華中科技大学

课程实验报告

课程名称： 数据结构实验

实验名称： 哈夫曼编解码

专业班级：__ _

学　号：_ _____

姓　名：_ _____

指导教师：_____

报告日期：_2023.4.29_____

网络空间安全学院

评分表

评分项		总分	关卡一	关卡二	关卡三	总得分
实验	需求分析	5				
	系统设计	20				
	算法设计	20				
	实现函数设计	5				
	测试结果	10				
总结	复杂度分析	10				
	实验小结	10				
格式	段落缩进是否规范	2				
	图表编号是否规范	2				
	图片是否清晰	2				
	字号字体是否统一	2				
	其他	2				

注：总得分=关卡一得分×30%+关卡二得分×40%+关卡三得分×30%

目 录

1 统计字符频度

1.1 问题描述

本次实验内容“统计字符频度”。实现程序在读入字符内容后，根据字符串的信息建立频度单链表。当输入的字符先前出现过，就对先前的结点的进行频度的增加，若没出现过则创建新结点并将其插入频度链表中。之后对单链表进行遍历，按照频度的大小由高到低进行重新连接。

1.2 系统设计

整个程序分为创造频度链表（分为频度增加和创建新结点）与按频度高低进行排序两个部分。

1.2.1 数据结构设计

1.频度链表的抽象数据结构

下面是用频度链表存储字符串的数据结构

ADT ListNode {

数据对象： $L = \{a1, a2, \ldots, an\}$

数据关系：对于任意的$1 <= i < n$，ai 指向 ai+1

基本操作：

Create(& Ltail, c, &Lhead)

操作结果：构造一个空的指针链表，在本题中所有的结点考虑用一个链表进行管理。同时对输入字符创建一个新的结点。

Find(&L , c)

初始条件：链表 L 已存在，c 是需要查询的字符。

操作结果：遍历链表，寻找链表中是否有 c，返回判断结果 0 或 1 分别表示无或有。

Sort(& LN , Lcurrent)

初始条件：新创建的头指针 LN，Lcurrent 是需要判断权值大小的结点。

操作结果：将 L 链表中结点频度按从高到低插入节点 LN。

Print(LN)

初始条件：链表 LN 已存在

操作结果：遍历 LN，进行结果的打印

} **ADT** ListNode

2. 实际数据结构对应的结构体设计

根据需要设计对应的结构体如下：

```
typedef struct ListNode {
   char c;                  //结点的字符
   int frequency;           // 字符的频度
   struct ListNode* next;    // 结点的后继结点(对频度链表结点有效)
}ListNode;
```

1.2.2 执行流程设计

本程序的主要执行流程在于响应两个事件，“新建字符的频度链表事件”、“对频度链表进行排序事件”下面介绍这两个事件的执行流程。

1. 新建字符的频度链表事件

根据输入信息，完成判定是否需要建立头节点之后，不重不漏地建立字符串的频度链表。根据输入的字符以及之前已建立的字符对应进行频度的增加或者创立新的结点。

2. 对频度链表进行排序事件

根据已经建立好的频度链表，建立新的头指针，遍历频度链表，循环新的头指针对频度值进行判断，将频度按从高到低接入。

两个事件的执行流程如图 1-1 所示。

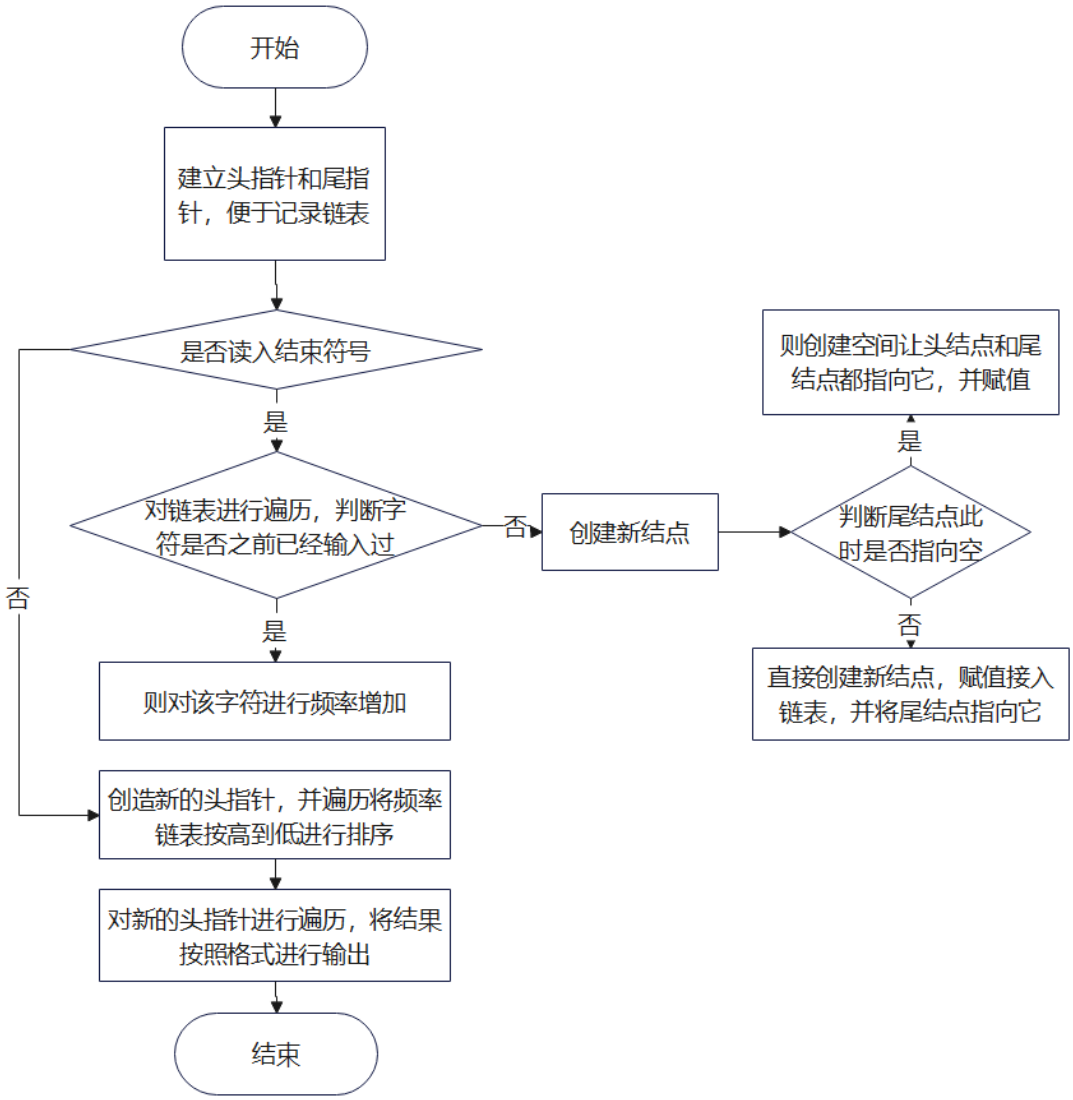

图 1-1 新建字符频度链表与对频度链表进行排序事件的完成流程

1.3 算法函数设计

表 1-1 主要函数及功能

函数名	主要功能
void Create(& Ltail, c, &Lhead)	创建一个新的节点并插入链表
int Find(&L , c)	遍历链表，寻找链表中是否有 c，返回 0 或 1 分别表示无或有
void Sort(& LN , Lcurrent)	构造新的头指针 LN，按频度从高到低插入节点
void Print(LN)	进行结果的打印

1.4 用例测试

因为本地对于字符串的输入结束标识会影响到相应字符数量的统计，本实验的用例测试使用头歌的自测程序进行验证。

使用头歌测试数据的结果如图 1-2，图 1-3 所示。

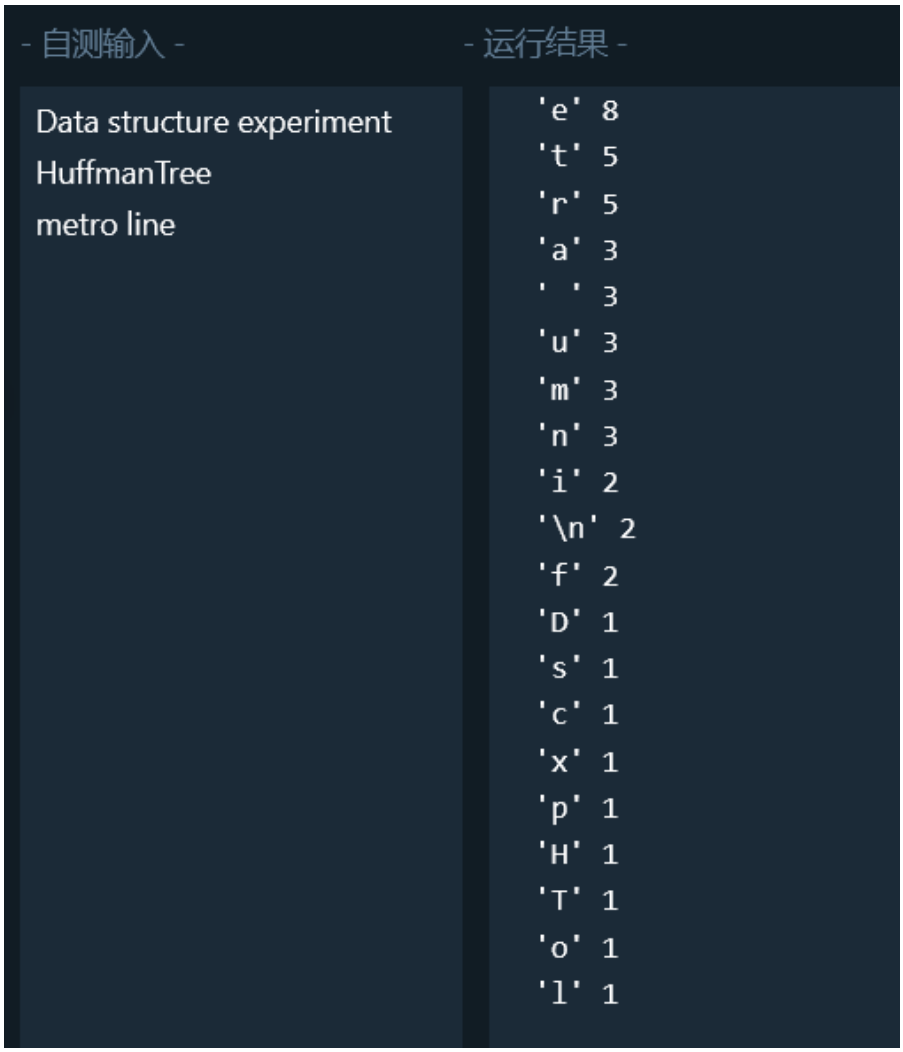

图 1-2 测试数据 1 在本地的运行结果

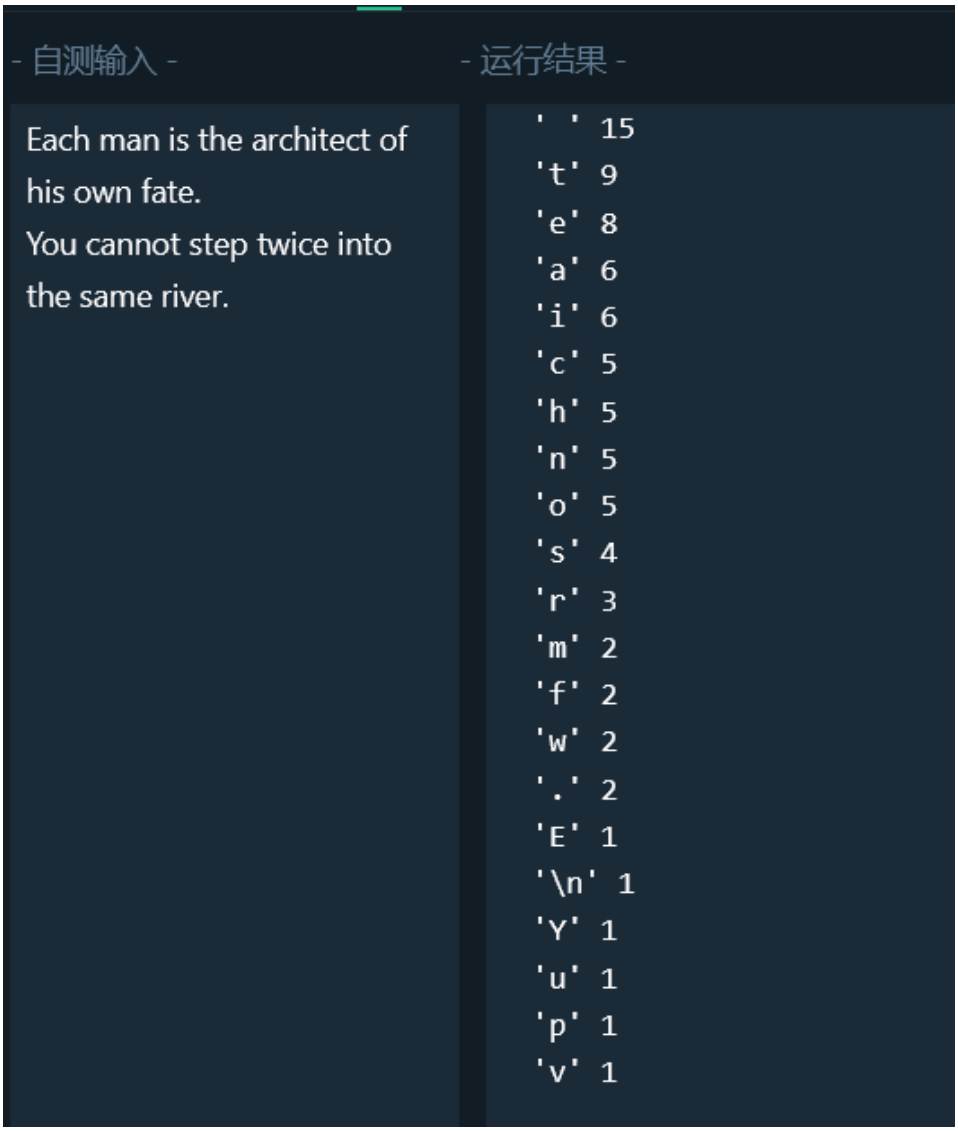

图 1-3 测试数据 2 在本地的运行结果

1.5 复杂度分析

在读入一个字符的信息后，要先查找字符是否存在，查找的最好情况是第一个结点就是要查找的字符，时间复杂度为O(1)，最差的情况是频度链表中均没有要查找的字符，则需要遍历该频度链表中所有结点。则在创建频度链表时间复杂度为$O(n)$，查找字符的总时间复杂度为$O(n\text{^}2)$。按照频率排序时，先是对频率表进行遍历，时间复杂度为$O(n)$，然后对于循环判断链表中频度的大小并接入，与上面查找字符同理，两者的总时间复杂度为$O(n\text{^}2)$。故总的时间复杂度为$O(n\text{^}2)$

而在建立空间的时候即为不同的字符创立了不同的空间，最好的情况是输

入的字符均相同，空间复杂度为$O(1)$，最差的情况是所有输入的字符都不同，空间复杂度为$O(n)$，同时建立了三个指针，除了链表，程序还使用了常量级别的额外空间。故总的空间复杂度为$O(n)$。

1.6 实验小结

本关整体难度不大，再明确了设计思路后，很顺利便完成了程序的编写。首先是忽视了开始的头指针进行空间的开辟会造成头指针与后面的链表断开，所以在函数里面增加了对于其为 NULL 时进行特殊处理，然后则是在代码中尝试用头接法和尾接法两种进行链表的连接，目的是为了加强自己对指针运用的理解，确实也花了较多的时间进行这部分的编写，然后对于如何减小时间复杂度也进行了思考，我写了一个用指针字符数组在输入字符对应 ASICC 码位置存储其位置，对于频度增加则无需查找直接通过指针字符数组进行调用，可以使时间复杂度降到$O(n)$，其实就是用空间换取时间，但是我还是保留了我最初循环查找的代码，更便于理解一点。

2 建立哈夫曼树并生成哈夫曼编码

2.1 问题描述

采用原址建立的方法，利用上一关得到的频度链表，将频度链表中的结点作为哈夫曼树中的结点，建立哈夫曼树。先序遍历哈夫曼树，得到各个叶子结点的哈夫曼编码，并按上一关中的顺序输出相应的哈夫曼编码。 并在最后一行输出哈夫曼树的带权路径长度，权重为字符的频度。

2.2 系统设计

首先读取字符构建链表，然后循环每次选取其中最小的两个结点，指向新创建的双亲结点，同时将双亲结点接入链表，则完成哈夫曼树的创建，然后通过递归先序遍历得到叶子结点的哈夫曼编码，接着将链表按照频率从高到低排序，最后输出字符及其频率和哈夫曼编码，以及哈夫曼树的带权路径长度。

2.2.1 数据结构设计

1.哈夫曼树的抽象数据结构

下面是构建哈夫曼树的数据结构，与二叉树相似

ADT HuffmanTree {

数据对象： D 是具有相同特性的数据元素的集合

数据关系：

若 D = Φ,则 R = Φ，称 HuffmanTree 为空二叉树

若 D ≠ Φ,则 R = $\{H\}$，H 有如下二元关系：

（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 - $\{root\}$ = $\{Dl, Dr\}$,且$Dl \cap Dr = \Phi$；

（3）若 Dl ≠ Φ,则 Dl 中存在唯一的元素 xl，$< root, xl >$∈H,且存在 Dl 上的关系$Hl \subset H$；

若 Dr ≠ Φ,则 Dr 中存在唯一的元素 xr，$< root, xr >$∈H,且存在 Dr 上的关系$Hr \subset H$；

H = $\{< root, xl >$ ， $< root, xr >$ ， $Hl, Hr\}$；

（4）$(Dl,\{Hl\})$是一棵符合本定义的二叉树，称为根的左子树，

$(Dr,\{Hr\})$是一棵符合本定义的二叉树，称为根的右子树；

基本操作：

minpoint(L)

初始条件：链表 L 已存在。

操作结果：在链表 L 中选择权值（频率）最小结点，返回最小结点的指针，或者返回 NULL。

Select(&L)

初始条件：链表 L 已存在。

操作结果：在链表 L 中选出权值最小的两个。

Insert_new(&Ltail, &parentnode)

初始条件：链表 L 已存在，新创建的双亲结点。

操作结果：将生成的双亲结点插入到频度链表中。

HTcode(&HTcurrent, char* cd)

初始条件：哈夫曼树 HT 已存在，用字符指针 cd 存编码。

操作结果：先序遍历哈夫曼树得到叶子结点的哈夫曼编码。

} **ADT** HuffmanTree

2. **实际数据结构对应的结构体设计**

根据需要设计对应的结构体如下：

```
typedef struct ListNode          //结点结构，哈夫曼树与频度链表共用
{
   char     c;                  //结点的字符
   int      frequency;          // 字符的频度
   char* code;          // 字符的编码(对哈夫曼树结点有效)
   struct ListNode* parent;          //结点的双亲结点(对哈夫曼树结点有效)
   struct ListNode* left;            //结点的左子树(对哈夫曼树结点有效)
   struct ListNode* right;           // 结点的右子树(对哈夫曼树结点有效)
   struct ListNode* next;            // 结点的后继结点(对频度链表结点有效)
} ListNode, HuffmanTree;
```

2.2.2 执行流程设计

本程序的主要执行流程在于响应两个事件，“建立哈夫曼树”、“得到叶子结

点的哈夫曼编码”下面介绍这两个事件的执行流程。

1. 建立哈夫曼树

输入的信息后，算法循环执行的每一次循环中，从还没有指定双亲指针的结点中选择频度最小的元素和频度次小的两结点。创建其二者的双亲结点并设置二者的双亲指针指向该结点，同时使双亲结点的左右孩指针指向两结点。之后将生成的双亲结点插入到频度链表中。循环执行至除根结点外的所有结点都具有双亲指针为止，哈夫曼树建立成功。

2. 得到叶子结点的哈夫曼编码

根据已经建立好的哈夫曼树，先序遍历哈夫曼树，通过递归函数得到各个叶子结点的哈夫曼编码，设置字符指针存哈夫曼编码，出递归赋值，然后按上一关中的顺序输出相应的哈夫曼编码。

事件一的执行事件二的执行分别如图 2-1、图 2-2 所示。

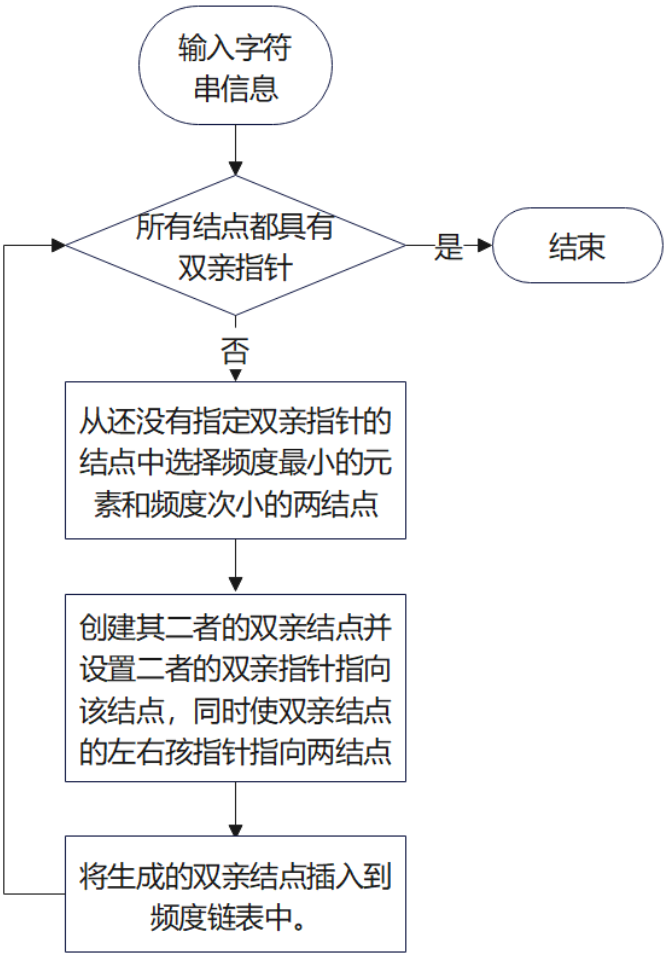

图 2-1 建立哈夫曼树的完成流程

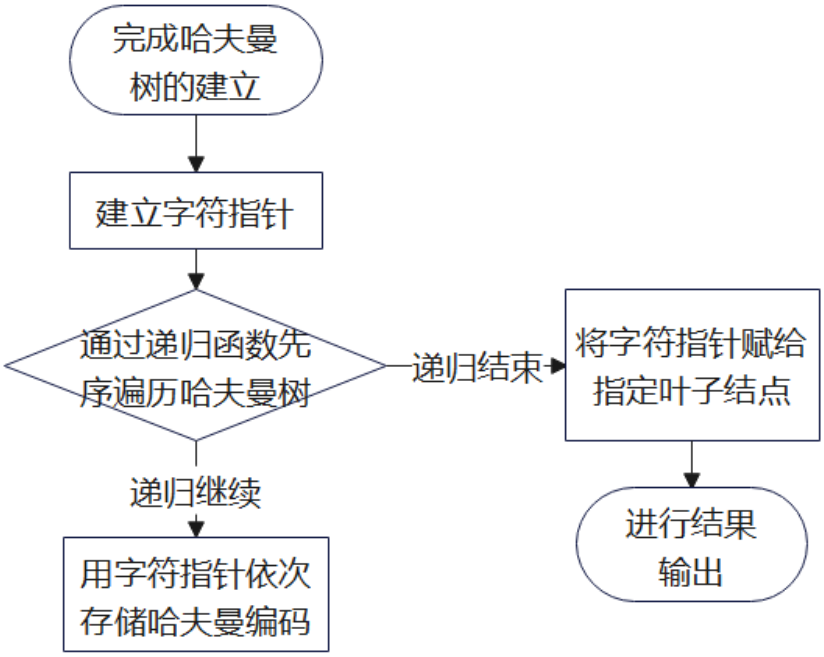

图 2-2 得到叶子结点的哈夫曼编码的完成流程

2.3 算法函数设计

表 2-1 主要函数及功能

函数名	主要功能
void stname(&L, c)	对结构体赋值操作
void Create(& Ltail, c, &Lhead)	创建一个新的节点并插入链表
int Find(&L, c)	遍历链表，寻找链表中是否有 c，返回 0 或 1 分别表示无或有
ListNode* minpoint(L)	在 L 中选择权值（频率）最小结点
ListNode* Select(&L)	在 L 中选出权值最小的两个
void Insert_new(&Ltail, &parentnode)	将生成的双亲结点插入到频度链表中
void HTcode(&HTcurrent, char* cd)	先序遍历哈夫曼树得到叶子结点的哈夫曼编码
void Sort(&LN, Lcurrent)	按频率从高到低插入节点

void Print(LN, num)	进行结果的打印

2.4 用例测试

使用头歌测试数据的结果如图 2-3，图 2-4 所示。

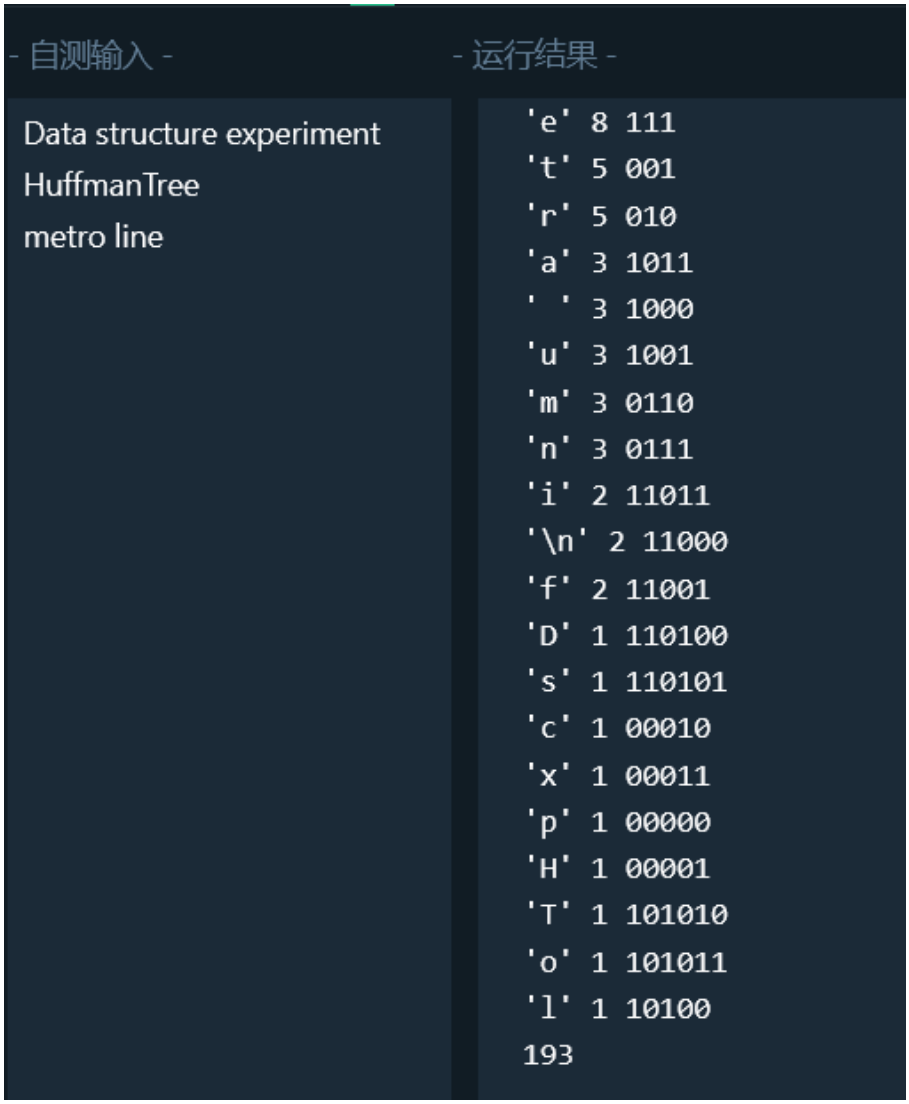

图 2-3 测试数据 1 在本地的运行结果

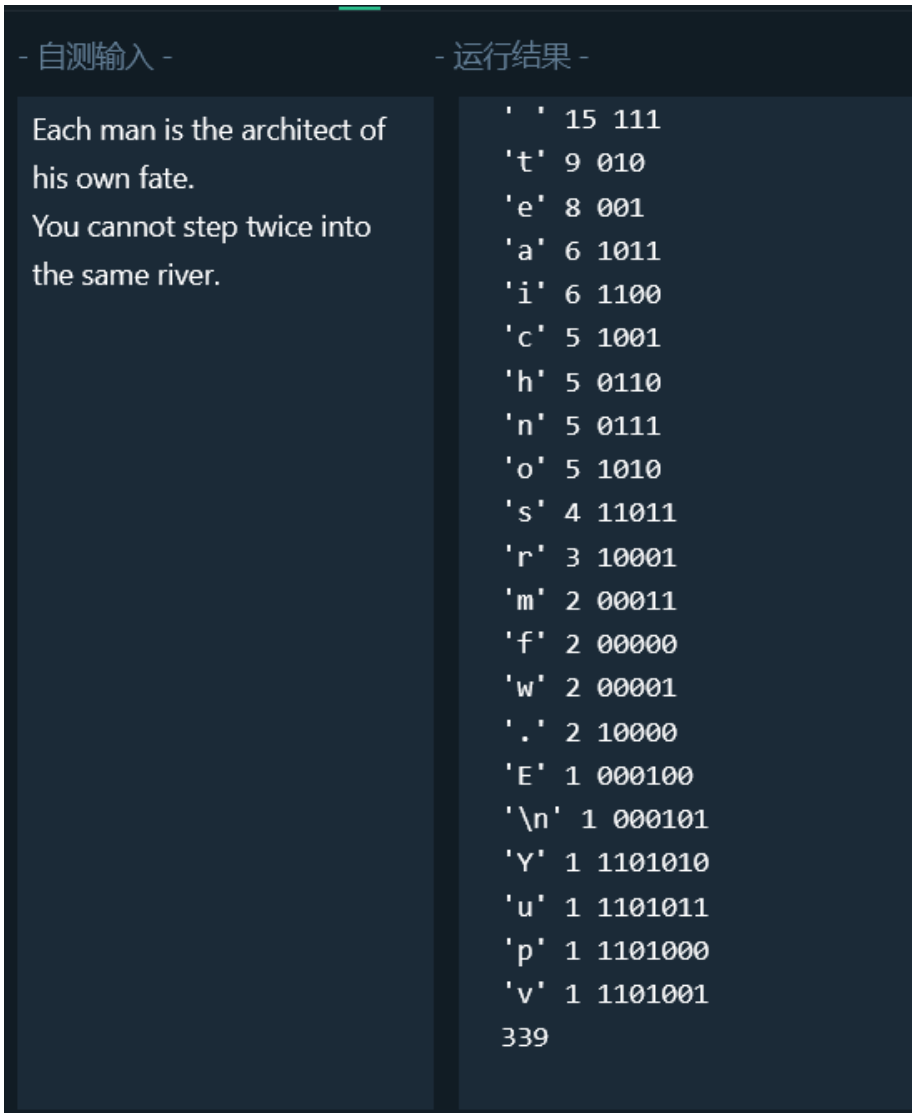

图 2-4 测试数据 2 在本地的运行结果

2.5 复杂度分析

创建链表的过程中，对于每个字符需要遍历一次链表来判断是否已经存在，因此最坏情况下需要遍历整个链表，时间复杂度为 $O(n)$。哈夫曼树的构建是通过不断选出权值最小的两个节点来进行的，每次选择时需要遍历整个链表以找到权值最小的节点，因此总共需要遍历 2n-1 次，时间复杂度为 $O(nlogn)$。得到叶子结点的哈夫曼编码的过程中，需要对哈夫曼树进行先序遍历，遍历的次数与叶子节点的个数成正比，因此时间复杂度为 $O(n)$。最后对链表按照频率从高到低排序，遍历链表的次数与链表的长度成正比，因此时间复杂度为 $O(nlogn)$。因此，总的时间复杂度为 $O(nlogn)$。

需要开辟动态存储空间来存储链表和哈夫曼树，空间复杂度为 $O(n)$。在得到叶子节点哈夫曼编码的过程中，需要递归调用，每次递归都会申请新的存储空间，因此空间复杂度为 $O(n)$。在排序过程中，需要开辟新的存储空间来存储排好序的链表，因此空间复杂度为 $O(n)$。因此，总的空间复杂度为 $O(n)$。

2.6 实验小结

本关任务对如何建立二叉树和遍历二叉树进行了更加深入的学习和探索，相比于伪码，真实地操作更加考验我们的思维严谨能力，经常会遇见一些很细节的问题导致花大量的时间去重新审查代码，进行一遍遍的运行和检查以及查阅一些资料去寻找问题所在，在完成代码的同时，还对 c 语言的一些基础知识重新进行巩固，例如对于结构体初始赋值不太清楚，结构体不会对元素自动赋值，需要自己进行赋值，否则程序会在运行中出现问题，也对指针和指针的地址的运用进行了更深入地了解和运用，同时代码过程中运用了大量地指针 malloc 开辟了一片空间，要记得将指针空间 free。虽然过程中一度很艰辛，但是整个过程下来仍是收获颇丰。

3 哈夫曼编码和解码

3.1 问题描述

编写编码程序和解码程序，利用上一关得到的哈夫曼编码对所给文本进行编码和解码。并输出编码后的文本，解码后的文本以及编码后文本的长度。

3.2 系统设计

整个程序分为对叶子结点进行编码（包括寻找叶子结点）与对叶子结点进行解码两个部分。

3.2.1 数据结构设计

1. 哈夫曼树的抽象数据结构

下面是用于用哈夫曼树编码和解码的数据结构

ADT HuffmanTree{

数据对象：$HT = \{$*二叉树*$\}$

数据关系：$HT = \{$*非叶子节点为权值，左右子树为*0、1 *的二叉树*$\}$

基本操作：

FindHTNode(HT, c)

初始条件：哈夫曼树 HT 已存在，c 是寻找的字符。

操作结果：查找二叉树的结点，返回结点指针。

Encode(HT, char* mag, len, char* code)

初始条件：哈夫曼树 HT 已存在。

操作结果：寻找叶子结点进行编码。

Decode(HT, char* code, n, char* text)

初始条件：哈夫曼树 HT 已存在。

操作结果：判断叶子结点进行解码。

} **ADT** HuffmanTree

3.2.2 执行流程设计

本程序的主要执行流程在于响应两个事件，“对叶子结点进行编码”、“对叶子结点进行解码”下面介绍这两个事件的执行流程。

1. 对于叶子结点进行编码

先创建字符指针 code，对于创建好的哈夫曼树进行先序遍历依次寻找其叶子结点，再将每一个叶子结点的编码连续存进字符指针开辟的空间，连成编码。

2. 对叶子节点进行解码

先创建字符指针 text，按 code 编码的顺序在哈夫曼树中，通过左右孩子指针的循环中寻找相应的叶子结点，并将其字符连续存入字符指针 text 开辟的空间，形成解码，最后进行输出。

两个事件的执行流程如图 3-1 所示。

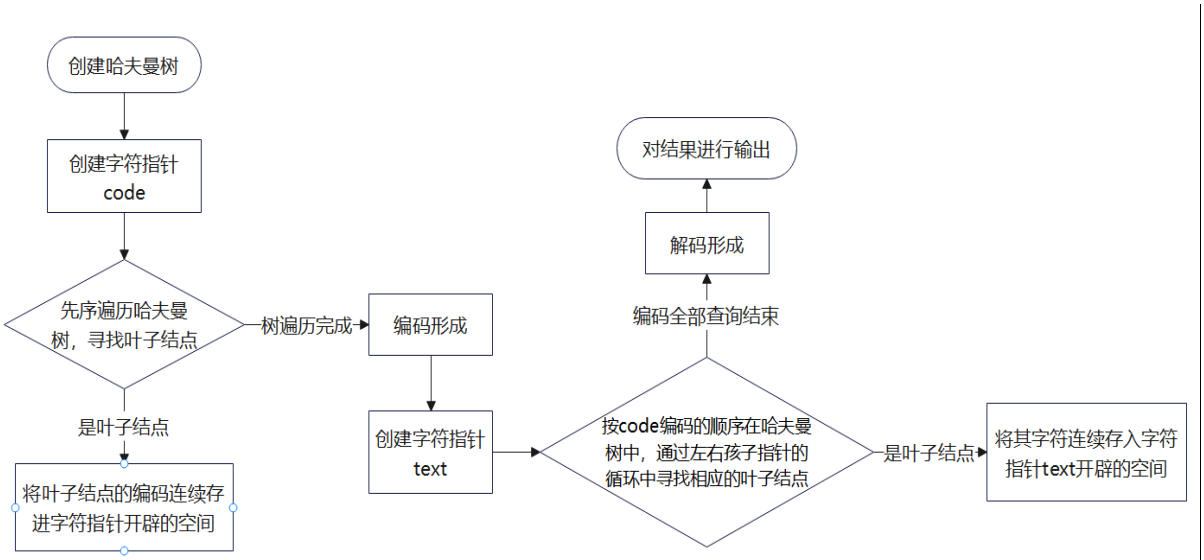

图 3-1 对叶子结点进行编码和解码事件的完成流程

3.3 算法函数设计

表 3-1 主要函数及功能

函数名	主要功能
void stname(&L, c)	对结构体赋值操作

void Create(& Ltail, c, &Lhead)	创建一个新的节点并插入链表
int Find(&L, c)	遍历链表，寻找链表中是否有 c，返回 0 或 1 分别表示无或有
ListNode* minpoint(L)	在 HT 中选择权值（频率）最小结点
ListNode* Select(&L)	在 HT 中选出权值最小的两个
void Insert_new(&Ltail, &parentnode)	将生成的双亲结点插入到频度链表中
void HTcode(&HTcurrent, char* cd)	先序遍历哈夫曼树得到叶子结点的哈夫曼编码
HuffmanTree* FindHTNode(HT, c)	查找二叉树的结点
void Encode(HT, char* mag, len, char* code)	寻找叶子结点进行编码
void Decode(HT, char* code, n, char* text)	判断叶子结点进行解码

3.4 用例测试

部分数据过长，为了结果的清晰截图时会省略一部分，下面是使用头歌测试数据的结果如图 3-2，图 3-3 所示。

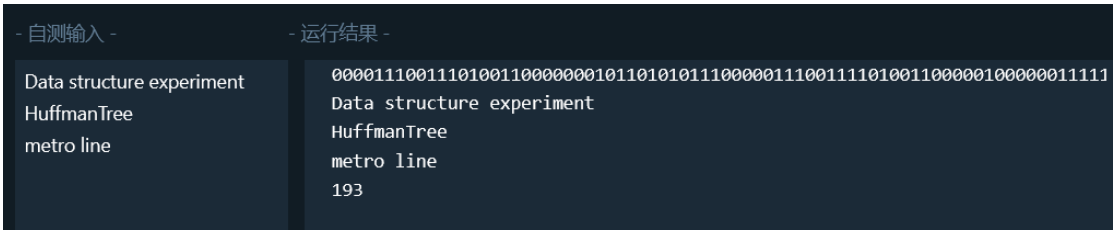

图 3-2 测试数据 1 在本地的运行结果

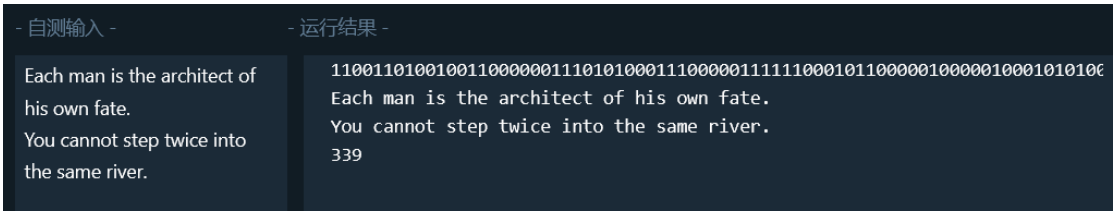

图 3-3 测试数据 2 在本地的运行结果

3.5 复杂度分析

创建链表和创建哈夫曼树的总时间复杂度由上一关可知为$O(nlogn)$。得到叶子结点的哈夫曼编码时间复杂度为$O(nlogn)$，因为需要对哈夫曼树进行先序

遍历，遍历每一个叶子节点，并将其编码存储起来。哈夫曼树中叶子节点的个数最多为 n，而每一个叶子节点至多被访问一次，所以需要进行 nlogn 次操作。编码和解码的时间复杂度均为$O(nlogn)$，因为编码和解码需要依据哈夫曼树进行操作，而哈夫曼树的高度不超过 logn，所以编码和解码的时间复杂度都是$O(nlogn)$。综上所述，此代码的时间复杂度为 O(nlogn)。

创建链表和创建哈夫曼树的总空间复杂度由上一关可知为$O(n)$。得到叶子结点的哈夫曼编码的空间复杂度为$O(n)$，需要分别为每个叶子节点存储其编码，编码长度不超过 logn。编码和解码的空间复杂度均为$O(n)$，需要分别为输入文本、编码后文本、解码后文本和哈夫曼编码存储空间。综上所述，此代码的空间复杂度为 O(n)。

3.6 实验小结

本关相比第二关的思维更加固定，就是多次对哈夫曼树进行遍历，然后寻找叶子结点或者按照编码对叶子结点的哈夫曼编码进行比对，我是专门再设置了一个函数进行编码，其实在开始建立哈夫曼编码的时候就可以直接进行编码，这里就不再赘述，这里还是以头歌上传的代码来进行分析。这说明代码常改常新，多次对写的代码进行审查会有新的思考和收获。

通过本次实验让我自己设计了 ADT，强化了我对于 ADT 的理解，明白了 ADT 对于程序编写规范性、复用性的重要意义。同时更加直观的体会到通过建立单链表进行数据的储存，同时对链表进行排序的操作。数据结构并不是一门仅停留于理论的课程，只有亲自去使用这些数据结构，编写相应的程序，才能真正掌握所学的知识。

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] 严蔚敏等.数据结构题集(C 语言版). 清华大学出版社

附录一 Huffman 编解码的操作实现

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include<string.h>

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2
//第一关
// 定义一个结构体,便于建立链表
typedef struct ListNode          //结点结构，哈夫曼树与频度链表共用
{
   char     c;                  //结点的字符
   int      frequency;          // 字符的频度
   char* code;          // 字符的编码(对哈夫曼树结点有效)
   struct ListNode* parent;          //结点的双亲结点(对哈夫曼树结点有效)
   struct ListNode* left;            //结点的左子树(对哈夫曼树结点有效)
   struct ListNode* right;            // 结点的右子树(对哈夫曼树结点有效)
   struct ListNode* next;             // 结点的后继结点(对频度链表结点有效)
}ListNode,HuffmanTree;

// 创建一个新的节点并插入链表
void Create(ListNode** tail, char ch,ListNode** head) {
   if (*tail == NULL) {//如果尾指针为 null 则为 head 开辟空间
      *head = (ListNode*)malloc(sizeof(ListNode));
      (*head)->c = ch;
      (*head)->frequency = 1;
      (*head)->next = NULL;
      *tail = *head;
   }
   else {  //否则在尾部插入指针，并将尾指针移位
      ListNode* new_node = (ListNode*)malloc(sizeof(ListNode));
      new_node->c = ch;
      new_node->frequency = 1;
      new_node->next = NULL;
      (*tail)->next = new_node;
      *tail = new_node;
   }
```

```
}

// 遍历链表，寻找链表中是否有 ch
int  Find(ListNode** head, char ch) {
    ListNode* current = *head;
    while (current != NULL) {
        if (current->c == ch) {
            current->frequency++;
            return 1;
        }
        current = current->next;
    }
    return 0;
}

// 按频度从高到低插入节点
void Sort(ListNode** head, ListNode* node) {
    if (*head == NULL || (*head)->frequency < node->frequency) {
        node->next = *head;
        *head = node;
        return;
    }

    ListNode* current = *head;
    while (current->next != NULL && current->next->frequency >= node-
>frequency) {
        current = current->next;
    }
    node->next = current->next;
    current->next = node;
}

//进行结果的打印
void Print(ListNode* head) {
    for (; head != NULL; head = head->next) {
        if (head->c == '\n') {
            printf("'\\n' %d\n", head->frequency); //将字符回车单独拿出来进行讨
论
        }
        else {
            printf("'%c' %d\n", head->c, head->frequency);
        }
    }
}
```

```
int main() {
    ListNode* head = NULL;
    ListNode* tail = NULL;
    char ch;

    // 读取字符并构建链表
    while ((ch=getchar()) != EOF) {
        if (!Find(&head, ch)) {
            Create(&tail, ch,&head);
        }
    }

    // 将链表按照频率从高到低排序
    ListNode* sorted_head = NULL;
    while (head != NULL) {
        ListNode* next = head->next;
        Sort(&sorted_head, head);
        head = next;
    }

    // 输出字符及其频率
    Print(sorted_head);
    return 0;
}

//第二关
//对结构体赋值操作
void stname(ListNode** list,char ch){
     (*list)->c = ch;
    (*list)->frequency = 1;
    (*list)->next = NULL;
    (*list)->parent = NULL;
    (*list)->left = NULL;
    (*list)->right = NULL;
}

// 创建一个新的节点并插入链表
void Create(ListNode** tail, char ch, ListNode** head) {
    if (*tail == NULL) {//如果尾指针为 null 则为 head 开辟空间
        *head = (ListNode*)malloc(sizeof(ListNode));
        stname(head,ch);
        *tail = *head;
    }
```

```
        else { //否则在尾部插入指针，并将尾指针移位
            ListNode* new_node = (ListNode*)malloc(sizeof(ListNode));
            stname(&new_node,ch);
            (*tail)->next = new_node;
            *tail = new_node;
        }
    }

    // 遍历链表，寻找链表中是否有 ch
    int  Find(ListNode** head, char ch) {
        ListNode* current = *head;
        while (current != NULL) {
            if (current->c == ch) {
                current->frequency++;
                return 1;
            }
            current = current->next;
        }
        return 0;
    }

    //在 HT 中选择权值（频率）最小结点
    ListNode* minpoint(ListNode* head)
    {
        ListNode* current = head;
        ListNode* min = NULL;  //用来存放 weight 最小且 parent 为 0 的结点

        //选出 weight 最小且 parent 为 0 的元素，并用 min 指向它
        for (; current != NULL; current = current->next)
        {
            if (min == NULL && (current->parent) == NULL) min = current;
                if(min != NULL && (current->parent) == NULL && current-
>frequency < min->frequency) {
                min = current;
            }
        }
        return min;
    }

    //在 HT 中选出权值最小的两个
    ListNode* Select(ListNode** head)
    {
        ListNode* new_node = (ListNode*)malloc(sizeof(ListNode)); //创建其二者
的双亲结点
```

```
    ListNode* s1 = NULL,*s2 = NULL; //储存最小的权值且 parent 为 0 的结点
    s1 = minpoint(*head);
    s1->parent = new_node;
    new_node->right = s1;
    new_node->frequency += s1->frequency;

    s2 = minpoint(*head);
    if(s2==NULL) return NULL;
    s2->parent = new_node; //设置二者的双亲指针指向该结点
    new_node->left = s2;   //使双亲结点的左右孩指针指向两结点
    new_node->frequency += s2->frequency;

    new_node->next = NULL;
    new_node->parent = NULL;
    return new_node;
}


//将生成的双亲结点插入到频度链表中
void Insert_new(ListNode** tail, ListNode* nnode) {
    (*tail)->next = nnode;
    *tail = nnode;
}

int ans;
//先序遍历哈夫曼树得到叶子结点的哈夫曼编码
void HTcode(HuffmanTree* current,char* cd) {
    if (current->left == NULL && current->right == NULL) {
      int lo=strlen(cd);
        current->code = (char*)malloc((lo + 1) * sizeof(char));
            strcpy(current->code, cd);
            ans+=(current->frequency)*lo;
    }
    else {
        strcat(cd, "0");
        HTcode(current->left, cd);
        cd[strlen(cd) - 1] = 0;
        strcat(cd, "1");
        HTcode(current->right, cd);
        cd[strlen(cd) - 1] = 0;
    }
}

// 按频率从高到低插入节点
```

```
void Sort(ListNode** head, ListNode* node) {
    if (*head == NULL || (*head)->frequency < node->frequency) {
        node->next = *head;
        *head = node;
        return;
    }

    ListNode* current = *head;
    while (current->next != NULL && current->next->frequency >= node->frequency) {
        current = current->next;
    }
    node->next = current->next;
    current->next = node;
}

//进行结果的打印
void Print(ListNode* head,int num) {
    for (int i = 0;i <= num ;++i, head = head->next) {
        if (head->c == '\n') {
            printf("'\\n' %d %s\n", head->frequency,head->code); //将字符回车单独拿出来进行讨论
        }
        else {
            printf("'%c' %d %s\n", head->c, head->frequency,head->code);
        }
    }
}

int main()
{
    ListNode* head = NULL;
    ListNode* tail = NULL;
    char ch;

    // 读取字符并构建链表
    while ((ch = getchar()) != EOF) {
        if (!Find(&head, ch)) {
            Create(&tail, ch, &head);
        }
    }

    //创建哈夫曼树
    ListNode* current = NULL;
```

```
    int num = 0;
    while (1)
    {
        current = Select(&head);
        if (current == NULL) break;
            num++;
        Insert_new(&tail, current);
    }

    //得到叶子结点的哈夫曼编码
    HuffmanTree* HT = tail;
    char* cd = (char*)malloc(num * sizeof(char));
    cd[0] = 0;
    HTcode(HT,cd);
    free(cd);


    // 将链表按照频率从高到低排序
    ListNode* sorted_head = NULL;
    for (int i = 0;i <= num;++i) {
        ListNode* next = head->next;
        Sort(&sorted_head, head);
        head = next;
    }

    // 输出字符及其频率和哈夫曼编码
    Print(sorted_head,num);
    printf("%d",ans);
    return 0;
}

//第三关
//对结构体赋值操作
void stname(ListNode** list,char ch){
     (*list)->c = ch;
    (*list)->frequency = 1;
    (*list)->next = NULL;
    (*list)->parent = NULL;
    (*list)->left = NULL;
    (*list)->right = NULL;
}
// 创建一个新的节点并插入链表
void Create(ListNode** tail, char ch, ListNode** head) {
    if (*tail == NULL) {//如果尾指针为 null 则为 head 开辟空间
```

```
            *head = (ListNode*)malloc(sizeof(ListNode));
            stname(head,ch);
            *tail = *head;
        }
        else { //否则在尾部插入指针，并将尾指针移位
            ListNode* new_node = (ListNode*)malloc(sizeof(ListNode));
            stname(&new_node,ch);
            (*tail)->next = new_node;
            *tail = new_node;
        }
    }

    // 遍历链表，寻找链表中是否有 ch
    int  Find(ListNode** head, char ch) {
        ListNode* current = *head;
        while (current != NULL) {
            if (current->c == ch) {
                current->frequency++;
                return 1;
            }
            current = current->next;
        }
        return 0;
    }

    //在 HT 中选择权值（频率）最小结点
    ListNode* minpoint(ListNode* head)
    {
        ListNode* current = head;
        ListNode* min = NULL;  //用来存放 weight 最小且 parent 为 0 的结点

        //选出 weight 最小且 parent 为 0 的元素，并用 min 指向它
        for (; current != NULL; current = current->next)
        {
            if (min == NULL && (current->parent) == NULL) min = current;
                if(min != NULL && (current->parent) == NULL && current-
>frequency < min->frequency) {
                min = current;
            }
        }
        return min;
    }

    //在 HT 中选出权值最小的两个
```

```
ListNode* Select(ListNode** head)
{
   ListNode* new_node = (ListNode*)malloc(sizeof(ListNode)); //创建其二者的双亲结点
   ListNode* s1 = NULL,*s2 = NULL; //储存最小的权值且 parent 为 0 的结点
   s1 = minpoint(*head);
   s1->parent = new_node;
   new_node->right = s1;
   new_node->frequency += s1->frequency;

   s2 = minpoint(*head);
   if(s2==NULL) return NULL;
   s2->parent = new_node; //设置二者的双亲指针指向该结点
   new_node->left = s2;  //使双亲结点的左右孩指针指向两结点
   new_node->frequency += s2->frequency;

   new_node->next = NULL;
   new_node->parent = NULL;
   return new_node;
}

//将生成的双亲结点插入到频度链表中
void Insert_new(ListNode** tail, ListNode* nnode) {
   (*tail)->next = nnode;
   *tail = nnode;
}

//先序遍历哈夫曼树得到叶子结点的哈夫曼编码
void HTcode(HuffmanTree* current,char* cd) {
   if (current->left == NULL && current->right == NULL) {
      current->code = (char*)malloc((strlen(cd) + 1) * sizeof(char));
          strcpy(current->code, cd);
   }
   else {
      strcat(cd, "0");
      HTcode(current->left, cd);
      cd[strlen(cd) - 1] = 0;
      strcat(cd, "1");
      HTcode(current->right, cd);
      cd[strlen(cd) - 1] = 0;
   }
}

//查找二叉树的结点
```

```
HuffmanTree* FindHTNode(HuffmanTree* HT, char ch) {
    if (HT == NULL) return NULL;
    else if (HT->left == NULL && HT->right == NULL) {
        if (HT->c == ch) return HT;
        else return NULL;
    }
    else {
        return FindHTNode(HT->left, ch) == NULL ? FindHTNode(HT->right,
ch) : FindHTNode(HT->left, ch);
        //返回找到的叶子结点
    }
}

//寻找叶子结点进行编码
void Encode(HuffmanTree* HT, char* mag,int mlen, char* code) {
    for (int i = 0; i < mlen; ++i)
    {
        HuffmanTree* p = FindHTNode(HT, mag[i]);
        if (p) strcat(code, p->code);
    }
}

//判断叶子结点进行解码
int len;
void Decode(HuffmanTree* HT, char* code, int n, char* text)
{
    HuffmanTree* current = HT;
    for (int i = 0; i < n; i++)
    {
        if (code[i] == '0') current = current->left;
        else current = current->right;

        //判断是否为叶子结点
        if (current->left == NULL && current->right == NULL)
        {
            text[len++] = current->c;
            current = HT;
        }
    }
}

int main()
{
   ListNode* head = NULL;
```

```
    ListNode* tail = NULL;
    char ch;

    // 读取字符并构建链表
    int mlen = 0;
    char mag[100000] = "";
    while ((ch = getchar()) != EOF) {
        mag[mlen++]=ch;
        if (!Find(&head, ch)) {
            Create(&tail, ch, &head);
        }
    }

    //创建哈夫曼树
    ListNode* current = NULL;
    int num = 0;
    while (1)
    {
        current = Select(&head);
        if (current == NULL) break;
            num++;
        Insert_new(&tail, current);
    }

    //得到叶子结点的哈夫曼编码
    HuffmanTree* HT = tail;
    char* cd = (char*)malloc(num * sizeof(char));
    cd[0] = 0;
    HTcode(HT,cd);
    free(cd);

     //进行编码和解码
     char* code=(char*)malloc(100000*sizeof(char));
     char* text=(char*)malloc(100000*sizeof(char));
     code[0]=0;
     text[0]=0;
     Encode(HT, mag, mlen, code);
     int clen=strlen(code);
     Decode(HT, code, clen, text);
     printf("%s\n", code);
     printf("%s\n", text);
     printf("%d", clen);
    return 0;
}
```