

**课 程 实 验 报 告**

**课程名称： 数据结构实验**

实验名称： Huffman编解码

**专业班级：** 网安2406

**学 号：** U202412361

**姓 名：** 黄文彬

**指导教师：** 冯君

**报告日期：** 2025.5.12

**网络空间安全学院**

# 实验报告评分表

|  |  |  |  |
| --- | --- | --- | --- |
| **评分项** | | **总分** | **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** | | 20 |  |
| **教师签名： 日期：** | | | |

**目 录**

[1](#_Toc124441966) **[统计字符频度](#_Toc124441966)** [4](#_Toc124441966)

[1.1 问题描述 4](#_Toc124441967)

[1.2 系统设计 4](#_Toc124441968)

[1.3 算法函数设计 6](#_Toc124441971)

[1.4 用例测试 7](#_Toc124441972)

[1.5 复杂度分析 9](#_Toc124441973)

[1.6 实验小结 1](#_Toc124441974)0

[2](#_Toc124441975)**[建立哈夫曼树并生成哈夫曼编码](#_Toc124441975)** [1](#_Toc124441975)1

[2.1问题描述 1](#_Toc124441976)1

[2.2 系统设计 1](#_Toc124441977)1

[2.3 算法函数设计 1](#_Toc124441980)4

[2.4 用例测试 1](#_Toc124441981)5

[2.5 复杂度分析 1](#_Toc124441982)7

[2.6 实验小结 1](#_Toc124441983)7

[3](#_Toc124441975)**[哈夫曼编码和解码](#_Toc124441975)** [19](#_Toc124441975)

[3.1问题描述 1](#_Toc124441976)9

[3.2 系统设计 1](#_Toc124441977)9

[3.3 算法函数设计 2](#_Toc124441980)2

[3.4 用例测试 2](#_Toc124441981)3

[3.5 复杂度分析 2](#_Toc124441982)3

[3.6 实验小结 2](#_Toc124441983)4

**[参考文献](#_Toc124441984)** [2](#_Toc124441984)5

**[附录一 基于](#_Toc124441986)**[Huffman](#_Toc124441986)**[编解码的操作实现](#_Toc124441986)** [2](#_Toc124441986)6

**[附录二 实验报告格式要求](#_Toc124441987)** [36](#_Toc124441987)

# 1 **统计字符频度**

## 问题描述

本次实验内容“统计字符频度”。实现程序读取输入字符，统计每个字符的出现频率，构建单链表保存各字符及其频度。如果字符已经在链表中，则对应频度加一；如果没有，则创建新结点插入链表尾部。统计完毕后，需要对链表进行排序，使字符按照频度从高到低排列；若频度相同，则保持输入顺序。最后，按要求格式输出每个字符及其频度（特殊字符如换行需特殊显示），每行输出一个结果。

## 1.2 系统设计

整个程序分为处理输入信息建立字符频度链表与排序并输出字符频度两个部分。

### 1.2.1 数据结构设计

1. 单链表的抽象数据结构

如下为用于存储字符频度的链表的抽象数据结构：

**ADT** FreqList{

数据对象：所有输入的不同字符，各自出现的次数。

数据关系：各字符用结点结构存储，结点间通过next指针顺序链接成单链表。每个结点包含字符字段和出现频度字段。

基本操作：

InitList(&L)

操作结果： 构造一个空的频度链表，即L=NULL。

InsertOrAddFreq(&L, c)

初始条件： 频度链表L已存在，c为输入的待处理字符。

操作结果： 在链表L中查找字符c，若c已在L中，频度frequency+1；若无该字符，则创建新结点并插入表尾。

SortList(&L)

初始条件： 频度链表L已存在。

操作结果： 按频度frequency从高到低（若频度frequency相等，则顺序不变）排序链表L。

PrintFreqList(&L)

初始条件： 频度链表L已存在。

操作结果： 按顺序输出每个字符及其频度frequency，特殊字符如换行输出为“\n”。

} **ADT** FreqList

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;

int frequency;

char \*code; // 哈夫曼编码，此处未用

struct ListNode \*parent; // 哈夫曼树用

struct ListNode \*left; // 哈夫曼树用

struct ListNode \*right; // 哈夫曼树用

struct ListNode \*next; // 频度链表用

} 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，“统计字符频度并建立链表事件”、“排序并输出频度表事件”下面介绍这两个事件的执行流程。

1. 统计字符频度并建立链表事件

初始化空链表charListHead。循环接收输入字符，对每个字符顺序遍历链表，判断字符c是否已存在：若存在，则该节点frequency加1；若不存在，则分配新节点，frequency设为1，将新节点插入到链表表尾。循环直到输入结束，链表中保存了每一种字符及其出现频度，顺序与首次出现顺序一致。

2.排序并输出频度表事件

调用sortList，对链表按照frequency从高到低排序，实现算法为链表冒泡（保持稳定）。遍历排序后的链表，依次输出每个节点的字符及出现频率，若字符为\n，按'\\n'频率输出，其余按'字符'频率输出。

两个事件的执行流程如图1-1所示。

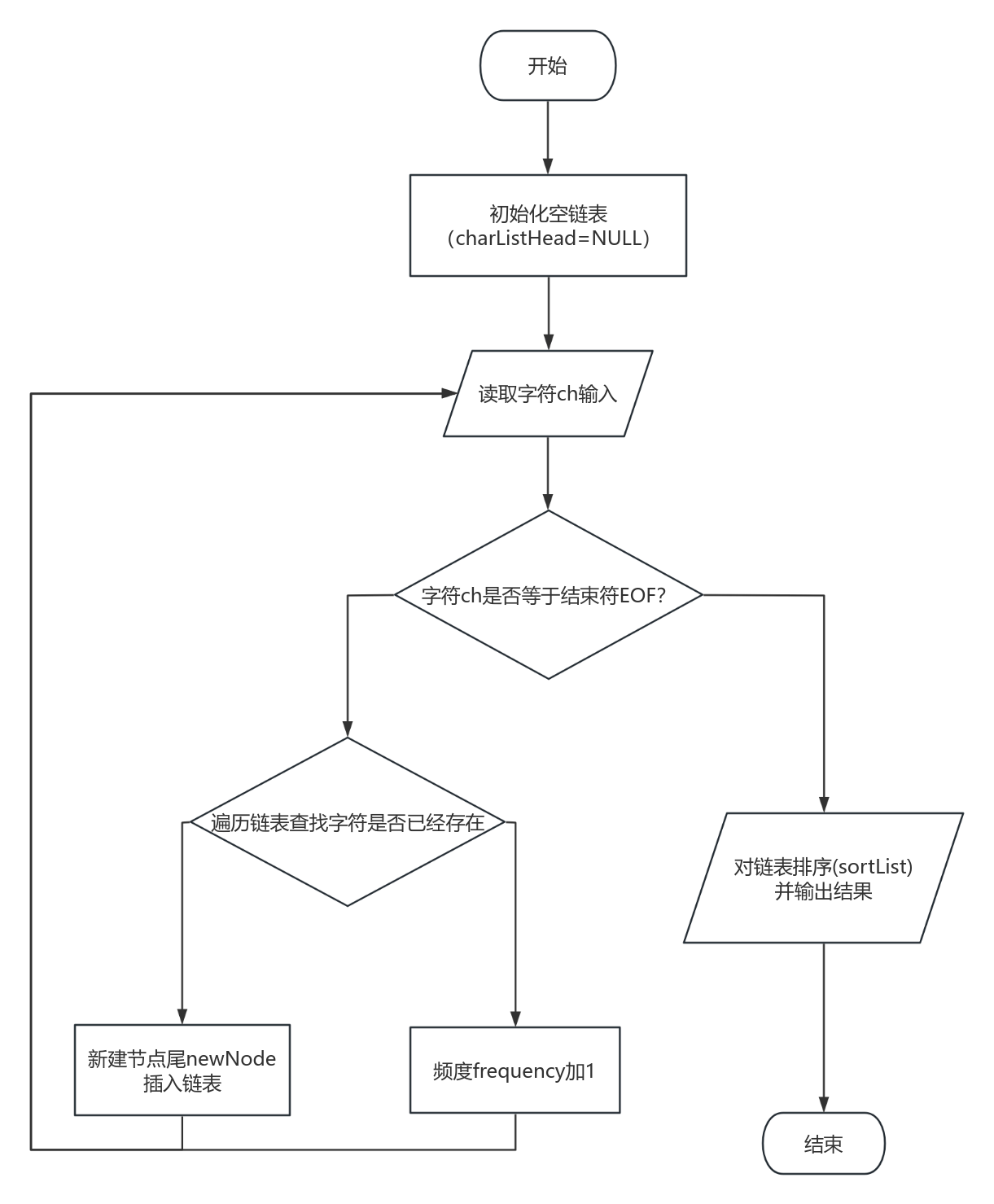


图1-1 统计字符频度建立链表并排序输出事件的完成流程

## 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 |
| int getListLength(ListNode \*head) | 统计并返回链表中节点（即不同字符）的数量。 |
| void sortList(ListNode \*\*headRef) | 将链表按frequency字段从大到小排序，频度相同保持原有顺序。 |
| int main() | 程序主流程：依次读入字符，统计频度，排序后格式化输出。 |

## 1.4 用例测试

使用头歌及自测测试数据的结果如图1-2，图1-3，图1-4所示。

图1-2 测试数据1在本地的运行结果

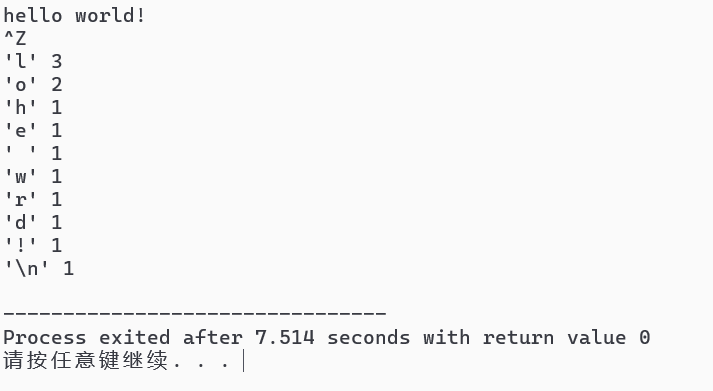


图1-2 测试数据1在本地的运行结果

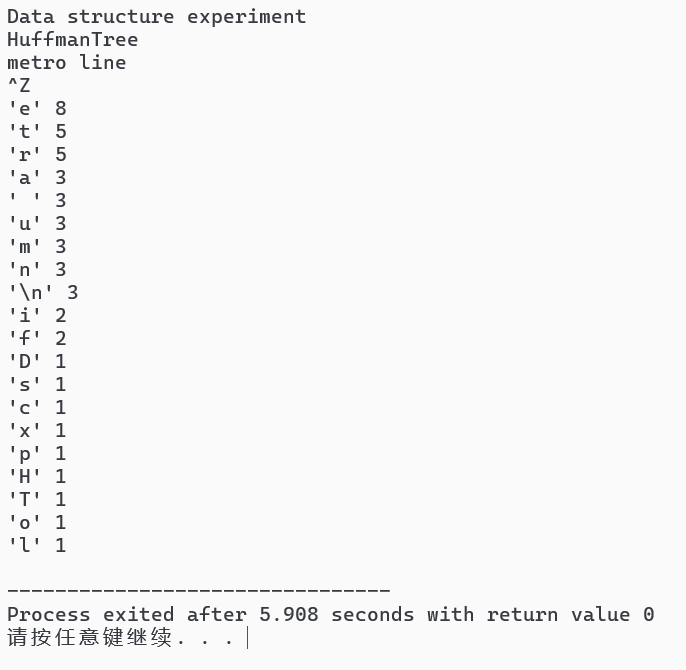
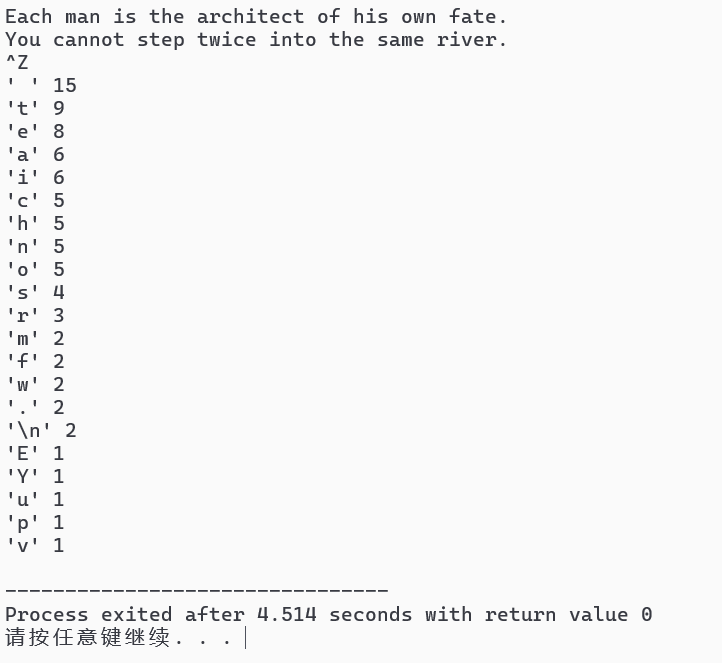


图1-3 测试数据2在本地的运行结果

图1-4 测试数据3在本地的运行结果

## 1.5 复杂度分析

时间复杂度分析：首先，对于每个输入的字符，需要遍历现有链表查找是否存在。在最坏情况下（所有字符都不同），总比较次数为1+2+3+…+n=，此时的时间复杂度为，其中n是输入字符总数。之后，再使用冒泡排序对链表进行排序。链表长度为m（不同字符的数量，m≤n），则冒泡排序的时间复杂度为，最坏情况下m≈n，因此时间复杂度为。最后遍历链表输出结果，时间复杂度为=。总时间复杂度：+=。

空间复杂度分析：链表节点数量与不同字符的数量m成正比，每个节点占用常数空间。空间复杂度为，最坏情况下m=n（所有字符不同），因此为。

## 1.6 实验小结

本关实验在明确设计思路后，整体实现过程较为顺利。通过链表统计每个字符出现的频度，并对链表进行排序，能够较好地完成本关的基本要求。在本地环境下测试时，程序输出结果与预期一致。然而在头歌平台提交后，发现输出结果与平台预期存在差异。

经过查阅相关资料，并与同学和老师交流后，发现问题主要出现在对特殊字符（如换行符\n、空格等）的处理上，以及不同平台对字符编码的差异。最初链表节点的设计未能充分考虑这些情况，导致部分字符的频度统计不完整。针对这一问题，对节点处理逻辑和输出格式进行了调整，最终顺利通过了平台的所有测试用例。

通过本次实验，进一步加深了对单链表操作的理解和掌握，包括节点的查找、插入以及链表的排序等内容。同时认识到，编写程序时需要充分考虑各种边界情况，确保程序在不同平台和环境下均能正确运行。此次实验不仅提升了对数据结构实际应用的能力，也为后续学习哈夫曼树等内容打下了良好的基础。

# 2建立哈夫曼树并生成哈夫曼编码

## 2.1问题描述

本实验基于字符频度链表动态构建哈夫曼树：循环遍历链表，每次选取两个无父结点且频度最小的结点，合并生成父结点（频度为子结点之和），设置父结点的左右指针指向子结点，并插入链表尾部，直至所有结点连通为完整树结构；随后从各叶子结点回溯路径至根结点，根据左右分支生成逆向二进制编码（左0右1），反转后得到最终哈夫曼编码；最终按原始字符统计顺序遍历链表，输出字符、频度及对应编码，并累加各字符频度与编码位数的乘积作为带权路径长度，完成数据压缩的核心逻辑构建，其关键在于动态维护结点父子关系与路径回溯的编码生成机制。

## 2.2 系统设计

本系统基于动态链表结构构建哈夫曼树，通过频度优先合并策略生成二叉树，采用路径回溯法生成二进制编码。核心流程分为频度统计、树构建、编码生成三个阶段：首先遍历输入建立有序频度链表，循环选取最小频度节点合并生成父节点并动态维护链表结构，最后通过逆向路径追踪生成编码。系统通过父子指针建立树形关系，保持原始字符顺序实现稳定输出。

### 2.2.1 数据结构设计

1. 链表的抽象数据结构

**ADT** LinkedList{

数据对象：包含若干个结点，每个结点保存一个字符、其对应频度、哈夫曼编码、父节点指针、左右孩子指针以及链表的下一个结点指针。

数据关系：各节点之间通过链表next域构成频度链表，哈夫曼树结构通过parent、left、right指针建立父子关系。频度统计时只维护链表关系，建树过程中链表和树结构并存。

基本操作：

CreateList(&L)

操作结果：创建一个空的频度链表L。

InsertChar(&L,ch)

初始条件：频度链表L已存在，ch为新读入的字符。

操作结果：在链表L中查找字符ch，若存在则frequency加一，否则新建节点加入链表末尾。

SortList(&L)

初始条件：频度链表L已存在。

操作结果：对链表L按频度降序排序，相同频率的按初次出现保持原次序。

GetListLength(L)

初始条件：频度链表L已存在

操作结果：返回链表L的节点数量。

BuildHuffmanTree(&L)

初始条件：频度链表L已存在，并统计好字符频度

操作结果：反复从链表L中选出频度未被用过的最小的两个节点，建立新父节点，并插入链表末尾，直到所有节点除根外均有parent。

} **ADT** LinkedList

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; // 字符内容

int frequency; // 字符频度

char \*code; // 哈夫曼编码字符串

struct ListNode \*parent; // 父节点指针

struct ListNode \*left; // 左子结点指针

struct ListNode \*right; // 右子结点指针

struct ListNode \*next; // 下一个频度链表节点

}ListNode,HuffmanTree;

### 2.2.2 执行流程设计

本程序的主要执行流程可以划分为如下几个事件：

1. 建立频度链表事件

接收并读取全部输入字符流，并依次统计各字符出现次数；插入链表，存在则frequency加一，不存在则建立新节点追加到链表末尾。

2. 频度链表排序事件

对链表按频率降序排序，为编码输出和后续建树算法做准备。

3. 原址建立哈夫曼树事件

记录初始各叶节点顺序到nodes[]数组，以便保持输入统计顺序；在所有未指定parent指针的节点中寻找频度最小的两个节点，建立新父节点并双向链接；新父节点加入链表末尾，循环往复直到只剩根节点。

4. 哈夫曼编码生成与结果输出事件

对每个初始字符节点自底向上遍历parent链，生成哈夫曼编码并写入code字段；按nodes顺序输出字符、频率、编码信息，换行符特判为’\n’格式输出；顺序统计所有字符的frequency × code长度，得到WPL并输出。

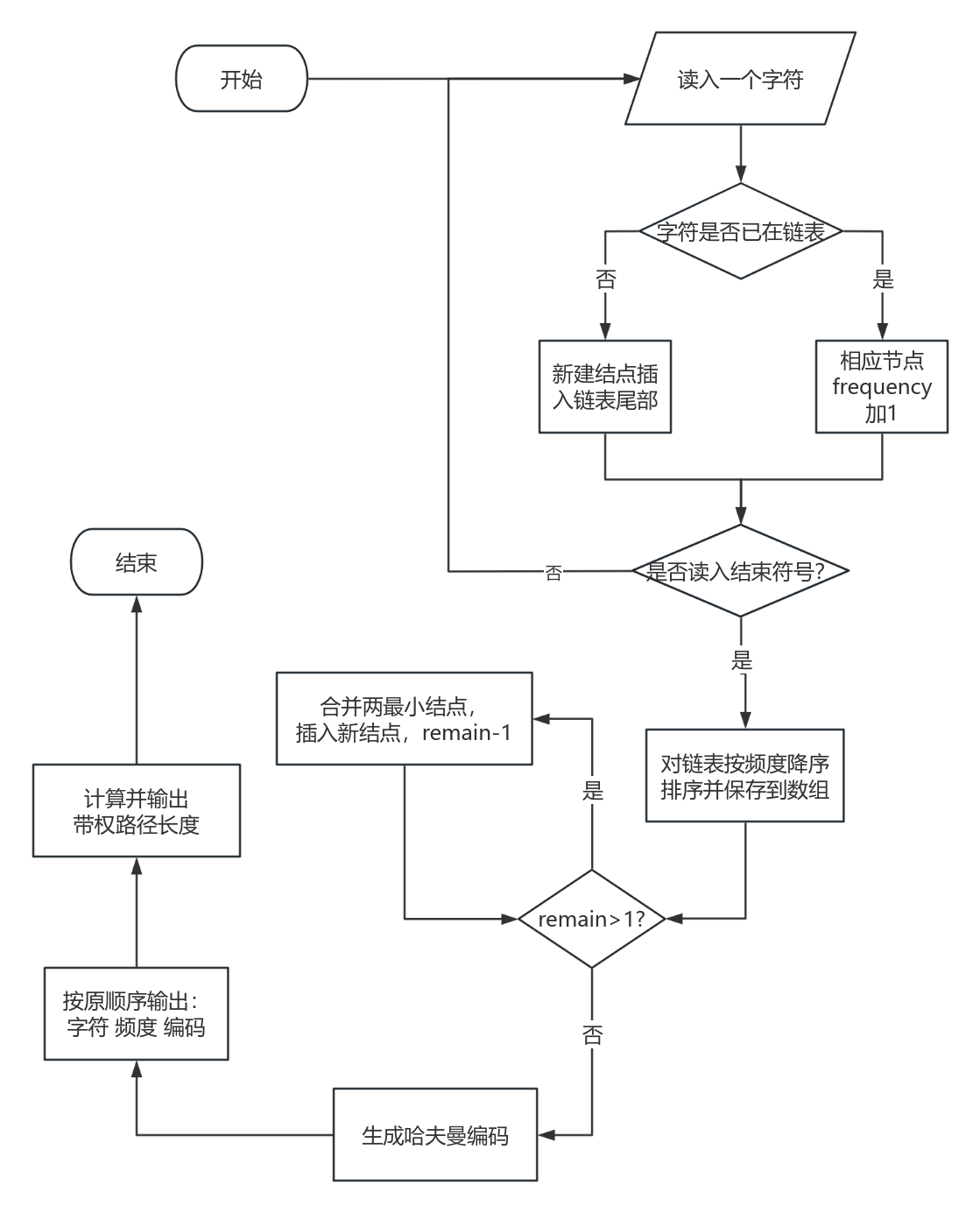


图2-1 建立哈夫曼树并生成哈夫曼编码事件的完成流程

## 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 |
| int getListLength(ListNode \*head) | 计算并返回链表的节点数量 |
| void sortList(ListNode \*\*headRef) | 对链表按frequency字段进行稳定降序排序 |
| int main() | 整体流程控制，包含字符频度统计、排序、哈夫曼树构建、编码生成、输出 |

## 2.4 用例测试

使用头歌及自测测试数据的结果如图2-2，图2-3，图2-4所示。

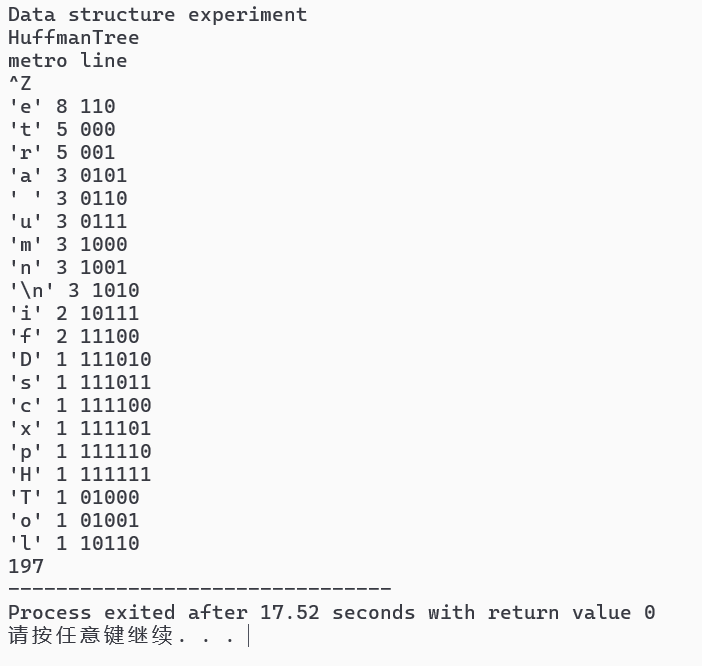


图2-2 测试数据1在本地的运行结果

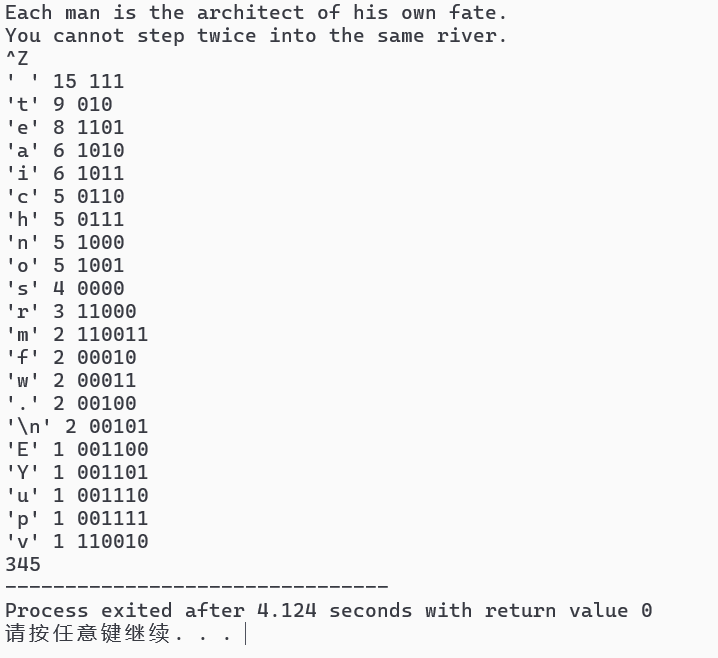


图2-3 测试数据2在本地的运行结果

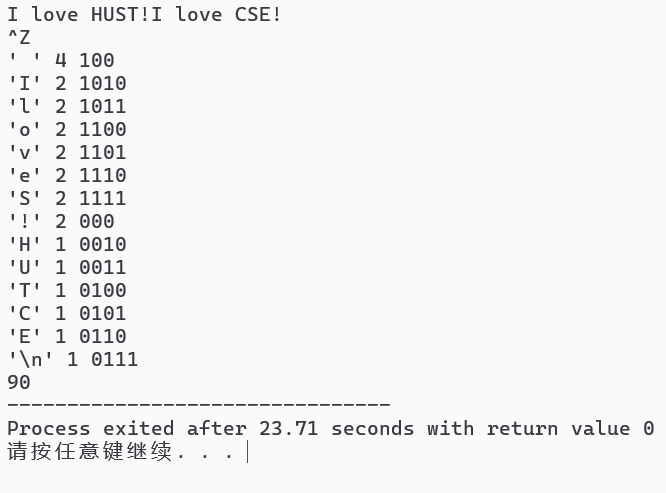


图2-4 测试数据3在本地的运行结果

## 2.5 复杂度分析

时间复杂度分析：第一，在统计每个字符出现频率时，需要对当前链表查找，最坏情况下复杂度为（n为总字符数，m为不同字符种类数）；第二，在构建哈夫曼树及排序过程中，因多次遍历和冒泡排序，复杂度为。因此，整体时间复杂度为。

空间复杂度：主要用于存储不同字符节点和哈夫曼树节点，总体为。

## 2.6 实验小结

本关实验整体难度适中，在理清设计思路后，程序的实现较为顺利。实验开始时，借助上一关构建的频度链表作为哈夫曼树的初始结点，按照哈夫曼树的构建原则，反复在链表中查找频度最小且尚未指定双亲的两个结点，生成父结点并插入链表，逐步完成了哈夫曼树的搭建。

在实验过程中，主要的挑战在于如何准确地在链表中定位频度最小且未指定双亲的结点，以及如何保证哈夫曼编码的输出顺序与输入顺序一致。为此，查阅了相关资料，并向老师请教，最终通过遍历链表查找合适结点，并利用数组保存原始顺序，有效地解决了上述问题。同时，对特殊字符（如换行符）的处理也给予了关注，确保了输出格式的规范性。

本次实验不仅加深了对哈夫曼树构建方法的理解，也提升了链表与二叉树结合应用的能力。在实际编程过程中，进一步体会到数据结构理论与实践的紧密联系，并认识到在程序设计中细致处理边界和特殊情况的重要性。这些收获为今后更深入地学习和应用数据结构奠定了坚实的基础。

# 3 哈夫曼编码和解码

## 问题描述

本关任务“实现哈夫曼编码和解码的全过程”。首先，程序会统计输入文本中所有字符（包括换行符）的出现频率，并据此构建哈夫曼树。然后，通过哈夫曼树为每个字符生成唯一的哈夫曼编码表。实现过程中，需将原文本编码为二进制字符串，并利用哈夫曼树对编码结果进行解码，恢复为原始文本。最后，程序输出编码后的比特流、解码后的文本及编码长度。

## 系统设计

整个程序分为四个核心部分：哈夫曼树构建、编码生成、文本编码、文本解码。

### 3.2.1 数据结构设计

1. 哈夫曼树的抽象数据结构

下面是用于存储地铁线路图的数据结构

**ADT**HuffmanTreeList{

数据对象：HuffmanNode数组HT存储包含权重、父节点/左右子节点指针的树形结构；叶子节点字符集leaf\_char用来记录所有有效字符；编码表codes用以字符与二进制编码的映射关系。

数据关系：树形结构遵循哈夫曼树的构造规则（权值越大路径越短）；编码表通过后序遍历树结构生成，叶子节点与字符一一对应，非叶节点为合并节点。

基本操作：

CreateHuffman(&HT, text)

操作结果：初始化各字符的频率统计，所有节点的parent/left/right初始化为-1，构建基础HuffmanNode结构。

BuildHuffmanTree(&HT, num\_leaf)

初始条件：所有叶子节点数据已就位。

操作结果：根据最小权值优先组合原则构建哈夫曼树。

GenerateCodes(&HT, num\_leaf)

操作结果：为每个字符生成对应的二进制编码字符串。

Encode(&HT, text)

初始条件：哈夫曼编码表已生成。

操作结果：将输入文本转换为二进制编码字符串。

Decode(&HT, code\_str, root\_idx)

初始条件：哈夫曼树结构完整。

操作结果：根据编码流反向检索原始文本。

} **ADT** HuffmanTreeList

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 {

int weight; // 权重(字符出现次数)

int parent; // 父节点索引(根节点为-1)

int left; // 左子节点索引(-1表示无)

int right; // 右子节点索引(-1表示无)

} HuffmanTreeNode;

辅助结构体：

int freq[MAX\_CHAR]; // ASCII字符频率表

char \*leaf\_char; // 叶子节点字符数组

int char\_map[MAX\_CHAR]; // 字符→叶子索引的映射表

char \*\*codes; // 编码表指针数组

### 3.2.2执行流程设计

本程序由四个核心事件组成：频率统计、建树、编码、解码。具体流程如下：

1. 频率统计事件

输入流进入动态缓冲区，遍历每个字符，同时更新freq数组。特别注意的是，在处理换行符时，需要将其视为'\n'字符参与统计。

2. 建树事件

遍历freq数组，创建有效叶子节点，之后循环执行任务：先找出权重最小的两个未处理节点，再创建合并节点并建立父子关系,最后更新总权重直至形成完整树结构。

3. 编码生成事件

从叶子节点回溯到根节点，同时记录路径方向（左0右1）；动态分配编码字符串，并建立字符，最后编码映射表。

4. 编解码转换事件

编码流程：输入文本，逐字符查表替换，动态扩展编码缓冲区。在处理大文本时，可以采用倍增策略扩容缓冲区。

解码流程：编码流，从根节点开始树遍历，到达叶子节点时：首先追加对应字符到输出缓冲区，再重置指针到根节点继续遍历，在处理换行符时，通过leaf\_char数组正确还原。

事件的执行流程如图3-1所示。

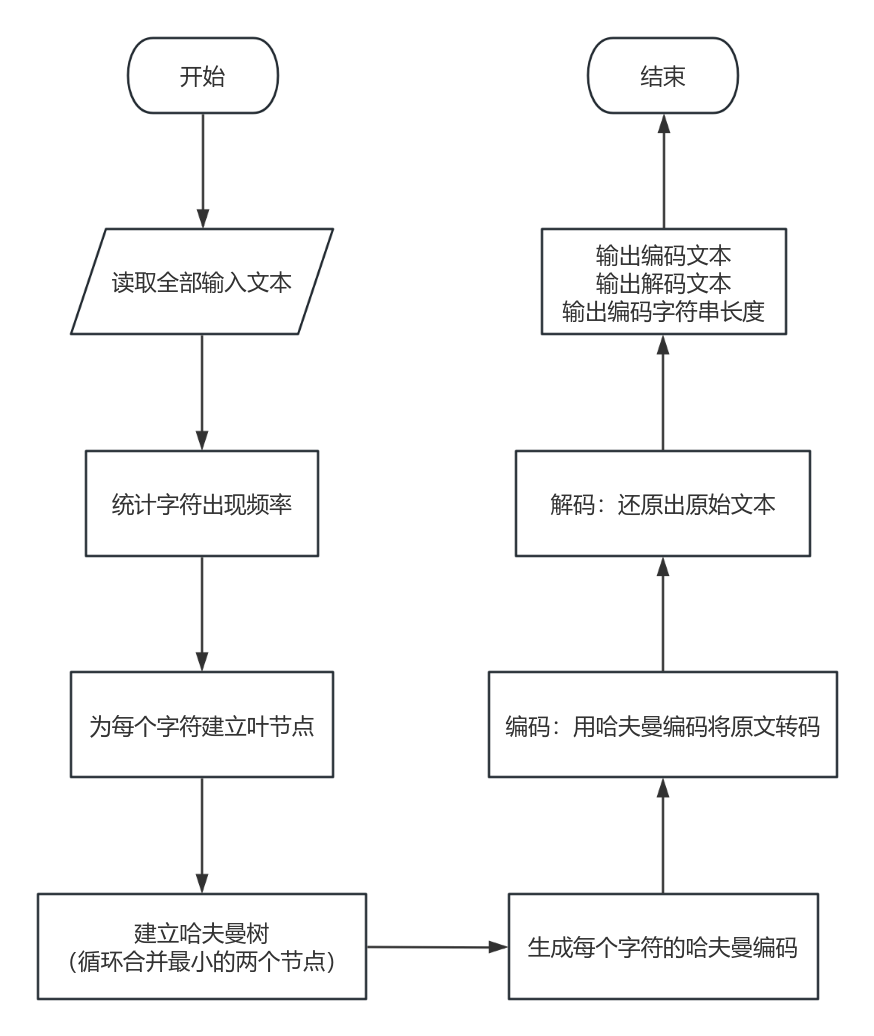


图3-1 哈夫曼编解码事件的完成流程

## 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 count\_freq(text, freq) | 统计输入文本中各字符的出现频率，填充频率数组 |
| int find\_min(HT, n) | 在哈夫曼节点数组前n项中查找权重最小的可用节点 |
| void build\_huffman\_tree(HT, num\_leaf) | 根据叶子节点数量构建完整的哈夫曼树结构 |
| void generate\_codes(HT, codes, …) | 通过回溯树路径生成所有字符的二进制编码 |
| char\* encode(text, codes, …) | 将原始文本逐字符转换为哈夫曼编码字符串 |
| char\* decode(HT, root, code, …) | 根据哈夫曼树遍历编码流还原原始文本 |

## 3.4 用例测试

使用头歌及自测测试数据的结果如图3-2，图3-3，图3-4所示。

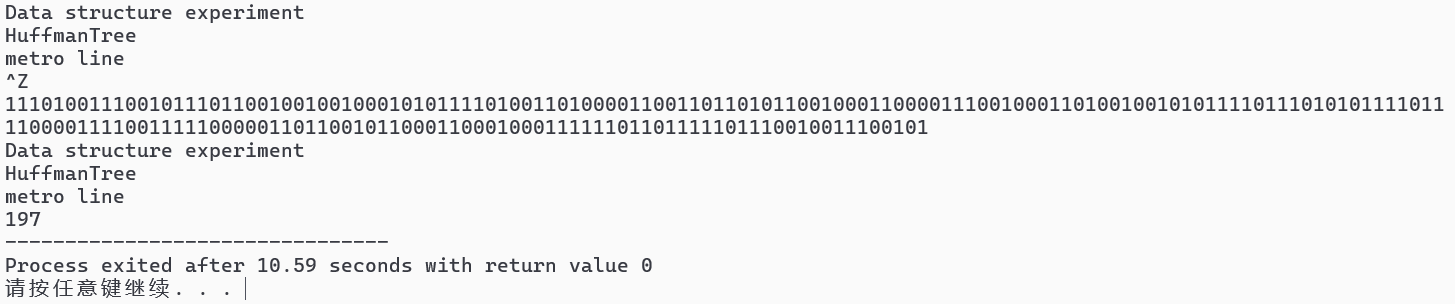


图3-2 测试数据1在本地的运行结果

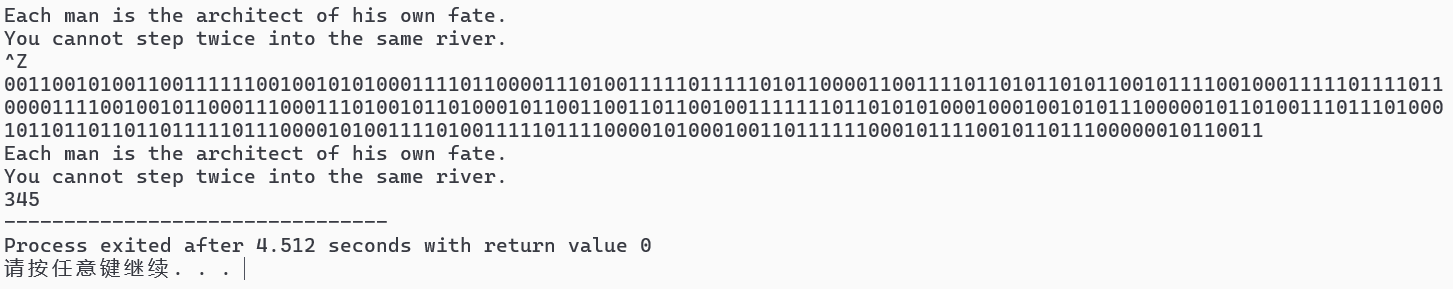


图3-3 测试数据2在本地的运行结果

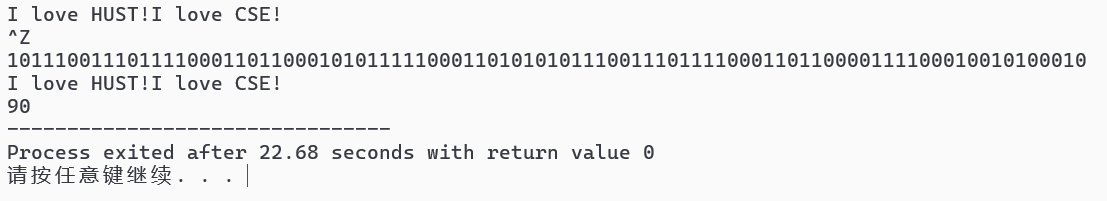


图3-4 测试数据3在本地的运行结果

## 3.5 复杂度分析

时间复杂度分析：该哈夫曼编码实现的时间复杂度为，其中n为输入文本长度。首先，在查找最小权值节点（find\_min）时，最好情况是第一个未处理的节点即为最小值，时间复杂度为，最坏情况需遍历全部未处理节点（共k个），时间复杂度为，该步骤单次操作平均复杂度：；之后，再创建哈夫曼树（build\_huffman\_tree）时，合并操作次数：m-1次（m为有效字符数），每次合并复杂度：两次find\_min调用（总次数为2\*(m-1)），总时间复杂度：，但是因find\_min嵌套调用，m最大为256，实际退化为；随后，在生成字符编码（generate\_codes）时，单字符编码生成：回溯路径长度等于树高h（最优树高为log₂m，最差树高为m），总时间复杂度：，但实际因 m 固定为 256，视为；最后，编码与解码（encode/decode），编码（encode）：遍历输入文本长度n，时间复杂度为，解码（decode）：遍历编码后二进制串长度 k，时间复杂度为，平均k≈8n，仍为。总时间复杂度：创建哈夫曼树：（因 m≤256）；生成编码表；编码与解码

空间复杂度为，主要由输入缓冲区、编码结果和动态分配的哈夫曼树节点（MAX\_NODE=1024）主导，字符频率表、编码表等辅助结构因规模固定（256种字符）仅占用常数空间，总体空间消耗与输入规模成线性关系。

## 3.6 实验小结

首先，在读取输入文本时，发现本地测试和头歌平台的结果不一致。具体表现为：本地能够正确输出编码和解码结果，但在平台上部分字符（尤其是换行符和空格）没有被正确处理，导致最终输出与预期不符。为了解决这个问题，我首先对比了本地和平台的输入输出方式，怀疑是输入读取方法存在漏洞。于是，我将原本按行读取的方式，改为按字符逐个读取，确保包括换行符、空格等所有字符都能被统计和编码。修改后，我增加了调试输出，逐步检查每个字符的处理情况，最终确认所有字符都被正确编码和解码。

此外，在实现哈夫曼树的过程中，我还遇到了优先队列（最小堆）节点比较的问题。起初直接用默认比较方法，结果在节点权值相同时出现了异常。为此，我重载了节点的比较函数，确保在权值相等时也能正确排序，避免了构建哈夫曼树时的歧义。

通过本次实验，我不仅掌握了哈夫曼树的构建、编码和解码的具体实现方法，还体会到细节处理在实际编程中的重要性。尤其是在跨平台开发时，输入输出的处理方式可能会影响程序的正确性。这次实验让我对数据结构的实际应用有了更深入的理解，也增强了解决实际问题的能力。

# 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] [Larry Nyhoff](http://www.calvin.edu/~nyhl/index.html). [ADTs, Data Structures, and Problem Solving with C++.](http://vig.prenhall.com/catalog/academic/product/0,1144,0131409093,00.html)Second Edition, [Calvin College](http://cs.calvin.edu/), 2005

[3] 严蔚敏等. 数据结构题集(C语言版). 清华大学出版社

[4] 王晓东. 数据结构(第3版). 高等教育出版社, 2018

[5] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser. Data Structures and Algorithms in Python. Wiley, 2013

[6] Clifford A. Shaffer. Data Structures and Algorithm Analysis Edition 3.2 (C++ Version). Dover Publications, 2022

[7] 王红梅, 王晓东. 数据结构与算法分析. 机械工业出版社, 2020

[8] Mark Allen Weiss. Data Structures and Algorithm Analysis in C++ (4th Edition). Pearson, 2014

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms (3rd Edition). MIT Press, 2009

[10] 邓俊辉. 数据结构(C++语言版). 清华大学出版社, 2013

[11] Timothy M. Henry, Frank M. Carrano. Data Structures and Abstractions with Java (5th Edition). Pearson, 2018

[12] 王道论坛编著. 数据结构考研复习指导. 电子工业出版社, 2021

# **附录一 Huffman编解码的操作实现**

**第一关：**

#include<stdio.h>

#include<stdlib.h>

typedef struct ListNode {

    char c;

    int frequency;

    char \*code;

    struct ListNode \*parent;

    struct ListNode \*left;

    struct ListNode \*right;

    struct ListNode \*next;

} ListNode, HuffmanTree;

// 获取链表长度

int getListLength(ListNode \*charListHead) {

    int listLength = 0;

    while (charListHead != NULL) {

        listLength++;

        charListHead = charListHead->next;

    }

    return listLength;

}

// 按频率降序排序链表

void sortList(ListNode \*\*headRef) {

    if (\*headRef == NULL || (\*headRef)->next == NULL) {

        return;

    }

    int listLength = getListLength(\*headRef);

    for (int i = 0; i < listLength - 1; i++) {

        ListNode \*\*pre = headRef;

        ListNode \*current = \*headRef;

        ListNode \*nextNode = current ? current->next : NULL;

        for (int j = 0; j < listLength - i - 1; j++) {

            if (current == NULL || nextNode == NULL) break;

            if (current->frequency < nextNode->frequency) {

                current->next = nextNode->next;

                nextNode->next = current;

                \*pre = nextNode;

                pre = &(nextNode->next);

                nextNode = current->next;

            } else {

                pre = &(current->next);

                current = nextNode;

                nextNode = nextNode->next;

            }

        }

    }

}

int main() {

    ListNode \*charListHead = NULL;

    int inputChar;

    while ((inputChar = getchar()) != EOF) {

        char ch = (char)inputChar;

        ListNode \*searchNode = charListHead;

        int found = 0;

        while (searchNode != NULL) {

            if (searchNode->c == ch) {

                searchNode->frequency++;

                found = 1;

                break;

            }

            searchNode = searchNode->next;

        }

        if (!found) {

            ListNode \*newNode = (ListNode \*)malloc(sizeof(ListNode));

            newNode->c = ch;

            newNode->frequency = 1;

            newNode->code = NULL;

            newNode->parent = NULL;

            newNode->left = newNode->right = NULL;

            newNode->next = NULL;

            if (charListHead == NULL) {

                charListHead = newNode;

            } else {

                ListNode \*tail = charListHead;

                while (tail->next != NULL) tail = tail->next;

                tail->next = newNode;

            }

        }

    }

    sortList(&charListHead);

    ListNode \*printNode = charListHead;

    while (printNode != NULL) {

        if (printNode->c == '\n') {

            printf("'\\n' %d\n", printNode->frequency);

        } else {

            printf("'%c' %d\n", printNode->c, printNode->frequency);

        }

        printNode = printNode->next;

    }

    return 0;

}

**第二关：**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

typedef struct ListNode {

    char c;

    int frequency;

    char \*code;

    struct ListNode \*parent;

    struct ListNode \*left;

    struct ListNode \*right;

    struct ListNode \*next;

} ListNode, HuffmanTree;

void sortList(ListNode \*\*headRef);

int getListLength(ListNode \*head);

int main() {

    ListNode \*head = NULL;

    int inputChar;

    // 建立频度链表

    while ((inputChar = getchar()) != EOF) {

        char ch = (char)inputChar;

        ListNode \*current = head;

        int found = 0;

        while (current) {

            if (current->c == ch) {

                current->frequency++;

                found = 1;

                break;

            }

            current = current->next;

        }

        if (!found) {

            ListNode \*node = (ListNode\*)malloc(sizeof(ListNode));

            node->c = ch;

            node->frequency = 1;

            node->parent = node->left = node->right = NULL;

            node->code = NULL;

            node->next = NULL;

            if (!head) head = node;

            else {

                ListNode \*tail = head;

                while (tail->next) tail = tail->next;

                tail->next = node;

            }

        }

    }

    // 排序链表（稳定降序）

    sortList(&head);

    // 保存原顺序

    int len = getListLength(head);

    ListNode \*\*nodes = (ListNode\*\*)malloc(len \* sizeof(ListNode\*));

    ListNode \*p = head;

    for (int i = 0; i < len; i++) {

        nodes[i] = p;

        p = p->next;

    }

    // 构建哈夫曼树

    int remain = len;

    while (remain > 1) {

        // 找两个最小且无parent的节点

        ListNode \*min1 = NULL, \*min2 = NULL;

        for (ListNode \*curr = head; curr; curr = curr->next) {

            if (curr->parent) continue;

            if (!min1 || curr->frequency < min1->frequency) {

                min2 = min1;

                min1 = curr;

            } else if (!min2 || curr->frequency < min2->frequency) {

                min2 = curr;

            }

        }

        if (!min2) break;

        // 创建父节点

        ListNode \*parent = (ListNode\*)malloc(sizeof(ListNode));

        parent->c = '\0';

        parent->frequency = min1->frequency + min2->frequency;

        parent->parent = parent->left = parent->right = NULL;

        parent->code = NULL;

        parent->next = NULL;

        min1->parent = parent;

        min2->parent = parent;

        parent->left = min1;

        parent->right = min2;

        // 插入链表末尾

        ListNode \*\*tail = &head;

        while (\*tail) tail = &(\*tail)->next;

        \*tail = parent;

        remain--;

    }

    // 生成编码

    for (int i = 0; i < len; i++) {

        char code[256] = {0};

        int idx = 0;

        ListNode \*curr = nodes[i];

        ListNode \*child = curr;

        while (curr->parent) {

            if (curr->parent->left == curr) code[idx++] = '0';

            else code[idx++] = '1';

            curr = curr->parent;

        }

        // 反转得到根到叶子的编码

        for (int j = 0; j < idx/2; j++) {

            char t = code[j];

            code[j] = code[idx-1-j];

            code[idx-1-j] = t;

        }

        nodes[i]->code = strdup(code);

    }

    // 输出结果

    for (int i = 0; i < len; i++) {

        if (nodes[i]->c == '\n')

            printf("'\\n' %d %s\n", nodes[i]->frequency, nodes[i]->code);

        else

            printf("'%c' %d %s\n", nodes[i]->c, nodes[i]->frequency, nodes[i]->code);

    }

    // 计算带权路径长度

    int wpl = 0;

    for (int i = 0; i < len; i++)

        wpl += nodes[i]->frequency \* strlen(nodes[i]->code);

    printf("%d", wpl);

    return 0;

}

int getListLength(ListNode \*head) {

    int cnt = 0;

    while (head) {

        cnt++;

        head = head->next;

    }

    return cnt;

}

void sortList(ListNode \*\*headRef) {

    int len = getListLength(\*headRef);

    for (int i = 0; i < len-1; i++) {

        ListNode \*\*pre = headRef;

        ListNode \*curr = \*headRef;

        ListNode \*next = curr->next;

        for (int j = 0; j < len-i-1; j++) {

            if (curr->frequency < next->frequency) {

                curr->next = next->next;

                next->next = curr;

                \*pre = next;

                pre = &(next->next);

                next = curr->next;

            } else {

                pre = &curr->next;

                curr = next;

                next = next->next;

            }

        }

    }

}

**第三关：**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#define MAX\_NODE 1024    // 扩展节点容量

#define MAX\_CHAR 256     // 支持所有8-bit字符

#define INPUT\_SIZE 100000// 输入缓冲区扩容

typedef struct {

    int weight;

    int parent, left, right;

} HuffmanNode;

// 动态内存版本频率统计

void count\_freq(const char \*text, int \*freq) {

    for (int i = 0; text[i]; i++)

        freq[(unsigned char)text[i]]++;

}

int find\_min(HuffmanNode \*HT, int n) {

    int min\_val = INT\_MAX, idx = -1;

    for (int i = 0; i < n; i++) {

        if (HT[i].parent == -1 && HT[i].weight > 0 && HT[i].weight < min\_val) {

            min\_val = HT[i].weight;

            idx = i;

        }

    }

    return idx;

}

void build\_huffman\_tree(HuffmanNode \*HT, int num\_leaf) {

    int total = 2 \* num\_leaf - 1;

    for (int i = num\_leaf; i < total; i++) {

        int m1 = find\_min(HT, i);

        if (m1 == -1) break;

        HT[m1].parent = i;

        int m2 = find\_min(HT, i);

        if (m2 == -1) break;

        HT[m2].parent = i;

        HT[i].weight = HT[m1].weight + HT[m2].weight;

        HT[i].left = m1;

        HT[i].right = m2;

        HT[i].parent = -1;

    }

}

void generate\_codes(HuffmanNode \*HT, char \*\*codes, char \*leaf\_char, int num\_leaf) {

    char \*stack = malloc(MAX\_CHAR \* sizeof(char));

    for (int i = 0; i < num\_leaf; i++) {

        int top = MAX\_CHAR - 1;

        stack[top] = '\0';

        int cur = i, parent = HT[cur].parent;

        while (parent != -1) {

            top--;

            stack[top] = (HT[parent].left == cur) ? '0' : '1';

            cur = parent;

            parent = HT[cur].parent;

        }

        codes[i] = strdup(stack + top);

    }

    free(stack);

}

char\* encode(const char \*text, char \*\*codes, int \*char\_map) {

    size\_t capacity = strlen(text) \* MAX\_CHAR;

    char \*output = malloc(capacity);

    size\_t pos = 0;

    for (int i = 0; text[i]; i++) {

        char \*code = codes[char\_map[(unsigned char)text[i]]];

        size\_t code\_len = strlen(code);

        if (pos + code\_len >= capacity) {

            capacity \*= 2;

            output = realloc(output, capacity);

        }

        strcpy(output + pos, code);

        pos += code\_len;

    }

    output[pos] = '\0';

    return output;

}

char\* decode(HuffmanNode \*HT, int root, const char \*code, char \*leaf\_char) {

    size\_t capacity = strlen(code) / 8 + 1;

    char \*output = malloc(capacity);

    size\_t pos = 0;

    int node = root;

    for (int i = 0; code[i]; i++) {

        node = (code[i] == '0') ? HT[node].left : HT[node].right;

        if (HT[node].left == -1) {

            if (pos >= capacity - 1) {

                capacity \*= 2;

                output = realloc(output, capacity);

            }

            output[pos++] = leaf\_char[node];

            node = root;

        }

    }

    output[pos] = '\0';

    return output;

}

int main() {

    // 动态读取输入

    char \*input = malloc(INPUT\_SIZE);

    size\_t input\_len = 0;

    int c;

    while ((c = getchar()) != EOF && input\_len < INPUT\_SIZE - 1)

        input[input\_len++] = c;

    input[input\_len] = '\0';

    // 频率统计

    int \*freq = calloc(MAX\_CHAR, sizeof(int));

    count\_freq(input, freq);

    // 初始化哈夫曼结构

    HuffmanNode \*HT = calloc(MAX\_NODE, sizeof(HuffmanNode));

    char \*leaf\_char = malloc(MAX\_NODE);

    int \*char\_map = calloc(MAX\_CHAR, sizeof(int));

    int num\_leaf = 0;

    // 创建叶子节点

    for (int i = 0; i < MAX\_CHAR; i++) {

        if (freq[i] > 0) {

            HT[num\_leaf] = (HuffmanNode){freq[i], -1, -1, -1};

            leaf\_char[num\_leaf] = (char)i;

            char\_map[i] = num\_leaf++;

        }

    }

    build\_huffman\_tree(HT, num\_leaf);

    // 生成编码表

    char \*\*codes = calloc(num\_leaf, sizeof(char\*));

    generate\_codes(HT, codes, leaf\_char, num\_leaf);

    // 执行编码

    char \*encoded = encode(input, codes, char\_map);

    // 执行解码

    char \*decoded = decode(HT, 2\*num\_leaf-2, encoded, leaf\_char);

    // 输出结果

    printf("%s\n%zu", decoded, strlen(encoded));

    // 释放内存

    free(input);

    free(freq);

    free(HT);

    free(leaf\_char);

    free(char\_map);

    for (int i = 0; i < num\_leaf; i++) free(codes[i]);

    free(codes);

    free(encoded);

    free(decoded);

    return 0;

}

# **附录二 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。