

项目说明文档

数据结构课程设计

——家谱管理系统

作者姓名 林继申
学 号 2250758
指导教师 张 颖
学院专业 软件学院 软件工程



同濟大學
TONGJI UNIVERSITY

二〇二三年十二月十三日

目录

1 项目分析.....	1
1.1 项目背景分析.....	1
1.2 项目需求分析.....	1
1.3 项目功能分析.....	1
1.3.1 完善家谱功能.....	2
1.3.2 添加家庭成员功能.....	2
1.3.3 解散家庭成员功能.....	2
1.3.4 更改家庭成员姓名功能.....	2
1.3.5 统计家庭成员功能.....	2
1.3.6 异常处理功能.....	2
2 项目设计.....	2
2.1 数据结构设计.....	2
2.2 结构体与类设计.....	3
2.2.1 MyLinkNode 结构体的设计.....	3
2.2.1.1 概述.....	3
2.2.1.2 结构体定义.....	4
2.2.1.3 数据成员.....	4
2.2.1.4 构造函数.....	4
2.2.2 MyQueue 类的设计.....	4
2.2.2.1 概述.....	4
2.2.2.2 类定义.....	4
2.2.2.3 私有数据成员.....	5
2.2.2.4 构造函数.....	5
2.2.2.5 析构函数.....	5
2.2.2.6 公有成员函数.....	5
2.2.3 MyBinTreeNode 结构体的设计.....	6
2.2.3.1 概述.....	6
2.2.3.2 结构体定义.....	6
2.2.3.3 数据成员.....	6
2.2.3.4 构造函数.....	6
2.2.4 MyBinaryTree 类的设计.....	7
2.2.4.1 概述.....	7
2.2.4.2 类定义.....	7

2.2.4.3 私有数据成员	8
2.2.4.4 VisitFunction 类型定义	9
2.2.4.5 构造函数	9
2.2.4.6 析构函数	9
2.2.4.7 私有成员函数	9
2.2.4.8 公有成员函数	10
2.2.4.9 运算符重载	11
2.2.5 Person 结构体的设计	12
2.2.5.1 概述	12
2.2.5.2 结构体定义	12
2.2.5.3 数据成员	12
2.2.5.4 构造函数	12
2.2.5.5 运算符重载	12
2.3 项目主体架构设计	13
3 项目功能实现.....	14
3.1 项目主体架构的实现	14
3.1.1 项目主体架构实现思路	14
3.1.2 项目主体架构核心代码	15
3.1.3 项目主体架构示例	16
3.2 完善家谱功能的实现	16
3.2.1 完善家谱功能实现思路	16
3.2.2 完善家谱功能核心代码	17
3.2.3 完善家谱功能示例	19
3.3 添加家庭成员功能的实现	19
3.3.1 添加家庭成员功能实现思路	19
3.3.2 添加家庭成员功能核心代码	20
3.3.3 添加家庭成员功能示例	21
3.4 解散家庭成员功能的实现	21
3.4.1 解散家庭成员功能实现思路	21
3.4.2 解散家庭成员功能核心代码	22
3.4.3 解散家庭成员功能示例	22
3.5 更改家庭成员姓名功能的实现	22
3.5.1 更改家庭成员姓名功能实现思路	22
3.5.2 更改家庭成员姓名功能核心代码	23
3.5.3 更改家庭成员姓名功能示例	24

3.6 统计家庭成员功能的实现	24
3.6.1 统计家庭成员功能实现思路	24
3.6.2 统计家庭成员功能核心代码	25
3.6.3 统计家庭成员功能示例	25
3.7 异常处理功能的实现	26
3.7.1 动态内存申请失败的异常处理	26
3.7.2 MyQueue 类队列空的异常处理	26
3.7.3 MyBinaryTree 类自赋值的异常处理	26
3.7.4 Person 结构体自赋值的异常处理	26
3.7.5 输入非法的异常处理	27
3.7.5.1 某家庭成员的儿女人数输入非法的异常处理	27
3.7.5.2 某家庭成员姓名输入非法的异常处理	28
3.7.5.3 操作类型输入非法的异常处理	28
4 项目测试	29
4.1 家庭成员姓名输入功能测试	29
4.2 完善家谱功能测试	29
4.3 添加家庭成员功能测试	30
4.4 解散家庭成员功能测试	31
4.5 更改家庭成员姓名功能测试	32
4.6 统计家庭成员功能测试	32
4.7 退出家谱管理系统功能测试	33
5 集成开发环境与编译运行环境	33

1 项目分析

1.1 项目背景分析

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，是珍贵的人文资料，对于历史学、民俗学、人口学、社会学和经济学的深入研究，有着不可替代的独特作用。

本项目是一个家谱管理系统，用于创建和管理一个家族的家谱。家谱管理系统对于记录和维护家族的历史、成员以及家族之间的关系非常重要。它可以帮助用户理解家族历史，追溯血缘关系，并保存重要的家族信息。这样的系统在文化传承、家族研究以及保存家族历史方面有着重要作用。

1.2 项目需求分析

基于以上背景分析，本项目需要实现需求如下：

(1)实现对家族成员信息的完善、添加、解散、更改、统计等功能，确保数据的准确、高效管理；

(2)设计简单直观的控制台界面，使操作便捷、容易上手，适应不同用户的操作习惯；

(3)选择合适的数据结构，以支持对家族成员信息的高效操作，同时考虑信息的关联性和复杂度；

(4)实现异常处理机制，确保系统稳定性和安全性，避免因用户输入错误导致系统崩溃或信息丢失；

(5)设计系统以支持未来的扩展和功能增加，满足不同用户、不同应用场景下的需求。

1.3 项目功能分析

本项目旨在通过模拟家谱管理过程，实现完善家谱、添加家庭成员、解散家庭成员、更改家庭成员姓名、统计家庭成员等功能，从而实现对家族成员的高效管理。需要设计合适的数据结构、开发用户友好的控制台界面，并考虑系统的稳定性、安全性以及未来的扩展性。通过该项目的实施，可以提高家谱管理系统的效率和准确性。下面对项目的功能进行详细分析。

1.3.1 完善家谱功能

允许用户为特定成员添加直系后代。这是建立家谱时的关键功能，它允许用户逐步构建完整的家族树。

1.3.2 添加家庭成员功能

用户可以添加新的家庭成员到现有的家谱中。这对于记录新生儿或通过婚姻等方式加入家族的成员非常重要。

1.3.3 解散家庭成员功能

提供删除特定家庭成员的后代的选项，这对于维护准确和最新的家谱数据很重要。

1.3.4 更改家庭成员姓名功能

允许修改家庭成员的姓名。这一功能在家庭成员改名或者录入错误时非常有用。

1.3.5 统计家庭成员功能

提供统计和查看特定家庭成员及其后代的功能，便于用户了解家族规模和成员构成。

1.3.6 异常处理功能

实现异常处理机制，处理用户可能输入的非法信息，确保系统的稳定性和安全性。

2 项目设计

2.1 数据结构设计

基于项目分析，家谱管理系统的设计中选择使用二叉树结构作为数据结构而不是一般树结构，主要基于以下几个考虑：

(1) 简化结构和操作：在家谱管理中，每个家庭成员通常有多个孩子，但使用二叉树可以将这种复杂性简化。第一个孩子作为左孩子，其余孩子作为右孩子的兄弟链表可以简化添加、删除等操作。

(2) 时间和空间效率：二叉树的遍历、搜索、插入和删除操作相对一般树来说更加简单明了。二叉树相较于一般树在存储上更加高效。每个节点只需存储两个子节点的链接，减少了存储空间的浪费。对于家谱管理系统，大部分家庭成员

的后代数量并不会非常庞大，因此使用二叉树不会造成太多的空间浪费。

(3) 易于扩展和修改：二叉树的结构使得对家谱的扩展和修改变得容易。例如，添加新成员或更改现有成员关系时，只需调整有限的节点链接。对于家族成员的插入和删除操作，二叉树提供了较高的灵活性和效率。

(4) 便于家族层级展示：使用二叉树可以方便地展示家族成员的层级关系。通过左孩子和右孩子的关系，可以清晰地表达家族成员之间的直系和旁系血缘关系。家谱的图形化展示在使用二叉树时更加直观和易于理解。

(5) 算法和数据结构的成熟：二叉树是计算机科学中一个非常成熟的数据结构，有大量的研究和实现算法可以借鉴。对于开发者来说，基于二叉树的操作和维护相对简单，有利于提高系统的开发效率和稳定性。

基于对二叉树结构的分析，在实现二叉树数据结构时，使用链表实现节点而不是传统数组，主要基于以下几个考虑：

(1) 动态大小需求：链表可以动态地分配内存，适应不同数量的家族成员信息，而数组需要预先确定大小，可能会导致内存浪费或不足；

(2) 插入和删除操作效率高：链表对于插入和删除操作效率较高，因为只需要调整节点的指针即可，而数组需要移动元素，时间复杂度效率较高；

(3) 频繁的数据修改：如果家族成员信息需要频繁修改，例如更改家庭成员姓名、解散家庭成员等功能，链表更适合，因为修改节点的指针比修改数组元素更高效；

(4) 不需要随机访问：如果系统不需要通过索引随机访问家族成员信息，而只是按一定顺序处理，链表可以满足需求；

(5) 内存分配灵活性：链表的内存分配比较灵活，可以根据需求动态分配，而数组需要一次性分配固定大小的内存空间；

(6) 避免数组扩展的开销：使用数组可能需要额外的扩展和拷贝操作，而链表避免了这种开销；

(7) 考虑系统维护的复杂度：考虑系统的维护和扩展性，链表可以更方便地进行扩展和修改，而数组可能需要重新设计和实现。

基于上述分析，在设计家谱管理系统时，选择链表实现的二叉树结构作为数据结构更合适。

2.2 结构体与类设计

2.2.1 MyLinkNode 结构体的设计

2.2.1.1 概述

MyLinkNode 结构体是一个用于构建链表节点的模板结构体。该结构体用于

表示链表中的每个节点，其中包括节点存储的数据以及指向下一个节点的指针。经典的链表一般包括两个抽象数据类型（ADT）——链表结点类（`LNode`）与链表类（`LinkList`）。本项目希望链表结点类可以直接访问链表结点，所以使用 `struct` 而不是 `class` 描述链表结点类。

2.2.1.2 结构体定义

```
template <typename Type>
struct MyLinkNode {
    Type data;
    MyLinkNode<Type>* link;
    MyLinkNode(MyLinkNode<Type>* ptr = NULL) { link = ptr; }
    MyLinkNode(const Type& item, MyLinkNode<Type>* ptr = NULL)
{ data = item; link = ptr; }
};
```

2.2.1.3 数据成员

`Type data`: 数据域，存储节点的数据

`MyLinkNode<Type>* link`: 指针域，指向下一个节点的指针

2.2.1.4 构造函数

`MyLinkNode(MyLinkNode<Type>* ptr = NULL);`

构造函数，初始化指针域。

`MyLinkNode(const Type& item, MyLinkNode<Type>* ptr = NULL);`

构造函数，初始化数据域和指针域。

2.2.2 MyQueue 类的设计

2.2.2.1 概述

该通用模板类 `MyQueue` 用于存储和管理数据元素，遵循先进先出（**First-in-First-Out, FIFO**）的原则。该队列是基于链表数据结构实现的，链表节点由 `MyLinkNode` 结构体表示，其中包含数据和指针域。此模板类 `MyQueue` 允许在队列的前端（`front`）添加元素，以及从队列的前端（`front`）移除元素。

2.2.2.2 类定义

```
template <typename Type>
class MyQueue {
private:
```



```

    MyLinkNode<Type>* front;
    MyLinkNode<Type>* rear;
public:
    MyQueue() : front(NULL), rear(NULL) {}
    ~MyQueue() { makeEmpty(); }
    bool isEmpty(void) const { return front == NULL; }
    void makeEmpty(void);
    void enqueue(const Type& item);
    bool dequeue(Type& item);
    bool getFront(Type& item) const;
    int getSize(void) const;
};

```

2.2.2.3 私有数据成员

`MyLinkNode<Type>* front`: 指针指向队列的前端，即队列中的第一个元素

`MyLinkNode<Type>* rear`: 指针指向队列的后端，即队列中的最后一个元素

2.2.2.4 构造函数

```
MyQueue() : front(NULL), rear(NULL) {}
```

默认构造函数，创建一个空的队列。

2.2.2.5 析构函数

```
~MyQueue() { makeEmpty(); }
```

析构函数，清空队列并释放内存。

2.2.2.6 公有成员函数

```
bool isEmpty(void) const { return front == NULL; }
```

检查队列是否为空。

```
void makeEmpty(void);
```

清空队列，释放所有元素占用的内存。

```
void enqueue(const Type& item);
```

将元素添加到队列的末尾。

```
bool dequeue(Type& item);
```

移除队列的前端元素并将其值通过引用返回。

```
bool getFront(Type& item) const;
```

获取队列的前端元素的值。

```
int getSize(void) const;
```

获取队列中元素的数量。

2.2.3 MyBinTreeNode 结构体的设计

2.2.3.1 概述

MyBinTreeNode 结构体是一个模板结构体，用于构建和表示一个二叉树的节点。二叉树是一种重要的数据结构，广泛应用于各种计算机科学领域。在这个结构体中，每个节点包含一个数据元素和两个指向其子节点的指针：左子节点和右子节点。

2.2.3.2 结构体定义

```
template <typename Type>
struct MyBinTreeNode {
    Type data;
    MyBinTreeNode<Type>* leftChild;
    MyBinTreeNode<Type>* rightChild;
    MyBinTreeNode() : leftChild(NULL), rightChild(NULL) {}
    MyBinTreeNode(Type item, MyBinTreeNode<Type>* l = NULL,
MyBinTreeNode<Type>* r = NULL) : data(item), leftChild(l),
rightChild(r) {}
};
```

2.2.3.3 数据成员

Type data: 数据域，存储节点的数据

MyBinTreeNode<Type>* leftChild: 指针域，指向左子节点的指针

MyBinTreeNode<Type>* rightChild: 指针域，指向右子节点的指针

2.2.3.4 构造函数

```
MyBinTreeNode() : leftChild(NULL), rightChild(NULL) {}
```

构造函数，初始化指针域。

```
MyBinTreeNode(Type item, MyBinTreeNode<Type>* l = NULL,
MyBinTreeNode<Type>* r = NULL) : data(item), leftChild(l),
rightChild(r) {}
```

构造函数，初始化数据域和指针域。

2.2.4 MyBinaryTree 类的设计

2.2.4.1 概述

MyBinaryTree 类是一个模板类，用于创建和管理二叉树结构。二叉树是一种基础且重要的数据结构，广泛应用于多种场景，如管理、排序和搜索算法等。本模板类提供了二叉树的基本操作，包括插入、查找、遍历、修改和删除节点等功能。

2.2.4.2 类定义

```
template <typename Type>
class MyBinaryTree {
private:
    MyBinTreeNode<Type>* root;
    typedef void (MyBinaryTree::*
VisitFunction)(MyBinTreeNode<Type>*);
    MyBinTreeNode<Type>* copy(const MyBinTreeNode<Type>*
subTree);
    void outputNode(MyBinTreeNode<Type>* node) { if (node !=
NULL) std::cout << node->data << " "; }
public:
    MyBinaryTree() : root(NULL) {}
    MyBinaryTree(Type& item);
    MyBinaryTree(MyBinaryTree<Type>& other) { root =
copy(other.root); }
    ~MyBinaryTree() { destroy(root); }
    void destroy(MyBinTreeNode<Type>*& subTree);
    bool isEmpty(void) { return root == NULL; }
    int getHeight(MyBinTreeNode<Type>* subTree) { return
(subTree == NULL) ? 0 : (std::max(getHeight(subTree->leftChild),
getHeight(subTree->rightChild)) + 1); }
    int getSize(MyBinTreeNode<Type>* subTree) { return
(subTree == NULL) ? 0 : (getSize(subTree->leftChild) +
getSize(subTree->rightChild) + 1); }
    MyBinTreeNode<Type>* getRoot(void) { return root; }
    MyBinTreeNode<Type>* getParent(MyBinTreeNode<Type>*
current, MyBinTreeNode<Type>* subTree);
```

```

        MyBinTreeNode<Type>* getLeftChild(MyBinTreeNode<Type>*
current) { return current == NULL ? NULL : current->leftChild; }
        MyBinTreeNode<Type>* getRightChild(MyBinTreeNode<Type>*
current) { return current == NULL ? NULL : current->rightChild; }
        MyBinTreeNode<Type>* findNode(const Type& item,
MyBinTreeNode<Type>* subTree);
        void preOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
        void inOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
        void postOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
        void levelOrder(VisitFunction visit,
MyBinTreeNode<Type>* subTree);
        void preOrderOutput(MyBinTreeNode<Type>* subTree)
{ preOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        void inOrderOutput(MyBinTreeNode<Type>* subTree)
{ inOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        void postOrderOutput(MyBinTreeNode<Type>* subTree)
{ postOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        void levelOrderOutput(MyBinTreeNode<Type>* subTree)
{ levelOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        bool leftInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
        bool rightInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
        bool modifyNode(const Type& oldItem, const Type& newItem,
MyBinTreeNode<Type>* subTree);
        MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>&
other);
    };

```

2.2.4.3 私有数据成员

MyBinTreeNode<Type>* root: 指向树的根节点的指针

2.2.4.4 VisitFunction 类型定义

```
typedef void (MyBinaryTree::* VisitFunction)(MyBinTreeNode<Type>*);
```

在 `MyBinaryTree` 类中, `VisitFunction` 是一个特殊的类型定义, 属于类定义的一部分。这是一个函数指针类型, 用于指向 `MyBinaryTree` 类的成员函数, 这些成员函数接受一个指向 `MyBinTreeNode<Type>` 类型的指针作为参数并返回 `void`。

`VisitFunction` 类型主要用于树的遍历操作中, 允许将成员函数作为参数传递给遍历函数, 从而实现对树中每个节点执行特定操作的功能。通过这种方式, 可以在遍历树时灵活地应用不同的处理逻辑, 例如打印节点数据、计算节点总数、修改节点数据等。

2.2.4.5 构造函数

```
MyBinaryTree() : root(NULL) {}
```

默认构造函数, 初始化一个空的二叉树, 即根节点设置为 `NULL`。

```
MyBinaryTree(Type& item);
```

转换构造函数, 创建一个二叉树, 并设置根节点的数据为提供的 `item`, 适用于已知根节点数据的情况, 便于直接构建含有根节点的二叉树。

```
MyBinaryTree(MyBinaryTree<Type>& other) { root = copy(other.root); }
```

复制构造函数, 创建一个新的二叉树, 其结构和数据是另一个二叉树 (`other`) 的深拷贝。

2.2.4.6 析构函数

```
~MyBinaryTree() { destroy(root); }
```

析构函数, 用于在 `MyBinaryTree` 类的对象不再需要时, 安全地销毁该对象。它负责释放二叉树中所有节点占用的内存资源, 防止内存泄漏。

2.2.4.7 私有成员函数

```
MyBinTreeNode<Type>* copy(const MyBinTreeNode<Type>* subTree);
```

深拷贝一个子树。

```
void outputNode(MyBinTreeNode<Type>* node) { if (node != NULL) std::cout << node->data << " "; }
```

输出一个节点的数据。

2.2.4.8 公有成员函数

```
void destroy(MyBinTreeNode<Type>*& subTree);
```

递归地销毁整个子树。

```
bool isEmpty(void) { return root == NULL; }
```

检查树是否为空。

```
int getHeight(MyBinTreeNode<Type>* subTree) { return  
(subTree == NULL) ? 0 : (std::max(getHeight(subTree->leftChild),  
getHeight(subTree->rightChild)) + 1); }
```

计算树或子树的高度。

```
int getSize(MyBinTreeNode<Type>* subTree) { return (subTree  
== NULL) ? 0 : (getSize(subTree->leftChild) +  
getSize(subTree->rightChild) + 1); }
```

计算树或子树的节点总数。

```
MyBinTreeNode<Type>* getRoot(void) { return root; }
```

获取树的根节点。

```
MyBinTreeNode<Type>* getParent(MyBinTreeNode<Type>* current,  
MyBinTreeNode<Type>* subTree);
```

查找指定节点的父节点。

```
MyBinTreeNode<Type>* getLeftChild(MyBinTreeNode<Type>*  
current) { return current == NULL ? NULL : current->leftChild; }
```

获取指定节点的左子节点。

```
MyBinTreeNode<Type>* getRightChild(MyBinTreeNode<Type>*  
current) { return current == NULL ? NULL : current->rightChild; }
```

获取指定节点的右子节点。

```
MyBinTreeNode<Type>* findNode(const Type& item,  
MyBinTreeNode<Type>* subTree);
```

在树中查找包含特定数据的节点。

```
void preOrder(VisitFunction visit, MyBinTreeNode<Type>*  
subTree);
```

前序遍历 (Pre-Order Traversal) 树。先访问根节点，然后递归地进行左子树的前序遍历，最后递归地进行右子树的前序遍历。

```
void inOrder(VisitFunction visit, MyBinTreeNode<Type>*  
subTree);
```

中序遍历 (In-Order Traversal) 树。首先递归地进行左子树的中序遍历，然后访问根节点，最后递归地进行右子树的中序遍历。

```
void postOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
```

后序遍历 (Post-Order Traversal) 树。首先递归地进行左子树的后序遍历, 然后递归地进行右子树的后序遍历, 最后访问根节点。

```
void levelOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
```

层序遍历 (Level-Order Traversal) 树。按照树的层次从上到下, 从左到右逐层访问每个节点。通常使用队列来实现。将根节点入队, 然后在循环中不断出队一个节点并将其子节点入队, 直到队列为空。

```
void preOrderOutput(MyBinTreeNode<Type>* subTree)
{ preOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

前序遍历树输出。

```
void inOrderOutput(MyBinTreeNode<Type>* subTree)
{ inOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

中序遍历树输出。

```
void postOrderOutput(MyBinTreeNode<Type>* subTree)
{ postOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

后续遍历树输出。

```
void levelOrderOutput(MyBinTreeNode<Type>* subTree)
{ levelOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

层数遍历树输出。

```
bool leftInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
```

在指定节点插入左子节点。

```
bool rightInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
```

在指定节点插入右子节点。

```
bool modifyNode(const Type& oldItem, const Type& newItem,
MyBinTreeNode<Type>* subTree);
```

修改树中特定数据的节点。

2.2.4.9 运算符重载

```
MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>&
other);
```

重载赋值运算符, 用于实现树的深拷贝。

2.2.5 Person 结构体的设计

2.2.5.1 概述

Person 结构体用于表示一个人的基本信息，在家谱管理系统中作为一个关键组件使用。它主要包含人的姓名，并提供了重载的赋值运算符和比较运算符，以及重载输入（右移）输出（左移）运算符的友元声明，从而实现简单且直观的数据操作。在此基础上可以继续添加性别、年龄、籍贯等信息，可扩展性强。

2.2.5.2 结构体定义

```
struct Person {  
    enum { DescendantsMaxNumber = 16, NameMaxLength = 32 };  
    char name[NameMaxLength + 1];  
    Person() { strcpy(name, ""); }  
    Person& operator=(const Person& other) { if (this !=  
&other) strcpy(name, other.name); return *this; }  
    bool operator==(const Person& other) { return strcmp(name,  
other.name) == 0; }  
    friend std::ostream& operator<<(std::ostream& out, const  
Person& person);  
    friend std::istream& operator>>(std::istream& in,  
Person& person);  
};
```

2.2.5.3 数据成员

enum { DescendantsMaxNumber = 16, NameMaxLength = 32 }:
DescendantsMaxNumber 定义一个人后代的最大数量，NameMaxLength 定义一个人姓名的最大长度

char name[NameMaxLength + 1]: 姓名信息

2.2.5.4 构造函数

```
Person() { strcpy(name, ""); }  
构造函数，初始化 name 为空字符串。
```

2.2.5.5 运算符重载

```
Person& operator=(const Person& other) { if (this != &other)  
strcpy(name, other.name); return *this; }
```

重载赋值运算符，用于实现将一个 Person 对象的姓名复制到另一个 Person

对象。

```
bool operator==(const Person& other) { return strcmp(name, other.name) == 0; }
```

重载比较运算符，用于比较两个 Person 对象的姓名是否相同。

```
friend std::ostream& operator<<(std::ostream& out, const Person& person);
```

重载左移运算符，实现 Person 对象姓名的标准输出。重载函数的具体实现代码如下：

```
std::ostream& operator<<(std::ostream& out, const Person& person)
{
    return (out << person.name);
}
```

```
friend std::istream& operator>>(std::istream& in, Person& person);
```

重载右移运算符，实现 Person 对象姓名的标准输入。重载函数的具体实现代码如下：

```
std::istream& operator>>(std::istream& in, Person& person)
{
    std::cin >> person.name;
    person.name[Person::NameMaxLength] = '\0';
    std::cin.clear();
    std::cin.ignore(INT_MAX, '\n');
    return in;
}
```

2.3 项目主体架构设计

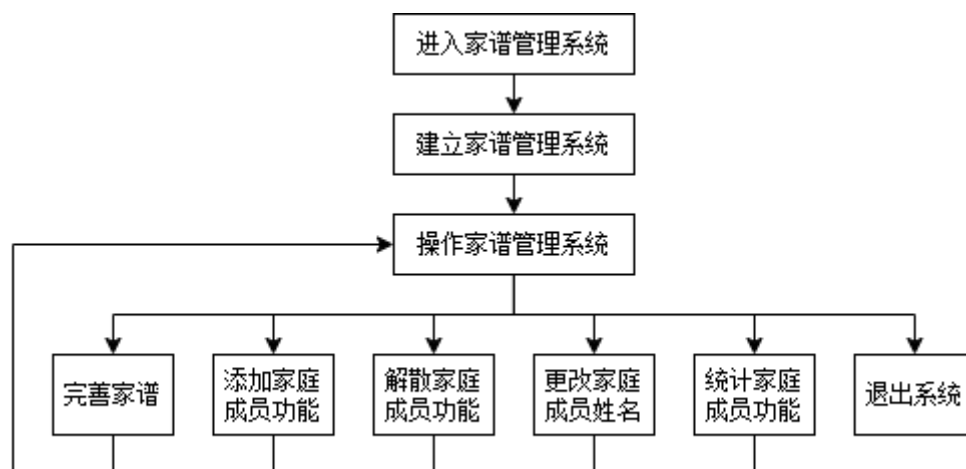


图 2.3.1 项目主体架构设计流程图

项目主体架构设计为：

- (1) 进入家谱管理系统；
- (2) 建立家谱管理系统；
- (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、解散家庭成员、更改家庭成员姓名、统计家庭成员等函数，这些函数实现了对家族成员信息的各种管理和操作；
- (4) 由用户选择退出家谱管理系统。

3 项目功能实现

3.1 项目主体架构的实现

3.1.1 项目主体架构实现思路

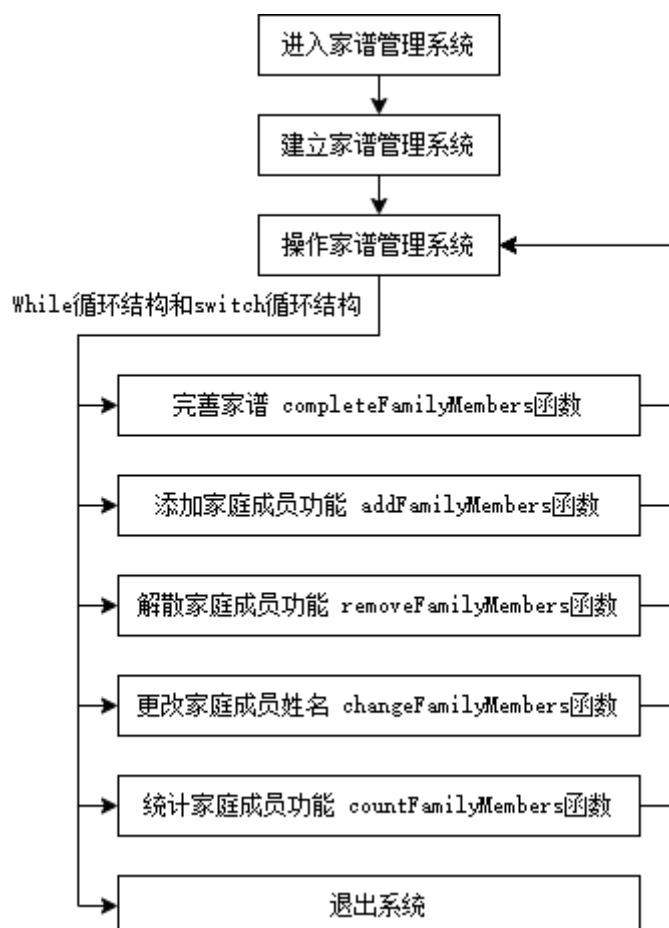


图 3.1.1.1 项目主体架构实现流程图

项目主体架构实现思路为：

- (1) 进入家谱管理系统：系统启动时，首先进入主函数 `main`。这是整个家谱管理系统的入口点；

(2) 建立家谱管理系统: 在 `main` 函数中, 首先进行家谱管理系统的初始化。这包括创建家谱树的实例 `MyBinaryTree<Person>` 和录入祖先信息。这是通过 `MyBinaryTree` 类的构造函数实现的, 它初始化根节点为祖先成员;

(3) 通过循环结构操作家谱管理系统: 系统进入一个循环, 允许用户反复执行不同的操作。这是通过 `while` 循环结构和 `switch` 语句实现的, 循环中首先调用 `selectOptn` 函数, 这个函数通过用户输入来决定执行哪种操作, 根据用户的选择, `switch` 语句调用相应的函数, 如:

- ① `completeFamilyMembers`: 完善家谱功能;
- ② `addFamilyMembers`: 添加家庭成员功能;
- ③ `removeFamilyMembers`: 解散家庭成员功能;
- ④ `changeFamilyMembers`: 更改家庭成员姓名功能;
- ⑤ `countFamilyMembers`: 统计家庭成员功能;

这些函数利用 `MyBinaryTree` 类的各种方法来执行具体的家谱树操作;

(4) 由用户选择退出家谱管理系统: 在循环的每个周期结束时, 用户有机会选择退出系统。当用户选择退出, `main` 函数执行结束, 家谱管理系统关闭。

3.1.2 项目主体架构核心代码

```
int main()
{
    /* System entry prompt */
    std::cout << "+-----+" << std::endl;
    std::cout << "|          家谱管理系统          |" << std::endl;
    std::cout << "| Genealogy Management System |" << std::endl;
    std::cout << "+-----+" << std::endl <<
std::endl;

    /* Establish a genealogy management system */
    std::cout << ">>> 请建立家谱管理系统" << std::endl;
    std::cout << std::endl << ">>> [姓名输入要求] 不超过 " <<
Person::NameMaxLength << " 个英文字符或 " << Person::NameMaxLength / 2 <<
" 个汉字字符组成的字符串, 超出部分将被截断" << std::endl;
    std::cout << std::endl << "请输入祖先姓名: ";
    Person ancestor;
    std::cin >> ancestor;
    MyBinaryTree<Person> genealogy(ancestor);
    std::cout << std::endl << ">>> 家谱建立成功 (祖先: " <<
genealogy.getRoot()->data << ")" << std::endl;

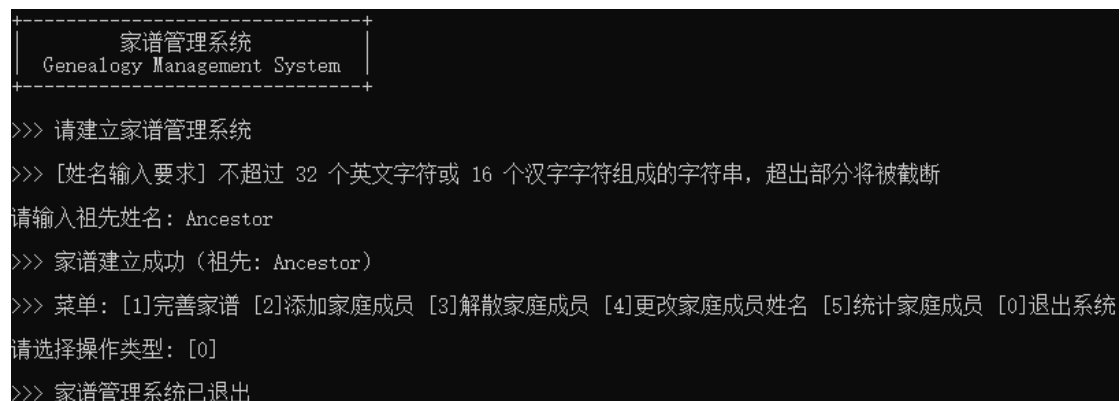
    /* Perform operations on genealogy management system */
    while (true) {
```

```

switch (selectOptn()) {
    case 1:
        completeFamilyMembers(genealogy);
        break;
    case 2:
        addFamilyMembers(genealogy);
        break;
    case 3:
        removeFamilyMembers(genealogy);
        break;
    case 4:
        changeFamilyMembers(genealogy);
        break;
    case 5:
        countFamilyMembers(genealogy);
        break;
    default:
        std::cout << ">>> 家谱管理系统已退出" << std::endl;
        return 0;
}
}
}

```

3.1.3 项目主体架构示例



```

+-----+
| 家谱管理系统 |
| Genealogy Management System |
+-----+

>>> 请建立家谱管理系统
>>> [姓名输入要求] 不超过 32 个英文字符或 16 个汉字字符组成的字符串，超出部分将被截断
请输入祖先姓名: Ancestor
>>> 家谱建立成功 (祖先: Ancestor)
>>> 菜单: [1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型: [0]
>>> 家谱管理系统已退出

```

图 3.1.3.1 项目主体架构示例

3.2 完善家谱功能的实现

3.2.1 完善家谱功能实现思路

完善家谱功能的函数名为 `completeFamilyMembers`，完善家谱功能实现的思路为：

(1) 输入家庭成员姓名：函数首先提示用户输入要建立家庭的人的姓名。这

个步骤涉及基本的输入输出操作；

(2) 查找家庭成员：使用 `genealogy.findNode` 方法在家谱树中查找输入的成员。如果成员不存在，则输出相应的信息并返回；

(3) 检查家庭状态：检查找到的家庭成员是否已经有家庭（检查左子节点是否存在）。如果已有家庭，则输出相应信息并返回；

(4) 输入子女数量：如果家庭成员没有建立家庭，那么提示用户输入该成员的子女数量。这里使用 `inputInteger` 函数确保输入的是有效的整数值；

(5) 录入子女信息：根据输入的子女数量，循环提示用户输入每个子女的姓名。对于每个输入的子女姓名，检查其是否已在家谱树中存在。如果存在，则提示错误信息，并要求重新输入。若子女姓名在家谱中不存在，则将其作为新节点添加到家谱树中。这是通过 `genealogy.leftInsertNode` 和 `genealogy.rightInsertNode` 方法实现的；

(6) 输出家庭成员信息：完成子女信息的录入后，输出该家庭成员的所有直系子女的姓名，以确认家谱更新情况。

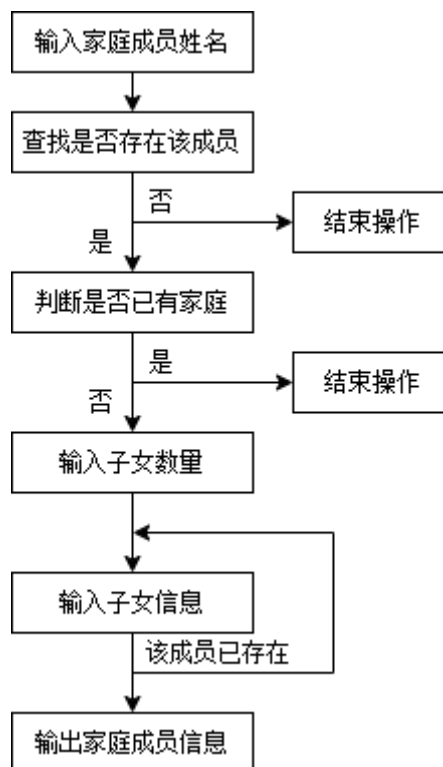


图 3.2.1.1 完善家谱功能实现流程图

3.2.2 完善家谱功能核心代码

```
void completeFamilyMembers(MyBinaryTree<Person>& genealogy)
{
    Person person;
    std::cout << "请输入要建立家庭的人的姓名：";
```

```

        std::cin >> person;
        char tmp[Person::NameMaxLength * 2];
        std::cout << std::endl;
        MyBinTreeNode<Person>* p = genealogy.findNode(person,
genealogy.getRoot());
        if (p == NULL) {
            std::cout << ">>> 未在家谱中找到" << person << std::endl;
            return;
        }
        else if (p->leftChild != NULL) {
            std::cout << ">>> " << person << "已建立家庭" << std::endl;
            return;
        }
        strcat(strcpy(tmp, person.name), "的儿女人数");
        int num = inputInteger(1, Person::DescendantsMaxNumber, tmp);
        std::cout << std::endl << ">>> 请依次输入" << person << "的儿女的姓
名" << std::endl << std::endl;
        Person* descendants = new(std::nothrow) Person[num];
        if (descendants == NULL) {
            std::cerr << "Error: Memory allocation failed." << std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
        for (int i = 0; i < num; i++) {
            std::cout << "请输入" << person << "的第" << i + 1 << "个儿女的
姓名: ";
            std::cin >> descendants[i];
            if (genealogy.findNode(descendants[i], genealogy.getRoot()))
            {
                std::cout << std::endl << ">>> " << descendants[i--] << "
在家谱中已存在" << std::endl << std::endl;
                continue;
            }
            if (i == 0)
                genealogy.leftInsertNode(genealogy.findNode(person,
genealogy.getRoot()), descendants[i]);
            else
                genealogy.rightInsertNode(genealogy.findNode(descendants[i
- 1], genealogy.getRoot()), descendants[i]);
        }
        std::cout << std::endl << ">>> " << person << "的下一代子孙是: ";
        for (int i = 0; i < num; i++)
            std::cout << (i == 0 ? "" : " ") << descendants[i];
        std::cout << std::endl;
    }
}

```

3.2.3 完善家谱功能示例

```
>>> 菜单: [1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型: [1]
请输入要建立家庭的人的姓名: Ancestor
请输入Ancestor的儿女人数 [整数范围: 1~16]: 4
>>> 请依次输入Ancestor的儿女的姓名
请输入Ancestor的第1个儿女的姓名: PersonA
请输入Ancestor的第2个儿女的姓名: PersonB
请输入Ancestor的第3个儿女的姓名: PersonC
请输入Ancestor的第4个儿女的姓名: PersonD
>>> Ancestor的下一代子孙是: PersonA PersonB PersonC PersonD
```

图 3.2.3.1 完善家谱功能示例

3.3 添加家庭成员功能的实现

3.3.1 添加家庭成员功能实现思路

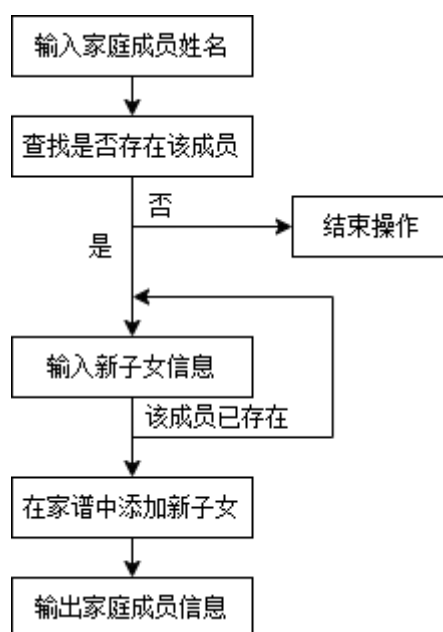


图 3.3.1.1 添加家庭成员功能实现流程图

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

(1) 输入要添加子女的家庭成员姓名: 用户首先输入想要为其添加子女的家庭成员的姓名;

(2) 在家谱中查找该成员: 使用 `genealogy.findNode` 方法在家谱树中查找输入的成员姓名。如果该成员不存在于家谱中, 则输出提示信息并返回;

(3) 输入新子女的姓名: 提示用户输入新添加的子女的姓名。检查输入的子女姓名是否已经存在于家谱中。如果已存在, 则提示错误信息并要求重新输入,

直到输入一个在家谱中不存在的姓名；

(4)在家谱中添加新子女：再次查找刚输入的家庭成员的节点。如果该成员还没有子女（左子节点为空），则将新子女添加为该成员的左子节点。如果该成员已有子女，则沿右子节点链向下查找，直到找到一个没有右子节点的子女节点，并将新子女添加为该节点的右子节点；

(5)输出更新后的家庭成员信息：输出添加新子女后，该家庭成员的所有子女的姓名，以确认家谱更新情况。

3.3.2 添加家庭成员功能核心代码

```
void addFamilyMembers(MyBinaryTree<Person>& genealogy)
{
    Person person, descendant;
    std::cout << "请输入要添加家庭成员的人的姓名：";
    std::cin >> person;
    if (!genealogy.findNode(person, genealogy.getRoot())) {
        std::cout << std::endl << ">>> 未在家谱中找到" << person <<
std::endl;
        return;
    }
    while (true) {
        std::cout << std::endl << "请输入" << person << "新添加的儿女的
姓名：";
        std::cin >> descendant;
        if (genealogy.findNode(descendant, genealogy.getRoot())) {
            std::cout << std::endl << ">>> " << descendant << "在家谱
中已存在" << std::endl;
            continue;
        }
        break;
    }
    MyBinTreeNode<Person>* p = genealogy.findNode(person,
genealogy.getRoot());
    std::cout << std::endl << ">>> " << person << "的下一代子孙是：";
    if (p->leftChild == NULL)
        genealogy.leftInsertNode(p, descendant);
    else {
        MyBinTreeNode<Person>* current = p->leftChild;
        std::cout << current->data << " ";
        while (current->rightChild != NULL) {
            current = current->rightChild;
            std::cout << current->data << " ";
        }
    }
}
```



```

        genealogy.rightInsertNode(current, descendant);
    }
    std::cout << descendant << std::endl;
}

```

3.3.3 添加家庭成员功能示例

```

>>> 菜单: [1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型: [2]
请输入要添加家庭成员的人的姓名: Ancestor
请输入Ancestor新添加的儿女的姓名: PersonE
>>> Ancestor的下一代子孙是: PersonA PersonB PersonC PersonD PersonE

```

图 3.3.3.1 添加家庭成员功能示例

3.4 解散家庭成员功能的实现

3.4.1 解散家庭成员功能实现思路

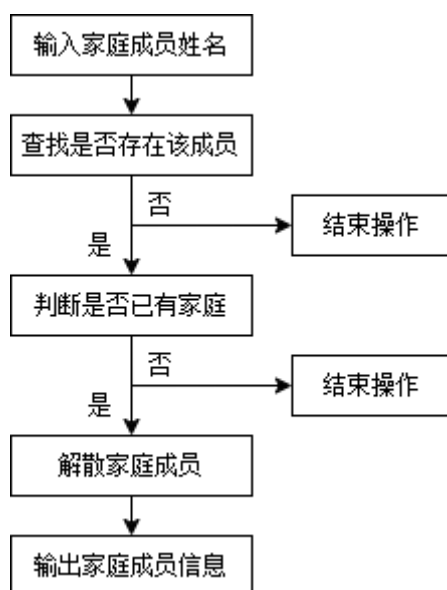


图 3.4.1.1 解散家庭成员功能实现流程图

解散家庭成员功能的函数名为 `removeFamilyMembers`，解散家庭成员功能实现的思路为：

- (1) 输入要解散家庭的成员姓名：用户首先输入要解散其家庭的家庭成员的姓名；
- (2) 在家谱中查找该成员：使用 `genealogy.findNode` 方法在家谱树中查找输入的成员姓名。如果该成员不存在于家谱中，则输出提示信息并返回；
- (3) 检查家庭成员状态：检查找到的家庭成员是否有家庭（即检查左子节点是否存在）。如果该成员没有家庭成员（左子节点为空），则输出提示信息并返回；

(4)输出并解散家庭成员：输出该家庭成员的所有直系家庭成员（通过中序遍历实现）。调用 `genealogy.destroy` 方法删除该成员的左子节点及其所有子节点，从而解散该家庭成员的家庭。输出被解散的家庭成员信息。

3.4.2 解散家庭成员功能核心代码

```
void removeFamilyMembers(MyBinaryTree<Person>& genealogy)
{
    Person person;
    std::cout << "请输入要解散家庭成员的人的姓名：";
    std::cin >> person;
    MyBinTreeNode<Person>* p = genealogy.findNode(person,
genealogy.getRoot());
    if (p == NULL) {
        std::cout << std::endl << ">>> 未在家谱中找到" << person <<
std::endl;
        return;
    }
    else if (p->leftChild == NULL) {
        std::cout << std::endl << ">>> " << person << "无家庭成员" <<
std::endl;
        return;
    }
    std::cout << std::endl << ">>> " << person << "的家庭成员是：";
    genealogy.inOrderOutput(p->leftChild);
    genealogy.destroy(p->leftChild);
    std::cout << std::endl << std::endl << ">>> " << person << "的家
庭成员已解散" << std::endl;
}
```

3.4.3 解散家庭成员功能示例

```
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[3]
请输入要解散家庭成员的人的姓名：Ancestor
>>> Ancestor的家庭成员是：PersonA PersonB PersonC PersonD PersonE
>>> Ancestor的家庭成员已解散
```

图 3.4.3.1 解散家庭成员功能示例

3.5 更改家庭成员姓名功能的实现

3.5.1 更改家庭成员姓名功能实现思路

更改家庭成员姓名功能的函数名为 `changeFamilyMembers`，更改家庭成员

姓名功能实现的思路为：

(1)输入要更改姓名的家庭成员的当前姓名：用户首先输入需要更改姓名的家庭成员的当前（更改前）姓名；

(2)在家谱中查找该成员：使用 `genealogy.findNode` 方法在家谱树中查找输入的成员姓名。如果该成员不存在于家谱中，则输出提示信息并返回；

(3)输入新姓名：用户输入该家庭成员的新（更改后）姓名。检查新姓名是否已存在于家谱中。如果新姓名已存在，则提示错误信息，并要求用户重新输入，直到输入一个在家谱中不存在的新姓名；

(4)更改家庭成员的姓名：调用 `genealogy.modifyNode` 方法，将家庭成员的当前姓名更改为新姓名；

(5)输出更名确认信息：输出一条确认信息，表明家庭成员的姓名已从原姓名更改为新姓名。

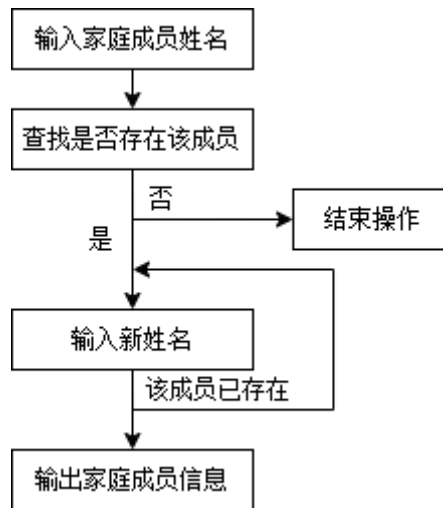


图 3.5.1.1 更改家庭成员姓名功能实现流程图

3.5.2 更改家庭成员姓名功能核心代码

```
void changeFamilyMembers(MyBinaryTree<Person>& genealogy)
{
    Person personBeforeChange, personAfterChange;
    std::cout << "请输入要更改家庭成员姓名的人的更改前姓名：";
    std::cin >> personBeforeChange;
    if (!genealogy.findNode(personBeforeChange, genealogy.getRoot()))
    {
        std::cout << std::endl << ">>> 未在家谱中找到 " <<
personBeforeChange << std::endl;
        return;
    }
    while (true) {
        std::cout << std::endl << "请输入要更改家庭成员姓名的人的更改后
```

```

    姓名: ";
        std::cin >> personAfterChange;
        if (genealogy.findNode(personAfterChange,
genealogy.getRoot())) {
            std::cout << std::endl << ">>> " << personAfterChange << "
在家谱中已存在" << std::endl;
            continue;
        }
        break;
    }
    genealogy.modifyNode(personBeforeChange, personAfterChange,
genealogy.getRoot());
    std::cout << std::endl << ">>> " << personBeforeChange << "已更名
为" << personAfterChange << std::endl;
}

```

3.5.3 更改家庭成员姓名功能示例

```

>>> 菜单: [1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型: [4]
请输入要更改家庭成员姓名的人的更改前姓名: Ancestor
请输入要更改家庭成员姓名的人的更改后姓名: 祖先
>>> Ancestor已更名为祖先

```

图 3.5.3.1 更改家庭成员姓名功能示例

3.6 统计家庭成员功能的实现

3.6.1 统计家庭成员功能实现思路

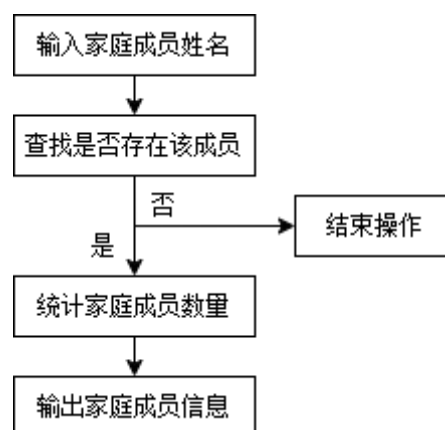


图 3.6.1.1 统计家庭成员功能实现流程图

统计家庭成员功能的函数名为 `countFamilyMembers`，统计家庭成员功能实现的思路为：

(1) 输入要统计的家庭成员姓名：用户首先输入想要统计其家庭成员的家庭

成员的姓名；

(2) 在家谱中查找该成员：使用 `genealogy.findNode` 方法在家谱树中查找输入的成员姓名。如果该成员不存在于家谱中，则输出提示信息并返回；

(3) 统计家庭成员数量：使用 `genealogy.getSize` 方法计算该家庭成员的后代数量（即该成员的左子树大小）；

(4) 输出家庭成员信息：输出该家庭成员的后代总数。如果有后代，使用 `genealogy.inOrderOutput` 方法输出所有后代的姓名。

3.6.2 统计家庭成员功能核心代码

```
void countFamilyMembers(MyBinaryTree<Person>& genealogy)
{
    Person person;
    std::cout << "请输入要统计的家庭成员的人的姓名：";
    std::cin >> person;
    MyBinTreeNode<Person>* p = genealogy.findNode(person,
genealogy.getRoot());
    if (p == NULL) {
        std::cout << std::endl << ">>> 未在家谱中找到" << person <<
std::endl;
        return;
    }
    int num = genealogy.getSize(p->leftChild);
    std::cout << std::endl << ">>> " << person << "共有" << num << "
个儿女";
    if (num != 0) {
        std::cout << ", 分别是：";
        genealogy.inOrderOutput(p->leftChild);
    }
    std::cout << std::endl;
}
```

3.6.3 统计家庭成员功能示例

```
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[5]
请输入要统计的家庭成员的人的姓名：祖先
>>> 祖先共有0个儿女
```

图 3.6.3.1 统计无家庭成员功能示例

```
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[5]
请输入要统计的家庭成员的人的姓名：Ancestor
>>> Ancestor共有5个儿女，分别是：PersonA PersonB PersonC PersonD PersonE
```

图 3.6.3.2 统计有家庭成员功能示例

3.7 异常处理功能的实现

3.7.1 动态内存申请失败的异常处理

在进行 `MyLinkNode` 类和 `MyBinTreeNode` 类等的动态内存申请时，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，这时程序将执行以下操作：

(1) 向标准错误流 `std::cerr` 输出一条错误消息 "Error: Memory allocation failed."，指出内存分配失败；

(2) 调用 `exit` 函数，返回错误码 `MEMORY_ALLOCATION_ERROR`（通过宏定义方式定义为-1），用于指示内存分配错误，并导致程序退出。

下面是动态内存申请的异常处理的一个代码示例：

```
template <typename Type>
MyBinaryTree<Type>::MyBinaryTree(Type& item)
{
    root = new(std::nothrow) MyBinTreeNode<Type>(item);
    if (root == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
}
```

3.7.2 MyQueue 类队列空的异常处理

在调用 `MyQueue` 类的 `deQueue` 和 `getFront` 等成员函数时，如果队列为空（`isEmpty()`返回 `true`），则函数返回 `false`，表示队列不包含任何元素。

3.7.3 MyBinaryTree 类自赋值的异常处理

`MyBinaryTree` 类重载了赋值运算符，用于将一个二叉树赋值给另一个二叉树。自赋值是一种常见的错误操作，可能导致资源泄漏和其他问题，该运算符重载函数对自赋值（将二叉树赋值给自身）的情况进行了处理。

程序会检查 `this` 指针和传入的 `other` 二叉树对象是否相同，如果 `this` 指针与 `other` 相同，表示自赋值操作，为了防止自赋值，运算符重载函数不会执行赋值操作，而是直接返回当前对象的引用 `*this`。这个错误处理机制确保了当尝试将二叉树赋值给自身时，不会导致资源泄漏或其他问题。

3.7.4 Person 结构体自赋值的异常处理

`Person` 结构体重载了赋值运算符，用于将一个 `Person` 对象赋值给另一个 `Person` 对象。自赋值是一种常见的错误操作，可能导致资源泄漏和其他问题，

该运算符重载函数对自赋值（将 `Person` 对象赋值给自身）的情况进行了处理。

程序会检查 `this` 指针和传入的 `other` 对象是否相同，如果 `this` 指针与 `other` 相同，表示自赋值操作，为了防止自赋值，运算符重载函数不会执行赋值操作，而是直接返回当前对象的引用 `*this`。这个错误处理机制确保了当尝试将 `Person` 对象赋值给自身时，不会导致资源泄漏或其他问题。

3.7.5 输入非法的异常处理

3.7.5.1 某家庭成员的儿女人数输入非法的异常处理

程序通过调用 `inputInteger` 函数输入某家庭成员的儿女人数。`inputInteger` 函数用于获取用户输入的整数，同时限制输入必须在指定的范围内，函数的代码如下：

```
int inputInteger(int lowerLimit, int upperLimit, const char* prompt)
{
    while (true) {
        std::cout << "请输入" << prompt << " [整数范围: " << lowerLimit
        << "~" << upperLimit << "]: ";
        double tempInput;
        std::cin >> tempInput;
        if (std::cin.good() && tempInput == static_cast<int>(tempInput)
        && tempInput >= lowerLimit && tempInput <= upperLimit) {
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
            return static_cast<int>(tempInput);
        }
        else {
            std::cerr << std::endl << ">>> " << prompt << "输入不合法，
            请重新输入" << prompt << "! " << std::endl << std::endl;
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
        }
    }
}
```

`inputInteger` 函数对输入非法的情况进行了处理，代码具体执行逻辑如下：

- (1) 进入一个无限循环，它会一直运行直到用户提供有效的输入；
- (2) 用户的输入被读取到 `tempInput` 变量中，这里采用 `double` 类型来接收输入以便后续检查；

(3) 进行输入验证：`std::cin.good()` 检查输入流的状态是否正常，确保没有发生数据类型输入错误，`tempInput==static_cast<int>(tempInput)` 检查用户输入是否为整数，通过将其转换为整数再比较，`tempInput>=lowerLimit`

和 `tempInput<=upperLimit` 确保输入在指定的范围内;

(4) 合法输入处理: 如果用户提供了合法的输入, 函数会清除输入流的错误状态, 丢弃输入缓冲区中的任何剩余内容, 然后返回转换后的整数值;

(5) 非法输入处理: 如果用户提供的输入不合法, 函数会输出错误消息, 清除输入流的错误状态, 丢弃输入缓冲区中的内容, 并继续循环以等待用户提供合法的输入。

3.7.5.2 某家庭成员姓名输入非法的异常处理

为了更好地对家谱管理系统的数据进行管理, 并提高程序的适用性与扩展性, 本程序对家庭成员信息的输入格式和数据类型进行了规定: 姓名输入要求为不超过 32 个英文字符或 16 个汉字字符组成的字符串, 超出部分将被截断。

相关操作通过 `Person` 结构体的重载输入 (右移) 运算符函数实现, 该重载函数的实现代码如下:

```
std::istream& operator>>(std::istream& in, Person& person)
{
    std::cin >> person.name;
    person.name[Person::NameMaxLength] = '\0';
    std::cin.clear();
    std::cin.ignore(INT_MAX, '\n');
    return in;
}
```

3.7.5.3 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:

```
int selectOptn(void)
{
    std::cout << std::endl << ">>> 菜单: [1]完善家谱 [2]添加家庭成员 [3]
解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统" << std::endl;
    std::cout << std::endl << "请选择操作类型: ";
    char optn;
    while (true) {
        optn = _getch();
        if (optn == 0 || optn == -32)
            optn = _getch();
        else if (optn >= '0' && optn <= '5') {
            std::cout << "[" << optn << "]" << std::endl << std::endl;
            return optn - '0';
        }
    }
}
```

这段代码会一直等待用户输入, 只有当用户输入有效的数字字符 ('0' 到 '5')

时，才会返回该数字的整数值，表示用户选择的选项。如果用户输入无效字符，循环会继续等待用户提供有效的输入。这段代码的具体执行逻辑如下：

(1) 显示提示信息，其中包含选项[1]到[5]，以及[0]退出系统的选项；

(2) 进入一个无限循环，以等待用户输入；

(3) 用户输入的字符会被`_getch()`函数获取。这个函数通常用于在控制台应用程序中获取单个字符而不显示在屏幕上。用户的输入存储在变量 `optn` 中；

(4) 代码检查用户输入是否是数字字符（'0'到'5'）或特殊的控制字符。如果用户按下非数字字符或特殊控制字符，它将不执行下面的操作；

(5) 如果用户输入是数字字符（'0'到'5'），它会在屏幕上显示用户输入的数字字符，并返回对应的整数值；

(6) 如果用户输入的不是数字字符（'0'到'5'），代码将保持在循环中，继续等待有效的输入，这确保了只有在用户输入正确的选项时才会退出循环。

4 项目测试

4.1 家庭成员姓名输入功能测试

家庭成员姓名输入要求为不超过 32 个英文字符或 16 个汉字字符组成的字符串，超出部分将被截断。

```
>>> [姓名输入要求] 不超过 32 个英文字符或 16 个汉字字符组成的字符串，超出部分将被截断
请输入祖先姓名：测试输入超过16个汉字字符组成的字符串的截断功能
>>> 家谱建立成功（祖先：测试输入超过16个汉字字符组成的字）
```

图 4.1.1 家庭成员姓名输入功能测试

4.2 完善家谱功能测试

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

输入某家庭成员的儿女人数时，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，可以验证程序对输入非法的情况进行了处理。

输入某家庭成员的儿女姓名时，输入在家谱中已存在的家庭成员，程序要求重新输入，可以验证程序对输入在家谱中已存在的家庭成员的情况进行了处理。

输入在家谱中已存在家庭成员的家庭成员，程序结束操作，可以验证程序对输入家谱中已存在家庭成员的家庭成员的情况进行了处理。

```

>>> 请建立家谱管理系统
>>> [姓名输入要求] 不超过 32 个英文字符或 16 个汉字字符组成的字符串，超出部分将被截断
请输入祖先姓名：Ancestor
>>> 家谱建立成功（祖先：Ancestor）
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[1]
请输入要建立家庭的人的姓名：TestPerson
>>> 未在家谱中找到TestPerson
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[1]
请输入要建立家庭的人的姓名：Ancestor
请输入Ancestor的儿女人数 [整数范围：1~16]：0
>>> Ancestor的儿女人数输入不合法，请重新输入Ancestor的儿女人数！
请输入Ancestor的儿女人数 [整数范围：1~16]：17
>>> Ancestor的儿女人数输入不合法，请重新输入Ancestor的儿女人数！
请输入Ancestor的儿女人数 [整数范围：1~16]：5.5
>>> Ancestor的儿女人数输入不合法，请重新输入Ancestor的儿女人数！
请输入Ancestor的儿女人数 [整数范围：1~16]：a
>>> Ancestor的儿女人数输入不合法，请重新输入Ancestor的儿女人数！
请输入Ancestor的儿女人数 [整数范围：1~16]：abc
>>> Ancestor的儿女人数输入不合法，请重新输入Ancestor的儿女人数！
请输入Ancestor的儿女人数 [整数范围：1~16]：4
>>> 请依次输入Ancestor的儿女的姓名
请输入Ancestor的第1个儿女的姓名：PersonA
请输入Ancestor的第2个儿女的姓名：PersonB
请输入Ancestor的第3个儿女的姓名：PersonC
请输入Ancestor的第4个儿女的姓名：PersonA
>>> PersonA在家谱中已存在
请输入Ancestor的第4个儿女的姓名：PersonD
>>> Ancestor的下一代子孙是：PersonA PersonB PersonC PersonD
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[1]
请输入要建立家庭的人的姓名：Ancestor
>>> Ancestor已建立家庭

```

图 4.2.1 完善家谱功能测试

4.3 添加家庭成员功能测试

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

输入某家庭成员的儿女姓名时，输入在家谱中已存在的家庭成员，程序要求重新输入，可以验证程序对输入在家谱中已存在的家庭成员的情况进行了处理。

```

>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[2]
请输入要添加家庭成员的人的姓名：TestPerson
>>> 未在家谱中找到TestPerson
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[2]
请输入要添加家庭成员的人的姓名：Ancestor
请输入Ancestor新添加的儿女的姓名：PersonE
>>> Ancestor的下一代子孙是：PersonA PersonB PersonC PersonD PersonE
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[2]
请输入要添加家庭成员的人的姓名：PersonA
请输入PersonA新添加的儿女的姓名：PersonB
>>> PersonB在家谱中已存在
请输入PersonA新添加的儿女的姓名：PersonA1
>>> PersonA的下一代子孙是：PersonA1
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[2]
请输入要添加家庭成员的人的姓名：PersonA
请输入PersonA新添加的儿女的姓名：PersonA1
>>> PersonA1在家谱中已存在
请输入PersonA新添加的儿女的姓名：PersonA2
>>> PersonA的下一代子孙是：PersonA1 PersonA2

```

图 4.3.1 添加家庭成员功能测试

4.4 解散家庭成员功能测试

```

>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[3]
请输入要解散家庭成员的人的姓名：TestPerson
>>> 未在家谱中找到TestPerson
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[3]
请输入要解散家庭成员的人的姓名：PersonA
>>> PersonA的家庭成员是：PersonA1 PersonA2
>>> PersonA的家庭成员已解散
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[3]
请输入要解散家庭成员的人的姓名：PersonE
>>> PersonE无家庭成员

```

图 4.4.1 解散家庭成员功能测试

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

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

4.5 更改家庭成员姓名功能测试

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

输入某家庭成员的更改后姓名时，输入在家谱中已存在的家庭成员，程序要求重新输入，可以验证程序对输入在家谱中已存在的家庭成员的情况进行了处理。

```
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[4]
请输入要更改家庭成员姓名的人的更改前姓名：PersonF
>>> 未在家谱中找到PersonF
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[4]
请输入要更改家庭成员姓名的人的更改前姓名：PersonE
请输入要更改家庭成员姓名的人的更改后姓名：PersonA
>>> PersonA在家谱中已存在
请输入要更改家庭成员姓名的人的更改后姓名：PersonF
>>> PersonE已更名为PersonF
```

图 4.5.1 更改家庭成员姓名功能测试

4.6 统计家庭成员功能测试

```
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[5]
请输入要统计的家庭成员的人的姓名：Ancestor
>>> Ancestor共有5个儿女，分别是：PersonA PersonB PersonC PersonD PersonF
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[5]
请输入要统计的家庭成员的人的姓名：PersonE
>>> 未在家谱中找到PersonE
>>> 菜单：[1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型：[5]
请输入要统计的家庭成员的人的姓名：PersonA
>>> PersonA共有0个儿女
```

图 4.6.1 统计家庭成员功能测试

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

输入在家谱中已存在家庭成员的家庭成员，程序输出该家庭成员的儿女数量和儿女姓名，可以验证程序对输入家谱中已存在家庭成员的家庭成员的情况进行了处理。

输入在家谱中不存在家庭成员的家庭成员，程序输出该家庭成员的儿女数量为 0，可以验证程序对输入家谱中不存在家庭成员的家庭成员的情况进行了处理。

4.7 退出家谱管理系统功能测试

在操作家谱管理系统时，选择选项[0]退出家谱管理系统。

```
>>> 菜单: [1]完善家谱 [2]添加家庭成员 [3]解散家庭成员 [4]更改家庭成员姓名 [5]统计家庭成员 [0]退出系统
请选择操作类型: [0]
>>> 家谱管理系统已退出
```

图 4.7.1 退出家谱管理系统功能测试

5 集成开发环境与编译运行环境

Windows 系统: Windows 11 x64

Windows 集成开发环境: Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境: 本项目适用于 x86 架构和 x64 架构

Linux 系统: CentOS 7 x64

Linux 编译命令:

```
g++ '/root/桌面/Share_Folder/genealogy_management_system.cpp' -o '/root/桌面/Share_Folder/genealogy_management_system' -lncurses
```

Linux 运行命令:

```
'/root/桌面/Share_Folder/genealogy_management_system'
```



图 5.1 Linux 环境程序运行示例

本项目使用条件编译解决 Windows 系统和 Linux 系统编译环境的差异, 示例

代码如下。

```
#ifdef _WIN32
#include <conio.h>
#elif __linux__
#include <ncurses.h>
#endif
```