实习三：哈夫曼编码

2020年6月29日 星期一

姓名： 李泽林 系年级： 2016海洋生命学院

学号： 16040031011 专业班级： 生物科学1班

1. **需求分析**

1、需求

求出两个文本文档（不同长度的英文文章）中所有ASCII字符的频度，非ASCII字符统一记为ASCII字符的“NUL”（”\0”，ASCII值为0）字符，据此频度生成哈夫曼编码。然后分别以该哈夫曼编码和等长编码（ASCII码），对两个文章进行编码和译码，记录这些编码和译码的运算时间；产生译码文件的大小（压缩效率）等信息。最后比较解码后的文件与原文件有无差异。即：

（1）建立哈夫曼树和哈夫曼编码。

（2）利用哈夫曼编码对文件进行编码和译码。

（3）比较哈夫曼编码与等长编码（例如ASCII码）的压缩效率和运算时间。

2、输入

输入需准备2个纯文本文档，分别是ShortStory.txt（1000-3000字的英文文章，本人实习所准备的该文章长度为2980字）和LongStory.txt（30000-50000字的英文文章，本人实习所准备的该文章长度为31960字），字数统计以一个单词为一字。

3、输出

3.1、输出到命令行内容

1. 字符总数、字符种数。
2. 哈夫曼树：包括结点位置、字符值（非叶子结点则无此值，显示为trunk）、权重、parent、左孩子、右孩子的位置（无则为0）。
3. 哈夫曼编码：包括字符值、权重、其对应的哈夫曼编码。
4. 译码结果与原文章比较结果，显示差异字符数。

3.2、输出文件

1. ShortHuf.txt、LongHuf.txt（根据统计出的频度求出的各字符对应哈夫曼编码）
2. ShortCode.txt、LongCode.txt（根据得到的哈夫曼编码对文章编码得到的结果）
3. ShortDecode.txt、LongDecode.txt（由哈夫曼编码结果译码得到的结果）
4. ShortASCCode.txt、LongASCCode.txt（根据ASCII编码对文章编码得到的结果）
5. ShortASCDecode.txt、LongASCDecode.txt（由ASCII编码结果译码得到的结果）
6. TimeCompare.txt（存储各文章/编码方式的编码和译码的运算时间）

3.3、输出形式

1. 在所有输出中（包括命令行和文件），表示字符时，同时显示了其ASCII值和字符本身，以“字符(该字符ASCII值)”的形式展现。
2. 输出的编码结果文件，为单行显示，所有0、1均无间隔。
3. 输出的哈夫曼编码对照字符文件（ShortHuf.txt、LongHuf.txt）格式为“字符(该字符ASCII值)\t哈夫曼编码”。
4. 输出的运算时间统计，包含8行，每一行格式为“输入的文章名+编码/译码方式（Huffman/binary-ASCII coding/decoding）：运算时间（单位毫秒）”。

4、程序功能

（1）统计文章中的字符总数、字符种数、各ASCII字符频度。

（2）建立并输出哈夫曼树和哈夫曼编码。

（3）利用哈夫曼编码和等长编码（ASCII码）对文件进行编码和译码。

（4）求出哈夫曼编码与等长编码对文件编码和译码的运算时间。

5、测试数据

见源文件中的ShortStory.txt和LongStory.txt。

1. **概要设计**

1、抽象数据类型

ADT story {

数据对象：D={ ai | ai∈TermSet, i=1,2,…,m, m≥0 }

（TermSet中的每个元素包含一个字符）

数据关系：R1={ <ai-1, ai> | ai-1, ai∈D, 且ai-1中的值<ai中的值, i=1,2,…,n }

基本操作：

read\_txt(char file\_name[], story &P)

初始条件：文件名file\_name变量存在。

操作结果：从名称为file\_name的文本文档逐个读取字符存储到线性表P。

print\_story(story P)

初始条件：线性表P存在。

操作结果：遍历显示线性表P的内容到命令行窗口。

count\_ascii(story P, int ascii[])

初始条件：线性表P、空的数组ascii[]存在。

操作结果：统计线性表P中各字符频度记录到数组ascii[]中。

code\_huff(story P, huffcode HC[], int n, char out\_file\_name[])

初始条件：线性表P、整数n、哈夫曼编码结构体组HC[]、文件名out\_file\_name存在。

操作结果：根据各字符哈夫曼编码对线性表P的元素编码生成编码存储到文本文档。

code\_ascii\_bin (story P, char out\_file\_name[])

初始条件：线性表P存在，文件名out\_file\_name变量存在

操作结果：遍历线性表P并将其译为二进制ASCII码并输出到文本文档。

char\_compare(story P, char file2\_name[])

初始条件：线性表P存在，文件名file2\_name变量存在。

操作结果：逐个比较线性表P与file2\_name文件中每一位的字符，输出差异字符数量。

} ADT story

ADT list {

数据对象：D={ ai | ai∈TermSet, i=1,2,…,m, m≥0 }

（TermSet中的每个元素包含一个表示ASCII值的整数和表示权重的整数）

数据关系：R1={ <ai-1, ai> | ai-1, ai∈D, 且ai-1中的值<ai中的值, i=1,2,…,n }

基本操作：

init\_list(list &L)

操作结果：构造一个空的带头结点线性表L。

bubble\_sort(list &L)

初始条件：线性表L存在。

操作结果：将线性表L按照权重从大到小对元素进行冒泡排序。

print\_list(list L)

初始条件：线性表L存在。

操作结果：遍历显示线性表所有元素（包括字符及其ASCII值、权重值）。

array\_to\_list(int ascii[], list &L)

初始条件：非空数组ascii[]存在。

操作结果：将数组内容转移到线性表L当中（此过程自动产生线性表L并对其排序）。

gen\_hufftree(list L, hufftree &HT)

初始条件：线性表L存在。

操作结果：由线性表L包含的频度信息生成其对应的哈夫曼树HT。

} ADT list

ADT hufftree {

数据对象：D={ ai | ai∈TermSet, i=1,2,…,m, m≥0 }

（TermSet中的每个元素包含一个表示ASCII值的整数和表示权重的整数）

数据关系：R1={ <ai-1, ai> | ai-1, ai∈D, 且ai-1中的值<ai中的值, i=1,2,…,n }

基本操作：

min\_two(hufftree HT, int n, int &s1, int &s2)

初始条件：哈夫曼树HT、整数n存在。

操作结果：从哈夫曼树中得到权值最小的两个结点。

gen\_hufftree(list L, hufftree &HT)

初始条件：线性表L存在。

操作结果：由线性表L生成哈夫曼树。

print\_hufftree(hufftree HT, int m)

初始条件：哈夫曼树HT、整数m（树的结点总数）存在。

操作结果：显示哈夫曼树的结构。

gen\_huffcode(int size, hufftree HT, huffcode &HC[])

初始条件：整数size（叶子结点数）、哈夫曼树HT存在。

操作结果：由哈夫曼树生成哈夫曼编码。

decode\_huff(hufftree HT, int n, char in\_file\_name[], char out\_file\_name[])

初始条件：哈夫曼树HT、整数n（叶子结点数\*2 -1，此时表示根结点位置）、文件名in\_file\_name、out\_file\_name存在。

操作结果：读取哈夫曼编码结果，根据哈夫曼树（哈夫曼编码）将其译码成原文章。

} ADT hufftree

ADT huffcode {

数据对象：D={ ai | ai∈TermSet, i=1,2,…,m, m≥0 }

（TermSet中的每个元素包含一个表示ASCII值的整数和表示权重的整数，以及一个存储哈夫曼编码的整数数组、一个表示数组起始位置的整数）

数据关系：R1={ <ai-1, ai> | ai-1, ai∈D, 且ai-1中的值<ai中的值, i=1,2,…,n }

基本操作：

gen\_huffcode(int size, hufftree HT, huffcode HC[])

初始条件：整数size（叶子结点数）、哈夫曼树HT存在。

操作结果：由哈夫曼树生成哈夫曼编码。

print\_huffcode(huffcode HC[], int n, char out\_file\_name[])

初始条件：哈夫曼编码结构体组HC[]、整数n、文件名out\_file\_name存在。

操作结果：展示哈夫曼编码并将其输出到文本文档。

code\_huff(story P, huffcode HC[], int n, char out\_file\_name[])

初始条件：线性表P、哈夫曼编码结构体组HC[]、整数n、文件名out\_file\_name存在。

操作结果：将线性表P中的文章编码为哈夫曼编码，结果存储到文本文档。

} ADT huffcode

2、主程序流程

（1）读取文章内容到存储文章的线性表结构中；

（2）计算此表中字符频度，频度统计结果存储到数组中，数组序号代表ASCII值；

（3）将此数组转化为线性表结构，ASCII值和频度成为并列关系，以频度大小排序线性表；

（4）由频度统计线性表生成哈夫曼树，并显示它；

（5）由哈夫曼树生成哈夫曼编码，并将它输出到文件；

（6）对原文进行哈夫曼编码并计时，编码结果输出到文件，计时输出到计时文件；

（7）对哈夫曼编码结果文件译码并计时，译码结果输出到文件，计时输出到计时文件；

（8）对原文进行等长编码并计时，编码结果输出到文件，计时输出到计时文件；

（9）对等长编码结果文件译码并计时，译码结果输出到文件，计时输出到计时文件。

3、函数调用关系

共计18个函数（不包括主函数main）。

（1）主函数main调用read\_txt、count\_ascii、array\_to\_list、gen\_hufftree、gen\_huffcode、print\_huffcode、code\_huff、decode\_huff、code\_ascii\_bin、decode\_ascii\_bin、char\_compare函数，可选择调用或不调用print\_list函数（默认不调用）、print\_hufftree函数（默认调用）；

（2）read\_txt函数可选择调用或不调用print\_story函数（默认不调用）；

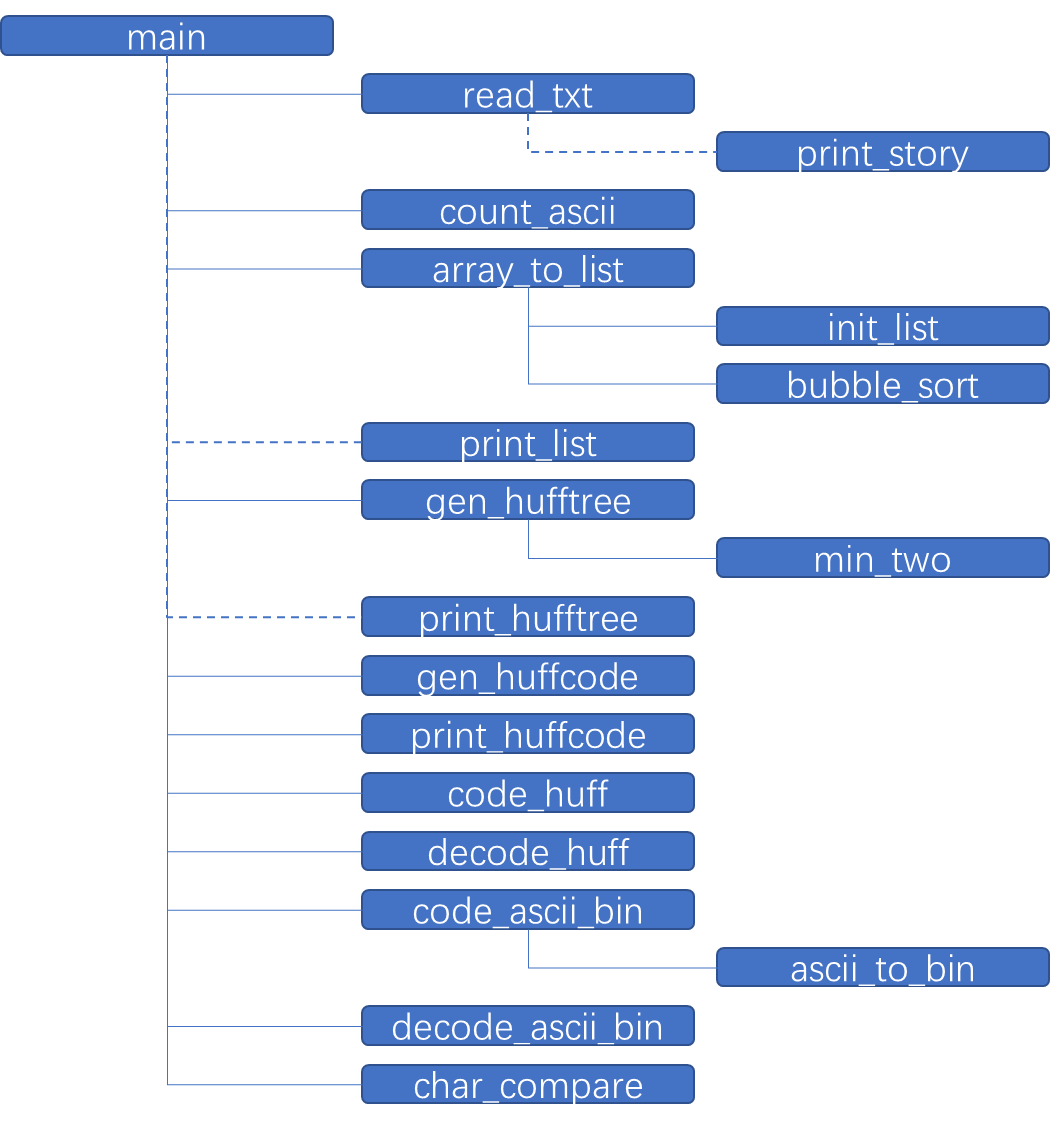
（3）array\_to\_list函数调用init\_list和bubble\_sort函数；

（4）gen\_hufftree函数调用min\_two函数；

（5）code\_ascii\_bin函数调用ascii\_to\_bin函数。

1. **详细设计**

1、函数调用关系图



2、实现功能

（1）定义存储文章、字符频度统计结果的线性表，以及哈夫曼树结构、存储哈夫曼编码的结构体组；

（2）可读取文章存储到线性表，也可在命令行输出显示文章；

（3）可求线性表中的ASCII字符频度并存入一维数组；

（4）可将存于一维数组的字符频度转化到线性表存储，还可将其排序；

（5）可根据字符频度生成哈夫曼树，并依此产生各字符的哈夫曼编码；

（6）可依生成的哈夫曼树和哈夫曼编码对文章进行编码和解码，并得出运算时间；

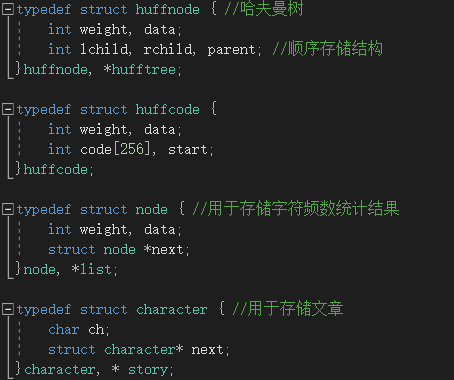
（7）可使用等长编码（ASCII码二进制形式）对文章进行编码和解码，并得出运算时间；

（8）可比较两文章（此处为编码前、解码后）的差异。

3、具体代码实现

（1）定义结构

定义了4种结构，分别用于存储哈夫曼树、哈夫曼编码、字符频数统计结果、文章所有字符。

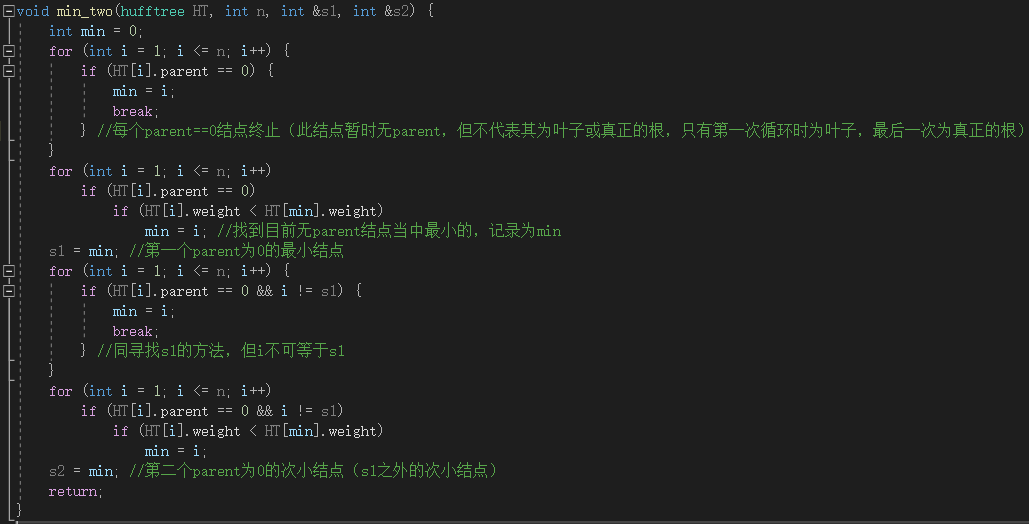


（2）自动寻找树中权重最小的2个结点子函数

需要四个参数，第1个为哈夫曼树，第2个为哈夫曼树的非叶子结点存储位置（哈夫曼树结构总大小为“叶子结点数\*2-1”，前面存放叶子结点，后面存放枝干-trunk结点，其数据域中不存放ASCII值，只包含权重值），第3个为找到的当前树最小结点，第4个为找到的当前树次小结点。

先对哈夫曼树的全部结点进行遍历（第一个for），找到此时的所有无parent的结点记录为min，（值得注意的是，哈夫曼树是从下往上构建，因此无parent的结点不一定是叶子也不一定是根）然后继续进行遍历（第二个for），寻找所有无parent结点当中最小的（不断循环更新），得到最小的无parent结点，完成这两次循环后记录s1=min，s1即为当前树中所有结点里最小的；然后重复前两个循环类似的方法，但加上条件i不等于s1，即可寻找到第二小的结点，记录此结点为s2。

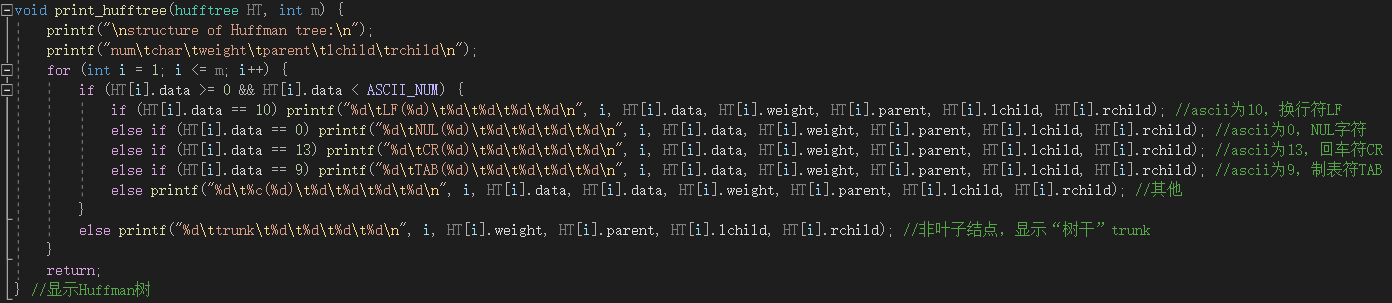
由于s1、s2均由外部输入，所以他们二者的赋值均在此函数中被赋值、改变（会返回），因此它们可以用到此函数外的函数的运算当中。



（3）展示哈夫曼树子函数

需要两个输入参数， 其中第1个为哈夫曼树，第2个为哈夫曼树的结点数。

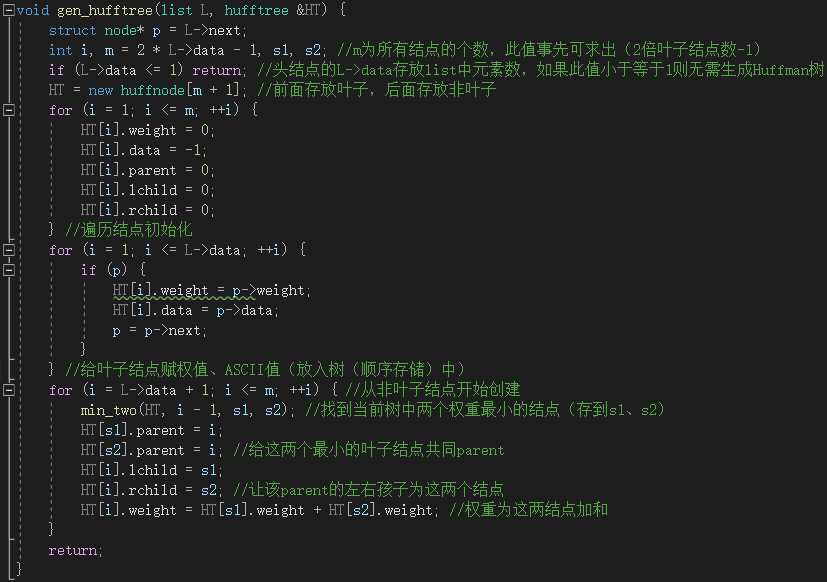
可以展示当前树的结构，输出内容包括结点位置、ASCII值、权重、parent子树位置、左子树位置、右子树位置。对于一些特别的字符例如换行符、回车符、制表符、NUL符，将显示其字母简写而非字符本身，非叶子结点没有ASCII值，则此位置显示trunk。



（4）生成哈夫曼树子函数

需要两个参数，一个是储存字符频度的链表，另一个是哈夫曼树名称。

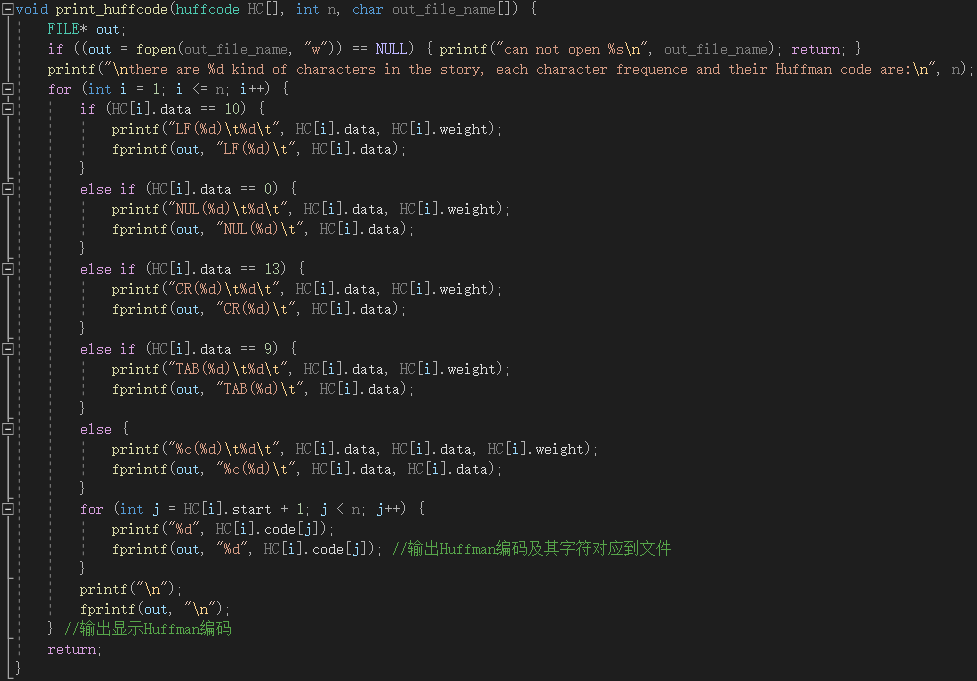
首先遍历哈夫曼树结构，对其初始化赋值（第一个for），第二次遍历时同时遍历了字符频度链表，此过程将字符频度（权重）、字符转移到了哈夫曼树结构中；完成了对哈夫曼树的叶子节点赋值（第二个for）。第三次遍历，每次将最小的和次小的结点合并生成枝干结点（存储位置位于n+1到2\*n-1，n为叶子结点数），其权重等于两孩子结点之和；枝干结点的左右孩子即为两个被合成的结点，这两个孩子结点的parent结点也就是此枝干结点。



（5）展示、输出哈夫曼编码子函数

需要三个参数，第1个是哈夫曼编码结构体组，第2个是哈夫曼编码包括的字符种数，第3个是输出哈夫曼编码的文件名。

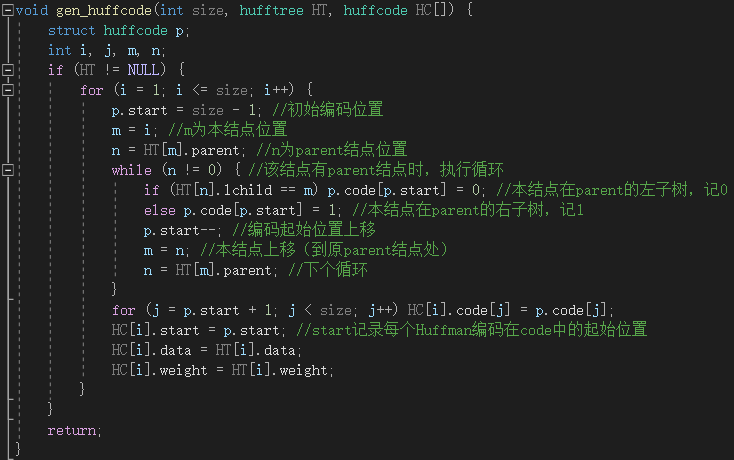
在命令行显示字符（及其ASCII码）、权重、其对应的哈夫曼编码；在输出文件中显示字符（及其ASCII码）、其对应的哈夫曼编码。



（6）生成哈夫曼编码子函数

需要三个参数，第1个为哈夫曼编码的字符种数，第2个为哈夫曼树，第3个为生成的哈夫曼编码结构体组。

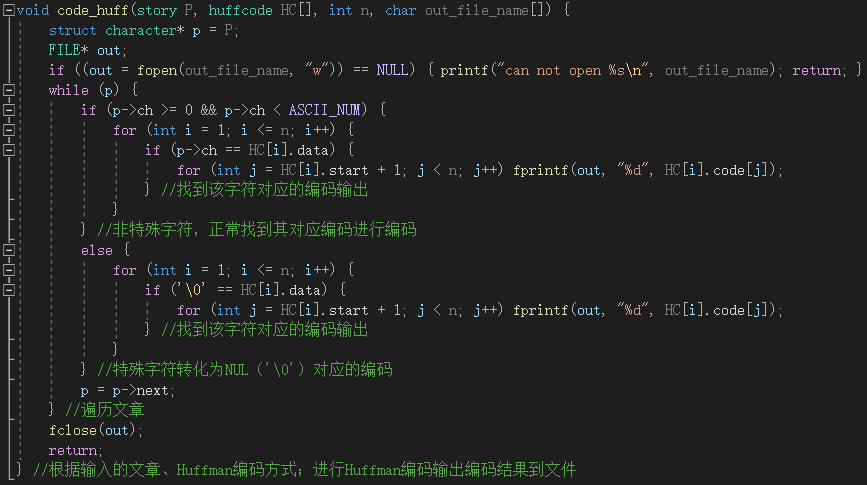
首先判断哈夫曼树是否为空，如果空则不进行后续操作。然后执行循环（for），记录初始编码位置和当前结点位置、当前结点parent结点位置，再嵌套循环（while）判断当前结点是否是其parent结点的左孩子，如果是则记录哈夫曼编码0，如果不是则为右孩子，记录编码1；然后将编码起始位置前移，并将当前结点改为其parent结点，parent结点再上移为其自己的parent结点，重复此循环，最终完成该字符的哈夫曼编码存储于p.code当中了。然后再将编码完成的数据全部存入哈夫曼编码结构体当中，包括哈夫曼编码数组、编码起始位置（start）、ASCII值、权重。



（7）哈夫曼编码子函数

需要四个参数，第1个为存储文章字符的线性表，第2个为哈夫曼编码结构体组，第3个为字符种数，第4个为输出文件名。

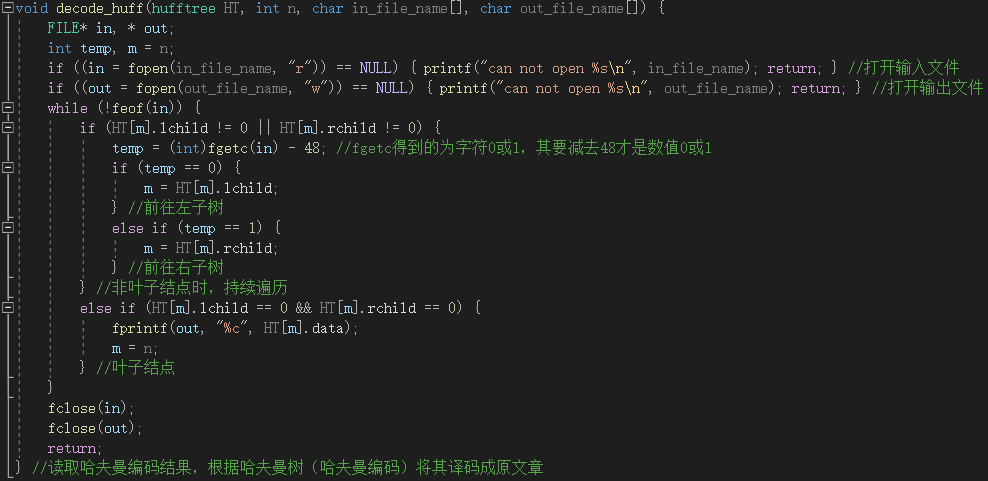
遍历输入的文章字符，其中的ASCII字符按照每个字符对应的哈夫曼编码结构体组进行编码，而非ASCII字符按照NUL字符对应的哈夫曼编码来编码。这其中的for循环（对i循环）即为找到每个字符对应结构体组中的哪个哈夫曼编码的遍历，而里面的对j的for循环则是对哈夫曼编码结构体内code数组的循环输出，输出出该字符对应的哈夫曼编码。



（8）哈夫曼译码子函数

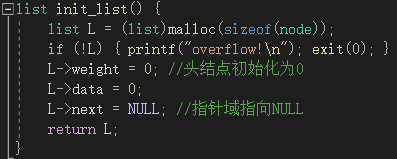
需要四个参数，第1个为哈夫曼树，第2个为哈夫曼树结点总数（2倍字符种数-1，此数值即为根结点位置对应的数值），第3个为哈夫曼编码的文件名，第4个为哈夫曼译码要输出的文件名。

遍历输入文件（哈夫曼编码的文件，由二进制数01构成），当哈夫曼树不处于叶子结点时（由临时变量m确定），让临时数值等于遍历到的字符，若为0，则前往左孩子，若为1则前往右孩子；直到抵达叶子结点时，将叶子结点存储的ASCII值并以字符形式输出，令m重新等于根结点位置，重复循环，完成全部译码。



（9）初始化线性表子函数

初始化生成带头结点的线性表，给头结点的数据域赋值为0，指针域指空，返回值是初始化成功的线性表。



（10）冒泡排序子函数

输入参数为已有的线性表（储存字符和其频度数据）。

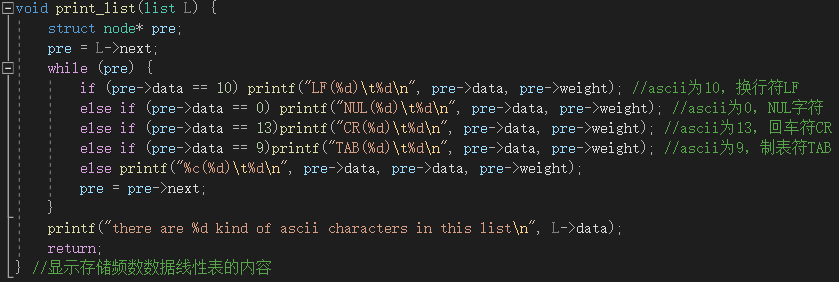
遍历线性表，通过冒泡排序算法调整元素位置，按权重值从小到大排序。



（11）展示频度统计线性表内容子函数

输入参数为已有的线性表（储存字符和其频度数据）。

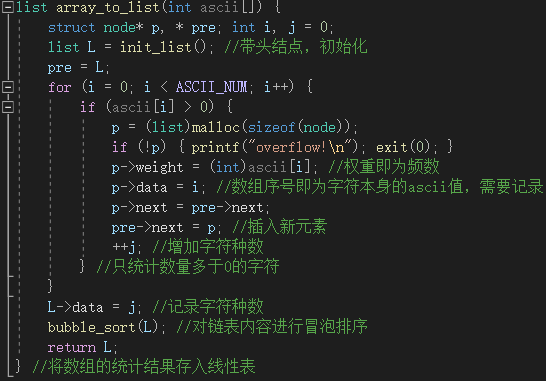
遍历线性表，输出字符（及其对应的ASCII码）、权重，并最终显示线性表表长。



（12）数组转线性表子函数

输入参数为储存字符频度的数组。

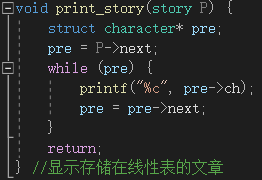
首先初始化线性表，然后遍历数组将其中频度大于0（有统计到频度的字符）的字符和其频度（权重）储存到线性表当中，同时统计出字符种数。形成线性表后，再对其进行冒泡排序，使其中元素按权重从小到大排序。



（13）展示文章内容线性表子函数

输入参数为按顺序储存文章全部字符的线性表。

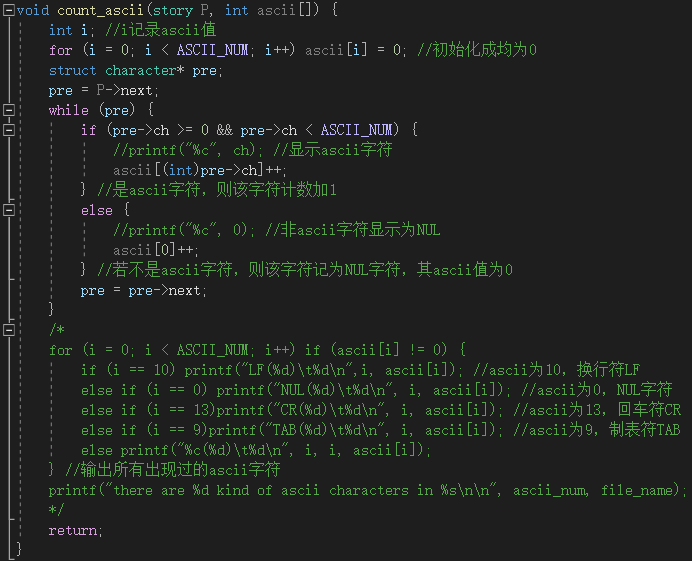
遍历该线性表，依次将字符显示到命令行窗口。



（14）求文章ASCII字符频度子函数

需要的参数为按顺序储存文章全部字符的线性表和一个一维数组。

首先初始化该数组，然后遍历线性表，遇到ASCII字符则将数组相应位置（数组大小为256，每个ASCII码对应一个位置）的值加1，若为非ASCII字符则记为NUL字符，故数组的0位加1（NUL字符对应ASCII值为0）。



（15）读取文章子函数

需要的输入参数为储存英文文章文件名。

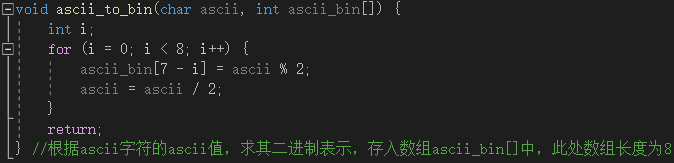
首先创建了线性表P，其没有头结点。然后遍历输入文件，将每个检测输入的字符按读取顺序存入线性表P，先读取的靠近表头，后读取的加入表尾。在遍历的同时记录文章包含的字符总数，最后在完成读取后显示统计结果输出到命令行。



（16）ASCII码转二进制码子函数

需要输入参数为ASCII字符、储存二进制码的数组。

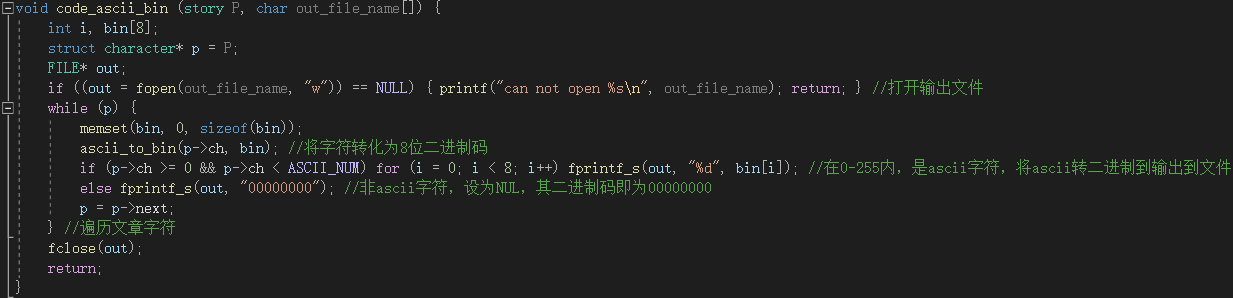
按照二进制码与ASCII码转换算法得出换算结果，储存到数组。



（17）等长编码子函数

输入参数为储存文章字符的线性表、输出等长编码结果的文件名。

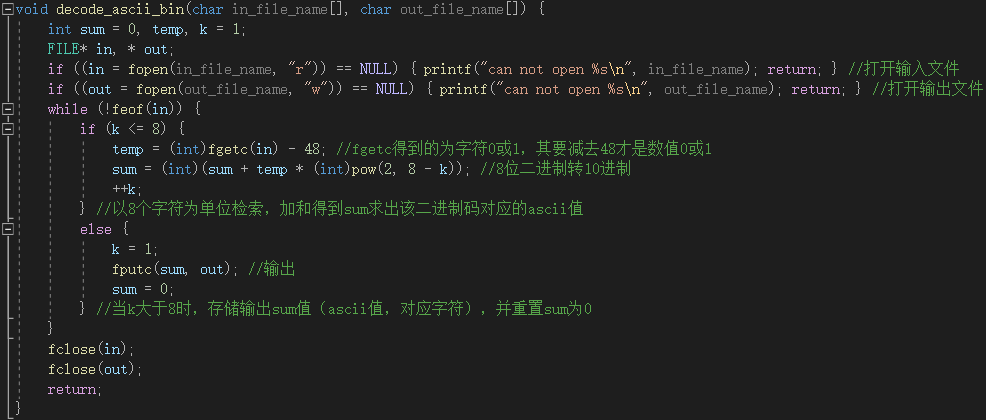
遍历线性表，将ASCII字符转换为二进制码并将其输出到文件，对于非ASCII字符则输出8个0（对应NUL字符）。



（18）等长译码子函数

输入参数为输入输出文件名。

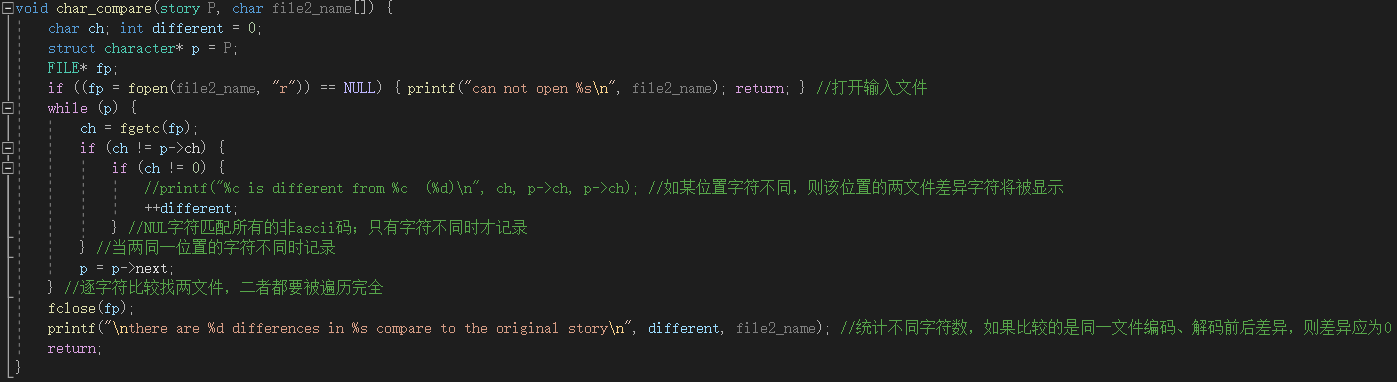
遍历输入文件，当标记值小于8时持续读取，直到读至8位根据计算求和结果得出此8位对应的ASCII码才将该ASCII字符输出到文件，再令标记值归位、求和值归0。



（19）字符比较子函数

输入参数为储存文章全部字符的线性表和译码结果文件名。

遍历线性表，逐一比较每个字符与译码结果字符是否有差异，完成遍历后统计出差异字符总数并显示到命令行。



（20）主函数

主函数调用了几乎所有子函数，展示了它们的使用顺序和方法。

1. **调试分析**

1、调试情况

在统计ASCII字符频度时本先考虑使用自定义结构，但发现若想同时确定ASCII值和其量，仍是数组最为方便，故选择先将频度统计录入数组，然后再转入线性表（去除频度为0的并完成排序）。

设计译码子函数的时候频繁发现译码结果大量乱码，或是无法译码，后发现问题出在读取译码结果的01时读取的为字符而非整数值，未减48，产生一系列错误。修正后结果正常。

2、算法的时空分析

min\_two时间复杂度为O(4n)；

print\_hufftree时间复杂度为O(n)；

gen\_hufftree时间复杂度为O(3n-1+(n-1)\*4\*(n-1))；

print\_huffcode时间复杂度为O(n)；

gen\_huffcode时间复杂度为O(n\*(n-1))；

code\_huff时间复杂度为O(m\*n)；（m、n分别指文章字符数、编码字符种数）

decode\_huff时间复杂度为O(n)；

init\_list时间复杂度为O(1)；

bubble\_sort时间复杂度为O(n^2)。

print\_list时间复杂度为O(n)；

array\_to\_list时间复杂度为O(n)；

print\_story时间复杂度为O(n)；

count\_ascii时间复杂度为O(n)；

read\_txt时间复杂度为O(n)；

ascii\_to\_bin时间复杂度为O(1)；

code\_ascii\_bin时间复杂度为O(n)；

decode\_ascii\_bin时间复杂度为O(n)；

char\_compare时间复杂度为O(n)；

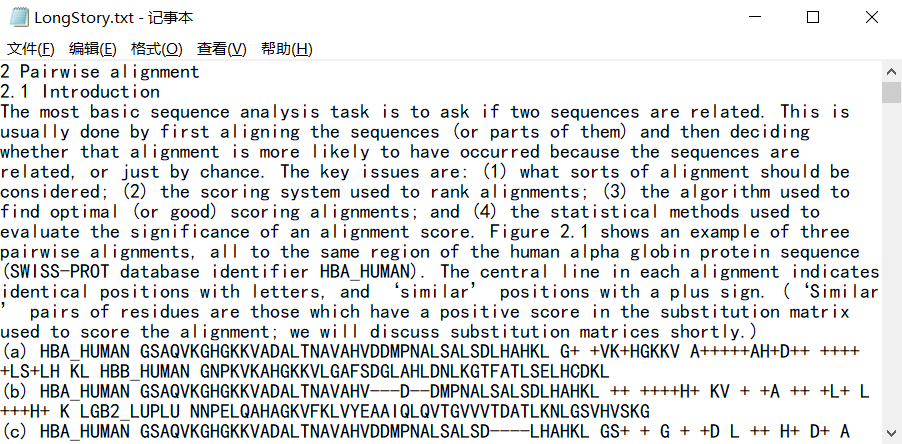
它们的空间复杂度均为O(1)。

3、经验和体会

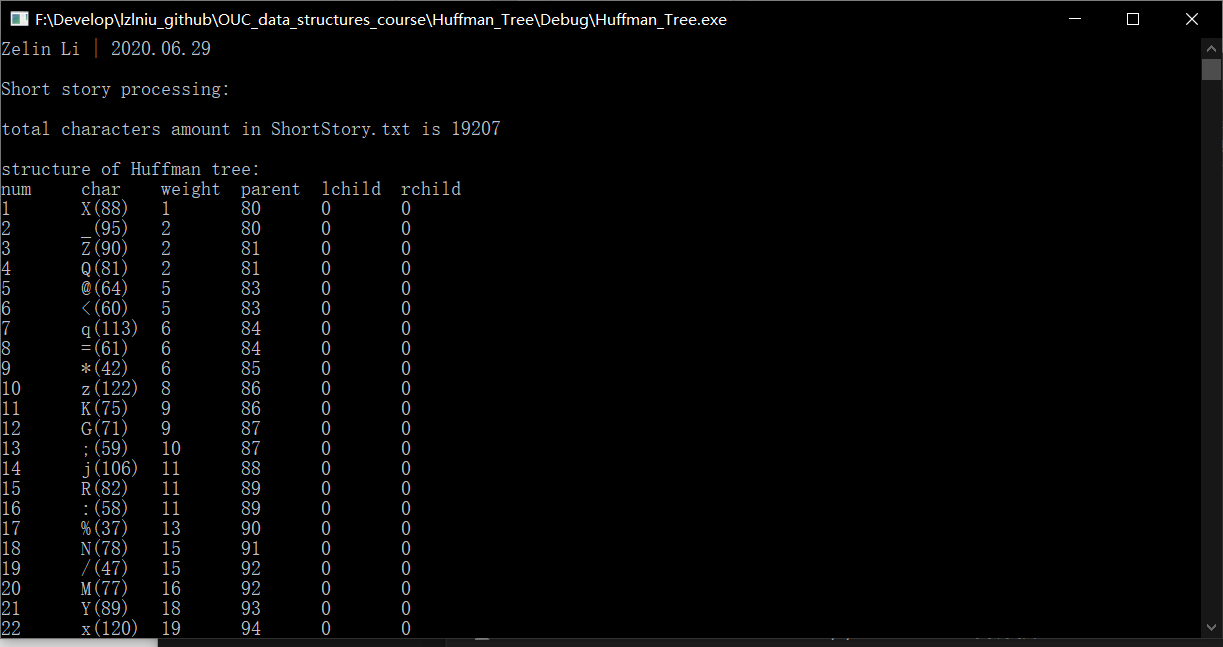
本程序理论上可以更高效。主要有2点：一是最好不将数据（字符频度）在不同阶段存于不同的结构中，这样导致数据需要在结构间传递，增加了运算耗时。二是最好文章全部字符的输出和转换在结构中进行，例如编码解码，都可以存储在story结构里运算，而不是每次输出到文件，然后再从文件读入；应该只有在需要输出成文件时才输出，在需要编码译码时才输入，这样可以减少I/O次数，增大运算效率。

1. **用户使用说明**

准备两个英文文章，其文本文档放于可执行程序（Huffman\_Tree.exe）目录下，文章内容类似如下图所示：

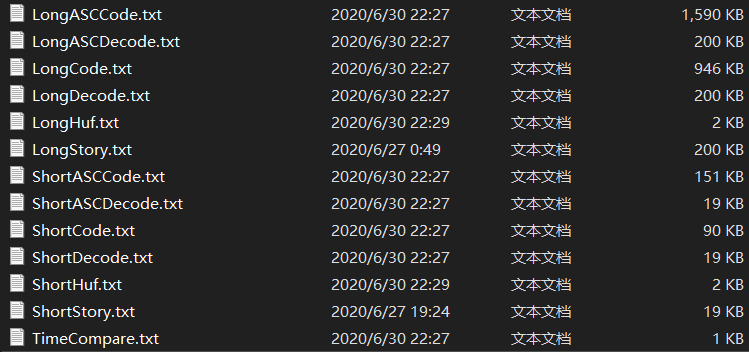


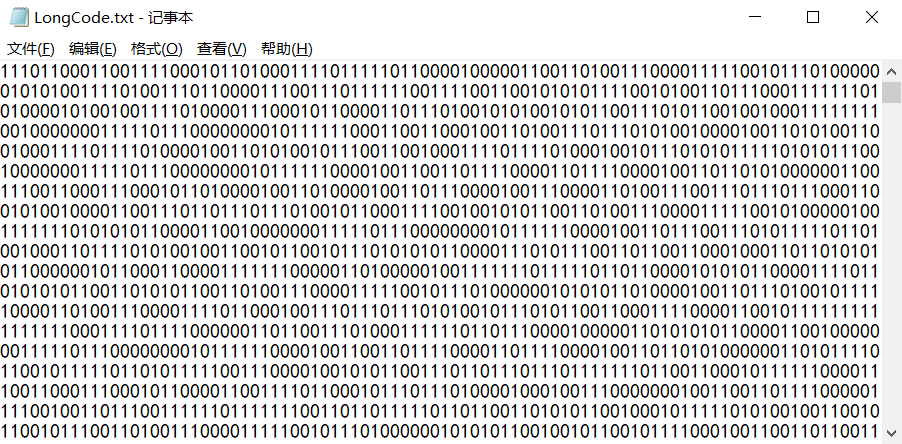
双击可执行程序得到输出：





其输出文件如下图所示（具体结果详见源文件目录各文档）：





1. **测试结果**

测试数据见源文件目录下ShortStory.txt、LongStory.txt。

进行数据测试，得出：

|  |  |  |
| --- | --- | --- |
| 统计项\文章 | 短 | 长 |
| 字数 | 2980 | 31960 |
| 字符数 | 19207 | 203446 |
| 字符种数 | 79 | 88 |
| 最短哈夫曼编码位数 | 3 | 3 |
| 最长哈夫曼编码位数 | 13 | 16 |
| 哈夫曼编码译码与原文差异字符数 | 0 | 0 |
| 等长(ASCII)编码译码与原文差异字符数 | 0 | 0 |
| 哈夫曼编码码长(字符0/1数) | 91416 | 967875 |
| 等长编码码长(字符0/1数) | 153656 | 1627568 |
| 哈夫曼编码耗时(ms) | 70 | 740 |
| 哈夫曼译码耗时(ms) | 29 | 284 |
| 等长编码耗时(ms) | 110 | 1158 |
| 等长译码耗时(ms) | 39 | 383 |
| 全部编码译码总耗时(ms) | 248 | 2565 |

注：运算耗时数据取自测试结果“TimeCompare1.txt”；编码结果码长由Notepad++统计。

1、运算时间分析：

两文件字符数相差1059.2%，总编码耗时相差1034.3%，二者接近，可见编码耗时与文件大小成正比。

短文章等长编码与哈夫曼编码耗时之比157.14%，长文章等长编码与哈夫曼编码耗时之比156.49%。短文章等长译码与哈夫曼译码耗时之比134.48%，长文章等长译码与哈夫曼译码耗时之比134.86%。可见文章长短对两种编码译码效率之比几乎无影响，哈夫曼编码效率更优，且哈夫曼编码比译码在效率提升上更突出。

2、码长分析：

本例中两文件的字符数之比近1059.2%，哈夫曼编码码长之比1058.8%，等长编码码长之比1059.2%，三者接近，可见码长与文件大小成正比。

短文章等长编码与哈夫曼编码码长之比168.08%，长文章等长编码与哈夫曼编码码长之比168.16%，可见哈夫曼编码更优（码长更短），但哈夫曼编码相对等长编码对长短文章编码的优势程度（比例）没有明显差异。

1. 附录

带注释的源代码见源文件中的“源.cpp”。