华 中 科 技 大 学

网络空间安全学院

本科：《数据结构实验》实验报告

姓　　名 邹涛声

班　　级 计算机类 2401 班

学　　号 U202410848

联系方式 13479743814

分　　数

评 分 人

网络空间安全学院

评 分 细 则 表

评分项		总分	得分	备注
实验	需求分析	5		问题描述部分，用自己的语言对本关需完成的内容加以陈述。 4-5 分：清晰简洁。 0-3：照抄任务书或内容很少。
	系统设计	20		包括数据结构设计、功能模块设计和系统功能流程图等内容。 15-20 分：各方面设计满足要求，思路逻辑阐述清晰。 8-14 分：缺少某方面的内容，或某方面有错误、描述很少。 0-7 分：缺少多个方面内容，或有多个方面明显错误。
	算法设计	20		对于关键函数中使用的算法和实现做简要描述。 15-20 分：关键函数算法描述合理、清晰，有必要的辅助图示。 8-14 分：关键函数算法有基本描述。 0-7 分：关键函数缺少合理描述，或逻辑有明显错误。
	函数设计	5		根据主要函数设计以表格形式进行描述。 4-5 分：函数设计合理，参数合理正确。 0-3 分：函数设计存在不合理，或缺少参数说明。
	测试分析	10		组织合理的测试样例进行测试，对测试结果进行分析。 7-10 分：头歌样例和自己构造的测试用例，对测试结果有分析和说明。 4-6 分：只有头歌样例的运行界面截图，简单的结果分析。 0-3 分：只有少量运行界面的截图，缺少结果的分析。
总结	复杂度分析	10		分析程序关键算法实现的时间复杂度与空间复杂度。 7-10 分：关键算法的复杂度分析正确，有必要的评价。 4-6 分：有基本的复杂度分析，或重点部分不突出。 0-3 分：存在明显错误的复杂度分析。
	实验小结	10		实验中遇到的问题、解决问题的过程及方法，实验心得。 7-10 分：问题、解决过程及收获，心得和思政方面内容。 4-6 分：缺少某方面的内容，或存在照搬网上的内容。 0-3 分：内容少，缺少多方面的内容。
格　　式		20		报告的格式规范，包括封面、文字和插图的排版效果等。 15-20 分：各方面规范，排版认真，格式一致。 8-14 分：插图内容看不清楚，文字大小不统一等。 0-7 分：存在明显的排版错误，不一致方面明显。
总　分		100		

目 录

1 统计字符频度

1.1 问题描述

本次实验内容是“Huffman 编码”。实现程序在读入一段英文文本后，建立相应的 Huffman 树并给出相应的 Huffman 编码与解码方案。第一部分是整个实验的基础，要求建立频度链表来统计输入的内容中各字符的频度。要求使用单链表来存储输入的内容中所含的字符，并通过算法对频度链表进行排序，从而实现字符的频度按从高到低的顺序排列。

1.2 系统设计

整个程序分为以下几个部分。第一，处理输入的英文文本，将其所有字符及其频度存入频度链表；第二，对频度链表按频度高低进行重新排列；第三，根据输出示例格式化输出频度链表内容。

1.2.1 数据结构设计

1. 频度链表的抽象数据结构

下面是用于存储各字符的频度链表的数据结构。

ADT List{

数据对象：$D = \{a_i | a_i \in ElemSet^{1},\ i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D,\ i = 1,2,\dots,n\}$

基本操作：

InitList(&L)

操作结果：构造一个空的单链表，在本关中即建立一个新的频度链表，L 即该链表的头结点。

Input(&L, *text)

初始条件：L 已存在，text 字符串用于存储英文文本。

[1] $ElemSet$为给定的元素集合，在本次实验中即所有可能出现在英文文本中的字符组成的集合，包括英文字母、标点符号、空格、换行符等，下同。

操作结果：根据输入的英文文本内容，将其中所含的字符及其频度存入L，并返回L中字符的个数。

SortList(&L, n)

初始条件：L已存在且非空，*n*为L中字符结点的个数。

操作结果：对L进行排序，在本关中即对频度链表按字符频度的高低进行重新排列。

PrintList(L)

初始条件：L存在、非空且按字符的频度非递增排列。

操作结果：遍历L，格式化输出L中的内容。

} **ADT** List

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
} ListNode;
```

1.2.2 执行流程设计

本关的关键执行流程在于两个步骤，一是“建立频度链表”，二是“频度链表排序”。下面简单介绍这两个步骤的执行流程。

1. 建立频度链表

根据输入信息，遍历全部英文文本。如果频度链表中已包含正在读取的字符，则只需将其频度加1；否则，需要初始化新的结点结构，并将其添加到频度链表中。算法的关键在于执行上述判断，其具体流程如图1-1所示。

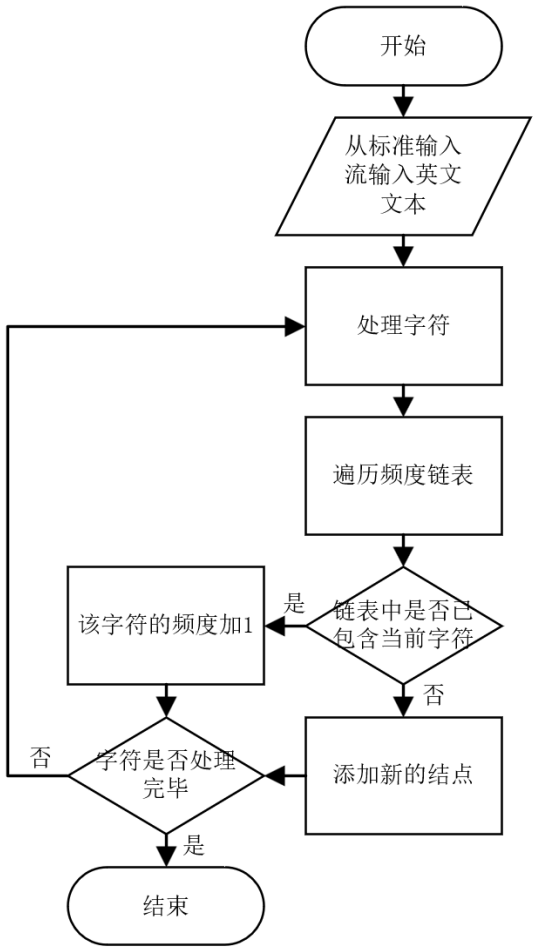

图 1-1 “建立频度链表”步骤的执行流程

2. 频度链表排序

对已经建立好的频度链表重新排序，使其按照字符频度非递增排列。排序的实现是这一步骤中最关键的算法，同时也是整个关卡中最为核心的部分。我在此使用了常用的冒泡排序，通过重复遍历待排序的频度链表，比较相邻两个字符结点的频度高低，若它们的顺序错误就交换它们，直到没有需要交换的元素为止。其具体执行流程如图 1-2 所示。

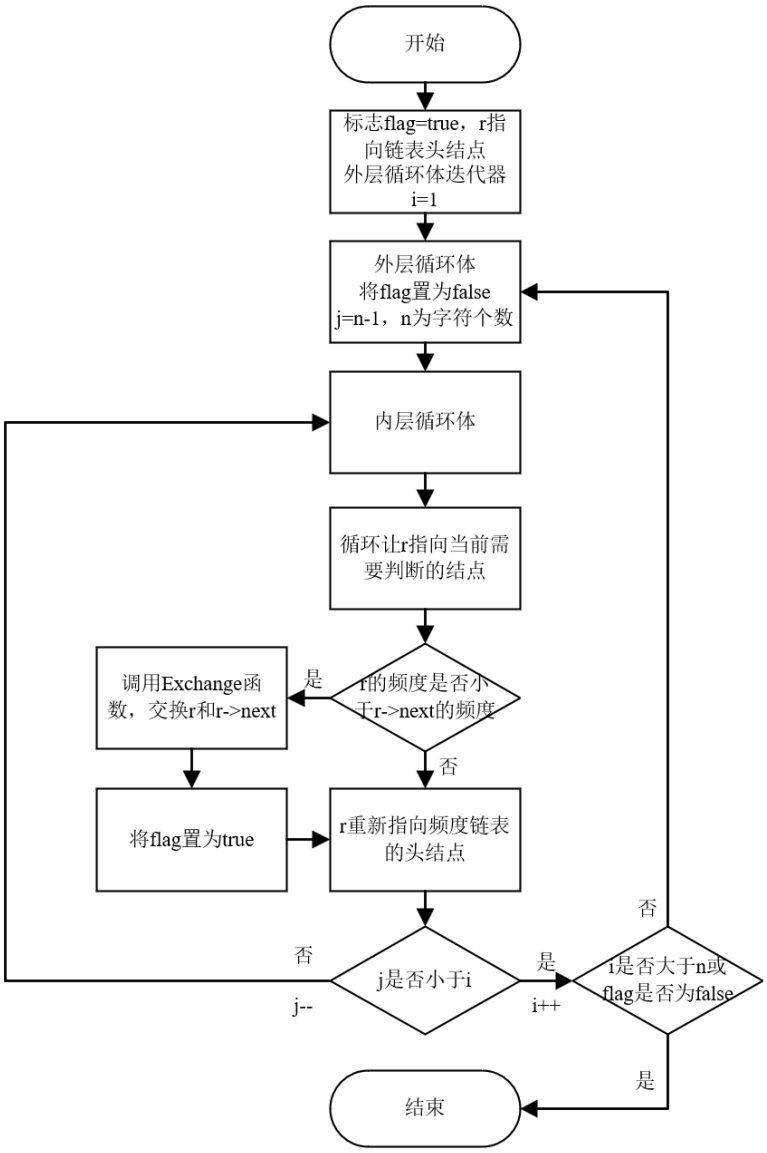

图 1-2 “频度链表排序”步骤的执行流程

本关涉及到的核心算法的关键代码将在附录一中呈现。

1.3 算法函数设计

表 1-1 本关主要函数及其功能

函数名	主要功能
ListNode* InitListNode()	建立一个空的频度链表，初始化头结点并将其返回

void Input(ListNode* head, char* x, int* n)	从标准输入流输入英文文本，head 指向频度链表的头结点，*x* 用于存储输入的英文文本，*n* 用于记录字符的个数
void Exchange(ListNode* head, ListNode* p)	在频度链表 head 中，交换 p 结点和 p 的前驱结点，辅助排序函数的实现
void SortList(ListNode* head, int n)	排序函数，对频度链表按字符频度高低重新排列，*n* 为字符的个数
void PrintList(ListNode* head)	格式化输出频度链表内容

1.4 用例测试

使用头歌测试数据 1 的运行结果如图 1-3 所示，使用“Do not go gentle into that good night.”作为自定义测试数据的运行结果如图 1-4 所示。在两个用例中，空格均是频度最高的字符，因此都被排在了表头。其余字符则根据频度高低有序排列，符合我所设计的算法的逻辑。

```
Each man is the architect of his own fate.
You cannot step twice into the same river.
' ' 15
't' 9
'e' 8
'a' 6
'i' 6
'c' 5
'h' 5
'n' 5
'o' 5
's' 4
'r' 3
'm' 2
'f' 2
'w' 2
'.' 2
'E' 1
'\n' 1
'Y' 1
'u' 1
'p' 1
'v' 1

D:\Program Files\Microsoft Visual Studio\source\repos\ConsoleApplication73\x64\Debug\Consol
ted with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->
ebugging stops.
Press any key to close this window . . .
```

图 1-3 头歌测试数据 1 在本地的运行结果

```
Do not go gentle into that good night.
' ' 7
'o' 6
't' 6
'n' 4
'g' 4
'e' 2
'i' 2
'h' 2
'D' 1
'l' 1
'a' 1
'd' 1
'.' 1
D:\Program Files\Microsoft Visual Studio\source\repos\ConsoleApplication73\x64\D
2448) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->
le when debugging stops.
Press any key to close this window . . .
```

图 1-4 自定义测试数据“Do not go gentle into that good night.”在本地的运行结果

1.5 复杂度分析

在我设计的程序中，时间复杂度的主要来源是 Input 函数、SortList 函数和 PrintList 函数。每当读入一个字符，需要遍历一次频度链表以完成查找与可能的插入操作，这一过程的时间复杂度为$O(n)$（n是链表中已存在的字符数）。假设输入的总字符数为m，则 Input 函数的总时间复杂度为$O(m \cdot n)$。SortList 函数中，我利用了冒泡排序算法对频度链表进行排序，每次排序需要进行嵌套循环，时间复杂度为$O(n^3)$。而 PrintList 函数遍历了整个频度链表并进行输出，因此其时间复杂度为$O(n)$。所以，整个程序的时间复杂度主要由 Input 和 SortList 函数所决定，为$O(m \cdot n + n^3)$。空间复杂度的主要来源是用于存储每个字符及其频度的频度链表，而程序运行时的辅助变量与占用的栈空间可以忽略不计，所以总空间复杂度为$O(m')$（m'为输入的不同字符数）。

1.6 实验小结

本关整体系统设计难度并不高。借助头歌平台中的“学习内容”，我迅速明确了编程目标，确定了所需设计的数据结构及关键算法。然而在实施过程中，我遇到了两个主要问题。

其一，头歌平台推荐的数据结构中，频度链表与后续关卡中的 Huffman 树结构共用。刚开始我对此理解困难，后来经过深入研究，选择暂时忽略结构体中与

Huffman 树相关的部分，最终得以完成 ADT 的设计。从后续关卡中，我发现这种数据结构“共用”的方式确实优势明显，能使程序代码更加简洁，而且不影响各关卡中不同 ADT 的分析设计，关键则是在于学会取舍。

其二，如何将换行符\n 存入字符数组成为我遇到的最大难题。我尝试了多种标准输入流，如 scanf、fgets，甚至 C++中的 cin.getline 等等，但它们均难以处理换行符输入，也无法很好地判断输入结束符“EOF”。在陷入困境后，我并没有选择放弃，而是通过大量搜索网络资料，最终确定使用 getchar() 函数逐字符处理，利用关键语句“while ((x[index] = getchar()) != EOF)”判断输入结束，成功解决了问题。

通过本次实验，我有以下两点深刻体会：第一，学以致用、灵活类比至关重要。在设计频度链表排序算法时，我曾经想过利用辅助顺序结构来实现，但这过于复杂，同时偏离了单链表的特性。于是，我联想到曾经学习过的冒泡排序算法，并经过我的改良，成功将其应用到了本关。 其二，“Practice makes perfect.” 上学期我所学习的编程课使用的是 C++语言，所以我对 C 语言的语法特性掌握不很熟练。而本关正让我通过实践加深了 C 语言中多个模板库函数的使用技巧，进一步巩固了 C 语言知识。

2 建立哈夫曼树并生成哈夫曼编码

2.1 问题描述

编写程序，利用上一关中得到的频度链表，建立相应的 Huffman 树，具体给出 Huffman 编码的实现方案，并得到各个字符的 Huffman 编码。

2.2 系统设计

本关共分为三个部分。首先，根据上一关中已得到的频度链表，建立相应的 Huffman 树；然后，根据 Huffman 树的结构，对频度链表中的各个字符进行编码；最后，根据输出示例，格式化输出各字符的编码信息。

2.2.1 数据结构设计

1. 频度链表的抽象数据结构

下面是用于存储各字符的频度链表的数据结构。

ADT List{

数据对象：$D = \{a_i | a_i \in ElemSet,\ i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D,\ i = 1,2,\dots,n\}$

基本操作：

InitList(&L)

操作结果：构造一个空的单链表，在本关中即建立一个新的频度链表，L 即该链表的头结点。

Input(&L, *text)

初始条件：L 已存在，text 字符串用于存储英文文本。

操作结果：根据输入的英文文本内容，将其中所含的字符及其频度存入 L，并返回 L 中字符的个数。

SortList(&L, n)

初始条件：L 已存在且非空，n 为 L 中字符结点的个数。

操作结果：对 L 进行排序，在本关中即对频度链表按字符频度的高低进行重新排列。

} **ADT** List

2. Huffman 树的抽象数据结构

由于 Huffman 树是一棵二叉树，于是我们可以借助二叉树的定义来设计 Huffman 树的抽象数据结构。

ADT BinaryTree{

数据对象：$D = \{a_i | a_i \in ElemSet,\ i = 1,2,\dots,n, n \geq 0\}$，

数据关系R：若$D = \phi$，则称为空二叉树；否则$R = \{H\}$，H是如下二元关系：

（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$D - \{root\} \neq \phi$，则存在$D - \{root\} = \{D_l, D_r\}$，且$D_l \cap D_r = \phi$；

（3）若$D_l \neq \phi$，则D_l中存在唯一元素x_1，$< root, x_1 > \in H$，且存在D_l上的关系$H_l \subset H$；若$D_r \neq \phi$，则D_r中存在唯一的元素x_r，$< root, x_r > \in H$，且存在D_r上的关系$H_r \subset H$；$H = \{< root, x_l >, < root, x_r >, H_l, H_r\}$；

（4）$(D_l, \{H_l\})$是一棵符合本定义的二叉树，称为根的左子树；$(D_r, \{H_r\})$也是一棵符合本定义的二叉树，称为根的右子树。

基本操作：

InitBiTree(&T)

操作结果：构造一棵空二叉树，在本关中即构造一棵空的 Huffman 树。

BulidHuffmanTree(&T, n)

初始条件：T 存在、非空且按字符的频度非递增排列，n 为 T 中字符结点的个数。

操作结果：根据频度链表中的内容，构建 T。

HuffmanCoding(&T, n)

初始条件：T 存在、非空且已建立，n 为 T 中字符结点的个数。

操作结果：对 T 中所有字符按 T 的结构进行编码，并存入相应的 code 空间。

PrintTree(T)

初始条件：T 存在、非空、已建立且已进行编码。

操作结果：遍历 L，格式化输出 L 中所有字符的编码内容，并返回 T 的带权路径长度（WPL）。

} **ADT** BinaryTree

3. **实际数据结构对应的结构体设计**

将频度链表结构和 Huffman 树结构进行共用，设计的结构体如下：

```
typedef struct ListNode
{
    char c;                     // 结点的字符
    int frequency;              // 字符的频度
    char *code;                 // 字符的编码
    struct ListNode *parent;    // 结点的双亲结点
    struct ListNode *left;      // 结点的左子树
    struct ListNode *right;     // 结点的右子树
    struct ListNode *next;      // 结点的后继结点
} ListNode, HuffmanTree;
```

2.2.2 执行流程设计

本程序的主要执行流程包括两个步骤，即“建立 Huffman 树”、“进行 Huffman 编码”。下面详细介绍这两个步骤并分析其中算法的逻辑。

1. **建立 Huffman 树**

根据上一关中已得到的有序的频度链表，将频度链表中的结点作为 Huffman 树中的结点，建立 Huffman 树。由于我们后续仍需要按照上一关中的字符顺序进行处理，所以我们应该在原有链表的基础上进行操作。因此在本关中，建立 Huffman 树的本质就是完善频度链表中各个字符结点的双亲结点和孩子结点的指针信息。而在这一过程中，我们只需要置新建的 Huffman 树结点的字符为'\0'，就可以与链表中的有效字符进行区分。这一步骤的执行过程及关键算法逻辑如图 2-1 所示。

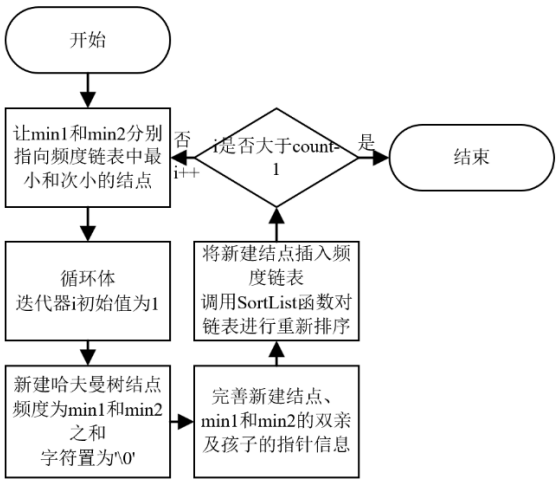

图 2-1 建立 Huffman 树的执行过程及关键算法逻辑

2. 进行 Huffman 编码

根据建立好的 Huffman 树，对每个字符进行 Huffman 编码。同样，我们仍可以基于原有的频度链表，通过遍历，找到其中字符不为'\0'的结点，此即 Huffman 树中的叶子结点；然后，通过不断向上回溯，根据相应的 Huffman 树结构，即可从叶子到根逆向求出各个字符的 Huffman 编码。这一步骤的执行过程及关键算法逻辑如图 2-2 所示。

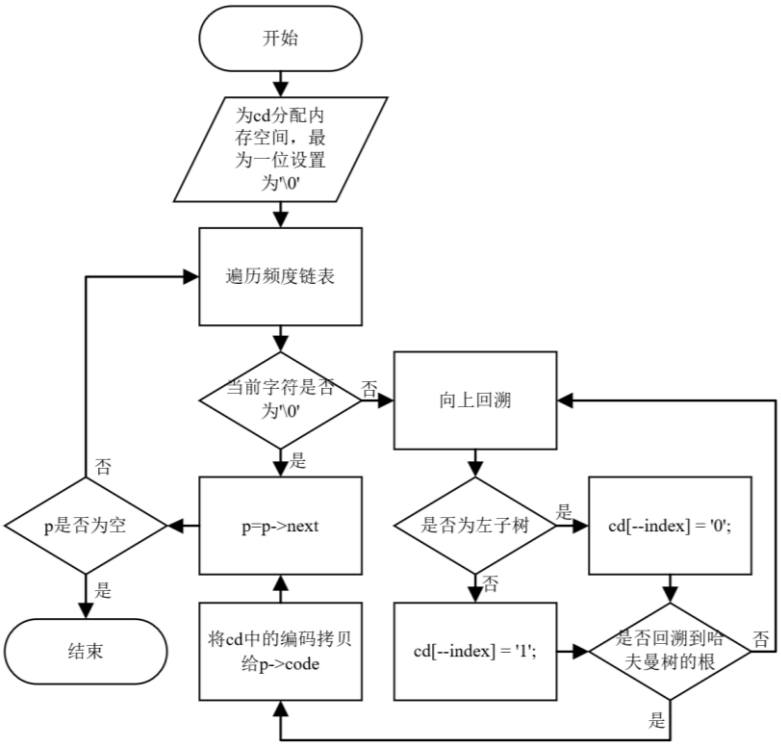

图 2-2 进行 Huffman 编码的执行过程及关键算法逻辑

本关涉及到的核心算法的关键代码将在附录二中呈现。

2.3 算法函数设计

表 2-1 本关主要函数及其功能

函数名	主要功能
void BuildHuffmanTree(ListNode* head, int count)	根据上一关中已得到的有序的频度链表，将频度链表中的结点作为 Huffman 树中的结点，建立 Huffman 树，count 为频度链表中字符的个数
void HuffmanCoding(ListNode* head, int count)	根据建立好的 Huffman 树，对每个字符进行 Huffman 编码
int PrintList(ListNode* head)	格式化输出频度链表中的编码内容，并返回 Huffman 树的带权路径长度（WPL）

2.4 用例测试

使用头歌测试数据 1 的运行结果如图 2-3 所示，使用“Do not go gentle into that good night.”作为自定义测试数据的运行结果如图 2-4 所示。在两个用例中，均是频度越高的字符编码长度越短，这符合 Huffman 编码的原则，使得 WPL 较小，说明我所设计的算法逻辑正确。

```
Each man is the architect of his own fate.
You cannot step twice into the same river.
' ' 15 000
't' 9 101
'e' 8 0010
'a' 6 0011
'i' 6 0100
'c' 5 0110
'h' 5 0111
'n' 5 1000
'o' 5 1001
's' 4 1111
'r' 3 01010
'm' 2 11010
'f' 2 11011
'w' 2 11100
'.' 2 11101
'E' 1 010110
'\n' 1 010111
'Y' 1 110000
'u' 1 110001
'p' 1 110010
'v' 1 110011
339
D:\Program Files\Microsoft Visual Studio\source\repos\ConsoleApplication74\x64\D
5216) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->
le when debugging stops.
Press any key to close this window . . .
```

图 2-3 头歌测试数据 1 在本地的运行结果

```
Do not go gentle into that good night.
' ' 7 000
'o' 6 001
't' 6 010
'n' 4 101
'g' 4 110
'e' 2 1000
'i' 2 1001
'h' 2 1110
'D' 1 01100
'l' 1 01101
'a' 1 01110
'd' 1 01111
'.' 1 11110
130
D:\Program Files\Microsoft Visual Studio\source\repos\ConsoleApplication74\x64\D
6048) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->
le when debugging stops.
Press any key to close this window . . .
```

图 2-4 自定义测试数据“Do not go gentle into that good night.”在本地的运行结果

2.5 复杂度分析

在 BuildHuffmanTree 函数中，构建 Huffman 树的核心操作是不断从频度链表中取出两个最小的结点，合并成一个新的结点，并将新结点插入到链表中。而每次合并操作需要遍历链表找到最小值和次小值，因此这一过程的时间复杂度为$O(n)$；由于总共需要构建 count-1 个 Huffman 树结点，以上过程共重复了$O(n)$次，因此总的时间复杂度为$O(n^2)$；空间复杂度则亦是来源于这新创建的 count-1 个结点，因此为$O(n)$。在 HuffmanCoding 函数中，对于每个字符，我们需要从叶子结点不断向上回溯到根节点来生成编码，而 Huffman 树的高度在最坏情况下是$O(n)$，因此对每个字符，生成编码的时间复杂度为$O(n)$，总共有$O(n)$个字符需要编码，因此时间复杂度为$O(n^2)$；为每个字符的编码分配内存空间时，每个编码的长度最多为$O(n)$，总共有$O(n)$个字符，因此空间复杂度也为$O(n^2)$。

2.6 实验小结

本次实验第二关的编程难度显著高于第一关，在编写 BuildHuffmanTree 和 HuffmanCoding 两个关键函数时，我遇到了诸多棘手难题。

在 BuildHuffmanTree 函数的实现过程中，我最初考虑另外开辟新的内存空

间来完整存储树的结构，但这样做会大幅增加程序的空间复杂度，同时题目要求输出时按照原有链表的顺序进行，这使得这一方案难以满足需求。经过多次尝试未果后，我果断放弃了这一思路。通过仔细研究，我意识到本关的突破口依然在于“频度链表”和“Huffman 树”结构的“共用”，而构建 Huffman 树的本质就是完善各结点的指针信息。基于此，我想到了利用特殊字符'\0'来区分 Huffman 树结点和有效字符，并借助已经编写好的 SortList 函数，对频度链表进行重新排序。然而，判断 Huffman 树是否构建完毕又成为了新的难题。我尝试了多种方法，如统计频度链表中孩子为 NULL 的结点个数等，但都不能有效地处理所有情况。最终，我联想到“Huffman 树中不存在度数为 1 的结点”这一重要性质，通过控制循环次数，较为完美地解决了该问题。

在编写 HuffmanCoding 函数时，我最初打算采用递归的方式，从根开始正向求出每个字符的 Huffman 编码。但在实现过程中，我多次遇到了解引用空指针的报错情况。经过仔细调试仍无法解决后，我转变了思路，采用从叶子结点向上回溯的方式，最终成功完成了函数的编写。这种方式只需要遍历频度链表，而无需考虑递归过程中的复杂结束条件，因此提高了函数的可读性。

第二关是我在整次实验中耗时最长的部分。不过，通过这一关程序的编写，我深刻认识到，解决问题的方式多种多样，尝试不同的算法固然重要，但最终找到一个易于理解、实现且健壮的解决方案才是关键。同时，通过在调试过程中处理大量解引用空指针等报错问题，也显著提升了我根据 IDE 的错误提示对程序进行纠错的能力。

3 哈夫曼编码和解码

3.1 问题描述

利用上一关得到的各个字符的 Huffman 编码，编写程序，对输入的整段英文文本进行 Huffman 编码和解码。

3.2 系统设计

本关共分为三个部分。首先，根据上一关得到的各个字符的 Huffman 编码，遍历整段英文文本，对其进行编码；随后，遍历文本编码后的字符串，根据各个字符的 Huffman 编码，逐一进行比对，从而完成文本的 Huffman 解码。最后，根据输出示例，格式化输出编码后的文本、解码后的文本以及对应的 WPL 即可。

3.2.1 数据结构设计

1. 频度链表的抽象数据结构

存储各字符的频度链表的数据结构与第二关完全一致，详细参见 2.2.1。

2. Huffman 树的抽象数据结构

ADT BinaryTree{

BinaryTree 的数据对象和数据关系R与第二关完全一致，详细参见 2.2.1。

基本操作：

InitBiTree(&T)

操作结果：构造一棵空二叉树，在本关中即构造一棵空的 Huffman 树。

BulidHuffmanTree(&T, n)

初始条件：T 存在、非空且按字符的频度非递增排列，n 为 T 中字符结点的个数。

操作结果：根据频度链表中的内容，构建 T。

HuffmanCoding(&T, n)

初始条件：T 存在、非空且已建立，n 为 T 中字符结点的个数。

操作结果：对 T 中所有字符按 T 的结构进行编码，并存入相应的 code 空间。

GenerateCode(T, *text)

初始条件：T 中所有字符已进行编码。

操作结果：根据 T，对英文文本 text 进行编码，返回编码后的字符串和 T 的 WPL。

Decode(T, *code)

初始条件：T 中所有字符已进行编码。

操作结果：根据 T，对编码 code 进行解码，返回解码后的字符串。

} **ADT** BinaryTree

3. **实际数据结构对应的结构体设计**

将频度链表结构和 Huffman 树结构进行共用，设计的结构体与第二关完全一致，详细参见 2.2.1。

3.2.2 执行流程设计

本程序的主要执行流程包括两个步骤，即“对文本进行 Huffman 编码”、“对编码进行 Huffman 解码”。下面详细介绍这两个步骤并重点分析其中的关键算法。

1. **对文本进行 Huffman 编码**

根据 Huffman 树中各个字符的 Huffman 编码，对程序开始时输入的英文文本进行编码。遍历整个字符串，在频度链表中找到当前字符的 Huffman 编码，并调用 string.h 标准库中的 strncat 函数，将当前字符的编码追加到整段文本的 code 编码中即可。这一步骤的执行过程及算法逻辑如图 3-1 所示。

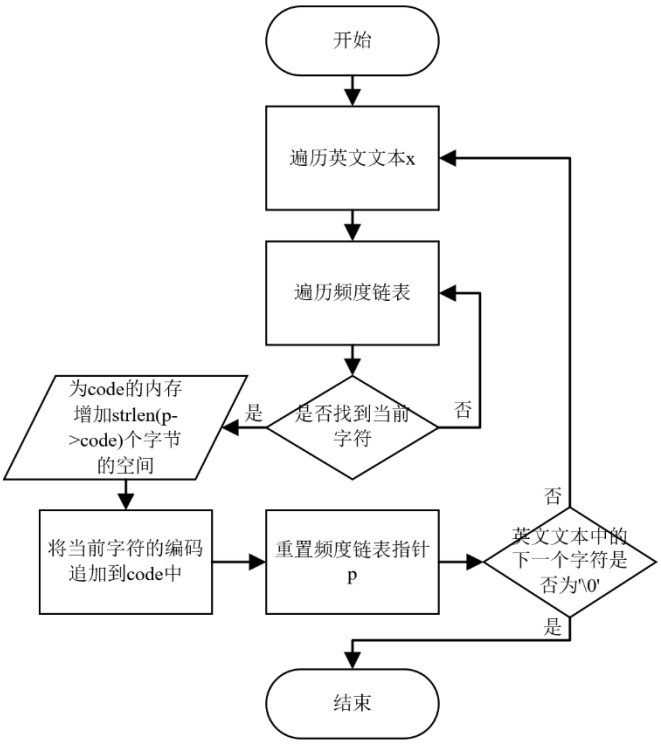

图 3-1 对文本进行 Huffman 编码的执行流程

2. 对编码进行 Huffman 解码

根据文本编码后的字符串，结合各个字符的 Huffman 编码，完成文本的解码。设置一个临时存储空间 temp，不断将当前编码值添加到 temp 中，然后利用 strcmp 函数，遍历频度链表，将当前的 temp 与各个字符的 Huffman 编码进行逐一比对，从而将相应的字符追加到解码空间 decode 中，重复以上过程，遍历整段编码即可。这一步骤的执行过程及算法逻辑如图 3-2 所示。

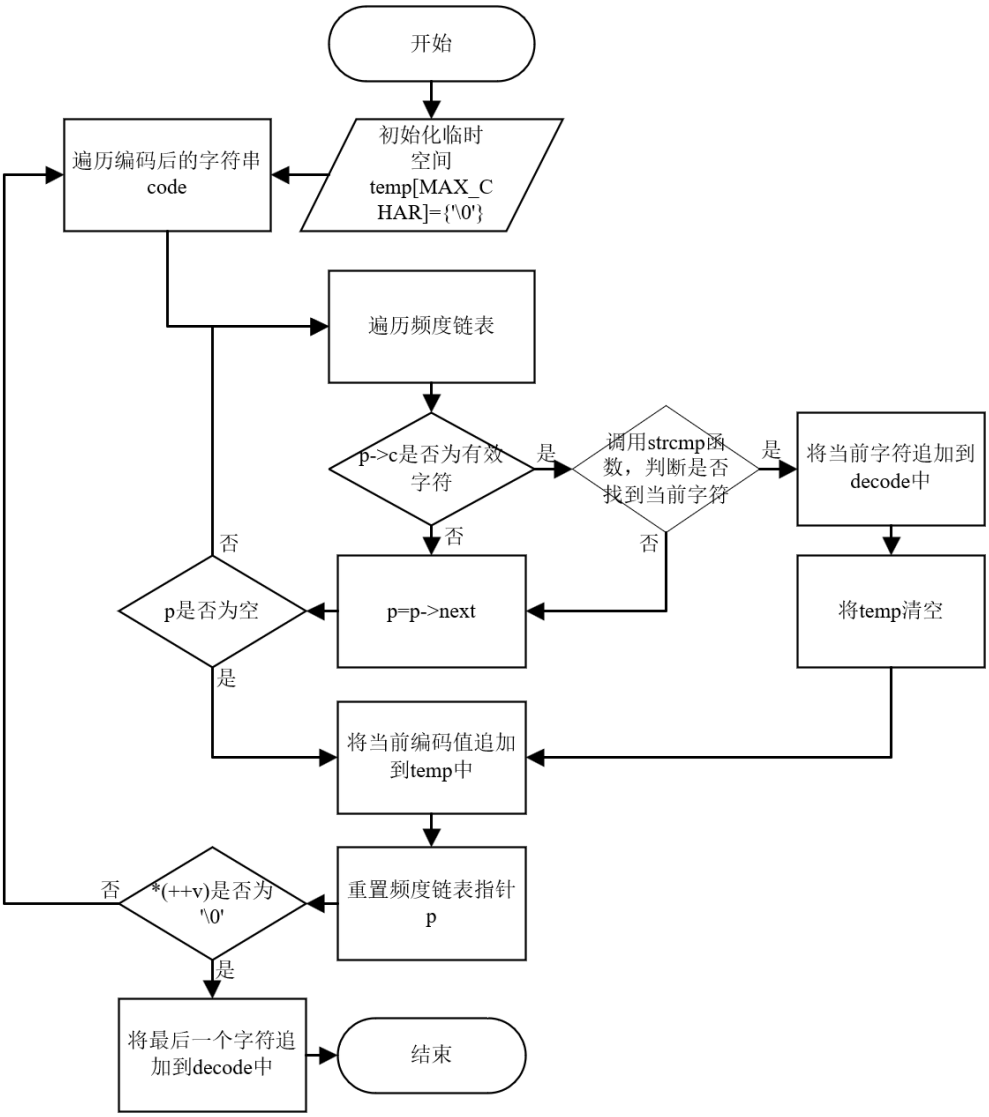

图 3-2 对编码进行 Huffman 解码的执行流程

本关涉及到的核心算法的关键代码将在附录三中呈现。

3.3 算法函数设计

表 3-1 本关主要函数及其功能

函数名	主要功能
char* GenerateCode(ListNode* head, char* x, int* WPL)	根据各个字符的 Huffman 编码，对输入的英文文本 *x* 进行编码，返回编码后的字符串，同时记录 Huffman 树的 WPL

void Decode(ListNode* head, char* code, char decode[])	根据各个字符的 Huffman 编码，对已编码的英文文本进行解码，并将解码后的文本存储到 decode 中

3.4 用例测试

使用头歌测试数据 4 的运行结果如图 3-3 所示，使用“Do not go gentle into that good night”整首诗的内容[2]作为自定义测试数据的运行结果如图 3-4 所示。在两个用例中，输入的原始英文文本和经 Huffman 编码后再解码的文本完全相同，说明我所设计的两个函数的算法逻辑正确。

```
Gazing on Mount Tai

O peak of peaks, how high it stands!
One boundless green o'erspreads two states.
A marvel done by Nature's hands,
O'er light and shade it dominates.
Clouds rise there from and lave my breast;
My eyes are strained to see birds fleet.
Try to ascend the mountain's crest;
It dwarfs all peaks under our feet.
0100000101010111110001001110011010100100000110000110011010000011111110001100011001110010101001100101001000010011101101010
1010100010001000000111000011101101010101010001000011001111001111010000001000110011110101001110101111010010100101100010001011
0010111000000100010111110110010010000111001010011101110000011111110000001111101010001000100111010110001011011100001000001110
0010110000001110110100010101010000100010010111001000011000000010001011001010110101000111100110010011111100010111010101100011101
0001011111000100001000001100101001110111011110001011111110101011011111100010111100000010011110101011100000010001100111110010
0100001111000101100000111110010011101011110101110001010111000000100100011110101010000110100101001011000100001000000111010100
1100010101101010001111001100101001100011110000001111100001000100110000100100011010010111011011000101001011100100000000011
1010010101110000001001111100101110100010100101110101111000111011110001010101000101101101001100101001101011110001101011101011
0001001010110001010010001011010000101010011100101000010010111000000001000110110100111011101001100000001000100101110011110101
0101101110011001010011101000011110001011000000001010100010100000010111000000100101101110110100101110100000111111100011001010
1001110011100000010010100000010001010001011011010011001010011001011000100001010001101011000011100000100101011111011110001110
1101010101010001000010011111111100000011011000001000001111110000010111001011010110111001
Gazing on Mount Tai

O peak of peaks, how high it stands!
One boundless green o'erspreads two states.
A marvel done by Nature's hands,
O'er light and shade it dominates.
Clouds rise there from and lave my breast;
My eyes are strained to see birds fleet.
Try to ascend the mountain's crest;
It dwarfs all peaks under our feet.
1451
D:\Program Files\Microsoft Visual Studio\source\repos\Huffman编解码\x64\Debug\Huffman编解码.exe (process 43640) exited with
code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console w
hen debugging stops.
Press any key to close this window . . .
```

图 3-3 头歌测试数据 4 在本地的运行结果

[2] 具体内容见附录四

```
Do not go gentle into that good night
Do not go gentle into that good night,
Old age should burn and rave at close of day;
Rage, rage against the dying of the light.
Though wise men at their end know dark is right,
Because their words had forked no lightning they
Do not go gentle into that good night.
Good men, the last wave by, crying how bright
Their frail deeds might have danced in a green bay,
Rage, rage against the dying of the light.
Wild men who caught and sang the sun in flight,
And learn, too late, they grieved it on its way,
Do not go gentle into that good night.
Grave men, near death, who see with blinding sight
Blind eyes could blaze like meteors and be gay,
Rage, rage against the dying of the light.
And you, my father, there on the sad height,
Curse, bless me now with your fierce tears, I pray.
Do not go gentle into that good night.
Rage, rage against the dying of the light.
1101000011100001010111001000001100111000011011101001010101111000101101010010011100000101001100000100000110011101110011
1000010110110110100100101101111010000111000010101110010000011001110000110111010100101010111100010110101001001110000001010
0110000001000001100111011100111000010110110110100100101100111011101000011101010011100010000110111000110001001011111010110
101001110000100001110101010010101000100001010011100001001100011010011110001000001000010100011010101111100011100001110011
0000000011110000011010011001000110110100000011000011011111100100000100110000110111000100001101000101101011100000010000001010
0111100000111001101101101010110000011100110000000001010011110001010110110110100100100100010110110011001111001011111010101
1010010001010011011110001110000100011111010100010000010000000101001111101101001000111010100111000101000000101011110100100
0001111000010011010000000001011110000000100110110110100100101100111011010000000111101000110001101011100011100000101001111
101101001000101001011101001001111100000001001100000111000001100001110100110100000011100111000010101110001010110110110 10010
0100101101101010110000000101001111001101110111101000011100001010111001000001100111000011011101010010101011110001011010100
1001110000001010011000000100000110011101110011100001011011011010010010010001011011010000000101110111001110000100011111101011
1001000000101001111000101011000110000001000010100110001101001111000001000010011011100100010100010100100110110110101011100001
0010111101001000010000101001101101101001001011011001100111100111110110100100000110000100110001011101010000011111111110011
111000000001000111011011010010010000100110001101001111000000111100001011010001111001110001011010100010000000011001001111111
0101000010000110000001101110011101101000000110000110111111001000001001100000110111000100001101000101101011100000010000000101001
11100000011100110110110101011000001110011000000000101001111000101011011011010010010010001011011001100100110111010100111000
0100011111010100010100110010111000101000110001101010110100100100001000010100111000110001000010101100000001010011110001100
0110101010100010110101000000110001010110110110100100101100111011101000010010100111000101011111000010010101110010000010011
1011100010101100000010111111001000000101001111001101000011001001101111111010011110011100010110010000001110101000101100101100
0000101001100000110111001110111101000011100001010111001000001100111000011011101010010101011110001011010100100111000000101
00110000001000001100111011100111000010110110110100100100010110110100000001010011000110100111100001000111110101110010000
10111110000100100000011111110000001010011100100010100110010111000110001111110001010011011001010010000100001101011011010100
1111011010101100001100010110110100100101101101000000001010110110101001110001110011011111110000000101000101111110101101010011
100001000011010110000001100101011100010101101110100000011100001000111110010111011101001110000000100001010011100001000011110
00011010000001101110011101101000000110000110111111001000001001100000110111000100001101000101101011100000010000000101001111000 00
1110011011011010101100000111001100000000010100111100010101101101101001001001000101101110100001001010011100000110101111101
0111001000010001100110100000110001000000101001111010011100100000010100111101001111000001110101000000101001111000110001000001
1100010011111011011010010010110011101100110010111101010100111000111110010000100001101011111100011000000010001111100000101
0111101001000101001101100101001000001101011111010101001000001100010111110100110100011110000001011110000100111000110010000
011001100000001100110101001100000110101000101101111010000111000010101110010000011001110000110111010100101010111100010110
1010010011100000010100110000001000000110011101110011100001011011011010010010010001011011010000001100001011111100100001001100
001101110001000011010001011010111000000100000001010011110000011100110110110101011000001110011000000000101001111000101011011
0110100100100100010
Do not go gentle into that good night
Do not go gentle into that good night,
Old age should burn and rave at close of day;
Rage, rage against the dying of the light.
Though wise men at their end know dark is right,
Because their words had forked no lightning they
Do not go gentle into that good night.
Good men, the last wave by, crying how bright
Their frail deeds might have danced in a green bay,
Rage, rage against the dying of the light.
Wild men who caught and sang the sun in flight,
And learn, too late, they grieved it on its way,
Do not go gentle into that good night.
Grave men, near death, who see with blinding sight
Blind eyes could blaze like meteors and be gay,
Rage, rage against the dying of the light.
And you, my father, there on the sad height,
Curse, bless me now with your fierce tears, I pray.
Do not go gentle into that good night.
Rage, rage against the dying of the light.
3859
D:\Program Files\Microsoft Visual Studio\source\repos\Huffman编解码\x64\Debug\Huffman编解码.exe (process 25888) exited w
ith code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

图 3-4 自定义测试数据“Do not go gentle into that good night”在本地的运行结果

3.5 复杂度分析

在 GenerateCode 函数中，我们通过遍历英文文本中的每个字符，并在频度链表中查找对应的 Huffman 编码，然后将这些编码拼接起来，从而形成最终的编码字符串。对于每个字符，查找其编码的时间复杂度均为$O(n)$，其中n为频度链表

中字符的个数。假设输入的字符数为m，则总的时间复杂度为$O(m \cdot n)$。而GenerateCode 函数需要动态分配内存来存储编码后的字符串，其最差的情况是所有字符的编码长度均为 n，所以空间复杂度也为$O(m \cdot n)$。在 Decode 函数中，我们通过逐字符地处理编码字符串 code，并在频度链表中查找对应的编码来重建原始字符串。对于 code 中的每一个编码值，查找其对应的原始字符的时间复杂度均为 $O(n)$，假设编码字符串的长度为m'，则 Decode 函数的时间复杂度为$O(m' \cdot n)$。Decode 函数需要存储解码后的字符串，由于它的长度与原始字符串的长度相同，所以其空间复杂度即$O(m)$。

3.6 实验小结

在完成前两关实验的基础上，第三关的程序编写过程，我进行得比较顺利。明确了编程任务之后，我迅速搭建起本关 GenerateCode 和 Decode 两个关键函数的框架。在 GenerateCode 函数中，我使用动态内存分配与字符串追加的方法，高效地完成了编码字符串的任务；而在 Decode 函数中，我采用逐字符比对的遍历方式，实现了原始字符串的重建。

前前后后算下来，我在这次实验中投入了至少 10 个小时的时间。不过，尽管耗费时间巨大，但对我的意义非凡。首先，我确保了程序的高度原创性。每个关键算法，或是源于课内知识的迁移改良，或是出于我的仔细构思。这让我深刻体会到作为一位开发者应有的责任感。其次，在程序的调试过程中，面对频频报错的情况，我没有惧怕，也没有放弃，这逐渐培养起我坚韧不拔的编程精神。最后，通过逐步剖析复杂问题，这次实验也很好地锻炼了我在思维层面的灵活性与韧性，提升了我系统性解决问题的能力。

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] 谭浩强. C 语言程序设计(第 5 版). 清华大学出版社

[3] 李赤松等. 程序设计基础(C++). 电子工业出版社

[4] 严蔚敏等.数据结构题集(C 语言版). 清华大学出版社

附录一 统计字符频度

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_CHAR 4096       //字符集的最大长度
typedef struct ListNode           //结点结构
{
    char c;                        // 结点的字符
    int frequency;                 // 字符的频度
    struct ListNode *next;         // 结点的后继结点
} ListNode;
typedef enum bool{          //自定义枚举类型 bool，包含 true 和 false
    false, true
}bool;
```

Input 函数中的关键代码：

```
int index = 0;          //index 作为字符数组 x 的下标
//使用 getchar 函数循环读取输入，直到遇到 EOF
while ((x[index] = getchar()) != EOF) {
    index++;
}
x[index] = '\0';      //在 x 的末尾添加结束符\0
```

SortList 函数中的关键代码：

```
bool flag = true;
ListNode* r = head;
int i, j, k;
//类比一维数组中的冒泡排序
for (i = 1; i <= n && flag; i++) {
    //flag 标志变量用于判断频度链表是否已经排好
    flag = false;
    for (j = n - 1; j >= i; j--) {
        //n 为字符链表中字符的个数，在此发挥作用
        for (k = 1; k <= j; k++) {    //通过循环让 r 指向当前需要判断的结点
            r = r->next;
        }
        //未排好，调用 exchange 函数进行交换
        if (r->frequency < r->next->frequency) {
            exchange(head, r->next);      //交换 r 和 r->next
            flag = true;    //置 flag 为 true，表示当前链表仍未排好
        }
        r = head;    //一轮判断结束，需要让 r 重新指向头结点
    }
}
```

附录二 建立哈夫曼树并生成哈夫曼编码

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_CHAR 4096       //字符集的最大长度
typedef struct ListNode
{
      char c;                          // 结点的字符
      int frequency;                   // 字符的频度
      char *code;                      // 字符的编码
      struct ListNode *parent;         // 结点的双亲结点
      struct ListNode *left;           // 结点的左子树
      struct ListNode *right;          //  结点的右子树
      struct ListNode *next;           // 结点的后继结点
} ListNode, HuffmanTree;
typedef enum bool{          //自定义枚举类型 bool，包含 true 和 false
      false, true
}bool;
```

BuildHuffmanTree 函数中的关键代码：

```
ListNode* min1, * min2;
//min1 和 min2 分别指向频度链表中最小和次小的结点
min1 = head;
min2 = min1->next;
while (min2->next) {
      min1 = min1->next;
      min2 = min2->next;
}
//字符数为 n，共有 2n-1 个结点，因此需要构造 n-1 个 Huffman 树结点
for (int i = 1; i <= count - 1; ++i) {
      //临时结点 temp 用于存储新建的 Huffman 树结点
      ListNode* temp = (ListNode*)malloc(sizeof(ListNode));
      //初始化新建的 Huffman 树结点
      temp->c = '\0';     //新建结点的字符为'\0'，与有效字符进行区分
      temp->frequency = min1->frequency + min2->frequency;
      //新建结点的频度为 min1 和 min2 的频度之和
      temp->left = min1;     //左子树为 min1
      temp->right = min2;   //右子树为 min2
      temp->parent = NULL;   //新建结点的双亲结点为 NULL
      min1->parent = min2->parent = temp;   //min1 和 min2 的双亲结点均为新建的结点
      //将新建的 Huffman 树结点插入频度链表中
      temp->next = head->next;   //先将新建的结点插入频度链表头部
```

```
        head->next = temp;
        //调用 SortList 函数对频度链表进行排序，此时链表长度已变为 count+i
        SortList(head, count + i);
        ListNode* u = min1;   //u 用于记录 min1 的位置
        //重新设置 min1 和 min2
        min1 = head;
        min2 = min1->next;
        //链表重新排序后，下一轮的 min2->next 即指向该轮次的次小结点
        while (min2->next != u) {
            min1 = min1->next;
            min2 = min2->next;
        }
    }
```

HuffmanCoding 函数中的关键代码：

```
    ListNode* p = head->next;       //p 指向频度链表的第一个结点
    char* cd = (char*)malloc(sizeof(char) * count);     //cd 为编码的临时工作空间
    int index;       //index 作为 cd 的下标
    cd[count - 1] = '\0';     //cd 的最后一位为'\0'
    //遍历频度链表
    while (p) {
        //当 p->c 不为\0'时，说明 p->c 是有效字符
        if (p->c != '\0') {
            index = count - 1;     //cd 的下标从 count-1 开始，向前存储编码
            ListNode* g, * f;
            //从 p 结点开始向上回溯，逆向求出该字符的 Huffman 编码
            for (g = p, f = g->parent; f; g = f, f = g->parent) {
                //如果 g 是 f 的左子树，则 cd 的下标减 1，存储 0
                if (f->left == g) {
                    cd[--index] = '0';
                }
                //否则 cd 的下标减 1，存储 1
                else {
                    cd[--index] = '1';
                }
            }
            //为 p->code 编码分配内存
            p->code = (char*)malloc(sizeof(char) * (count - index));
            //有效编码从 cd[index]开始，因此应向 strcpy 函数传入&cd[index]
            strcpy(p->code, &cd[index]);
        }
        p = p->next;       //p 指向链表中的下一个结点
    }
```

附录三 哈夫曼编码和解码

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_CHAR 4096       //字符集的最大长度
typedef struct ListNode
{
      char c;                          //  结点的字符
      int frequency;                   //  字符的频度
      char *code;                      //  字符的编码
      struct ListNode *parent;         //  结点的双亲结点
      struct ListNode *left;           //  结点的左子树
      struct ListNode *right;          //   结点的右子树
      struct ListNode *next;           //  结点的后继结点
} ListNode, HuffmanTree;
typedef enum bool{          //自定义枚举类型 bool，包含 true 和 false
      false, true
}bool;
```

GenerateCode 函数中的关键代码：

```
ListNode* p = head->next;
//code 用于存储字符串的编码，v 用于遍历字符串 x
char* code = (char*)malloc(sizeof(char)), * v = x;
code[0] = '\0';   //初始化 code
while (*v != '\0') {   //遍历字符串 x
      while (p->c != *v) {     //遍历频度链表，找到当前字符
            p = p->next;
      }
      code = (char*)realloc(code, sizeof(char) * (strlen(code) + strlen(p->code) + 1));
//动态分配内存
      strncat(code, p->code, strlen(p->code) + 1);   //将当前字符的编码添加到
code 中
      *WPL += strlen(p->code);   //更新 WPL
      p = head->next;   //让 p 重新指向头结点的下一个结点
      v++;
}
return code;     //返回编码后的字符串
```

Decode 函数中的关键代码：

```
ListNode* p = head->next;
char temp[MAX_CHAR] = { '\0' };   //temp 用于临时存储当前字符的编码
char* v = code;     //v 用于遍历编码后的字符串
while (1) {
```

```
        //遍历频度链表，找到当前字符的编码
        while (p) {
            if (p->c != '\0') {     //p->c != '\0'表明 p->c 为频度链表中的有效字符
                if (strcmp(p->code, temp) == 0) {    //找到当前字符
                    decode[strlen(decode)] = p->c; //将当前字符添加到 decode 中
                    memset(temp, '\0', sizeof(temp)); //将 temp 清空
                    break;
                }
            }
            p = p->next;
        }
        temp[strlen(temp)] = *v;    //将当前编码值添加到 temp 中
        p = head->next;    //让 p 重新指向头结点
        v++;     //v 指向下一个编码值
        if (*v == '\0') {    //编码后的字符串已经遍历完毕，处理最后一个字符
            while (p) {
                if (p->c != '\0') {
                    if (strcmp(p->code, temp) == 0) {
                        decode[strlen(decode)] = p->c;
                        break;
                    }
                }
                p = p->next;
            }
            break;
        }
    }
    decode[strlen(decode)] = '\0';    //添加字符串结束符'\0'
```

附录四 自定义测试用例

3.4 中的自定义测试用例“Do not go gentle into that good night”整首诗的内容如下：

Do not go gentle into that good night

Do not go gentle into that good night,

Old age should burn and rave at close of day;

Rage, rage against the dying of the light.

Though wise men at their end know dark is right,

Because their words had forked no lightning they

Do not go gentle into that good night.

Good men, the last wave by, crying how bright

Their frail deeds might have danced in a green bay,

Rage, rage against the dying of the light.

Wild men who caught and sang the sun in flight,

And learn, too late, they grieved it on its way,

Do not go gentle into that good night.

Grave men, near death, who see with blinding sight

Blind eyes could blaze like meteors and be gay,

Rage, rage against the dying of the light.

And you, my father, there on the sad height,

Curse, bless me now with your fierce tears, I pray.

Do not go gentle into that good night.

Rage, rage against the dying of the light.

附录五 实验报告格式要求

1. 标题格式

大标题文字选用黑体，小二，加粗；数字字母选用 Times New Roman 小二，加粗。二级标题文字黑体，四号，加粗；数字字母选用 Times New Roman 四号，加粗。三级标题文字黑体，小四，加粗；数字字母选用 Times New Roman 小四，加粗。

2. 正文格式

正文文字选择宋体，小四；数字字母选用 Times New Roman 小四。正文行距为 1.5 倍。

3. 图片格式

所有出现图片须有图号，图号格式例如：**图 1-1 链式队列入队/出队系统**

文字选择黑体，小四；数字字母选用 Times New Roman 小四。编号与名称之间须有空格。图片与图号须居中。

为达规范，所有报告内出现算法图应使用 Visio 进行绘制。其中算法图中文字字体为黑体，数字及字母选用 Times New Roman。图中出现所有文字，数字，字母大小应一致。在绘图时，应确保算法图为白底黑框黑字，如本示例图中所示。同时算法图应紧凑，大方，美观。图中箭头不宜过长，同时箭头不应有交叉。

运行结果截图须为白底，同时不要有多余边框。

4. 代码格式

报告中出现代码应选用 Times New Roman 小四，单倍行距。