创新性探索课题实践报告

选题：利用概率统计方法实现英语文本压缩

小组成员

姓 名	学 号
张 闯	18325071
郝裕玮	18329015
张航悦	19335262
许 群	19335236

摘要

在信息的存储与快速传输过程中，数据压缩有着非常重要的作用。本文以哈夫曼算法为例，探究利用概率统计方法对英语文本进行压缩的基本原理，给出 C++编程实现压缩指定及随机文本的实践过程，并在此基础上对哈夫曼压缩算法的性能进一步分析。

引言

随着计算机技术的快速发展，各种系统数据量越来越大，给信息存储及网络运输带来诸多困难。为了节省信息的存储空间和提高信息的传输效率，必须对大量的实际数据进行压缩。实践证明，采用数据压缩技术可以节省 80%以上的费用，且一些难点问题能够通过压缩技术得以实现。

文本压缩是根据一定方法对大量数据进行编码处理以达到信息压缩存储的过程，被压缩的数据应该能够通过解码恢复到压缩以前的原状态，而不会发生信息丢失现象。在数据的存储和表示中常常存在一定的冗余度，一些研究者提出了不同的理论模型和编码技术来降低数据的冗余度。Ziv Jacob 提出了一种基

于字典模型的压缩方法。而基于统计模型的哈夫曼方法以统计推理为数学基础，算法简单明了，是数据压缩中的一个重要方法，有着广泛的应用。

在数据压缩领域中，算法需要结合应用场景的垂直化趋势也越来越明显，在选择或评价压缩算法时一定要结合实际加以考虑。由此，本文将给出利用哈夫曼编码对 ASCII 码文件进行压缩的 C++程序实现，并对该算法优劣进行初步分析对比。

方法介绍

文件压缩的基本思想是对字符设计长度不等的编码方案，对出现次数多的字符用尽可能短的编码表示，这样文件的总长度就会降低，实现文件压缩。比如有字符串“ABACADA”，4 个字符需要两个比特编码。假设 A、B、C、D 的编码分别是 00、01、10 和 11，则整个字符串可表示成 00010010001100，总长度为 14 个比特。但如果 A、B、C、D 的编码分别是 0、10、110 和 111，则字符串总长度为 12 比特。设计长短不等的编码方案时，必须满足一个字符的编码不能是另一个字符编码的前缀，以保证解码成字符的转换过程中不发生歧义，这种编码称为前缀编码。

哈夫曼算法提出了一种编码方法，使得文本总长度最短。其基本思想是利用字符的频率作为权重，以字符作为叶结点构造一颗最优二叉树(即哈夫曼树)，使得带权路径长度

$$WPL = W_1L_1 + W_2L_2 + \cdots + W_nL_n$$

最小，其中W_i表示第 i 个字符结点的权重，L_i表示第 i 个字符结点的路径长度。

哈夫曼编码对文件进行压缩的过程大致分为五个步骤：1.打开需要压缩的文件；2.统计文本中出现字符个数及对应出现频率，将字符按频率排序；3.创建哈夫曼树并生成哈夫曼编码；4.将需要压缩的文件中的每个 ASCII 码对应的哈夫曼编码按 bit 单位输出；5.文件压缩结束。其中，步骤 2 和步骤 4 是压缩过程的关键。步骤 2 所要做的工作是得到哈夫曼数中各叶子结点字符出现的频率并根据这个频率创建哈夫曼树。步骤 4 将需要压缩的文件中的每个 ASCII 码对应的哈夫曼编码按 bit 单位输出，这里涉及到“转换”和“输出”两个关键步骤:“转换”部分大可不必去通过遍历哈夫曼树来找到每个字符对应的哈夫曼编码，可

以将每个哈夫曼码值及其对应的字符存放于结构体中，这使得查找过程变得尤为轻松。

哈夫曼压缩算法的优势如下：

1.它是无损数据压缩，即对使用后的数据进行重构，重构后的数据与原始的数据相同；

2.哈夫曼编码在数据有噪音（非规律数据，如 RLE 等）的情况下效果很好，此时大多数基于字典方式的编码器都易出现问题；

3.哈夫曼编码有一个十分重要的特性：任何一个编码绝不会是其他编码的前缀。这一特性保证了编码、译码的唯一性，使得该方法在字符出现频率不均时仍有很好的压缩效果。

实验结果

根据哈夫曼算法，首先我们对给出的 txt01.txt 文本进行压缩，得到的结果如图 1 所示。

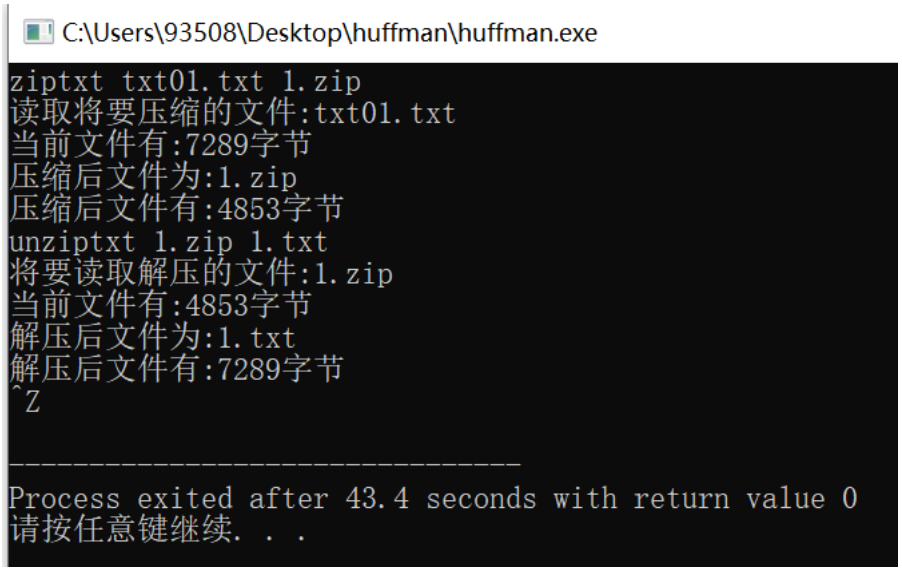

图 1

之后我们再从中 https://wiki.ubuntu.com/RandomPage 随机选取 4 个网页的文本进行压缩来检测平均压缩率，最终结果如下表 1 所示：

相关数据 / 文件名	压缩前字节数	压缩后字节数	压缩率
txt01	7289	4853	66.580%
txt02	26598	17151	64.482%
txt03	8519	5302	62.237%
txt04	15247	9735	63.849%
txt05	31758	19661	61.909%

表 1

所以平均压缩率为 63.811%

PS：在发送到助教邮箱的压缩包中，txt 编号.txt 为源文件（待压缩的文件），编号.zip 为压缩后的压缩文件，编号.txt 为解压缩后的文件。

发展到现在已经有很多关于文本压缩的算法，可总体分为无损压缩和有损压缩两种。主要有哈夫曼编码、算术编码等无损压缩和预测编码、量化、变换编码等有损压缩。

下文将针对常见的三种无损压缩：算术编码、游程编码、LZ 编码与哈夫曼编码进行比较研究。

1.算术编码

算术编码是基于统计的、无损数据压缩效率最高的方法。其与霍夫曼编码的一个很大的不同就是它跳出了分组编码的范畴，而是从全序列出发，采用递推形式的连续编码。它不是将单个的信源符号映射成一个码字，而是将整段要压缩的整个数据序列映射到一段实数半封闭范围内的某一区段。其长度等于该序列的概率，即是所有使用在该信息内的符号出现概率全部相乘后的概率值。当要被编码的信息越来越长时，用来代表该信息的区段就会越来越窄，用来表达这个区段所需的二进位就越多。

因此，相较于哈夫曼算法，它避开了用一系列特定码字代替输入符号的思想，而用一个单独的浮点数来代替一串输入符号，提高了压缩率。同时也避开了哈夫

曼编码中比特数必须取整的问题。当信源概率比较接近时，建议使用算术编码，此时哈夫曼编码的结果趋于定长码，效率不高。

但在实际应用中，算术编码比哈夫曼编码复杂，特别是硬件。高度复杂的计算量并不适用，且很难在具有固定精度的计算机完成无限精度的算术操作。此时，哈夫曼算法则避开了这些问题。

2.基于字典的 LZ 系列编码

字典算法是最为简单的压缩算法之一。它是把文本中出现频率较高的字符组合做成一个对应的字典列表，并用特殊代码来表示这个字符。基于字典的 LZ 系列编码包括：LZ77 算法、LZSS 算法、LZ78 算法、LZW 算法等几种基本算法，是通用数据压缩主流。LZ78 和 LZW 两种算法的编译码方法较为复杂，实现起来较为困难，而 LZ77 算法的压缩率又相对较低，比较而言 LZ77 算法在单片机上实现起来较为理想，其压缩率较高，编译码算法也较为简单。因此，主要论述 LZ77 算法，它的字典模型使用自适应方式，基本的思路是搜索目前待压缩串是否在以前出现过，如果出现过则利用前次出现的位置和长度来代替现在的待压缩串，输出该字符串的出现位置及长度，否则输出新的字符串，从而起到压缩的目的。

LZSS 算法最大的好处是压缩算法的细节处理不同只对压缩率和压缩时间有影响，却不会影响到解压程序。但是 LZSS 算法最大的问题是速度，每次都需要向前搜索到原文开头，对于较长的原文需要的时间是不可忍受的，且分解码段较为复杂。此时哈夫曼算法中将哈夫曼编码也就是 8bit 分割更为简洁快速。

3.游程编码

游程编码是针对一些文本数据的特点所设计的，这种算法通过统计待压缩数据中的重复字符、去除文本中的冗余字符或字节中的冗余位，从而达到减少数据文件所占的存储空间的目的。压缩处理的流程类似于空白压缩，区别仅在于要在压缩指示字符之后加上一个字符，用于表明压缩对象。随后是该字符的重复次数。由于该算法是针对文件的某些特点所设计的，所以应用起来具有一定的局限性，哈夫曼算法更为通用。

但对某些性质的信息文件，因建立的二叉树过于庞大而降低了编码的效率，不能很好的利用信息文件特定部分规律性强的特点。游程编码则能压缩这种重复率高的文件，因此可以得到更高的压缩率，提高编码效率。

程序说明

程序的输入为需压缩的 txt 文件，输出为压缩得到的 zip 文件。整个程序由三个函数构成，分别为压缩函数，解压函数和主函数。通过固定文字指令（ziptxt 和 unziptxt）来运行，压缩时执行方式为命令：ziptxt <源文件名> <压缩文件名>，解压时执行方式为命令 unziptxt <压缩文件名> <解压后文件名>。

代码实现如下，具体实现过程详见注释：

```
#include<iostream>
#include<string>
#include<cstring>
using namespace std;
struct head
{
    int b;                          //字符
    long count;                     //文件中该字符出现的次数
    long parent, lch, rch;          //make a tree
    char bits[256];                 //the huffuman code
};

struct head header[512], tmp;  //节点树

//函数：compress()
//作用：读取文件内容并加以压缩
//将压缩内容写入另一个文档
int compress(const char *filename,const char *outputfile)
{
    char buf[512];
    unsigned char c;
    long i, j, m, n, f;
    long min1, pt1, flength;
    FILE *ifp, *ofp;//ifp：读指针    ofp：写指针
    int per = 10;
    ifp = fopen(filename, "rb");                    //以二进制方式打开原始文件，并只允许读操作
    if (ifp == NULL){
        cout << "打开文件失败:" << filename << endl;
```

```
        //printf("打开文件失败:%s\n",filename);
        return 0;                                   //如果打开失败，则输出错误信息
    }
    ofp = fopen(outputfile,"wb");                   //打开压缩后存储信息的文件，并只允许写操作
    if (ofp == NULL){
        cout << "打开文件失败:" << outputfile << endl;
        return 0;
    }
    flength = 0;
    while (!feof(ifp)){ //输入输出函数，检查文件是否结束，如结束，则返回非零值，否则返回 0
        fread(&c, 1, 1, ifp); // fread(buffer,size,count,fp);
        //（1）buffer：是一个指针，对 fread 来说，它是读入数据的存放地址。对 fwrite 来说，是要输出数据的地址。
        //（2）size：要读写的字节数；
        //（3）count:要进行读写多少个 size 字节的数据项；
        //（4）fp:文件型指针。
        header[c].count ++;                         //读文件，统计字符出现次数
        flength ++;                                 //记录文件的字符总数
    }
    flength --;
    header[c].count --;
    // feof 的特性
    for (i = 0; i < 512; i ++){                     //HUFFMAN 算法中初始节点的设置
        if (header[i].count != 0){ //如果该字符的数量不为 0
            header[i].b = (unsigned char) i; //则将结构体中的字符 b 赋值为字母 i
        }
        else{
            header[i].b = -1; //没出现就赋值为-1
        }
        header[i].parent = -1;//父亲初值为-1
        header[i].lch = header[i].rch = -1; //左右子树初值也为-1
    }

    for (i = 0; i < 256; i ++){                     //将节点按出现次数排序(冒泡排序)（降序）
        for (j = i + 1; j < 256; j ++){
            if (header[i].count < header[j].count){
                tmp = header[i];
```

```
                header[i] = header[j];
                header[j] = tmp;
            }
        }
    }


    for (i = 0; i < 256; i ++){                    //前面已经把出现的字符按降序排好了，所以这里需要找出排序好的那些字符中最后一个字符所在的位置
    //因为排序之后实际上是 5 4 3 2 1 0 0 0 ...0 这样的排列，0 之前的都是各个字符的出现次数
    //也就是统计出现的字符数量
        if (header[i].count == 0){
            break;
        }
    }

    n = i;
    //n 为出现的字符数量
    m = 2 * n - 1;
    //m 为 huffman 树的节点数
    for (i = n; i < m; i ++){ //构造 huffman 树，总共 n 个节点，在数组中的位置为 n 到 2n-1（正好是 n 个） 正好从 i=n 开始全是空节点（权值为 0）
        min1 = 999999999;
        for (j = 0; j < i; j ++){//找最小节点
            if (header[j].parent != -1){//找出还未加入 huffman 树（即没有父亲节点）的节点
                continue;
            }
            if (min1 > header[j].count){
                pt1 = j;
                min1 = header[j].count;
                //找出当前权值最小的节点，并保存其权值
                //pt1 保存节点位置，min1 保存权值
                continue;
            }
        }
        header[i].count = header[pt1].count;//当前空节点的权值改为最小节点的权值
        header[pt1].parent = i;//最小节点的父亲节点为 i，相当于 tn
        header[i].lch = pt1;//左子树为最小节点
        min1 = 999999999;
        for (j = 0; j < i; j ++){//找第二小节点
            if (header[j].parent != -1){
```

```
                continue;
            }
            if (min1 > header[j].count){
                pt1 = j;
                min1 = header[j].count;
                continue;
            }
        }
        header[i].count += header[pt1].count;//当前节点权值改为最小节点和次小节点的权值和
        header[i].rch = pt1;//右子树为第二小节点
        header[pt1].parent = i;//第二小节点的父亲节点也为 i
    }

    for (i = 0; i < n; i ++){                          //构造 huffman 树，设置字符的编码
        f = i;//i 只用于计数，用 f 来代替 i 的操作
        header[i].bits[0] = 0;
        while (header[f].parent != -1){//若当前节点无父亲节点
            j = f;//同理，j 来代替 f
            f = header[f].parent;
            if (header[f].lch == j){//判断 j 是左子树还是右子树
            //header[f].lch = header[header[f].parent].lch
                j = strlen(header[i].bits);
                memmove(header[i].bits + 1, header[i].bits, j + 1);
                //memmove 用于拷贝字节，如果目标区域和源区域有重叠的话，memmove 能够保证源串在被覆盖之前将重叠区域的字节拷贝到目标区域中，
                //但复制后源内容会被更改。但是当目标区域与源区域没有重叠则和 memcpy 函数功能相同。
                /*void *memmove( void* dest, const void* src, size_t co
unt );
                由 src 所指内存区域复制 count 个字节到 dest 所指内存区域。*/
                header[i].bits[0] = '0';//类似于递归
            }
            else{
                j = strlen(header[i].bits);
                memmove(header[i].bits + 1, header[i].bits, j + 1);
                header[i].bits[0] = '1';
            }
        }
    }

    //下面的就是读原文件的每一个字符，按照设置好的编码替换文件中的字符
```

```
    fseek(ifp, 0, SEEK_SET);
     //将指针定在文件起始位置
    //int fseek(FILE *stream, long int offset, int whence) 设置
流 stream 的文件位置为给定的偏移 offset，参数 offset
    //意味着从给定的 whence 位置查找的字节数。
    fseek(ofp, 8, SEEK_SET);                                    //以 8 位二进
制数为单位进行读取
    buf[0] = 0;
    f = 0;
    pt1 = 8;
    cout << "读取将要压缩的文件:" << filename << endl;
    cout << "当前文件有:" << flength << "字节" << endl;

    while (!feof(ifp)){
        c = fgetc(ifp); // 每次读取一个字符
        f ++;
        for (i = 0; i < n; i ++){//开始进行匹配
            if (c == header[i].b) break;
        }
        strcat(buf, header[i].bits);//buf 中存取当前读入字符的编码
        j = strlen(buf);//j 为编码长度
        c = 0;
        while (j >= 8){                                          //当
剩余字符数量不小于 8 个时
            for (i = 0; i < 8; i ++){                           //按照
八位二进制数转化成十进制 ASCII 码写入文件一次进行压缩
                if (buf[i] == '1'){
                    c = (c << 1) | 1;
                }
                else{
                    c = c << 1;
                }
            }
            fwrite(&c, 1, 1, ofp);// size_t fwrite(const void *ptr, siz
e_t size, size_t nmemb, FILE *stream)
            /*ptr-- 这是指向要被写入的元素数组的指针。
            size-- 这是要被写入的每个元素的大小，以字节为单位。
            nmemb-- 这是元素的个数，每个元素的大小为 size 字节。
            stream-- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个输出
流。*/
            pt1 ++;
            strcpy(buf, buf + 8);//相当于 buf 指针右移 8 位，便于存储下一个读
入字符的 8 位编码
            j = strlen(buf);
```

```
        }
        if (f == flength){//文件中所有字符读完后跳出
            break;
        }
    }

    if (j > 0){                                                    //当
剩余字符数量少于 8 个时
        strcat(buf, "00000000");                                   //对
不足的位数进行补零
        for (i = 0; i < 8; i ++){
            if (buf[i] == '1'){
                c = (c << 1) | 1;
            }
            else{
                c = c << 1;
            }
        }
        fwrite(&c, 1, 1, ofp);
        pt1 ++;
    }
    fseek(ofp, 0, SEEK_SET);                                       //将
编码信息写入存储文件
    fwrite(&flength,1,sizeof(flength),ofp);//写入信息总长度（所有信息的编
码总长度）
    fwrite(&pt1, sizeof(long), 1, ofp);// 写入文件末尾，即文件终止位置的指
针
    fseek(ofp, pt1, SEEK_SET);
    fwrite(&n, sizeof(long), 1, ofp);//字符个数
    for (i = 0; i < n; i ++){
        tmp = header[i];

        fwrite(&(header[i].b), 1, 1, ofp);//写入字符
        pt1++;
        c = strlen(header[i].bits);
        fwrite(&c, 1, 1, ofp);//写入编码长度
        pt1++;
        j = strlen(header[i].bits);

        if (j % 8 != 0){                                           //当
位数不满 8 时，对该数进行补零操作
            for (f = j % 8; f < 8; f ++){
                strcat(header[i].bits, "0");
            }
```

```
        }

        while (header[i].bits[0] != 0){
            c = 0;
            for (j = 0; j < 8; j ++){
                if (header[i].bits[j] == '1'){
                    c = (c << 1) | 1;
                }
                else{
                    c = c << 1;
                }
            }
            strcpy(header[i].bits, header[i].bits + 8);
            fwrite(&c, 1, 1, ofp);
        //将所得的编码信息写入文件
            pt1++;
        }
        header[i] = tmp;
    }
    fclose(ifp);
    fclose(ofp);
       //关闭文件
    cout<<"压缩后文件为:"<<outputfile<<endl;
    cout<<"压缩后文件有:"<<pt1+4<<"字节"<<endl;
    return 1;                                          //返回压缩成功信息
}



//函数：uncompress()
//作用：解压缩文件，并将解压后的内容写入新文件
int uncompress(const char *filename,const char *outputfile){
    char buf[255], bx[255];
    unsigned char c;
    char out_filename[512];
    long i, j, m, n, f, p, l;
    long flength;
    int per = 10;
    int len = 0;
    FILE *ifp, *ofp;
    char c_name[512] = {0};
    ifp = fopen(filename, "rb");
       //打开文件
    if (ifp == NULL){
```

```
        return 0;      //若打开失败，则输出错误信息
    }

                                                        //读取原文件长
    if(outputfile){
        strcpy(out_filename,outputfile);
    }
    else{
        strcpy(out_filename,c_name);
    }

    ofp = fopen(out_filename, "wb");
         //打开文件
    if (ofp == NULL){
        return 0;
    }

    fseek(ifp,0,SEEK_END);
    len = ftell(ifp);
    fseek(ifp,0,SEEK_SET);
    cout << "将要读取解压的文件:" << filename << endl;
    cout << "当前文件有:" << len << "字节" << endl;

    fread(&flength, sizeof(long), 1, ifp);
       //读取原文件长
    fread(&f, sizeof(long), 1, ifp);
    fseek(ifp, f, SEEK_SET);
    fread(&n, sizeof(long), 1, ifp);
       //读取原文件各参数
    for (i = 0; i < n; i ++){
        //读取压缩文件内容并转换成二进制码
        fread(&header[i].b, 1, 1, ifp);
       //读取字符
        fread(&c, 1, 1, ifp);
       //读取字符对应编码长度
        p = (long) c;
        header[i].count = p;
       //将编码长度存储到 header[i].count
        header[i].bits[0] = 0;
        if (p % 8 > 0){
            m = p / 8 + 1;
       //m 为编码所占字节数
        }
        else{
```

```
            m = p / 8;
        }
        for (j = 0; j < m; j ++){
      //按字节读取内容
            fread(&c, 1 , 1 , ifp);
        //读取编码对应的 ASCLL 码
            f = c;
            _itoa(f, buf, 2);
        //ASCLL 码转化为二进制字符串，存储到 buf 中
            f = strlen(buf);
        //f 为字符串长度
            for (l = 8; l > f; l --){
                strcat(header[i].bits, "0");
        //位数不足，执行补零操作，在每个字节最前面补 0
            }
            strcat(header[i].bits, buf);
        //将该位添加到 header[i].bits 末尾
        }
        header[i].bits[p] = 0;
    }

    for (i = 0; i < n; i ++){
        //对 header[].bits 的位数进行排序，位数更小的出现频率更高，下标更小利
于遍历
        for (j = i + 1; j < n; j ++){
            if (strlen(header[i].bits) > strlen(header[j].bits)){
                tmp = header[i];
                header[i] = header[j];
                header[j] = tmp;
            }
        }
    }

    p = strlen(header[n-
1].bits);                                          //p 为 bits 编码的最
大位数
    fseek(ifp, 8, SEEK_SET);
    m = 0;
       //已解压的字符个数
    bx[0] = 0;


    while (1){
```

```
        while (strlen(bx) < (unsigned int)p){
        //读取至少 p 位
            fread(&c, 1, 1, ifp);
            f = c;
            _itoa(f, buf, 2);
        //整数转换为字符串
            f = strlen(buf);
            for (l = 8; l > f; l --){
                strcat(bx, "0");
        //位数不足，执行补零操作，在每个字节最前面补 0
            }
            strcat(bx, buf);
        //将该位添加到 bx 末尾
        }
        for (i = 0; i < n; i ++){
            if (memcmp(header[i].bits, bx, header[i].count) == 0){
        //将 bx 的前 header[i].count 位与 header[i].bits 比较，如果相同就 break
                break;
            }
        }
        strcpy(bx, bx + header[i].count);
        //bx 后移 header[i].count 位
        c = header[i].b;
        //找到编码对应的字符
        fwrite(&c, 1, 1, ofp);
        //写入 c
        m ++;
        //已解压的字符个数++
        if (m == flength){
            break;
        }
    }
    fclose(ifp);
    fclose(ofp);
    cout << "解压后文件为:" << out_filename << endl;
    cout << "解压后文件有:" << flength <<"字节" << endl;
    return 1;                    //输出成功信息
}

int main()
{
    memset(&header,0,sizeof(header));
    memset(&tmp,0,sizeof(tmp));
    char *a = "ziptxt";
```

```
    char *b = "unziptxt";
    char filenameToCompress[200];//需要压缩的文件名
    char compressFilename[200];//压缩文件命名
    char filenameToUncompress[200];//需要解压的文件名
    char uncompressFilename[200];//解压文件命名
    char order[200];//指令
    while(cin >> order){
        if(strcmp(order, a) == 0){
            cin >> filenameToCompress >> compressFilename;
            compress(filenameToCompress, compressFilename);
        }
        if(strcmp(order, b) == 0){
            cin >> filenameToUncompress >> uncompressFilename;
            uncompress(filenameToUncompress, uncompressFilename);
        }
    }
    return 0;
}
```

参考文献

[1]包冬梅.数据压缩算法研究[J].无线互联科技,2019,16(21):112-113.

[2]张红军,徐超.基于改进哈夫曼编码的据压缩方法研究[J].唐山师范学院学报,2014,36(05):40-43.

[3]程佳佳,熊志斌.哈夫曼算法在数据压缩中的应用[J].电脑编程技巧与维护,2013(02):35-37.

[4]路炜,门玉梅,李建俊.用哈夫曼编码对文件进行压缩的 C 语言实现[J].福建电脑,2012,28(01):48-49.

[5]蔡茂蓉,姜龙,丁光辉,杨文辉.哈夫曼树的实现及其在文件压缩中的应用[J].现代计算机(专业版),2008(11):99-102.

[6]徐成俊,舒毅,柴蓉,张其斌,田全红,郝涛.文本压缩算法的比较研究[J].甘肃科技,2006(12):81-83.

[7]李晓明.数据压缩问题[J].中国信息技术教育,2020(19):22-26.

[8]余兴阁.无损数据压缩与解压算法的介绍与实现[J].信息与电脑(理论版),2016(01):64-65.

具体分工说明

张航悦（19335262）：在本次大作业中，我主要负责查找有关英文文本压缩方法的文献与实现方法，并撰写实践报告。通过查阅文献，我首先了解到了多种英文文本压缩方法的原理与特性，丰富了自己的知识。其次，在大一下离散

数学的学习过程中，虽初步接触了哈夫曼算法，但并未有很深的记忆，特别是对哈夫曼编码的实际应用没有什么了解。通过此次自己参与到文本压缩算法的学习实践中，做到了能够自己编写代码实现哈夫曼算法，并了解到了哈夫曼算法在文本压缩中的作用，巩固了自己的知识。

许群（19335236）：本次实践中我主要负责查找算法原理资料及撰写摘要、引言及部分方法介绍，从而对实践报告进行完善。通过前期查找资料我对数据压缩技术的研究背景及意义、数据压缩技术种类、不同压缩算法间性能上的优劣对比、压缩算法的选择与实际应用场景的关系等等有了初步理解。同时课程内对哈夫曼一般只是给出哈夫曼树的定义和编码，偏向基础性的运用，但本次通过对压缩算法的学习，我对概率统计模型在压缩中的应用有了更深的体会，由此对算法理论有了深入理解。

郝裕玮（18329015）：本次实践中，我负责学习前两位同学查找到的参考文献，并根据所学知识来编写代码，且主要负责代码中的主函数和压缩函数部分。通过此次实践，我对哈夫曼算法有了更深刻的认识，并且对其原理也有了更深一步的理解。且初步学会了文件的读写以及对各种与文件相关的函数的使用。同时，我也从参考文献中学习到了其他的几种压缩算法，明白了这几种算法之间的逻辑关系和优劣比较。最后，在此次实践中我也提高了自己的编程能力，虽然只是个小项目，但是最后能够成功运行，对不同大小的文本文件保持稳定的压缩率，还是让我充满了成就感。

张闯（18325071）：本次实践中，我负责学习前两位同学查找到的参考文献，并根据所学知识来编写代码，且主要负责代码中的解压缩函数部分。虽然在大一离散和大二数据结构的课程中我均学习了哈夫曼算法的原理和具体的计算方法，但我对于哈夫曼算法的理解只停留在理论部分。也可能是自己的学习热情不够，没有在课余时间主动去编程复现这一算法。但借着此次实践的机会，我成功和郝裕玮同学一起合作完成了该项目的代码编写。并且在学习了张航悦和许群同学提供的参考文献后，我也初步掌握了其他几种压缩算法的原理和实现方法，希望自己在未来的学习中能够抽出时间来实现这几种算法，并以此来增强自己的编程能力。