同济大学
TONGJI UNIVERSITY

数据结构课程设计

项目说明文档

家谱管理系统

姓　　名：　　马小龙

学　　号：　　2353814

学　　院：计算机科学与技术学院（软件学院）

专　　业：　　软件工程

指导教师：　　张颖

二〇二三年十一月十九日

目录

第 1 章 项目分析

1.1 项目背景分析

家谱，又称族谱、宗谱等。家谱是一种以表谱形式，记载一个家族的世系繁衍及重要人物事迹的书。同时，家谱是中华文明史中具有平民特色的文献，记载的是同宗共祖血缘集团世系人物和事迹等方面情况的历史图籍家谱属珍贵的人文资料，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。家谱既是国家历史文献的重要组成部分，也是中华民族精神血脉赓续、文化传承的重要依托。

在数字化信息管理的背景下，家谱管理系统的构建不仅可以帮助用户保存和追溯家族信息，还能通过便捷的操作对家族成员信息进行维护和更新，为家族历史的记录与传承提供支持。

家谱管理系统旨在通过现代化手段实现家族信息的高效记录、存储和查询，为家庭传承提供便利。通过数字化管理，用户不仅可以追溯家族历史，还能维护血缘关系，记录家族重要事件，使得传统家谱文化得以传承和发扬。

本项目是针对家谱管理的模拟实现，主要目标是通过程序完成家谱数据的建立、查询、修改、插入和删除等核心功能，以帮助用户管理家族成员信息并保存家族历史。

1.2 项目需求分析

家谱管理系统旨在帮助用户对家族成员的信息进行高效管理，包括家谱的创建、维护、更新和扩展等操作。为满足用户需求，系统需提供完善的功能来实现家族成员信息的动态操作，并通过合理的数据结构支持成员关系的表达。同时，系统需要设计直观的用户界面，确保操作简单易懂，让用户能快速上手使用。

在技术实现上，系统应采用适合表示层次结构的树形数据结构来构建家谱，保证数据组织的高效性和关联性。此外，系统需具备良好的健壮性和容错能力，能够妥善处理用户的输入错误或异常操作，避免出现数据丢失或系统中断的问题。

1.3 项目功能分析

本项目旨在开发一套高效的家谱管理系统系统，以解决在家谱管理中面临的各种挑战。系统支持完善家谱、添加家庭成员、解散局部家庭、更改家庭成员姓名等基本功能，从而实现对家谱的高效管理。以下是项目主要功能的详细信息。

1.3.1 完善家谱功能

该功能允许用户为家谱中的任意成员添加后代信息，逐步构建完整的家族树。

1.3.2 添加家庭成员功能

该功能支持新增家族成员至家谱，并将其与现有成员建立关系，便于记录家族扩展情况。

1.3.3 解散局部家庭功能

该功能允许用户删除指定成员及其后代，从而维护数据的正确性和时效性。

1.3.4 更改家庭成员姓名功能

项目提供编辑成员姓名的功能，以应对录入错误或成员更改姓名等场景。

1.3.8 异常处理功能

处理非法输入或操作带来的异常情况，确保系统运行的稳定性和用户数据的安全性。

第 2 章 项目设计

2.1 数据结构设计

本项目设计了两种数据结构，分别是二叉树和 String（string 类型自己实现）。

（一）使用二叉树作为家谱信息的主要数据结构，采用链表的方式存整个家谱的信息，主要是基于以下方面：

1.适应家谱的层次结构：使用二叉树表示家谱中的层次关系，每个节点可直观地存储一个家庭成员的信息，使用其左子树表示其后代，右子结点表示其兄弟姐妹，从而清晰地反映家谱的血缘关系。

2.动态扩展性：二叉树结构具有动态扩展的能力，可以随着家族规模的增长，灵活地增加新的节点，而无需对现有结构进行大幅调整。这种特性非常适合家谱信息的长期维护和管理。

3.时间和空间效率：二叉树在遍历、搜索、插入和删除操作上比一般树更加简洁高效，逻辑清晰且实现简单。在存储方面，二叉树较于一般树更为节省资源，每个节点仅需存储两个子节点的地址，避免了存储空间的过度消耗。

4.便于实现递归操作：二叉树的递归性质使其在实现复杂的操作（如遍历、统计成员数量等）时非常方便。这种特性简化了家谱管理系统的逻辑设计和代码实现。

（二）使用 String 来储存家族成员的姓名主要是基于一下方面：

1.string 类型具有良好的可读性和易用性，使得处理和展示家族成员的姓名在代码中，string 的使用能够增强可维护性，方便后续的调试和修改。

2.string 类型的动态特性使得其在处理长度不确定的文本时非常灵活。家族成员的姓名长度可能各异，string 能够自动管理内存，适应不同长度的内容，避免了固定大小数组带来的限制。

3.string 也能够方便地与其他数据结构进行结合。

2.2 结构体与类设计

为了使二叉树更加通用，本链表设计为模板设计。

2.2.1 二叉树结点（BinTreeNode）设计

2.2.1.1 概述

BinTreeNode 用于储存信息，包括结点储存的具体信息内容和左右孩子的地址。

2.2.1.2 结构体定义

```
template <typename T>
struct BinTreeNode {
    T data;
    BinTreeNode<T>* leftChild, * rightChild;
    BinTreeNode() : leftChild(nullptr), rightChild(nullptr) {}
    BinTreeNode(T x, BinTreeNode<T>* left =nullptr, BinTreeNode<T>* right
= nullptr) : data{x}, leftChild(left), rightChild(right) { }
};
```

2.2.1.3 数据成员

T data: 数据域，存储结点的数据

BinTreeNode<T>* leftChild: 指针域，指向左孩子的指针

BinTreeNode<T>* rightChild: 指针域，指向右孩子的指针

2.2.1.4 构造函数

BinTreeNode()；

构造函数，初始化指针域。

BinTreeNode(T x, BinTreeNode<T>* left =nullptr, BinTreeNode<T>* right = nullptr)；

构造函数，初始化数据域和指针域。

2.2.2 二叉树（BinaryTree）设计

2.2.2.1 概述

该通用模板类 BinaryTree 用于表示二叉树。链表结点由 BinTreeNode 结构体表示，其中包含数据和指向左右孩子的指针。该链表提供了一系列基本操作函数，包括左右孩子的插入、 地址获取，具体结点的查找，根节点的获取，树的删除等，以及二叉树的构造和析构，满足了常见的二叉树操作需求。

2.2.2.2 BinaryTree 类定义

```
template <typename T>
class BinaryTree {
```

```
public:
    BinaryTree();
    BinaryTree(T& item);
    ~BinaryTree();
    bool IsEmpty();
    BinTreeNode<T>* Parent(BinTreeNode<T>* current);
    BinTreeNode<T>* LeftChild(BinTreeNode<T>* current);
    BinTreeNode<T>* RightChild(BinTreeNode<T>* current);
    T Data(BinTreeNode<T>* current);
    bool rootInsert(const T& item);
    bool leftInsert(BinTreeNode<T>* current, const T& item);
    bool rightInsert(BinTreeNode<T>* current, const T& item);
    void StoreRoot(BinTreeNode<T>* subTree) { root = subTree; }
    void ChangeLeftChild(BinTreeNode<T>* parent,BinTreeNode<T>*
child);
    void ChangeRightChild(BinTreeNode<T>* parent,BinTreeNode<T>*
child);
    BinTreeNode<T>* GetRoot();
    BinTreeNode<T>* Search(const T& item, BinTreeNode<T>* subTree);
    void Destory(BinTreeNode<T>* subTree);
private:
    BinTreeNode<T>* root;
};
```

2.2.2.3 私有数据成员

BinTreeNode<T>* root: 指向根结点的指针

2.2.2.4 构造函数与析构函数

BinaryTree();

默认构造函数，创建一个空树。

BinaryTree(T& item);

构造函数，创建一个包含根节点结点树。

~BinaryTree();

析构函数，释放二叉树的内存资源，包括所有结点的内存。

2.2.2.5 公有成员函数

bool IsEmpty()；

检查二叉树是否为空树；

BinTreeNode<T>* Parent(BinTreeNode<T>* current);

获得父母结点的指针；

BinTreeNode<T>* LeftChild(BinTreeNode<T>* current);

获取左孩子结点指针；

BinTreeNode<T>* RightChild(BinTreeNode<T>* current);

获取有孩子结点指针；

T Data(BinTreeNode<T>* current);

获取当前结点储存的数据；

bool rootInsert(const T& item);

插入根节点；

bool leftInsert(BinTreeNode<T>* current, const T& item);

插入左孩子结点；

bool rightInsert(BinTreeNode<T>* current, const T& item);

插入右孩子节点；

void StoreRoot(BinTreeNode<T>* subTree)；

储存或更改根结点数据；

void ChangeLeftChild(BinTreeNode<T>* parent,BinTreeNode<T>* child)；

更改左孩子结点数据；

void ChangeRightChild(BinTreeNode<T>* parent,BinTreeNode<T>* child)；

更改右孩子结点数据

BinTreeNode<T>* GetRoot()；

获取根节点指针

BinTreeNode<T>* Search(const T& item, BinTreeNode<T>* subTree);

在二叉树查找指定节点；

void Destory(BinTreeNode<T>* subTree);

摧毁二叉树，释放内存空间。

2.2.3 String 类设计

2.2.3.1 概述

String 为自定义数据类型，可以用来储存及处理字符串，例如姓名、性别和报考类别。

2.2.3.2 String 类定义

```
class String {
private:
    char* _str;
    size_t _str_len;
    size_t _str_cap;
public:
    static constexpr size_t npos = -1;
    typedef char* iterator;
    typedef const char* const_iterator;
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    String(const char* str = "");
    String(const String& s);
   ~String();
    const char* C_str ()const;
    size_t Size()const;
    size_t Capacity()const;
    void Swap(String& s);
    void Reserve(size_t n);
    void Resize(size_t n, char ch = '\0');
    void PushBack(char ch);
    void Append(const char* str);
    void Insert(size_t pos, char ch, size_t n=1);
    void Insert(size_t pos, const char* str);
    void Erase(size_t pos, size_t len = npos);
    size_t Find(char ch, size_t pos = 0) const;
    size_t Find(const char* str, size_t pos = 0) const;
    String Substr(size_t pos = 0, size_t len = npos) const;
    void Clear();
    static int Memcmp(const void *dst,const void *src,size_t n) ;
    static int Strlen(const char* src);
```

```
    static char* Strstr(const char* str1, const char* str2);
    String& operator+=(char ch);
    String& operator+=(const char* str);
    bool operator<(const String& s) const;
    bool operator==(const String& s) const;
    bool operator<=(const String& s) const;
    bool operator>(const String& s) const;
    bool operator>=(const String& s) const;
    bool operator!=(const String& s) const;
    char& operator[](size_t pos);
    const char& operator[](size_t pos) const;
};
```

2.2.3.3 私有数据成员

char* _str: 存储字符串的字符数组

size_t _str_len: 当前长度

size_t _str_cap: 当前容量

2.2.3.4 公有数据成员

static constexpr size_t npos = -1: npos 常量表示无效位

2.2.3.5 构造函数与析构函数

String(const char* str = "");

构造函数，默认空字符串

String(const String& s);

拷贝构造函数

~String();

析构函数，释放字符串所占内存

2.2.3.6 公有成员函数

iterator begin();

获取迭代器开始位置。

iterator end();

获取迭代器结束位置。

const_iterator begin() const;

获取常量迭代器开始位置。

const_iterator end() const;

获取常量迭代器结束位置。

const char* C_str ()const;

获取当前字符串。

size_t Size()const;

获取字符串长度。

size_t Capacity()const;

获取当前容量。

void Swap(String& s);

交换字符串内容。

void Reserve(size_t n);

预留容量，扩展字符串容量。

void Resize(size_t n, char ch = '\0');

调整当前字符串大小。

void PushBack(char ch);

在字符串末尾添加字符。

void Append(const char* str);

在字符串末尾添加字符串。

void Insert(size_t pos, char ch, size_t n=1);

在指定位置插入 n 个字符。

void Insert(size_t pos, const char* str);

在指定位置插入字符串。

void Erase(size_t pos, size_t len = npos);

删除指定位置指定长度的字符串。

size_t Find(char ch, size_t pos = 0) const;

在字符串中查找指定字符位置。

size_t Find(const char* str, size_t pos = 0) const;

在字符串查找指定字符串位置。

String Substr(size_t pos = 0, size_t len = npos) const;

提取子字符串。

void Clear();

清空字符串。

static int Memcmp(const void *dst,const void *src,size_t n) ;

检查字符串长度是否相等，根据返回结果进行判断 1 为 dst 大于 src，-1 为 dst 小于 srcdst ，0 为等于 src。

static int Strlen(const char* src);

获取字符串长度。

static char* Strstr(const char* str1, const char* str2);

查找子字符串。

String& operator+=(char ch);

+=运算符重载，将新的字符放在字符串末尾。

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

+=运算符重载，将新的字符串放在字符串末尾。

bool operator<(const String& s) const;

小于运算符重载，用于字符串大小比较。

bool operator==(const String& s) const;

等于运算符重载，用于字符串大小比较。

bool operator<=(const String& s) const;

小于等于运算符重载，用于字符串大小比较。

bool operator>(const String& s) const;

大于运算符重载，用于字符串大小比较。

bool operator>=(const String& s) const;

大于等于运算符重载，用于字符串大小比较。

bool operator!=(const String& s) const;

不等于运算符重载，用于字符串大小比较。

char& operator[](size_t pos);

下标运算符重载，用于查找指定位置字符。

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

下标运算符重载，用于查找指定位置字符。

String& operator=(const String& s);

赋值运算符重载，用于 String 类的赋值。

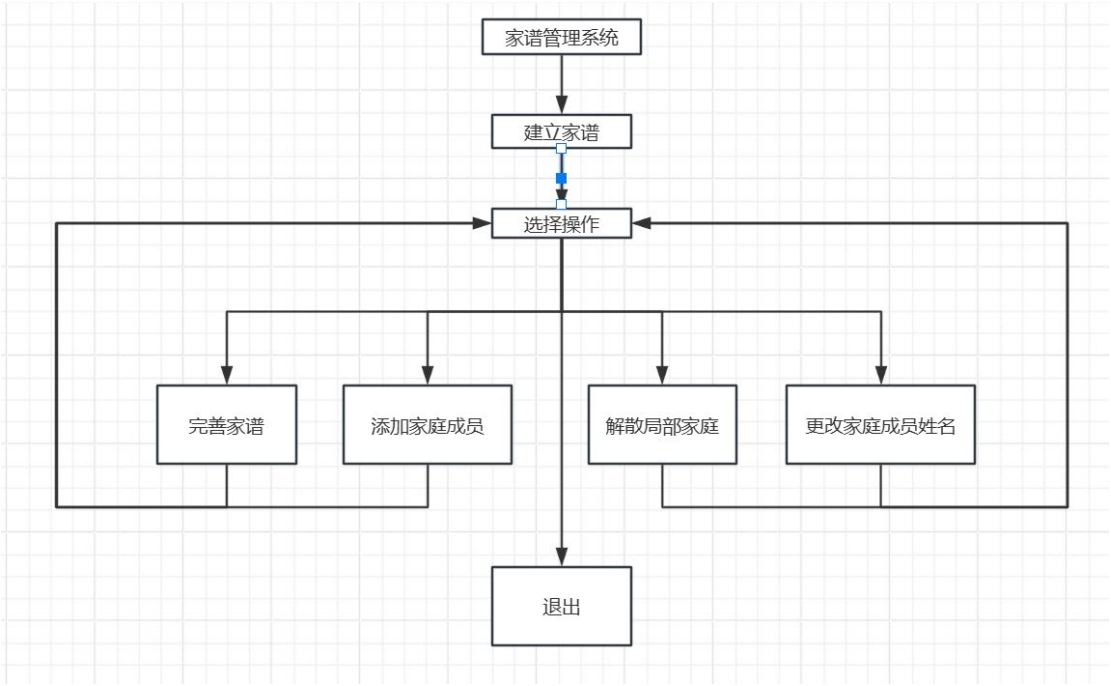
2.2.3.7 输入重载

inline std::istream& operator>>(std::istream& input, String& s)；

输入流操作符重载，规范 String 数据类型的输入

2.3 项目框架设计

2.3.1 项目框架流程图



2.3.2 项目框架流程

1.进入考试报名系统。

2.输入祖先姓名，以建立家谱。

3.通过循环结构操作家谱管理系统，调用包括完善家谱、添加家庭成员、解散局部家庭、更改家庭成员姓名等操作函数，以实现所需的功能。

4.实现特定功能后，输出被操作成员的第一代后代或修改结果；

4.用户选择退出后，退出循环及家谱管理系统。

第 3 章 项目功能实现

3.1 项目主体架构

3.1.1 实现思路

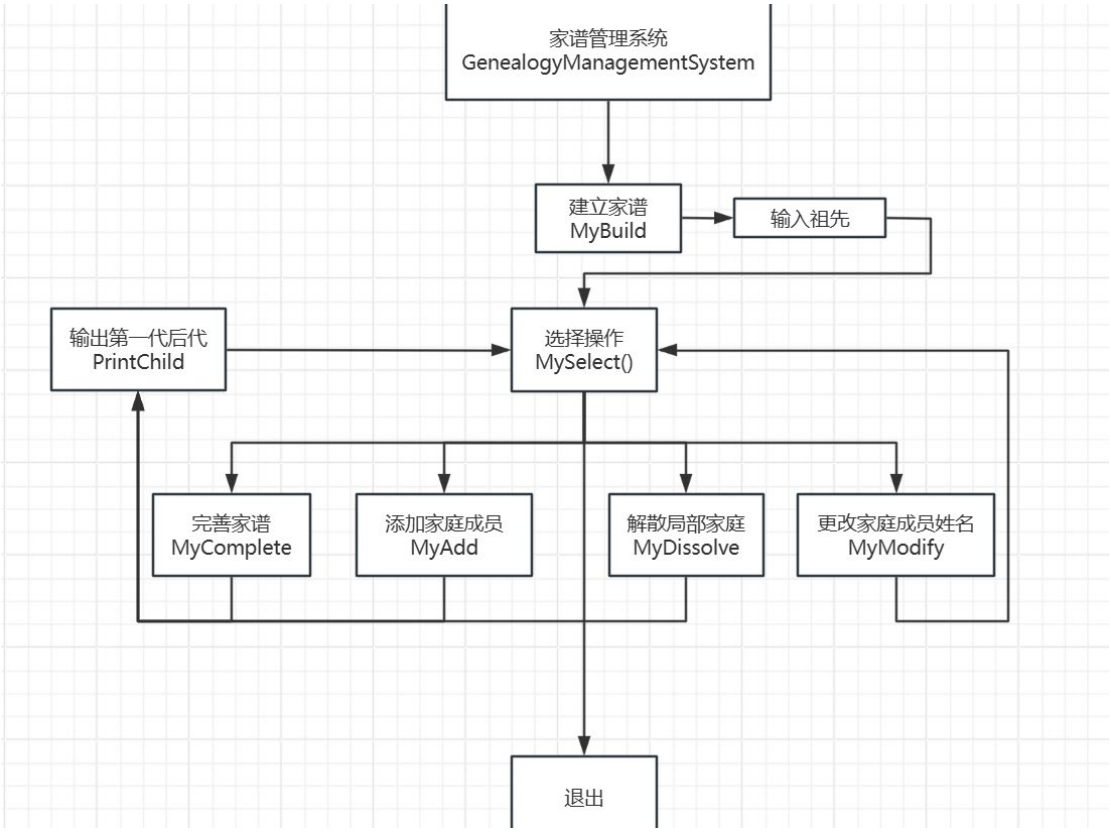
1.进入 GenealogyManagementSystem 函数以进入家谱管理系统。

2.调用 MyBuild 函数开始建立家谱，输入祖先姓名完成考生信息系统建立。

3.进入循环，调用 MySelect 函数选择要进行地操作后，回到循环，根据选择的操作开始执行不同的函数，包括 MyComplete 完善家谱、MyAdd 添加家庭成员、MyDissolve 解散局部家庭、MyModify 修改家庭成员姓名，之后再调用 PrintChild 函数打印当前被操作成员的第一代后代（修改姓名操作不调用），然后循环执行；直到选择退出则退出循环。

4.退出家谱管理系统。

3.1.2 流程图



3.1.3 核心代码

```
void GenealogyManagementSystem()
{
    std::cout<<"+---------------------------------------+\n";
    std::cout<<"|              家谱管理系统             |\n";
    std::cout<<"|   Genealogy Management System |\n";
    std::cout<<"+---------------------------------------+\n";
    BinaryTree<mine::String> genealogy;
    MyBuild(genealogy);
    std::cout<<"+-----------------------------------+\n";
    std::cout<<"|             操作类型              |\n";
    std::cout<<"|     A ---  完善家谱               |\n";
    std::cout<<"|     B ---  添加家庭成员           |\n";
    std::cout<<"|     C ---  解散局部家庭           |\n";
    std::cout<<"|     D ---  更改家庭成员姓名       |\n";
    std::cout<<"|     E ---  退出程序员             |\n";
    std::cout<<"+-----------------------------------+\n";
    while(true) {
        char select=MySelect();
        if(select=='A')
            MyComplete(genealogy);
        else if(select=='B')
            MyAdd(genealogy);
        else if(select=='C')
            MyDissolve(genealogy);
        else if(select=='D')
            MyModify(genealogy);
        else
            break;
    }
    std::cout<<"\n 成功退出家谱管理系统！ \n";
}
```

3.1.4 示例

3.2 完善家谱功能

3.2.1 实现思路

完善家谱功能的函数名为 MyComplete，完善家谱功能实现的思路为：

1. 首先输入需要添加后代的成员的姓名，根据输入姓名调用 genealogy.Search 函数查找对应的结点指针；

2. 若查找成功且该结点不存在左孩子（即没有建立家庭），返回结点对应的指针并进行下一步操作；否则提示错误信息，退出函数；

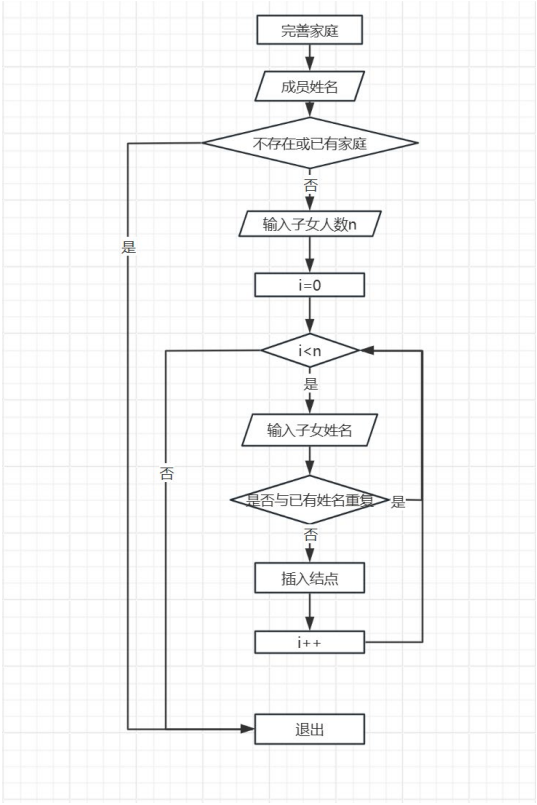
3. 进入一个 While 无限循环，输入需要建立的家庭的子女人数；此处需要输入一个正整数，才可以退出 While 循环，否则需要重新输入；

4. 根据输入的人数，进入一个 for 循环，依次输入其子女的姓名；每次输入时，都需要保证其子女与家族中任何人都不能重名，否则需要重新输入；

5. 输入即插入完成后，打印该成员第一代子女的信息；

6. 退出 MyComplete，该功能实现完毕。

3.2.2 流程图



3.2.3 核心代码

```
void MyComplete(BinaryTree< mine::String>& genealogy)
{
    std::cout<<"请输入要建立家庭的人的姓名：";
    mine::String parent;
    std::cin>>parent;
    BinTreeNode< mine::String>* root=genealogy.GetRoot();
    BinTreeNode< mine::String>* current=genealogy.Search(parent,root);
    if(current==nullptr) {
        std::cout<<"未查找到该成员！\n\n";
        return;
    }
    if(genealogy.LeftChild(current)) {
        std::cout<<"该成员已存在家庭，只能添加家庭成员或者解散局部
家庭！\n";
        return;
    }
```

```
        double n;
        while(true) {
            std::cout<<"请输入"<<parent<<"的儿女的人数：";
            std::cin>>n;
            if(std::cin.fail()||n!=static_cast<int>(n)) {
                std::cout<<"输入非法，请重新输入！\n";
                ClearBuffer();
                continue;;
            }
            break;
        }
        int number = static_cast<int>(n);
        BinTreeNode< mine::String>* temp=current;
        std::cout<<"请依次输入"<<parent<<"的儿女的姓名：\n";
        for(int i=0;i<number;i++) {
            mine::String child;
            std::cin>>child;
            if(genealogy.Search(child,root)) {
                std::cout<<"族谱中的名字不能重复！请重新输入。\n";
                i-=1;
                continue;
            }
            if(i==0) {
                genealogy.leftInsert(temp,child);
                temp=genealogy.LeftChild(temp);
            }
            else {
                genealogy.rightInsert(temp,child);
                temp=genealogy.RightChild(temp);
            }
        }
        PrintChild(genealogy,parent,current);
    }
```

3.2.4 示例

```
+-----------------------------------------+
|              家谱管理系统               |
|        Genealogy Management System      |
+-----------------------------------------+
首先建立一个家谱！
请输入祖先的姓名：P0
家谱管理系统建立成功！
此族谱的祖先为：P0

+-------------------------------------+
|            操作类型                 |
|          A --- 完善家谱             |
|          B --- 添加家庭成员         |
|          C --- 解散局部家庭         |
|          D --- 更改家庭成员姓名     |
|          E --- 退出程序员           |
+-------------------------------------+
请选择要执行的操作：A
请输入要建立家庭的人的姓名：P0
请输入P0的儿女的人数：4
请依次输入P0的儿女的姓名：
P1 P2 P3 P4
P0的第一代子孙为：
P1 P2 P3 P4
```

3.3 添加家庭成员功能

3.3.1 实现思路

添加家庭成员功能的函数名为 MyAdd，添加家庭成员功能实现的思路为：

1. 首先输入需要添加家庭成员的家谱成员的姓名，根据输入姓名调用 genealogy.Search 函数查找对应的结点指针；

2. 若查找成功则进行下一步操作；否则提示错误信息，退出函数；

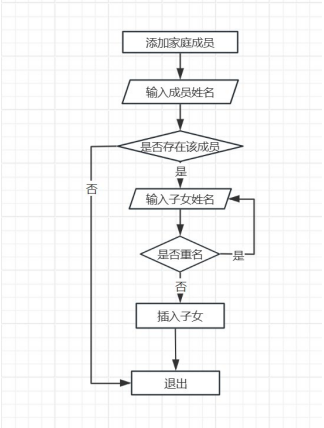
3. 进入一个 while 循环，输入该成员需要添加的子女的姓名，输入时，需要保证该子女与家族中任何人都不能重名，否则需要重新输入；

4. 检查该成员左孩子是否存在，若不存在，则将新子女作为左孩子；若存在，则将新子女作为其左孩子一右到底的结点的右孩子；

5. 插入完成后，打印该成员第一代子女的信息；

6. 退出 MyAdd，该功能实现完毕。

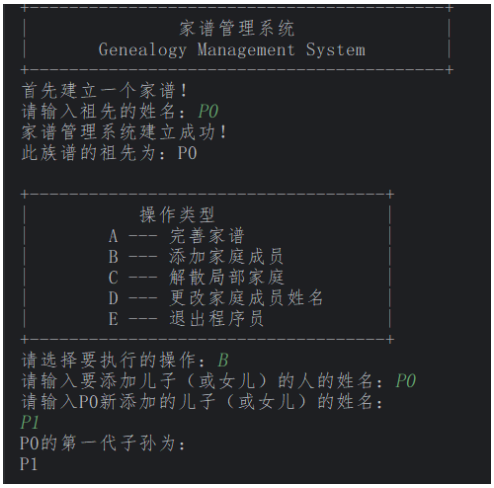
3.3.2 流程图



3.3.3 核心代码

```
void MyAdd(BinaryTree< mine::String>& genealogy) {
    std::cout<<"请输入要添加儿子（或女儿）的人的姓名：";
    mine::String parent;
    std::cin>>parent;
    BinTreeNode< mine::String>* root=genealogy.GetRoot();
    BinTreeNode< mine::String>* current=genealogy.Search(parent,root);
    if(current==nullptr) {
        std::cout<<"未查找到该成员！\n\n";
        ClearBuffer();
        return;
    }
    std::cout<<"请输入"<<parent<<"新添加的儿子（或女儿）的姓名：\n";
    mine::String child;
    while(true) {
        std::cin>>child;
        if(genealogy.Search(child,root)) {
            std::cout<<"族谱中的名字不能重复！请重新输入。\n";
            continue;
        }
        break;
    }
    BinTreeNode<mine::String>* temp=current;
    if(genealogy.LeftChild(temp)==nullptr)
        genealogy.leftInsert(temp,child);
    else {
        temp=genealogy.LeftChild(temp);
        while(genealogy.RightChild(temp)!=nullptr)
            temp=genealogy.RightChild(temp);
        genealogy.rightInsert(temp,child);
    }
    PrintChild(genealogy,parent,current);
}
```

3.3.4 示例



3.4 解散局部家庭功能

3.4.1 实现思路

解散局部家庭功能的函数名为 MyDissolve，解散局部家庭功能实现的思路为：

1.首先输入需要解散局部家庭的成员，根据输入姓名调用 genealogy.Search 函数查找对应的结点指针；

2.若查找成功且该成员左孩子不为空，则进行下一步操作；否则提示错误信息，退出函数；

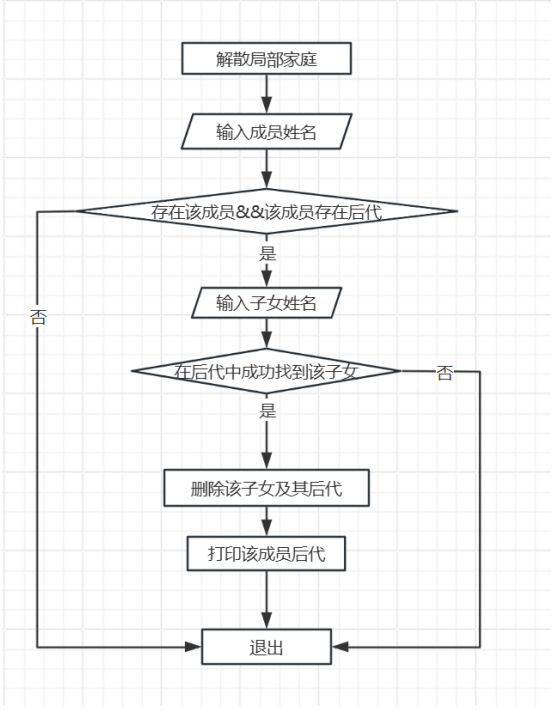
3.输入需要该成员需要被解散的孩子姓名；

4.在该成员左孩子以及左孩子一右到底的所有结点中查找该子女，若查找成功，则调用 genealogy.Destory 删除该子女及其后代，否则输出错误信息并返回；

5.删除完成后，打印该成员第一代子女的信息；

6.退出 MyDissolve，该功能实现完毕。

3.4.2 流程图

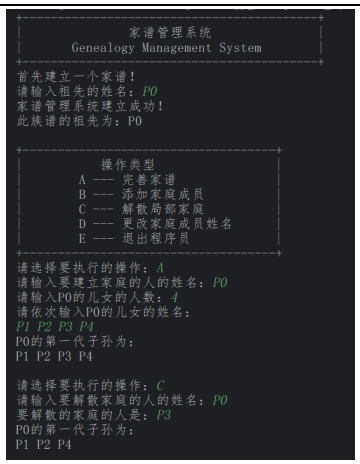


3.4.3 核心代码

```
void MyDissolve(BinaryTree<mine::String>& genealogy) {
    std::cout << "请输入要解散家庭的人的姓名：";
    mine::String parent;
    std::cin >> parent;
    BinTreeNode<mine::String>* root = genealogy.GetRoot();
    BinTreeNode<mine::String>* current = genealogy.Search(parent, root);
    if (current == nullptr) {
        std::cout << "未查找到该成员！\n\n";
        ClearBuffer();
        return;
    }
    if (genealogy.LeftChild(current) == nullptr) {
        std::cout << "该成员不存在后代！\n\n";
        return;
    }
    std::cout << "要解散的家庭的人是：";
    mine::String son;
    std::cin >> son;
    BinTreeNode<mine::String>* temp = current;
```

```
        bool found = false;
        if (genealogy.Data(genealogy.LeftChild(temp)) == son) {
            BinTreeNode<mine::String>* to_del = genealogy.LeftChild(temp);
            genealogy.ChangeLeftChild(temp, genealogy.RightChild(to_del));
            genealogy.ChangeRightChild(to_del, nullptr);
            genealogy.Destory(to_del);
            found = true;
        }
        else {
            temp = genealogy.LeftChild(temp);
            while (genealogy.RightChild(temp) != nullptr) {
                if (genealogy.Data(genealogy.RightChild(temp)) == son) {
                    BinTreeNode<mine::String>* to_del =
genealogy.RightChild(temp);
                    genealogy.ChangeRightChild(temp,
genealogy.RightChild(to_del));
                    genealogy.ChangeRightChild(to_del, nullptr);
                    genealogy.Destory(to_del);
                    found = true;
                    break;
                }
                temp = genealogy.RightChild(temp);
            }
        }
        if (!found) {
            std::cout<<"未找到需要解散家庭的人!";
            return;
        }
        PrintChild(genealogy,parent,current);// 打印剩余子女
    }
```

3.4.4 示例



3.5 更改家庭成员姓名功能

3.5.1 实现思路

添加家庭成员功能的函数名为 MyModify，添加家庭成员功能实现的思路为：

1.首先输入需要添加家庭成员的家谱成员的姓名，根据输入姓名调用 genealogy.Search 函数查找对应的结点指针；

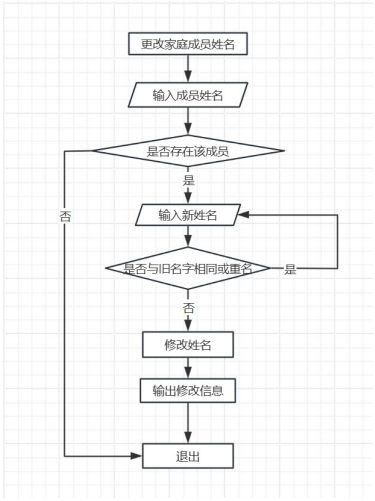
2.若查找成功则进行下一步操作；否则提示错误信息，退出函数；

3.进入一个 while 循环，输入该成员新的姓名，输入时，需要保证新名字不能与其他家庭成员重名，也不能与旧名字相同，否则需要重新输入；

4.修改该成员姓名，并输出修改信息；

5.退出 MyModify，该功能实现完毕。

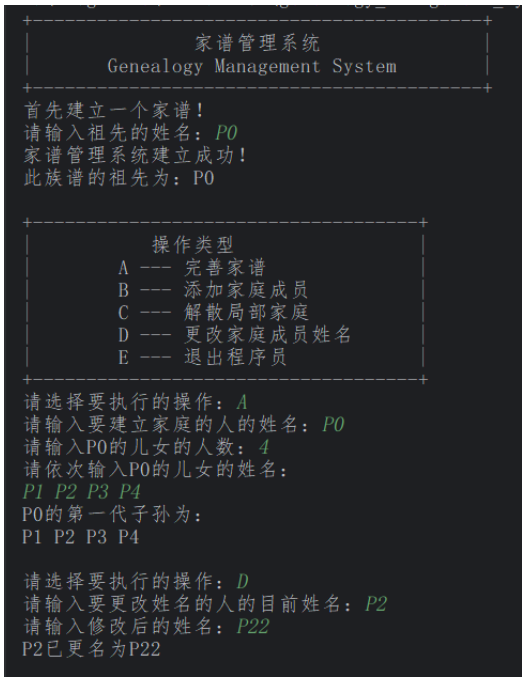
3.5.2 流程图



3.5.3 核心代码

```
void MyModify(BinaryTree< mine::String>& genealogy) {
    std::cout<<"请输入要更改姓名的人的目前姓名：";
    mine::String name;
    std::cin>>name;
    BinTreeNode< mine::String>* root=genealogy.GetRoot();
    BinTreeNode< mine::String>* current=genealogy.Search(name,root);
    if(current==nullptr) {
        std::cout<<"未查找到该成员！\n\n";
        ClearBuffer();
        return;
    }
    std::cout<<"请输入修改后的姓名：";
    mine::String new_name;
    while (true) {
        std::cin >> new_name;
        if(new_name==name) {
            std::cout<<"新名字不能与旧名字相同，请重新输入！\n";
            continue;
        }
        if (genealogy.Search(new_name, root)) {
            std::cout << "族谱中的名字不能重复，请重新输入！\n";
            continue;
        }
        break;
    }
    current->data=new_name;
    std::cout<<name<<"已更名为"<<new_name<<"\n\n";
}
```

3.5.4 示例



3.6 异常处理功能

3.6.1 输入非法的异常处理

3.6.1.1 子女人数输入非法的异常处理

在完善家谱功能中，需要输入成员子女人数，通过一个 while 无限循环来确保输入的子女人数为正整数；当输入错误时，会清除缓冲区，继续执行循环函数，当输入正确时，则会通过 break 退出循环，继续下一步输入操作。具体代码如下：

```
double n;
while(true) {
    std::cout<<"请输入"<<parent<<"的儿女的人数：";
    std::cin>>n;
    if(std::cin.fail()||n!=static_cast<int>(n)) {
        std::cout<<"输入非法，请重新输入！\n";
        ClearBuffer();
        continue;;
    }
    break;
```

```
}
int number = static_cast<int>(n);
```

3.6.1.2 操作类型输入非法的异常处理

操作类型输入非法的异常处理通过如下代码实现

```
char MySelect() {
    char ch;
    while(true) {
        std::cout<<"请选择要执行的操作：";
        std::cin>>ch;
        if(ch<'A'||ch>'E') {
            std::cout<<"输入非法，请重新输入！\n";
            ClearBuffer();
            continue;;
        }
        ClearBuffer();
        break;
    }
    return ch;
}
```

这段代码的具体执行逻辑如下：

1. 显示输入提示信息；

2. 进入一个无限循环，以等待用户输入；

3. 用户以字符的方式输入选项，检查用户输入是否规范且在一定范围，规范则返回 char 类型选项；若不规范或不符合范围则清除当前输入状态和缓冲区，重新输入。

3.6.2 动态内存申请失败的异常处理

在进行 BinaryTree 类与 String 类的动态内存申请时，程序使用 new(std::nothrow) 来尝试分配内存。new(std::nothrow)在分配内存失败时不会引发异常，而是返回一个空指针（NULL 或 nullptr），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，对于内存分配失败，可以采用断言的方式来处理错误，例如：

```
template <typename T>
BinaryTree<T>::BinaryTree(T& item)
{
    root = new(std::nothrow) BinTreeNode<T>(item);
    assert(root != nullptr);
}
```

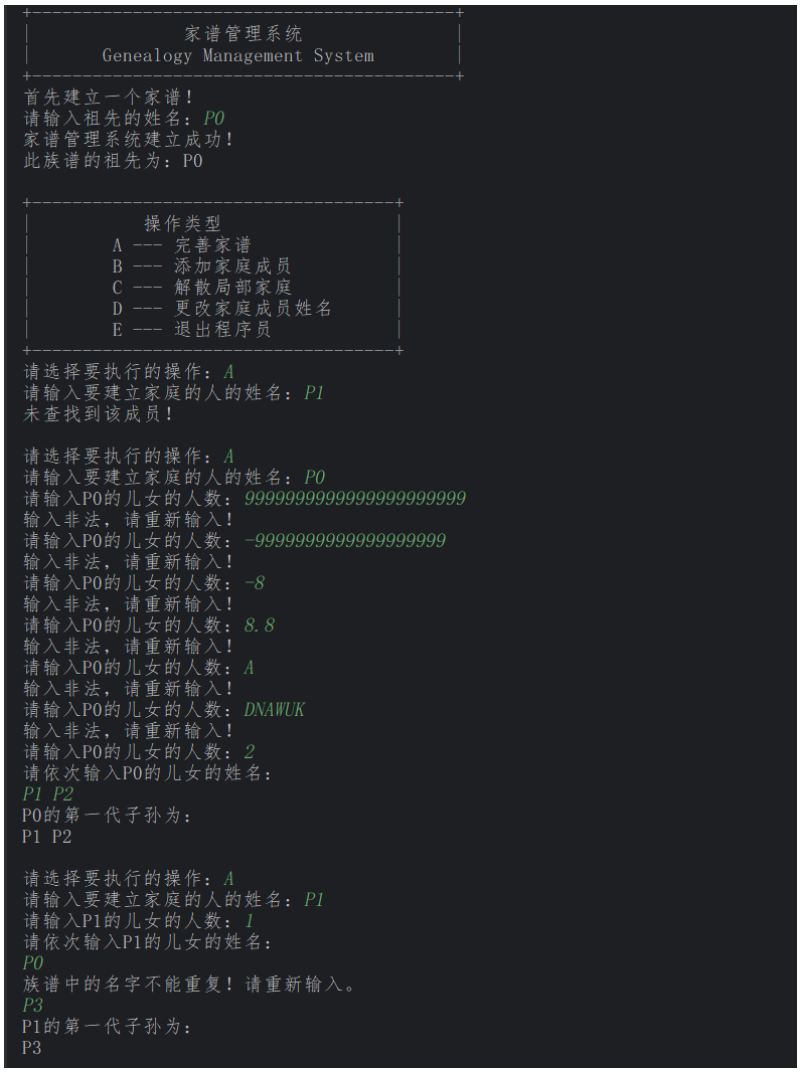
第 4 章 项目测试

4.1 完善家谱功能测试

输入在家谱中不存在的成员或该成员已经拥有家庭时，程序结束操作，验证程序对输入家谱中不存在的家庭成员或该成员已经拥有家庭的情况进行了处理。

输入某家庭成员的子女人数时，分别输入超过上下限的整数、浮点数、字符、字符串，程序要求重新输入，验证程序对输入非法的情况进行了处理。

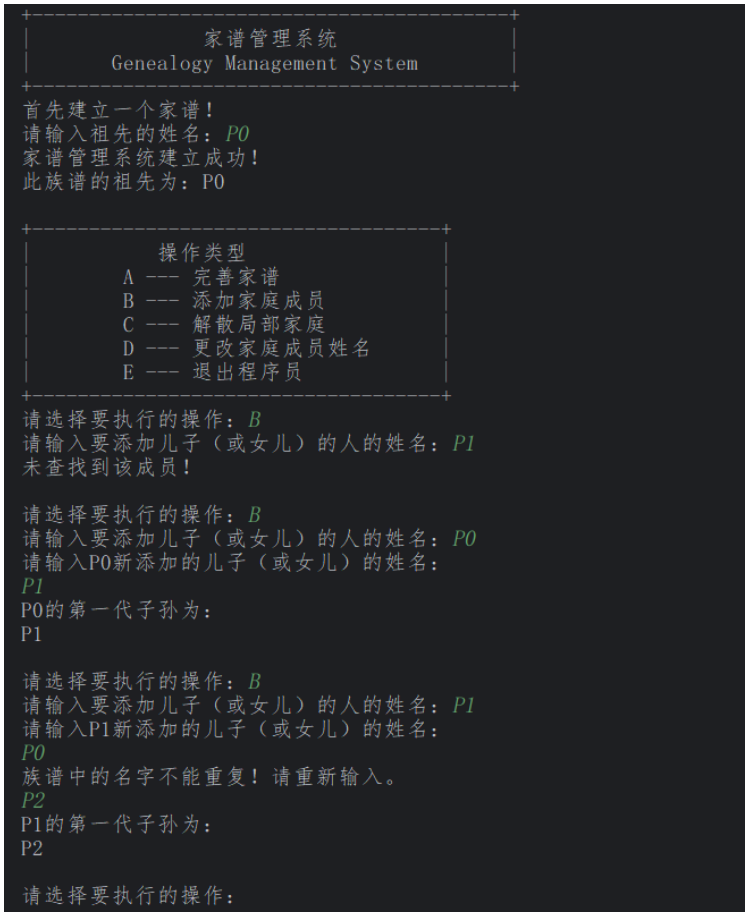
输入某家庭成员的子女姓名时，输入姓名与家谱中已存在的成员重复时，程序要求重新输入，可以验证程序对重名的情况进行了处理。



4.2 添加家庭成员功能测试

输入在家谱中不存在的成员，程序结束操作，验证程序对输入家谱中不存在的成员的情况进行了处理。

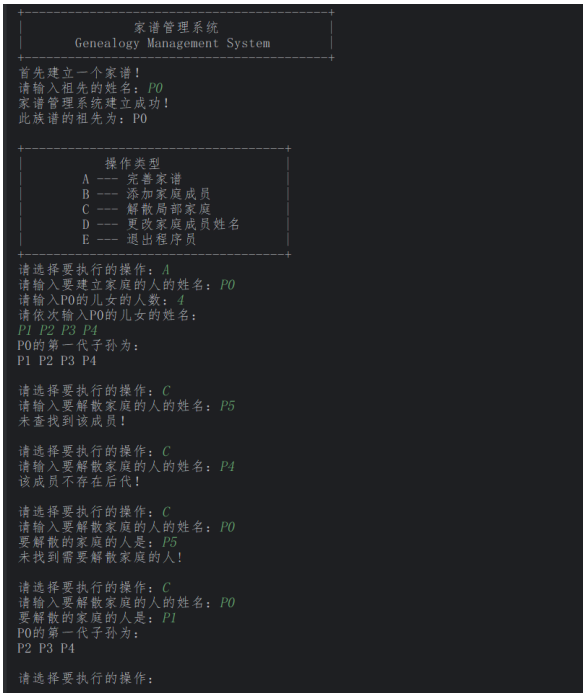
输入某家庭成员的儿女姓名时，输入姓名与家谱中已存在的成员重复时，程序要求重新输入，可以验证程序对重名的情况进行了处理。



4.3 解散局部家庭功能测试

输入在家谱中不存在的成员或成员没有子女（即结点没有左孩子），程序结束操作，验证程序对输入家谱中不存在的成员或成员没有子女的情况进行了处理。

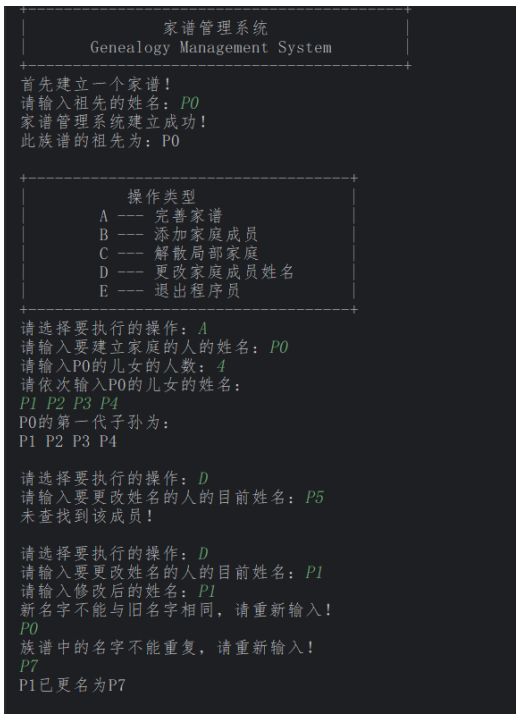
输入的子女不在该成员的家庭中时程序结束操作，验证程序对输入不存在成员的家庭中的子女的情况进行了处理。



4.4 更改家庭成员姓名功能测试

输入在家谱中不存在的成员，程序结束操作，验证程序对输入家谱中不存在的成员的情况进行了处理。

输入某家庭成员的更改后姓名时，若新名字与旧名字相同或新名字与家谱中已存在的成员相同，程序要求重新输入。



第 5 章 相关说明

5.1 编程语言

本项目全部.cpp 文件以及.h 文件均使用 C++编译完成，使用 UTF-8 编码。

5.2 Windows 环境

Windows 系统：Windows 11 x64

Windows 集成开发环境：CLion 2024

工具集：MinGW 11.0 w64

5.3 Linux 环境

基于 Linux 内核的操作系统发行版：Ubuntu 24.04.1

Linux 命令编译过程为：

1. 定位包含项目所在文件夹，包括.pp 与.h 文件；具体命令为: cd /home/bruce/programe/ genealogy_management_systerm

2.编译项目，生成可执行文件；具体命令为: g++ -static -o genealogy_management_systerm_linux genealogy_management_systerm.cpp my_bianry_tree.h my_string.h;

其中指令含义分别为：

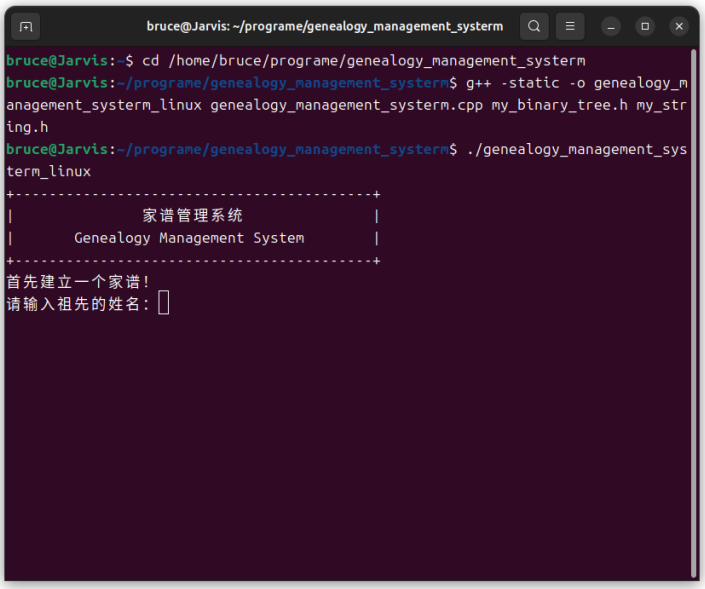
g++: 调用 GNU 的 C++编译器

-static: 使用静态链接而非动态链接，将所有依赖库直接嵌入到可执行文件，文件存储空间变大，但可以单独运行

-o genealogy_management_systerm_linux: -o 表示输出文件选项，genealogy_management_systerm_linux 为可执行文件名

genealogy_management_systerm.cpp my_binary_tree.h my_string.h:编译所需要的文件。

3.运行可执行文件；具体命令为 ./ genealogy_management_systerm_linux


bruce@Jarvis: ~/programe/genealogy_management_systerm
bruce@Jarvis:~$ cd /home/bruce/programe/genealogy_management_systerm
bruce@Jarvis:~/programe/genealogy_management_systerm$ g++ -static -o genealogy_management_systerm_linux genealogy_management_systerm.cpp my_binary_tree.h my_string.h
bruce@Jarvis:~/programe/genealogy_management_systerm$ ./genealogy_management_systerm_linux
+--+
| 家谱管理系统 |
| Genealogy Management System |
+--+
首先建立一个家谱!
请输入祖先的姓名: