1 完善家谱实现流程图](#_Toc495668167) 14

[3.3.2 完善家谱实现思路](#_Toc495668167) 15

[3.3.3 完善家谱实现主要代码](#_Toc495668167) 15

[3.4 添加家庭成员实现](#_Toc495668162) 17

[3.4.1 添加家庭成员实现流程图](#_Toc495668167) 17

[3.4.2 添加家庭成员实现思路](#_Toc495668167) 18

[3.4.3 添加家庭成员实现主要代码](#_Toc495668167) 18

[3.5 修改家庭成员信息实现](#_Toc495668162) 19

[3.5.1 修改家庭成员信息实现流程图](#_Toc495668167) 19

[3.5.2 修改家庭成员信息实现思路](#_Toc495668167) 20

[3.5.3 修改家庭成员信息实现主要代码](#_Toc495668167) 20

[3.6 解散局部家庭实现](#_Toc495668162) 21

[3.6.1 解散局部家庭实现流程图](#_Toc495668167) 21

[3.6.2 解散局部家庭实现思路](#_Toc495668167) 22

[3.6.3 解散局部家庭实现递归方法主要代码](#_Toc495668167) 22

[3.6.4 解散局部家庭实现非递归方法思路](#_Toc495668167) 22

[3.6.5 解散局部家庭实现非递归方法主要代码](#_Toc495668167) 22

[3.7 家族树展示实现](#_Toc495668162) 23

[3.7.1 家族树展示思路](#_Toc495668167) 23

[3.7.2 家族树展示主要代码](#_Toc495668167) 23

[3.8 搜索实现](#_Toc495668162) 24

[3.8.1 搜索思路](#_Toc495668167) 24

[3.8.2 递归深度优先搜索实现主要代码](#_Toc495668167) 24

[3.8.3 哈希表搜索实现主要代码](#_Toc495668167) 24

[3.9 哈希表实现](#_Toc495668162) 25

[3.9.1 哈希表的大小](#_Toc495668167) 25

[3.9.2 哈希表的扩大](#_Toc495668167) 25

[3.9.3 哈希函数](#_Toc495668167) 26

[3.9.4 哈希表的函数实现方法](#_Toc495668167) 26

[4 项目测试](#_Toc495668161) 27

[4.1 初始化家谱测试](#_Toc495668174) 27

[4.2 完善家庭](#_Toc495668174) 27

[4.2.1 添加单个子女测试](#_Toc495668167) 27

[4.2.2 添加多个子女测试](#_Toc495668167) 28

[4.2.3 多代多个子女测试](#_Toc495668167) 28

[4.3 完善家庭错误输入测试](#_Toc495668174) 28

[4.3.1 尝试建立已建立家庭个人的家庭](#_Toc495668167) 28

[4.3.2 输入家谱中不存在成员](#_Toc495668167) 28

[4.3.3 输入姓名与已有成员重名](#_Toc495668167) 29

[4.4 添加家庭成员正确测试](#_Toc495668174) 29

[4.5 添加家庭成员输入错误测试](#_Toc495668174) 29

[4.5.1 输入家谱中不存在成员](#_Toc495668167) 29

[4.5.2 输入姓名与已有成员重名](#_Toc495668167) 29

[4.6 解散局部家庭正确测试](#_Toc495668174) 30

[4.7 解散局部家庭输入错误测试](#_Toc495668174) 30

[4.7.1 输入家谱中不存在成员](#_Toc495668167) 30

[4.8 更改家庭成员姓名正确测试](#_Toc495668174) 30

[4.9 更改家庭成员姓名输入错误测试](#_Toc495668174) 31

[4.9.1 输入家谱中不存在成员](#_Toc495668167) 31

[4.9.2 输入姓名与已有成员重名](#_Toc495668167) 31

[4.10 输入错误操作测试](#_Toc495668174) 31

[4.11 单链家谱测试](#_Toc495668174) 32

[4.12 扁平家谱测试](#_Toc495668174) 32

1 项目概述

* 1. 项目背景

在中国传统观念中，家族一直被公认为是人们生活工作中十分重要的东西，从宗族制度建立伊始，家谱伴随着中国人民走过了几千年的历程。因此家谱对于中国人具有特殊并且重要的意义。

本项目聚焦于对将家谱依托计算机的载体在屏幕上呈现，实现了时代的融合，将旧事物传承，新事物发展，具有时代的烙印又不失科技的创新。由此，我们对于保存家族的记忆增多另一种方式，“寻根”等追忆家族往事变得更加方便。

* 1. 项目功能

本项目是对家谱管理的简单模拟，支持建立家谱、完善家庭成员、增添家庭成员、更改家庭成员姓名、解散家庭成员、展示家族树等功能。各项功能由用户自行选择，并配备丰富的提示信息，方便用户理解与操作。

* 1. 项目分析

丰富的提示信息，保证用户正确输入；在用户输入错误的情况下，程序将输出输入错误的提示信息，保证程序不会崩溃，得以顺利运行。

适宜的数据结构。关于族谱，优先想到的是树形结构，而对于个人的子女数量不确定性，进而我们考虑利用多叉树，以保证对族谱的存储。

较高的效率。对于频繁的查询、插入、删除等操作，经过算法与数据结构的改良，具有较少的时间复杂度。关于此，特别是查询操作是其他操作的基础，经过递归、迭代等等方法查询，我们最终考虑利用哈希表存储和查询。考虑到现代计算机容量大大提升，而对于时间效率的要求越来越高，而哈希表正好符合这一要求。

2 项目设计

2.1 总体设计

对于数据结构的选取，我们考虑到家谱所对应的树形结构，而对于多叉树的存储方式，例如：双亲数组法、左长子右兄弟法等等，我们选取利用动态数组开辟的方法存储子女。原因如下：首先对于每个结点，可以轻松地访问其第几个子女的信息，相较于双亲数组法和左长子右兄弟法，它能以O(1)的时间复杂度寻找到某个结点的第i个结点，这是另外两种方法所不具有的优点；其次对于遍历操作，双亲数组的遍历实现上有一定的困难；而对于第二种方法，在空间利用效率上需要增加将每个结点连接的指针；对于本方法，根据所需要的子女开辟所需要的空间，具有灵活性的同时更具有简易操作性。

对于增删改查等操作，均是以查找操作为基础，因此要求查找操作的时间复杂度较低，于是我们考虑哈希表查找。

对于是否家谱中是否有重名的情况，本项目将名字作为关键码，因此不允许有重名，当然可以利用家谱中成员的编号作为关键码存储，这样可以实现重名的并且可以区分的操作。但两种方法各有优劣，我们思考到，在现实生活中查询个人信息首先是通过名字查询，作为名字这个个人的属性，之后在为用户列出查询到的所有个人信息，再通过其他个人属性进行区分。基于此可以对程序做出相应的改进，以防止出现“百年孤独”家谱(多次出现重名人物)导致程序不能优秀地运行的情况。

2.2 类设计

2.2.1 Vector类

本项目主要利用该类实现动态开辟数组，以实现对于子女数量不定的考虑。

向量作为存储同一种类型数据的一维数组，相当于是数组的扩展，在其基础上增加方便程序员工作的操作。其中的优点在于：对于查找某个位置的元素的时间复杂度为O(1)，但是它的缺点也十分明显：向量的删除与插入的时间代价是巨大的，时间复杂度基本是O(n)级别；但特殊的是，在最后插入元素时，时间复杂度为O(1)。

我们设计Vector类的同时，同样设计了它的迭代器。

主要成员函数如下：

·//构造函数

Vector();

Vector(int size);

Vector(const Vector<T>& V);

Vector(int size, const T& val);

·//清空Vector

void clear();

·//最后添加元素

void emplace\_back(const T& x);

·//最后删除元素

void pop\_back();

·//返回最后元素

const T back() const ;

T& back() ;

·//返回开头元素

const T front() const ;

T& front() ;

·//插入

void insert(const Vector<T>::vector\_iterator& it, const T& x);

·//删除

void remove(const Vector<T>::vector\_iterator& it);

·//迭代器begin

inline Vector<T>::vector\_iterator begin() ;

·//迭代器end

inline Vector<T>::vector\_iterator end() ;

·//返回迭代器的查找

typename Vector<T>::vector\_iterator find(const T& x);

·//重载[]

T& operator[](const int index) ;

·//重载=

Vector<T>& operator=(const Vector<T>& V);

·//重新设置大小

void resize(int size);

void resize(int size, const T& val);

2.2.2 String类

字符串作为使用频率较高的一种数据结构，在程序发挥着巨大的作用。C++string类中配套的多种函数方便我们对字符串的处理，例如：比较、连接、替换、搜索等等。将其完全实现实现任务艰巨。因此我们实现其中的主要功能，以完成对主要的程序的要求。

该类主要成员函数如下：

·//空构造函数String();

·//赋值构造函数String(const String& str);

·//字符串构造函数String(const char\* str);

·//大小为size的字符串构造函数String(const char\* str, int size);

·//字符串长度int length()const;

·//重载=，StirngString& operator=(const String& str);

·//重载=，字符串String& operator=(const char\* str);

·//重载=，一个字符 String& operator=(const char& ch);

·//重载比较函数friend int strCmp(const String& str1, const String& str2);

·//重载< > <= >= friend bool operator<(const String& str1, const String& str2);

friend bool operator>(const String& str1, const String& str2);

friend bool operator<=(const String& str1, const String& str2);

friend bool operator>=(const String& str1, const String& str2);

·//重载[]inline char& operator[](const int index);

·//重载== bool operator==(const String& str);

bool operator==(const char\* str);

·//重载+ String operator+(const String& str);

String operator+(const char\* str);

·//重载+= String& operator+=(const String& str);

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

String& operator+=(const char ch);

·//返回字符串类型char\* c\_str();

·//模式匹配 int find(const char\* str);

int find(String& str);

·//删除所有的某个字符void erase(char ch);

·//翻转void reverse();

·//清空void clear();

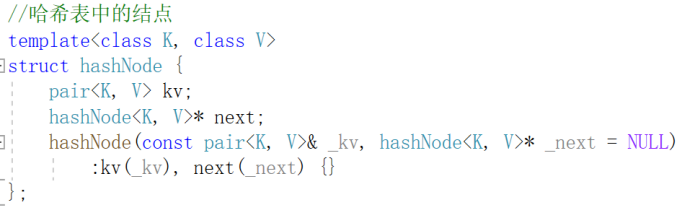
·//删掉最后一个void pop\_back();

·//获得最后一个字符char back() const;

2.2.3 hashtable类

哈希表是一种在搜索上平均时间复杂度为O(1)的数据结构，它将元素的存储位置与关键码之间建立一个确定的函数对应关系，使得每一个关键码与结构中的一个唯一的存储位置相对应。但是仍然会存在哈希冲突——即不同关键码通过哈希函数映射到同一个存储位置，通过开地址法或链地址法，我们可以解决冲突，考虑到数据数量的不确定性，本项目使用链地址法解决冲突。具体实现将在后文实现部分展开说明。

哈希结点包括二元组包含的键值对以及指向下一个结点的指针



主要成员函数：

//构造函数 hashtable();

hashtable(const size\_t sz);

//析构函数~hashtable();

//查找Node\* find(const K& key);

//插入void insert(const pair<K, V>& p);

//删除void erase(const K& key);

//判断是否为空bool empty()const;

//返回元素个数bool size() const;

2.2.4 Queue类

队列和栈一样，同样是一种运算受限的线性表，它允许的操作是在表的前端(front)进行删除，表的后端(rear)进行插入，遵循先进先出的原则。其中的前端称为队头，后端称为队尾。队列的实现同样分为两种方式，顺序队列和链式队列，对于顺序队列，为了实现空间的循环利用，采用循环队列的方式，其中的要点在于队头指针和队尾指针往后移动增加1后对最大空间大小取模。而对于链式队列，其优点在于没有空间的限制，可以灵活地申请空间。

基于上述考虑，我们采用以链表为基础的队列。考虑到链表无论是在头或尾的插入删除结点的时间复杂度均为O(1)，因此我们将链表头作为队列头，链表为作为队列尾。

该类主要成员函数如下：

//判断队列是否为空

inline bool empty() const {

return data.empty();

}

//获得队列中元素个数

inline int size() const {

return data.Size;

}

//清空队列

void clear() {

data.clear();

}

//在队列为插入元素

void emplace(const T& x) {

data.emplace\_back(x);

}

//在队列头删除元素

void pop() {

data.pop\_front();

}

//查询队列头的元素值

T& front() {

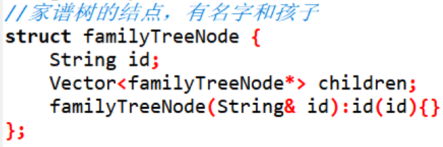
return data.getHead()->next->val;

}

2.2.5 fanmilyTreeNode类

该类包含家族树中一个结点做存储的信息，其中包括：姓名、指向孩子结点的指针数组。

代码如下图所示：



2.2.6 familyTree类

该类面向本项目进行编写，其中数据成员包括：根结点、记录共有多少个结点的整形Size以及存储键值对为名字、节点地址的哈希表。根据项目分析、其成员函数包括如下：

//通过名字递归寻找familyTreeNode\* search(String& id, familyTreeNode\* cur);

//通过hashtable寻找familyTreeNode\* \_search(String& id);

//递归删除孩子void removeChildren(familyTreeNode\* cur);

//迭代删除孩子void \_removeChildren(familyTreeNode\* cur);

//构造函数familyTree();

//析构函数~familyTree();

//完善家谱void buildFamily();

//添加子女void addChild();

//更改姓名void changeId();

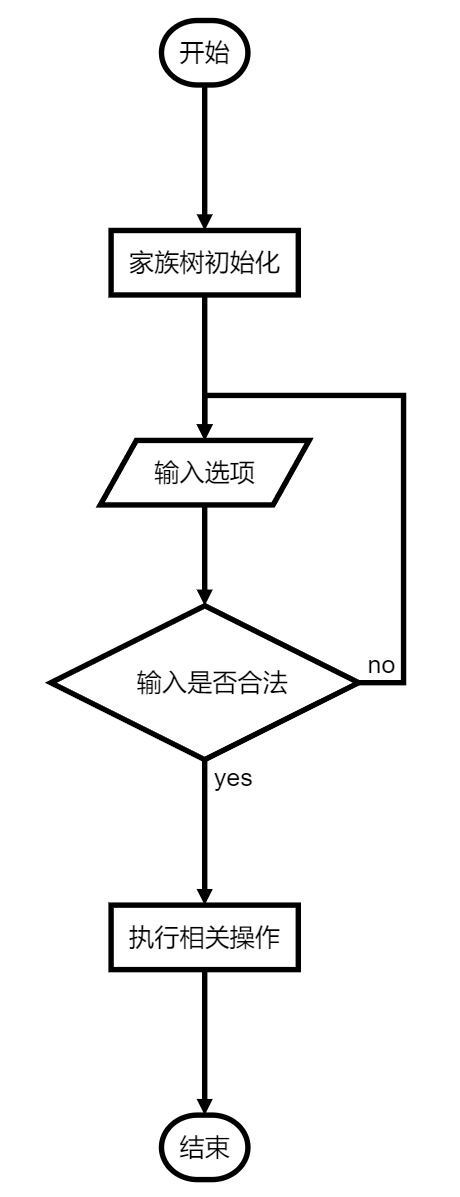
//解散家庭，没有子女void removeChildren();

//展示树形结构void show();

3 项目实现

3.1 总体实现

3.1.1 总体实现流程图



3.1.2 总体实现思路

考虑到家谱树中每个结点需要动态申请空间，为了防止出现申请空间失败而导致程序崩溃的情况，这里对new使用nothrow参数，以判断是否申请成功。原因在于本应该通过try catch捕捉异常，但有些编译器不支持该操作，于是我们考虑nothrow，在分配内存失败的时候返回空指针，由此再输出错误提示。

主要代码如下：

root = new(nothrow) familyTreeNode(tempId);

if (root == NULL){

cerr << "申请空间失败" << endl;

exit(1);

}

提供操作提示，根据用户的输入进行对应的函数实现。此处对于用户错入输入的处理同样体现了程序的健壮性，保证程序的顺利运行。

3.1.3 总体实现主要代码

主要代码如下：

while (loop){

cout << "请选择要执行的操作: ";

cin >> op;

switch (op[0]){

case 'a':

case 'A':

ft.buildFamily();

break;

case 'b':

case 'B':

ft.addChild();

break;

case 'c':

case 'C':

ft.removeChildren();

break;

case 'd':

case 'D':

ft.changeId();

break;

case 'e':

case 'E':

loop = false;

break;

default:

cout << "请输入正确操作" << endl;

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

cin.clear();

break;

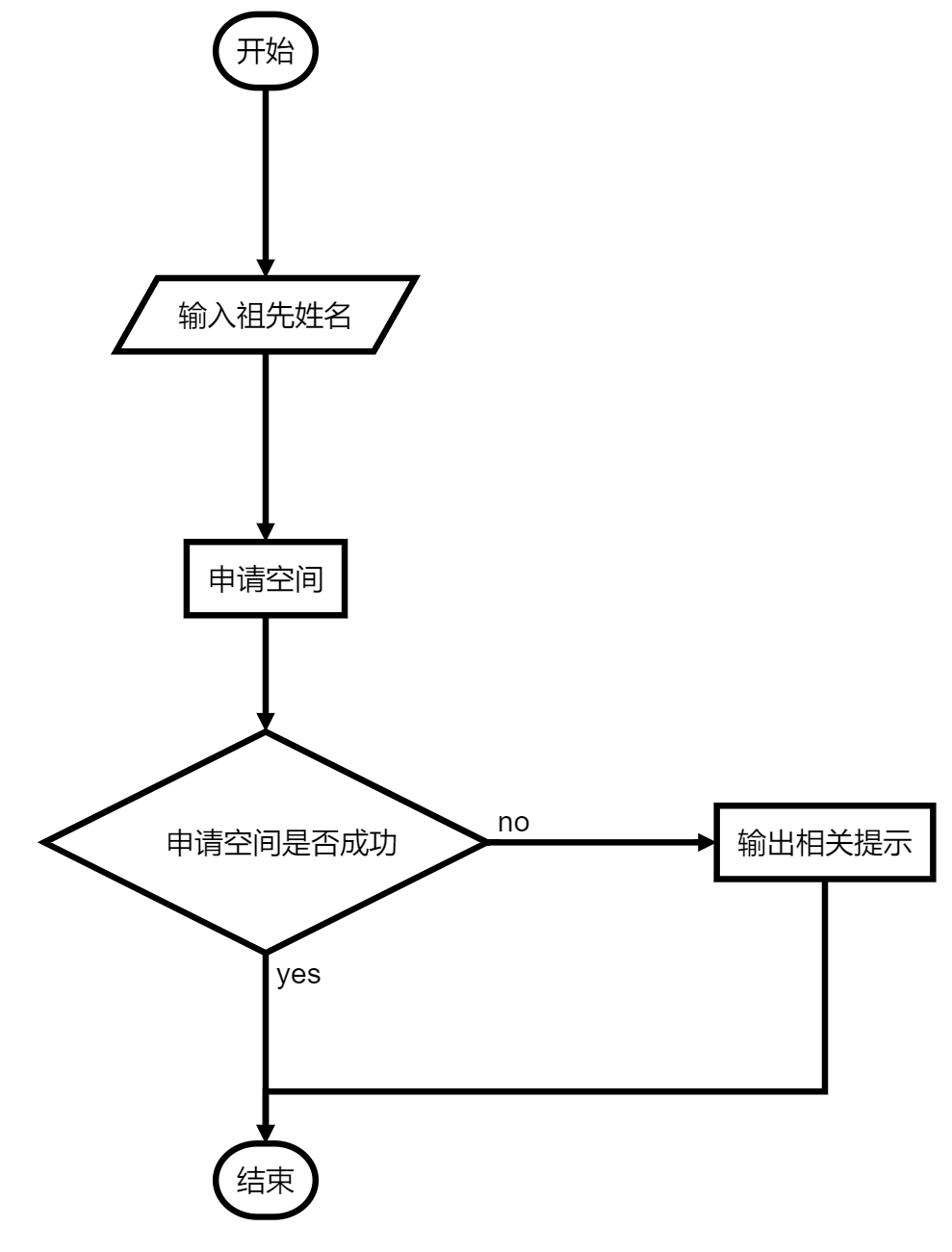
}

ft.show();

}

3.2 初始化家谱实现

3.2.1 初始化家谱实现流程图



3.2.2 初始化家谱实现思路

按照项目要求，一开始必须输入家谱中的祖先的名字以完成家谱的建立。因此我们将此环节设计于 familyTree类的构造函数中，使得在该类实例化对象的一开始，先完成祖先名字的输入与家谱的建立。

3.2.3 初始化家谱实现主要代码

cout << "首先建立一个家谱!" << endl;

cout << "请输入祖先的姓名: ";

String tempId;

cin >> tempId;

root = new(nothrow) familyTreeNode(tempId);

if (root == NULL){

cerr << "申请空间失败" << endl;

exit(1);

}

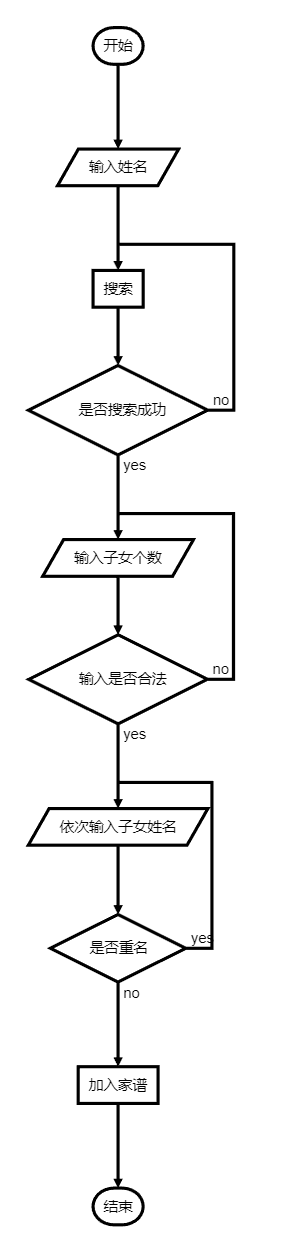
Size = 1;

//插入姓名地址键值对

ht.insert(make\_pair(tempId, root));

3.3 完善家庭成员实现

3.3.1 完善家庭成员实现流程图



3.3.2 完善家庭成员实现思路

对于完善家庭成员操作，其具体功能是：首先提示用户输入需要建立家庭的人的姓名，接着调用搜索函数查找是否存在具有该姓名的结点以及该姓名的人时候已经建立过家庭，由此判断是否需要重新输入。接着程序将提示用户输入需要建立的儿女的个数，并按照个数，需要用户依次输入儿女的姓名。这里在输入姓名申请新的结点空间之前，同样需要判断改名字是否与家谱中已有名字重名，如果是，则提示用户重新输入，以保证家谱中不会有重名的情况发生。最后注意的细节为更新结点的数量。

3.3.3 完善家庭成员实现主要代码

while (1){

cout << "请输入要建立家庭的人的姓名: ";

cin >> tempId;

fa = \_search(tempId);

if (fa == nullptr){

cout << "家谱中没有这个人，请重新输入!" << endl;

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

}

else if (!fa->children.empty()){

cout << tempId << "已经建立过家庭，请重新输入" << endl;

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

}

else

break;

}

cout << "请输入" << tempId << "的儿女个数: ";

while (1){

cin >> childNum;

if (cin.fail() || childNum <= 0){

cout << "请正确输入一个正整数!" << endl;

cin.clear();

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

}

else

break;

}

cout << "请依次输入" << tempId << "的儿女的姓名: ";

String childId;

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

cin >> childId;

//auto temp = search(childId, root);

auto temp = \_search(childId);

if (temp){//找到了

cout << "请不要与家谱已有成员重名！请重新输入" << endl;

--i;

}

else{//没有重名就加入家族树

familyTreeNode\* child = new(nothrow) familyTreeNode(childId);

if (child == NULL){

cerr << "申请空间失败" << endl;

exit(1);

}

fa->children.emplace\_back(child);

ht.insert(make\_pair(childId, child));

}

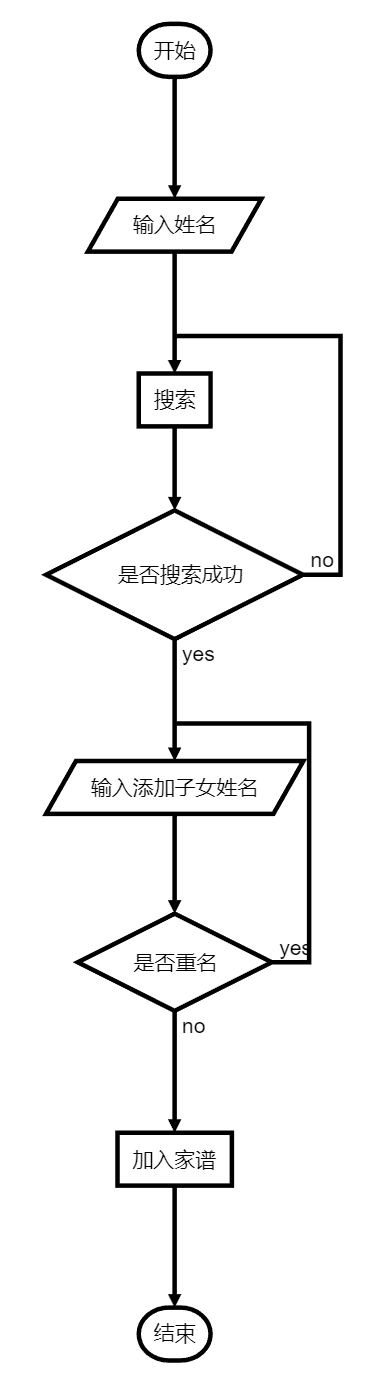
}

//更新结点数量

Size += childNum;

3.4 添加家庭成员实现

3.4.1 添加家庭成员实现流程图



3.4.2 添加家庭成员实现思路

这里首先利用循环请用户输入需要添加子女的人的姓名，并调用搜索函数确认是否存在该人，存在则退出循环，不存在则提示用户重新输入。接着请用户输入添加子女的姓名，这里同样利用循环检查是否和家谱中已有成员重名，重名则提示重新输入，反之退出循环。这里在该人的对应的结点中的指向孩子结点的指针数组中在最后一位添加，相较于实现了O(1)时间复杂度的操作。总体来说，由于搜索函数利用哈希表查找，因此该操作的时间复杂度为O(1)。

3.4.3 添加家庭成员实现主要代码

while (1){

cout << "请输入要添加儿子（或女儿）的人的姓名: ";

cin >> tempId;

fa = \_search(tempId);

if (fa == nullptr){

cout << "家谱中没有这个人，请重新输入!" << endl;

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

}

else

break;

}

String childId;

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

while (1){

cin >> childId;

auto temp = \_search(childId);

if (temp)//找到了

cout << "请不要与家谱已有成员重名！请重新输入" << endl;

else

break;

}

familyTreeNode\* child = new(nothrow) familyTreeNode(childId);

if (child == NULL){

cerr << "申请空间失败" << endl;

exit(1);

}

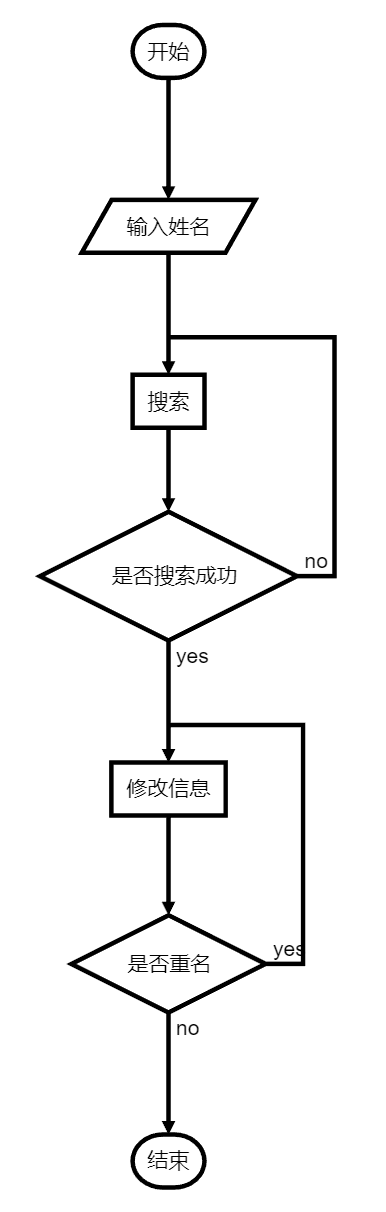
fa->children.emplace\_back(child);//此时还不能释放空间，否则访问不到

++Size;//更新结点数量

ht.insert(make\_pair(childId, child));//更新哈希表

3.5 修改家庭成员信息实现

3.5.1 修改家庭成员信息实现流程图



3.5.2 修改家庭成员信息实现思路

这里利用循环，直到用户输入了家谱中存在的成员，循环退出，接着请用户输入修改后的姓名，这里同样需要判断是否与家谱中已存在的成员重名，因此频繁调用搜索函数，这里同样利用哈希表进行搜索，付出一定的空间代价的同时获得较好的时间效率。

3.5.3 修改家庭成员信息实现主要代码

String newId;

cout << "请输入更改后的姓名: ";

while (1){

cin >> newId;

//判断新名字是否与家族树已有成员重名

auto res = \_search(newId);

if (res)//有重名

cout << "请不要与家谱已有成员重名！请重新输入" << endl;

else

break;

}

//哈希表中重新构建键值对

ht.erase(fa->id);

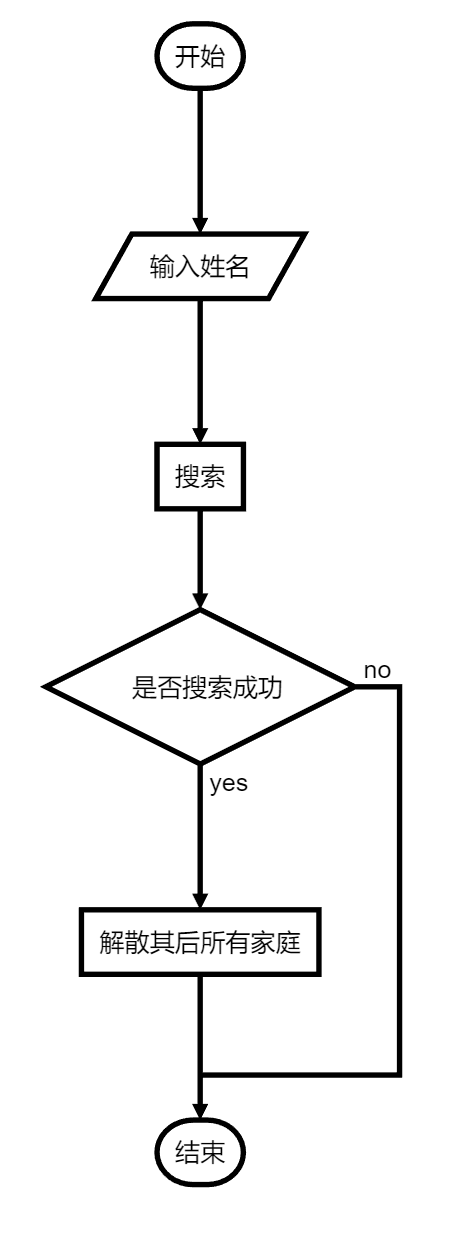
fa->id = newId;

ht.insert(make\_pair(fa->id, fa));

cout << tempId << "已更名为" << newId << endl;

3.6 解散局部家庭实现

3.6.1 解散局部家庭实现流程图



3.6.2 解散局部家庭递归方法实现思路

这里所谓解散局部家庭成员，在我们日常生活中的理解中可能仅仅是关乎一个家庭的事情，但对于本项目来说，家族树中某一个家庭的解散，意味着该家庭之后的所有家庭都不存在于家族树中。因此，这里需要我们将输入的人的家庭之后的所有家庭解散。

很容易想到该操作的递归实现，自底向上的思想。从输入的姓名的人的结点开始，我们需要检查它的所有子女结点是否还有子女结点，如果有则进入递归函数；直到其子女都没有子女了才能将本层的子女结点空间释放。

3.6.3 解散局部家庭实现递归方法主要代码

//检查所有子女是否都没有子女，有 进行递归， 都没有就删除cur->children

for (int i = 0; i < cur->children.size(); ++i){

if (!cur->children[i]->children.empty())

removeChildren(cur->children[i]);

}

//递归清除之后，删除当前children，释放孩子结点的空间

for (int i = 0; i < cur->children.size(); ++i){

ht.erase(cur->children[i]->id);

delete cur->children[i];

}

//更新结点数量

Size -= cur->children.size();

//释放存放指针的空间

cur->children.clear();

3.6.4 解散局部家庭实现非递归方法思路

考虑到递归算法的局限性以及空间时间的消耗，我们应该思考其递归展开，于是我们可以想到利用队列实自上而下的结点的删除。具体过程为：将要删除的第一个结点入队，遍历到队头结点时，将其所有孩子结点入队，实现层序遍历的删除操作。

3.6.5 解散局部家庭实现非递归方法主要代码

//利用队列删除

while (!q.empty()){

temp = q.front();

q.pop();

//当前结点的孩子入队

for (int i = 0; i < temp->children.size(); ++i)

q.emplace(temp->children[i]);

//释放存放孩子指针数组

temp->children.clear();

//哈希表中删除结点

ht.erase(temp->id);

//释放当前结点

delete temp;

--Size;//更新数量

}

//释放最老的

cur->children.clear();

3.7 家族树的展示实现

3.7.1 家族树展示实现思路

考虑到用户友好性，我们通过树形结构将家谱更加清晰的展示。参考linux系统中tree的功能，利用深度优先的思想，从上往下依次输出。这里定义结构体存储结点的地址，是否是最后一个，以及层数三个信息以便输出。

3.7.2 家族树展示实现主要代码

//参数：isLast是否是最后一个，node家族树的结点，level是第几层

struct info {

info(bool isLast = false, familyTreeNode\* node = nullptr, int level = 0)

:isLast(isLast), node(node), level(level) {}

bool isLast;

familyTreeNode\* node;

int level;

};

List<info> stk;

info temp;

stk.emplace\_back(info(Size == 1, root, 0));

//树型结构

while (!stk.empty())

{

//取栈顶元素

temp = stk.getTail()->val;

stk.pop\_back();

for (int i = 0; i < temp.level; ++i)

cout << "│ ";

//是否是孩子们中的最后一个

if (temp.isLast)

cout << "└── ";

else

cout << "├── ";

cout << temp.node->id << endl;

//放入temp的孩子，注意是否是最后一个孩子

int childSize = temp.node->children.size();

for (int i = childSize - 1; i >= 0; --i)

stk.emplace\_back(info(i == childSize - 1

, temp.node->children[i], temp.level + 1));

}

3.8 搜索实现

3.8.1 搜索实现思路

关于在树型结构中的搜索，我们首先想到的是深度优先搜索或广度优先搜索，但考虑到在其他函数中搜索函数的频繁调用，我们更倾向于利用哈希表搜索。

3.8.2 递归深度优先搜索实现主要代码

递归深度优先搜索代码如下：

//递归结束条件

if (cur == nullptr)

return nullptr;

if (cur->id == id)

return cur;

//向子树寻找

familyTreeNode\* temp = nullptr;

for (int i = 0; i < cur->children.size(); ++i)

{

temp = search(id, cur->children[i]);

if (temp)//temp不为空说明找到了直接退出循环

break;

}

return temp;

3.8.3 哈希表搜索实现主要代码

哈希表搜索如下：

//通过哈希表寻找返回 id, 结点地址 键值对

auto res = ht.find(id);

//如果找到，返回的地址非空

if (res)

return res->kv.second;

//找不到返回空地址

return NULL;

3.9 哈希表的实现

3.9.1 哈希表的大小

关于哈希表的大小，我们利用质数表来规定。利用质数作为哈希表的大小，在其做取余运算的时候拥有极好的均匀分布的特性，由此减少哈希冲突，更好地提高搜索效率。

质数表如下：

//28个素数作为hashtable的size

const int primeNum = 28;

static const unsigned long primeList[primeNum] ={

53ul, 97ul, 193ul, 389ul, 769ul,

1543ul, 3079ul, 6151ul, 12289ul, 24593ul,

49157ul, 98317ul, 196613ul, 393241ul, 786433ul,

1572869ul, 3145739ul, 6291469ul, 12582917ul, 25165843ul,

50331653ul, 100663319ul, 201326611ul, 402653189ul, 805306457ul,

1610612741ul, 3221225473ul, 4294967291ul

};

3.9.1 哈希表的扩大

关于对哈希表的大小的扩展，这里通过给出当前哈希表的大小，查找在质数表中当前大小的下一个指数。这里使用二分查找，可以提高查找效率。

//返回大于n的最小素数，扩容的时候需要

inline unsigned long getNextPrime(unsigned long num){

if (num == 4294967291ul)//最后一位数字了没有下一位了

return num;

//二分查找

int l = 0, r = primeNum - 1;

while (l < r){

int mid = (l + r) >> 1;//右移除以二

if (primeList[mid] <= num)

l = mid;

else

r = mid - 1;

}

//返回该数字的下一位

return primeList[l + 1];

}

3.9.3 哈希函数

考虑到利用哈希表的时候常常以字符串作为关键码，因此哈希函数的模板用String类提前写好。再将其作为仿函数作为hashtable类的参数模板。

代码如下：

template<>

struct hashfunc<String> {

size\_t operator()(const String& str) {

size\_t hash = 0;

size\_t Size = str.length() + 1;

for (size\_t i = 0; i < Size; ++i) {

hash \*= 31;

hash += str.c\_str()[i];

}

return hash;

}

};

3.9.1 哈希表的函数实现方法

对于hashtable类的实现，首先定义参数模板，其中包括键、值的类型以及哈希函数。其中的私有成员包括hash节点的指针数组、申请的指针数组的大小，哈希表中元素的个数以及哈希函数。通过哈希函数将传入的key进行映射到存储位置，根据操作的不同，其实质就是对链表的操作，这里以查找为例。

hashtable类的定义：

template<class K, class V, class Func = hashfunc<K>>

class hashtable;

私有成员：

typedef hashNode<K, V> Node;

Node\*\* table = NULL;

size\_t Size = 0;

size\_t capacity = 0;

Func hf;

查找函数代码实现：

template<class K, class V, class Func>

hashNode<K, V>\* hashtable<K, V, Func>::find(const K& key) {

size\_t index = hf(key) % capacity;//通过哈希函数映射存储地址

Node\* cur = table[index];

while (cur) {

if (cur->kv.first == key)//找到了退出循环

break;

cur = cur->next;//没找到继续找

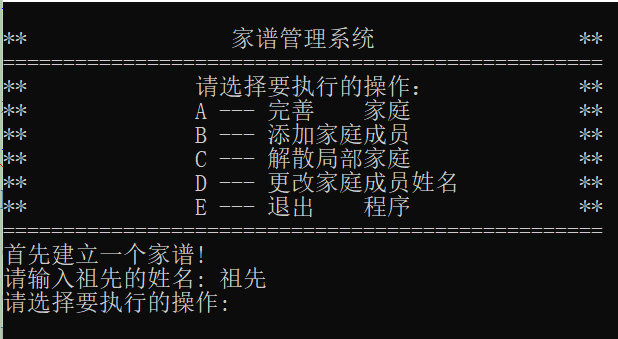
}

return cur;

}

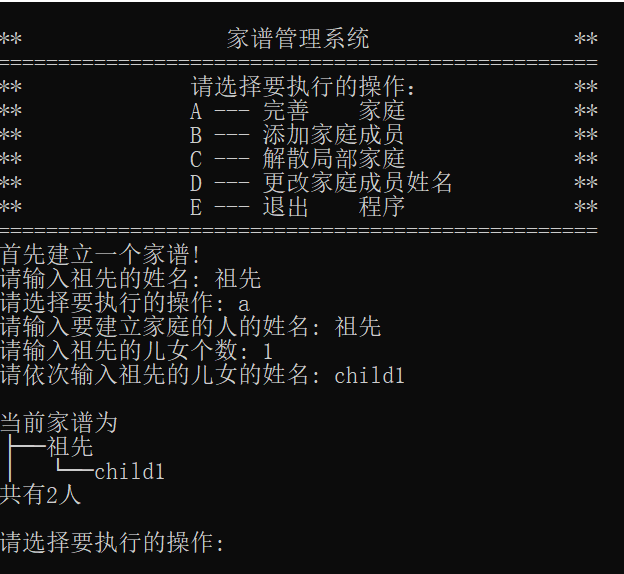
4 项目测试

4.1 初始化家谱测试

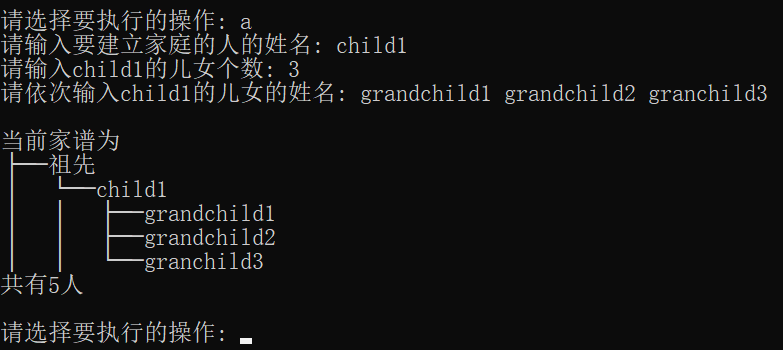


4.2 完善家庭

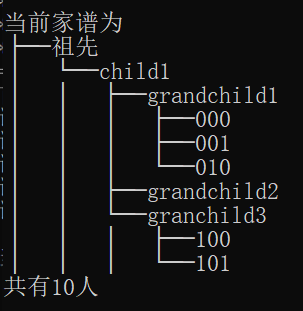
4.2.1 添加单个子女测试



4.2.2 添加多个子女测试

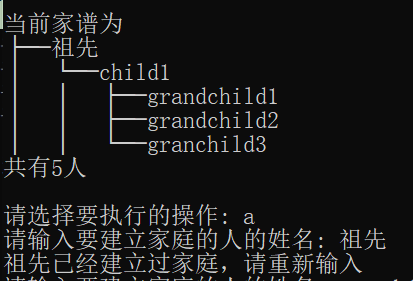


4.2.3 多代多个子女测试

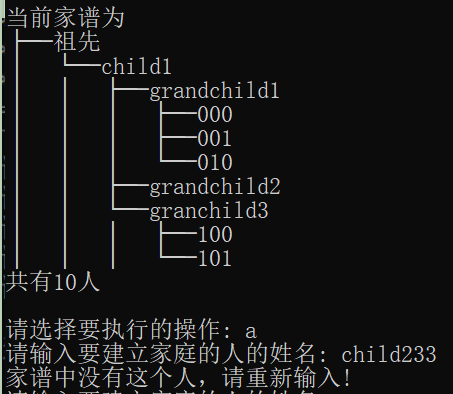


4.3 完善家庭错误输入测试

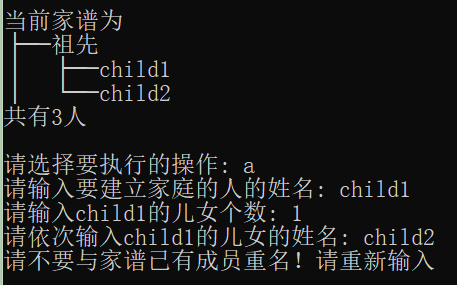
4.3.1 尝试建立已建立家庭个人的家庭



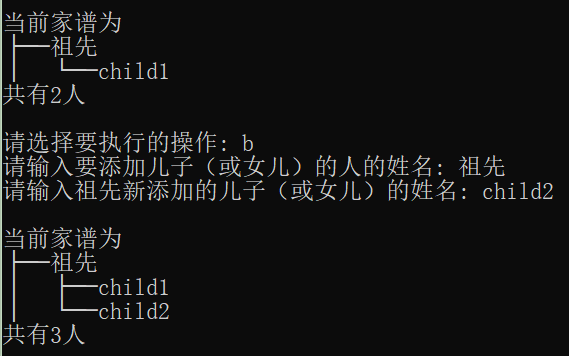
4.3.2 输入家谱中不存在成员



4.3.3 输入姓名与已有成员重名

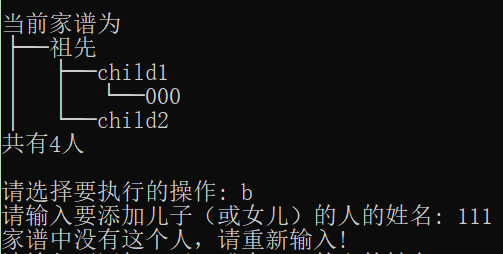


4.4 添加家庭成员正确测试

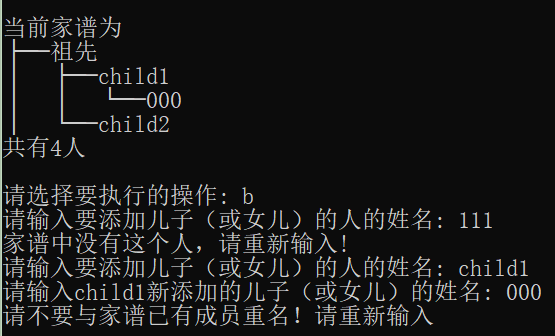


4.5 添加家庭成员输入错误测试

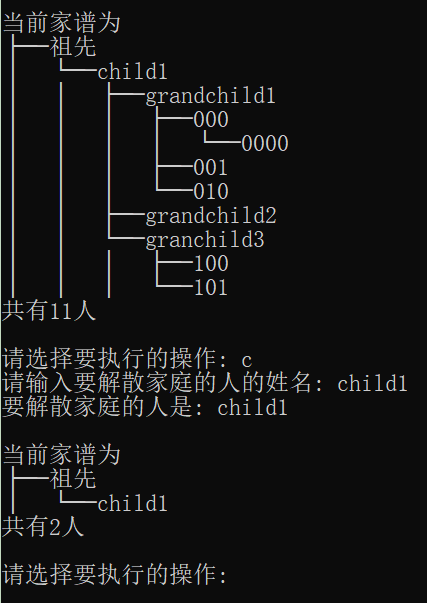
4.5.1 输入家谱中不存在成员



4.5.2 输入姓名与已有成员重名

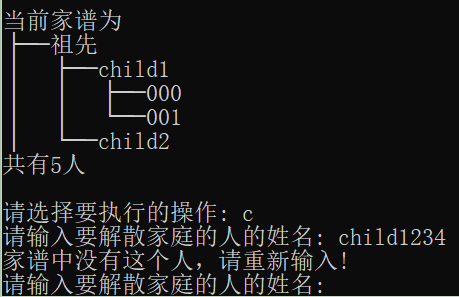


4.6 解散局部家庭正确测试

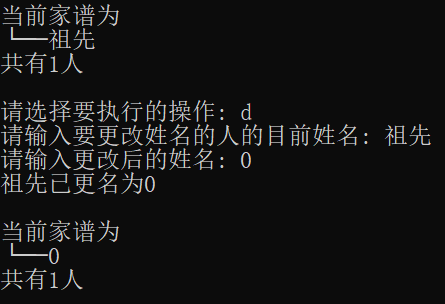


4.7 解散局部家庭输入错误测试

4.7.1 输入家谱中不存在成员

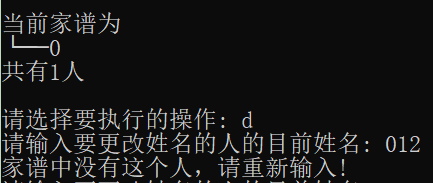


4.8 更改家庭成员姓名正确测试

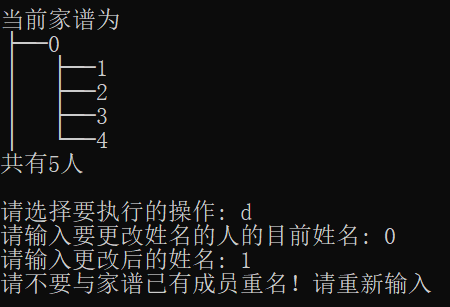


4.9 更改家庭成员姓名输入错误测试

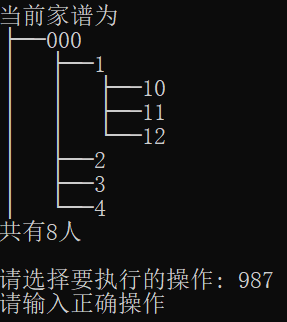
4.9.1 输入家谱中不存在成员



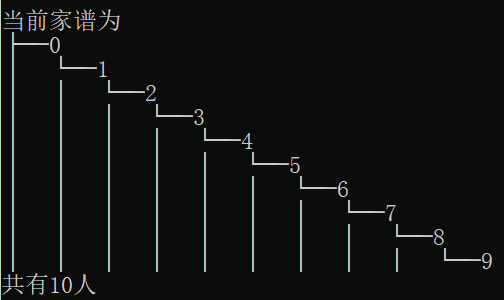
4.9.2 更改为与家谱中重名姓名



4.10 输入错误操作测试



4.11 单链家谱



4.12 扁平家谱

