1. 课题简介和设计要求：

该课题名称为王的传承，旨在对文本中家族成员之间的关系进行分析，确定其逻辑结构后，利用数据结构知识采用适合的数据结构，将家族成员信息以及他们之间的逻辑关系存储到一个族谱系统中，在该族谱系统中可以实现族谱关系图的展示、家族成员的添加、家族成员的修改、家族成员的删除、家族成员的多种查询以及保存家族成员信息到一个txt文件的功能。

1. 总体设计：

通过分析，该系统需要包含以下功能：

1. 录入功能：能够从txt文件中录入成员的基本信息，并存储到相应的数据结构中，为该系统提供初始的数据项。
2. 展示功能：能够根据录入的数据和数据结构保存的逻辑关系，生成成员关系图。
3. 添加功能：能够在初始化数据的基础上，添加新的成员，并录入新成员的基本信息，能够通过对逻辑关系的分析选择添加位置。
4. 修改功能：能够对该系统中所有的成员的基本信息进行修改，且修改过后，成员的基本信息以及逻辑结构仍然合法。
5. 删除功能：能够对系统中的成员以及该成员的后代进行一个删除操作。
6. 查询功能：能够对系统中成员的基本信息做一个查询处理，根据查询的方法不同又可以分为：名字查询、代查询、父亲查询、兄弟查询以及孩子查询。
7. 保存功能：能够将操作过后的成员数据保存到一个txt文件中。
8. 退出系统功能：完成操作后可以退出系统。流程图如下：

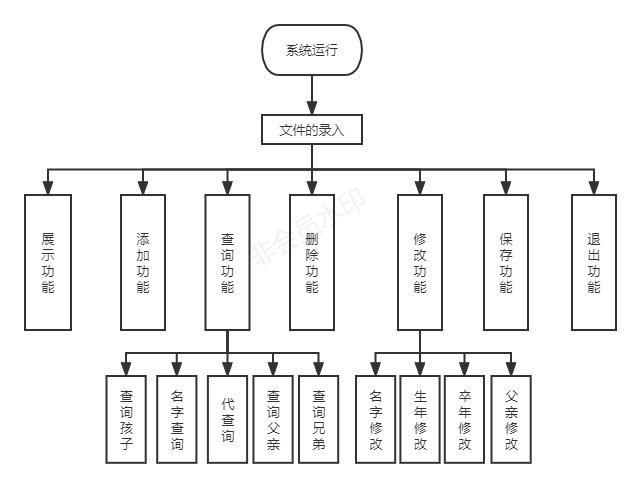


图 1：功能流程图

1. 详细设计：
   1. 节点类型的设计：

据分析，每一个数据由四个数据项组成，包括名字(Name)、生年(BornTime)、卒年（DiedTime）和父亲的名字(FatherName)。将其打包到一个类中People，同时为了展现各成员之间的关系，需要设计一个节点Node,包含数据项为下标No和一个Node指针next;然后将People和Node打包到一个类中即为DateNode。最后，为了存储这些数据，需要建立一个DateNode型数组，系统存储数据成员的最大容量MaxSize和当前容量Acsize，将其包含在类Genealogy中。节点表如下：

表 1：数据表(People)

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 数据项名称 | 数据类型 | 备注 |
| 1 | Name | ElemType | 名字 |
| 2 | BornTime | Years | 生年 |
| 3 | DiedTime | Years | 卒年 |
| 4 | FatherName | ElemType | 父亲的名字 |

表 2：节点表(Node)

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 数据项名称 | 数据类型 | 备注 |
| 1 | No | int | 下标编号 |
| 2 | next | Node\* | 指向下一个节点的指针 |

表 3：NodeDate

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 数据项名称 | 数据类型 | 备注 |
| 1 | date | People | 成员数据 |
| 2 | first | Node\* | 指向Node节点的指针 |

表 4：Genealogy

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 数据项名称 | 数据类型 | 备注 |
| 1 | num | NodeDate\* | 存储成员数据的数组指针 |
| 2 | MaxSize | int | 存储成员数据数组的最大容量 |
| 3 | Acsize | int | 存储成员数据数组的当前容量 |

3.2采用的逻辑结构：

根据分析，该系统中每一个成员可以有多个后代，满足一对多的关系，而树形结构则满足该要求，因此采用树形结构作为该系统的数据结构。

3.3采用的存储结构：

存储结构主要分为两种，一种是线性存储，一种是链式存储。根据分析，该系统采用的树形结构，需要对成员的信息进行一个存储，同时也要存储成员之间的关系，因此两种存储结构都采用。利用线性存储将每个成员的信息存储到一个线性表中，在存储每个成员的线性表元素后面存储一个指针，利用链式存储，存储该成员的后代的下标，从而存储成员之间的关系。类似于树的双亲-孩子表示法，只是线性表元素中存储的不是双亲的下标，而是双亲的名字。

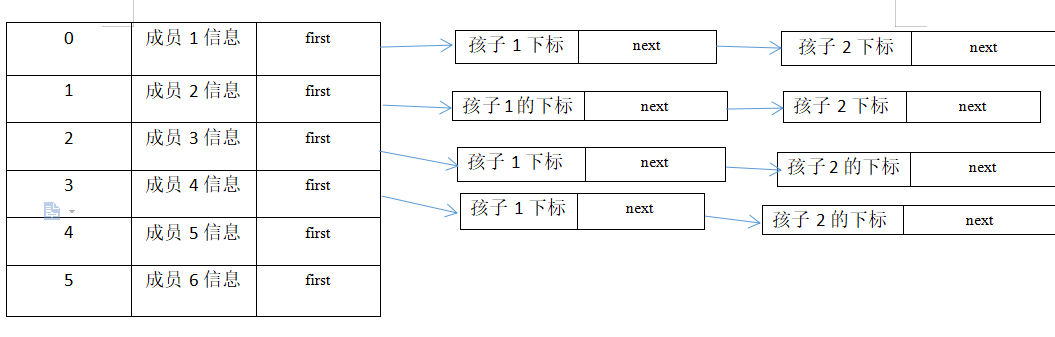


图 2:存储结构

3.4相关算法：

3.4.1 展示族谱算法：

族谱的展示可以用递归的方法实现，展示的难点在于如何让名字在easyX界面上展示不重叠，因此确定每个名字的坐标很重要。先在辅助函数外画出始祖的名字，然后调用辅助递归函数传入始祖的名字，在辅助函数中，先对传入名字的孩子做一个统计，然后将传入的高度除以孩子的数目，为每个孩子分到的高度空间，这样可以避免后代名字发生重叠的现象，影响族谱图的美观。如果孩子数目不为0，进入循环，从第一个孩子开始，先画出孩子的名字，然后将传入的x坐标+150，避免不同代之间的成员名字重叠，传入孩子的y坐标，传入孩子所分到的高度空间，然后传入孩子的名字，对孩子进行一个递归。直到所有的孩子递归结束，族谱图完成。流程图如下：

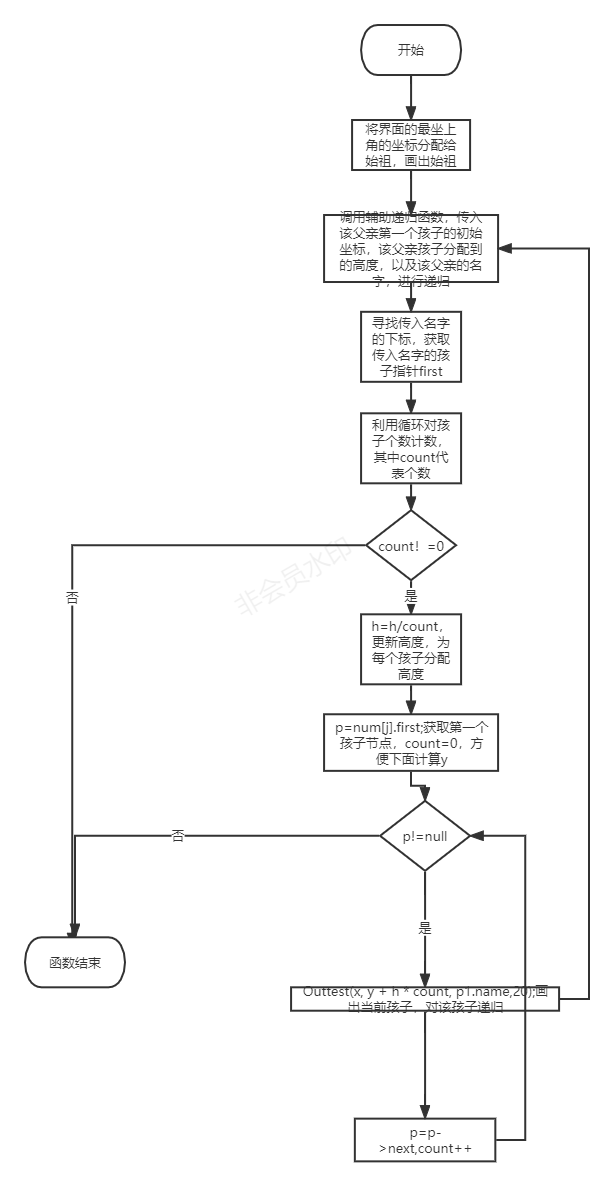


图 3：展示算法流程图

3.4.2添加成员算法：

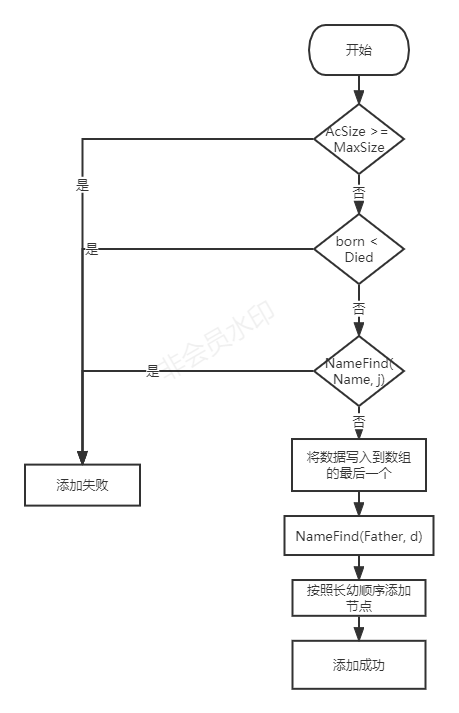
添加一个成员，先要判断该成员是否存不存在，以及数组的最大容量是否已经满了，因此先要根据添加成员的名字调用函数FindName，看看能不能找到，如果找不到，可以添加。添加时，先将该成员添加到数组的最后一个，然后根据添加元素中父亲的名字，调用FindFather函数找到父亲，然后在父亲的孩子节点中通过比较出生日期，按照长幼顺序进行插入。从而完成添加。流程图如下：

图 4：添加算法流程图

3.4.3 删除成员算法：

删除一个成员也是采用递归的算法，先通过传入的删除成员的名字，调用NameFind，在找到的情况下，根据其后面的孩子节点的下标，递归该函数，一个接一个，删除其后代，由于在删除过程中，该成员的位置可能会发生变化，因此需要再调用FindName函数，重新找到该成员的下标。等到该成员所有的后代删除完毕后，利用FindFather找到其父亲，删除该成员在其父亲那的节点，然后利用线性表中的最后一个元素，将其顶替，修改最后一个元素父亲那的孩子节点的下标，AcSize--，从而完成删除。流程图如下：

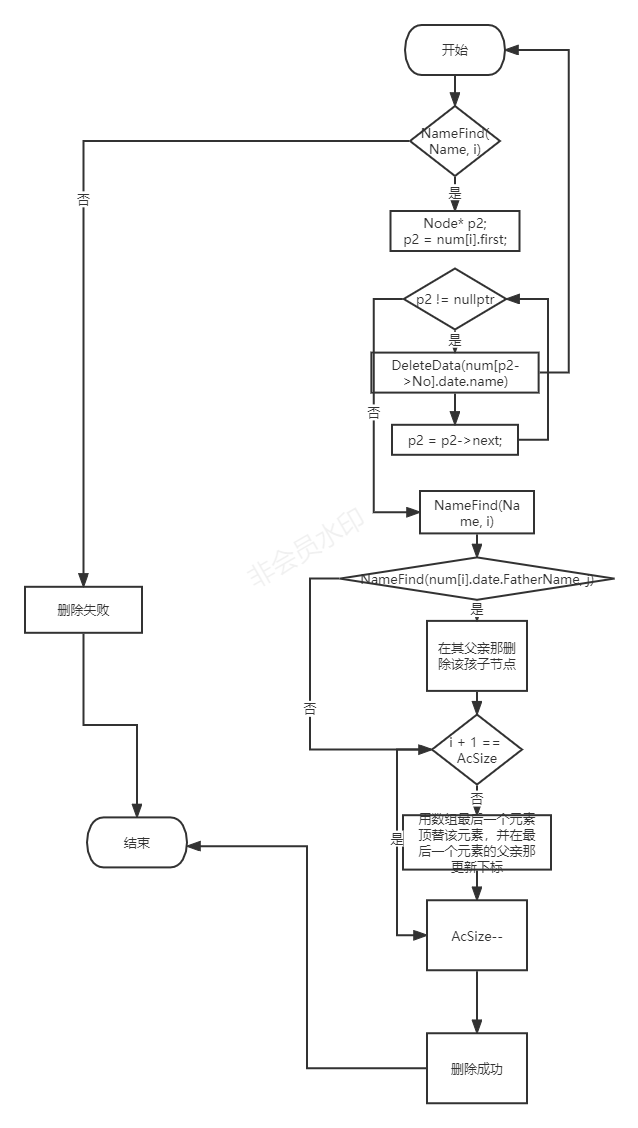


图 5：删除流程图

3.4.4 修改成员算法：

修改一个成员,先要用FindName是否能寻找到这个成员，如果能够找到，由于该系统将每个数据都设置为一个按钮，因此想修改哪一个就点击哪一个按钮，进行修改。其中修改生年，不仅需要满足修改后的生年小于卒年，而且在修改完之后，需要重新在其父亲那进行长幼排序。修改卒年需要满足修改后的卒年不能小于生年。修改父亲后，需要将该成员从原来的父亲那删除，添加到新的父亲的孩子队列中，如果修改后的父亲不存在，则表示修改失败。

3.4.5 查询成员算法：

查询父亲，查询名字采用的方法类似，都是对线性表进行遍历，返回与寻找名字匹配的成员，返回该成员的数据。查询孩子是在查询到其名字的基础上，用一个容器存储其孩子节点中的下标，返回该容器，然后调用GetDate函数查询到其所有孩子的值。查询兄弟类似，先查到其父亲，然后将其父亲除该人以外的所有的孩子下标存储到容器中返回，调用GetDate函数查询。代查询使用了BFS广度优先遍历的思想，用三个变量count，count2，count3，其中，count用来记录上一代成员的个数，count2用来记录当代成员的个数，而count3记录第几代。每次出队一个count就减一，每次入队一个，count2就加一，当count等于0时，说明上一代已经全部出队，将count2的值赋给count，然后count2归0，count3加一，当count3等于传入的要查询的代数g时，将当代的所有的成员的下标压入到容器中返回。然后调用GetDate函数查询。

4编码的设计：

4.1**数据结构的定义**

struct Node//双亲孩子表示法中的代表孩子的节点

{

int No;//孩子在线性表中的下标

Node\* next;

Node();

Node(int n, Node\* N=nullptr);

};

template<class ElemType, class Weight>

struct People//双亲孩子表示法中，线性表中的数据域

{

ElemType name;//名字

Weight BornTime;//出生时间

Weight DiedTime;//死亡时间

ElemType FatherName;//父亲的名字

People();

People(ElemType n, Weight b, Weight d, ElemType f=nullptr);

People &operator=(const People&b);

};

template<class ElemType,class Weight>

struct DateNode//孩子双亲表示法中，线性表元素成员

{

People<ElemType,Weight> date;

Node\* first;

DateNode();

DateNode(ElemType name, Weight b, Weight d, ElemType f);

};

template<class ElemType,class Weight>

class Genealogy

{protected:

DateNode<ElemType,Weight>\* num;

int MaxSize;

int AcSize;

vector<int>Count;//记录每一代的个数

4.2 功能函数设计

void Initialize();//完成树的双亲孩子表示法结构的初始化

void Display(void(\*visited)(const People<ElemType,Weight>&));//族谱成员名字的展示

vector<int>GetCount();//获得每一代的个数的数组

DateNode<ElemType,Weight>\* GetNum();//获取数据成员Num数组

int GetAcSize();//获取数据成员AcSize

People<ElemType, Weight>GetData(int j);//根据下标获取族谱成员的信息

vector<int> GenerationFind(int g);//根据输入的代数获取该代成员的下标

vector<int>ChildFind(ElemType Name);//根据传入的名字获取他的孩子的下标

bool FatherFind(ElemType Name, int& j);//根据传入的名字获取他父亲的下标j

bool InsertDataNode(ElemType Name, Weight born, Weight Died, ElemType Father);//插入一个数据成员

bool NameFind(ElemType Name,int &j);//根据名字，查找该成员的下标j

bool ChangeName(ElemType PreName, ElemType Name);//改名字

bool ChangeBornTime(ElemType Name, Weight AfBornTime);//改出生时间

bool ChangeDeidTime(ElemType Name, Weight AfDiedTime);//改死亡时间

bool ChangeFather(ElemType Name, ElemType AfFatherName);//改父亲名字

bool DeleteData(ElemType Name);//根据名字删除成员

vector<int>BroFind(ElemType Name);//根据名字寻找他的兄弟，并将其兄弟的下标位置返回

bool Save();//保存当前数据到txt文件中

4.3 主要函数设计：

4.3.1 按钮类的设计

class button

{

public:

int x;

int y;//(x,y)按键左上角的位置

int width;//按键宽度

int height;//按键高度

COLORREF color;//按键的颜色，颜色全部都是大写的英文单词

char\* str;//按键上的字

button(int x\_, int y\_, int width\_, int height\_, COLORREF color\_, const char\* str\_) //对按键进行初始化，这里也可以直接用构造函数

{

x = x\_;

y = y\_;

width = width\_;

height = height\_;

color = color\_;

str = new char[strlen(str\_) + 1];//字符指针需要预先分配内存

strcpy(str, str\_);//复制字符串

}

void drawbutton()//画出按键

{

setfillcolor(color);//设置填充的颜色为按键的颜色

settextstyle(20, 0, "楷体");//设置字体属性，宽度20，高度0（自适应），字体楷体

//setlinecolor(BLACK);//设置边框线条颜色

settextcolor(BLACK);//设置字体颜色

setbkmode(TRANSPARENT);//设置字体背景为透明treansparent

fillrectangle(x, y, x + width, y + height);//填充出一个矩形，参数为该矩形的左上和右下的坐标，颜色为上面所设置的颜色默认为黑色

int w = (width - (13 \* (lstrlen(str)-1))) / 2;

int h = (height - 20) / 2;

outtextxy(x+w, y+h, str);//在x+20,y+10位置输出设置好的字体文字str

}

~button() {//析构函数

delete[]str;

}

};

4.3.2 主要成员函数的实现

4.3.2.1族谱图展示实现：

template<class ElemType,class Weight>

void GenealogicGraph(Genealogy<ElemType,Weight>G, int x, int y, DateNode<ElemType,Weight>\* num,int h,ElemType Name)//辅助函数，x，y代表名字输出的

//起始坐标，num是数据成员数组，h是每一代分配到的高度，Name是要画的孩子的父亲

{

int j = -1;

G.NameFind(Name, j);

int count = 0;

Node\* p = num[j].first;

while (p != nullptr)

{

p = p->next;

count++;

}//先计算孩子的个数

if (count != 0)

{

h = h / count;//为每个孩子分配高度空间

if (h < 30)

{

h = 30;

}//保持美观

p = num[j].first;//从第一个孩子开始

People<string, int>p1;

if (count >=3)

{

y -= 60;

}

int d = count;

count = 0;

while (p != nullptr)

{

p1 = G.GetData(p->No);

Outtest(x, y + h \* count, p1.name,20);//写出当前孩子的名字

int j = 0;

if (d >= 3)

{

j = 60;

}

line(x - 90, y + 10+j, x, y + h \* count+10);//将父亲与当前孩子之间连线

GenealogicGraph(G, x +150, y + h \* count, num, h, p1.name);//调用递归函数，对当前孩子进行递归

p = p->next;

count++;//用来控制每一个孩子的起始纵坐标，避免孩子之间名字的重叠

}

}

}

template<class ElemType,class Weight>

void GenealogicGraph(Genealogy<ElemType,Weight> G)

{

DateNode<ElemType, Weight>\* num;

num = G.GetNum();

People<string, int>p1;

if (G.GetAcSize()!=0)//判断还有没有元素

{

p1 = G.GetData(0);

Outtest(150, 60, p1.name, 20);//画出始祖

GenealogicGraph(G, 300, 60, num, 825, p1.name);//调用递归画出其孩子

}

}

4.3.2.2添加成员实现：

template<class ElemType,class Weight>

bool Genealogy<ElemType, Weight>::InsertDataNode(ElemType Name, Weight born, Weight Died, ElemType Father)

{

if (AcSize >= MaxSize)//元素已满

{

return false;

}

int j = -1;

if (born < Died)

{

return false;

}

if (NameFind(Name, j))//插入的成员是否已经存在

{

return false;

}

else

{

num[AcSize].date.name = Name;

num[AcSize].date.BornTime = born;

num[AcSize].date.DiedTime = Died;

num[AcSize].date.FatherName = Father;

AcSize++;//在数据数组中为插入元素建立，当前元素数目+1

int d = -1;

if (NameFind(Father, d))//找到插入元素的父亲，建立一个父亲孩子之间关系的节点

//根据他的出生时间，按照长幼将该节点插入

{

Node\* P;

Node\* p = num[d].first;

Node\* pre = num[d].first;

while (p != NULL)

{

if (num[p->No].date.BornTime <= num[AcSize-1].date.BornTime)

{

pre = p;

p = p->next;

}

else

{

break;

}

}

if (pre == p)

{

P = new Node();

P->No = AcSize-1;

P->next = num[d].first;

num[d].first = P;

}

else

{

P = new Node();

P->No = AcSize-1;

P->next = pre->next;

pre->next = P;

}

return true;

}

return false;

}

}

4.3.2.3修改成员实现

4.3.2.3.1修改名字：

template<class ElemType,class Weight>

bool Genealogy<ElemType, Weight>::ChangeName(ElemType PreName, ElemType Name)

{

int j = -1;

if (NameFind(PreName, j))//判断能不能找到该人

{

num[j].date.name = Name;

return true;

}

return false;

4.3.2.3.2修改出生时间：

template<class ElemType,class Weight>

bool Genealogy<ElemType, Weight>::ChangeBornTime(ElemType Name, Weight AfBornTime)

{

int j = -1,i=-1;

if (AfBornTime > num[j].date.DiedTime)

{

return false;

}

if (NameFind(Name, j))

{

num[j].date.BornTime = AfBornTime;

if (FatherFind(Name, i))//修改长幼，由于修改了出生时间，可能会导致长幼顺序发生变化，

//从而需要重新改变长幼顺序

{

Node\* P;

Node\* p = num[i].first;

Node\* pre = num[i].first;

while (p->No != j)

{

pre = p;

p = p->next;

}

if (p == pre)

{

num[i].first = num[i].first->next;

}

else

{

pre->next = p->next;

}delete p;//先删除在其父亲那指向该元素的节点

p = num[i].first;

pre = num[i].first;

while (p != NULL)//新建一个节点，根据出生时间的大小，重新对节点进行插入

{

if (num[p->No].date.BornTime <= num[j].date.BornTime)

{

pre = p;

p = p->next;

}

else

{

break;

}

}

if (pre == p)

{

P = new Node();

P->No = j;

P->next = num[i].first;

num[i].first = P;

}

else

{

P = new Node();

P->No = j;

P->next = pre->next;

pre->next = P;

}

}

return true;

}

return false;

}

4.3.2.4删除成员实现：

template<class ElemType,class Weight>

bool Genealogy<ElemType, Weight>::DeleteData(ElemType Name)//节点删除模块

{

int i = -1, j = -1;//分别记录删除节点的下标和删除节点父亲的下标

if (NameFind(Name, i))//找到当前数据成员的下标

{

Node\* p2;

p2 = num[i].first;

while (p2 != nullptr)//如果有孩子，利用递归先将其每一个孩子和其后代删除

{

DeleteData(num[p2->No].date.name);

p2 = p2->next;

}num[i].first = nullptr;

NameFind(Name, i);//在孩子节点的删除过程中，当前元素节点的下标位置可能会因为

//删除而导致的下标位置的移动而移动，需要重新获取下标

if (p2 == nullptr)//当前成员已经没有了孩子，删除当前成员

{

if (NameFind(num[i].date.FatherName, j))//删除父亲与孩子之间的指针

{

if (j == 4)

{

int w = 1;

}

Node\* p, \* pre;

p = pre = num[j].first;

while (p->No != i && p != nullptr)

{

pre = p;

p = p->next;

}

if (p == num[j].first)

{

num[j].first = num[j].first->next;

}

else

{

pre->next = p->next;

delete p;

}

}

if (i + 1 == AcSize)//如果删除孩子是最后一个，直接将当前容量减一来实现删除

{

AcSize--;

}

else//如果不是，将最后一个节点移动到当前删除节点处，将容量减一

{

num[i].date = num[AcSize - 1].date;

num[i].first = num[AcSize - 1].first;

int k = -1;

if (NameFind(num[AcSize - 1].date.FatherName, k))

{

Node\* p;

p = num[k].first;

while (p->No != AcSize - 1 && p != nullptr)

{

p = p->next;

}

p->No = i;

}

AcSize--;

}

}

return true;

}

return false;

}

4.3.2.5查找函数实现

4.3.2.5.1代查找：

template<class ElemType,class Weight>

vector<int> Genealogy<ElemType, Weight>::GenerationFind(int g)//返回第g代的成员下标

{

vector<int>gen;

LinkQueue<int>link;//创建一个队列

link.EnQueue(0);

int count = 1;

int count2 = 0;//count,和count2相互配合来判断当前代是否已经遍历完成

if (g == 0&&AcSize!=0)

{

gen.push\_back(0);

}

int count3 = 1;

while (!link.IsEmpty())//利用BFS广度优先算法的思想

{

int i;

link.DeQueue(i);

Node\* p = num[i].first;

while (p != nullptr)

{

if (count3 == g)//当记录的代数和传入的g相等时，将这一代元素下标压入gen中

{

gen.push\_back(p->No);

}

link.EnQueue(p->No);

p = p->next;

count2++;

}

count--;

if (count == 0)

{

count3++;//记录第几代

count = count2;

count2 = 0;

}

}

return gen;

}

4.3.2.6主界面的设计：

void First()//最开始的页面展示，由于每个模块由两个页面组成，第二个页面按回退键返回初始页面，

//于是将初始页面封装为一个函数

{

initgraph(1880, 830, SHOWCONSOLE); // 创建绘图窗口，大小为 1880\*830像素

setbkcolor(WHITE);//设置背景板颜色

setbkmode(TRANSPARENT);//设置背景模式为透明，文字输出时不会破坏背景

cleardevice();

TCHAR s[50] = "显示族谱";

TCHAR s1[50] = "添加成员";

TCHAR s2[50] = "修改成员";

TCHAR s3[50] = "删除成员";

TCHAR s4[50] = "查询";

TCHAR s5[50] = "保存";

TCHAR s6[50] = "退出";

button b0(650, 160, 150, 50, RGB(170, 172, 184), s);

button b1(650, 230, 150, 50, RGB(164, 168, 180), s1);

button b2(650, 300, 150, 50, RGB(180, 179, 186), s2);

button b3(650, 370, 150, 50, RGB(160, 162, 174), s3);

button b4(650, 440, 150, 50, RGB(209, 198, 194), s4);

button b5(650, 510, 150, 50, RGB(131, 129, 129), s5);

button b6(650, 580, 150, 50, RGB(170, 161, 151), s6);//设置主页面按钮

IMAGE background;

loadimage(&background, "Khan.jpg", 1880, 830, 1);//为当前背景设置一个图片

putimage(0, 0, &background);

settextcolor(RGB(247, 171, 0));

settextstyle(100, 0, "STXINWEI");

outtextxy(550, 50, "王的家族");

b0.drawbutton();

b1.drawbutton();

b2.drawbutton();

b3.drawbutton();

b4.drawbutton();

b5.drawbutton();

b6.drawbutton();//画出按钮

}

5测试

5.1测试用例

程序开始时，系统自动将GenghisKhan.txt中的内容读取到系统中，并存储，进入主菜单，主菜单中显示显示族谱、添加成员、修改成员、删除成员、查询、保存、退出7个按钮。

初始化测试如表5

表 5：初始化测试用例

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 系统初始化功能测试 |
| 输出标准 | 系统将GenghisKhan.txt中的内容读入到系统中，并存储，进入easyX主菜单界面,主菜单中显示显示族谱、添加成员、修改成员、删除成员、查询、保存、退出7个按钮。 |

显示族谱测试

表 6：显示族谱测试

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 显示族谱测试 |
| 输入标准 | 点击主菜单显示族谱按键 |
| 输出标准 | 跳转到族谱显示界面，按照树状图横向显示族谱图。  点击后退按钮，回到主界面。 |

添加成员测试用例

表 7：添加成员测试

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 添加成员测试 |
| 输入标准 | 点击主菜单添加按键，输入添加成员的个数，添加成员的名字，生年，卒年，父亲的名字 |
| 输出标准 | 跳转到添加界面，如果添加的成员已经存在，或者生年大于卒年或者系统中没有其父亲的名字，则输出添加失败，否则输出添加成功  点击后退按钮，回到主界面。 |

修改成员测试用例

表 8：修改成员测试

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 修改成员测试 |
| 输入标准 | 点击主菜单添加按键，输入要修改的成员名字 |
| 输出标准 | 跳转到修改界面，如果能够查询到该成员信息，会将成员信息生成为四个按键，点击要修改的信息按键。   1. 修改名字 2. 修改生年，如果修改后的生年大于卒年，点击确认修改后，弹出修改失败 3. 修改卒年，如果修改后的卒年小于生年，点击确认修改后，弹出修改失败 4. 修改父亲，如果修改后的父亲不存在，点击确认修改后，弹出修改失败 5. 点击确认修改后，展示修改后成员信息，显示修改成功 6. 点击后退按钮，回到主界面。 |

删除成员测试用例

表 9：删除成员测试

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 删除成员测试 |
| 输入标准 | 点击主菜单添加按键，输入要删除成员的个数，输入要删除成员的名字 |
| 输出标准 | 跳转到删除界面，如果删除的成员不存在，退出本次删除，展示原因。否则展示要删除的成员信息，并弹出确认删除按钮，点击后删除成功，待所有的删除操作做完或者所有的成员都已经删除时，弹出全部删除已完成。  点击后退按钮，回到主界面。 |

查询成员测试

表 10：查询成员测试

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 查询成员测试 |
| 输入标准 | 点击主菜单查询按键   1. 点击“名字查询”，输入要查询的名字 2. 点击“按代查询”，输入要查询的代数 3. 点击“查询兄弟“，输入查询人 4. 点击“查询父亲”，输入查询人 5. 点击”查询孩子“，输入查询人 6. 点击后退按钮 |
| 输出标准 | 1. 输出查询名字的人的所有信息 2. 依次输出这一代人的所有信息 3. 依次输出所有兄弟的信息 4. 输出父亲的信息 5. 依次输出所有孩子的信息 6. 回到主界面 |

保存测试

表 11：保存测试

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 保存测试 |
| 输入标准 | 点击主菜单保存按键 |
| 输出标准 | 跳转到保存界面，如果保存成功，弹出保存成功，如果保存失败，弹出保存失败。点击后退按钮，回到主界面。 |

退出测试

表 12：退出测试

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| 测试项 | 退出测试 |
| 输入标准 | 点击主菜单退出按键 |
| 输出标准 | 退出系统，关闭控制台 |

5.2程序运行结果

根据功能分析，利用easyX设计主页面，一共分为显示族谱、添加成员、修改成员、删除成员、查询、保存、退出7个模块，每个模块设置成一个按键，除退出模块外，其他所有的模块内部都包含一个界面，可以通过界面内的后退按钮来实现主界面和模块界面的切换。程序运行时，加载GenghisKhan.txt中的数据，将输入存储到设计好的双亲-孩子存储结构中

如图6



图 6：开始界面

点击显示族谱，横向展示族谱树状图，如图7

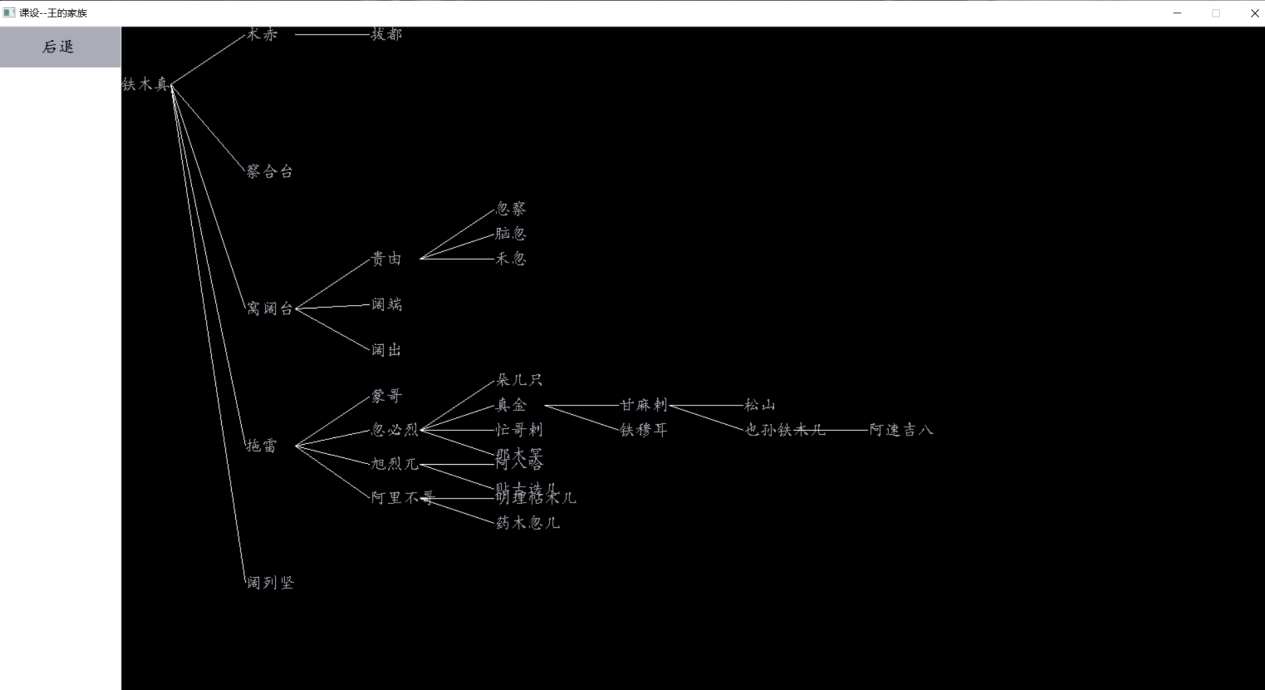


图 7：显示族谱界面

点击后退键，又回到主界面。

点击添加成员，在控制台输入要添加成员的个数，输入添加成员的信息，如图8



图 8：添加成员输入界面

如果添加的成员已经存在，那么添加失败如图9



图 9：添加失败

如果输入信息都正确，则添加成功如图10

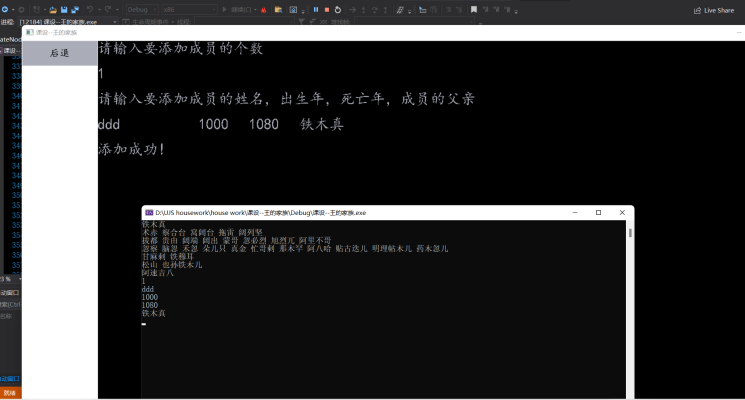


图 10：添加成功

点击修改成员，进入修改成员界面，输入要修改的成员的个数，输入成员名字，如果查询不到会输出未找到该人，如果能够查询到，会以按键的形式展示成员信息，点击要修改的信息，进行修改，如果完成全部修改，点击旁边的完成修改，会展示修改完后的成员信息。注意，当修改生年，卒年，父亲时要注意合法性，否则会修改失败。等待所有修改已经完成。修改成功如图11，图12，图13



图 11：修改成功界面



图 12：完成修改界面

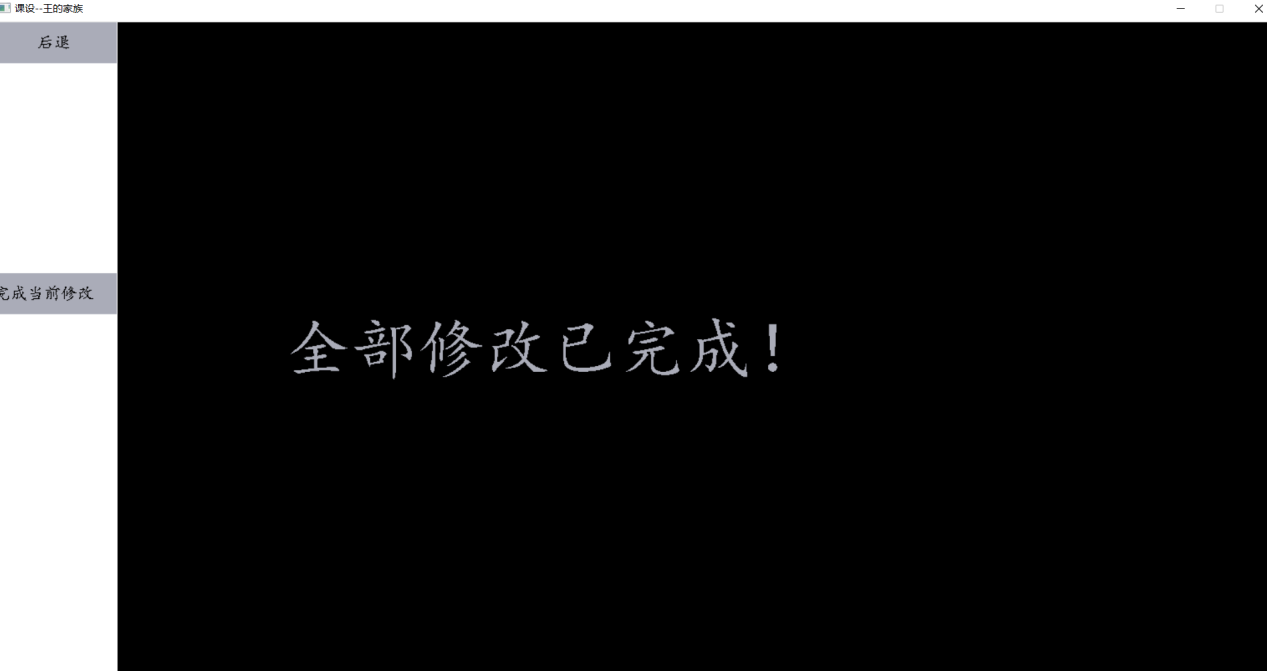


图 13：全部修改已经完成

点击删除成员，进入删除界面，输入要删除的个数，然后依次输入要删除的成员名单，如果要删除人在系统中没有，会提示找不到此人，如果有，会展示该人的信息，点击确认删除后，完成删除，当所有人都删除了或者删除次数已经完成，会弹出全部删除操作已经完成。如图14，15



图 14：删除成功界面

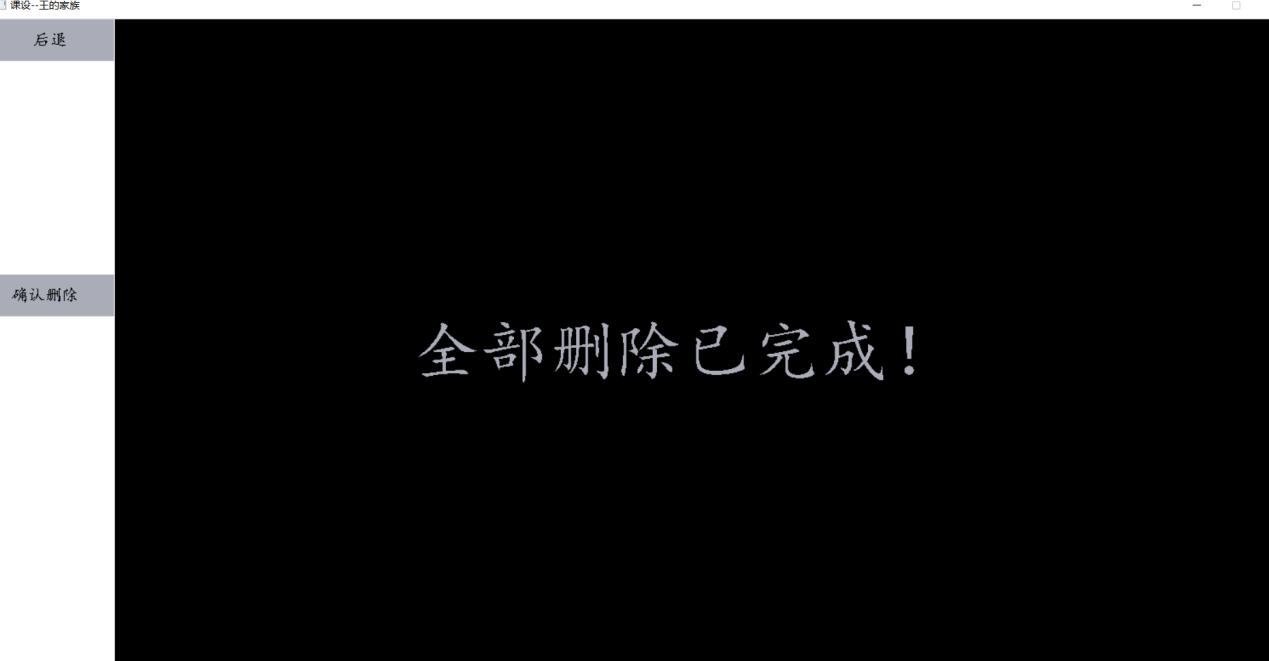


图 15：全部删除成功界面

点击查询，进入查询界面，如图16

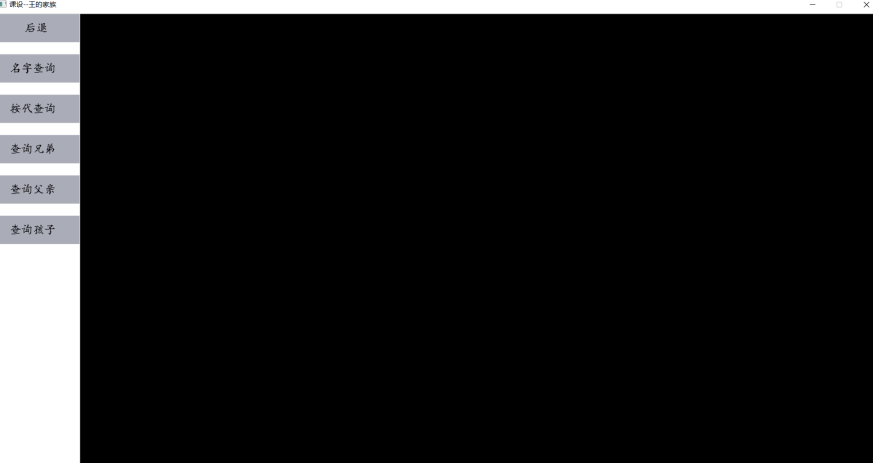


图 16：查询界面

点击名字查询，输入名字后，如果能查询到，会显示查询人的信息，查询成功，如果不能，查询失败。如图17



图 17：名字查询

查询代，输入查询的代数，会显示当前代数的所有成员信息。如图18

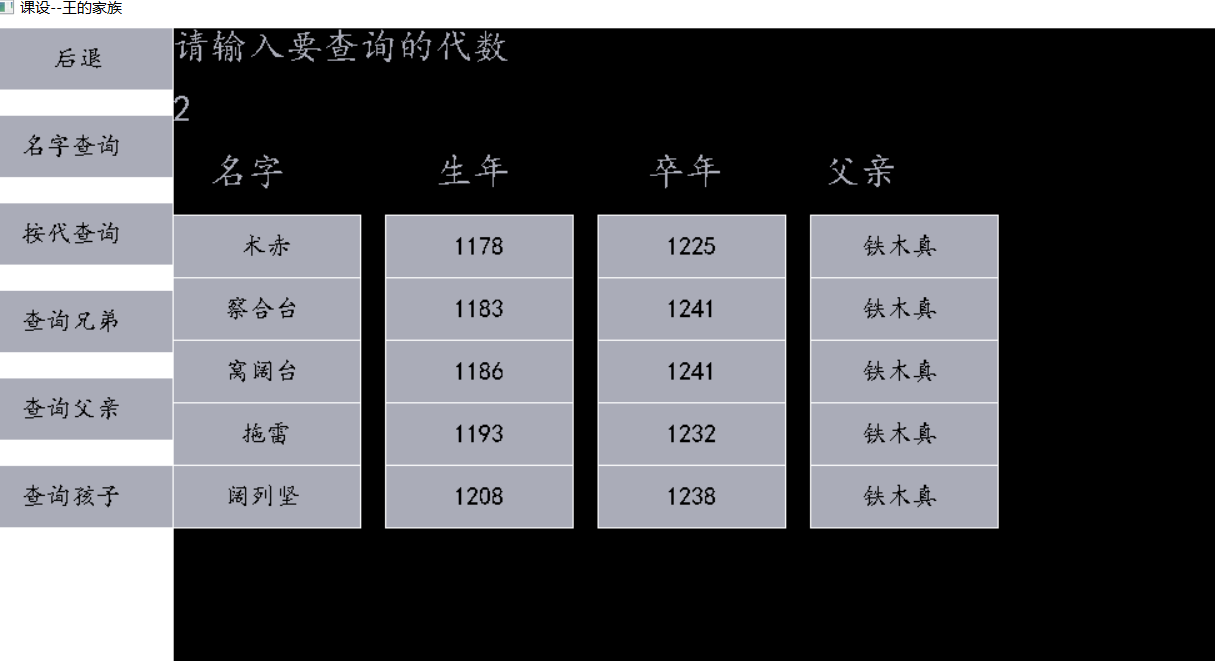


图 18：代查询

查询兄弟，查询孩子和查询代差不多，分别输入查询人的名字，输出其所有的兄弟，孩子的信息。查询父亲与名字查询类似，输入查询人的名字，显示其父亲的信息。

点击保存按键，进入保存界面，如果保存成功，输出保存成功，信息会被保存到文件中如图19，20



图 19：保存成功界面

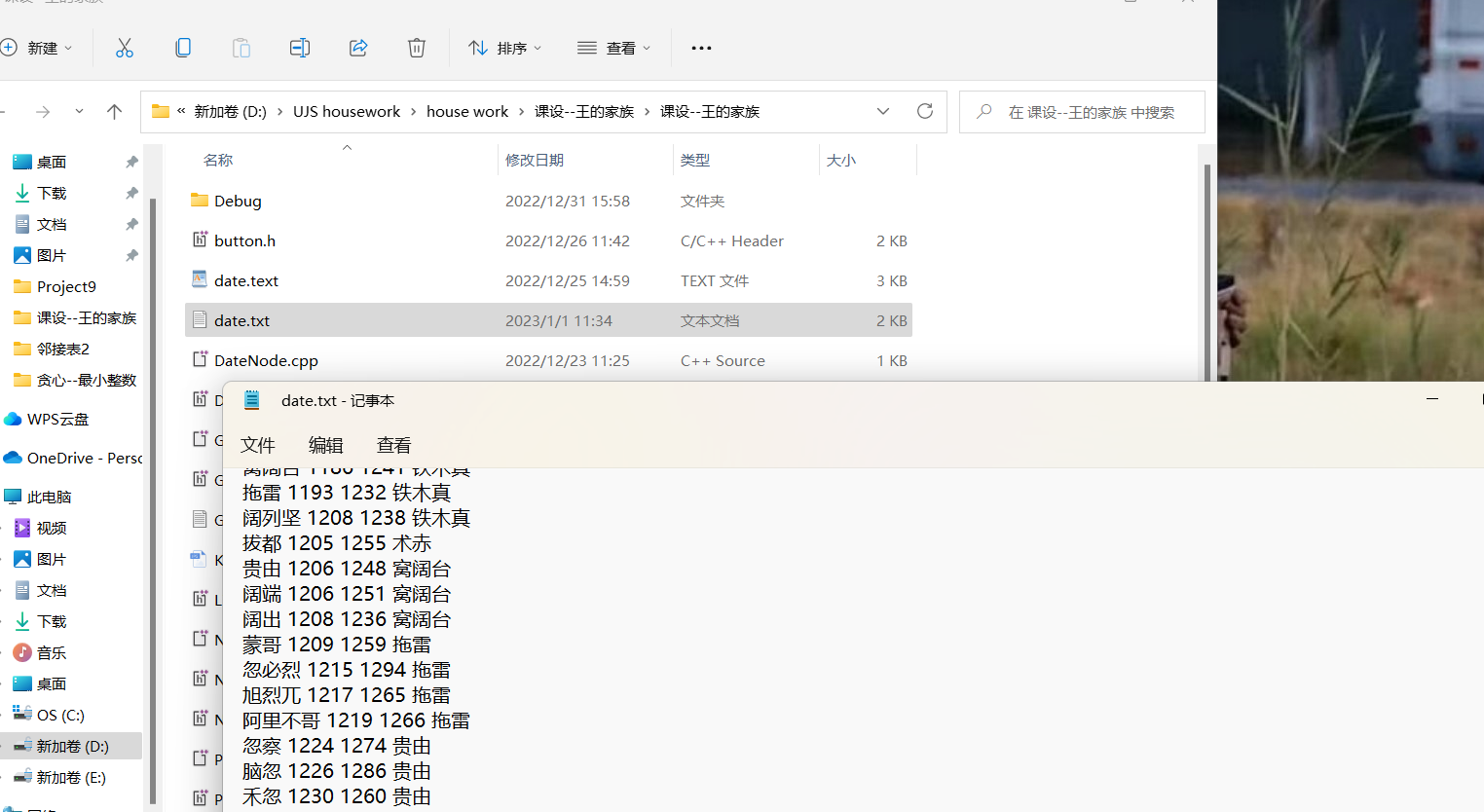


图 20：保存到文件中

点击后退按键，关闭控制台，如图21

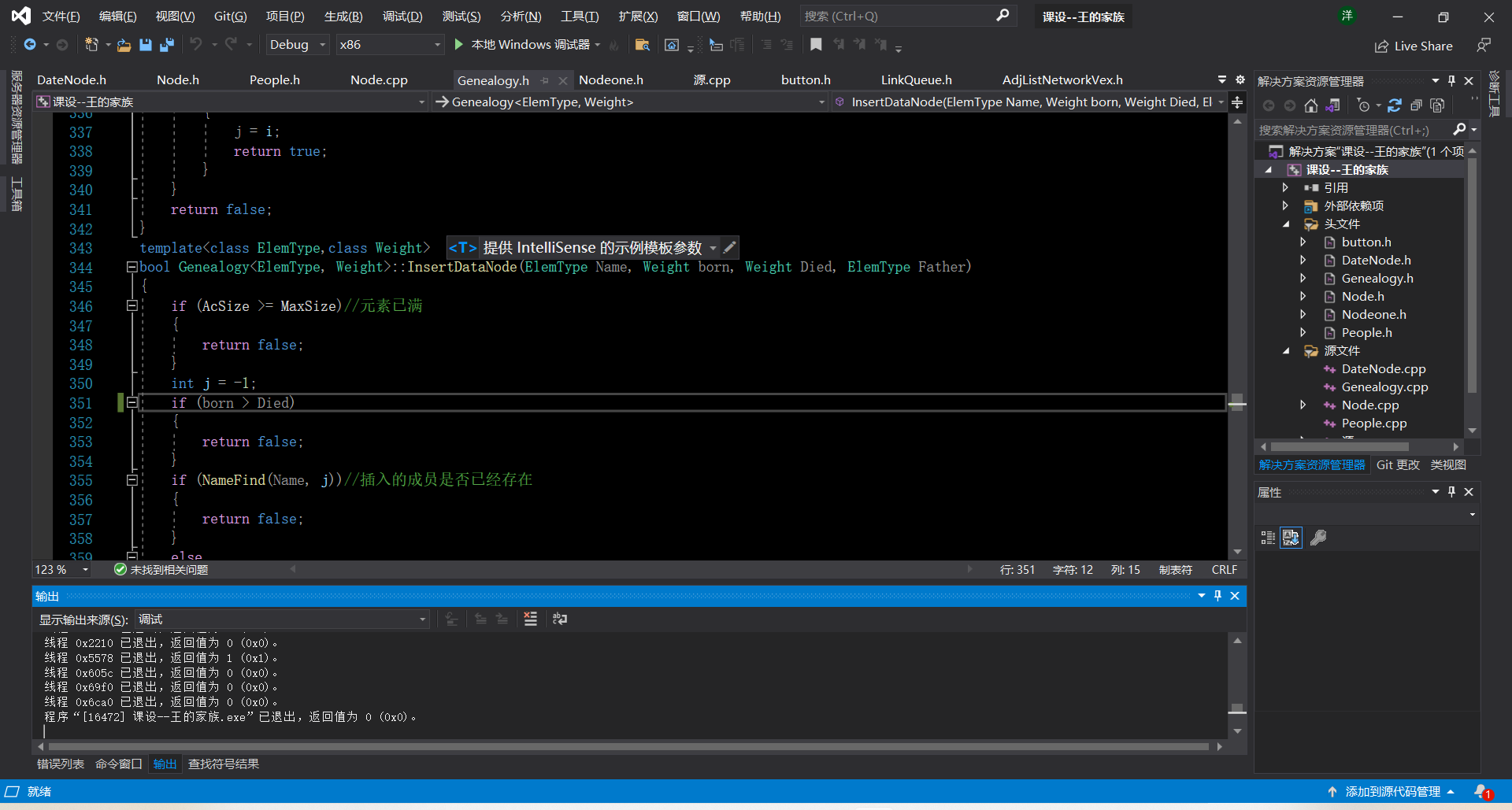


图 21：关闭控制台

7 参考文献

1. 缪淮扣 沈俊 顾训穰 编著 数据结构---c++实现（第二版）
2. http://docs.easyx.cn/zh-cn/intro