# 哈夫曼压缩文件实训指导

## 一、压缩原理

同样的信息，不同的编码方式，会产生出不同长度的编码，利用哈夫曼编码进行文件压缩的基本原理就是利用哈夫曼编码所产生的编码是所有的编码方式中，编码最短的编码方式。从而达到压缩的目的。

思考：利用哈夫曼编码对文件进行压缩，其压缩率的合理范围是多少？你还知道哪些压缩的方法？

### 1.1 等长编码

任何文件在计算机内存中都是以字节为最小单位进行存储的，所以可以把任何文件看作是利用8位（一个字节）作为等长编码编码方式，编码而成的文件。往往这种编码方式并不是最优的编码方式。

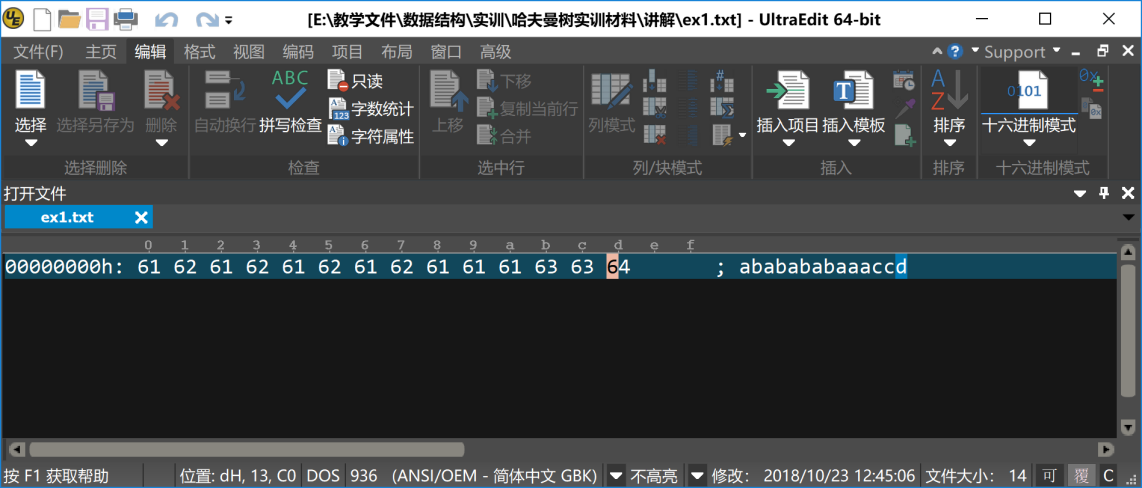
例如：对电文“ababababaaaccd”利用8位的ASCII码进行等长编码，其编码如下：

|  |  |
| --- | --- |
| 原文字符 | 等长编码 |
| a | 01100001 |
| b | 01100010 |
| c | 01100011 |
| d | 01100100 |

则源文件存储的编码为（备注：为便于察看，在每个字符编码之间添加了空格，实际编码时没有这个空格）：

01100001 01100010 01100001 01100010 01100001 01100010 01100001 01100010 01100001 01100001 01100001 01100011 01100011 01100100

使用文本编辑软件uedit 按2进制的方式查看该文本，可以看出以下的内容



可以计算出上述纯文本文件的编码长度为14个字节112位（备注：当文件的大小不足1K，操作系统也会给它分配1K的存储空间。）

### 1.2 哈夫曼编码

如果对电文“ababababaaaccd”进行哈夫曼编码，则其编码方式如下：

|  |  |  |
| --- | --- | --- |
| 原文字符 | 字符频度 | 哈夫曼编码 |
| a | 7 | 0 |
| b | 4 | 11 |
| c | 2 | 101 |
| d | 1 | 100 |

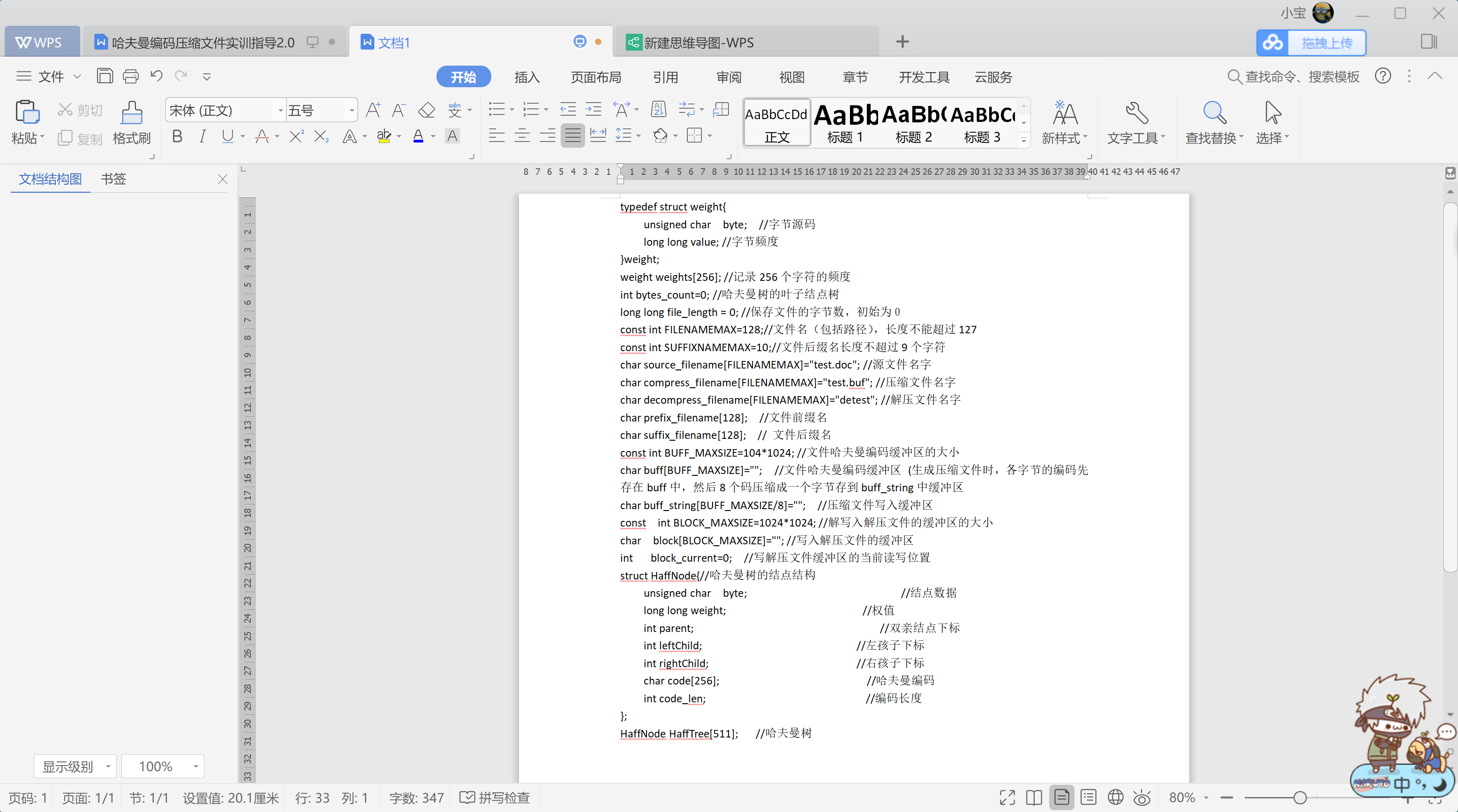
则源文件可以利用哈夫曼编码，编码为：0 11 0 11 0 11 0 11 0 0 0 101 101 100

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | a | b | a | b | a | b | a | a | a | c | c | d |
| 0 | 11 | 0 | 11 | 0 | 11 | 0 | 11 | 0 | 0 | 0 | 101 | 101 | 100 |

可以计算出上述编码长度为24位，共需要3个字节（备注：计算以字节为单文，所有当存储哈夫曼编码的文件的时候，若编码长度不是8的倍数时，最后一个字节需要补充为一个字节。）

## 二、实验步骤

### 2.1 全局变量定义



### 2.2 生成压缩文件

#### 1、初始化哈夫曼树

实现函数：void initHaffTree();

功能：初始化HaffTree[511]数组的每一个元素的每一项的初始值如下表所示。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 角标i | byte | weight | lchild | rchild | parrent | code[] |
| 0 | 0 | 0 | -1 | -1 | -1 | “”（空串） |
| 1 | 1 | 0 | -1 | -1 | -1 | “” |
| 2 | 2 | 0 | -1 | -1 | -1 | “” |
| 3 | 3 | 0 | -1 | -1 | -1 | “” |
| 4 | 4 | 0 | -1 | -1 | -1 | “” |
| 5 | 5 | 0 | -1 | -1 | -1 | “” |
| 6 | 6 | 0 | -1 | -1 | -1 | “” |
| ... |  | 0 | -1 | -1 | -1 | “” |
| 510 | 510 | 0 | -1 | -1 | -1 | “” |

#### 2、读源文件，统计源文件中各字节出现的次数

实现函数：void Statistics(char fileName[]) ; //统计文件各字节的频度

功能：以只读方式打开filename[]指定的二进制文件，利用HaffTree[i].weight统计文件中各种字节出现的次数，并统计weight值大于0的字节个数bytes\_count。

以test.txt为例，其结果如下表所示

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 角标i | byte | weight | lchild | rchild | parrent | code[] |
| 0 | 0 | 0 | -1 | -1 | -1 | “”（空串） |
| ... |  |  |  |  |  |  |
| 97 | 97(‘a’) | 7 | -1 | -1 | -1 | “” |
| 98 | 98(‘b’) | 4 | -1 | -1 | -1 | “” |
| 99 | 99(‘c’) | 2 | -1 | -1 | -1 | “” |
| 100 | 100(‘d’) | 1 | -1 | -1 | -1 | “” |
| ... | --- | 0 | -1 | -1 | -1 | “” |
| 511 | --- | 0 | -1 | -1 | -1 | “” |

bytes\_count=4

（注意：byte的类型必须定义为unsigned char类型，上标中byte的值是按10进制显示的）

#### 以weight作为关键字，进行降序排序

参考代码如下：

//使用sort需要包含以下文件

#include <algorithm>

//定义两个元素的比较方式

bool compare(HaffNode a,HaffNode b){

return a.weight >b.weight ; //降序

}

//HaffTee中的元素进行排序

//参数说明：待排序区间开始地址，待排序区间结束地址，比较方式

sort(HaffTree,HaffTree+255,compare);

功能说明：利用sort函数对HaffTree[0]~HaffTree[255]的元素以weight为排序关键字进行降序（可以通过修改compare函数改变排序方式）排序。

以test.txt为例，其结果如下表所示

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 角标i | byte | weight | lchild | rchild | parrent | code[] |
| 0 | 97(‘a’) | 7 | -1 | -1 | -1 | “”（空串） |
| 1 | 98(‘b’) | 4 | -1 | -1 | -1 | “” |
| 2 | 99(‘c’) | 2 | -1 | -1 | 4 | “” |
| 3 | 100(‘d’) | 1 | -1 | -1 | 4 | “” |
| 4 | --- | 3 | 3 | 2 | -1 | “” |
| 5 | --- | -1 | -1 | -1 | -1 | “” |
| 6 | --- | 0 | -1 | -1 | -1 | “” |
| ... | --- | 0 | -1 | -1 | -1 | “” |
| 511 | --- | 0 | -1 | -1 | -1 | “” |

（注意：排序是为了把没有出现的字节过滤掉，对字节进行编码时不需要对这些没有出现的字节进行编码，因此上例中，只需对4个字节进行编码，所以在构造哈夫曼树时，实际上只需要用到7个结点的存储空间，即HaffTree[0]~HaffTree[bytes\_count\*2-2]。））

#### 构造哈夫曼树

实现函数：void creatHaffCode( )

功能：根据数组HaffTree中各叶子结点的权值，构造哈夫曼树。

算法框架

void creatHaffTree(){

初始化哈夫曼树结构（initHaffTree(weights);）

求哈夫曼树叶子结点树bytes\_count；

for(int i=1;i<bytes\_count;i++){ //经过bytes\_count -1 次树的合并。

查找0~bytes\_count结点中parrent值为-1，且weight值最小的结点位置，记为min1

查找0~bytes\_count结点中parrent值为-1，且weight值第二小的结点位置，记为min2 合并min1和min2两个子树，合并出的根结点存放在bytes\_count+i-1结点上，即：

1. HaffTree[min1].parrent= bytes\_count+i-1
2. HaffTree[min2].parrent= bytes\_count+i-1
3. HaffTree[bytes\_count+i-1].lchild= min1 (左子树的权值小于等于右子树)
4. HaffTree[bytes\_count+i-1].rchild= min2
5. HaffTree[bytes\_count+i-1].weight=HaffTree[min1].weight+HaffTree[min2].weight

}

以示例文件为例，根据文件中各字节出现的频度的统计结果，构造出的哈夫曼树的逻辑结构如下图所示：



其存储结构如下表所示：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 角标i | byte | weight | lchild | rchild | parrent | code[] |
| 0 | 97(‘a’) | 7 | -1 | -1 | 6 | “”（空串） |
| 1 | 98(‘b’) | 4 | -1 | -1 | 5 | “” |
| 2 | 99(‘c’) | 2 | -1 | -1 | 4 | “” |
| 3 | 100(‘d’) | 1 | -1 | -1 | 4 | “” |
| 4 | --- | 3 | 3 | 2 | 5 | “” |
| 5 |  | 7 | 4 | 1 | 6 | “” |
| 6 | --- | 14 | 0 | 5 | -1 | “” |
| ... | --- | 0 | -1 | -1 | -1 | “” |
| 255 | --- | 0 | -1 | -1 | -1 | “” |

#### 构造哈夫曼编码

实现函数：void creatHaffTree()

功能：根据HaffTree[]数组中的哈夫曼树的，构造出每个结点（包括分支结点）编码字符串



算法框架

void creatHaffCode(){

计算根结点位置root=bytes\_count\*2-2; 设置根结点的编码为””(空串)

for(int i=root-1;i>=0;i--){ //从下往上遍历数组

if(第i个结点是其双亲的右孩子结点){

第i个结点的编码为其双亲的编码后再接上‘1’

}

else{

第i个结点的编码为其双亲的编码后再接上‘0’

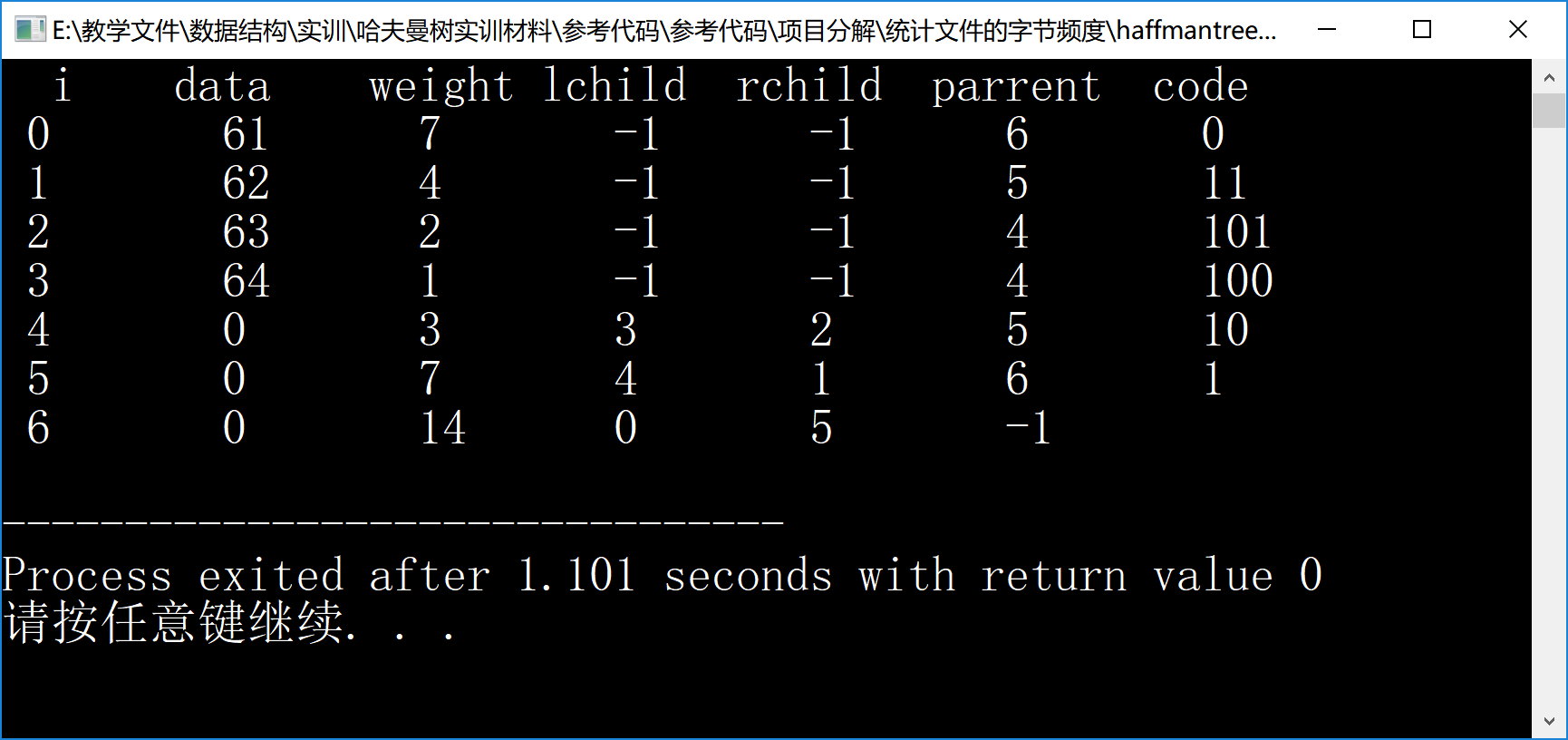
}

}

}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 角标i | byte | weight | lchild | rchild | parrent | code[] |
| 0 | 97(‘a’) | 7 | -1 | -1 | 6 | “0” |
| 1 | 98(‘b’) | 4 | -1 | -1 | 5 | “11” |
| 2 | 99(‘c’) | 2 | -1 | -1 | 4 | “101” |
| 3 | 100(‘d’) | 1 | -1 | -1 | 4 | “100” |
| 4 | --- | 3 | 3 | 2 | 5 | “10” |
| 5 | --- | 4 | 4 | 1 | 6 | “1” |
| 6 | --- | 7 | 0 | 5 | -1 | “” |
| ... | --- | -1 | -1 | -1 | -1 | “” |
| 255 | --- | -1 | -1 | -1 | -1 | “” |

注意：建议在做到这一步时，利用上述的例子进行测试，并输出哈夫曼编码的构造结果（如下图所示，下图输出中data即byte，其值有16进制输出），确保哈夫曼编码的构造是否正确的，当然要注意哈夫曼树是不唯一的



#### 生成压缩文件

实现函数：void writeCompressFile()

功能：根据所求的哈夫曼编码方式，对源文件中的各个字节进行重新编码，生成压缩文件。

原理说明：压缩文件有两部分所组成。第一部分是包含源文件基本信息的文件头部信息，第二部分是源文件进行哈夫曼编码后的正文信息。

第一部分：头部信息，包括源文件的扩展名长度，扩展名，源文件的字节总数，被编码的字节总数n，字节1，字节1的频度，字节2，字节2的频度，......，字节n，字节n的频度。

例：以上述的电文压缩为例，文件的头部可以写入以下信息

3，txt，14，4，a，7，b，4，c，2，d，1

说明：3(后缀名长度)， txt(原文件的后缀名)，14(电文中字节总数)，4(有4个不同的编码)，a(被编码的原字节),7(a在源文件中出现了7次),……(后面也是各个字节出现的次数。)

注意：1、在写入这些信息的时候，要注意各项信息所占的字节数，这对后续解压的正确读取非常重要。

1. 编写程序到此处时，应该尝试把压缩文件的头部信息写入一个文本文件并查看一下信息写入是否正确，确保正确无误之后才进行下一步编程。

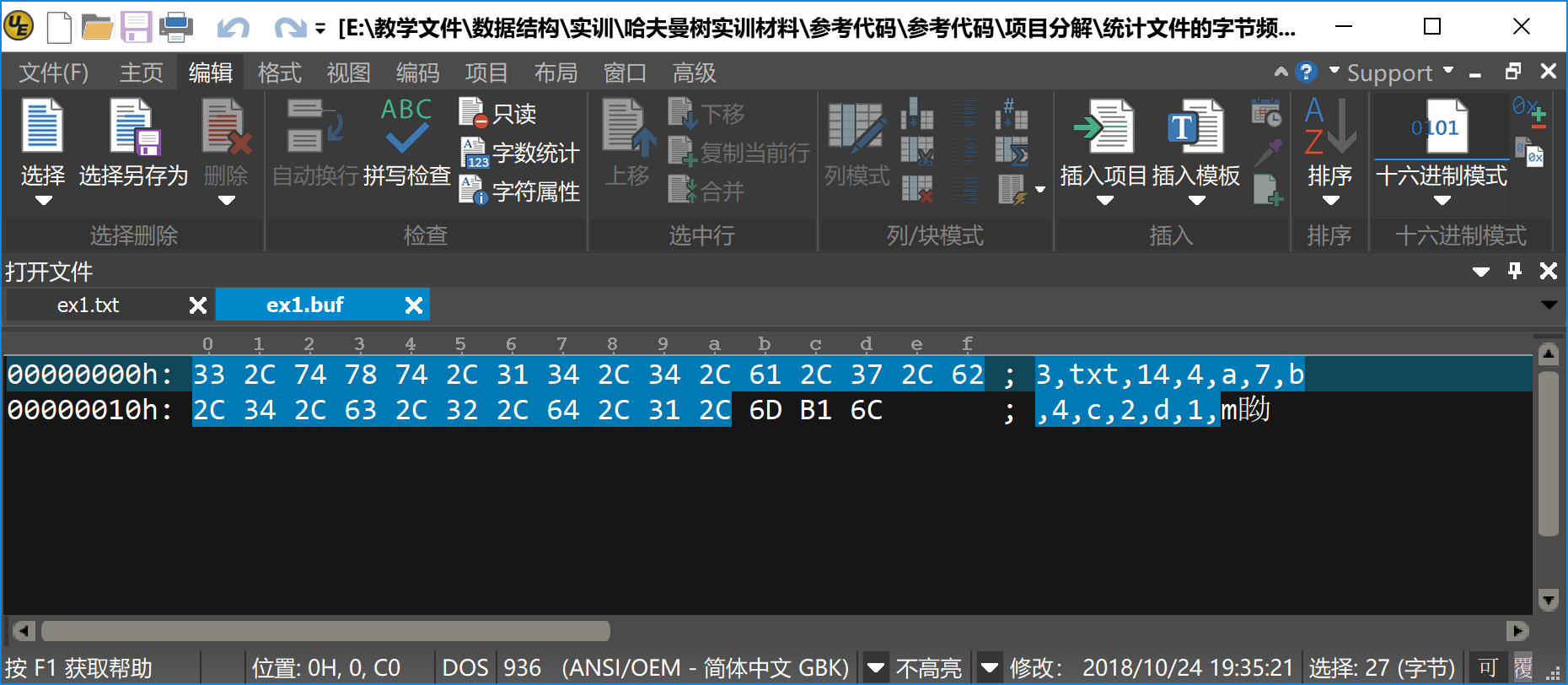
第二部分：正文信息，原文中的各个字节以哈夫曼编码重新编码后，再按8个码一个字节的方式重新组织的正文信息。

例：以上述的电文压缩为例，文件的正文14个字节，利用哈夫曼编码重新编码后为：

0 11 0 11 0 11 0 11 0 0 0 101 101 100（注意编码中不含空格）

然后对上述编码进行8位一个字节的方式进行切割，得到了一下3个字节

|  |  |  |  |
| --- | --- | --- | --- |
| 二进制 | 01101101 | 10110001 | 01101100 |
| 16进制 | 6D | B1 | 6C |



**头部**

**头部**

**正文**

代码框架

void writeCompressFile( ){//生成压缩文件

从源文件的名中获取文件的类型

以读的方式打开二进制源文件ifp；以写的方式打开二进制压缩文件ofp;

向ofp文件中写入头部信息(注意，一定要先确保头部信息无误才往下编写程序)

清空buff缓冲区；

从源文件ifp中读取一个字节到 c

while (ifp文件没有结束)

{

if(缓冲区剩余的空间至少还可以多放一个字节的哈夫曼编码){

（注意：哈夫曼编码的长度小于256）

查出c字节对应的哈夫曼编码字符串，接入到缓冲区中。

从源文件ifp中再读取一个字节到 c

}

else{//缓冲区快要满了

把缓冲区中尽可能多的内容写入文件中。

（注意：缓冲区中的一个元素，对应一位；即需要把缓冲区中的每8个元素先压成一个字节，才能写入文件压缩文件中，建议在切分字节之前，先把哈夫曼编码数出来看一看，确保无误之后在切分。）

}

}

把缓冲区中剩余的字符写入文件

（注意：若缓冲区中剩余的字符数量不是8的倍数时，即最后一个字节是不完整的，需要在低位进行补0，再写入）

}

关闭ofp文件

关闭ifp文件

}

### 2.3 解压压缩文件

实现函数：void uncompress()

功能：解压压缩文件，生成与源文件类型相同的文件（文件的后缀名必须要相同）

原理说明：首先从压缩文件中读取出文件的头部信息：源文件的扩展名长度，扩展名，源文件的字节总数，被编码的字节总数n，字节1，字节1的频度，字节2，字节2的频度，......，字节n，字节n的频度。然后在通过上述字节频度信息还原哈夫曼树（哈夫曼编码可以不构造）。最后，读压缩文件的正文，以哈夫曼树为字典，识别出原字节后，把原字节写入解压文件中。

void deCompressFile( ){//解压文件件

打开压缩文件

开始读文件头部（包括源文件后缀名,字节总数，被编码字节数，各字节的字节频度）

initHaffTree( );//初始化哈夫曼树存储结构

creatHaffTree();//构造哈夫曼树

初始化读出数据缓冲区buff为空（压缩文件批量读出，有利于提高解压速度）

初始化写入数据缓冲区block空（解压出的字节，批量写入有利于提高解压速度）

初始化解压出来的字节数refile\_lenght=0；

初始化读缓冲区字节游标buff\_current=0；

初始化读缓冲区位游标byte\_current=0;

初始化当前哈夫曼结点tree\_current=bytes\_count-2;// 根结点位置

读压缩文件，装满读缓冲区 ；

while(refile\_lenght != 源文件的字节总数){ //源文件未被解压完成

if(tree\_current位置的结点是叶子结点){

把该结点的的原字节，接入block缓冲区中，

如果block缓冲区满，则把block缓冲区中的内容写入解压文件中。

tree\_current回到根结点位置。

}

else{

If(buff缓冲区中的buff\_current位置的从左数第byte\_current位是1)

tree\_current=tree\_current结点的右孩子结点;

else

tree\_current=tree\_current结点的左孩子结点；

}

if(byte\_current<7) //一个字节未完

byte\_current++’；// 字节游标不变，位游标加1

else{

byte\_current=0;

buff\_current++;

If(buff\_current==BUFF\_MAXSIZE){ //缓冲区中的数据已经用完了

buff装入新的数据;

buff\_current=0;

}//if

}//else

}//while

把block缓冲区中剩余的字节，写入解压文件中。

关闭ifp文件；

关闭ofp文件；

}