

# 《数据结构》上机报告

2022 年 10 月 25 日

姓名：郑博远 学号：2154312 班级：计科 1 班 得分：

实验题目	哈夫曼编码和译码
实验目的	1. 理解最优二叉树，即哈夫曼树(Huffman tree)的概念； 2. 熟悉哈夫曼树的构造过程。
问题描述	<p>实现对 ASCII 字符文本进行 Huffman 压缩,并且能够进行解压。</p> <p>将给定的文本文件使用哈夫曼树进行压缩,并解压。</p> <p>用一个二叉树表示哈夫曼树,因为 ASCII 表一共只有 127 个字符,可以直接使用数组来构造 Huffman 树。</p> <p>leftChild和rightChild分别表示当前结点左儿子和右儿子的下标。因为 1 到 127 分别表示与 ASCII 表的符号相对应,所以这里无需记录每个结点代表什么符号。</p> <pre>struct Node{     int leftChild;     int rightChild; } tree[256];</pre> <p>而在解压的时候,由于不知道每个结点实际表示什么符号,所以需要要在树上记录下。</p> <pre>struct NodeV{     int leftChild;     int rightChild;     char c; } tree[256];</pre>
基本要求	1. 能将文本文件压缩、打印压缩后编码、解压压缩后的文件。对实际使用的具体数据结构除必须使用二叉树外不做要求; 2. 程序要添加适当的注释,程序的书写要采用缩进格式; 3. 程序要具在一定的健壮性,即当输入数据非法时, 程序也能适当地做出反应,如插入删除时指定的位置不对等等;

	<div>4. 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作；</div> <div>5. 根据实验报告模板详细书写实验报告, 在实验报告中给出主要算法的复杂度分析；</div>	
选做要求		
	已完成选做内容（序号）	
数据结构设计	<div>由于哈夫曼编码的长短不等，因此若要保证正确的读入与解码，则必须任意一个字符的编码都不是另一个字符的编码的前缀（称为前缀编码）。本题中，利用二叉树来设计二进制的前缀编码。由于存储时频次为零的结点不做记录，因此不采用题目描述中用数组下标代表字符 ASCII 码的方式，无论压缩与解压均统一在哈夫曼树结点信息中记录字符值 value。每一个哈夫曼树的结点存储权重（该字符出现频次）、字符值、父亲结点与左右子孩子结点。由于记录方式不同，不使用课本中 int 数组来传入频次值，而使用自定义的 CharWithNode 结构体数组，分别记录字符值与其出现次数。编码后的 01 字符串存放在二维数组中。最后，将哈夫曼树封装在类中，便于各个功能的使用。</div> <div>具体数据结构如下：</div> <div>//哈夫曼树的结点 typedef struct HTNode {     unsigned int weight;    //结点的权重     char value;    //结点的字符值     unsigned int parent, lchild, rchild;     //结点的双亲、左孩子、右孩子 }* HuffmanTree;  //存放字符出现频数的数组元素类型 struct CharWithNode {     char ch;    //字符     unsigned int weight; //权重-出现的次数 };  //哈夫曼编码表(二维数组) typedef char** HuffmanCode;</div>	

	<pre> /* 哈夫曼树的结构体 */ class HuffmanTree { private:     HTPointer tree;           //哈夫曼树数组     CharWithNode* char_list;  //字符出现频率的列表     HuffmanCode code_list;    //哈夫曼编码的列表     unsigned int n;           //字符种类数 } </pre>
功能(函数) 说明	<p>1. 哈夫曼树类的private成员函数</p> <pre> /**  * @brief 计算文件中每个字符出现的频数  * @param filename 输入的压缩文件  * @param filename1 输出的字符频数文件  * @param output 是否要输出到文件  */ void HuffmanTree::CalCharFreq(const char* filename, const char* filename1, bool output)  /**  * @brief 计算字符串中每个字符出现的频数(函数重载)  * @param str 输入的字符串  * @param output 是否要输出到文件  */ void HuffmanTree::CalCharFreq(const char* str)  /**  * @brief 读入字符频数文件信息写入list  * @param filename 读入的字符频数文件  */ void HuffmanTree::ReadCharFreq(const char* filename)  /**  * @brief 选择根结点中权值最小的两个结点  * @param max 从1-max的范围中查找  * @param s1 权值最小的结点  * @param s2 权值次小的结点  */ void HuffmanTree::Select(int max, int&amp; s1, int&amp; s2)  /**  * @brief 建立哈夫曼树 并计算对应每个字符的01串  */ void HuffmanTree::HuffmanCoding() </pre>

```

/**
 * @brief 获取某字符在list的下标
 * @param ch 要查找的字符
 */
int HuffmanTree::GetIndex(unsigned char ch)

/**
 * @brief 编码功能
 * @param infilename 待编码的文件名
 * @param outfilename 编码后的文件名
 */
void HuffmanTree::Inner_Encode(const char* infilename,
const char* outfilename)

/**
 * @brief 解码功能
 * @param infilename 待解码的文件名
 * @param outfilename 解码后的文件名
 */
void HuffmanTree::Inner_Decode(const char* infilename,
const char* outfilename)

/**
 * @brief 压缩功能
 * @param infilename 待压缩的文件名
 * @param outfilename 待解压的文件名
 */
void HuffmanTree::Inner_Compress(const char*
infilename, const char* outfilename)

/**
 * @brief 解压功能
 * @param infilename 待解压的文件名
 * @param outfilename 解压后的文件名
 */
void HuffmanTree::Inner-Decompress(const char*
infilename, const char* outfilename)

/**
 * @brief 打印哈夫曼树
 */
void HuffmanTree::PrintTree()

```

## 2. 哈夫曼树类的public成员函数

```
/**
 * @brief 构建哈夫曼树
 */
HuffmanTree::HuffmanTree()

/**
 * @brief 销毁哈夫曼树
 */
HuffmanTree::~HuffmanTree()

/**
 * @brief 编码模式
 * @param infilename 待编码文件
 * @param outfilelist 字符频数文件
 * @param outfilename 编码后文件
 */
void HuffmanTree::Encode(const char* infilename, const
char* outfilelist, const char* outfilename)

/**
 * @brief 解码模式
 * @param infilename 待解码文件
 * @param infilelist 字符频数文件
 * @param outfilename 解码后文件
 */
void HuffmanTree::Decode(const char* infilename, const
char* infilelist, const char* outfilename)

/**
 * @brief 压缩模式
 * @param infilename 待压缩文件
 * @param outfilename 压缩后文件
 */
void HuffmanTree::Compress(const char* infilename,
const char* outfilename)

/**
 * @brief 解压模式
 * @param infilename 待解压文件
 * @param outfilename 解压后文件
 */
void HuffmanTree::Decompress(const char* infilename,
const char* outfilename)
```

	<p>3. 队列类的成员函数（用于可视化打印哈夫曼树）</p> <p>（与PA2*作业中一致，此处省略）</p>
界面设计和使用说明	<div></div> <p>程序共分为 display 展示模式、（演示用）encode 编码模式、decode 解码模式以及（实际运用）compress 压缩模式、decompress 解压模式。当在命令行运行可执行文件不带参数或所带参数不符合程序正确输入时（如上图），程序能够给出提醒用户对应模式所需参数以及传入格式。下面对各个模式设计与功能进行详细说明：</p> <p>display 模式中，用户输入演示用的字符串后，程序统计各字符出现频数，并可视化展现构建的哈夫曼树。</p> <div></div> <p>encode 模式用于演示哈夫曼编码结果，输出的压缩文件将以文本形式的 0 与 1 呈现在文件当中，以使用户直观体会哈夫曼压缩后的编码结果。同时，各个字符的频数统计将会单独以 ASCII 码与出现频数被输出到文件中以便于 decode 解压（频数为 0 则省略）。命令行输入格式为“可执行文件名 --encode 待编码文件名 输出的字符频数文件名 编码后文件名”。具体如下图：</p>

```
命令提示符
C:\Users\BoyuanZheng\demo>Huffmancode --encode original.txt form.txt out.txt
文件original.txt已成功编码，请打开out.txt查看具体信息。
C:\Users\BoyuanZheng\demo>
```

样例输入中，待编码的文本文件 original.txt 内容如图所示：

```
original.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
Meet me at midnight
Oooh oooh oooh woah

Staring at the ceiling with you
Oh, you don't ever say too much
And you don't really read into
My melancholia
I've been under scrutiny (Yeah, oh yeah)
You handle it beautifully (Yeah, oh yeah)
All this shit is new to me (Yeah, oh yeah)

I feel the lavender haze creeping up on me
Surreal
I'm damned if I do give a damn what people say
No deal
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8
```

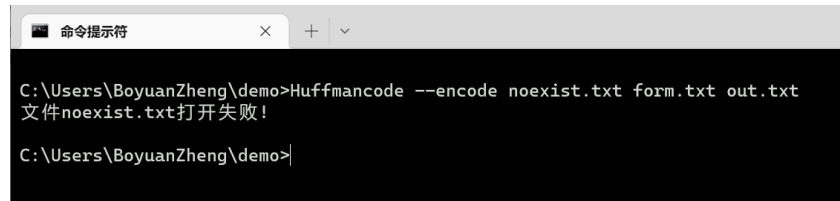
输出文件有两个，分别存放统计的字符出现频数与压缩后的 01 字符串。统计的字符出现频数信息将以每行 “[ASCII 码] [出现频数]” 的格式输出到指定的字符频数文件 form.txt 中：

```
form.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
10 60
13 60
32 261
39 12
40 11
41 11
44 7
45 1
48 3
49 3
53 3
57 3
65 4
66 1
71 7
73 22
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8
```

经过哈夫曼编码后的 01 字符串将以文本形式被输出到指定的编码文件 out.txt 中：

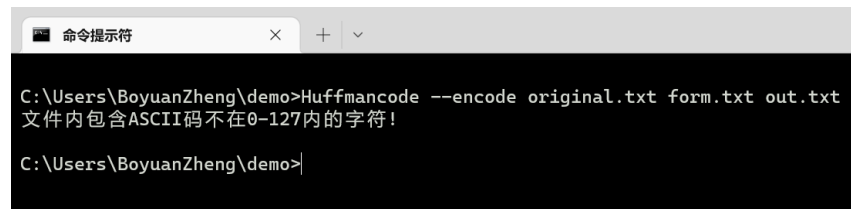
```
out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
1100101110000000101111100110001110100101111100111101001111101111010011000001101
011011101100100010001000011110100010001000011110100010001000011110010010100101000
11101110111100101100101101001101101101100011101001011101010001100011001011
100011010100101101011011000111001001110100101001111100010100101101101101001000
0110110010111100010100101101111011101001101100110001011100000101000001101111000
01010100011110101000100111100111011010010110011101111001010011011011111100010
10010110111101110100110111001100010111011010001010010100101000111101101000101001
1111111011011010101001011110010111010001111100110001001010101101100101110011010
01001011010101011101110010011000010100001110110010100000011011111011011011011110
000110111110000001011101101101011011011011000111101100111100100000010100011101
1001011101000011111100010001010001101110101011111001000010010110111100111011011011
111001000011111010010111011001010001010110101011101010011110110110010100101000111
101100111100100000010100011101100101110100001111110001000101000110111010101111100101
0010010100101110101001111010100001111000000111101001011111010100001111011000001001
111010101001111001100011011001111001000001010001110110010111010000111111000100010
```

若输入的文件或写入的文件无法正确打开，程序将会给出错误提示信息如下，提醒用户进行正确输入：



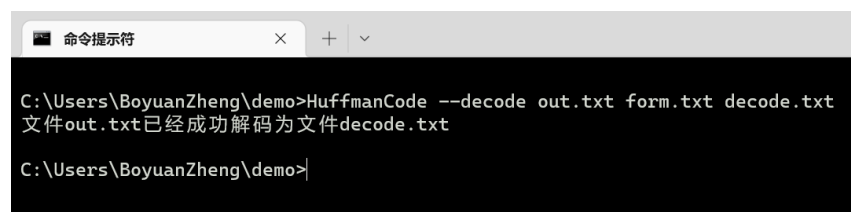
```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --encode noexist.txt form.txt out.txt
文件noexist.txt打开失败!
C:\Users\BoyuanZheng\demo>
```

若输入的文件中包含 ASCII 码不在 0-127 范围内的字符（测试样例中在原有的 original.txt 开头插入汉字字符“午夜”），则程序能够给出错误提示信息：



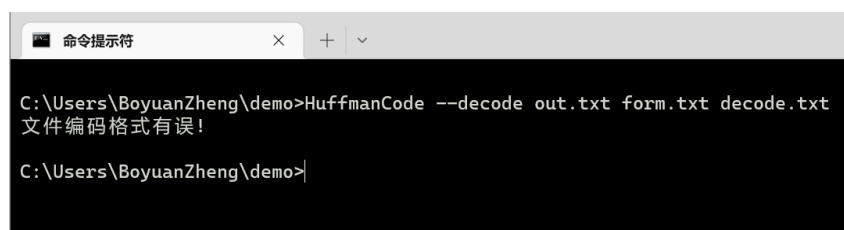
```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --encode original.txt form.txt out.txt
文件内包含ASCII码不在0-127内的字符!
C:\Users\BoyuanZheng\demo>
```

decode 模式与 encode 模式相对，用于演示哈夫曼编码文件的解压结果。decode 模式的输入分别为 encode 模式输出的两个文件，即记录各个字符出现频数的文件与经过哈夫曼编码后的 01 串文件；并能够将二者读入后解码到指定的输出文件中。命令行输入格式为“可执行文件名 --decode 待解码文件名 输入的字符频数文件名 解码后文件名”。具体如下图：



```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --decode out.txt form.txt decode.txt
文件out.txt已经成功解码为文件decode.txt
C:\Users\BoyuanZheng\demo>
```

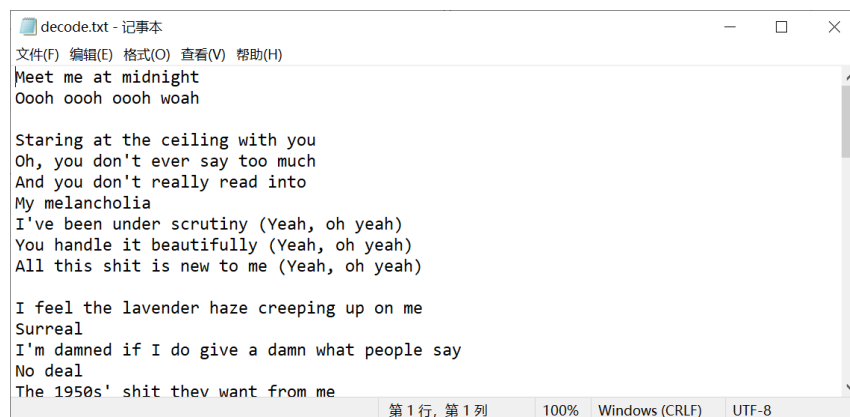
若输入的编码文件格式不正确（含有非 0、1 字符），则程序能够给出错误提示信息如下：



```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --decode out.txt form.txt decode.txt
文件编码格式有误!
C:\Users\BoyuanZheng\demo>
```



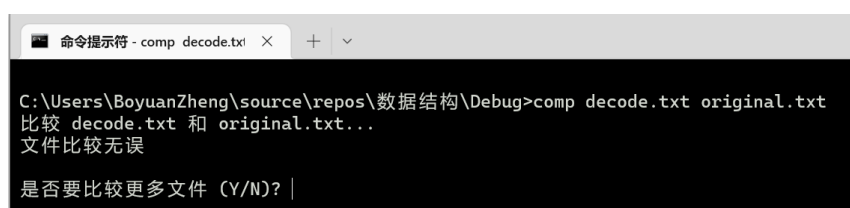
解码后输出的 decode.txt 文件如上图所示，与输入的 original.txt 文件相同，经 comp 比较无误。



```
decode.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
Meet me at midnight
Oooh oooh oooh woah

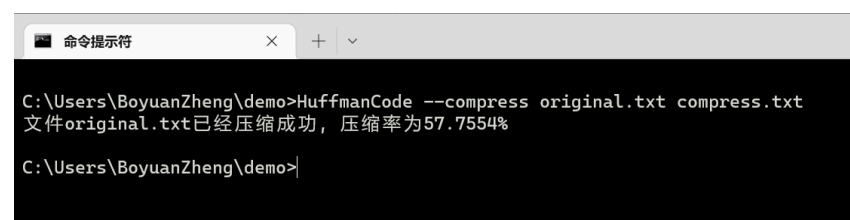
Staring at the ceiling with you
Oh, you don't ever say too much
And you don't really read into
My melancholia
I've been under scrutiny (Yeah, oh yeah)
You handle it beautifully (Yeah, oh yeah)
All this shit is new to me (Yeah, oh yeah)

I feel the lavender haze creeping up on me
Surreal
I'm damned if I do give a damn what people say
No deal
The 1950s' shit they want from me
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8
```



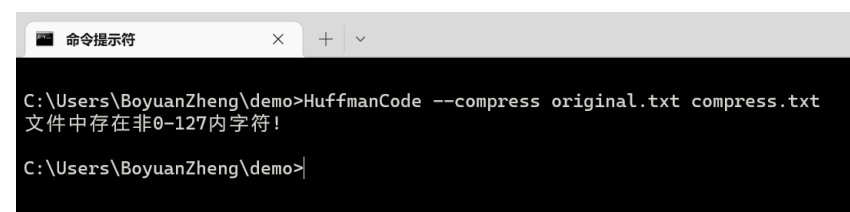
```
命令提示符 - comp decode.txt
C:\Users\BoyuanZheng\source\repos\数据结构\Debug>comp decode.txt original.txt
比较 decode.txt 和 original.txt...
文件比较无误
是否要比较更多文件 (Y/N)? |
```

compress 模式与 decompress 模式则用于压缩文件的实践。compress 模式中，压缩文件的全部信息将被以二进制形式（不同于 encode 模式中的文本模式 01 字符串）写入被压缩的文件中，模拟日常生活中的压缩软件。命令行输入格式为“可执行文件名 --compress 待压缩文件名 压缩后文件名”。具体功能展示如下：



```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --compress original.txt compress.txt
文件original.txt已经压缩成功，压缩率为57.7554%
C:\Users\BoyuanZheng\demo>
```

若文件压缩成功，则程序给出成功提示信息；并将压缩率（压缩后的文件大小 / 压缩前的文件大小 × 100%）打印；若文件中存在非法字符（不在 0-127 内），则程序给出错误提示信息如下：



```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --compress original.txt compress.txt
文件中存在非0-127内字符!
C:\Users\BoyuanZheng\demo>
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000h:	2E	0A	3C	00	00	00	0D	3C	00	00	00	20	05	01	00	00
00000010h:	27	0C	00	00	00	28	0B	00	00	00	29	0B	00	00	00	2C
00000020h:	07	00	00	00	2D	01	00	00	00	30	03	00	00	00	31	03
00000030h:	00	00	00	35	03	00	00	00	39	03	00	00	00	41	04	00
00000040h:	00	00	42	01	00	00	00	47	07	00	00	00	49	16	00	00
00000050h:	00	4D	02	00	00	00	4E	04	00	00	00	4F	05	00	00	00
00000060h:	53	04	00	00	00	54	09	00	00	59	07	00	00	00	61	00
00000070h:	6A	00	00	00	62	05	00	00	63	0A	00	00	64	2D	00	00
00000080h:	00	00	00	65	97	00	00	66	1C	00	00	00	67	15	00	00
00000090h:	00	00	68	52	00	00	69	3E	00	00	00	6A	07	00	00	00
000000a0h:	00	6B	0E	00	00	6C	34	00	00	6D	1F	00	00	00	00	00
000000b0h:	6E	49	00	00	00	6F	54	00	00	70	11	00	00	00	72	00
000000c0h:	2C	00	00	00	73	2D	00	00	74	55	00	00	75	1E	00	00
000000d0h:	00	00	00	76	13	00	00	77	12	00	00	00	79	2E	00	00
000000e0h:	00	00	7A	0B	00	00	00	CB	80	5F	98	F4	BF	3D	3F	7A
000000f0h:	60	D7	17	64	44	3E	88	87	D1	10	F9	29	47	8B	E2	F9
00000100h:	65	A6	EB	6C	7A	5E	A6	39	71	A9	6B	6C	72	74	A7	E2
00000110h:	96	E2	EC	87	65	E2	96	F7	A6	E6	2F	05	06	F8	54	7A
00000120h:	A2	7C	ED	2E	78	BE	53	6F	F1	4B	7B	D3	73	17	B4	54
00000130h:	A5	1E	D1	4F	FA	DA	A6	2F	97	47	E6	25	5B	2E	69	2D
00000140h:	56	2E	E4	C2	87	65	03	7E	DD	BC	37	C0	BB	6D	5D	6E
00000150h:	3D	9E	40	A3	B2	E8	7E	22	8D	D6	2F	90	96	F3	AD	BE
00000160h:	43	E9	7B	28	AB	57	53	DB	29	47	B3	C8	14	76	5D	0F
00000170h:	C4	51	BA	C5	F2	92	97	53	D4	3C	0F	4B	F5	0F	B0	4F
00000180h:	54	F9	8E	CF	20	51	D9	74	3F	11	46	EB	17	C5	DC	F3
00000190h:	81	2E	A6	3C	A8	A1	B7	86	F3	A7	63	97	68	04	6B	6C
000001a0h:	7B	48	E9	BF	98	C5	F2	D6	B5	A2	A5	8B	B9	33	3E	FA
000001b0h:	CF	61	FF	53	F7	3B	D3	B1	A2	87	AE	FA	CF	7C	93	A5
000001c0h:	E4	04	22	43	C2	A3	8B	E5	53	BC	54	B1	72	C6	3C	D6
000001d0h:	6F	36	9A	42	67	81	E9	7A	98	8F	26	B6	BE	76	A6	7F
000001e0h:	31	8B	B9	F2	6D	82	F2	6B	7B	AF	05	A8	FD	6F	A9	D2
000001f0h:	F9	51	43	6F	0D	E7	4E	C6	2E	C8	88	7D	11	0F	A2	21

输出到的编码文件中开头段为文件的编码方式信息。第一个字节（上图红色部分）为压缩前文件中出现的字符种类个数（假设为n）；后 n\*5 个字节为每个字符的 ASCII 码（1 字节）以及其出现的次数（4 字节，低位在前、高位在后）。如第 1 到 5 字节中，第 1 字节表示 ASCII 码为 10 的字符（换行符），第 2-5 字节(0x3C0000000)表示其出现了 60 次；之后的字节均为哈夫曼树编码文件信息。根据该存储方式，解压文件最少能够存储 2GB 文件信息，最多能够存储 256GB 的文件信息。

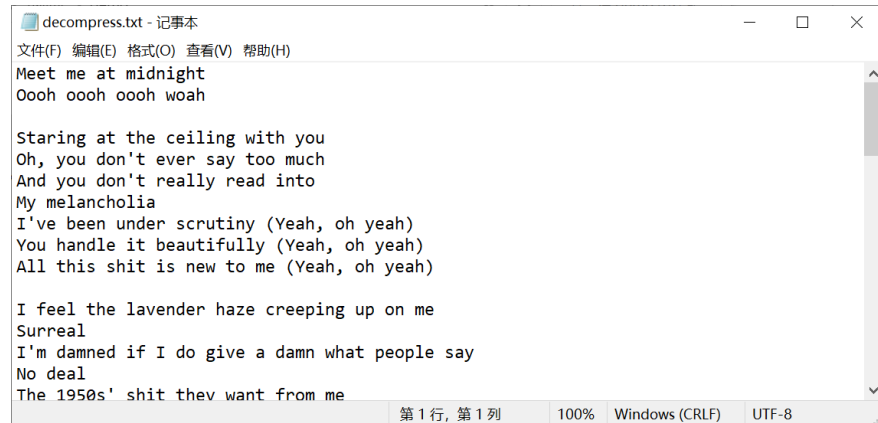
decompress 模式中，读入哈夫曼编码后的二进制文件；并通过文件头读入到的字符频数信息建立对应的哈夫曼树用于文件的解压。命令行输入格式为“可执行文件名 --decompress 待解压文件名 解压后文件名”。若解压成功，给出解压成功信息如下：

```

命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --decompress compress.txt decompress.txt
文件compress.txt已解压成功!

C:\Users\BoyuanZheng\demo>

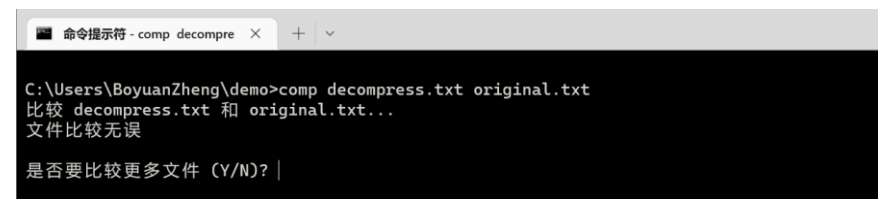
```



```
decompress.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
Meet me at midnight
Oooh oooh oooh woah

Staring at the ceiling with you
Oh, you don't ever say too much
And you don't really read into
My melancholia
I've been under scrutiny (Yeah, oh yeah)
You handle it beautifully (Yeah, oh yeah)
All this shit is new to me (Yeah, oh yeah)


I feel the lavender haze creeping up on me
Surreal
I'm damned if I do give a damn what people say
No deal
The 1950s' shit they want from me
第 1 行, 第 1 列 100% Windows (CRLF) UTF-8
```



```
命令提示符 - comp decompre
C:\Users\BoyuanZheng\demo>comp decompress.txt original.txt
比较 decompress.txt 和 original.txt...
文件比较无误
是否要比较更多文件 (Y/N)? |
```

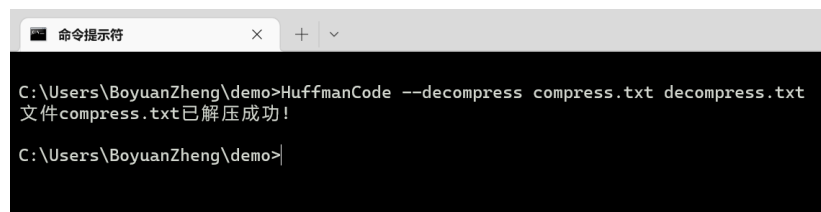
解压后的文件 decompress.txt 与原文件 original.txt 相同，经 comp 比较无误，压缩、解压均正确。

下面测试大文件的压缩与解压缩功能。输入的文件 ser.log 大小为 10073 KB，使用 compress 功能压缩：



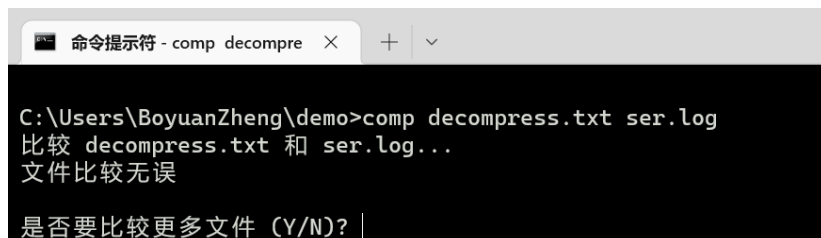
```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --compress ser.log compress.txt
文件ser.log已经压缩成功，压缩率为65.3826%
C:\Users\BoyuanZheng\demo>
```

压缩后文件大小为 6587 KB，压缩率为 65.38%。使用 decompress 功能解压缩如下：



```
命令提示符
C:\Users\BoyuanZheng\demo>HuffmanCode --decompress compress.txt decompress.txt
文件compress.txt已解压成功!
C:\Users\BoyuanZheng\demo>
```

解压后文件经 comp 与文件比较无误：



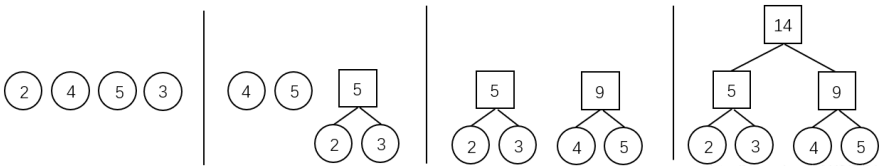
```
命令提示符 - comp decompre
C:\Users\BoyuanZheng\demo>comp decompress.txt ser.log
比较 decompress.txt 和 ser.log...
文件比较无误
是否要比较更多文件 (Y/N)? |
```

<p>调试分析</p>	<p>1. 编写程序时,HuffmanCoding 函数参照课本 P147 算法 6.12。根据课本算法,申请到的存放哈夫曼树型结构的数组 0 号单元未用,从下标为 1 开始使用。但课本的算法中,遍历 HT 数组给各个结点赋初值时,p 指针却从 HT(下标为 0)开始递增。这会导致之后的结点信息错位,且第 128 位置的字符信息未被初始化。在编写程序时,我没有注意到课本的笔误,导致了长时间的调试。将 p 指针从 HT+1(1 号单元)开始遍历即可;</p> <p>2. 读入文件、写入文件时,因为是一个字节一个字节的读取,而非文本方式读取,应该采用 ios::binary 二进制方式的读入。若用文本方式进行读写,很可能会意外读到终止信息,导致读入文件的提前结束,压缩文件中信息不全;</p> <p>3. 根据题目要求,本次程序只处理字符 ASCII 码在 0-127 之间的文件。尽管如此,进行错误输入处理时应该考虑到 128-255 的字符读入。起初我忽略了这一部分的字符,便会导致含有中文等其他字符的文件在数组中寻找编码时直接出错(GetIndex 函数找不到返回-1,带入数组下标时就造成了越界读)。调试过程中,对返回下标为-1 进行特别判断,在程序界面进行错误处理提示;</p> <p>4. 起初在压缩时,我希望在文件头部分每个字符能够用一字节存储 ASCII 码、并用一个字节存储出现次数(能够使文件头尽可能短,尽量小地影响到较小文件的压缩率)。但对于较大的文件,一个字节显然不足以存储某个字符出现的次数。因此在压缩大文件时,若某字符出现次数超过 255(即一字节存储的上限),写入文件头时的截断便会导致解压时错误的建树,使得解压出的文件完全错乱。由于对哈夫曼树的建树方式了解不够透彻,我起初采用用大小排序代替实际出现次数的方式,这样能使每个字符的权重都一定不超过 128。这样的方式能够正确地建树、压缩、解压,但会使得压缩后的文件压缩率不理想。这是因为排序代替次数虽然保留了权重之间的相对关系,但在建树过程中绝对的大小关系也会影响到树</p>
-------------	--

	<p>的建立。用排序代替次数会使得每个字符之间的权重十分平均，让频繁出现的字符难以分得尽可能短的编码，这就失去了哈夫曼树编码的很大一部分意义。因此，最后选择用四个字节来存储出现次数，即最少也可记录大小为 2GB 左右的文件信息，足够使用；</p> <p>5. 由于不能保证压缩文件用哈夫曼编码后的二进制数位数正好为 8 位，所以文件中的最后一个字节往往有许多位是补齐位数用的 0。若正好有某字符的编码是 0、00...就会导致文件末尾有若干个错误译码的字符。由于在文件头已经压入了每个字符对应的出现频数，相加便可得到文件的字节数。因此可以用字节数判断是否已经输出了所有编码的字符。若已经输出到了文件的字节个数，就直接停止输出，结束程序。</p>
心得体会	<p>本次实验中，主要实践了最优二叉树（哈夫曼树）的建立以及哈夫曼编码、解码。在传送电文、压缩文件时，为了能够构建总长度尽可能短的编码，可以对每个字符设计长度不等的编码。但不等长的编码若不做特殊设计，便会存在同一编码序列的多种译法；因此，必须任意一个字符的编码都不是另一个字符的编码的前缀，即前缀编码。可以利用二叉树来设计二进制前缀编码，即构建哈夫曼树。用二叉树的叶子结点表示某字符，从根节点到该叶子结点的路径来表示其编码（左子树为‘0’，右子树为‘1’），如下图所示：</p> <div data-bbox="660 1462 1131 1928"></div>

为了使编码的总长度，即压缩后文件的大小尽可能小，可以根据每个字符出现的频数为权重来建立哈夫曼树。若每种字符在文件中出现的频数为  $w_i$ ，编码长度为  $l_i$ ，则压缩文件的总长度为  $\sum w_i l_i$ 。将每个字符出现频数作为叶子结点的权重，则压缩文件的总长度即为哈夫曼树的带权路径长度 WPL。哈夫曼树，即最优二叉树的构建就是要使得 WPL 最小，从而让压缩后编码尽可能短。假设文件中有  $n$  种字符，则哈夫曼树有  $n$  个叶子结点；由于哈夫曼树是正则二叉树，没有度为 1 的结点，因此可以用大小为  $2n-1$  的数组来存储。

哈夫曼算法用于构造一棵哈夫曼树，算法大致如下：首先进行初始化，即由给定的  $n$  个权值构造  $n$  棵只有一个根结点的二叉树，得到一个二叉树集合；其次进行选取与合并，即从上述集合中选取根节点权值最小的两棵二叉树分别作为左右子树构造一棵新的二叉树，这棵新二叉树的根节点的权值为其左、右子树根结点的权值和；选取合并后，从集合中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到集合中；重复操作直至集合中只剩下一颗二叉树。下图即为哈夫曼树构建的图示：



而在解码时，由于前缀编码任意一个字符的编码都不是另一个字符的编码的前缀这一特性，便可以在哈夫曼树中从根节点开始根据 ‘0’ 或 ‘1’ 来访问到对应的子结点解码出字符信息。

在建立哈夫曼树时，每次查找所有根节点中权值最小的两个根节点，本次的 Select 函数采用的是直接遍历查找最小、次小权值结点的方式，时间复杂度为  $O(n)$ 。可以采用优先队列或小根堆的方式进行优化，但字符总数上限为 128，优化效果不明显。使用 decompress 模式解码时，若原文件字节数为  $n$ ，字符个数为  $m$ ，则时间复杂度为  $O(n \log m)$ ，由于  $m$  上限为 128，可以近似理解为  $O(n)$ 。

## 附.完整代码

### 1.哈夫曼树头文件 (HuffmanTree.h)

```
#pragma once
#include "../queue.hpp"

#define CHAR_NUM_TOT 256

/* 哈夫曼树的结点 */
typedef struct HTNode {
    unsigned int weight; //结点的权重
    unsigned char value; //结点的字符值
    unsigned int parent, lchild, rchild; //结点的双亲、左孩子、右孩子
}*HTPointer;

/* 字符-出现频数 */
struct CharWithNode {
    unsigned char ch; //字符
    unsigned int weight; //权重-出现的次数
};

/* 哈夫曼编码表 */
typedef char** HuffmanCode;

/* 哈夫曼树的结构体 */
class HuffmanTree {
private:
    HTPointer tree; //哈夫曼树数组
    CharWithNode* char_list; //字符出现频率的列表
    HuffmanCode code_list; //哈夫曼编码的列表
    unsigned int n; //字符种类数
    //统计字符出现频率
    void CalCharFreq(const char* filename, const char* filename1, bool
output = true);
    void CalCharFreq(const char* str);
    //读入字符出现频率
    void ReadCharFreq(const char* filename);
    //选择根节点中权值最小的
    void Select(int max, int& s1, int& s2);
    //建立哈夫曼树 并计算对应每个字符的01串
    void HuffmanCoding();
};
```

```

//获取某字符在list的下标
int GetIndex(unsigned char ch);
//编码功能
void Inner_Encode(const char* infilename, const char* outfilename);
//解码功能
void Inner_Decode(const char* infilename, const char* outfilename);
//压缩功能
void Inner_Compress(const char* infilename, const char* outfilename);
//解压功能
void Inner-Decompress(const char* infilename, const char*
outfilename);
//打印哈夫曼树
void PrintTree();
public:
//哈夫曼树的构建
HuffmanTree();
//哈夫曼树的销毁
~HuffmanTree();
//演示模式
void Display(const char* str);
//编码模式
void Encode(const char* infilename, const char* outfilelist, const
char* outfilename);
//解码模式
void Decode(const char* infilename, const char* infilelist, const char*
outfilename);
//压缩模式
void Compress(const char* infilename, const char* outfilename);
//解压模式
void Decompress(const char* infilename, const char* outfilename);
};

```

## 2.哈夫曼树源文件 (HuffmanTree.cpp)

```

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string>
#include <string.h>
#include <fstream>
#include "../HuffmanTree.h"
using namespace std;

```



```
/**
 * @brief 打印一定数量的空格(iomanip)
 * @param n 打印的空格数
 */
static void PrintSpace(int n)
{
    for (int i = 0; i < n; i++)
        cout << " ";
}
```

```
/**
 * @brief 求2的n次幂
 * @param n 求幂的数n
 * @return 2的n次幂结果
 */
static int pow2(int n)
{
    int ans = 1;
    for (int i = 0; i < n; i++)
        ans *= 2;
    return ans;
}
```

```
/**
 * @brief 构建哈夫曼树
 */
HuffmanTree::HuffmanTree()
{
    tree = NULL;
    char_list = NULL;
    code_list = NULL;
}
```

```
/**
 * @brief 销毁哈夫曼树
 */
HuffmanTree::~HuffmanTree()
{
    if(char_list)
        delete[] char_list;
    if(code_list)
        delete[] code_list;
    if (tree)
```

```

        delete[] tree;
    }

/**
 * @brief 计算文件中每个字符出现的频数
 * @param filename 输入的压缩文件
 * @param filename1 输出的字符频数文件
 * @param output 是否要输出到文件
 */
void HuffmanTree::CalCharFreq(const char* filename, const char* filename1, bool
output)
{
    int* templist = new(nothrow) int[CHAR_NUM_TOT];
    int order[CHAR_NUM_TOT];
    for (int i = 0; i < CHAR_NUM_TOT; i++)
        order[i] = i;

    if (!templist)
        exit(OVER_FLOW);
    memset(templist, 0, CHAR_NUM_TOT * sizeof(int));
    ifstream infile;
    ofstream outfile;
    infile.open(filename, ios::in | ios::binary);    //打开文件
    if (!infile.is_open()) {
        cout << "文件" << filename << "打开失败!" << endl;
        exit(0);
    }
    if (output) {
        outfile.open(filename1, ios::out); //打开文件
        if (!outfile.is_open()) {
            cout << "文件" << filename1 << "写入失败!" << endl;
            exit(0);
        }
    }

    unsigned char ch;
    while (infile.peek() != EOF) {
        ch = infile.get();
        templist[ch] += 1;
    }

    unsigned int i, j;

    n = 0;

```

```

    for (int i = 0; i < CHAR_NUM_TOT; i++)
        if (templist[i])
            ++n;

    char_list = new(nothrow) CharWithNode[n];
    if (!char_list)
        exit(OVER_FLOW);
    j = 0;
    for (i = 0; i < n; i++) {
        while (templist[j] == 0)
            j++;
        char_list[i].weight = templist[j];
        char_list[i].ch = (unsigned char)j;
        j++;
    }

    //演示时输出每个字频数的文件
    if (output) {
        for (unsigned int i = 0; i < n; i++)
            outfile << char_list[i].ch + 0 << " " << char_list[i].weight + 0 <<
endl;
    }

    delete[] templist;
    infile.close();
    if (output)
        outfile.close();
}

/**
 * @brief 计算字符串中每个字符出现的频数(函数重载)
 * @param str 输入的字符串
 * @param output 是否要输出到文件
 */
void HuffmanTree::CalCharFreq(const char* str)
{
    int* templist = new(nothrow) int[CHAR_NUM_TOT];
    int order[CHAR_NUM_TOT];
    for (int i = 0; i < CHAR_NUM_TOT; i++)
        order[i] = i;

    if (!templist)
        exit(OVER_FLOW);
    memset(templist, 0, CHAR_NUM_TOT * sizeof(int));

```

```

    for (unsigned int i = 0; i < strlen(str); i++) {
        templist[str[i]] += 1;
    }

    unsigned int i, j;

    n = 0;
    for (int i = 0; i < CHAR_NUM_TOT; i++)
        if (templist[i])
            ++n;

    char_list = new(nothrow) CharWithNode[n];
    if (!char_list)
        exit(OVER_FLOW);
    j = 0;
    for (i = 0; i < n; i++) {
        while (templist[j] == 0)
            j++;
        char_list[i].weight = templist[j];
        char_list[i].ch = (unsigned char)j;
        j++;
    }

    for (unsigned int i = 0; i < n; i++)
        cout << char_list[i].ch << "的出现频次为 : " << char_list[i].weight + 0
<< endl;

    delete[] templist;
}

/**
 * @brief 读入字符频数文件信息写入list
 * @param filename 读入的字符频数文件
 */
void HuffmanTree::ReadCharFreq(const char* filename)
{
    int* templist = new(nothrow) int[CHAR_NUM_TOT];
    if (!templist)
        exit(OVER_FLOW);
    memset(templist, 0, sizeof(int) * CHAR_NUM_TOT);

    ifstream infile;
    ofstream outfile;

```

```

infile.open(filename, ios::in);           //打开文件
n = 0;  //置零开始计数
while (infile.good()) {
    int order, num;
    infile >> order >> num;
    if (!infile.good())
        break;
    n++;
    templist[order] = num;
}

char_list = new(nothrow) CharWithNode[n];
if (!char_list)
    exit(OVER_FLOW);
int j = 0;
for (unsigned int i = 0; i < n; i++) {
    while (templist[j] == 0)
        j++;
    char_list[i].weight = templist[j];
    char_list[i].ch = (char)j;
    j++;
}

infile.close();
}

/**
 * @brief 选择根结点中权值最小的两个结点
 * @param max 从1-max的范围中查找
 * @param s1 权值最小的结点
 * @param s2 权值次小的结点
 */
void HuffmanTree::Select(int max, int& s1, int& s2)
{
    s1 = -1, s2 = -1;
    for (int i = 1; i <= max; i++) {
        //已经有父结点 跳过
        if (tree[i].parent)
            continue;
        if (s1 == -1 || tree[i].weight < tree[s1].weight) {
            s2 = s1;
            s1 = i;
        }
        else if (s2 == -1 || tree[i].weight < tree[s2].weight)
    }
}

```

```

        s2 = i;
    }
}

/**
 * @brief 建立哈夫曼树 并计算对应每个字符的01串
 */
void HuffmanTree::HuffmanCoding()
{
    unsigned int m = 2 * n - 1;
    HTPointer p;
    unsigned int i;
    tree = new(nothrow) HTNode[2 * n];
    if (!tree)
        exit(OVER_FLOW);
    CharWithNode* q = char_list;
    for (p = tree + 1, i = 1; i <= n; ++i, ++p, ++q) {
        p->weight = q->weight;
        p->value = q->ch;
        p->lchild = p->rchild = 0;
        p->parent = 0;
    }
    for (; i <= m; ++i, ++p) {
        p->weight = 0;
        p->value = 0;
        p->lchild = p->rchild = 0;
        p->parent = 0;
    }
    //下面开始建树
    for (unsigned int i = n + 1; i <= m; ++i) {
        int s1, s2;
        Select(i - 1, s1, s2);
        tree[s1].parent = i;
        tree[s2].parent = i;
        tree[i].lchild = s1;
        tree[i].rchild = s2;
        tree[i].value = '^';
        tree[i].weight = tree[s1].weight + tree[s2].weight;
    }

    code_list = new(nothrow) char* [n + 1];
    if (!code_list)
        exit(OVER_FLOW);
    char* cd = new(nothrow) char[n];

```

```

    if (!cd)
        exit(OVER_FLOW);
    cd[n - 1] = '\0';

    for (unsigned int i = 1; i <= n; ++i) {
        int start = n - 1;
        int f, c;
        for (c = i, f = tree[i].parent; f; c = f, f = tree[f].parent) {
            if (tree[f].lchild == c)
                cd[--start] = '0';
            else
                cd[--start] = '1';
        }
        code_list[i] = new(nothrow) char[n - start];
        if (!code_list[i])
            exit(OVER_FLOW);
        strcpy(code_list[i], &cd[start]);
    }

    delete[] cd;
}

/**
 * @brief 获取某字符在list的下标
 * @param ch 要查找的字符
 */
int HuffmanTree::GetIndex(unsigned char ch)
{
    for (unsigned int i = 1; i <= n; i++)
        if (tree[i].value == ch)
            return i;
    return -1;
}

/**
 * @brief 编码功能
 * @param infilename 待编码的文件名
 * @param outfilename 编码后的文件名
 */
void HuffmanTree::Inner_Encode(const char* infilename, const char*
outfilename)
{
    ifstream infile;
    ofstream outfile;

```

```

infile.open(infilename, ios::in);    //打开文件
if (!infile.is_open()) {
    cout << "文件" << infilename << "打开失败!" << endl;
    exit(0);
}
outfile.open(outfilename, ios::out); //打开文件
if (!outfile.is_open()) {
    cout << "文件" << outfilename << "写入失败!" << endl;
    exit(0);
}

char ch;
while (infile.peek() != EOF) {
    ch = infile.get();
    int ret = GetIndex(ch);
    if (ret == -1) {
        cout << "文件内包含ASCII码不在0-127内的字符!" << endl;
        exit(0);
    }
    outfile << code_list[ret];
}

cout << "文件" << infilename << "已成功编码，请打开" << outfilename << "查看具体信息。" << endl;

infile.close();
outfile.close();
}

/**
 * @brief 解码功能
 * @param infilename 待解码的文件名
 * @param outfilename 解码后的文件名
 */
void HuffmanTree::Inner_Decode(const char* infilename, const char*
outfilename)
{
    ifstream infile;
    ofstream outfile;
    infile.open(infilename, ios::in);    //打开输入文件
    if (!infile.is_open()) {
        cout << "文件" << infilename << "打开失败!" << endl;
        exit(0);
    }
}

```



```

outfile.open(outfilename, ios::out); //打开输出文件
if (!outfile.is_open()) {
    cout << "文件" << outfilename << "写入失败!" << endl;
    exit(0);
}

unsigned char ch;
HTPonter now = &tree[2 * n - 1];
while (infile.peek() != EOF) {
    ch = infile.get();
    if (ch == '0' && now->lchild) {
        now = &tree[now->lchild];
    }
    else if (ch == '1' && now->rchild) {
        now = &tree[now->rchild];
    }
    else if (ch != '1' && ch != '0') {
        cout << "文件编码格式有误!" << endl;
        exit(0);
    }
    else {
        outfile << now->value;
        now = &tree[2 * n - 1];
        if (ch == '0' && now->lchild) {
            now = &tree[now->lchild];
        }
        else if (ch == '1' && now->rchild) {
            now = &tree[now->rchild];
        }
    }
}

/* 补上最后一个字符 */
outfile << now->value;
cout << "文件" << infilename << "已经成功解码为文件" << outfilename << endl;

infile.close();
outfile.close();
}

/**
 * @brief 压缩功能
 * @param infilename 待压缩的文件名
 * @param outfilename 待解压的文件名

```

```

*/
void HuffmanTree::Inner_Compress(const char* infilename, const char*
outfilename)
{
    ifstream infile;
    ofstream outfile;

    infile.open(infilename, ios::in | ios::binary);           //打开
输入文件
    if (!infile.is_open()) {
        cout << "文件" << infilename << "打开失败!" << endl;
        exit(0);
    }

    outfile.open(outfilename, ios::out | ios::binary);        //打开输出文件
    if (!outfile.is_open()) {
        cout << "文件" << outfilename << "写入失败!" << endl;
        exit(0);
    }

    outfile << (unsigned char)n;                               //把字符个数写入
    for (unsigned int i = 1; i <= n; i++) {
        outfile << tree[i].value;                             //二进制写入, 节省空间
        for (int j = 0; j < 4; j++) { //unsigned int分4次写入
            outfile << char(tree[i].weight >> (j * 8));
        }
    }

    //等到8个字节完整后输出一次
    unsigned char ch, outch = 0;
    int i = 1;
    int org_tot = 0, aft_tot = 0;                             //统计字符个数
    while (infile.peek() != EOF) {
        ch = infile.get();
        org_tot++;
        int ret = GetIndex(ch);
        if (ret == -1) {
            cout << "文件中存在非0-127内字符!" << endl;
            exit(0);
        }
        for (unsigned int j = 0; j < strlen(code_list[ret]); j++) {
            if (code_list[ret][j] == '1')
                outch |= (1 << (8 - i));
            i++;
        }
    }
}

```

```

        if (i == 9) {
            aft_tot++;
            outfile << outch;
            outch = 0;
            i = 1;
        }
    }
}

if (i != 1) {
    aft_tot++;
    outfile << outch;
}

cout << "文件" << infilename << "已经压缩成功, 压缩率为" << aft_tot * 100.0
/ org_tot << "%" << endl;

infile.close();
outfile.close();
}

/**
 * @brief 解压功能
 * @param infilename 待解压的文件名
 * @param outfilename 解压后的文件名
 */
void HuffmanTree::Inner-Decompress(const char* infilename, const char*
outfilename)
{
    ifstream infile;
    ofstream outfile;
    infile.open(infilename, ios::in | ios::binary);    //打开输入文件
    if (!infile.is_open()) {
        cout << "文件" << infilename << "打开失败!" << endl;
        exit(0);
    }
    outfile.open(outfilename, ios::out | ios::binary);    //打开输出文件
    if (!outfile.is_open()) {
        cout << "文件" << outfilename << "写入失败!" << endl;
        exit(0);
    }

    unsigned char ch;
    //读取n的信息

```

```

ch = infile.get();
//若所有字符都占 则会溢出为0
n = ch ? ch : 256;
//从文件头读取字符频数信息
char_list = new(nothrow) CharWithNode[n];
if (!char_list)
    exit(OVER_FLOW);

int tot_ch = 0, cur_ch = 0;           //统计文档总字符数量

for (unsigned int i = 0; i < n; i++) {
    ch = infile.get();
    char_list[i].ch = ch;
    char_list[i].weight = 0;
    for (int j = 0; j < 4; j++) {
        ch = infile.get();
        char_list[i].weight |= (ch << (j * 8));
    }
    tot_ch += char_list[i].weight;
}

HuffmanCoding();

unsigned char cur;
HTPonter now = &tree[2 * n - 1];           //这个位置是哈夫曼树的根
节点
while (infile.peek() != EOF) {
    cur = infile.get();
    //在哈夫曼树中找对应的编码
    for (int i = 1; i <= 8; i++) {
        char bit = !(cur & (1 << (8 - i)));
        if (bit == 0 && now->lchild) {
            now = &tree[now->lchild];
        }
        else if (bit == 1 && now->rchild) {
            now = &tree[now->rchild];
        }
        else {
            cur_ch++;           //记录字符数
            if (cur_ch > tot_ch) //文件长度到了 防止末尾0被误判
                break;
            outfile << now->value;
            now = &tree[2 * n - 1];
            if (bit == 0 && now->lchild) {

```

```

        now = &tree[now->lchild];
    }
    else if (bit == 1 && now->rchild) {
        now = &tree[now->rchild];
    }
    }
}

//最后一个字节正好结束 没来得及输出
if (cur_ch < tot_ch)
    outfile << now->value;

cout << "文件" << infilename << "已解压成功!" << endl;

infile.close();
outfile.close();
}

/**
 * @brief 打印哈夫曼树
 */
void HuffmanTree::PrintTree()
{
    int depth = 0;
    {
        struct NodeWithDepth {
            HTPointer node;
            int depth;
            NodeWithDepth(HTPointer bb = NULL, int dd = 1)
            {
                node = bb;
                depth = dd;
            }
        };
        SqQueue<NodeWithDepth> queue(1000); //queue的最大长度

        NodeWithDepth p;
        queue.Enqueue(NodeWithDepth(&tree[2 * n - 1], 1));
        while (!queue.QueueEmpty()) {
            queue.DeQueue(p);
            depth = max(depth, p.depth);
        }
    }
}

```

```

        if (p.node->lchild)
            queue.Enqueue(NodeWithDepth(&tree[p.node->lchild], p.depth +
1));
        if (p.node->rchild)
            queue.Enqueue(NodeWithDepth(&tree[p.node->rchild], p.depth +
1));
    }
}
/* 以上求算队列深度 */

/* 以下开始做输出 */
{
    struct NodeWithNo {
        HTPointer node;
        int depth;
        int no;
        NodeWithNo(HTPointer bb = NULL, int dd = 1, int nn = 1)
        {
            node = bb;
            depth = dd;
            no = nn;
        }
    };
    SqQueue<NodeWithNo> queue(1000), queue2(1000); //queue的最大长度

    queue.ClearQueue();
    queue.Enqueue(NodeWithNo(&tree[2 * n - 1], 1, 1));
    NodeWithNo p, q, r, s;
    //补第一行的前置空格
    PrintSpace(pow2(depth - 1) - 1);
    while (!queue.QueueEmpty()) {
        queue.DeQueue(p);
        if (p.node->lchild)
            queue.Enqueue(NodeWithNo(&tree[p.node->lchild], p.depth + 1, 2
* p.no));
        if (p.node->rchild)
            queue.Enqueue(NodeWithNo(&tree[p.node->rchild], p.depth + 1, 2
* p.no + 1));
        //输出当前节点
        cout << p.node->value;
        queue2.Enqueue(p); //塞入另一个队列 是为了下一行输出 / \
        if (queue.QueueEmpty()) //最后一个节点输出结束了 换行
            cout << endl;
        else {

```

```

        queue.GetHead(q);
        if (q.depth == p.depth) { //在相同层次
            PrintSpace((q.no - p.no) * pow2((depth - p.depth) + 1) - 1);
        }
        else { //在不同层次
            cout << endl;
            //先处理行首空格
            queue2.GetHead(r);
            PrintSpace(pow2(depth - p.depth) - 2 + (r.no - pow2(p.depth
- 1)) * pow2((depth - p.depth) + 1));
            while (!queue2.QueueEmpty()) {
                queue2.DeQueue(r);
                cout << (r.node->lchild ? '/' : ' ') << ' ' <<
(r.node->rchild ? '\\' : ' ');
                if (!queue2.QueueEmpty()) { //不是行末
                    queue2.GetHead(s);
                    PrintSpace((s.no - r.no) * pow2((depth - s.depth)
+ 1) - 3);
                }
            }
            cout << endl;
            PrintSpace(pow2(depth - q.depth) - 1 + (q.no -
pow2(p.depth)) * pow2((depth - q.depth) + 1));
        }
    }
}

/**
 * @brief 展示模式
 * @param str 输入的字符串
 */
void HuffmanTree::Display(const char* str)
{
    CalCharFreq(str);
    HuffmanCoding();
    PrintTree();
}

/**
 * @brief 编码模式
 * @param infilename 待编码文件
 * @param outfilelist 字符频数文件

```

```

    * @param outfilename 编码后文件
*/
void HuffmanTree::Encode(const char* infilename, const char* outfilelist, const
char* outfilename)
{
    CalCharFreq(infilename, outfilelist);
    HuffmanCoding();
    Inner_Encode(infilename, outfilename);
}

/**
 * @brief 解码模式
 * @param infilename 待解码文件
 * @param infilelist 字符频数文件
 * @param outfilename 解码后文件
*/
void HuffmanTree::Decode(const char* infilename, const char* infilelist, const
char* outfilename)
{
    ReadCharFreq(infilelist);
    HuffmanCoding();
    Inner_Decode(infilename, outfilename);
}

/**
 * @brief 压缩模式
 * @param infilename 待压缩文件
 * @param outfilename 压缩后文件
*/
void HuffmanTree::Compress(const char* infilename, const char* outfilename)
{
    CalCharFreq(infilename, NULL, false); //该模式不另外写入文件中
    HuffmanCoding();
    Inner_Compress(infilename, outfilename);
}

/**
 * @brief 解压模式
 * @param infilename 待解压文件
 * @param outfilename 解压后文件
*/
void HuffmanTree::Decompress(const char* infilename, const char* outfilename)
{

```



```
Inner-Decompress(infile, outfile);  
//先从文件中读入 因此在函数内调用HuffmanCoding  
}
```

### 3. 队列类源文件 (queue.hpp)

```
#pragma once  
#include <iostream>  
using namespace std;  
  
#define OK 0  
#define QERROR -1  
#define QOVERFLOW -2  
typedef int Status;  
  
template <class QElemType>  
class SqQueue {  
private:  
    QElemType* base;  
    int MAXQSIZE;  
    int front;  
    int rear;  
public:  
    SqQueue(int maxqsize);    //建立空队列（构造函数）  
    ~SqQueue();              //销毁队列（析构函数）  
    int QueueLength();        //获取队列长度  
    Status EnQueue(QElemType e);    //新元素入队  
    Status DeQueue(QElemType& e);    //队首元素出队  
    Status GetHead(QElemType& e);    //获取队首元素  
    Status ClearQueue();    //清空队列  
    bool QueueEmpty();    //判断队列是否为空  
};  
  
template <class QElemType>  
SqQueue <QElemType>::SqQueue(int maxqsize)  
{  
    MAXQSIZE = maxqsize + 1;    //空一个位置以区分满和空  
    base = new(nothrow) QElemType[MAXQSIZE];  
    if (!base)  
        exit(QOVERFLOW);  
}
```

```

        front = rear = 0;
    }

template <class QElemType>
SqQueue <QElemType>::~SqQueue()
{
    if (base)
        delete base;
    front = rear = 0;
}

template <class QElemType>
Status SqQueue <QElemType>::ClearQueue()
{
    front = rear = 0;
    return OK;
}

template <class QElemType>
int SqQueue <QElemType> ::QueueLength()
{
    return (rear - front + MAXQSIZE) % MAXQSIZE;
}

template <class QElemType>
Status SqQueue <QElemType> ::EnQueue(QElemType e)
{
    if ((rear + 1) % MAXQSIZE == front)
        return QERROR;
    base[rear] = e;
    rear = (rear + 1) % MAXQSIZE;
    return OK;
}

template <class QElemType>
Status SqQueue <QElemType> ::DeQueue(QElemType& e)
{
    if (front == rear)
        return QERROR;
    e = base[front];
    front = (front + 1) % MAXQSIZE;
    return OK;
}

```

```

template <class QElemType>
Status SqQueue <QElemType> ::GetHead(QElemType& e)
{
    if (front == rear)
        return QERROR;
    e = base[front];
    return OK;
}

template <class QElemType>
bool SqQueue <QElemType>::QueueEmpty()
{
    return front == rear;
}

```

#### 4. 主演示文件 (main.cpp)

```

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <algorithm>
#include <string.h>
#include <fstream>
#include <iomanip>
#include "../HuffmanTree.h"

using namespace std;

/* void usage(const char* const procname, const int args_num)
 * @brief 打印函数提示信息
 * @param procname 可执行文件名称
 * @param args_num 参数个数
 */
void usage(const char* const procname, const int args_num)
{
    const int wkey = 7 + strlen(procname) + 1;
    const int wopt = 7 + strlen(procname) + 4;

    cout << endl;
    cout << "Usage: " << procname << " { --display }" << endl;
    cout << setw(wkey) << ' ' << "{ --encode filename1 filename2 filename3 }"
    << endl;
    cout << setw(wkey) << ' ' << "{ --decode filename1 filename2 filename3 }"

```

```

<< endl;
    cout << setw(wkey) << ' ' << "{ --compress filename1 filename2 }" <<
endl;
    cout << setw(wkey) << ' ' << "{ --decompress filename1 filename2 }" <<
endl;
    cout << endl;

    cout << setw(wkey) << ' ' << "必选项：指定程序功能(五选一)" << endl;
    cout << setw(wopt) << ' ' << "--display：展示哈夫曼树型结构" << endl;
    cout << setw(wopt) << ' ' << "--encode：(演示用)将文件进行哈夫曼编码 [待
编码文件] [输出的字符频数文件] [编码后文件]" << endl;
    cout << setw(wopt) << ' ' << "--decode：(演示用)将哈夫曼编码文件解码 [待
解码文件] [输入的字符频数文件] [解码后文件]" << endl;
    cout << setw(wopt) << ' ' << "--compress：将文件用哈夫曼编码压缩 [待压
缩文件] [压缩后文件]" << endl;
    cout << setw(wopt) << ' ' << "--decompress：将哈夫曼编码文件解压 [待解
压文件] [解压后文件]" << endl;
    cout << endl;
}

int main(int argc, char** argv)
{
    HuffmanTree tree;

    if (argc == 1)
        usage(argv[0], argc);

    /* display模式 */
    else if (strcmp(argv[1], "--display") == 0 && argc == 2) {
        char input[128];
        cout << "\n请输入演示用字符串：";
        cin >> input;
        tree.Display(input);
    }

    /* encode模式 */
    else if (strcmp(argv[1], "--encode") == 0 && argc == 5)
        tree.Encode(argv[2], argv[3], argv[4]);

    /* decode模式 */
    else if (strcmp(argv[1], "--decode") == 0 && argc == 5)
        tree.Decode(argv[2], argv[3], argv[4]);
}

```

```
/* compress模式 */
else if (strcmp(argv[1], "--compress") == 0 && argc == 4)
    tree.Compress(argv[2], argv[3]);

/* decompress模式 */
else if (strcmp(argv[1], "--decompress") == 0 && argc == 4)
    tree.Decompress(argv[2], argv[3]);

else
    usage(argv[0], argc);

return 0;
}
```