

数据结构课程设计
项目说明文档

家谱管理系统

作者姓名:	汪明杰
学 号:	1851055
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 项目分析	4
1.1 项目背景	4
1.2 项目需求分析	4
1.3 项目要求	5
1.3.1 功能要求	5
1.3.2 输入格式	5
1.3.3 输出格式	5
1.3.4 项目示例	5
2 项目设计	6
2.1 数据结构设计	6
2.2 类设计	6
2.2.1 向量类 (Vector)	6
2.2.2 双向链表类 (List)	8
2.2.3 队列类 (Queue)	9
2.2.4 Pair类 (Pair)	10
2.2.5 家谱树类 (familyTree)	11
3 项目实施	13
3.1 项目主体功能	13
3.1.1 项目主体功能流程图	13
3.1.2 项目主体功能代码	14
3.2 家谱树管理代码	15
3.2.1 建立家庭	15
3.2.2 添加家庭成员	16
3.2.3 解散家庭	17
3.2.4 修改姓名	17
3.3 绘制家谱树	18
3.3.1 横向绘制家谱树	19
3.3.2 竖向绘制家谱树	20
4 项目测试	28
4.1 建立家谱测试	28

4.2 完善只有单个子女的家庭	28
4.3 完善有多个子女的家庭	29
4.4 多代家谱树测试	29
4.5 添加家庭成员测试	30
4.6 解散局部家庭测试	30
4.7 更改家庭成员姓名测试	31
4.8 横版家谱树测试	31
4.9 边界情况	32
4.9.1 多代单传家谱树测试	32
4.9.2 一代多子女家谱树测试	32

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人物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。

经历了历朝历代的连年战乱和社会动荡，历史上传世的家谱几乎丧失殆尽，许多家族的世系也因此断了线、失了传。流传至今的古代家谱，大多是明清两代纂修。在我国明清时期，出现了专门替人伪造家谱世系的“谱匠”。

本项目旨在完成一个家谱系统，并实现家谱树所需要的查找、插入、搜索和删除等相关功能。

1.2 项目需求分析

针对于家谱管理系统这一项目，本项目在实现的过程中，应当能够满足以下的需求：

✓ 功能完善

系统应当实现家谱树中所有需要的功能，即包括插入、删除、查找、修改等操作。

✓ 执行效率高

针对数据量比较大的情况，本系统也应该具有在较短时间内生成正确的家谱树的能力。

✓ 健壮性

当用户输入的数据非法时，系统应当能够识别并处理错误，而非直接崩溃退出。

✓ 可视化

在用户输入生成的家谱树建成后，系统应当可以通过绘制图像的方法使得家谱树可以被直观的看见，更易于用户的使用。

1.3 项目要求

1.3.1 功能要求

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。

1.3.2 输入格式

输入相应的操作。

1.3.3 输出格式

执行相应的操作，绘制家谱树。

1.3.4 项目示例

```

**          家谱管理系统          **
=====
**          请选择要执行的操作：          **
**          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                **
=====
首先建立一个家谱！
请输入祖先的姓名：P0
此家谱的祖先是：P0

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的儿女人数：2
请依次输入P0的儿女的姓名：P1 P2
P0的第一代子孙是：P1 P2

请选择要执行的操作：A
请输入要建立家庭的人的姓名：P1
请输入P1的儿女人数：3
请依次输入P1的儿女的姓名：P11 P12 P13
P1的第一代子孙是：P11 P12 P13

请选择要执行的操作：B
请输入要添加儿子（或女儿）的人的姓名：P2
请输入P2新添加的儿子（或女儿）的姓名：P21
P2的第一代子孙是：P21

请选择要执行的操作：C
请输入要解散家庭的人的姓名：P2
要解散家庭的人是：P2
P2的第一代子孙是：P21

请选择要执行的操作：D
请输入要更改姓名的人的目前姓名：P13
请输入更改后的姓名：P14
P13已更名为P14

请选择要执行的操作：E
Press any key to continue_
```

2 项目设计

2.1 数据结构设计

本项目需要实现存储一个家谱树，也即多叉树。多叉树的存储有两种方式：第一种是左子女、右兄弟，即采用二叉树的存储结构存储多叉树。其中，每一个节点的左节点是其长子，右节点则是其兄弟；第二种方式则是每一个节点内部存储有一个**向量（Vector）**，用来存储所有的子女。本项目中采用了第二种方法进行实现，即每一个节点内部都存储一个 **children**，用来存取所有的儿子节点。

此外，本项目在绘制竖版树状图的时候，进行了**层次遍历（Level Traverse）**操作。在层次遍历过程中，使用**队列（Queue）**作为辅助工具。

2.2 类设计

2.2.1 向量类（Vector）

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。需要注意的是，向量具有以下两个特性：

（1）顺序序列：

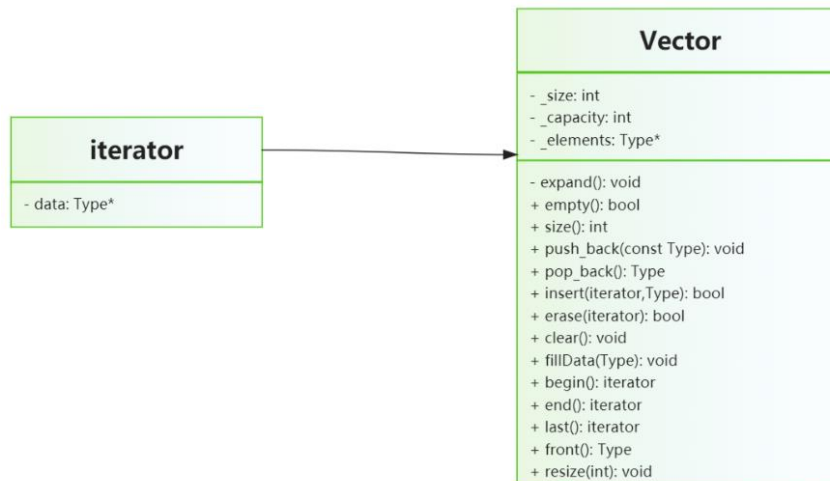
顺序容器中的元素按照严格的线性顺序排序，可以通过元素在序列中的位置访问对应的元素。本类中即通过重载取下标运算符[]实现了此功能。

（2）动态数组：

支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算术进行该操作，提供了在序列末尾相对快速地添加或者删除元素的操作。

本项目为了实现向量类，在内部定义了一个动态指针以及当前数组的大小。同时，封装了较多的函数以便使用。此外，为了便于对向量进行遍历、插入、删除、查找等操作，增加了一个 **iterator** 类。**iterator** 类内部存储一个数组元素指针，可以直接对向量的每个元素进行操作。同时，通过运算符重载相应的自增、自减、判等等操作。

该类和其内部的 **iterator** 类的 UML 图如下所示：



本类中的主要函数如下所示：

◆ `inline int size()const`

返回向量中元素的个数，也即_size 的大小。

◆ `inline bool empty()const`

判断向量是否为空，也即_size 是否为 0。

◆ `void push_back(const Type data)`

在向量尾端加入一个元素。

◆ `Type pop_back()`

删除向量最后一个元素，并且加以返回。

◆ `bool insert(const Vector<Type>::iterator place, Type item)`

在指定迭代器的位置插入元素，返回是否插入成功。

◆ `bool erase(const Vector<Type>::iterator place)`

删除指定迭代器位置的元素，返回是否删除成功。

◆ `void clear()`

清空向量中所有元素。

◆ `void fillData(const Type data)`

将向量中的元素统一赋值为同一个值。

◆ `iterator begin()`

返回向量首元素的迭代器。

◆ `iterator end()`

返回向量尾元素的下一个位置的迭代器。

◆ `iterator last()`

返回向量尾元素的迭代器。

◆ `Type& front()const`

返回向量的首个元素值。

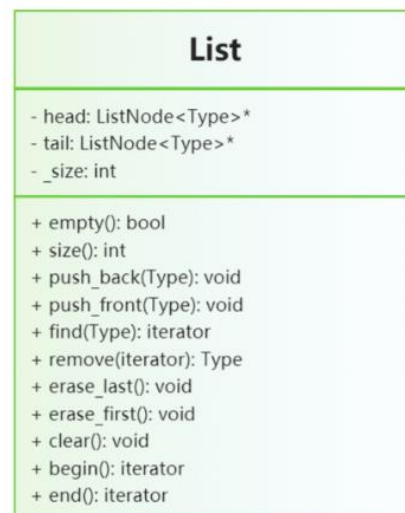
◆ `void resize(int sz)`

将向量重新设定大小，如果变小则删除多余元素。

2.2.2 双向链表类 (List)

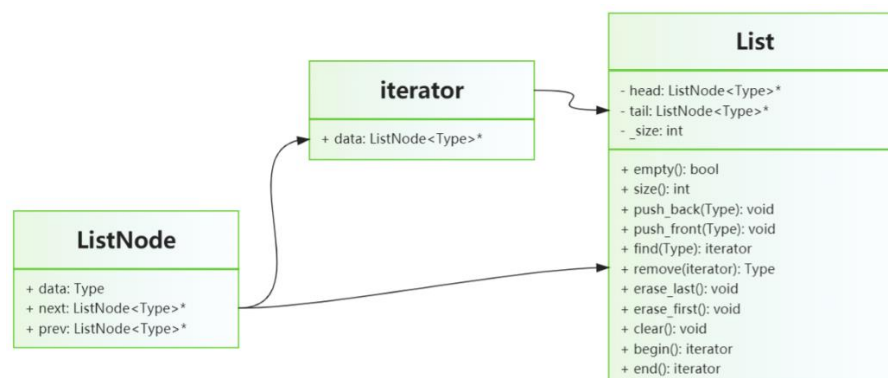
链表的实现原理大同小异，不同之处在于：是否带头结点、是否带尾结点、每一个结点是否带前驱结点等。为了使得链表中各种操作的时间复杂度都尽可能低，因此这里选择了带头结点和尾结点的双向链表来实现链表中的各种操作。也即：选择了牺牲空间来达到降低时间复杂度的效果。

链表的 UML 图如下所示：



为了便于对链表进行遍历、插入、删除、查找等操作，增加了一个 `iterator` 类。`iterator` 类内部存储一个链表节点指针，同时，通过运算符重载相应的自增、自减、判等等操作。

`iterator` 类、`ListNode` 类和 `List` 类的关系如下图所示：



其中，`List` 类中的主要函数如下所示：

◆ `inline int size()const`

返回链表中结点的个数，不包括头结点。

◆ `inline bool empty()const`

判断链表是否为空，也即链表中结点的个数是否为 0。

◆ `void push_back(Type data)`

在链表尾部插入新的数据，也即新增一个结点并且加入到链表末端。

◆ **void push_front(Type data)**

在链表头部插入新的数据，也即新增一个结点并且加入到链表的头部。

◆ **iterator find(const Type& data) const**

在链表中查找值为 data 的元素是否存在，返回该位置的迭代器，若查找失败返回空指针对应的迭代器。

◆ **Type remove(iterator index)**

移除迭代器所处位置的元素，返回移除位置元素的值。

◆ **void erase_last()**

移除链表末端的元素，即最后的结点。

◆ **void erase_first()**

移除链表首端的元素，即第一个结点。

◆ **void clear()**

清空链表，即删除链表中所有的结点。

◆ **iterator begin()**

返回链表第一个元素所在位置的迭代器。

◆ **iterator end()**

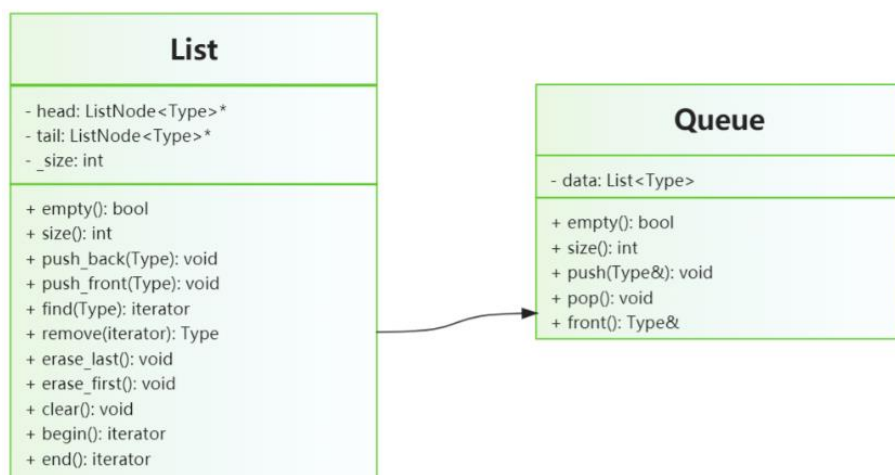
返回链表尾结点后的空结点的迭代器。

2.2.3 队列类 (Queue)

队列是计算机程序中常用的数据结构，常常用于计算机模拟现实事务，如：排队等。队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，因此是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，被称为空队列。

队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

队列的包括了顺序队列和循环队列，实现存储的底层数据可以通过向量或者链表完成。本项目中的队列以链表作为底层数据结构，实现了顺序队列。其 UNL 图如下所示：



队列中主要函数如下所示：

◆ `inline bool empty()const`

判断队列是否为空，也即队列内部链表是否为空。

◆ `inline int size()const`

返回队列中链表节点的个数。

◆ `void push(const Type& i)`

在队列尾部加入一个元素，也即入队。

◆ `void pop()`

删除队列头部的元素，也即出队。

◆ `const Type& front()const`

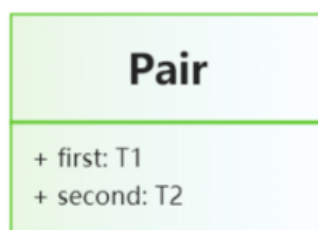
获取队首的元素值。

通过上述函数操作，即完成了一个队列所需要的最基本操作。

2.2.4 Pair 类（Pair）

为了方便对于存在一定关系的数据进行操作，Pair 类能够同时存储两种不同类型的数据结构。因此，在函数需要同时返回两个参数的时候，该类就能发挥比较便利的作用。

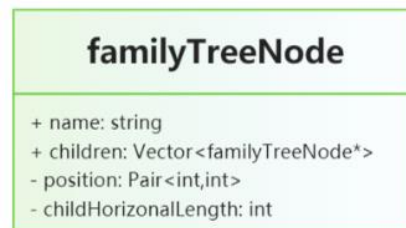
该类的 UML 图如下所示：



因此，在使用 Pair 类的时候，即可同时存储两种数据类型。

2.2.5 家谱树类 (familyTree)

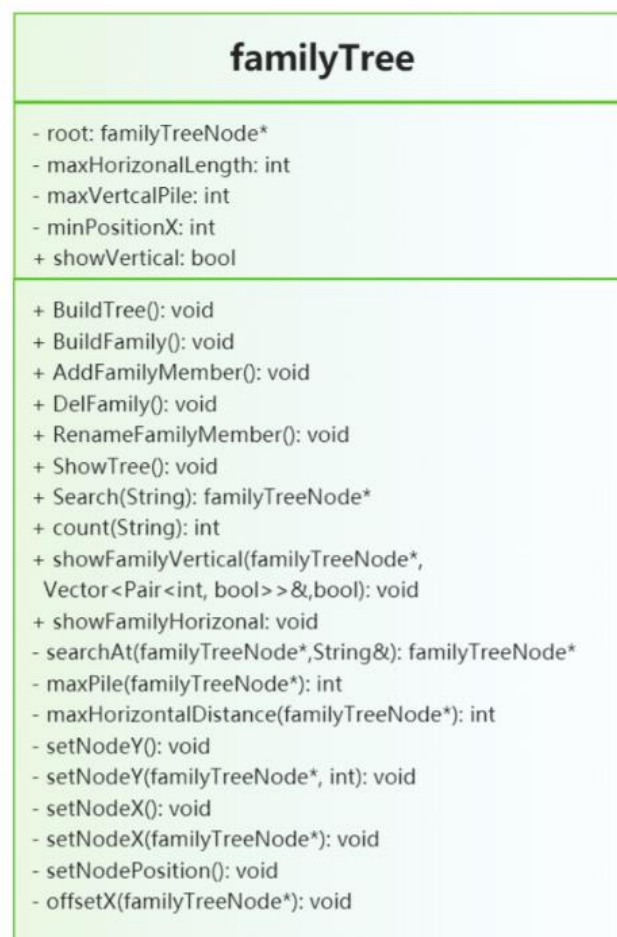
为了实现家谱树，本项目首先定义了家谱树节点，内部包括了当前节点的名字、子结点向量、所处位置等。其 UML 图如下所示：



同时，本系统封装了一个家谱树类 (**familyTree**)。将家谱树中需要使用到的初始化、添加家庭成员、删除家庭成员、修改家庭成员、绘制家谱树等函数封装在内。

此外，为了直观的绘制出家谱树，本系统提供了两种绘制方法，横向绘制和竖向绘制。横向绘制直接通过递归来进行绘制；竖向绘制前先计算出每一节点在绘制时的纵横坐标，在绘制过程中通过层次遍历来进行绘制操作。

家谱树的 UML 图如下：



本类中相关函数较多，其功能如下所示：

◆ **void BuildTree()**

用于家谱树的初始化，即设置该家谱树祖先的相关信息。

◆ **void BuildFamily()**

用于给家谱树中一个还未建立过家庭的成员建立家庭。

◆ **void AddFamilyMember()**

为一个家庭成员增加一个子女。

◆ **void DelFamily()**

解散某一家庭成员的家庭。

◆ **void RenameFamilyMember()**

更改某一家庭成员的姓名。

◆ **void ShowTree()**

绘制当前家庭的家族谱树。

◆ **familyTreeNode* search(const string& name) const**

查找某一家庭成员在家谱树中所对应的节点。

◆ **inline int count(const string& name) const**

返回家谱树中姓名所对应的成员数量。

◆ **void showFamilyVertical(familyTreeNode*, Vector<Pair>, bool)**

以横向的方式绘制家谱树。

◆ **void showFamilyHorizontal();**

以竖向的方式绘制家谱树。

◆ **familyTreeNode* searchAt(familyTreeNode*, const string&) const**

递归的在节点中查询某一姓名。

◆ **int maxPile(familyTreeNode*)**

递归的计算某一节点子树的最大层数。

◆ **int maxHorizontalDistance(familyTreeNode* ptr)**

计算某一节点子树竖向绘制所需要的距离。

◆ **void setNodeY()**

设置家谱树所有节点竖向绘制的纵坐标。

◆ **void setNodeY(familyTreeNode* ptr, int pile)**

设置某一节点竖向绘制的纵坐标。

◆ **void setNodeX()**

设置家谱树所有节点竖向绘制的横坐标。

◆ **void setNodeX(familyTreeNode* ptr)**

设置某一节点竖向绘制的横坐标。

◆ **void setNodePosition()**

设置家谱树所有节点竖向绘制的坐标。

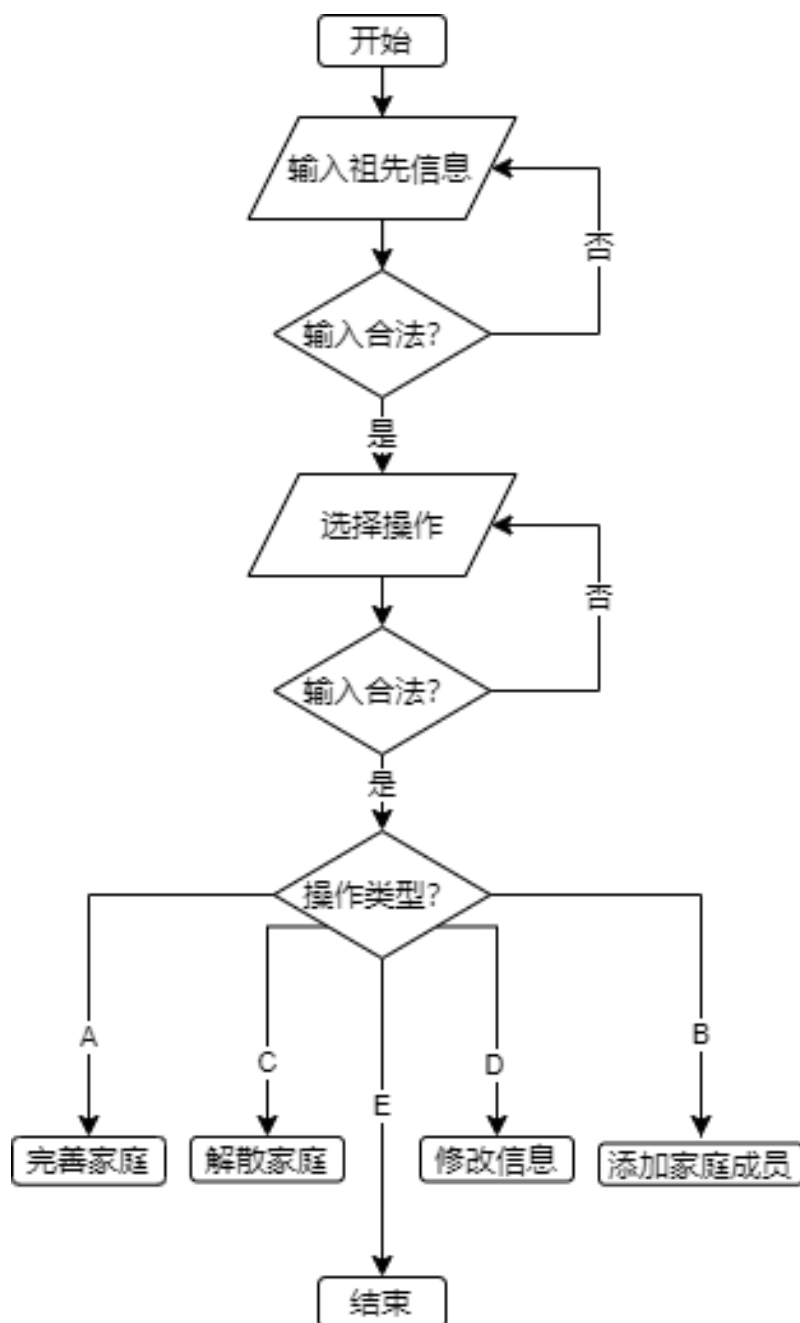
◆ **void offsetX(familyTreeNode* ptr)**

修正竖向绘制所出现的横坐标偏差。

3 项目实施

3.1 项目主体功能

3.1.1 项目主体功能流程图



3.1.2 项目主体功能代码

```
int main()
{
    cout << endl << "*** 家谱管理系统 ***" << endl;
    cout << "===== " << endl;
    cout << "***\t\t 请选择要执行的操作: \t\t**" << endl;
    cout << "***\t\tA --- 完善    家庭\t\t**" << endl;
    cout << "***\t\tB --- 添加家庭成员\t\t**" << endl;
    cout << "***\t\tC --- 解散局部家庭\t\t**" << endl;
    cout << "***\t\tD --- 更改家庭成员姓名\t\t**" << endl;
    cout << "***\t\tE --- 退出    程序\t\t**" << endl;
    cout << "***\t\tF --- 切换树形显示模式\t\t**" << endl;
    cout << "===== " << endl;
    familyTree tree;
    tree.BuildTree();
    char op;
    bool exitTree = false;
    while (!exitTree)
    {
        cout << endl << "请选择要执行的操作: ";
        cin >> op;
        switch (op)
        {
            case 'A':
            case 'a':
                tree.BuildFamily();
                tree.ShowTree();
                break;
            case 'B':
            case 'b':
                tree.AddFamilyMember();
                tree.ShowTree();
                break;
            case 'C':
            case 'c':
                tree.DelFamily();
                tree.ShowTree();
                break;
            case 'D':
            case 'd':
                tree.RenameFamilyMember();
```

```

        tree.ShowTree();
        break;
    case 'E':
    case 'e':
        exitTree = true;
        break;
    case 'F':
    case 'f':
        if (tree.showVertical)
        {
            cout << "当前树的显示模式已切换为横向显示! " << endl;
            tree.showVertical = false;
        }
        else
        {
            cout << "当前树的显示模式已切换为竖向显示! " << endl;
            tree.showVertical = true;
        }
        tree.ShowTree();
        break;
    default:
        cin.clear();
        cin.ignore(1024, '\n');
        cout << "请输入合法的执行操作符" << endl;
    }
}
return 0;
}

```

3.2 家谱树管理代码

3.2.1 建立家庭

```

void familyTree::BuildFamily()
{
    cout << "请输入要建立家庭的人的姓名: ";
    string name;
    cin >> name;
    familyTreeNode* familyMember = search(name);
    if (familyMember == nullptr)
    {
        cout << "本家谱内不存在该家庭成员! " << endl;
    }
}

```

```

        return;
    }
    if (!familyMember->children.empty())
    {
        cout << "该成员已经建立过家庭! " << endl;
        return;
    }
    int numChildren;
    cout << "请输入 " << name << "的儿女个数: ";
    cin >> numChildren;
    if (cin.fail() || numChildren < 0)
    {
        cout << "请输入一个正整数! " << endl;
        cin.clear();
        cin.ignore(1024, '\n');
        return;
    }
    cout << "请依次输入 " << name << "的儿女的姓名";
    for (int i = 0; i < numChildren; ++i)
    {
        cin >> name;
        if (!count(name))
        {
            familyMember->children.push_back(new familyTreeNode(name));
        }
        else
        {
            cout << "姓名为 " << name << "的成员已存在! " << endl;
        }
    }
}

```

3.2.2 添加家庭成员

```

void familyTree::AddFamilyMember()
{
    cout << "请输入要添加儿子（或女儿）的人的姓名: ";
    string name;
    cin >> name;
    familyTreeNode* familyMember = search(name);
    if (familyMember == nullptr)
    {
        cout << "本家谱内不存在该家庭成员! " << endl;
    }
}

```



```

        return;
    }
    cout << "请输入" << name << "新添加儿子（或女儿）的姓名：";
    cin >> name;
    if (!count(name))
    {
        familyMember->children.push_back(new familyTreeNode(name));
    }
    else
    {
        cout << "姓名为" << name << "的成员已存在！" << endl;
    }
}

```

3.2.3 解散家庭

```

void familyTree::DelFamily()
{
    cout << "请输入要解散家庭的人的姓名";
    string name;
    cin >> name;
    familyTreeNode* familyMember = search(name);
    if (familyMember == nullptr)
    {
        cout << "本家谱内不存在该家庭成员！" << endl;
        return;
    }
    cout << "要解散家庭的人是" << name << endl;
    familyMember->children.clear();
}

```

3.2.4 修改姓名

```

void familyTree::RenameFamilyMember()
{
    cout << "请输入要更改姓名的人的目前姓名";
    string name;
    cin >> name;
    familyTreeNode* familyMember = search(name);
    if (familyMember == nullptr)
    {
        cout << "本家谱内不存在该家庭成员！" << endl;
        return;
    }
}

```

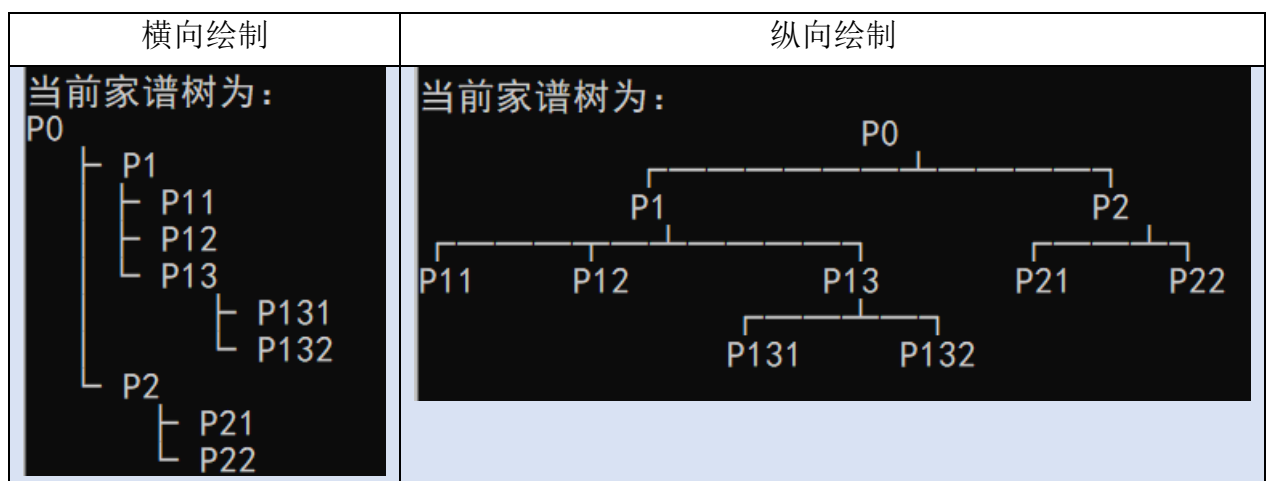
```

}
cout << "请输入更改后的姓名: ";
cin >> name;
if (!count(name))
{
    cout << familyMember->name << "已更名为" << name << endl;
    familyMember->name = name;
}
else
{
    cout << "姓名为" << name << "的成员已存在!" << endl;
}
}
}

```

3.3 绘制家谱树

本系统在绘制家谱树过程中，提供了两种模式：一种是横向绘制的方法；另一种则是纵向绘制的方法。示例如下所示：



用户可以在操作界面通过操作 F 来选择使用横向绘制还是纵向绘制：

```

if (tree.showVertical)
{
    cout << "当前树的显示模式已切换为横向显示!" << endl;
    tree.showVertical = false;
}
else
{
    cout << "当前树的显示模式已切换为竖向显示!" << endl;
    tree.showVertical = true;
}

```

在绘制家谱树的时候，会根据当前树形来选择绘制：

```
void familyTree::ShowTree()
{
    cout << endl << "当前家谱树为: " << endl;

    if (showVertical)
        showFamilyHorizontal();
    else
    {
        Vector<Pair<int, bool>> memberPlace;
        showFamilyVertical(root, memberPlace, false);
    }
}
```

3.3.1 横向绘制家谱树

横向绘制家谱树主要是通过递归函数来完成的，即计算出当前正在绘制的层数，依次进入子女结点进行绘制。代码如下：

```
void familyTree::showFamilyVertical(familyTreeNode* ptr, Vector<Pair<int, bool>>& memberPlace, bool lastMember) const
{
    if (memberPlace.empty())
    {
        cout << ptr->name << endl;
        memberPlace.push_back(Pair<int, bool>(ptr->name.length(), true));
    }
    else
    {
        //将memberPlace 除最后一项外的元素全部以| 形式输出(或者为空格)
        //最后一项根据是否为lastMember 确定
        int blankNumber = 0;
        //对于前面n-2 个元素
        for (int i = 0; i < memberPlace.size() - 1; ++i)
        {
            while (memberPlace[i].first > blankNumber++)
            {
                cout << " ";
            }

            if (memberPlace[i].second)
                cout << "|";
        }
    }
}
```

```

        else
            cout << "  ";
    }

    while (memberPlace[memberPlace.size() - 1].first > blankNumber+
+)
    {
        cout << " ";
    }
    if (lastMember)
        cout << "L ";
    else
        cout << "└ ";
    cout << ptr->name << endl;
    memberPlace.push_back(Pair<int, bool>(ptr->name.length() + (*mem
berPlace.last()).first, true));
}

if (!ptr->children.empty())
{
    for (auto i = ptr->children.begin(); i != ptr->children.last();
++i)
    {
        showFamilyVertical(*i, memberPlace, false);
    }
    // 到最后一个了, 此时应该把上一层的second 更改为false
    if (!memberPlace.empty())
        memberPlace[memberPlace.size() - 1].second = false;
    showFamilyVertical(*ptr->children.last(), memberPlace, true);
}
memberPlace.pop_back();
}

```

3.3.2 竖向绘制家谱树

竖向绘制家谱树相对而已比较复杂。在具体绘制之前, 需要先计算出每一个节点在绘制时的横坐标和纵坐标, 该过程通过递归函数来完成计算。

在获得每一个节点的坐标后, 在绘制竖版家谱树的过程中, 借助队列来实现层次遍历。即不断的从队列中取出节点, 绘制该节点的信息, 再将该节点的所有子女加入到队列中。

本过程中涉及到的函数如下所示:

```

void familyTree::showFamilyHorizontal()
{
    if (root == nullptr)
        return;
    // 获取每一个结点的位置
    this->setNodePosition();

    // 利用队列层次遍历输出
    Queue<familyTreeNode*> treeQueue;
    treeQueue.push(root);
    int nodeNumber = 1; // 当前层队列结点数
    int positionX = 0, positionY = 1;

    Queue<Pair<familyTreeNode*, Pair<char,int>>> LineQueue;
    LineQueue.push(Pair<familyTreeNode*, Pair<char,int>>(root, Pair<char,int>('o',0)));

    int positionLineX = 0;
    while (!treeQueue.empty())
    {
        bool newPart = false;
        // 绘制线条
        for (int i = 0; i < nodeNumber; ++i)
        {
            familyTreeNode* ptr = LineQueue.front().first;

            char mark = LineQueue.front().second.first;
            int halfPlace = LineQueue.front().second.second;
            LineQueue.pop();

            if (mark == 's' && !newPart)
                newPart = true;

            // 把ptr的子结点加入
            for (int j = 0; j < ptr->children.size(); ++j)
            {
                if (ptr->children.size() == 1)
                {
                    LineQueue.push(Pair<familyTreeNode*, Pair<char,int>
>
                    (ptr->children[j], Pair<char,int>('o',
                    ptr->position.first+ptr->name.length()/2)))
;
                }
            }
        }
    }
}

```

```

        else if (j == 0)
        {
            LineQueue.push(Pair<familyTreeNode*, Pair<char, int>
>>
                (ptr->children[j], Pair<char, int>('s',
                ptr->position.first + ptr->name.length() /
2)));
        }
        else if (j == ptr->children.size() - 1)
        {
            LineQueue.push(Pair<familyTreeNode*, Pair<char, int>
>>
                (ptr->children[j], Pair<char, int>('l',
                ptr->position.first + ptr->name.length() /
2)));
        }
        else
        {
            LineQueue.push(Pair<familyTreeNode*, Pair<char, int>
>>
                (ptr->children[j], Pair<char, int>('n',
                ptr->position.first + ptr->name.length() /
2)));
        }
    }

    if (ptr == root)
        break;

    // 处理ptr
    while (positionLineX < ptr->position.first)
    {
        if (newPart && positionLineX >= halfPlace)
        {
            cout << "└";
            cout << " ";
            positionLineX += 2;
            newPart = false;
        }
        else
        {
            switch (mark)
            {
                case 'o':

```

```

        case 's':
            cout << " ";
            ++positionLineX;
            break;
        case 'l':
        case 'n':
            cout << "-";
            cout << " ";
            positionLineX += 2;
            break;
    }
}

switch (mark)
{
    case 'o':
        cout << "|";
        cout << " ";
        break;
    case 's':
        cout << "┌";
        cout << " ";
        break;
    case 'l':
        cout << "┐";
        cout << " ";
        break;
    case 'n':
        cout << "└";
        cout << " ";
        break;
}
positionLineX += 2;

if (i == nodeNumber - 1)
    cout << endl;
}

positionLineX = 0;

int newNodeNumber = 0;
// 绘制结点
for (int i = 0; i < nodeNumber; ++i)

```

```

    {
        familyTreeNode* ptr = treeQueue.front();
        treeQueue.pop();
        // 将ptr 的元素全部入队
        newNodeNumber += ptr->children.size();
        for (auto i = ptr->children.begin(); i != ptr->children.end
()); ++i)
        {
            treeQueue.push(*i);
        }
        // 将ptr 的name 输出, 注意需要利用空格到达当前的x 坐标的位置
        while (positionX < ptr->position.first)
        {
            cout << " ";
            ++positionX;
        }
        cout << ptr->name << " ";
        positionX += ptr->name.length() + 1;
    }
    // 当前层结束, 输出一个回车
    cout << endl;
    positionX = 0;
    ++positionY;
    nodeNumber = newNodeNumber;
}
}

int familyTree::maxPile(familyTreeNode* ptr)
{
    if (ptr->children.empty())
        return 1;
    else
    {
        int maxPileOfChild = 1;
        for (auto i = ptr->children.begin(); i != ptr->children.end();
++i)
            maxPileOfChild = MAX(maxPileOfChild, maxPile(*i));
        return maxPileOfChild;
    }
}

int familyTree::maxHorizontalDistance(familyTreeNode* ptr)
{
    // 首先获取自身字符串的长度

```



```

    int childLength = 0;
    if (ptr->children.empty())
    {
        ptr->childHorizontalLength = 0;
        return ptr->name.length();
    }
    else
    {
        for (auto i = ptr->children.begin(); i != ptr->children.end();
++i)
        {
            childLength += maxHorizontalDistance(*i);
        }
        // 空格
        childLength += (ptr->children.size() - 1) * 5;
    }
    ptr->childHorizontalLength = childLength;

    return MAX(int(ptr->name.length()), childLength);
}

void familyTree::setNodeY(familyTreeNode* ptr, int pile)
{
    ptr->position.second = pile;
    for (auto i = ptr->children.begin(); i != ptr->children.end(); ++i)
        setNodeY(*i, pile + 1);
}

void familyTree::setNodeY()
{
    if (root != nullptr)
        setNodeY(root, 0);
}

void familyTree::setNodeX()
{
    // 首先获取最大坐标
    if (root == nullptr)
        return;

    this->maxHorizontalLength = maxHorizontalDistance(root);

    root->position.first = maxHorizontalLength / 2 - int(root->name.leng
gth()) / 2;

```

```

    this->minPositionX = 0;
    this->setNodeX(root);

    if (minPositionX < 0)
        this->offsetX(root);
}

void familyTree::setNodeX(familyTreeNode* ptr)
{
    if (ptr->children.empty())
        return;
    else if (ptr->children.size() == 1)
        // 一个孩子
        {
            ptr->children[0]->position.first = ptr->position.first + int(ptr->name.length()) / 2
                - int(ptr->children[0]->name.length()) / 2;
            if (minPositionX > ptr->children[0]->position.first)
            {
                minPositionX = ptr->children[0]->position.first;
            }
            setNodeX(ptr->children[0]);
            return;
        }
    // 获取其下一层结点的长度(从第一个名字的起始位置到最后一个名字的末尾位置)
    int childLength = ptr->childHorizontalLength
        +
        (-MAX(int(ptr->children[0]->childHorizontalLength), int(ptr->children[0]->name.length()))
        + int(ptr->children[0]->name.length())
        - MAX(ptr->children[ptr->children.size() - 1]->childHorizontalLength,
            int(ptr->children[ptr->children.size() - 1]->name.length()))
        + int(ptr->children[ptr->children.size() - 1]->name.length())) / 2;

    // 计算起始位置
    int startPosition = ptr->position.first + (-childLength
        + int(ptr->name.length())) / 2;

    if (minPositionX > startPosition)
    {
        minPositionX = startPosition;
    }
}

```

```

    }

    for (auto i = ptr->children.begin(); i != ptr->children.end(); ++i)
    {
        if (i != ptr->children.begin())
        {
            startPosition += (MAX((*i)->childHorizonallLength, int((*i)-
>name.length())) -
                int((*i)->name.length())) / 2 ;
        }

        (*i)->position.first = startPosition;
        // 处理该结点
        setNodeX(*i);
        // 开始位置
        startPosition += (MAX((*i)->childHorizonallLength, int((*i)->nam
e.length())) +
            int((*i)->name.length())) / 2 + 5;
    }
}

void familyTree::setNodePosition()
{
    this->setNodeX();
    this->setNodeY();
}

// 修正偏移量
void familyTree::offsetX(familyTreeNode* ptr)
{
    ptr->position.first += -minPositionX;
    for (auto i = ptr->children.begin(); i != ptr->children.end(); ++i)
        offsetX(*i);
    return;
}

```

4 项目测试

本项目以《红楼梦》贾府建立家谱树为例。

4.1 建立家谱测试

测试样例：贾源

实验结果：



```

D:\学习文件\数据结构\数据结构课设\6-家谱管理系统\Debug\6-家谱管理系统.exe

**                      家谱管理系统                      **
=====
**          请选择要执行的操作:          **
**          A --- 完善      家庭          **
**          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          **
**          F --- 切换树形显示模式          **
**          =====          **

首先建立一个家谱!
请输入祖先的姓名: 贾源
此家谱的祖先是: 贾源

请选择要执行的操作: A

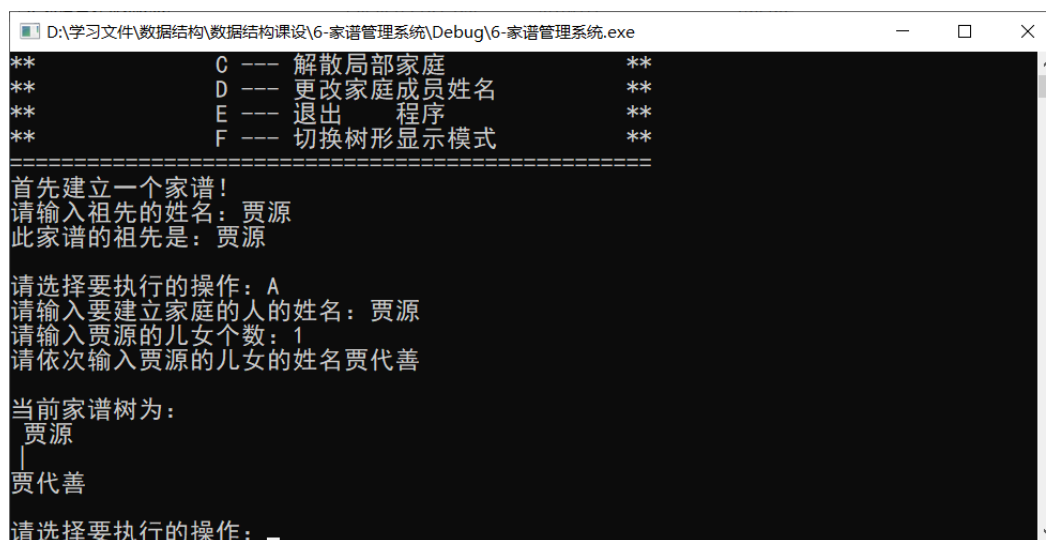
```

4.2 完善只有单个子女的家庭

测试样例：A 贾源 1 贾代善

预期输出：二代家谱树

实验结果：



```

D:\学习文件\数据结构\数据结构课设\6-家谱管理系统\Debug\6-家谱管理系统.exe

**          C --- 解散局部家庭          **
**          D --- 更改家庭成员姓名          **
**          E --- 退出      程序          **
**          F --- 切换树形显示模式          **
**          =====          **

首先建立一个家谱!
请输入祖先的姓名: 贾源
此家谱的祖先是: 贾源

请选择要执行的操作: A
请输入要建立家庭的人的姓名: 贾源
请输入贾源的儿女个数: 1
请依次输入贾源的儿女的姓名贾代善

当前家谱树为:
  贾源
  |
  贾代善

请选择要执行的操作:

```

4.3 完善有多个子女的家庭

测试样例：A 贾代善 3 贾赦 贾政 贾敏

预期输出：三代家谱树

实验结果：

```
D:\学习文件\数据结构\数据结构课设\6-家谱管理系统\Debug\6-家谱管理系统.exe
请依次输入贾源的儿女的姓名贾代善
当前家谱树为：
  贾源
  |
  贾代善
请选择要执行的操作：A
请输入要建立家庭的人的姓名：贾代善
请输入贾代善的儿女个数：3
请依次输入贾代善的儿女的姓名贾赦 贾政 贾敏
当前家谱树为：
      贾源
      |
      贾代善
    /  |  \
  贾赦 贾政 贾敏
请选择要执行的操作：_
```

4.4 多代家谱树测试

测试样例：执行操作 A 以建立完整家谱树

预期输出：贾府家谱树

实验结果：

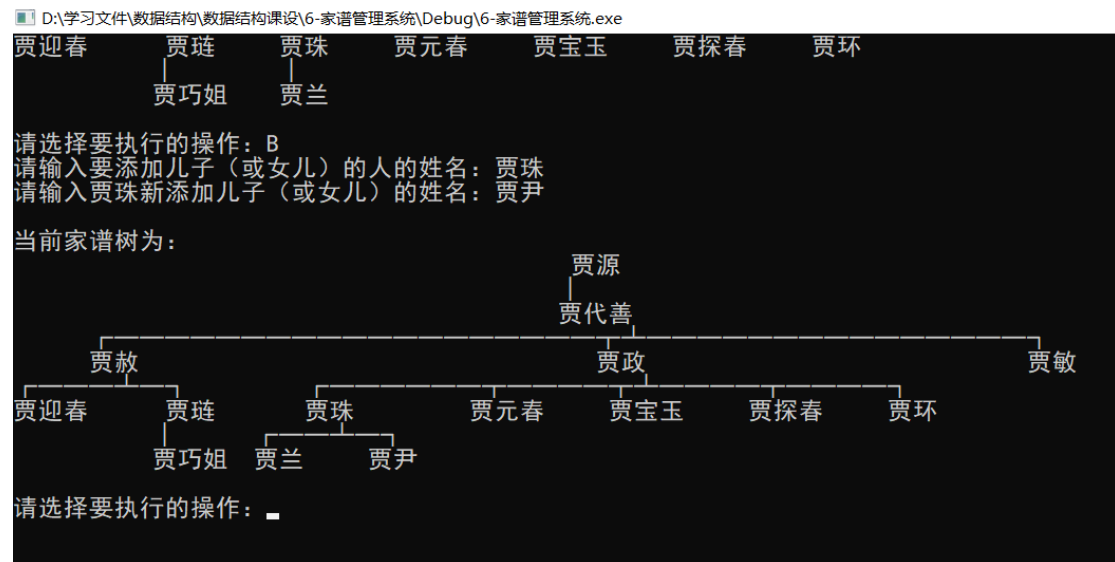
```
D:\学习文件\数据结构\数据结构课设\6-家谱管理系统\Debug\6-家谱管理系统.exe
      贾巧姐
      |
      贾源
      |
      贾代善
    /  |  \
  贾赦 贾政 贾敏
 /  |  \
贾迎春 贾琏 贾珠 贾元春 贾宝玉 贾探春 贾环
 |      |
贾巧姐 贾兰
请选择要执行的操作：_
```

4.5 添加家庭成员测试

测试用例：B 贾珠 贾尹

预期结果：增加了一个子女的家谱树

实验结果：

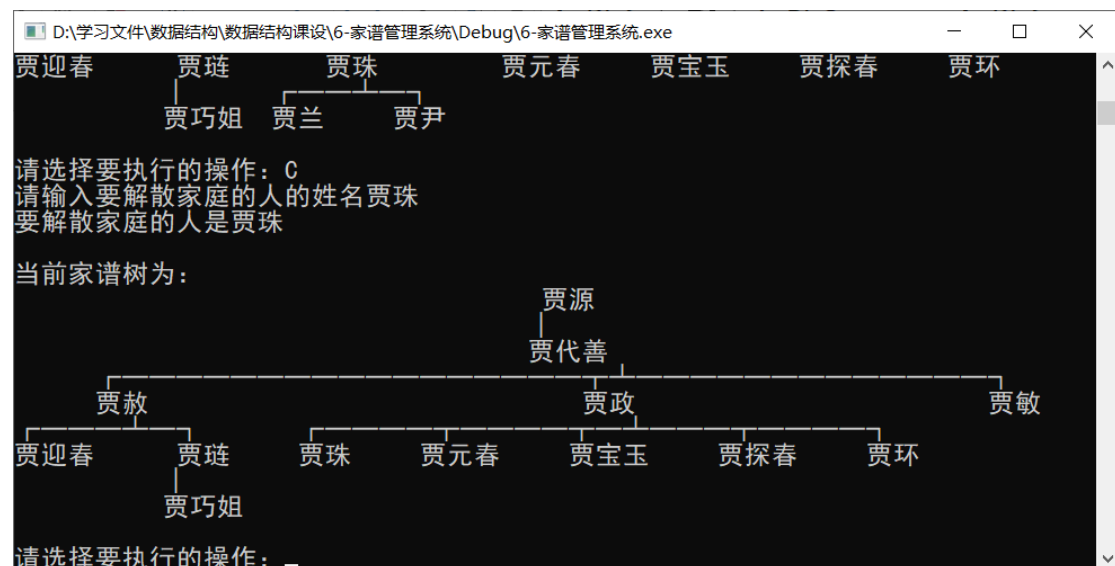


4.6 解散局部家庭测试

测试用例：C 贾珠

预期结果：被解散家庭后的家谱树

实验结果：

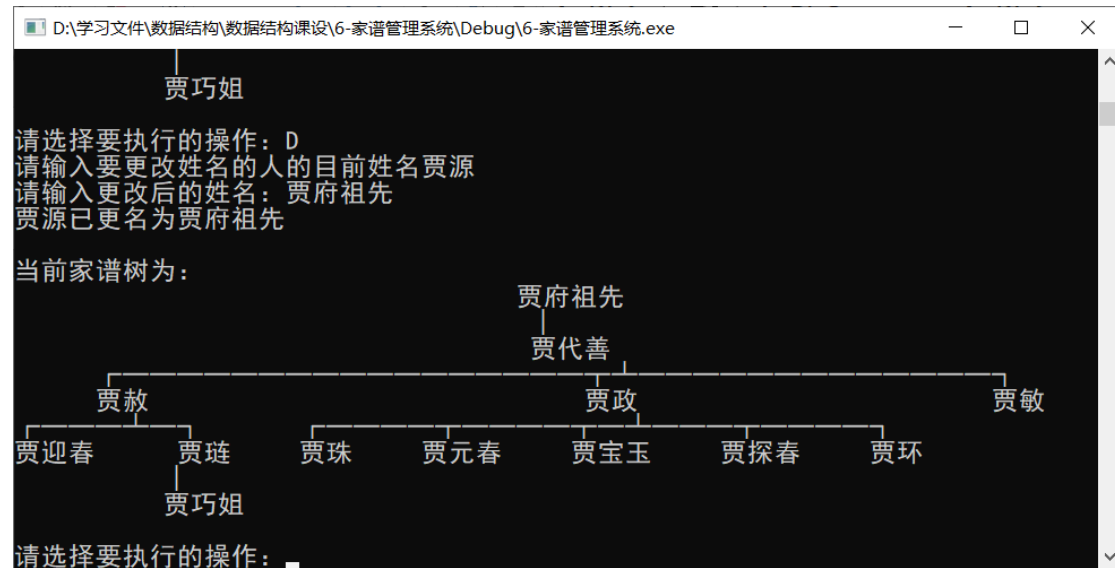


4.7 更改家庭成员姓名测试

测试用例：D 贾源 贾府祖先

预期结果：更改完姓名后的家谱图

实验结果：

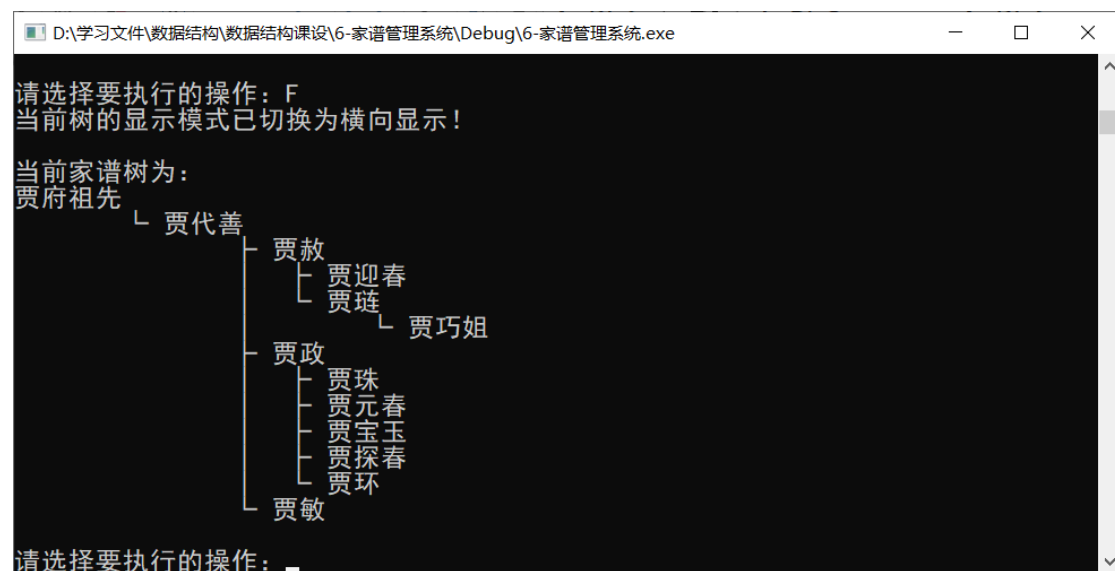


4.8 横版家谱树测试

测试用例：F

预期结果：横版绘制的家谱树

实验结果：



4.9 边界情况

4.9.1 多代单传家谱树测试

测试用例：每一代只有一个子女
预期输出：竖版和横版打印的家谱树
实验结果：

横向绘制	纵向绘制
<div>当前家谱树为： P0 └─ P1 └─ P2 └─ P3 └─ P4 └─ P5</div>	<div>当前家谱树为： P0 P1 P2 P3 P4 P5</div>

4.9.2 一代多子女家谱树测试

测试用例：一代中有多个子女
预期输出：竖版和横版打印的家谱树
实验结果：

横向绘制	纵向绘制
<div>当前家谱树为： P0 ├─ P1 ├─ P2 ├─ P3 ├─ P4 ├─ P5 ├─ P6 ├─ P7 ├─ P8 ├─ P9 └─ P10</div>	<div>当前家谱树为： P0 ├─ P1 ── P2 ── P3 ── P4 ── P5 ── P6 ── P7 ── P8 ── P9 ── P10</div>