哈尔滨工业大学计算学部

实验报告

课程名称：数据结构与算法

课程类型：专业核心基础课（必修）

实验项目：树形结构及其应用

实验题目：哈夫曼编码与译码方法

实验日期：2024 年 4 月 21 日

班级：22WL022

学号：2022110829

姓名：杨明达

设计成绩	报告成绩	任课老师
		张岩

一、实验目的

哈夫曼编码是一种以哈夫曼树（最优二叉树，带权路径长度最小的二叉树）为基础变长编码方法。其基本思想是：将使用次数多的字符转换成长度较短的编码，而使用次数少的采用较长的编码，并且保持编码的唯一可解性。在计算机信息处理中，经常应用于数据压缩。是一种一致性编码法（又称"熵编码法"），用于数据的无损压缩。本实验要求实现一个完整的哈夫曼编码与译码系统。

二、实验要求及实验环境

实验要求：

1. 从文件中读入任意一篇英文文本文件，分别统计英文文本文件中各字符（包括标点符号和空格）的使用频率。
2. 根据已统计的字符使用频率构造哈夫曼编码树，并给出每个字符的哈夫曼编码（字符集的哈夫曼编码表）。
3. 将文本文件利用哈夫曼树进行编码，存储成压缩文件（哈夫曼编码文件）；
4. 将哈夫曼编码文件译码为文本文件，并与原文件进行比较。
5. 计算你的哈夫曼编码文件的平均编码长度和压缩率，并与实验结果比较验证。
6. 利用堆结构（优先级队列），优化哈夫曼编码算法。

实验环境：Windows 11 && Visual Studio Code

三、设计思想（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系、核心算法的主要步骤）

1．逻辑设计

1.1 主程序的流程图

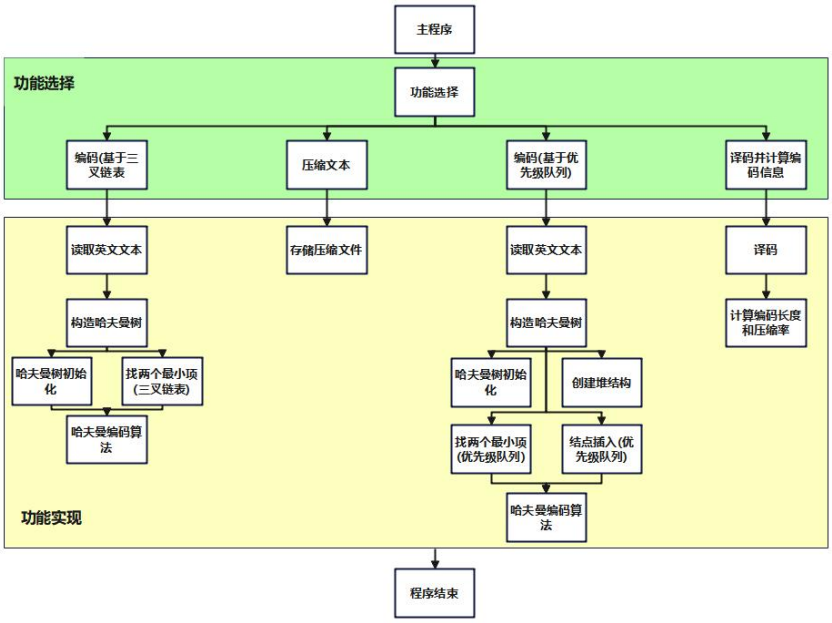

图 1 主程序流程图

1.2 各程序模块的调用关系

（1）从构造哈夫曼树（基于三叉链表）函数中调用哈夫曼树初始化、找到两个最小项（基于三叉链表）模块。

（2）从构造哈夫曼树（基于优先级队列）函数中调用哈夫曼树初始化、创建堆结构、找到两个最小项（基于优先级队列）、结点插入（优先级队列）模块。

（3）从创建堆结构函数中调用堆的初始化、结点插入（优先级队列）模块。

（4）从找到两个最小项（基于优先级队列）函数中调用删除最小项（基于优先级队列）模块。

（5）从删除最小项（基于优先级队列）模块调用判断堆空模块。

（6）从结点插入（基于优先级队列）函数调用判断堆满模块。

1.3 核心算法

（1）哈夫曼树构造（基于三叉链表）

在本次实验中，哈夫曼树的构造是实现哈夫曼编码的关键。在函数中，首先要对一个数初始化，然后进行 n-1 次合并，从哈夫曼树中找到两个权重最小的项，标记它的序号，之后建立双亲结点，使其的左右子树分别为这两个最小项，其权重为两个子树权重的和。

```
1.  void CreatHT(HuffmanCode H, HuffmanT T) {
2.      int p1, p2, n;
3.      n = Number_Tpye;
4.      InitHT(H, T);
5.      for (int i = n; i < 2 * n - 1; i++) {
6.          SelecMin(T, i, &p1, &p2);
7.          T[p1].parent = T[p2].parent = i;
8.          T[i].lchild = p1;
9.          T[i].rchild = p2;
10.         T[i].weight = T[p1].weight + T[p2].weight;
11.     }
12. }
```

（2）哈夫曼树构造（基于优先级队列）

在本次实验中，哈夫曼树的构造是实现哈夫曼编码的关键。在函数中，首先要对一个数初始化，然后创建堆，再进行 n-1 次合并，从哈夫曼树中找到两个权重最小的项，标记它的序号，之后建立双亲结点，使其的左右子树分别为这两个最小项，其权重为两个子树权重的和。

```
void CreatHT_Heap(HuffmanCode H, HuffmanT T) {
    int p1, p2, n;
    n = Number_Tpye;
    HEAP heap;
    InitHT(H, T);
    CreatHeap(H, &heap);
    for (int i = n; i < 2 * n - 1; i++) {
        Elementype new_node;
        SelecMin_Heap(&heap, &p1, &p2);
        T[p1].parent = T[p2].parent = i;
        T[i].lchild = p1;
        T[i].rchild = p2;
        T[i].weight = T[p1].weight + T[p2].weight;
        new_node.key = i;
        new_node.weight = T[i].weight;
        Heap_Insert(&heap, new_node);
    }
}
```

（3）哈夫曼编码算法的实现

在本次实验中，哈夫曼编码算法是将字符转换成 01 序列的关键所在，在函数中，c 和 p 分别指示 T 中的孩子和双亲的位置，cd 临时存放编码，start 指示编码在 cd 中的位置，第五行用来定义编码结束符。进入循环，依次求叶子 T[i]的编码：读入叶子 T[i]对应的字符，定义编码起始位置的初值，从叶子 T[i]开始上溯，进入循环，判断，若 T[c]是 T[p]的左孩子，则生成代码 0，否则生成代码 1，继续上溯直到上溯到 T[c]是树根位置。最后复制编码位串与编码表 H。

```
void CharSetHuffmanEncoding(HuffmanCode H, HuffmanT T) {
    int c, p, i;
    char cd[Number_Tpye];
    int start;
    cd[Number_Tpye] = '\0';
    for (i = 0; i < Number_Tpye; i++) {
        start = Number_Tpye;
        c = i;
        while ((p = T[c].parent) >= 0) {
            cd[--start] = (T[p].lchild == c) ? '0' : '1';
            c = p;
        }
        strcpy(H[i].bits, &cd[start]);
    }
}
```

2．物理设计（即存储结构设计）

本实验主要采用三叉链表存储字符，该存储结构包含四个数据，分别为权值（double 类型），左孩子链（int 类型），右孩子链（int 类型），双亲链（int 类型）。三叉链表的示意图如下图所示。

图 3 三叉链表存储

本实验存储字符结点，该存储结构包含三个数据，分别为字符（char 类型），字符编码（char 类型），字符频率（double 类型）。存储结构示意图如下图所示。

图 4 字符结点存储

本实验另一种方法主要利用优先级队列，即推结构存储字符，该存储结构包含两个主数据，分别为节点个数（int 类型），节点数据（Elementype 类型），其中 Elementype 类型由三个两个部分组成，分别为序号（int 类型），权重（double 类型）。堆结构的示意图如下图所示。

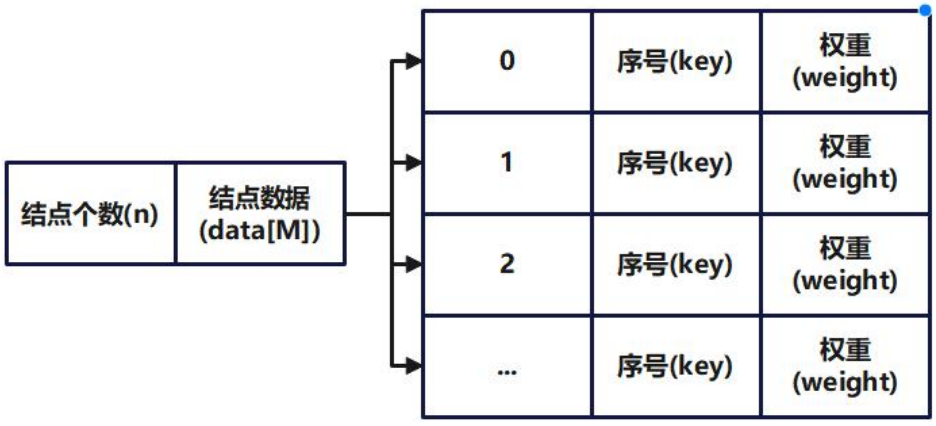

图 5 优先级队列存储

基于上述存储结构，实现哈夫曼编码的读取英文文本、构造哈夫曼树、存储压缩文本、解压等操作。

四、测试结果（包括测试数据、结果数据及结果的简单分析和结论，可以用截图得形式贴入此报告）

1. 基于三叉链表编码

```
请选择功能实现:1.编码(基于三叉链表);2.编码(基于优先级队列);3.压缩文本;4.译码并计算编码信息;0.退出
1
这个文本有5155个字符,有62种字符文本中各字符的哈夫曼编码及其频率:
字符
   编码:0100100    频率:0.006596
字符     编码:110    频率:0.151503
字符"    编码:1010101101    频率:0.001164
字符'    编码:001001110    频率:0.001552
字符,    编码:1111111    频率:0.010669
字符-    编码:01001101001    频率:0.000582
字符.    编码:11110100    频率:0.004850
字符0    编码:101010101    频率:0.002134
字符1    编码:111111001    频率:0.002522
字符2    编码:0100110101    频率:0.000970
字符3    编码:001001001    频率:0.001358
字符4    编码:01001100000    频率:0.000388
字符5    编码:0010011110    频率:0.000776
字符6    编码:11111100000    频率:0.000582
字符7    编码:01001100001    频率:0.000388
字符8    编码:01001100010    频率:0.000388
字符9    编码:11111100001    频率:0.000582
字符A    编码:1010101100    频率:0.000970
字符B    编码:11111100010    频率:0.000582
字符C    编码:11110101    频率:0.004850
字符D    编码:0100110011110    频率:0.000194
字符E    编码:11111100011    频率:0.000582
字符H    编码:001001100    频率:0.001358
字符I    编码:0010010000    频率:0.000582
字符J    编码:01001100011    频率:0.000388
字符L    编码:01001100100    频率:0.000388
字符M    编码:1010101110    频率:0.001164
字符N    编码:0010010001    频率:0.000582
字符O    编码:01001100101    频率:0.000388
字符P    编码:0100110011111    频率:0.000194
字符R    编码:0100110100000    频率:0.000194
字符S    编码:0100101    频率:0.006984
字符T    编码:00100101    频率:0.002716
字符U    编码:11111101    频率:0.005238
字符W    编码:0100111011    频率:0.001940
字符Y    编码:0100110100001    频率:0.000194
字符Z    编码:01001100110    频率:0.000388
字符a    编码:0111    频率:0.060912
字符b    编码:001000    频率:0.010863
字符c    编码:00101    频率:0.025218
字符d    编码:10010    频率:0.031814
字符e    编码:000    频率:0.090786
```

图 6 基于三叉链表编码 a

```
字符f    编码:101011    频率:0.017265
字符g    编码:100110    频率:0.015907
字符h    编码:10100    频率:0.033754
字符i    编码:1011    频率:0.074491
字符j    编码:1010101111    频率:0.001164
字符k    编码:001001101    频率:0.001358
字符l    编码:01000    频率:0.027158
字符m    编码:100111    频率:0.016877
字符n    编码:1000    频率:0.064210
字符o    编码:0101    频率:0.059360
字符p    编码:111100    频率:0.019011
字符r    编码:0011    频率:0.047915
字符s    编码:0110    频率:0.060718
字符t    编码:1110    频率:0.075267
字符u    编码:111110    频率:0.020563
字符v    编码:1010100    频率:0.008341
字符w    编码:0100111    频率:0.007565
字符x    编码:101010100    频率:0.001940
字符y    编码:1111011    频率:0.009893
字符z    编码:0010011111    频率:0.000776
请选择功能实现:1.编码(基于三叉链表);2.编码(基于优先级队列);3.压缩文本;4.译码并计算编码信息;0.退出
3
请选择功能实现:1.编码(基于三叉链表);2.编码(基于优先级队列);3.压缩文本;4.译码并计算编码信息;0.退出
4
这篇文章有 5155 个字符，共 62 种字符种类数。
哈夫曼树编码长度为:4.462658
编码文件的理论压缩率为:0.442168
编码文件的真实压缩率为:0.592214
请选择功能实现:1.编码(基于三叉链表);2.编码(基于优先级队列);3.压缩文本;4.译码并计算编码信息;0.退出
0
PS E:\C_Code\experiment_2>
```

图 7 基于三叉链表编码 b

2. 基于优先级队列编码

图 8 基于优先级队列编码 a

图 9 基于优先级队列编码 b

3. 输入文本

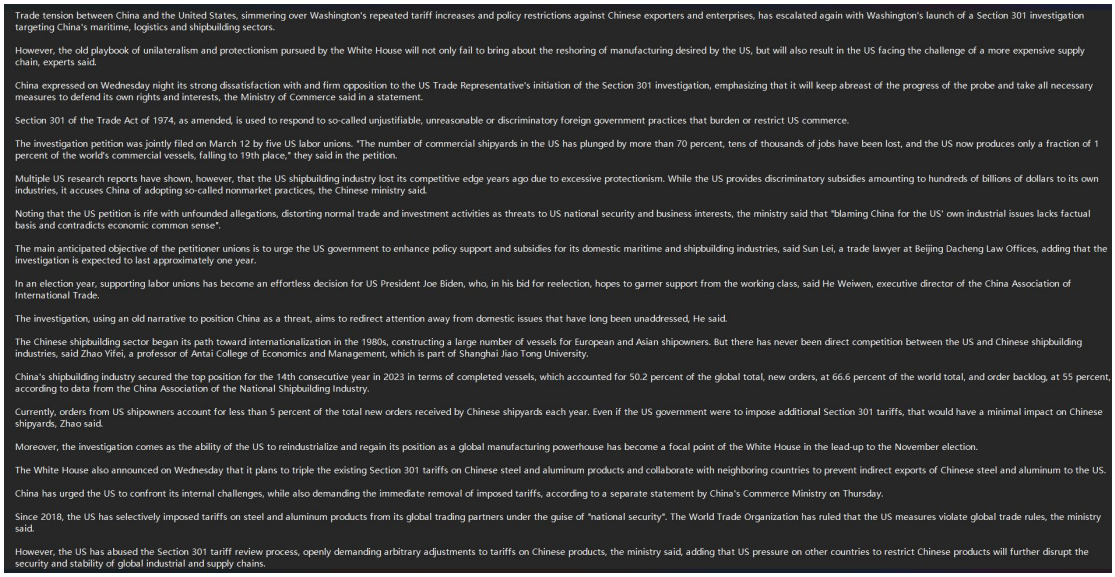

图 9 输入文本

4. 输出文本

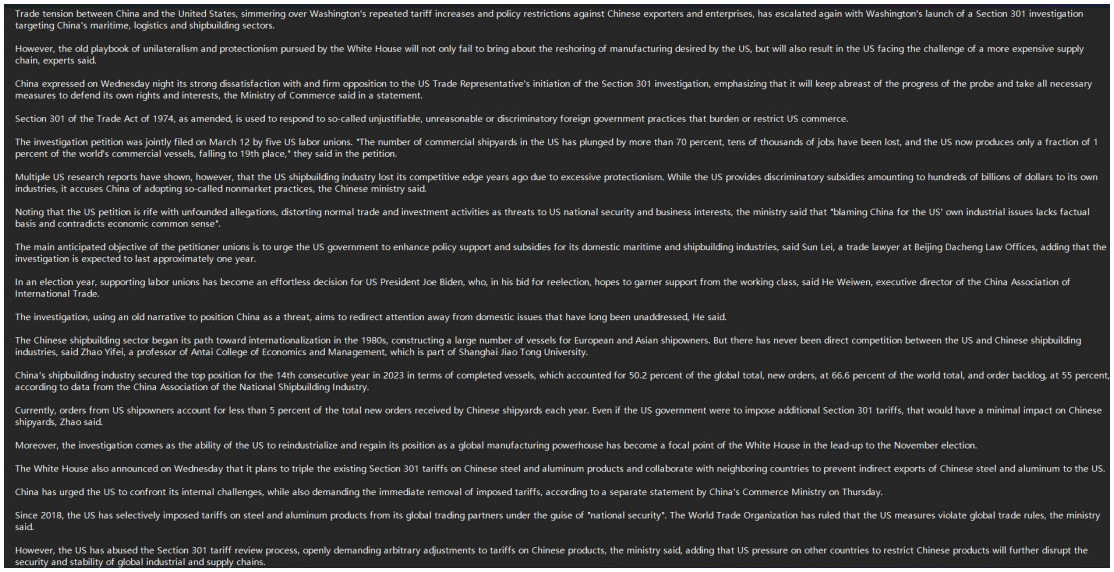

图 10 输出文本

五、经验体会与不足

经验体会：通过实现一个完整的哈夫曼编码与译码系统，熟练掌握了三叉链表链式存储结构和优先级队列（即堆结构）的实现及相关操作，巩固了对存储结构的认识，熟练 c 语言文件操作函数，熟练理解哈夫曼编码的原理到算法实现再到编码文件的压缩与解压的全过程。

不足：程序中的一些算法编写的不够高效简洁，还存在时间和空间的不必要消耗，并且程序的输出端口编写的也存在不规范之处。

六、附录：源代码（带注释）

```c
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#define N 128
#define M 2 * N - 1
// 三叉链表存储结构
typedef struct {
    double weight;
    int lchild;
    int rchild;
    int parent;
} HTNODE;
typedef HTNODE HuffmanT[M];
// 编码表的存储结构
typedef struct {
    char ch;
    char bits[M];
    double frequency;
} CodeNode;
typedef CodeNode HuffmanCode[N];
// 最小堆存储结构
typedef struct {
    int key;
    double weight;
} Elementype;
typedef struct {
    Elementype data[M];
    int n;
} HEAP;

long long Number_Character;  // 字符总数
int Number_Tpye;             // 字符种类数
// 堆的初始化
void Heap_Init(HEAP *heap) { heap->n = 0; }
// 判断堆空
int Heap_Empty(HEAP *heap) { return (!heap->n); }
// 判断堆满
int Heap_Full(HEAP *heap) { return (heap->n == N - 1); }
```

```
42.  // 结点插入(基于优先级队列)
43.  void Heap_Insert(HEAP *heap, Elementype item) {
44.      int i;
45.      if (!Heap_Full(heap)) {
46.          heap->n++;
47.          i = heap->n;
48.          while ((i != 1) && (item.weight < heap->data[i /
   2].weight)) {
49.              heap->data[i] = heap->data[i / 2];
50.              i /= 2;
51.          }
52.      }
53.      heap->data[i].key = item.key;
54.      heap->data[i].weight = item.weight;
55.  }
56.  // 删除最堆的最小项
57.  Elementype Heap_DeleteMin(HEAP *heap) {
58.      int parent = 1;
59.      int child = 2;
60.      Elementype item, tmp;
61.      if (!Heap_Empty(heap)) {
62.          item = heap->data[1];
63.          tmp = heap->data[heap->n--];
64.          while (child <= heap->n) {
65.              if ((child < heap->n) &&
66.                   (heap->data[child].weight > heap->data[ch
   ild + 1].weight)) {
67.                  child++;
68.              }
69.              if (tmp.weight <= heap->data[child].weight) {

70.                  break;
71.              }
72.              heap->data[parent] = heap->data[child];
73.              parent = child;
74.              child *= 2;
75.          }
76.      }
77.      heap->data[parent] = tmp;
78.      return item;
79.  }
80.  // 读取英文文本，统计字符总数和字符种类数
81.  void ReadText(HuffmanCode H) {
82.      int flag;
```

```
    int number_total, type_total;
    char Single_Character;
    int Character_Count[N] = {0};
    flag = 0;
    number_total = 0;
    type_total = 0;
    FILE *file = fopen("./Input/InputText.txt", "r");
    if (file == NULL) {
        printf("打开文件失败\n");
        exit(1);
    }
    while ((Single_Character = fgetc(file)) != EOF) {
        if (isascii(Single_Character)) {
            Character_Count[(int)Single_Character]++;
            number_total++;
        }
    }
    fclose(file);
    for (int i = 0; i < N; i++) {
        if (Character_Count[i]) {
            H[flag].ch = i;
            H[flag].frequency = (double)Character_Count[i] / number_total;
            flag++;
            type_total++;
        }
    }
    Number_Character = number_total;
    Number_Tpye = type_total;
}
// 哈夫曼树初始化
void InitHT(HuffmanCode H, HuffmanT T) {
    for (int i = 0; i < Number_Tpye; i++) {
        T[i].weight = H[i].frequency;
    }
    for (int i = 0; i < M; i++) {
        T[i].parent = -1;
        T[i].lchild = -1;
        T[i].rchild = -1;
    }
}
// 找到两个最小项(基于三叉链表)
void SelecMin(HuffmanT T, int n, int *p1, int *p2) {
    int i, j, temp;
```

```
126.     for (i = 0; i < n; i++) {
127.         if (T[i].parent == -1) {
128.             *p1 = i;
129.             break;
130.         }
131.     }
132.     for (j = i + 1; j < n; j++) {
133.         if (T[j].parent == -1) {
134.             *p2 = j;
135.             break;
136.         }
137.     }
138.     for (i = 0; i < n; i++) {
139.         if ((T[*p1].weight > T[i].weight) && (T[i].parent
    == -1) &&
140.             (*p2 != i)) {
141.             *p1 = i;
142.         }
143.     }
144.     for (j = 0; j < n; j++) {
145.         if ((T[*p2].weight > T[j].weight) && (T[j].parent
    == -1) &&
146.             (*p1 != j)) {
147.             *p2 = j;
148.         }
149.     }
150.     if (T[*p1].weight > T[*p2].weight) {
151.         temp = *p1;
152.         *p1 = *p2;
153.         *p2 = temp;
154.     }
155. }
156. // 构造哈夫曼树(基于三叉链表)
157. void CreatHT(HuffmanCode H, HuffmanT T) {
158.     int p1, p2, n;
159.     n = Number_Tpye;
160.     InitHT(H, T);
161.     for (int i = n; i < 2 * n - 1; i++) {
162.         SelecMin(T, i, &p1, &p2);
163.         T[p1].parent = T[p2].parent = i;
164.         T[i].lchild = p1;
165.         T[i].rchild = p2;
166.         T[i].weight = T[p1].weight + T[p2].weight;
167.     }
```

```
168. }
169. // 哈夫曼编码算法的实现
170. void CharSetHuffmanEncoding(HuffmanCode H, HuffmanT T) {
171.     int c, p, i;
172.     char cd[Number_Tpye];
173.     int start;
174.     cd[Number_Tpye] = '\0';
175.     for (i = 0; i < Number_Tpye; i++) {
176.         start = Number_Tpye;
177.         c = i;
178.         while ((p = T[c].parent) >= 0) {
179.             cd[--start] = (T[p].lchild == c) ? '0' : '1';
180.             c = p;
181.         }
182.         strcpy(H[i].bits, &cd[start]);
183.     }
184. }
185. // 利用哈夫曼树进行编码，存储成压缩文件
186. void Compressed(HuffmanCode H, HuffmanT T) {
187.     char Single_Character;
188.     FILE *file;
189.     file = fopen("./Input/InputText.txt", "r");
190.     if (!file) {
191.         printf("打开文件失败\n");
192.         exit(1);
193.     }
194.     FILE *bitfile;
195.     bitfile = fopen("./Output//Output.Huffman", "wb");
196.     if (!bitfile) {
197.         printf("打开文件失败\n");
198.         exit(1);
199.     }
200.     // 压入字符种类数
201.     fwrite(&Number_Tpye, sizeof(int), 1, bitfile);
202.     // 压入各字符
203.     for (int i = 0; i < Number_Tpye; i++) {
204.         fwrite(&H[i].ch, sizeof(char), 1, bitfile);
205.     }
206.     // 压入哈夫曼树节点
207.     for (int i = Number_Tpye; i < 2 * Number_Tpye - 1; i++) {
```

```
208.            fwrite(&T[i].lchild, sizeof(unsigned char), 1, bi
    tfile);
209.            fwrite(&T[i].rchild, sizeof(unsigned char), 1, bi
    tfile);
210.        }
211.        // 压入字符数
212.        fwrite(&Number_Character, sizeof(Number_Character), 1,
     bitfile);
213.        int flag;
214.        int bitflag;
215.        int i;
216.        unsigned char bit = 0;
217.        bitflag = 0;
218.        while ((Single_Character = fgetc(file)) != EOF) {
219.            for (i = 0; i < Number_Tpye; i++) {
220.                if (H[i].ch == Single_Character) {
221.                    break;
222.                }
223.            }
224.            for (flag = 0; H[i].bits[flag]; flag++) {
225.                if (H[i].bits[flag] == '0') {
226.                    bit = bit << 1;
227.                } else {
228.                    bit = (bit << 1) + 1;
229.                }
230.                bitflag++;
231.                if (bitflag == 8) {
232.                    fwrite(&bit, sizeof(unsigned char), 1, bi
    tfile);
233.                    bitflag = 0;
234.                    bit = 0;
235.                }
236.            }
237.        }
238.        if (bitflag) {
239.            bit = bit << (8 - bitflag);
240.            fwrite(&bit, sizeof(unsigned char), 1, bitfile);

241.            fwrite(&bitflag, sizeof(unsigned char), 1, bitfil
    e);
242.        }
243.        fclose(file);
244.        fclose(bitfile);
245. }
```

```
246. // 将哈夫曼编码文件译码为文本文件
247. void Decode(HuffmanCode H, HuffmanT T) {
248.     int type_total;
249.     long long number_total;
250.     FILE *bfile = fopen("./Output/Output.Huffman", "rb");
251.     FILE *file = fopen("./Output/OutputText.txt", "w");
252.     if (!bfile) {
253.         printf("打开文件失败\n");
254.         exit(1);
255.     }
256.     // 读取字符种类数
257.     fread(&type_total, sizeof(type_total), 1, bfile);
258.     // 读取各字符
259.     for (int i = 0; i < type_total; i++) {
260.         fread(&H[i].ch, sizeof(H[i].ch), 1, bfile);
261.     }
262.     // 读取哈夫曼树节点
263.     for (int i = type_total; i < 2 * type_total - 1; i++)
    {
264.         unsigned char p1, p2;
265.         fread(&p1, sizeof(p1), 1, bfile);
266.         fread(&p2, sizeof(p2), 1, bfile);
267.         T[i].lchild = (int)p1;
268.         T[i].rchild = (int)p2;
269.     }
270.     // 读取字符数
271.     fread(&number_total, sizeof(number_total), 1, bfile);
272.     printf("这篇文章有 %lld 个字符，共 %d 种字符种类数。
  \n", number_total,
273.            type_total);
274.     unsigned char b1;
275.     fread(&b1, sizeof(b1), 1, bfile);
276.     int i = 0;
277.     int j = 2 * type_total - 2;
278.     int total = 0;
279.     while (total < number_total) {
280.         if (((b1 >> (7 - i)) & 1))
281.             j = T[j].rchild;
282.         else
283.             j = T[j].lchild;
284.         i++;
285.         if (j < type_total) {
```

```
286.                 fprintf(file, "%c", H[j].ch);
287.                 total++;
288.                 j = 2 * type_total - 2;
289.             }
290.             if (i == 8) {
291.                 i = 0;
292.                 fread(&b1, sizeof(b1), 1, bfile);
293.             }
294.         }
295.         fclose(bfile);
296.         fclose(file);
297. }
298. // 计算哈夫曼树的编码长度和编码文件的压缩率
299. void Calculate(HuffmanCode H, HuffmanT T) {
300.     double lenth = 0;
301.     for (int i = 0; i < Number_Tpye; i++) {
302.         lenth = lenth + strlen(H[i].bits) * H[i].frequenc
   y;
303.     }
304.     printf("哈夫曼树编码长度为:%lf\n", lenth);
305.     double ratio_theoretical;
306.     double ratio_real;
307.     ratio_theoretical = 1 - lenth / 8;
308.             printf("编码文件的理论压缩率
   为:%lf\n", ratio_theoretical);
309.     struct _stat text;
310.     struct _stat bittext;
311.     double Size_Text;
312.     double Size_BitText;
313.     _stat("./Input/InputText.txt", &text);
314.     _stat("./Output/Output.Huffman", &bittext);
315.     Size_Text = text.st_size;
316.     Size_BitText = bittext.st_size;
317.     ratio_real = Size_BitText / Size_Text;
318.     printf("编码文件的真实压缩率为:%lf\n", ratio_real);
319. }
320. // 创建堆结构
321. void CreatHeap(HuffmanCode H, HEAP *heap) {
322.     Heap_Init(heap);
323.     for (int i = 0; i < Number_Tpye; i++) {
324.         Elementype new_node;
325.         new_node.key = i;
326.         new_node.weight = H[i].frequency;
327.         Heap_Insert(heap, new_node);
```

```
328.     }
329. }
330. // 找到两个最小项(基于优先级队列)
331. void SelecMin_Heap(HEAP *heap, int *p1, int *p2) {
332.     *p1 = Heap_DeleteMin(heap).key;
333.     *p2 = Heap_DeleteMin(heap).key;
334. }
335. // 构造哈夫曼树(基于优先级队列)
336. void CreatHT_Heap(HuffmanCode H, HuffmanT T) {
337.     int p1, p2, n;
338.     n = Number_Tpye;
339.     HEAP heap;
340.     InitHT(H, T);
341.     CreatHeap(H, &heap);
342.     for (int i = n; i < 2 * n - 1; i++) {
343.         Elementype new_node;
344.         SelecMin_Heap(&heap, &p1, &p2);
345.         T[p1].parent = T[p2].parent = i;
346.         T[i].lchild = p1;
347.         T[i].rchild = p2;
348.         T[i].weight = T[p1].weight + T[p2].weight;
349.         new_node.key = i;
350.         new_node.weight = T[i].weight;
351.         Heap_Insert(&heap, new_node);
352.     }
353. }
354. // 主函数
355. int main() {
356.     HuffmanCode H;
357.     HuffmanT T;
358.     HuffmanCode H_decode;
359.     HuffmanT T_decode;
360.     while (1) {
361.         int Function_Options;
362.         printf(
363.             "请选择功能实现:1.编码(基于三叉链表);2.编码(基于优先级队列);3."
364.             "压缩文本;"
365.             "4.译码并计算编码信息;0.退出\n");
366.         scanf("%d", &Function_Options);
367.         if (Function_Options == 1) {
368.             ReadText(H);
369.             CreatHT(H, T);
370.             CharSetHuffmanEncoding(H, T);
```

```
371.                 printf("这个文本有%d 个字符,有%d 种字符
   ", Number_Character,
372.                     Number_Tpye);
373.             printf("文本中各字符的哈夫曼编码及其频率:\n");
374.             for (int i = 0; i < Number_Tpye; i++) {
375.                         printf("字符%c      编码:%s      频
   率:%lf\n", H[i].ch, H[i].bits,
376.                             H[i].frequency);
377.             }
378.         } else if (Function_Options == 2) {
379.             ReadText(H);
380.             CreatHT_Heap(H, T);
381.             CharSetHuffmanEncoding(H, T);
382.                 printf("这个文本有%d 个字符,有%d 种字符
   ", Number_Character,
383.                     Number_Tpye);
384.             printf("文本中各字符的哈夫曼编码及其频率:\n");
385.             for (int i = 0; i < Number_Tpye; i++) {
386.                         printf("字符%c      编码:%s      频
   率:%lf\n", H[i].ch, H[i].bits,
387.                             H[i].frequency);
388.             }
389.         } else if (Function_Options == 3) {
390.             Compressed(H, T);
391.         } else if (Function_Options == 4) {
392.             Decode(H_decode, T_decode);
393.             Calculate(H, T);
394.         } else if (Function_Options == 0) {
395.             return 0;
396.         } else {
397.             printf("输入值非法,请重新输入\n");
398.         }
399.     }
400.     return 0;
```