目 录

1 基于链式存储结构的线性表实现 2

1.1 问题描述 2

1.1.1 菜单演示功能 2

1.1.2 链表基本功能 2

1.1.3 多个线性表的管理 2

1.1.4 扩展应用 2

1.2 系统设计 3

1.3 系统实现 10

1.4 实验小结 17

2 基于二叉链表的二叉树实现 19

2.1 问题描述 19

2.1.1 菜单演示功能 19

2.1.2 二叉树基本功能 19

2.1.3 多个二叉树管理 19

2.1.4 扩展应用 19

2.2 系统设计 20

2.3 系统实现 29

2.4 实验小结 37

参考文献 39

附录 A 基于链式存储结构线性表实现的源程序 41

附录 B 基于二叉链表二叉树实现的源程序 56

1 基于链式存储结构的线性表实现

1.1 问题描述

本次实验在 Windows10 x64 位操作系统下，在 Dev-C++ 6.0 的 IDE 中开发，来实现一个具有菜单演示功能的系统，演示系统实现程序运行参数的键盘输入，实现多个线性表的管理，每个线性表要实现初始化，删除元素，插入元素等等基本功能，并在此基础上实现一个扩展应用：“两个一元系数多项式的加减计算”。多项式的输入为，先输入项数，再根据项数分别输入每一项的系数和指数。输出的多项式不含有可合并的同类项，并按照次数从小到大输出，如果多项式为 0，则输出 0。

1.1.1 菜单演示功能

菜单演示系统以用户和计算机的对话方式执行，即在计算机终端上显示“提示信息”，由用户在键盘上输入演示系统中规定的功能序号，再由用户输入要实现当前功能所用的数据，来实现该功能。输入后若有相应的结果则显示在其后。

1.1.2 链表基本功能

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的如下功能：1）初始化表；2）销毁表；3）清空表；4）判定空表；5）求表长；6）获得元素；7）查找元素；8）获得前驱；9）获得后继；10）插入元素；11）删除元素；12）遍历表。

1.1.3 多个线性表的管理

本系统需要实现多个线性表的管理，采用数组的方式实现，数组中的元素是每个线性表的头结点指针，具体实现见系统设计部分。

1.1.4 扩展应用

本系统将利用线性表实现多项式，利用多个线性表的管理，实现“两个一元系数多项式的加减计算”，完成实验任务。

1.2 系统设计

为实现上述程序功能，应先实现单链表的各项功能，然后以单链表表示多项式，再进行多个链表管理来进行多项式加减。

1. 单链表的抽象数据类型定义为：

ADT LinkedList{

数据对象：D={a_i|a_i∈CharSet,i=1,2,…,n,n≥0}

数据关系：R1={<a_(i-1),a_i>|a_(i-1),a_i∈D,a_(i-1)<a_i,i=1,2,…,n}

基本操作：

⑴初始化表：函数名称是 InitaList(L)；初始条件是线性表 L 不存在已存在；操作结果是构造一个空的线性表。

⑵销毁表：函数名称是 DestroyList(L)；初始条件是线性表 L 已存在；操作结果是销毁线性表 L。

⑶清空表：函数名称是 ClearList(L)；初始条件是线性表 L 已存在；操作结果是将 L 重置为空表。

⑷判定空表：函数名称是 ListEmpty(L)；初始条件是线性表 L 已存在；操作结果是若 L 为空表则返回 TRUE,否则返回 FALSE。

⑸求表长：函数名称是 ListLength(L)；初始条件是线性表已存在；操作结果是返回 L 中数据元素的个数。

⑹获得元素：函数名称是 GetElem(L,i,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用 e 返回 L 中第 i 个数据元素的值。

⑺查找元素：函数名称是 LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回 L 中第 1 个与 e 满足关系 compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为 0。

⑻获得前驱：函数名称是 PriorElem(L,cur_e,pre_e)；初始条件是线性表 L 已存在；操作结果是若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，否则操作失败，pre_e 无定义。

⑼获得后继：函数名称是 NextElem(L,cur_e,next_e)；初始条件是线性表 L 已存在；操作结果是若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e

返回它的后继，否则操作失败，next_e 无定义。

⑽插入元素：函数名称是 ListInsert(L,i,e)；初始条件是线性表 L 已存在且非空，1≤i≤ListLength(L)+1；操作结果是在 L 的第 i 个位置之前插入新的数据元素 e。

⑾删除元素：函数名称是 ListDelete(L,i,e)；初始条件是线性表 L 已存在且非空，1≤i≤ListLength(L)；操作结果：删除 L 的第 i 个数据元素，用 e 返回其值。

⑿遍历表：函数名称是 ListTraverse(L,visit())，初始条件是线性表 L 已存在；操作结果是依次对 L 的每个数据元素调用函数 visit()。

}ADT LinkedList

2. 多项式的基本操作定义为：

⑴多项式排序：函数名称是 PolySort(LinkList L)；初始条件是当线性表 L 已存在；操作结果是将该线性表中的多项式按幂次递减排序。

⑵创建多项式：函数名称是 CreateList(int listnum)；初始条件是序号 listnum 指向的链表存在；操作结果是将链表 listnum 号根据输入情况创建出多项式。

⑶输出多项式：函数名称是 ShowList(int listnum)；初始条件是序号 listnum 指向的链表存在；操作结果是输出 listnum 号多项式。

⑷多项式相加：函数名称是 AddList(int a,int b,int c)；初始条件是 a 和 b 号多项式存在，c 号多项式初始化完成；操作结果是将多项式 a 和 b 相加结果存入 c 中。

⑸多项式相减：函数名称是 MinusList(int a,int b,int c)；初始条件是 a 和 b 号多项式存在，c 号多项式初始化完成；操作结果是将多项式 a 和 b 相减结果存入 c 中。

3. 本程序包含三个模块：

1）主程序模块：

```
int main(){
初始化;
显示菜单;
```

```
while(op){
接收命令;
处理命令
}
return 0;
}
```

2）单链表模块——实现单链表的抽象数据类型

3）多项式模块——实现多项式的各项基本功能

本程序需要实现多个链表的管理：

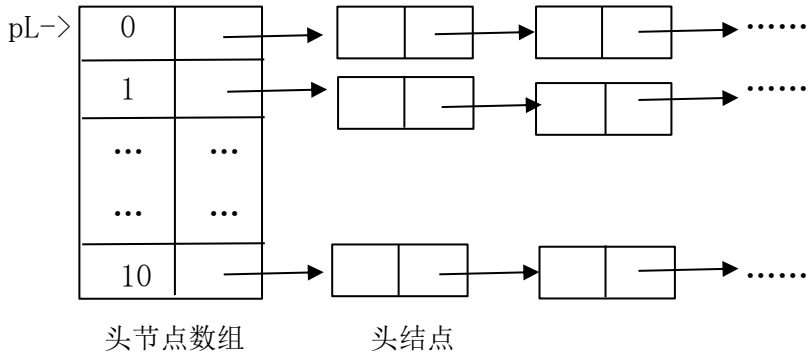


图 1.2-1 多链表管理

各模块之间的调用关系如下：

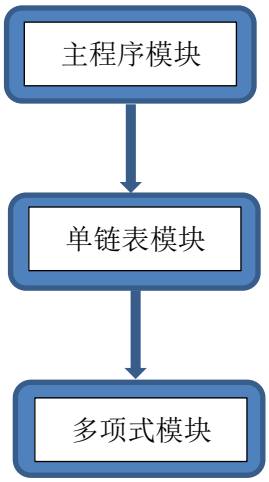


图 1.2-2 各模块调用关系

单链表模块具体如下：

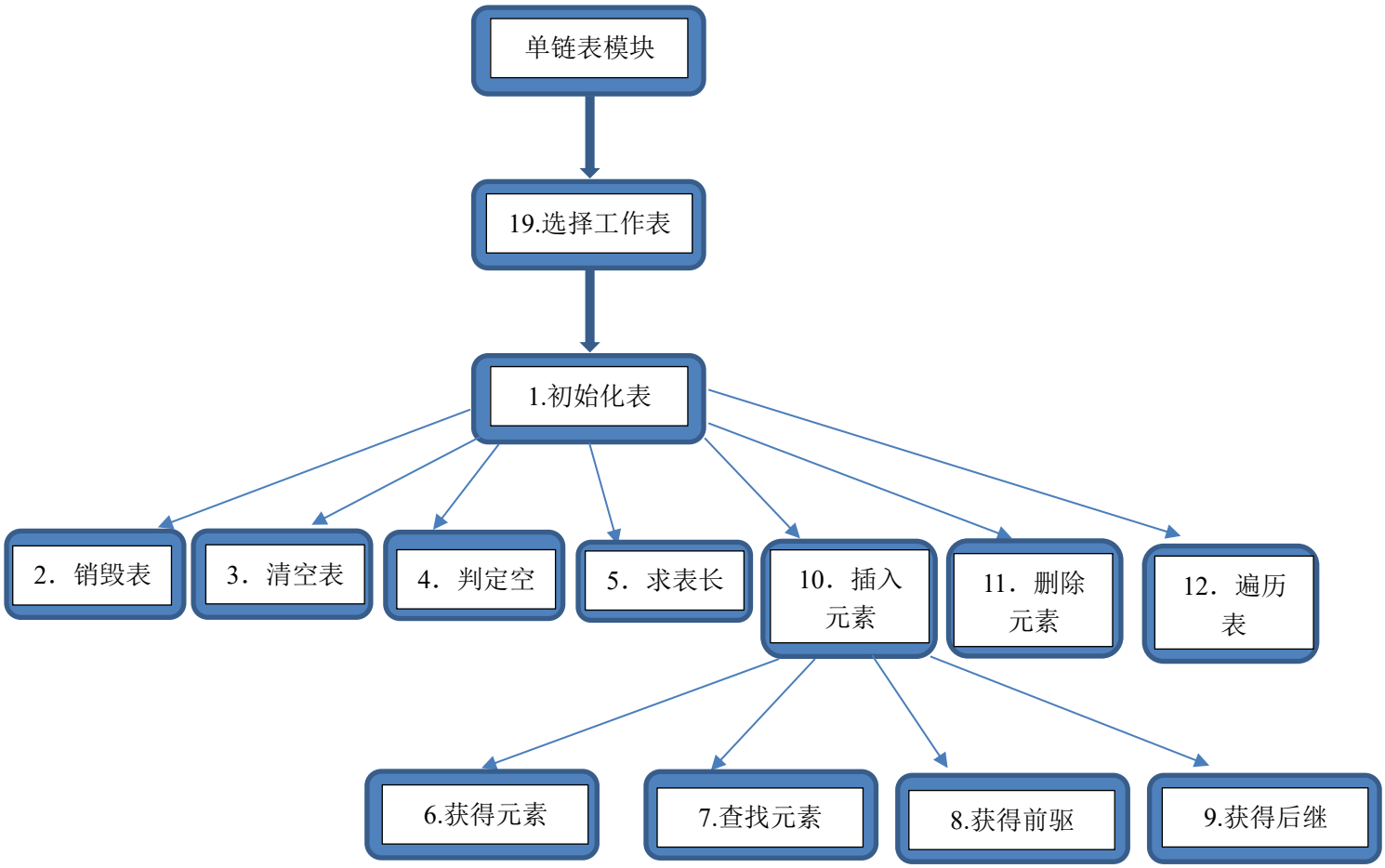
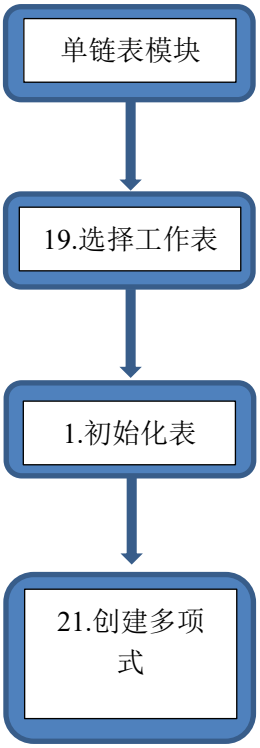


图 1.2-3 单链表模块



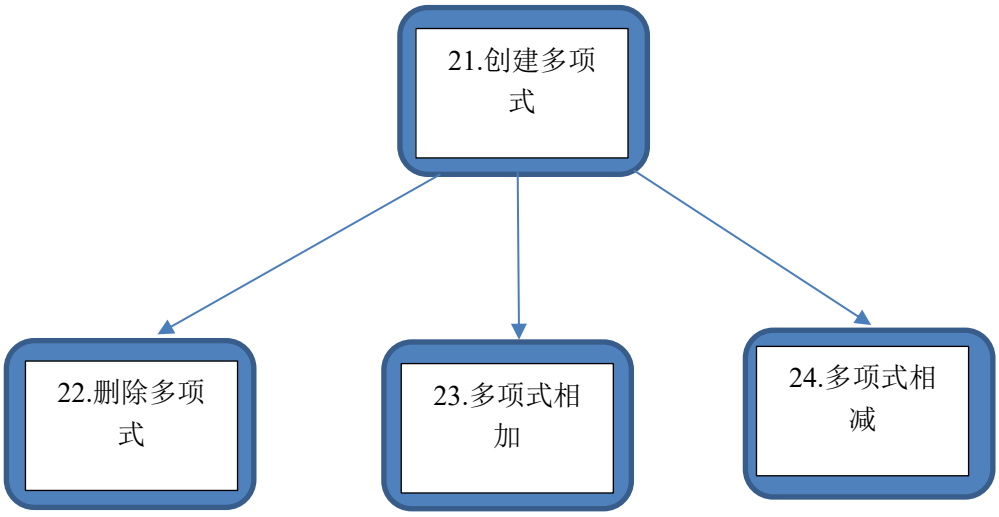


图 1.2-4 多项式模块

4. 本程序的主要算法：

1）InitaList(LinkList& L)：先为 L 开辟一块空间，如果开辟失败，则 exit(OVERFLOW)，开辟成功则将 next 域置空。

2）DestroyList(LinkList& L)：利用指针 p 分别指向 L 的每一个节点，然后分别删除其每一个节点，最后将头指针 L 置空。

3）ClearList(LinkList L)：先将 L 的 next 域置空，然后利用指针 p 和 q 分别释放除头结点外的每一个节点。

4）ListEmpty(LinkList L)：判断 L->next 域，若为空，返回 TRUE，若不空，返回 FALSE。

5）ListLength(LinkList L)：利用指针 p 和 int 类型变量 i（初始为 0），i 每加 1，p 每指向一个 next，直到 p 为空。

6）GetElem(LinkList L,int i,ElemType &e)：利用指针 p 和变量 j，从第一个节点开始找，若 p 为空，或 j>i，则返回错误，否则 e=p->data。

7）LocateElem(LinkList L,ElemType e)：利用指针 p，变量 i，从第一个开始找，直到找到元素 e，此时 i 为 e 的位置，若 p 不空，返回 i，否则返回 ERROR。

8）PriorElem(LinkList L,ElemType cur_e,ElemType& pre_e)：利用指

针 p 和 q 寻找，直到找到最后一个或 cur_e，p 是 q 的前驱，若 q 且 p 不等于 L，则 pre_e=p->data，返回 OK，否则返回 ERROR。

9）NextElem(LinkList L,ElemType cure_e,ElemType& next_e)：利用 p 和 q 找到 p 为 cur_e 或 q 空为止，q 是 p 的后继，若 p 为最后一个节点 q 会空，所以若 q 不空则 next_e=q->data，返回 OK。否则返回 ERROR。

10) LinkInsert(LinkList L,int i,ElemType e)：寻找第 i-1 个位置，开辟新节点，利用第 i-1 个节点的 next 域插入元素 e 到第 i 个位置。

11）ListDelete(LinkList L,int i,ElemType &e)：寻找第 i-1 个位置，p 是第 i-1 个节点，q 是第 i 个，则 p->next=q->next，再释放 q 节点。

12）LinkTraverse(LinkList L)：利用指针 p 遍历链表的每一个节点，并输出。

13）CheckIndex(float x,int y)：这个函数用来判断新输入得到项是否可以合并。先设 q=*pL，p=(*pL)->next，（pL 是全局变量的二级指针），若 p->index==y，跳出循环，否则一直找到 index 相等的，若 p 空，则返回 INFEASTABLE，否则找到相同指数的，p->coeff+=x，若相加后系数为 0，则删除该节点，返回 ERROR，否则返回 OK。

14）PolySort(LinkList L)：本函数采用冒泡排序思想，一轮比较找出一个最小的放在最后的位置。每轮比较都要从头开始，pre 是 p 的前驱，方便交换，第 i 轮第 j+1 次比较，若 p->index<q->index,则交换 p 指针指向的节点与 q 指针指向的节点，再交换 p 指针与 q 指针，进行下一次比较，每比较完一次指针后移，具体的排序算法如下：

```
for (i = 1; i <len; i++) {//第i轮比较
    pre = L;//每轮比较都要从头开始  pre是p的前驱 方便交换用
    p = pre->next;
    q = p->next;
    for (j = 0; j < len-i; j++) {//第i轮第j+1次比较
        if (q) {
            if (p->index < q->index) {
                pre->next = q;
                p->next = q->next;
                q->next = p;//交换p节点与q节点
                p = pre->next;
                q = p->next;//交换p指针与q指针位置 以便进行下一次比较
            }
            pre = pre->next;//每比较完一次指针后移
            p = p->next;
            q = q->next;
        }
    }
}
```

图 1.2-5 冒泡排序排多项式幂次

15）CreateList(int listnum)：先输入多项式的项数 n，每一次循环输入该项的系数和指数，如果系数是 0，直接跳到下一次循环输入下一项，利用 CheckIndex()函数检查原多项式是否有可以合并的项，有就合并，如果没有可以合并的，就建立新的节点，接到链表尾，如果多项式没有项，即 ListLength(*pL)==0，则该多项式为 0，否则 PolySort(*pL)。

16）ShowList(int listnum)：根据系数是 0，1，-1 和指数为 0 和 1 得到情况，分开讨论。先考虑第一项，如果系数是正的，如果系数是 1，可以忽略，如果指数是 1，可以忽略，如果指数是 0，输出 1，如果系数不是 1，如果指数是 1，可以忽略，如果指数是 0，直接输出系数。如果系数是 0，整个多项式是 0，直接输出 0.如果系数是负的，情况同正的时候。考虑第一项后面的项，如果系数是正的，则需补符号“+”，其余与第一项相同。

17）AddList(int a,int b,int c)：分别令 pa 和 pb 指向 a 和 b，若 a 和 b 都没有遍历完,遵从从指数大的项开始加的原则,比较 pa 和 pb 的 index，若不相等，则直接把 index 大的赋值给 pc，若相等，则系数相加，若相加系数为 0，则删除该节点。若多项式 b 先遍历结束，则开辟新节点直接为 a 此时节点相同，接在链表 c 后面，若 a 先遍历结束，情况同上。如果此操作后 c 不存在项，则为 0。若链表 c 长度不是 1，多项式中有 0，则去除该节点。

18）MinusList(int a,int b,int c)：将多项式 b 的每一项前添加负号，然后 AddList(int a,int b,int c)即可完成相减。

1.3 系统实现

本次实验在 Windows10 x64 位操作系统下，在 Dev-C++ 6.0 的 IDE 中完成，正如 1.2 中所写，本程序分为主程序模块，单链表模块，多项式模块。主程序主要实现菜单的显示，以及和用户的互动，接收命令，产生输出，所有的函数为：1. IntiaList；2. DestroyList；3. ClearList；4. ListEmpty；5. ListLength；6. GetElem；7. LocateElem；8. PriorElem；9. NextElem；10. ListInsert；11. ListDelete；12. ListTrabverse；19. SetListnum；21. CreateList；22. ShowList；23. AddList；24. MinusList 部分函数的主要程序行如下（具体可在源码中查看）：

```
Status PriorElem(LinkList L, ElemType cur_e, ElemType& pre_e) {//获得元素cur_e的前驱
    LNode* p, * q;
    p = L;
    q = p->next;
    while (q->next && q->data != cur_e) {//利用p和q寻找,一直找到最后一个或cur_e
        p = q;
        q = p->next;
    }
    if (q && p != L) {
        pre_e = p->data;//p是q的前驱 返回p的数据到pre_e
        return OK;
    }
    else
        return ERROR;
}
```

图 1.3-1 PriorElem 函数注释

```
Status CheckIndex(float x, int y) {//检查是否有可以合并的项
    LNode* p, * q;//p用来找有无相同的项 q用来判断若有相同的项 相加后系数是否为0
    q = *pL;
    p = (*pL)->next;
    while (p) {
        if (p->index == y) break;//如果当前项的指数是y 跳出循环
        q = p;
        p = p->next;
    }
    if (!p) return INFEASTABLE;//如果没有相同指数的项 返回没有
    else {
        p->coeff += x;//找到相同指数 则系数相加
        if (p->coeff == 0) {//若相加后系数为0 则删除节点
            q->next = p->next;
            delete[]p;
            return ERROR;
        }
        else return OK;
    }
}
```

图 1.3-2 CheckIndex 函数注释

```
Status CreateList(int listnum) {//序号listnum 建立一个多项式
    LNode* p, * q;
    int i = 1, n = 0;//i表示第i项 n表示一共有n项
    cout<<"Please input the quantity of item:";
    cin >> n;
    q = *pL;
    float x = 1;
    int y;
    while (x && i <= n) {
        cout<<"please input the coefficient of NO."<<i<<" item:";
        cin >> x;
        cout<<"please input the index of NO."<<i<<" item:";
        cin>>y;
        //x是第i项的系数，y是指数
        if (!x) {
            i++;
            continue;//如果系数是0，直接跳到下一个循环输入下一项
        }
        Status flag = CheckIndex(x, y);//检查原多项式是否有可以合并的项 有就合并
        if (flag == INFEASTABLE) {
            //如果没有可以合并的 就建立新的节点
            p = new LNode[sizeof(LNode)];
            p->coeff = x;//系数
            p->index = y;//指数
            p->next = NULL;//下一个节点为空
            q = Last(*pL);//令q指向最后一个节点 方便插入
            q->next = p;//把p接到链表尾
        }
        i++ ;
    }
    if (ListLength(*pL) == 0) {//没有项 该多项式为0
        p = new LNode[sizeof(LNode)];
        p->coeff = 0;
        p->index = 0;
        p->next = NULL;
        q->next = p;//q是当前链表头结点，下一项是0，再下一项空，整个多项式为0
    }
    else PolySort(*pL);//多项式不为0，按递减顺序排序
    return OK;
```

图 1.3-3 CreateList 函数注释

针对 1.1 所列重要功能，用 OnlineJudge 进行了详尽十组的测试，具体测试用例参照文件“实验 1 测试数据.pdf”，测试均通过，具体情况如表 1.3-1。

表 1.3-1 OnlineJudge 通过情况

ID	STATUS	MEMORY	TIME	SCORE
1	Accepted	3MB	1ms	10
2	Accepted	3MB	1ms	10
3	Accepted	4MB	1ms	10
4	Accepted	3MB	1ms	10
5	Accepted	3MB	1ms	10
6	Accepted	3MB	1ms	10
7	Accepted	3MB	1ms	10
8	Accepted	3MB	1ms	10

9	Accepted	3MB	1ms	10
10	Accepted	3MB	0ms	10

部分典型测试结果如下：

测试用例 2：主要测试单链表基本操作，测试了功能 19，1，10，3，12，5，6，7，8，9，11，3。

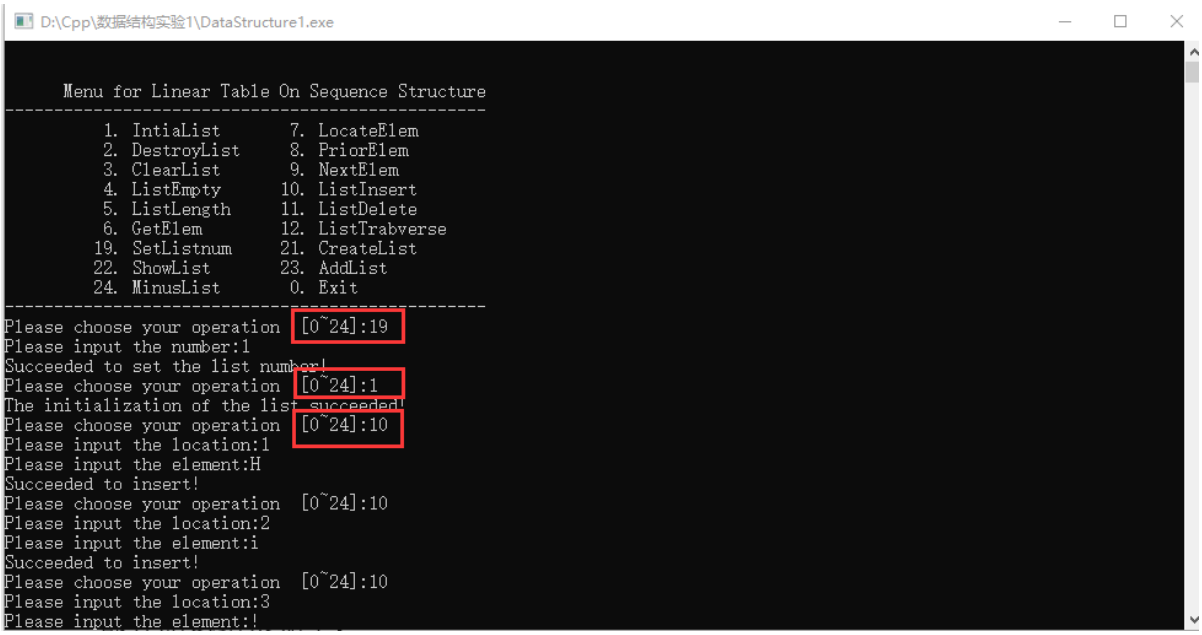


图 1.3-4 测试用例 2-1

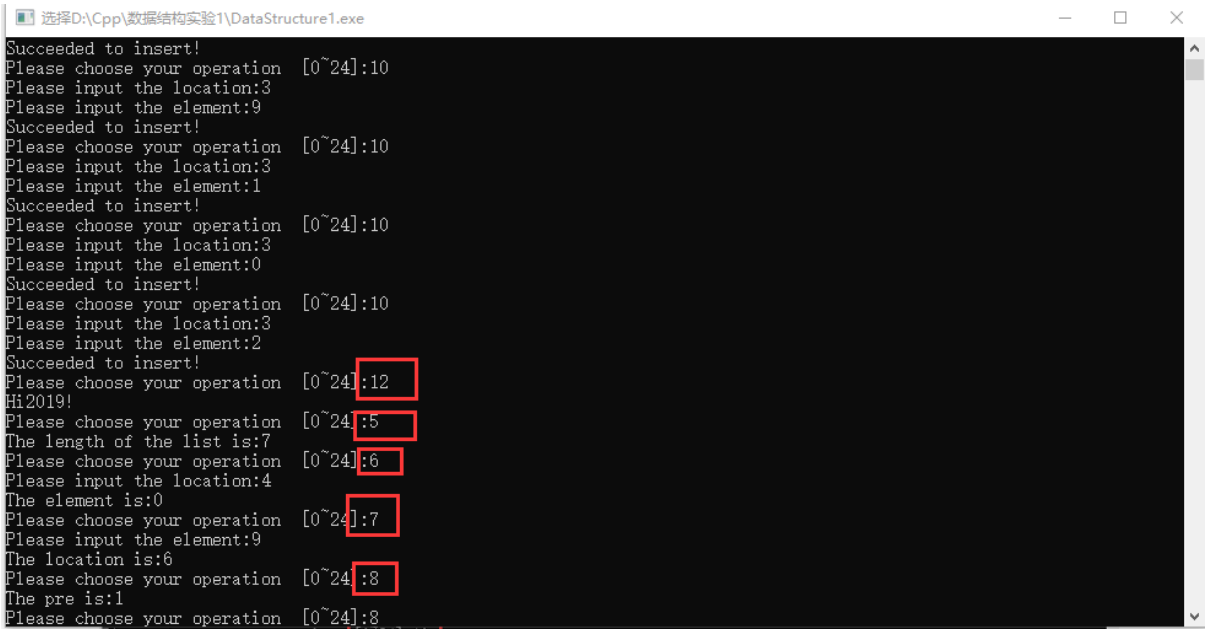


图 1.3-5 测试用例 2-2

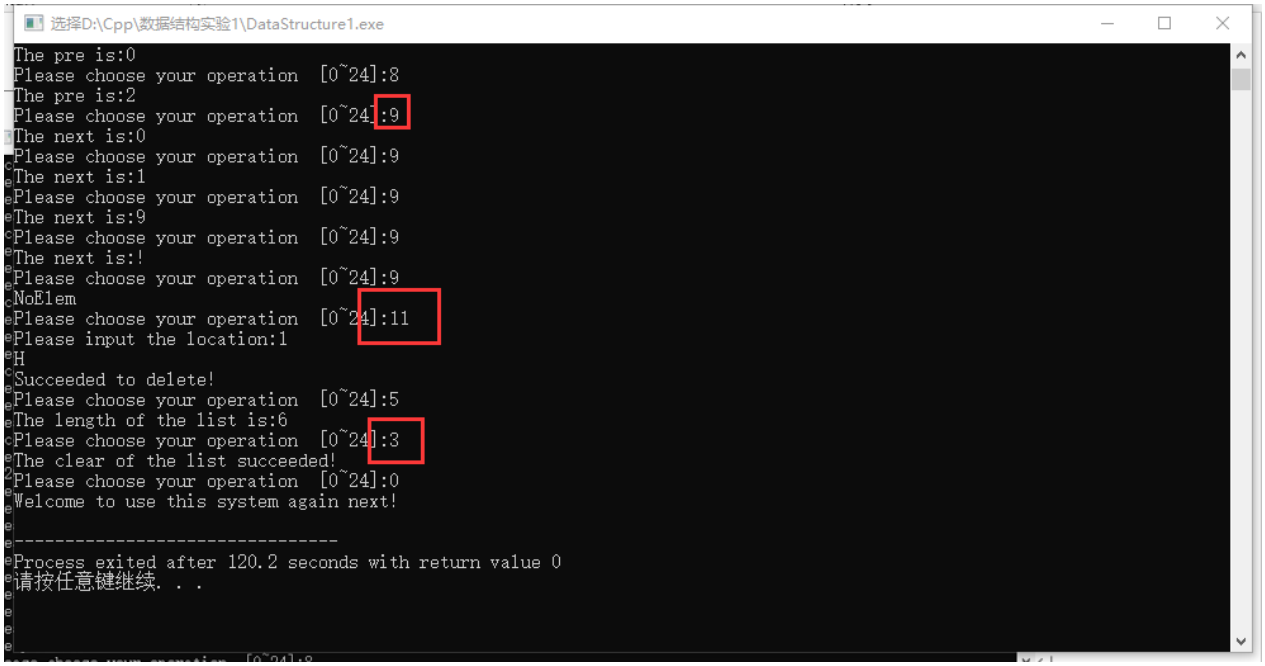


图 1.3-6 测试用例 2-3

测试用例 3：主要测试系数指数简单的多项式创建和运算，测试功能为 1，19，21，22，23，24。

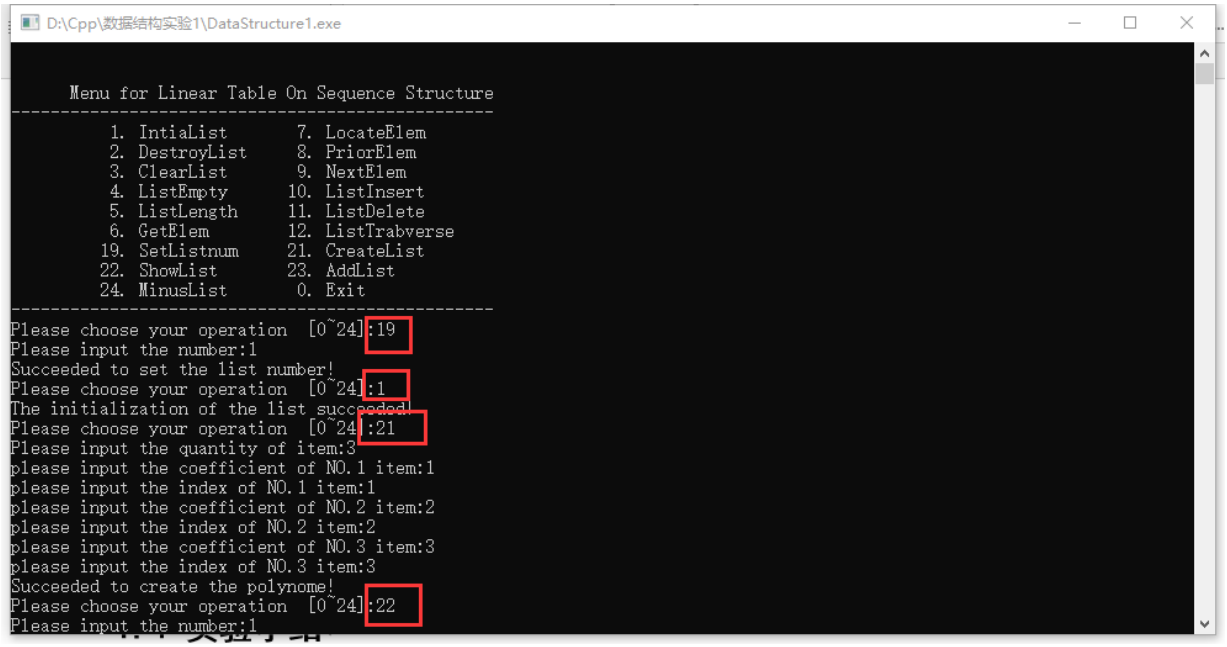


图 1.3-7 测试用例 3-1

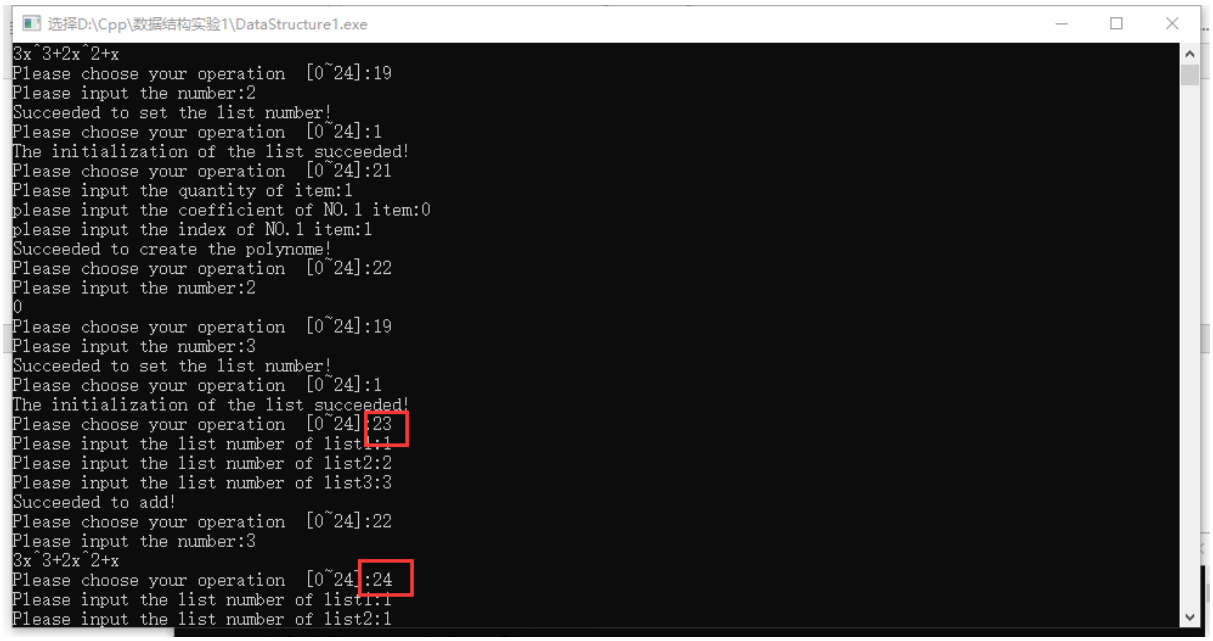


图 1.3-8 测试用例 3-2

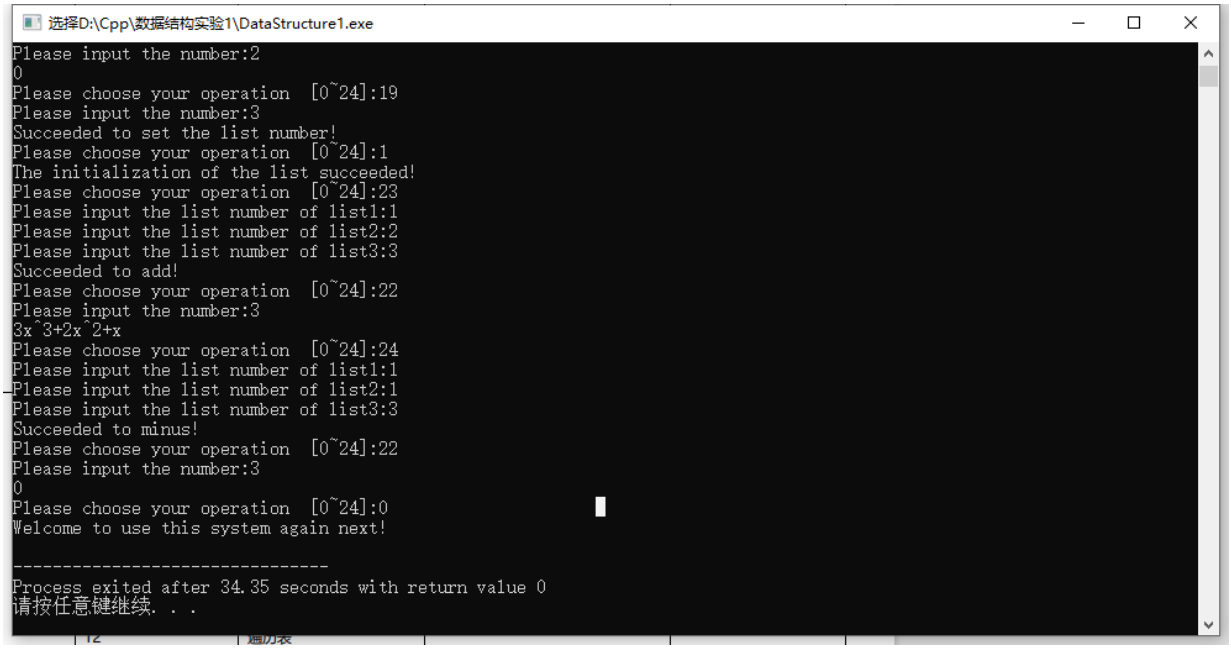


图 1.3-9 测试用例 3-3

测试用例 7：测试能否正确建立复杂系数的多项式，对多项式减法运算能否输出正确结果。

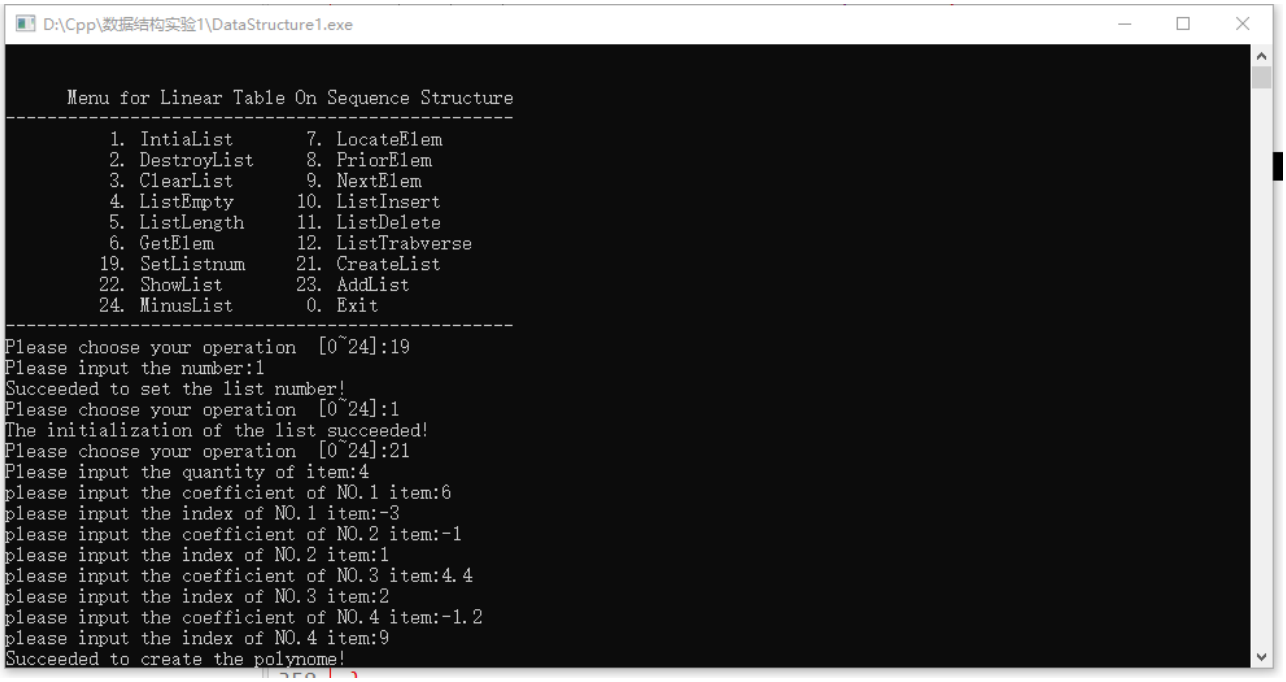


图 1.3-10 测试用例 7-1

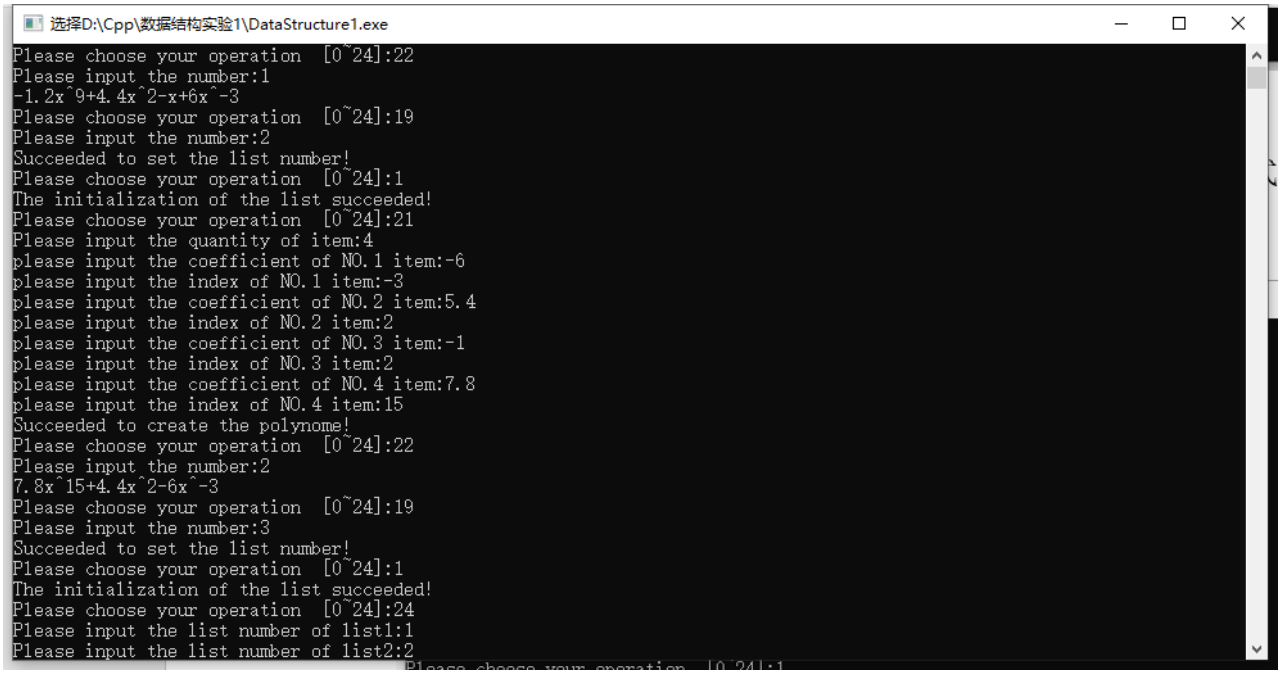


图 1.3-11 测试用例 7-2

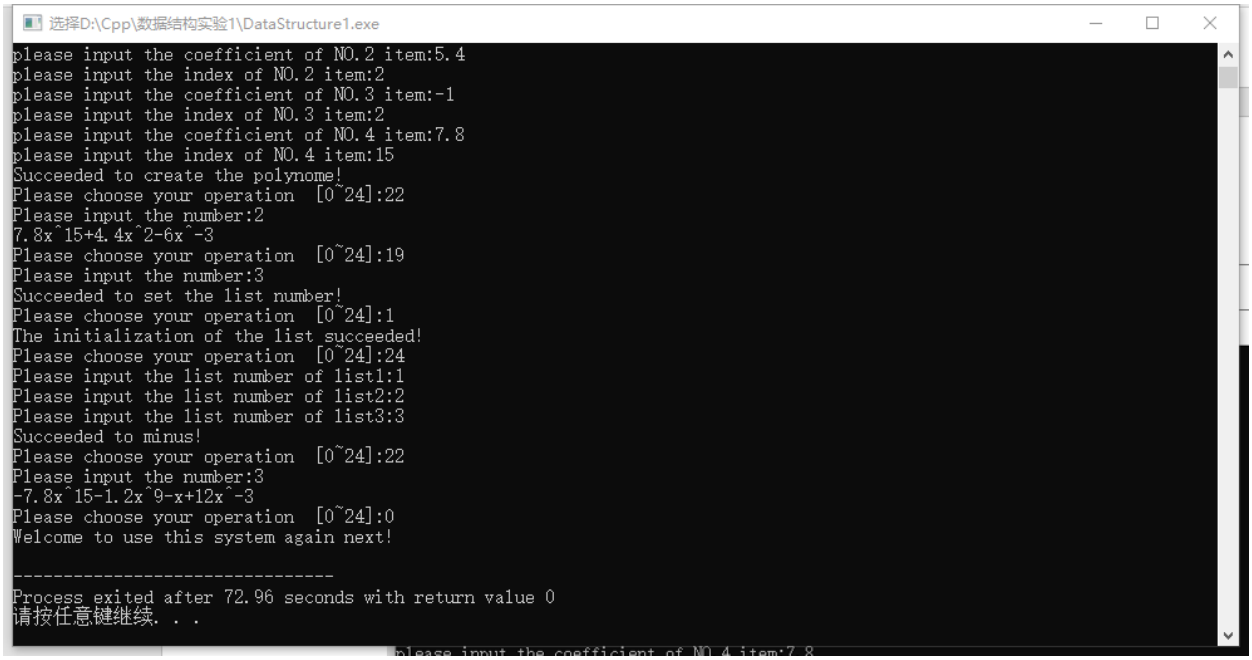


图 1.3-12 测试用例 7-3

1.4 实验小结

1.最开始没有注意使用 C++的引用，使得在链表初始化和销毁的过程中失败了，因为这两个过程需要改变头指针 L，要么使用双指针，要么使用引用，在以后的过程中需要注意这个问题。

2.最开始不知道如何进行多链表的管理，忘了全局变量和双指针的存在，走了很多弯路也没有实现，经过询问同学才得以解决。

3.在写 CreateList 函数时，遇到很多问题，比如碰到指数相同的项怎么去合并，如果为 0 怎么办，多项式排序怎么解决。后来想到用 CheckIndex 函数去解决合并同类项，而在排序时，我选择了最笨拙的冒泡排序，因为在做本次实验时只会冒泡排序，发现根据链表的 data 域来进行排序和直接对序列排序不同之处就是，本次实验排序时，不仅要交换 p 和 q 指向的节点，节点交换完还要交换 p 和 q 指针的指向，否则会发生错误，而在本次实验中，我也没有用改进后的冒泡排序,就是设立一个 exchange 变量,如果本轮没有产生交换,则令 exchange=0，跳出循环即可。造成了时间浪费。

4.本程序的模块划分较合理。主程序中进行交互，调用相应函数，函数间层次明显，感觉效率较高。

5.算法的时空分析

1）若设多项式的长度为 n，则销毁链表，清空链表，求链表长度，获得元素，获得元素位置，求前驱，求后继，插入元素，删除元素，遍历链表的时间复杂度为 O(n)，链表的初始化和判断是否为空时间复杂度为 O(1)，在多项式排序中，由于用了冒泡排序，所以时间复杂度是 O(n^2)，其余的多项式操作时间复杂度均是 O(n)。整体的时间复杂度不是很高，算法还算合适。

2）除了进行多项式加减需要额外的存储空间，空间复杂度为 O(n)，其余算法的空间复杂度均为 O(1)。

6.本次实验采用数据抽象的程序设计方法，将程序划分为三个模块，加深了对线性表的概念和基本运算的理解，熟练掌握了线性表的逻辑结构和物理结构的关系，物理结构采用单链表，熟练掌握线性表的基本运算的实现，并进行了扩展应用，学习掌握了程序框架的构建和通过键盘键入参数的方法，掌握了将各个基本运算功能模块组织在一个可执行系统中的方法，同时熟悉了 C++语言，总体来

说本次实验思路较清晰，调试顺利，收获很大，是一次良好的数据结构实验。

感谢老师和助教在本次实验中的帮助！

2 基于二叉链表的二叉树实现

2.1 问题描述

本次实验在 Windows10 x64 位操作系统下，在 Dev-C++ 6.0 的 IDE 中开发，来实现一个具有菜单演示功能的系统，演示系统实现程序运行参数的键盘输入，实现多个二叉树的管理，每个二叉树要实现初始化，求孩子节点，求兄弟节点，遍历等等基本功能，并在此基础上实现一个扩展应用:“哈夫曼编码器和解码器”。哈夫曼树的创建为带空节点的前序创建，利用创建的哈夫曼树进行前缀编码，进而对一个字符串进行编码，同时可以利用创建的哈夫曼树对编码的字符串进行解码，在编码时将编码写入文件 CodeFile.txt，解码时直接读入该文件。

2.1.1 菜单演示功能

菜单演示系统以用户和计算机的对话方式执行，即在计算机终端上显示“提示信息”，由用户在键盘上输入演示系统中规定的功能序号，再由用户输入要实现当前功能所用的数据，来实现该功能。输入后若有相应的结果则显示在其后。

2.1.2 二叉树基本功能

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的如下功能：1）初始化二叉树；2）销毁二叉树；3）创建二叉树；4）清空二叉树；5）判定空二叉树；6）求二叉树的深度；7）获得根节点；8）获得节点；9）节点赋值；10）获得双亲节点；11）获得左孩子节点；12）获得右孩子节点；13）获得左兄弟节点；14）获得右兄弟节点；15）插入子树；16）删除子树；17）前序遍历；18）中序遍历；19）后序遍历；20）层序遍历

2.1.3 多个二叉树管理

本系统需要实现多个二叉树的管理，采用数组的方式实现，数组中的元素是每个二叉树的根结点指针，具体实现见系统设计部分。

2.1.4 扩展应用

本实验要求在二叉树基础上实现“哈夫曼编码器和解码器”。详见后文。

2.2 系统设计

为实现上述程序功能，应先实现二叉树的各项功能，然后以二叉树表示哈夫曼树，再进行编码和解码。

1. 二叉树的抽象数据类型定义为：

ADT BinaryTree{

数据对象 D:D 是具有相同特性的数据元素的集合。

数据关系 R:

若 D=Φ，则 R=Φ，称 BinaryTree 为空二叉树；

若 D≠Φ，则 R={H}，H 是如下的二元关系：

（1） 在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；

（2） 若 D-{root}≠Φ，则存在 D-{root}={D_1,D_r}，且 D_1∩D_2=Φ；

（3） 若 D_1≠Φ，则 D_1 中存在唯一的元素 x_1，<root,x_1>∈H，且存在 D_1 上的关系 H_1⊆H；若 D_r≠Φ，则 D_r 中存在唯一的元素 x_r，<root,x_r> ∈ H ， 且 存 在 D_r 上 的 关 系 H_r⊆H ；H={<root,x_1>,<root,x_r>，H_1,H_r}；

（4） （D, {H_1}）是一棵符合本定义的二叉树，称为根的左子树，（D_r, {H_r}）是一棵符合本定义的二叉树，称为根的右子树。

基本操作 P：

⑴初始化二叉树：函数名称是 InitBiTree(T)；初始条件是二叉树 T 不存在；操作结果是构造空二叉树 T。

⑵销毁二叉树：树函数名称是 DestroyBiTree(T)；初始条件是二叉树 T 已存在；操作结果是销毁二叉树 T。

⑶创建二叉树：函数名称是 CreateBiTree(T,definition)；初始条件是 definition 给出二叉树 T 的定义；操作结果是按 definition 构造二叉树 T。

⑷清空二叉树：函数名称是 ClearBiTree (T)；初始条件是二叉树 T 存在；操作结果是将二叉树 T 清空。

⑸判定空二叉树：函数名称是 BiTreeEmpty(T)；初始条件是二叉树 T 存在；操作结果是若 T 为空二叉树则返回 TRUE，否则返回 FALSE。

⑹求二叉树深度：函数名称是 BiTreeDepth(T)；初始条件是二叉树 T 存在；操作结果是返回 T 的深度。

⑺获得根结点：函数名称是 Root(T)；初始条件是二叉树 T 已存在；操作结果是返回 T 的根。

⑻获得结点：函数名称是 Value(T,e)；初始条件是二叉树 T 已存在，e 是 T 中的某个结点；操作结果是返回 e 的值。

⑼结点赋值：函数名称是 Assign(T,&e,value)；初始条件是二叉树 T 已存在，e 是 T 中的某个结点；操作结果是结点 e 赋值为 value。

⑽获得双亲结点：函数名称是 Parent(T,e) ；初始条件是二叉树 T 已存在，e 是 T 中的某个结点；操作结果是若 e 是 T 的非根结点，则返回它的双亲结点指针，否则返回 NULL。

⑾获得左孩子结点：函数名称是 LeftChild(T,e)；初始条件是二叉树 T 存在，e 是 T 中某个节点；操作结果是返回 e 的左孩子结点指针。若 e 无左孩子，则返回 NULL。

⑿获得右孩子结点：函数名称是 RightChild(T,e)；初始条件是二叉树 T 已存在，e 是 T 中某个结点；操作结果是返回 e 的右孩子结点指针。若 e 无右孩子，则返回 NULL。

⒀获得左兄弟结点：函数名称是 LeftSibling(T,e)；初始条件是二叉树 T 存在，e 是 T 中某个结点；操作结果是返回 e 的左兄弟结点指针。若 e 是 T 的左孩子或者无左兄弟，则返回 NULL。

⒁获得右兄弟结点：函数名称是 RightSibling(T,e)；初始条件是二叉树 T 已存在，e 是 T 中某个结点；操作结果是返回 e 的右兄弟结点指针。若 e 是 T 的右孩子或者无有兄弟，则返回 NULL。

⒂插入子树：函数名称是 InsertChild(T,p,LR,c)；初始条件是二叉树 T 存在，p 指向 T 中的某个结点，LR 为 0 或 1，,非空二叉树 c 与 T 不相交且右子树为空；操作结果是根据 LR 为 0 或者 1，插入 c 为 T 中 p 所指结点的左或右子树，p 所指结点的原有左子树或右子树则为 c 的右子树。

⒃删除子树：函数名称是 DeleteChild(T.p.LR)；初始条件是二叉树 T 存在，p 指向 T 中的某个结点，LR 为 0 或 1。 操作结果是根据 LR 为 0

或者 1，删除 c 为 T 中 p 所指结点的左或右子树。

⑰前序遍历：函数名称是 PreOrderTraverse(T,Visit())；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果：先序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

⑱中序遍历：函数名称是 InOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果是中序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

⑲后序遍历：函数名称是 PostOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果是后序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

⑳按层遍历：函数名称是 LevelOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果是层序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

}ADT BinaryTree

2. 哈夫曼树的基本操作定义为：

⑴建立哈夫曼树：函数名称是 CreateHTree(HTnum)；初始条件是 Htnum 指向的二叉树已存在；操作结果是根据定义创建一棵哈夫曼树。

⑵字符序列编码：函数名称是 HTreeCode(String)；初始条件是给定一个字符串 String，且哈夫曼树已经创建；操作结果是将给定的字符串进行 01 字符串编码。

⑶解码为字符序列：函数名称是 HTreeDecode(code)；初始条件是给定一个编码的 01 字符串 code，且哈夫曼树已经创建；操作结果是将给定的 01 编码解码为字符串。

⑷字符序列编码解码比对：函数名称是 StringCheck(String)；初始条件是给定一个字符串 String，且哈夫曼树已经创建；操作结果是对 String 的先编码后解码的结果与 String 进行比对。

⑸编码解码与编码比对：函数名称是 CodeCheck(code)；初始条件是给定一个编码的 01 字符串 code，且哈夫曼树已经创建；操作结果是对 code 的先解码后编码的结果与 code 进行比对。

3.本程序包含三个模块：

1）主程序模块：

```
int main(){
初始化;
显示菜单;
while(op){
接收命令;  处理命令;} }
```

2）二叉树模块——实现二叉树的抽象数据类型

3）哈夫曼树模块——实现哈夫曼树的各项基本功能

本程序需要实现多个二叉树的管理：

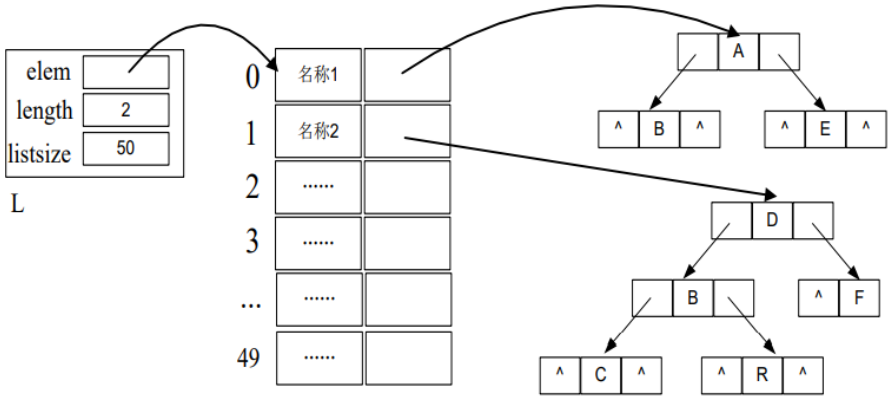


图 2.2-1 多二叉树管理

各模块之间的调用关系如下：

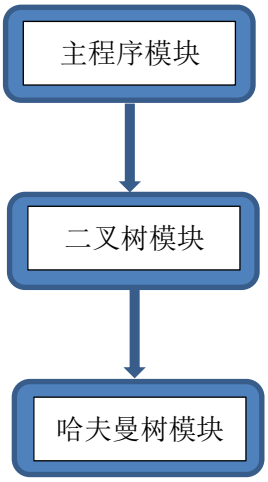


图 2.2-2 各模块调用关系

二叉树模块具体如下：

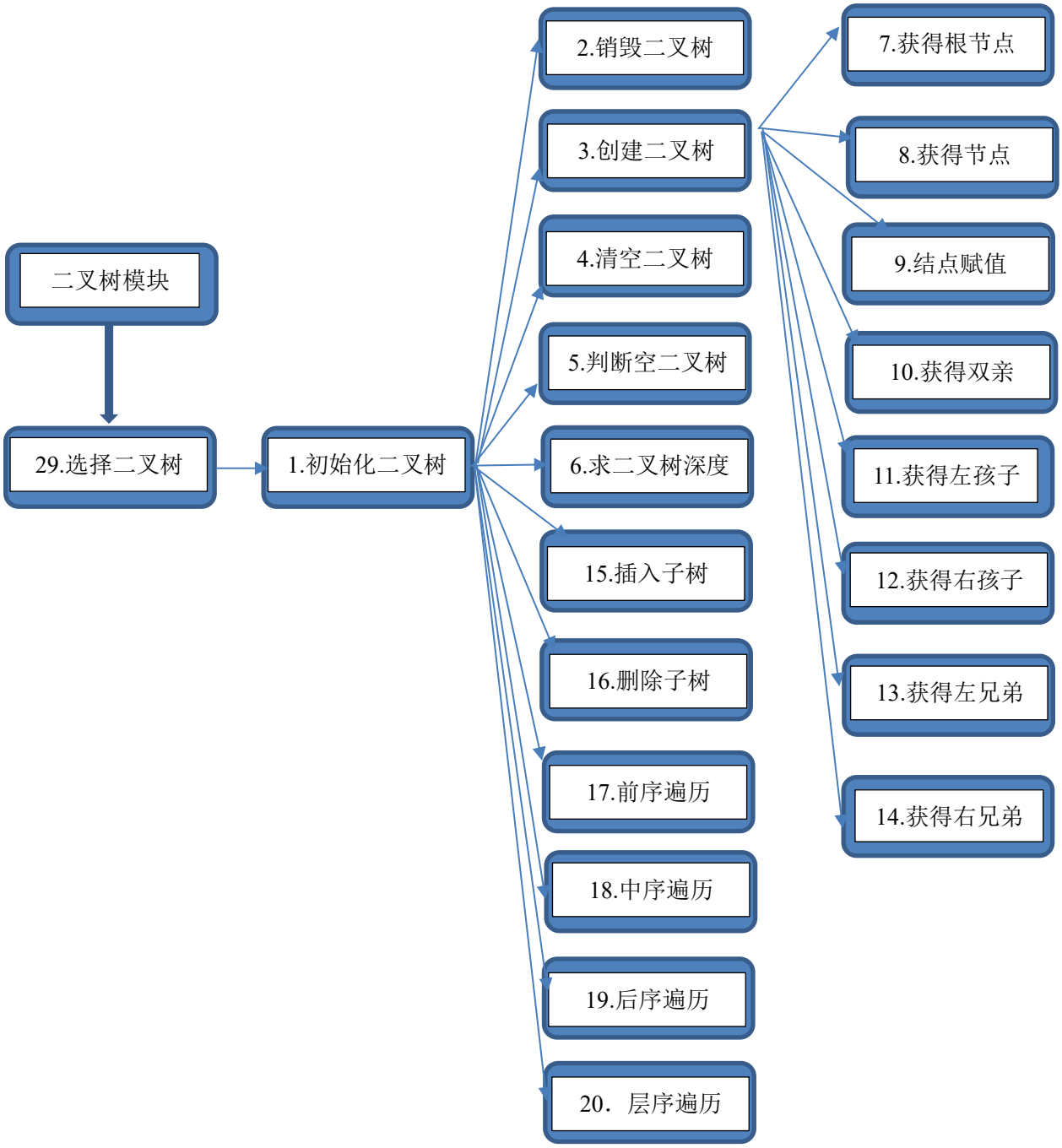


图 2. 2-3 二叉树模块

哈夫曼树模块具体如下：

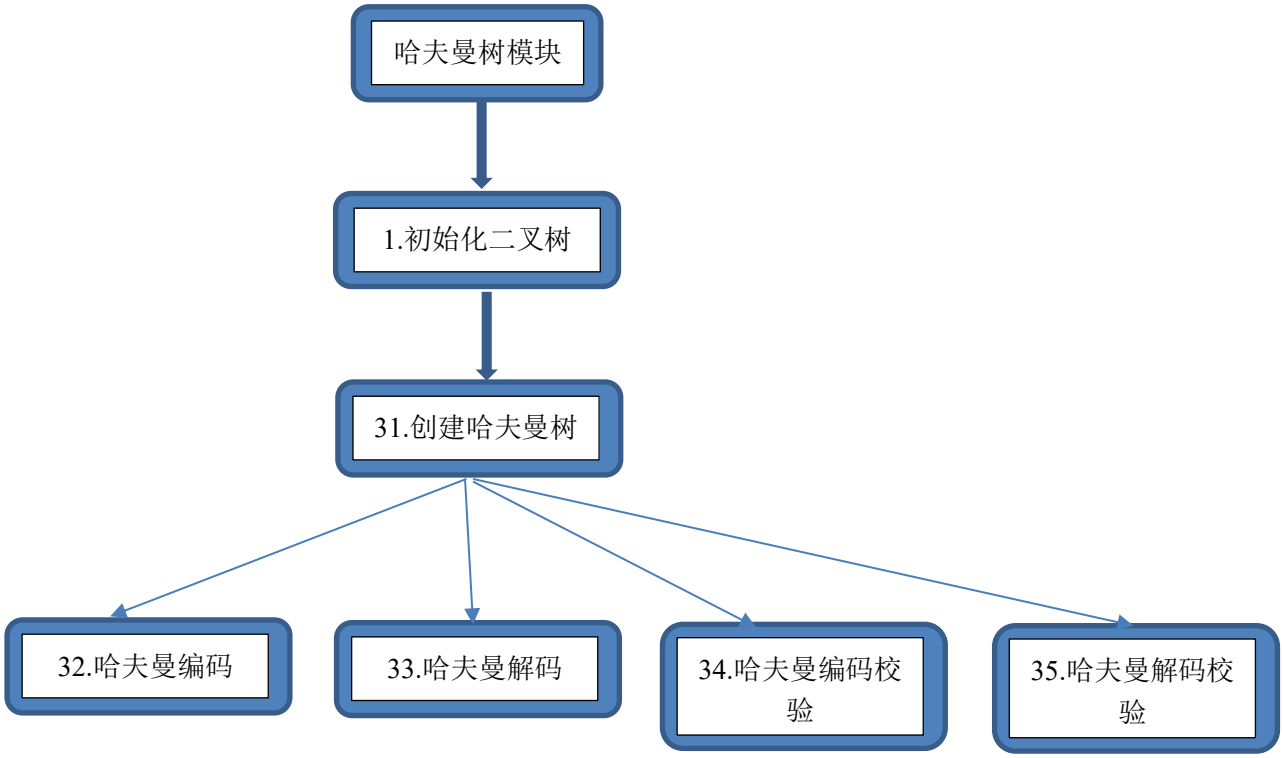


图 2.2-4 哈夫曼树模块

4.本程序主要的算法为：

（1）InitBiTree(T)：给当前序号的二叉树创建数据是^,孩子指针为空的根节点。

（2）DestroyBiTree(T):递归销毁二叉树，先销毁左子树，再销毁右子树，最后释放根节点。若根节点为空，则 return;。

（3）CreateBiTree(T,definition)：对给定的字符序列递归先序创建二叉树。先生成根节点，再递归创建左子树，右子树。

（4）ClearBiTree(T)：同 DestroyBiTree(T)。

（5）BiTreeEmpty(T)：判断根节点是否为^，若为^则返回 TRUE，否则返回 FALSE。

（6）BiTreeDepth(T)：递归。返回左子树与右子树中深度较大的值+1，若根节点为空，返回 0。

（7）Root(T):返回根节点 T。

（8）Value(T,e)：判断节点是否为空，若为空，返回^到 e，否则返回数据到 e。

（9）Assign(T,&e,value)：判断节点是否为空，若为空，返回 ERROR，否则传入参数的值到该节点。

（10）Parent(T,e)：采用队列对二叉树 T 进行层次遍历。访问每个节点时，判断左右孩子节点是否为 e，若是，则该节点即是 e 的双亲，若不是，则左右孩子入队，继续访问队列中的出队的节点，直到队列为空，若遍历结束仍未找到匹配的节点，则返回 NULL。

（11）LeftChild(T,e)：采用队列对二叉树 T 进行层次遍历。访问每个节点时，判断左孩子节点是否为 e，若是，则返回该节点的左孩子，若不是，则左右孩子入队，继续访问队列中的出队的节点，直到队列为空，若遍历结束仍未找到匹配的节点，则返回 NULL。

（12）RightChild(T,e)：采用队列对二叉树 T 进行层次遍历。访问每个节点时，判断右孩子节点是否为 e，若是，则返回该节点的右孩子，若不是，则左右孩子入队，继续访问队列中的出队的节点，直到队列为空，若遍历结束仍未找到匹配的节点，则返回 NULL。

（13）LeftSibling(T,e)：采用队列对二叉树 T 进行层次遍历。访问每个节点时，判断左孩子节点是否为 e 且是否有右孩子，若是，则返回该节点的右孩子，若不是，则左右孩子入队，继续访问队列中的出队的节点，直到队列为空，若遍历结束仍未找到匹配的节点，则返回 NULL。

（14）RightSibling(T,e)：采用队列对二叉树 T 进行层次遍历。访问每个节点时，判断右孩子节点是否为 e 且是否有左孩子，若是，则返回该节点的左孩子，若不是，则左右孩子入队，继续访问队列中的出队的节点，直到队列为空，若遍历结束仍未找到匹配的节点，则返回 NULL。

（15）InsertChild(T,p,LR,c)：判断原树、目标树、参数结点是否为空。原树为空则返回成功；若目标树、参数结点为空，则返回 ERROR。若原树插入为结点左子树，则结点的原左子树插入为原树的右子树，再将原树插入为结点的右子树。若原树插入为结点右子树，则结点的原右子树插入为原树的右子树，再将原树插入为结点的右子树。

（16）DeleteChild(T,p,LR,c)：判断树、结点是否为空。若空，则返回 ERROR。若删除结点的左子树，则用 DestroyBiTree()销毁左子树，并且建立新的结点，数据为^，孩子指针为空。若删除结点的右子树，则采用 DestroyBiTree()销毁右子树，并且建立新结点，数据为^，孩子指针为空。

（17）PreOrderTraverse(T,Visit())：采用递归。先遍历根节点，再遍历左子树，再遍历右子树。若为空，return;。

（18）InOrderTraverse(T,Visit())：采用非递归。用栈进行深度优先遍历，从根节点开始，每次循环直接找到最靠左的节点，途经节点全部入栈。若栈不空，访问栈顶元素，若栈顶元素右子树存在，则从右子树根节点开始重新进入循环，直至栈为空。

（19）PostOrderTraverse(T,Visit())：采用递归。先遍历左子树，再遍历右子树，再遍历根节点。若为空，return;。

（20）LevelOrderTraverse(T,Visit())：采用队列对二叉树 T 进行层次遍历。访问每个节点时，输出该节点内容，然后左右孩子入队，继续访问队列中的出队的节点，直到队列为空，若遍历结束仍未找到匹配的节点，则返回 NULL。

（21）CreateHTree(HTnum)：算法同 CreateBiTree()。递归先序创建二叉树。先创建根节点，然后递归创建左子树，递归创建右子树。

（22）CreateBeforeCode(T,vector<ElemType> temp)：先创建结构 Huffman，其中 char 型 ch 存放数据，string 类 str 存放对应编码。用向量 temp 给哈夫曼树 T 创建前缀编码，如果 T 是叶子节点，开始创建编码，ch 中存放 T 的 data 域数据，向量 temp 中存的是途经哈夫曼树的所有 0 和 1，所以用 for 循环遍历 temp 每一个元素放入 str 即可。如果不是叶子节点，如果有左孩子，向 temp 中存一个‘0’，再递归创建左子树的编码，创建完后弹出，开始创建下一次，每递归一次弹出一个，如果有右孩子，算法同左孩子。

（23）HTreeCode(string)：先用 CreateBeforeCode()给哈夫曼树编码，再用文件操作准备将编码写入 txt，要编码的字符串为 def，用第一层 for 循环，遍历 def 的每个字符，再用第二层 for 循环找与当前 def 字符相等的字符的 huffman 中的数据，找到后记录，跳出第二层循环。输出 def 当前字符的编码。

（24）HTreeDecode(code)：已经用 CreateBeforeCode()给哈夫曼树编码完，

且已经存在 CodeFile.txt 文件。读入 CodeFile.txt 文件，不管三七二十一顺着编码一直找到叶子节点，不是叶子节点则跳过本次循环，找到叶子节点后即是该编码对应的值，再回到根节点重新循环。

（25）StringCheck(string)：先用 HtreeCode()对 string 编码，再用 HtreeDecode()解码，若所得结果为 string，则成功，否则失败。

（26)CodeCheck(code)：先用 HtreeDecode()对 code 解码，再用 HtreeCode()编码，若所得结果为 code，则成功，否则失败

2.3 系统实现

本次实验在 Windows10 x64 位操作系统下，在 Dev-C++ 6.0 的 IDE 中完成，正如 2.2 中所写，本程序分为主程序模块，二叉树模块，哈夫曼树模块。主程序主要实现菜单的显示，以及和用户的互动，接收命令，产生输出，所有的函数为：1. setBTnum；2.InitBiTree；3.DestoryBiTree；4.CreateBiTree；5. ClearBiTree；6.BiTreeEmpty；7.BiTreeDepth；8.Root；9.Value；10. Assign；11.Parent；12LeftChild；13.RightChild；14.LeftSibling；15.RightSibling；16. InsertChild；17. DeleteChild；18.PreOrderTraverse；19. InOrderTraverse；20. PostOrderTraverse；21.LevelOrderTraverse；22.CreateHTree；23.HtreeCode；24. HtreeDecode；25.StringCheck；26.CodeCheck。部分函数的主要程序行如下（具体可在源码中查看）：

```
//求当前节点双亲 并指向双亲 此处层序遍历 要用队列
ElemType Parent(BiTree &T, BiTree& e) {//T是根节点指针 e是当前节点
    queue<BiTree> q;//创建队列q
    BiTree p = NULL;
    q.push(T);//根节点入队
    while (!q.empty()) {//队列不空
        p = q.front();
        q.pop();//p指向队列弹出的第一个元素
        if (p->lchild== e){//若p的左孩子为节点e
            tmp[n] = p;//指向双亲
            return p->data;//则p为e的双亲
        }
        else if (p->rchild== e) {//若p的右孩子为节点e
            tmp[n] = p;//指向双亲
            return p->data;//则p为e的双亲
        }
        if (p->lchild) q.push(p->lchild);//左孩子不空则入队
        if (p->rchild) q.push(p->rchild);//右孩子不空则入队
    }
        //若没有双亲
    return '^';//返回^
}
```

图 2.3-1 Parent()注释

```
//中序非递归遍历
void InOrderTraverse(BiTree T){
    BiTree st[1000];//建立一个栈
    int top = 0;//栈顶指针
    do {
        while (T) {//不管三七二十一 一直往左走
            if (top == 1000) exit(OVERFLOW);
            st[top++] = T;// 根节点一直入栈
            T = T->lchild;//一直往左走
        }
        if (top) {//栈不空
            T = st[--top];//弹出根节点
            cout << T->data;//遍历根节点
            T = T->rchild;//赋值右子树 进入下一次循环中序遍历右子树
        }
    } while (top||T);//若栈不空或T非空
}
```

图 2.3-3 InOrderTraverse()注释

```
//利用队列层序遍历二叉树
void LevelOrderTraverse(BiTree T){
    if (!T) return;
    queue<BiTree> q;//创建队列 层序遍历 依次把按层序遍历的元素顺序依次入队 然后依次出队
    BiTree p = NULL;
    q.push(T);//根节点入队
    while (!q.empty()) {//队列不空
        p = q.front();//p指向队列弹出的第一个元素
        q.pop();
        cout << p->data;//输出p的数据
        if (p->lchild) q.push(p->lchild);//左孩子不空则左孩子入队
        if (p->rchild) q.push(p->rchild);//右孩子不空则右孩子入队
    }
}
```

图 2.3-3 LevelOrderTraverse()注释

```
//先序创建了一个huffman树 只需给每个叶子节点编码
void CreateBeforeCode(BiTree T, vector<ElemType> temp) {//用向量temp给二叉树T创建前缀编码
    string s;//编码串
    if (isLeafNode(T)) {//如果是叶子节点 开始编码
        huffman[++huffnum].ch = T->data;//ch存放数据
        for (int i = 0; i < temp.size();i++) {
            s += temp[i];//temp中存放了从根节点到该叶子节点的编码
        }
        huffman[huffnum].str = s;//str存放ch对应的编码
    }
    else {
        if (T->lchild) {//如果有左孩子
            temp.push_back('0');//向temp中存一个0
            CreateBeforeCode(T->lchild, temp);//创建左孩子的编码
            temp.pop_back();//创建完后弹出 开始创建下一次 每递归一次弹出一个
        }
        if (T->rchild) {//同左孩子
            temp.push_back('1');
            CreateBeforeCode(T->rchild, temp);
            temp.pop_back();
        }

    }
}
```

图 2.3-4 CreateBeforeCode()注释

```
void HTreeCode(BiTree T) {//字符串编码函数
    vector<ElemType> temp;//创建编码用的向量

    int count = 0;//找huffman数组中与要编码的字符串的字符相同的数据，记下编号
    int i, j;//循环变量
    CreateBeforeCode(T, temp);//先用Huffman树T创建编码

    ofstream ofile;
    ofile.open("d:\\CodeFile.txt");//将编码写入D:\CodeFile.txt中

    for ( i = 0; i < def.size(); i++) {//第一层循环 循环的是def的每个字符
        for ( j = 0; j < 100; j++) {//第二层循环 找与当前def中的字符相等的huffman中的数据
            if (huffman[j].ch == def[i]) {
                count = j;//找到后记录 跳出第二层循环
                break;
            }
        }

        retcode+=huffman[count].str;
        ofile<<huffman[count].str;
        cout <<huffman[count].str;//输出该字符的编码
    }

    cout << endl;
    ofile.close();
}
```

图 2.3-5 HTreeCode()注释

针对 2.1 所列重要功能，用 OnlineJudge 进行了详尽十组的测试，具体测试用例参照文件“实验 2 测试数据.pdf”，测试均通过，具体情况如表 2.3-1。

表 2.3-1 OnlineJudge 通过情况

ID	STATUS	MEMORY	TIME	SCORE
1	Accepted	3MB	4ms	10
2	Accepted	3MB	0ms	10
3	Accepted	3MB	4ms	10
4	Accepted	3MB	4ms	10
5	Accepted	3MB	1ms	10
6	Accepted	3MB	1ms	10
7	Accepted	3MB	0ms	10
8	Accepted	3MB	0ms	10
9	Accepted	3MB	4ms	10
10	Accepted	3MB	0ms	10

部分典型测试结果如下：

测试用例 10：囊括大部分需要测试的基本操作函数，能很好地反映本次实验的效果。实验首先分别创建第二棵，第五棵，第六棵二叉树，在创建过程中完成

了对二叉树的自由访问孩子，双亲，兄弟，判空，求表长，求深度，求根等一系列操作，之后又重新选择第二棵树为工作数，是否能恰当回到第二棵树也是重点，之后又完成对二叉树的插入和删除子树，又使用后序遍历和层序遍历，是很有代表性的测试数据。

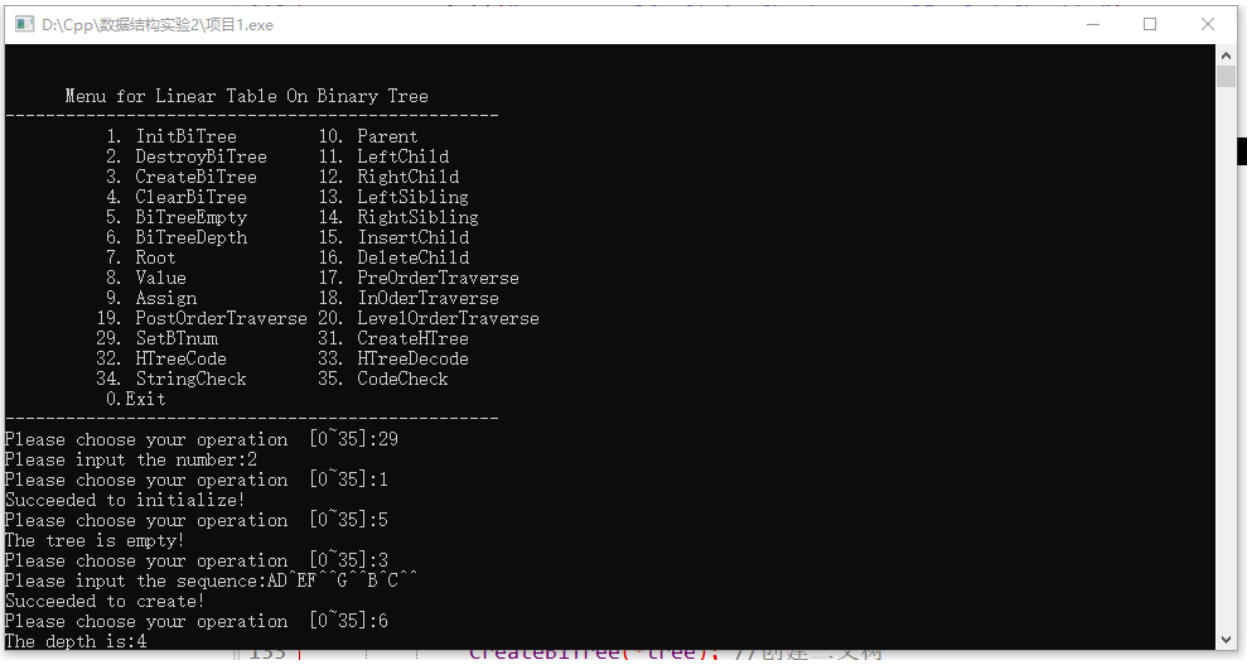


图 2.3-6 测试用例 10-1

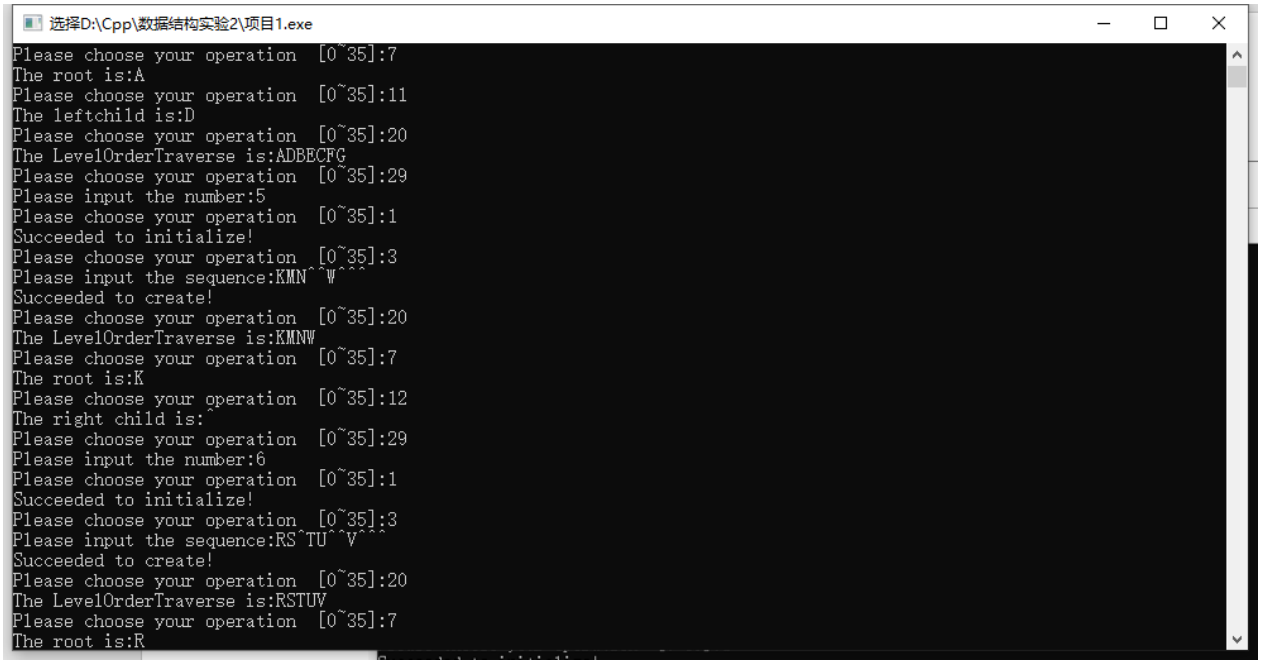


图 2.3-7 测试用例 10-2

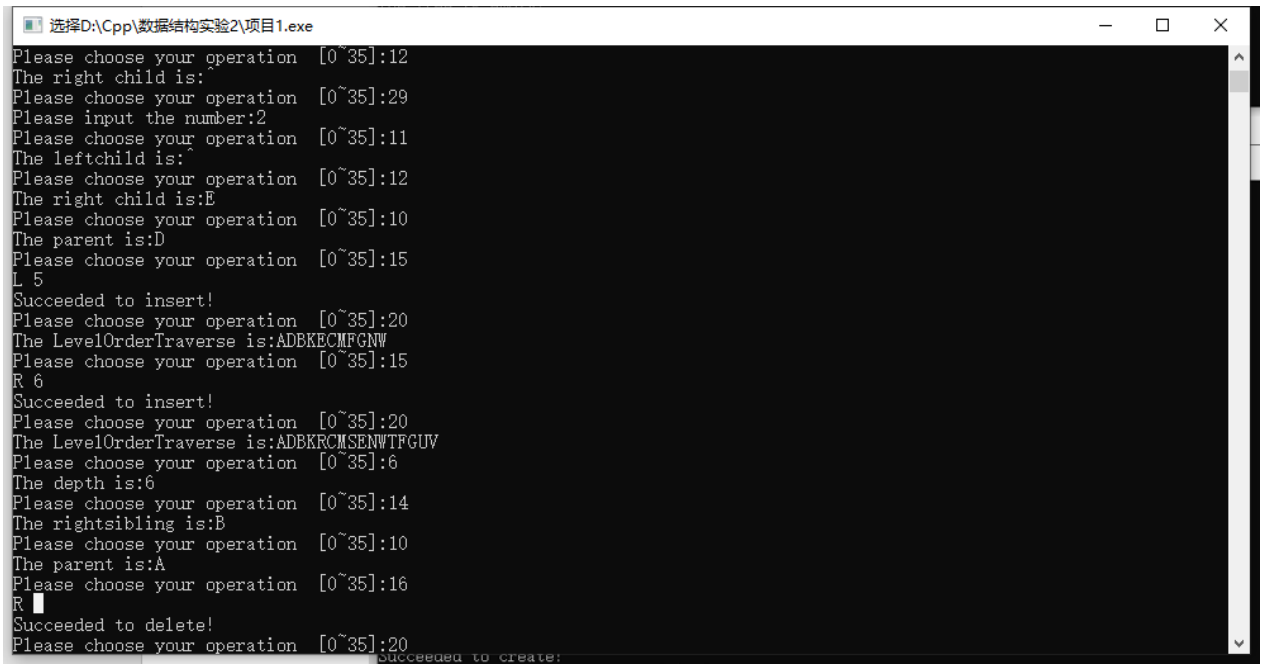


图 2.3-8 测试用例 10-3

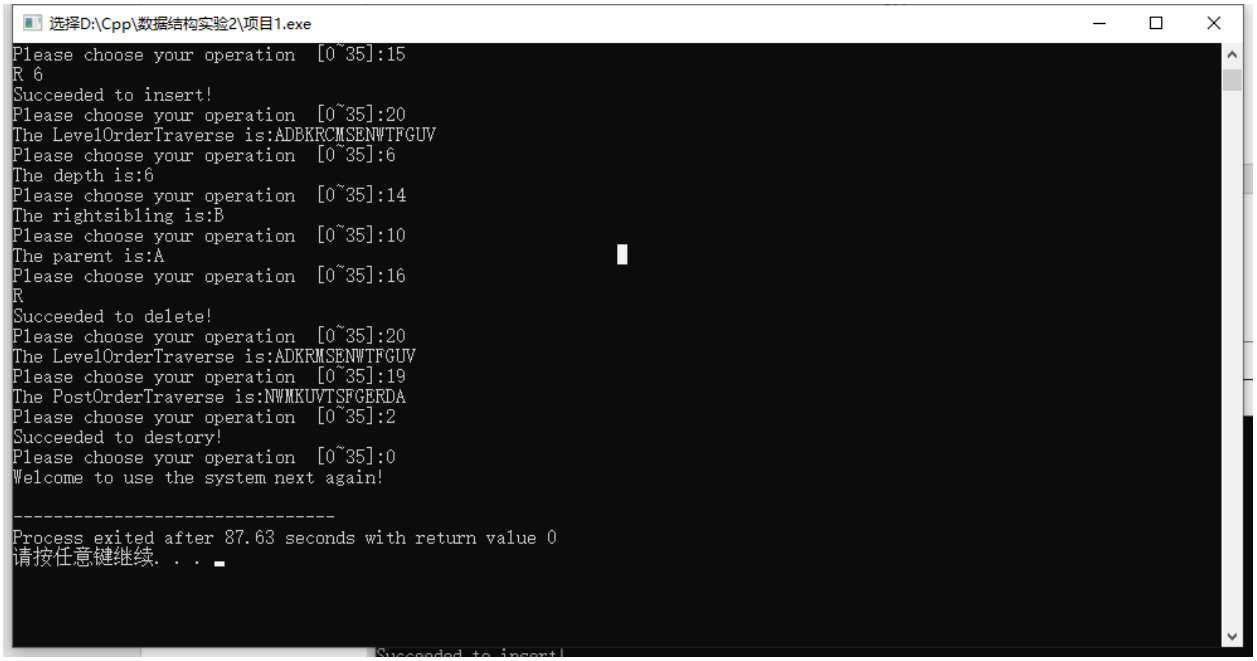


图 2.3-9 测试用例 10-4

测试用例 4：检验前序遍历。

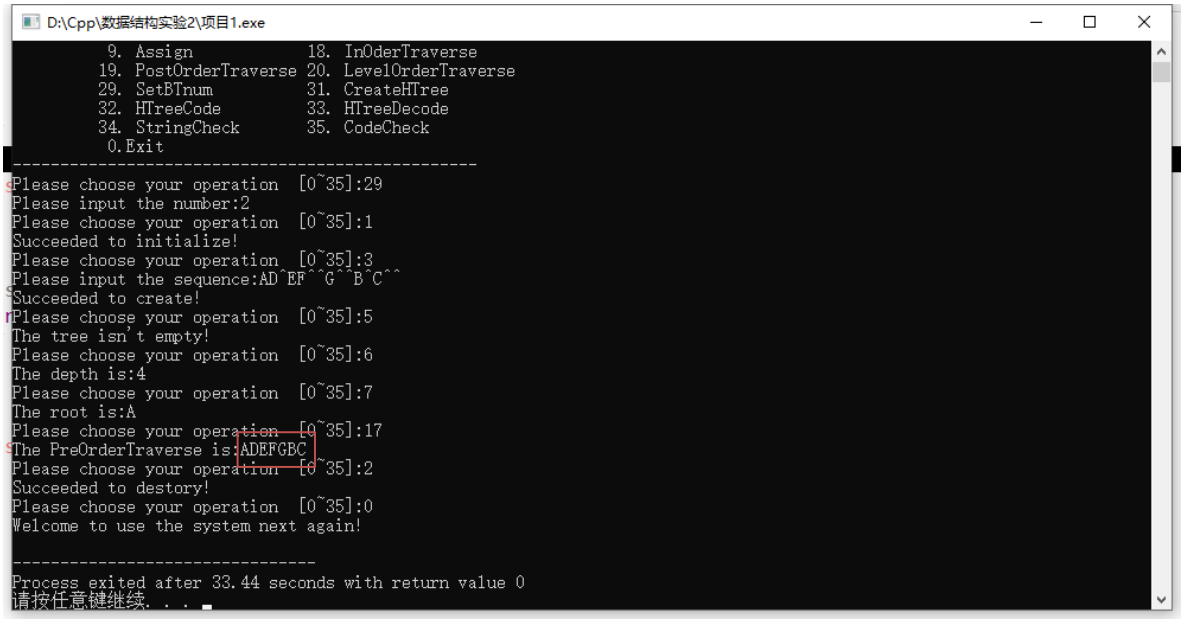


图 2.3-10 测试用例 4-1

测试用例 5：检验中序遍历。

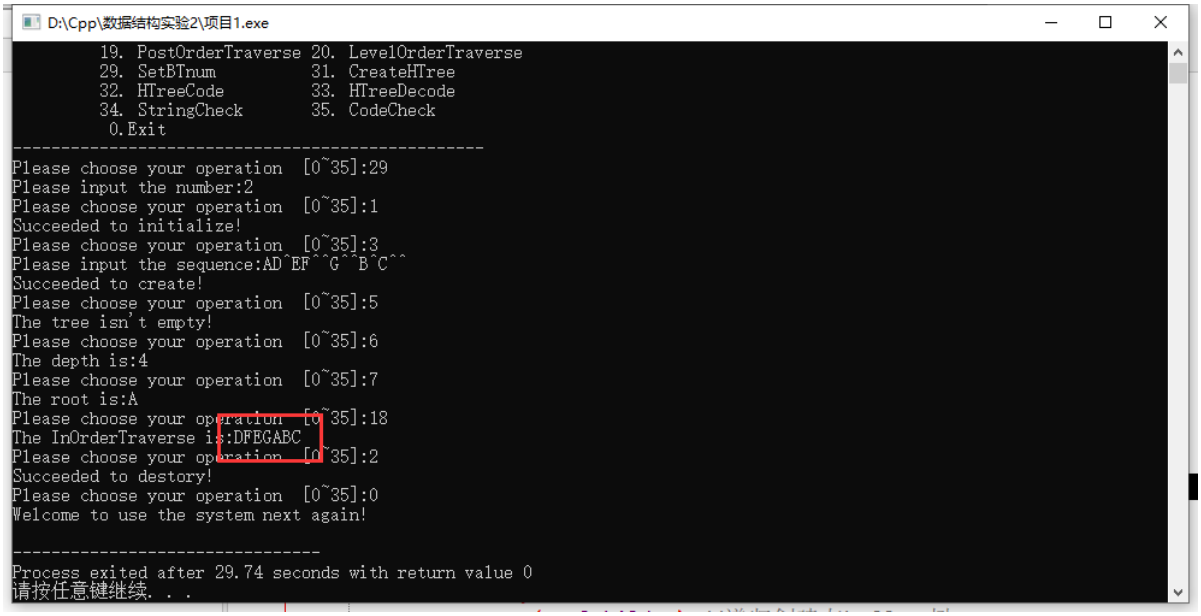


图 2.3-11 测试用例 5-1

测试用例 9：能测验是否正确建立哈夫曼树，以及哈夫曼编码和解码的正确性，故选择。首先选择第二棵二叉树建立哈夫曼树，检验树深度初步检测正确建立哈夫曼树，之后用编码解码校验函数，再调用解码编码校验函数，验证编码和解码的正确性。

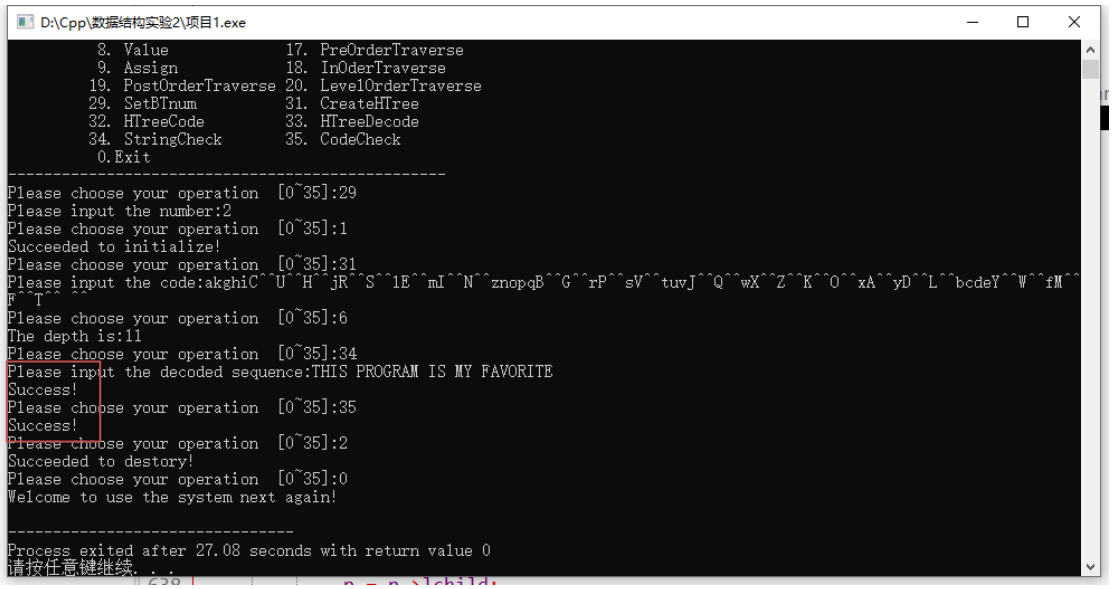


图 2.3-12 测试用例 9-1

文件操作内容测验：

测试用例 7：检验编码写入磁盘文件。

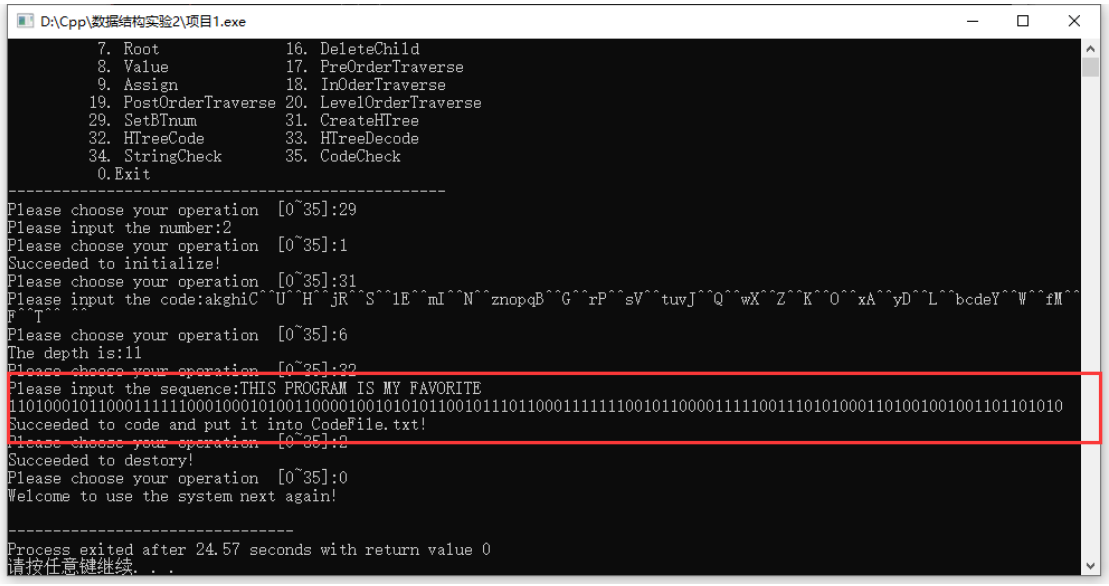


图 2.3-13 测试用例 7-1

打开 D 盘，发现确实有 CodeFile.txt，且内容就是“THIS PROGRAM IS MY FAVORITE”的编码：

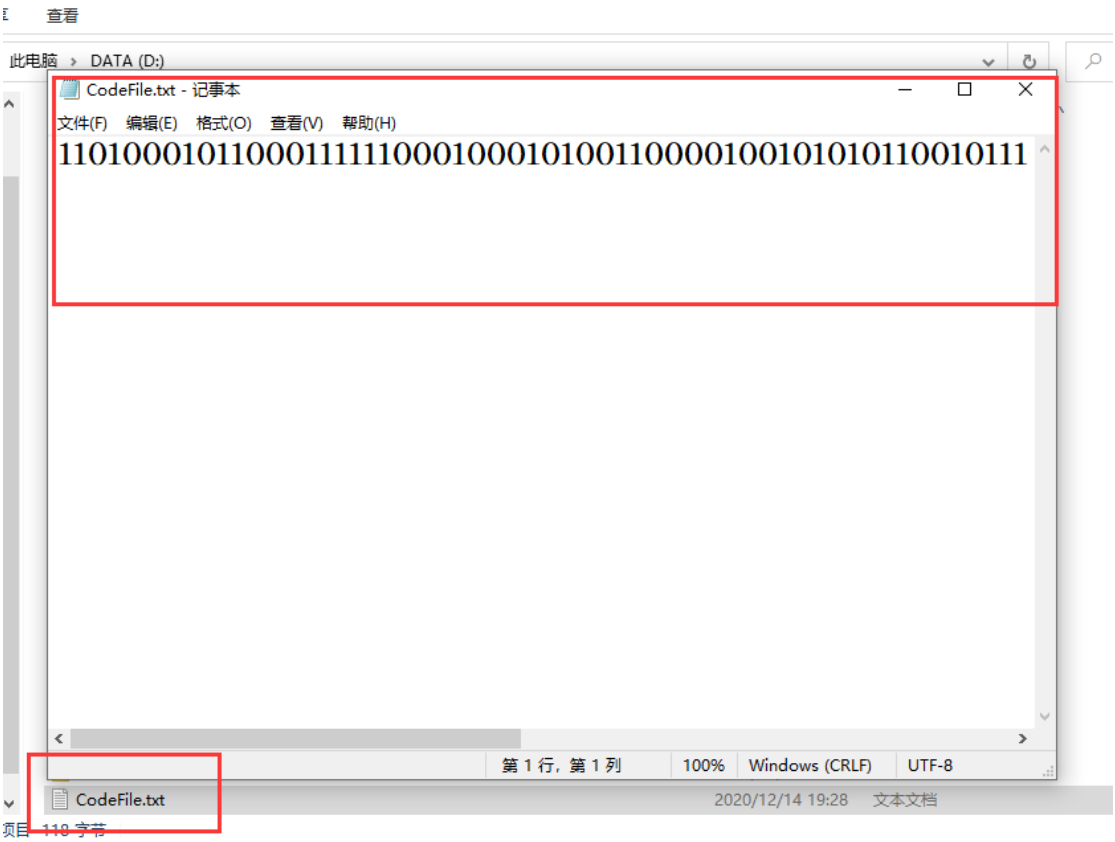


图 2.3-14 测试用例 7-2

测试用例 8：已经有测试用例 7 的 CodeFile.txt，来检测文件写入操作。

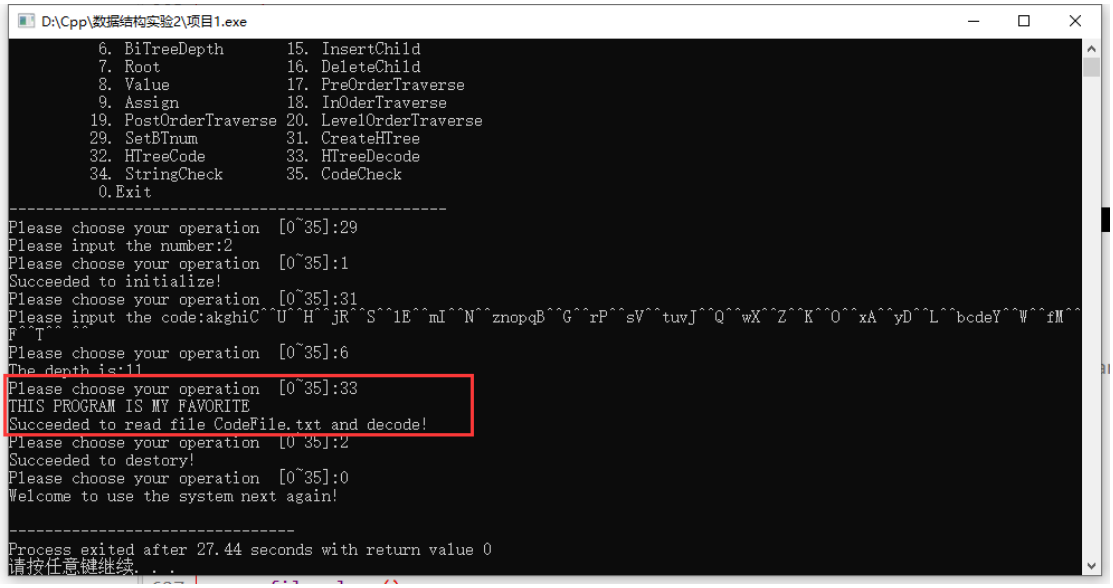


图 2.3-15 测试用例 8-1

2.4 实验小结

1.在本次实验中，因为树的层次遍历即广度优先遍历需要用到队列，在经过询问同学后，得知C++有已经写好的STL queue容易，里面有push()，pop()，front()等方法，拿来使用非常方便，一开始不知道容器的存在，走了很多弯路。

2.同queue，一开始不知道容器vector的存在，在创建前缀编码时费了很大的无用功，vector容器是顺序序列，动态数组，非常好用，有现成的push_back()，pop_back()方法，非常方便。

3.在进行编码解码时，需要用到文件操作，之前学习C++时，不知道C++的相关文件操作，通过查阅资料了解了ofstream和ifstream等文件流，在写这一部分时，遇到了很大的困难，当然与容器同样，在解决了之后也收获了许多。

4.本程序的模块划分较合理。主程序中进行交互，调用相应函数，函数间层次明显，感觉效率较高，huffman结构的创建，tmp和list数组的创建感觉比较简洁巧妙，但是有些函数进行了代码重写，而且全局变量的定义很冗杂，不够简洁。

5.在创建huffman树时，遇到了很大的问题，就是cin不能接收空格的输出。在网络上查阅了很多相关资料，得到cin与getline()的区别：getline()中的结束符，结束后，结束符不放入缓存区，cin的结束符，结束后，结束符还在缓存区，在使用 cin 后若要使用 getline() 必须要把前面cin遗留的结束符处理掉，解决方法为：在使用getline（）之前，加入一行getline（）来处理cin留下的结束符。同时学习了可以用cin>>noskipws来读取空白符。

6.算法的时空分析

1）若设多项式的长度为 n，DestroyBiTree()、CreateBiTree()、ClearBiTree()、BiTreeDepth()、Parent()、LeftChild()、RightChild()、LeftSbiling()、RightSbiling()、PreOrderTraverse()、InOrderTraverse()、PostOrderTraverse()、LevelOrderTraverse()、 CreateHtree()的时间复杂度为 O(n)，InitBiTree()、BiTreeEmpty()、Root()、Value()、Assign()、InsertChild()、DeleteChild()的时间复杂度为 O(1)，HTreeCode()、HtreeDecode()、 StringCheck()、 CodeCheck()的时间复杂度是O(n^2)，整体的时间复杂度不是很高，算法还算合适。

2）时间复杂度为O(n)和O(n^2)的算法空间复杂度基本都为O(n)，其余算法的空间复杂度均为O(1)。

7.本次实验使用了较多的递归，加深了我对递归的认识。

8.本次实验感觉二叉树本身做的难度不大，但是实现功能比较困难，主要是不仅仅要考虑二叉树，还要考虑堆栈和队列的使用，需要积累自己的经验，从理论到实践，攻克难关。本次实验采用数据抽象的程序设计方法，将程序划分为三个模块，加深了对二叉树的概念和基本运算的理解，熟练掌握了二叉树的逻辑结构和物理结构的关系，熟练掌握二叉树的基本运算的实现，并进行了扩展应用，学习掌握了程序框架的构建和通过键盘键入参数的方法，掌握了将各个基本运算功能模块组织在一个可执行系统中的方法，同时熟悉了C++语言的STL容器，输入输出流和文件操作，总体来说本次实验思路较清晰，调试顺利，收获很大，是一次良好的数据结构实验。

感谢老师和助教在本次实验中的帮助！

参考文献

[1] 严蔚敏等. 数据结构(C 语言版). 清华大学出版社

[2] Larry Nyhoff. ADTs, Data Structures, and Problem Solving with C++. Second Edition, Calvin College, 2005

[3] 殷立峰. Qt C++跨平台图形界面程序设计基础. 清华大学出版社,2014:192～197

[4] 严蔚敏等.数据结构题集(C 语言版). 清华大学出版社

指导教师评定意见

一、对实验报告的评语

二、对实验报告评分

评分项目（分值）	程序内容（36.8分）	程序规范（9.2分）	报告内容（36.8分）	报告规范（9.2分）	考勤（8分）	逾期扣分	合　计（100分）
得分							

附录 A 基于链式存储结构线性表实现的源程序

```
/*
Date:2020-11-1
Author:@徐子川-U201916322
Not For OnlineJudge
*/

#include<iostream>
#include<cstdlib>

using namespace std;

#define TRUE 1    //预定义
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2
#define LISTINCREMENT 10

typedef int Status;//状态
typedef char ElemType;//元素类型

typedef struct LNode {
    ElemType data;//元素类型 即多项式中的 x
    float coeff;//系数
    int index;//指数
    struct LNode* next;//指针域
}LNode, * LinkList;//LinkList=LNode*

LinkList list[11];//定义一个线性表数组便于管理多个线性表,多项式 10 用于减法函
//数
LinkList* pL;//二级指针指向链表的头指针
int number=0;//当前链表序号


Status InitaList(LinkList& L) {//初始化链表
    L = new LNode[sizeof(LNode)];//开辟空间
    if (!L)   exit(OVERFLOW);
    L->next = NULL;//置空
    return OK;
}
```

```
Status DestroyList(LinkList& L) {//销毁链表
    LNode* p;
    while (L) {
        p = L;
        L = L->next;
        delete[]p;//一个一个释放节点
    }
    L = NULL;//头指针置空
    return OK;
}

Status ClearList(LinkList L) {//清空表
    LNode* p, * q;
    p = L->next;
    L->next = NULL;//置空
    while (p) {//利用 p 和 q 释放节点
        q = p->next;
        delete[]p;
        p = q;
    }
    return OK;
}

Status ListEmpty(LinkList L) {//判断链表是否为空
    if (L->next)
        return FALSE;
    else
        return TRUE;
}

Status ListLength(LinkList L) {//获得链表的长度(除头结点)
    LNode* p;
    int i = 0;
    p = L->next;
    while (p) {
        i++;
        p = p->next;
    }
    return i;
}

Status GetElem(LinkList L, int i, ElemType &e) {//获得 i 位置的元素 用 e 返
回
    LNode* p;
```

```
    p = L->next;
    int j = 1;
    while (p && j < i) {//p 从第一个节点开始找 找到第 i 个
        p = p->next;
        j++;
    }
    if (!p || j > i)
        return ERROR;
    e = p->data;//此时 p 指向第 i 个结点
    return OK;
}

Status  LocateElem(LinkList L, ElemType e) {//获得元素 e 的位置
    LNode* p;
    int i = 1;
    p = L->next;
    while (p && p->data != e) {//p 从第一个开始找 找到数据为元素 e
        p = p->next;
        i++;
    }
    if (p)
        return i;//返回 e 的位置
    else
        return ERROR;
}

Status PriorElem(LinkList L, ElemType cur_e, ElemType& pre_e) {//获得元
//素 cur_e 的前驱
    LNode* p, * q;
    p = L;
    q = p->next;
    while (q->next && q->data != cur_e) {//利用 p 和 q 寻找,一直找到最后一个
//或 cur_e
        p = q;
        q = p->next;
    }
    if (q && p != L) {
        pre_e = p->data;//p 是 q 的前驱 返回 p 的数据到 pre_e
        return OK;
    }
    else
        return ERROR;
}
```

```
Status NextElem(LinkList L, ElemType cur_e, ElemType& next_e) {//获得元
//素 cur_e 的后继
    LNode* p, * q;
    p = L->next;
    q = p->next;
    while (q&& p->data != cur_e) {//利用 p 和 q 找到 p 为 cur_e 或 q 空为止
        p = q;
        q = p->next;
    }
    if (q) {//q 是 p 的后继 若 p 为最后一个节点 q 会空
        next_e = q->data;//返回 p 的后继 q 的数据到 next_e
        return OK;
    }
    else
        return ERROR;
}

Status ListInsert(LinkList L, int i, ElemType e) {//在第 i 个位置插入元素 e
    LNode* p, * q;
    int j = 0;
    p = L;
    while (p && j < i - 1) {//结束循环时 j=i-1 由 0 到 i-1 使 p 指向第 i-1 个
//结点
        p = p->next;
        j++;
    }
    if (!p || j > i - 1) return ERROR;
    q = new LNode[sizeof(LNode)];//开辟新节点 连接在 p 后面
    if (!q)  return OVERFLOW;
    q->data = e;
    q->next = p->next;
    p->next = q;

    return OK;
}

Status ListDelete(LinkList L, int i, ElemType &e) {//删除第 i 个元素 用
e//返回
    LNode* p, * q;
    int j = 0;
    p = L;
    while (p && j < i - 1) {//找到第 i-1 个结点
        p = p->next;
        ++j;
```

```
    }
    if (!(p->next) || j > i - 1)
        return ERROR;
    q = p->next;//p 是第 i-1 个结点  则 q 是第 i 个
    e = q->data;//用 e 返回
    p->next = q->next;//删除 q 指向的结点
    delete[]q;//释放
    return e;
}

Status ListTraverse(LinkList L) {//遍历链表
    LNode* p;
    if (!L)  return INFEASTABLE;
    p = L->next;
    if (!p)  return ERROR;
    while (p) {
        cout<<p->data;
        p = p->next;//输出每个元素
    }
    cout << endl;
    return OK;
}

Status setListnum(int num) {//设置当前链表工作序号

    if ((num > 0) &&(num < 11)) {
        pL = &(list[num - 1]);//pL 是全局变量
        return OK;
    }
    else
        return ERROR;
}

LNode* Last(LinkList L) {//返回多项式链表最后一个节点
    LNode* q;
    q = L;
    while (q->next) {
        q = q->next;
    }
    return q;
}

Status CheckIndex(float x, int y) {//检查是否有可以合并的项
```

```
    LNode* p, * q;//p 用来找有无相同的项 q 用来判断若有相同的项 相加后系数是
//否为 0
    q = *pL;
    p = (*pL)->next;
    while (p) {
        if (p->index == y) break;//如果当前项的指数是 y 跳出循环
        q = p;
        p = p->next;
    }
    if (!p) return INFEASTABLE;//如果没有相同指数的项 返回没有
    else {
        p->coeff += x;//找到相同指数 则系数相加
        if (p->coeff == 0) {//若相加后系数为 0 则删除节点
            q->next = p->next;
            delete[]p;
            return ERROR;
        }
        else return OK;
    }
}

Status PolySort(LinkList L) {//指数递减顺序排序多项式 冒泡排序思想 一轮比较
//找出一个最小的放在最后的位置
    int len = ListLength(L);//len 为多项式长度
    int i, j;
    LNode* p, * q, * pre;
    for (i = 1; i <len; i++) {//第 i 轮比较
        pre = L;//每轮比较都要从头开始  pre 是 p 的前驱 方便交换用
        p = pre->next;
        q = p->next;
        for (j = 0; j < len-i; j++) {//第 i 轮第 j+1 次比较
            if (q) {
                if (p->index < q->index) {
                    pre->next = q;
                    p->next = q->next;
                    q->next = p;//交换 p 节点与 q 节点
                    p = pre->next;
                    q = p->next;//交换 p 指针与 q 指针位置 以便进行下一次比较
                }
                pre = pre->next;//每比较完一次指针后移
                p = p->next;
                q = q->next;
            }
        }
```

```
    }
    return OK;
}

Status CreateList(int listnum) {//序号 listnum 建立一个多项式
    LNode* p, * q;
    int i = 1, n = 0;//i 表示第 i 项 n 表示一共有 n 项
    cout<<"Please input the quantity of item:";
    cin >> n;
    q = *pL;
    float x = 1;
    int y;
    while (x && i <= n) {
        cout<<"please input the coefficient of NO."<<i<<" item:";
        cin >> x;
        cout<<"please input the index of NO."<<i<<" item:";
        cin>>y;
        //x 是第 i 项的系数，y 是指数
        if (!x) {
            i++;
            continue;//如果系数是 0，直接跳到下一个循环输入下一项
        }
        Status flag = CheckIndex(x, y);//检查原多项式是否有可以合并的项 有
//就合并
        if (flag == INFEASTABLE) {
            //如果没有可以合并的 就建立新的节点
            p = new LNode[sizeof(LNode)];
            p->coeff = x;//系数
            p->index = y;//指数
            p->next = NULL;//下一个节点为空
            q = Last(*pL);//令 q 指向最后一个节点 方便插入
            q->next = p;//把 p 接到链表尾
        }
        i++ ;
    }
    if (ListLength(*pL) == 0) {//没有项 该多项式为 0
        p = new LNode[sizeof(LNode)];
        p->coeff = 0;
        p->index = 0;
        p->next = NULL;
        q->next = p;//q 是当前链表头结点，下一项是 0，再下一项空，整个多项式
//为 0
    }
    else PolySort(*pL);//多项式不为 0，按递减顺序排序
```

```
    return OK;
}

Status ShowList(int listnum) {//显示多项式 注意系数为 0，1，-1 和指数为 0 和
//1 的情况 分开讨论
    LNode* p;
    p = list[listnum - 1];
    if (!p || !(p->next)) return ERROR;
    p = p->next;//先考虑第一项
    if (p->coeff > 0) {
        if (p->coeff == 1) {//如果系数是 1 可以忽略
            if (p->index == 1) cout << "x";//如果指数是 1 可以忽略
            else if (p->index == 0) cout << "1";//如果指数是 0 输出 1
            else cout << "x^" << p->index;
        }
        else {
            if (p->index == 1) cout << p->coeff << "x";//如果指数是 1 可
//以忽略
            else if (p->index == 0) cout << p->coeff;//如果指数是 0 直接
//输出系数
            else cout << p->coeff << "x^" << p->index;
        }
    }
    else if (p->coeff == 0) cout << "0";//整个多项式为 0 输出 0
    else {
        if (p->coeff == -1) {//负数情况同上
            if (p->index == 1) cout << "-x";
            else if (p->index == 0) cout << "-1";
            else cout << "-x^" << p->index;
        }
        else {
            if (p->index == 1) cout << p->coeff << "x";
            else if (p->index == 0) cout << p->coeff;
            else cout << p->coeff << "x^" << p->index;
        }
    }
    p = p->next;
    while (p) {//与第一项不同
        if (p->coeff > 0) {//如果系数正 需要补符号"+" 其余同上
            if (p->coeff == 1) {
                if (p->index == 1) cout << "+x";
                else if (p->index == 0) cout << "+1";
                else cout << "+x^" << p->index;
            }
```

```
            else {
                if (p->index == 1) cout << "+" << p->coeff << "x";
                else if (p->index == 0) cout << "+" << p->coeff;
                else cout << "+" << p->coeff << "x^" << p->index;
            }
        }
        else {
            if (p->coeff == -1) {
                if (p->index == 1) cout << "-x";
                else if (p->index == 0) cout << "-1";
                else cout << "-x^" << p->index;
            }
            else {
                if (p->index == 1) cout << p->coeff << "x";
                else if (p->index == 0) cout << p->coeff;
                else cout << p->coeff << "x^" << p->index;
            }
        }
        p = p->next;
    }
    cout << endl;
    return OK;
}

Status AddList(int a, int b, int c) {//a 和 b 多项式相加  结果储存在 c 里
    LNode* pa, * pb, * pc, * qa, * qb, * qc;
    ClearList(list[c - 1]);//先清空 c
    qa = list[a - 1];
    pa = qa->next;
    qb = list[b - 1];
    pb = qb->next;
    qc = list[c - 1];
    while (pa && pb) {//a 和 b 都没有遍历完
        pc = new LNode[sizeof(LNode)];
        if (pa->index > pb->index) {//从指数大的项开始加
            pc->coeff = pa->coeff;
            pc->index = pa->index;
            qc->next = pc;
            qc = qc->next;
            pc->next = NULL;
            pa = pa->next;
        }
        else if (pa->index < pb->index) {//从指数大的项开始加
            pc->coeff = pb->coeff;
```

```
            pc->index = pb->index;
            qc->next = pc;
            qc = qc->next;
            pc->next = NULL;
            pb = pb->next;
        }
        else {
            pc->coeff = pa->coeff + pb->coeff;
            if (pc->coeff == 0) delete[]pc;//如果相加为 0 释放该节点
            else {
                pc->index = pa->index;
                qc->next = pc;
                qc = qc->next;
                pc->next = NULL;
            }
            pa = pa->next;
            pb = pb->next;
        }
    }
    while (pa) {//多项式 b 先遍历结束
        pc = new LNode[sizeof(LNode)];//开辟新节点 和 a 此时结点相同 接在链
//表 c 后面
        pc->coeff = pa->coeff;
        pc->index = pa->index;
        qc->next = pc;
        qc = qc->next;
        pc->next = NULL;
        pa = pa->next;
    }
    while (pb) {//a 先遍历结束
        pc = new LNode[sizeof(LNode)];//开辟新节点 和 b 此时结点相同 接在链
//表 c 后面
        pc->coeff = pb->coeff;
        pc->index = pb->index;
        qc->next = pc;
        qc = qc->next;
        pc->next = NULL;
        pb = pb->next;
    }
    if (!(list[c - 1]->next)) {//如果操作后 c 不存在项
        pc = new LNode[sizeof(LNode)];
        pc->coeff = 0;
        pc->index = 0;
        pc->next = NULL;
```

```
        list[c - 1]->next = pc;
    }
    if(ListLength(list[c-1])!=1){//如果链表 c 长度不是 1
    qc = list[c - 1];
    pc = qc->next;
    while (pc) {//若多项式中有 0 去除该节点
        if (pc->coeff == 0) {
            qc->next = pc->next;
            delete[]pc;
            break;
        }
        else {
            qc = pc;
            pc = pc->next;
        }
    }
    }
    return OK;
}

Status MinusList(int a, int b, int c) {//多项式 a 和 b 相减，存在 c 里
    LNode* p, * q, * pb;
    setListnum(10);//10 暂存加了负号的多项式 b
    InitaList(*pL);
    q = *pL;
    pb = list[b - 1]->next;
    while (pb) {//将 b 添加负号，其余不变
        p = new LNode[sizeof(LNode)];
        p->coeff = -(pb->coeff);
        p->index = pb->index;
        p->next = NULL;
        q->next = p;
        q = q->next;
        pb = pb->next;
    }
    AddList(a, 10, c);//减法化为加法
    DestroyList(*pL);//销毁 10
    setListnum(c);
    return OK;
}

int main() {
    pL = NULL;//二级指针置空
    int op = 1;//op 是输入的序号
```

```
    int i = 0, a, b, c = 0;//i 是输入的变量，abc 是多项式序号
    int num = 0;//链表序号
    ElemType e = 0, cur_e = 0, pre_e = 0, next_e = 0;//元素，当前元素，前驱，后继
        system("cls");
        cout<<endl<<endl;
        printf("      Menu for Linear Table On Sequence Structure \n");
        printf("-------------------------------------------------\n");
        printf("          1. IntiaList       7. LocateElem\n");
        printf("          2. DestroyList     8. PriorElem\n");
        printf("          3. ClearList       9. NextElem \n");
        printf("          4. ListEmpty      10. ListInsert\n");
        printf("          5. ListLength     11. ListDelete\n");
        printf("          6. GetElem        12. ListTrabverse\n");
        printf("         19. SetListnum     21. CreateList\n");
        printf("         22. ShowList       23. AddList\n");
        printf("         24. MinusList       0. Exit\n");
        printf("-------------------------------------------------\n");
    while (op) {
        printf("Please choose your operation  [0~24]:");
        cin >> op;
        switch (op) {
        case 1:
            InitaList(*pL);
            cout<<"The initialization of the list succeeded!\n";
            break;
        case 2:
            DestroyList(*pL);
            cout<<"The destrction of the list succeeded!\n";
            break;
        case 3:
            ClearList(*pL);
            cout<<"The clear of the list succeeded!\n";
            break;
        case 4:
            if (ListEmpty(*pL) == TRUE)
                cout << "The list is empty!\n";
            else if (ListEmpty(*pL) == FALSE)
                cout << "The list isn't empty!\n";
            break;
        case 5:
            cout <<"The length of the list is:"<< ListLength(*pL) << endl;
            break;
```

```
        case 6:
            cout<<"Please input the location:";
            cin >> i;
            if (GetElem(*pL, i, e) == OK) cout<<"The element is:"<<e<<e
ndl;
            else  cout << "NoElem" << endl;       //不存在则打印 NoElem
            break;
        case 7:
            cout<<"Please input the element:";
            cin >> e;
            if (LocateElem(*pL, e) != ERROR)
                cout <<"The location is:"<<LocateElem(*pL, e) << endl;
            else
                cout << "NoElem\n";
            break;
        case 8:
            cur_e = e;//一直往前找前驱 要更改 cur_e
            if (PriorElem(*pL, cur_e, pre_e) == OK)
            {
                e = pre_e;//一直往前找前驱 要更改 cur_e
                cout << "The pre is:"<<pre_e << endl;
            }
            else
                cout << "NoElem\n";
            break;
        case 9:
            cur_e = e;//一直往后找后继 要更改 cur_e
            if (NextElem(*pL, cur_e, next_e) == OK)
            {
                e = next_e;//一直往后找后继 要更改 cur_e
                cout << "The next is:"<<next_e << endl;
            }
            else
                cout << "NoElem\n";
            break;
        case 10:
            cout<<"Please input the location:";
            cin >> i;
            cout<<"Please input the element:";
            cin >> e;
            ListInsert(*pL, i, e);
            cout<<"Succeeded to insert!\n";
            break;
        case 11:
```

```
        cout<<"Please input the location:";
        cin >> i;
        ListDelete(*pL, i, e);
        cout << e << endl;
        cout<<"Succeeded to delete!\n";
        break;
    case 12:
        ListTraverse(*pL);
        break;
    case 19:
        cout<<"Please input the number:";
        cin >> number;
        setListnum(number);
        cout<<"Succeeded to set the list number!\n";
        break;
    case 21:
        CreateList(number);
        cout<<"Succeeded to create the polynome!\n";
        break;
    case 22:
        cout<<"Please input the number:";
        cin >> num;
        ShowList(num);
        break;
    case 23:
        cout<<"Please input the list number of list1:";
        cin >> a;
        cout<<"Please input the list number of list2:";
        cin >> b;
        cout<<"Please input the list number of list3:";
        cin >> c;
        AddList(a, b, c);
        cout<<"Succeeded to add!\n";
        break;
    case 24:
        cout<<"Please input the list number of list1:";
        cin >> a;
        cout<<"Please input the list number of list2:";
        cin >> b;
        cout<<"Please input the list number of list3:";
        cin >> c;
        MinusList(a, b, c);
        cout<<"Succeeded to minus!\n";
        break;
```

```
        case 0:
            cout<<"Welcome to use this system again next!"<<endl;
            break;
        }//end of switch
    }//end of while
    return 0;
}
```

附录 B 基于二叉链表二叉树实现的源程序

```
/*
Date:2020-11-19
Author:@徐子川-U201916322
Not For OnlineJudge
*/
#include <iostream>//输入输出流
#include <queue>//队列
#include <vector>//向量
#include <string>//字符串
#include <fstream>//文件
#define OK 1    //预定义
#define TRUE 1
#define FALSE 0
#define ERROR 0
#define OVERFLOW -2

using namespace std;

typedef char ElemType;//定义元素数据类型
typedef int Status;//定义函数的状态
typedef struct BiTNode {//二叉树的结点
    ElemType data;//数据域
    struct BiTNode* lchild, * rchild;//左右孩子
}*BiTree,BiTNode;//BiTree==BiTNode*

typedef struct Huffman {//创建编码类型
    char ch;//ch 是要编码的字母
    string str;//str 是字母的编码 01 字符串
}Huffman;
//函数声明
void setBTnum(int num);
Status InitBiTree(BiTree &T);
void DestoryBiTree(BiTree &T);
void CreateBiTree(BiTree &T);
void ClearBiTree(BiTree &T);
Status BiTreeEmpty(BiTree T);
int BiTreeDepth(BiTree T);
Status Root(BiTree T);
ElemType Value(BiTree T, BiTNode e);
void Assign(BiTree T, BiTree& e, ElemType value);
ElemType Parent(BiTree &T, BiTree& e);
```

```
ElemType LeftChild(BiTree T, BiTree& e);//访问左右孩子和兄弟时，如果存在，
ElemType RightChild(BiTree T, BiTree& e);//则当前结点改变为所访问的结点，
ElemType LeftSibling(BiTree T, BiTree& e);//如果左右孩子或兄弟不存在，
ElemType RightSibling(BiTree T, BiTree& e); //则当前结点仍保持，不改变为空结点，但是输出显示为空指针^
Status InsertChild(BiTree& T, BiTNode* p, char LR, BiTree c);
Status DeleteChild(BiTree T, BiTNode* p, char LR);
void PreOrderTraverse(BiTree T);
void InOrderTraverse(BiTree T);
void PostOrderTraverse(BiTree T);
void LevelOrderTraverse(BiTree T);
void CreateHTree(BiTree& T, string s);
void HTreeCode(BiTree T);
void HTreeDecode(BiTree T);
void StringCheck();
void CodeCheck();

BiTree* tree;//双指针 tree 指向二叉树数组中的某个根节点指针
BiTree list[11];//二叉树数组
BiTree tmp[11];//当前二叉树结点 B 切换到二叉树 A 时 回到 A 的之前在用的节点 所以要用数组保存
Huffman huffman[100];//huffman 编码数组
int huffnum = 0;//创建前缀编码时用来给每个 Huffman 成员赋值数据和编码的变量
int n = 0;//二叉树数组的编号
string input;//创建 huffman 树的先序序列
int strcount = 0;//创建 huffman 树用来遍历先序序列的变量

string retcode;
string retdecode;
string def;//def 是要进行编码的字符串

string DecodeCheck;

/*  cin 与 getline()的区别：
    getline()中的结束符，结束后，结束符不放入缓存区；
    cin 的结束符，结束后，结束符还在缓存区；
    在使用 cin 后若要使用 getline() 必须要把前面 cin 遗留的结束符处理掉，
    解决方法为：在使用 getline（）之前，加入一行 getline（）来处理 cin 留下的结束符；*/
//主函数
int main() {
    int op = 1;//菜单选项
    char LR;//左右子树选项 L or R
```

```
    int num;//插入子树序号
    string decode;
    ElemType e,p,lc,rc,ls,rs;//e 在 value 函数用 之后分别是双亲，左孩子，右孩子，左兄弟，右兄弟

    for (int i = 0; i < 11; i++) {//先将二叉树数组与当前节点数组置空
        list[i] = NULL;
        tmp[i] = NULL;
    }
    string icode;
    ifstream ifile;
    tree = NULL;//二级指针置空
    system("cls");
    cout<<endl<<endl;

        printf("      Menu for Linear Table On Binary Tree    \n");
        printf("-------------------------------------------------\n");
        printf("          1. InitBiTree        10. Parent\n");
        printf("          2. DestroyBiTree     11. LeftChild\n");
        printf("          3. CreateBiTree      12. RightChild \n");
        printf("          4. ClearBiTree       13. LeftSibling\n");
        printf("          5. BiTreeEmpty       14. RightSibling\n");
        printf("          6. BiTreeDepth       15. InsertChild\n");
        printf("          7. Root              16. DeleteChild\n");
        printf("          8. Value             17. PreOrderTraverse\n")
;
        printf("          9. Assign            18. InOderTraverse\n");
        printf("         19. PostOrderTraverse 20. LevelOrderTraverse\n
");
        printf("         29. SetBTnum          31. CreateHTree\n");
        printf("         32. HTreeCode         33. HTreeDecode\n");
        printf("         34. StringCheck       35. CodeCheck\n");
        printf("          0.Exit                              \n");
        printf("-------------------------------------------------\n");

    while (op) {//进入菜单
        printf("Please choose your operation  [0~35]:");
    cin >> op;  //输入选项
    switch (op) {
        case 29://设置当前要工作的二叉树序号
            cout<<"Please input the number:";
            cin >> n;
            setBTnum(n);
            break;
```

```
case 1:
    InitBiTree(*tree);
    cout<<"Succeeded to initialize!\n";
    break;//初始化二叉树
case 2:
    DestoryBiTree(*tree);
    cout<<"Succeeded to destory!\n";
    break;//销毁二叉树
case 3:
    cout<<"Please input the sequence:";
    CreateBiTree(*tree); //创建二叉树
    tmp[n] = *tree;//设置当前节点为该二叉树根节点
    cout<<"Succeeded to create!\n";
    break;
case 4:
    ClearBiTree(*tree);
    cout<<"Succeeded to clear!\n";
    break;//同销毁二叉树
case 5:
    if(BiTreeEmpty(*tree))
        cout<<"The tree is empty!"<<endl;
    else
        cout<<"The tree isn't empty!"<<endl;
    break;
case 6:
    cout<<"The depth is:"<<BiTreeDepth(*tree)<<endl;
    break;//求树的深度
case 7:
    Root(*tree);
    break;//输出根节点
case 8:
    cout<<"The value is:"<<Value(*tree, *tmp[n])<<endl;
    break;//输出当前节点的值
case 9:
    cout<<"Please input the element:";
    cin >> e;
    Assign(*tree, tmp[n], e);//给当前节点赋值 e
    break;
case 10:
    p=Parent(*tree, tmp[n]);//求当前节点双亲 并指向双亲
    cout << "The parent is:"<<p << endl;
    break;
case 11:
    lc = LeftChild(*tree, tmp[n]);//求当前节点左孩子 并指向左孩子
```

```
            cout << "The leftchild is:"<<lc << endl;
            break;
        case 12:
            rc = RightChild(*tree, tmp[n]);//求当前节点右孩子 并指向右孩子
            cout << "The right child is:"<<rc << endl;
            break;
        case 13:
            ls = LeftSibling(*tree, tmp[n]);//求当前节点左兄弟 并指向左兄
弟
            cout << "The leftsilbling is:"<<ls << endl;
            break;
        case 14:
            rs = RightSibling(*tree, tmp[n]);//求当前节点右兄弟 并指向右兄
弟
            cout << "The rightsibling is:" <<rs << endl;
            break;
        case 15:
            cin >> LR >> num;//插入子树 选择往左还是右插
            InsertChild(*tree, tmp[n], LR, list[num]);
            cout<<"Succeeded to insert!\n";
            break;
        case 16:
            cin >> LR;//销毁子树 选择销毁左还是右子树
            DeleteChild(*tree, tmp[n], LR);
            cout<<"Succeeded to delete!\n";
            break;
        case 17:
            cout<<"The PreOrderTraverse is:";
            PreOrderTraverse(*tree);
            cout << endl;
            break;//先序遍历
        case 18:
            cout<<"The InOrderTraverse is:";
            InOrderTraverse(*tree);
            cout << endl;
            break;//中序遍历
        case 19:
            cout<<"The PostOrderTraverse is:";
            PostOrderTraverse(*tree);
            cout << endl;
            break;//后序遍历
        case 20:
            cout<<"The LevelOrderTraverse is:";
            LevelOrderTraverse(*tree);
```

```
            cout << endl;
            break;//层序遍历
        case 31:
            cout<<"Please input the code:";
            getline(cin, input);//读入编码的先序序列
            getline(cin, input);//读入回车
            CreateHTree(*tree, input);//根据序列创建 huffman 树
            tmp[n] = list[n];//设置当前节点为该 huffman 树的根节点
            break;
        case 32:
            cout<<"Please input the sequence:";
            getline(cin, def);//输入字符串
            getline(cin, def);//读入回车
            HTreeCode(*tree);//编码
            cout<<"Succeeded to code and put it into CodeFile.txt!\n";
            break;
        case 33:
            HTreeDecode(*tree);//解码
            cout<<"Succeeded to read file CodeFile.txt and decode!\n";
            break;
        case 34:
            cout<<"Please input the decoded sequence:";
            getline(cin, def);//输入字符串
            getline(cin, def);//读入回车
            StringCheck();
            if(retdecode==def)
            cout<<"Success!"<<endl;
            else
            cout<<"Failure!"<<endl;
            break;
        case 35:
            ifile.open("d:\\CodeFile.txt");
            ifile>>icode;
            CodeCheck();
        //  if(icode==retcode)
        //  cout<<"Success!"<<endl;
        //  else
        //  cout<<"Failure!"<<endl;
            break;
        case 0:
            cout<<"Welcome to use the system next again!\n";
        break;
        }
}
```

```
    return 0;
}

//设置当前要工作的二叉树序号
void setBTnum(int num) {
    if (num > 0 && num < 11)
        tree = &list[num];
}

//初始化二叉树 T
Status InitBiTree(BiTree& T) {
    T = new BiTNode[sizeof(BiTNode)];
    if (!T) return OVERFLOW;
        T = NULL;
        return OK;
}
//后序递归销毁二叉树 T
void DestoryBiTree(BiTree& T) {
    if (T) {
        if (T->lchild)//递归销毁左子树
            DestoryBiTree(T->lchild);
        if (T->rchild)//递归销毁右子树
            DestoryBiTree(T->rchild);
        delete[]T;//最后销毁根节点
        T = NULL;
    }
}

//先序创建二叉树 T
void CreateBiTree(BiTree &T) {
    ElemType s;
    cin >> s;//输入先序序列
    if (s =='^') T = NULL;//如果为^ 则该节点为空
    else {
        if (!(T = new BiTNode[sizeof(BiTNode)])) exit(OVERFLOW);
        T->data = s;//如果不是^ 则分配空间 数据域是 s
        CreateBiTree(T->lchild);//递归创建左子树
        CreateBiTree(T->rchild);//递归创建右子树
    }
}

//清空二叉树 同销毁二叉树
void ClearBiTree(BiTree &T) {
    DestoryBiTree(T);
```

```
}

//判断二叉树是否为空
Status BiTreeEmpty(BiTree T) {
    if (!T)
        return TRUE;
    else
        return FALSE;
}

//返回二叉树 T 的深度
int BiTreeDepth(BiTree T) {
    if (T == NULL) return 0;//空树返回 0 递归的出口
    else {
        int l = BiTreeDepth(T->lchild);
        int r = BiTreeDepth(T->rchild);
        return (l > r) ? l + 1 : r + 1;//递归返回左右子树的深度最大值+1
    }
}

//输出根节点的值
Status Root(BiTree T) {
    if (!T) return ERROR;
    else{
        cout << "The root is:"<<T->data << endl;
        return OK;
    }
}

//返回 e 节点的值
ElemType Value(BiTree T,BiTNode e) {
    return e.data;
}

//给节点 e 赋值 value
void Assign(BiTree T,BiTree &e,ElemType value) {
    if (T)
        e->data = value;
}

//求当前节点双亲 并指向双亲 此处层序遍历 要用队列
ElemType Parent(BiTree &T, BiTree& e) {//T 是根节点指针 e 是当前节点
    queue<BiTree> q;//创建队列 q
    BiTree p = NULL;
```

```
    q.push(T);//根节点入队
    while (!q.empty()) {//队列不空
        p = q.front();
        q.pop();//p 指向队列弹出的第一个元素
        if (p->lchild== e){//若 p 的左孩子为节点 e
            tmp[n] = p;//指向双亲
            return p->data;//则 p 为 e 的双亲
        }
        else if (p->rchild== e) {//若 p 的右孩子为节点 e
            tmp[n] = p;//指向双亲
            return p->data;//则 p 为 e 的双亲
        }
        if (p->lchild) q.push(p->lchild);//左孩子不空则入队
        if (p->rchild) q.push(p->rchild);//右孩子不空则入队
    }
        //若没有双亲
    return '^';//返回^
}

//返回当前节点的左孩子
ElemType LeftChild(BiTree T, BiTree& e) {
    if(e->lchild==NULL)
    return '^';
    else {
        e = e->lchild;//左孩子不空就把当前节点移到左孩子
        return e->data;//返回左孩子
    }
}

//返回当前节点的右孩子
ElemType RightChild(BiTree T, BiTree& e) {
    if(e->rchild==NULL)
    return '^';
    else {
        e = e->rchild;//右孩子不空就把当前节点移到右孩子
        return e->data;//返回右孩子
    }
}

//返回当前节点的左兄弟
ElemType LeftSibling(BiTree T, BiTree& e) { //算法同层序遍历
    queue<BiTree> q;//要用队列
    BiTree p = NULL;
    q.push(T);
```

```
    while (!q.empty()) {
        p = q.front();
        q.pop();
        if ((p->rchild == e)&&(p->lchild!=NULL)) {//只是把层序遍历的输出
改为比较判断
            tmp[n] = p->lchild;//e 有左兄弟就返回
            return tmp[n]->data;
        }
        if (p->lchild) q.push(p->lchild);//左孩子不空则入队
        if (p->rchild) q.push(p->rchild);//右孩子不空则入队
    }
    return '^';//若没有左兄弟 返回^
}

//返回当前节点的右兄弟
ElemType RightSibling(BiTree T, BiTree& e) {//算法同层序遍历
    queue<BiTree> q;//要用队列
    BiTree p = NULL;
    q.push(T);
    while (!q.empty()) {
        p = q.front();
        q.pop();
        if ((p->lchild == e) && (p->rchild != NULL)) {//只是把层序遍历的
输出改为比较判断
            tmp[n] = p->rchild; //e 有右兄弟就返回
            return tmp[n]->data;
        }
        if (p->lchild) q.push(p->lchild);//左孩子不空则入队
        if (p->rchild) q.push(p->rchild);//右孩子不空则入队
    }

    return '^';//若没有右兄弟 返回^
}

//插入子树
Status InsertChild(BiTree &T, BiTNode* p, char LR, BiTree c){//c 无右孩子
    if (BiTreeEmpty(c))//若 c 空 返回错误
        return FALSE;
    if (p == NULL)//若 p 空 返回错误
        return FALSE;
    if (LR == 'L') {//若往左子树插
        if (p->lchild) {
            c->rchild = p->lchild;//若 p 有左孩子 则把 p 的左孩子往 c 的右子树
插
```

```
        }
        p->lchild = c;//再把 c 插到 p 的左孩子
    }
    else {//若往右子树插
        if (p->rchild) {
            c->rchild = p->rchild;//若 p 有右孩子 则把 p 的右孩子往 c 的右子树
插
        }
        p->rchild = c;//再把 c 插到 p 的右孩子
    }
    return OK;
}

//删除子树
Status DeleteChild(BiTree T, BiTNode* p, char LR){
    if (!T || !p) return FALSE;//若 T 和 p 有一个空 返回错误
    if (LR == 'R' && p->rchild) {//若删除右子树且右孩子不空
        DestoryBiTree(p->rchild);//删除右子树
        return OK;
    }
    else if (LR == 'L' && p->lchild) {//若删除左子树且左孩子不空
        DestoryBiTree(p->lchild);//删除左子树
        return OK;
    }
    return OK;
}

//前序递归遍历
void PreOrderTraverse(BiTree T){
    if (!T)//空就返回 递归出口
        return;
    else {
        cout << T->data;//遍历根节点
        PreOrderTraverse(T->lchild);//递归遍历左子树
        PreOrderTraverse(T->rchild);//递归遍历右子树
    }
    return;
}

//中序非递归遍历
void InOrderTraverse(BiTree T){
    BiTree st[1000];//建立一个栈
    int top = 0;//栈顶指针
    do {
```

```
        while (T) {//不管三七二十一 一直往左走
            if (top == 1000) exit(OVERFLOW);
            st[top++] = T;// 根节点一直入栈
            T = T->lchild;//一直往左走
        }
        if (top) {//栈不空
            T = st[--top];//弹出根节点
            cout << T->data;//遍历根节点
            T = T->rchild;//赋值右子树 进入下一次循环中序遍历右子树
        }
    } while (top||T);//若栈不空或 T 非空
}

//后序递归遍历
void PostOrderTraverse(BiTree T){
    if (!T)//空就返回 递归出口
        return;
    else {
        PostOrderTraverse(T->lchild);//递归遍历左子树
        PostOrderTraverse(T->rchild);//递归遍历右子树
        cout << T->data;//遍历根节点
    }
    return;
}

//利用队列层序遍历二叉树
void LevelOrderTraverse(BiTree T){
    if (!T) return;
    queue<BiTree> q;//创建队列 层序遍历 依次把按层序遍历的元素顺序依次入
队 然后依次出队
    BiTree p = NULL;
    q.push(T);//根节点入队
    while (!q.empty()) {//队列不空
        p = q.front();//p 指向队列弹出的第一个元素
        q.pop();
        cout << p->data;//输出 p 的数据
        if (p->lchild) q.push(p->lchild);//左孩子不空则左孩子入队
        if (p->rchild) q.push(p->rchild);//右孩子不空则右孩子入队
    }
}

Status isLeafNode(BiTree T) {//判断 T 是否为叶子节点
    if ((T->lchild == NULL) && (T->rchild == NULL))
        return TRUE;
```

```
    else
        return FALSE;
}

//利用 string 类 先序创建 huffman 树
void CreateHTree(BiTree &T,string s) {//算法类似先序创建普通二叉树
    if (s[strcount] == '^') {
        T = NULL;//若为^ 该节点为空
        strcount++;//进入下一个字符的判断
    }
    else {
        T = new BiTNode[sizeof(BiTNode)];
        T->data = s[strcount];//不空则分配节点 数据域为当前字符
        strcount++;
        CreateHTree(T->lchild,s);//递归创建左 huffman 树
        CreateHTree(T->rchild,s);//递归创建右 huffman 树
    }
}

//先序创建了一个 huffman 树 只需给每个叶子节点编码
void CreateBeforeCode(BiTree T, vector<ElemType> temp) {//用向量 temp 给
二叉树 T 创建前缀编码
    string s;//编码串
    if (isLeafNode(T)) {//如果是叶子节点 开始编码
        huffman[++huffnum].ch = T->data;//ch 存放数据
        for (int i = 0; i < temp.size();i++) {
            s += temp[i];//temp 中存放了从根节点到该叶子节点的编码
        }
        huffman[huffnum].str = s;//str 存放 ch 对应的编码
    }
    else {
        if (T->lchild) {//如果有左孩子
            temp.push_back('0');//向 temp 中存一个 0
            CreateBeforeCode(T->lchild, temp);//创建左孩子的编码
            temp.pop_back();//创建完后弹出 开始创建下一次 每递归一次弹出一
个
        }
        if (T->rchild) {//同左孩子
            temp.push_back('1');
            CreateBeforeCode(T->rchild, temp);
            temp.pop_back();
        }

    }
```

```
}

void HTreeCode(BiTree T) {//字符串编码函数
    vector<ElemType> temp;//创建编码用的向量

    int count = 0;//找 huffman 数组中与要编码的字符串的字符相同的数据，记下编
号
    int i, j;//循环变量
    CreateBeforeCode(T, temp);//先用 Huffman 树 T 创建编码

    ofstream ofile;
    ofile.open("d:\\CodeFile.txt");//将编码写入 D:\CodeFile.txt 中

    for ( i = 0; i < def.size(); i++) {//第一层循环  循环的是 def 的每个字符
        for ( j = 0; j < 100; j++) {//第二层循环  找与当前 def 中的字符相等
的 huffman 中的数据
            if (huffman[j].ch == def[i]) {
                count = j;//找到后记录  跳出第二层循环
                break;
            }
        }

        retcode+=huffman[count].str;
        ofile<<huffman[count].str;
        cout <<huffman[count].str;//输出该字符的编码
    }

    cout << endl;
    ofile.close();
}

void HTreeDecode(BiTree T) {//根据 huffman 树 T 进行解码
    int k = 0;//string 的索引
    string s;//读入的编码
    BiTree p = T;//不要直接用根节点

    ifstream ifile;
    ifile.open("d:\\CodeFile.txt");
    ifile>>s;

    while (s[k] != '\0') {//不管三七二十一  顺着编码一直找到叶子节点
        if (s[k] == '0')
            p = p->lchild;
        else
```

```
            p = p->rchild;
        k++;
        if (!isLeafNode(p))//不是叶子节点直接跳过本次循环
            continue;
        retdecode+=p->data;
        cout << p->data;//找到叶子节点后即是该编码对应的值  输出
        p = T;//回到根节点开始
    }
    ifile.close();
    cout << endl;
}
void StringCheck(){
    vector<ElemType> temp;//创建编码用的向量

    int count = 0;//找 huffman 数组中与要编码的字符串的字符相同的数据，记下编
号
    int i, j;//循环变量
    CreateBeforeCode(*tree, temp);//先用 Huffman 树 T 创建编码

    ofstream ofile;
    ofile.open("d:\\CodeFile.txt");//将编码写入 D:\CodeFile.txt 中

    for ( i = 0; i < def.size(); i++) {//第一层循环  循环的是 def 的每个字符
        for ( j = 0; j < 100; j++) {//第二层循环  找与当前 def 中的字符相等
的 huffman 中的数据
            if (huffman[j].ch == def[i]) {
                count = j;//找到后记录  跳出第二层循环
                break;
            }
        }

        ofile<<huffman[count].str;
    }
    ofile.close();

    int k = 0;//string 的索引
    string s;//读入的编码
    BiTree p = *tree;//不要直接用根节点
    ifstream ifile;
    ifile.open("d:\\CodeFile.txt");
    ifile>>s;

    while (s[k] != '\0') {//不管三七二十一  顺着编码一直找到叶子节点
        if (s[k] == '0')
```

```
            p = p->lchild;
        else
            p = p->rchild;
        k++;
        if (!isLeafNode(p))//不是叶子节点直接跳过本次循环
            continue;
        retdecode+=p->data;
        p = *tree;//回到根节点开始
    }
    ifile.close();


}
void CodeCheck(){
    int k = 0;//string 的索引
    string s;//读入的编码
    BiTree p = *tree;//不要直接用根节点
    ifstream ifile;
    ifile.open("d:\\CodeFile.txt");
    ifile>>s;
    while (s[k] != '\0') {//不管三七二十一 顺着编码一直找到叶子节点
        if (s[k] == '0')
            p = p->lchild;
        else
            p = p->rchild;
        k++;
        if (!isLeafNode(p))//不是叶子节点直接跳过本次循环
            continue;
        DecodeCheck+=p->data;
        p = *tree;//回到根节点开始
    }
    ifile.close();
    vector<ElemType> temp;//创建编码用的向量
    int count = 0;//找 huffman 数组中与要编码的字符串的字符相同的数据，记下编
号
    int i, j;//循环变量
    CreateBeforeCode(*tree, temp);//先用 Huffman 树 T 创建编码

    for ( i = 0; i < retdecode.size(); i++) {//第一层循环 循环的是
retdecode 的每个字符
        for ( j = 0; j < 100; j++) {//第二层循环 找与当前 def 中的字符相等
的 huffman 中的数据
            if (huffman[j].ch == DecodeCheck[i]) {
                count = j;//找到后记录 跳出第二层循环
```

```
                break;
            }
        }
        retcode+=huffman[count].str;
    }
    if(retcode==s){
        cout<<"Success!\n";
    }else{
        cout<<"Failure!\n";
    }

}
```