Huffman 编码压缩/解压器

一、实验要求

1. 基本要求

基于 Huffman 编码实现一个压缩器和解压缩器，其中 Huffman 编码以字节作为统计和编码的基本符号单元，使其可以对任意的文件进行压缩和解压缩操作。针对编译生成的程序，要求压缩和解压缩部分可以分别独立运行。

- 每次运行程序时，用户可以指定**只压缩\只解压缩**指定路径的文件。采用命令行参数指定功能和输入/输出的文件路径。

2. 扩展

- 支持凹入表打印 Huffman 树

二、设计思路

1. 核心思路

- 在压缩阶段，需要完成 4 个任务：
  - 通过字符型读取原文件，统计每个字符的频数
  - 建立 Huffman 树
  - 对原文件进行压缩
  - 将包括 Huffman 树在内的附加信息写入到输出文件中，再将压缩后的信息写入输出文件
- 在解压缩阶段，需要完成2个任务
  - 读取压缩好的文件，对其进行拆分，将附加部分与压缩部分进行拆分
  - 根据附加信息建立 Huffman 树
  - 通过二进制读取压缩文件，处理，写入到新文件中

2. 使用到的数据结构及其 ADT

本程序中使用了树和栈，其 ADT 分别 为：

ADT Tree{

数据对象：D = { a_i | a_i ∈ ElemSet，$i = 1, 2..., n, n \geq 0$ }

数据关系：若 D 为空集，则称为空树；

若 D 仅含一个数据元素，则 R 为空集，否则 R={H}，H 是如下二元关系：

(1) 在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；

(2) 若 D-{root}≠Φ，则存在 D-{root}的一个划分 $D_1, D_2, \cdots, D_m$ (m>0) （D_i 表示构成第 i 棵子树的结点集），对任意 j≠k (1≤j, k≤m) 有 $D_j \cap D_k = \Phi$，且对任意的 i (1≤i≤m)，唯一存在数据元素 $x_i \in D_i$，有 $<root, x_i> \in H$（H 表示结点之间的父子关系）；

(3) 对应于 D-{root}的划分，$H-\{<root, x_1>, \cdots, <root, x_m>\}$有唯一的一个划分 $H_1, H_2, \cdots, H_m(m>0)$（$H_i$ 表示第 i 棵子树中的父子关系），对任意 j≠k(1≤j,k≤m)有 $H_j \cap H_k = \Phi$，且对任意 i(1≤i≤m)，H_i 是 D_i 上的二元关系，$(D_i, \{H_i\})$是一棵符合本定义的树，称为根 root 的子树。

基本操作：(本次实验使用到的)

InitTree(&T); // 创造空树

InsertChild(&T,p,i,c); // 插入孩子节点

TraverseTree(T,visit()); // 遍历树

}ADT Tree

ADT Stack{

数据对象：D = { a_i | $a_i \in$ ElemSet，$i = 1, 2..., n, n \geq 0$ }

数据关系：R = {$<a_{i-1}, a_i>$ | $a_{i-1}, a_i \in D, i = 1, 2, 3...n$}

基本操作：（本次实验使用到的）

InitStack(&S); // 构造一个空栈

GetTop(S,&e); // 用 e 返回栈顶元素

Push(&S,e); // 插入元素 e 作为新的栈顶元素

Pop(&S,&e); // 删除 S 的栈顶元素，并用 e 返回其值

StackTraverse(S,visit()); // 遍历栈

}ADT Stack

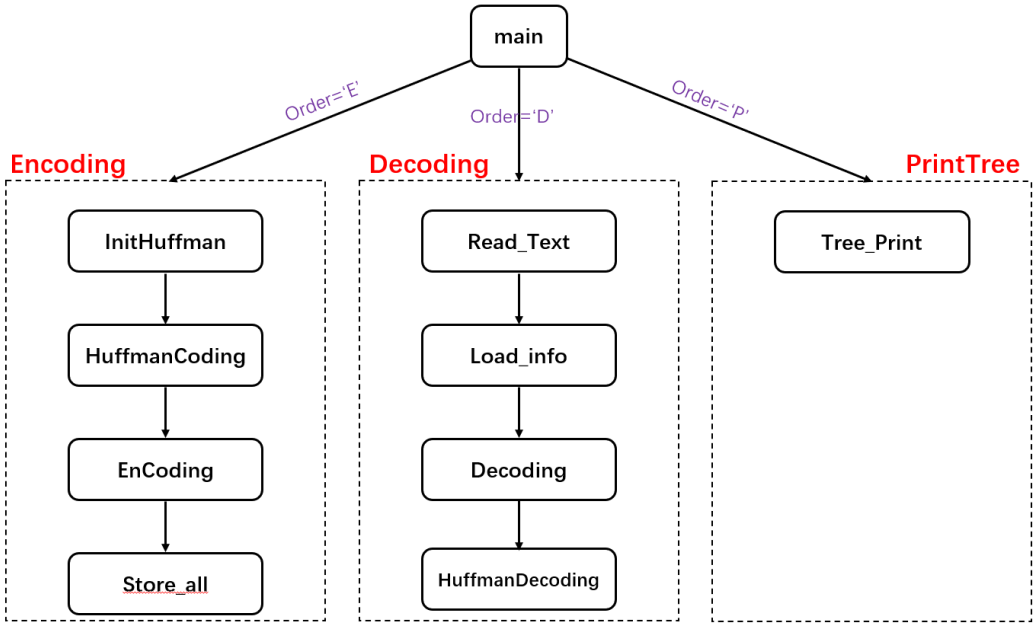
---

3. 核心模块

```
Cal_weight(char** argv);
    // 从文件读入字符，计算每个字符的频数
InitHuffman(HuffTree& HT);
    // 构建 Huffman 树
HuffmanCoding(HuffTree HT, int root, char stack[], int flag, int top);
    // 对每个字符求其 Huffman 编码
```

```
EnCoding(char** argv);
    // 读取要压缩的文件，求出每个字节的 Huffman 编码，输出到一个临时存储文件
Store_all(HuffTree HT);
    // 将 Huffman 树输出到压缩文件，再将临时存储文件信息输出到压缩文件
Read_Text(HuffTree& HT);
    // 读入压缩文件，将附加信息与压缩信息分离，根据附加信息建立 Huffman 树
DeCoding(HuffTree HT, char** argv);
    // 对于分理出的压缩信息，进行解压缩
Tree_Print(HuffTree HT, int root, int cnt);
    // 以凹入表形式打印 Huffman 树
```

4. 模块流程关系



三、关键代码讲解

1. InitHuffman(HuffTree& HT)

首先，由于 Huffman 没有度数为 1 的结点，所以根据叶子结点个数（即从文件读取到的字符个数）就能推出 Huffman 树的结点数。

直接创建顺序存储结构的 Huffman 树，每个结点包含的信息为：

```
typedef struct {
    int weight;
    int parent, lchild, rchild;
}HTNode;
```

然后不断选取当前没有双亲结点的结点中，权值最小的两个节点，进行合并。代码为：

```
void InitHuffman(HuffTree& HT)
```

```
{
    int m = num_char * 2 - 1;//树的节点个数
    HT = new HTNode[m + 1];
    for (int i = 1; i <= m; i++) {
        HT[i].weight = (i <= num_char ? w_char[i] : 0);
        HT[i].lchild = HT[i].rchild = HT[i].parent = 0;
    }
    for (int i = num_char + 1; i <= m; i++) {
        int s1, s2;
        Selete(HT, i, s1, s2);//在1...i-1中选取父节点为0，并且权值最小的两个结点
        HT[i].lchild = s1;
        HT[i].rchild = s2;
        HT[i].weight = HT[s1].weight + HT[s2].weight;
        HT[s1].parent = HT[s2].parent = i;
    }
}
```

2. HuffmanCoding(HuffTree HT, int root, char stack[], int flag, int top)

求每个结点的 Huffman 编码。调用栈，沿每一条从树根到叶结点的路径，向左孩子走就是 0，向右孩子走就是 1.

```
void HuffmanCoding(HuffTree HT, int root, char stack[], int flag, int top)
{//用栈来记录01序列，flag==0,向左儿子走，flag==1,向右儿子走
    if (flag == 0) {
        stack[top] = '0';
        top++;
    }
    if (flag == 1) {
        stack[top] = '1';
        top++;
    }
    if (HT[root].lchild == 0) {//当前节点为叶子节点
        for (int i = 0; i < top; i++)
            HC[root][i] = stack[i];
        HC[root][top] = '\0';
        top--;
        return;
    }
    HuffmanCoding(HT, HT[root].lchild, stack, 0, top);
    HuffmanCoding(HT, HT[root].rchild, stack, 1, top);
}
```

3. EnCoding(char** argv)

根据来自命令行的文件路径和名称，打开源文件，以二进制形式逐字节读取，计算对应的 Huffman 编码，以二进制形式存储到临时文件中

若要写入的 Huffman 编码的 bit 数不是字节的整数倍，则在末尾补零，并记录补零的个数

```
void EnCoding(char** argv)
{
    ifstream inFile(argv[1], ios::in | ios::binary);
```

```
    ofstream outFile("CodeFile1.txt", ios::out | ios::binary);//单纯存储Huffman编码后的文件
    char zero_one[10];//临时存储01编码序列
    int cnt_01 = 0;
    unsigned char a = 0;
    unsigned char tmp;
    while (inFile.read((char*)&tmp, sizeof(tmp))) {
        for (int i = 1; i <= num_char; i++) {
            if (tmp == Char[i]) {
                for (int j = 0; HC[i][j] != '\0'; j++) {
                    zero_one[cnt_01] = HC[i][j];
                    cnt_01++;
                    if (cnt_01 == 8) {//达到一个字节，可以存入
                        Cal_unsigned(zero_one, a);
                        outFile.write((char*)&a, sizeof(a));
                        a = 0;
                        cnt_01 = 0;
                        num_8++;
                    }
                }
            }
        }
    }
    if (cnt_01) {//不够一个字节，末尾补零
        num_0 = 8 - cnt_01;
        for (int i = 0; i < cnt_01; i++)
        {
            a = a * 2;
            if (zero_one[i] == '1') a++;
        }
        for (int i = 0; i < num_0; i++)
            a = a * 2;
        outFile.write((char*)&a, sizeof(a));
    }
    inFile.close();
    outFile.close();
}
```

4. Store_all(HuffTree HT)

将 Huffman 树作为附加信息存储到输出文件的头部，再将压缩信息存储到输出文件

不同信息之间加入断点，方便后续读取时进行分离

```
void Store_all(HuffTree HT)
{
    FILE* fp1;
    fp1 = fopen("CodeFile.txt", "w");
    fprintf(fp1, "%d %d ", num_8, num_0);
    fprintf(fp1, "%d ", num_char);
    for (int i = 1; i <= 2 * num_char - 1; i++)
    {
        fprintf(fp1, "%d %d %d %d ", HT[i].weight, HT[i].parent, HT[i].lchild, HT[i].rchild);
    }
    char tmpp = EOF;
    fprintf(fp1, "%c", tmpp);//前后间断点
```

```
    fclose(fp1);

    fp1 = fopen("CodeFile.txt", "ab");
    for (int i = 1; i <= num_char; i++) fprintf(fp1, "%c", Char[i]);

    ifstream inFile("CodeFile1.txt", ios::in | ios::binary);
    unsigned char tmp;
    while (inFile.read((char*)&tmp, sizeof(tmp))) {
        fprintf(fp1, "%c", tmp);
    }
    inFile.close();
    fclose(fp1);
}
```

5. Read_Text(HuffTree& HT)

读取压缩信息，首先进行分离，分离附加信息和文件压缩后的信息，再根据附加信息建立 Huffman 树

```
void Read_Text(HuffTree& HT)
{
    FILE* fp1, * fp2;
    fp1 = fopen("Pre_Info1.txt", "wb");
    ifstream inFile("CodeFile.txt", ios::in | ios::binary);
    unsigned char tmp;
    while (inFile.read((char*)&tmp, sizeof(tmp))) {
        if ((int)tmp == 255) break;
        fprintf(fp1, "%c", tmp);
    }
    fclose(fp1);

    fp1 = fopen("Pre_Info1.txt", "r");
    fscanf(fp1, "%d%d%d", &num_8, &num_0, &num_char);
    HT = new HTNode[2 * num_char];
    for (int i = 1; i <= 2 * num_char - 1; i++) {
        fscanf(fp1, "%d%d%d%d", &HT[i].weight, &HT[i].parent, &HT[i].lchild,
&HT[i].rchild);
    }
    fclose(fp1);

    fp2 = fopen("Pre_Info2.txt", "wb");
    for (int i = 1; i <= num_char; i++) {
        inFile.read((char*)&tmp, sizeof(tmp));
        fprintf(fp2, "%c", tmp);
    }
    fclose(fp2);

    ofstream outFile("CodeFile2.txt", ios::out | ios::binary);//单纯存储Huffman编码
后的文件
    while (inFile.read((char*)&tmp, sizeof(tmp))) {
        outFile.write((char*)&tmp, sizeof(tmp));
    }
    inFile.close();
    outFile.close();
}
```

6. Load_Info(HuffTree& HT)

更新字符记录数组

```
void Load_Info(HuffTree& HT)
{
    ifstream inFile("Pre_Info2.txt", ios::in | ios::binary);
    unsigned char tmp;
    int cnt_char = 1;
    while (inFile.read((char*)&tmp, sizeof(tmp))) {
        Char[cnt_char] = (unsigned char)tmp;
        cnt_char++;
    }
    inFile.close();
}
```

7. DeCoding(HuffTree HT, char** argv)

根据分离出的压缩文件信息，首先进行转化，将一个字节数据转化为对应的字符串，写入临时文件。之后不断调用 HuffmanDeCoding 函数，对临时文件中的字符串数据进行求原字符，实现解压缩功能。

```
void DeCoding(HuffTree HT, char** argv)
{
    FILE* fp, * fp2;
    fp = fopen("CodeFile2.txt", "rb");
    fp2 = fopen("TempTxt.txt", "w");
    unsigned char p = fgetc(fp);
    while (num_8) {
        int cnt = 128;
        int a = (int)p;
        while (cnt) {
            if (a >= cnt) {
                a = a - cnt;
                fputc('1', fp2);
            }
            else {
                fputc('0', fp2);
            }
            cnt = cnt / 2;
        }
        p = fgetc(fp);
        num_8--;
    }
    if (num_0) {
        int cnt = 128;
        int tmp = 8;
        int a = (int)p;
        while (tmp > num_0) {
            if (a >= cnt) {
                a = a - cnt;
                fputc('1', fp2);
            }
            else {
                fputc('0', fp2);
            }
            cnt = cnt / 2;
```

```
            tmp--;
        }
    }
    fclose(fp);
    fclose(fp2);
    FILE* fp1;
    fp1 = fopen("TempTxt.txt", "r");
    fp2 = fopen(argv[2], "wb");
    while (HuffmanDeCoding(fp1, fp2, HT, 2 * num_char - 1));
    fclose(fp1);
    fclose(fp2);
}
```

8. HuffmanDeCoding(FILE* fp1, FILE* fp2, HuffTree HT, int root)

递归调用，对每个 Huffman 编码求解对应的原字符

```
bool HuffmanDeCoding(FILE* fp1, FILE* fp2, HuffTree HT, int root)
{
    char p;
    if (!fp1 || !fp2) {
        cout << "error";
        return 0;
    }
    if (HT[root].lchild == 0) {
        fprintf(fp2, "%c", Char[root]);
        return 1;
    }
    else {
        p = fgetc(fp1);
        if (p == EOF) return 0;
    }
    if (p == '0') HuffmanDeCoding(fp1, fp2, HT, HT[root].lchild);
    if (p == '1') HuffmanDeCoding(fp1, fp2, HT, HT[root].rchild);
    return 1;
}
```

8. Tree_Print(HuffTree HT, int root, int cnt)

通过不断递归调用，实现凹入表打印 Huffman 树

```
void Tree_Print(HuffTree HT, int root, int cnt)
{
    for (int i = 1; i <= cnt; i++)
        cout << "     ";
    if (root <= num_char) cout << (int)Char[root];
    else cout << root;
    cout << endl;
    if (HT[root].lchild == 0) return;
    Tree_Print(HT, HT[root].lchild, cnt + 1);
    Tree_Print(HT, HT[root].rchild, cnt + 1);
    return;
}
```

四、调试分析

1.时间复杂度

- 读文件的时间复杂度是 O(N)，其中 N 为原文件的字符数目（即总字节数）
- 建立 Huffman 树，核心过程是不断将当前最小权值的两个节点合并，共需合并 $(n-1)$ 次，其中 n 为叶子节点个数（即不同字符的个数），每次合并需要寻找最小权值的两个结点，这个的时间复杂度是 O(n)，所以总的时间复杂度为 O(n^2). 但是考虑到在单字节压缩时，n 最大值为 255，远小于一般的文件大小，所以建立 Huffman 树的时间对整个程序的运行时间影响非常小
- HuffmanCoDing 操作是在求每个字符对应的 huffman 编码，需要遍历字符数组，对每个字符，需要寻找在 Huffman 树上从根节点到叶子结点的路径，所以总的时间复杂度为 O(nlogn)
- 求原文件中每个字节的 Huffman 编码，首先是遍历源文件，在读到每一个字节时，遍历字符数组，求其对应的 Huffman 编码，再写入文件。所以时间复杂度为 O(N*n)，N 为源文件的字符数目，n 为 Huffman 树的叶子结点个数。
- 解码的过程首先是读取压缩数据，将其转化为 01字符串，再根据 01 字符串逐次遍历 Huffman 树，求解其原字符。时间复杂度为 O(N*logn)，N 为压缩文件的字节数，n 为 Huffman 树的叶子节点个数，logn 是遍历一条从根节点到叶子结点的时间。但是注意到，解码过程涉及到了临时文件的处理，临时文件将原来的一个字节转化为长度为 8 字符串，所以时间复杂度的常数会比较大

2. 空间复杂度

- 由于大部分操作都在对文件进行处理，而且需要存储的大量数据直接存到文件中，所以程序内部需要的空间较少，最大的空时间是用就是在存储每个字节对应的 Huffman 编码，空间复杂度为 O(nlogn)，n 为不同的字节数目
- 压缩文件的大小小于等于原文件，如果原文件是文本类，那么压缩性能较好，而如果原文件是音频类，那么压缩文件并不比原文件小很多，或者几乎与原文件大小相同

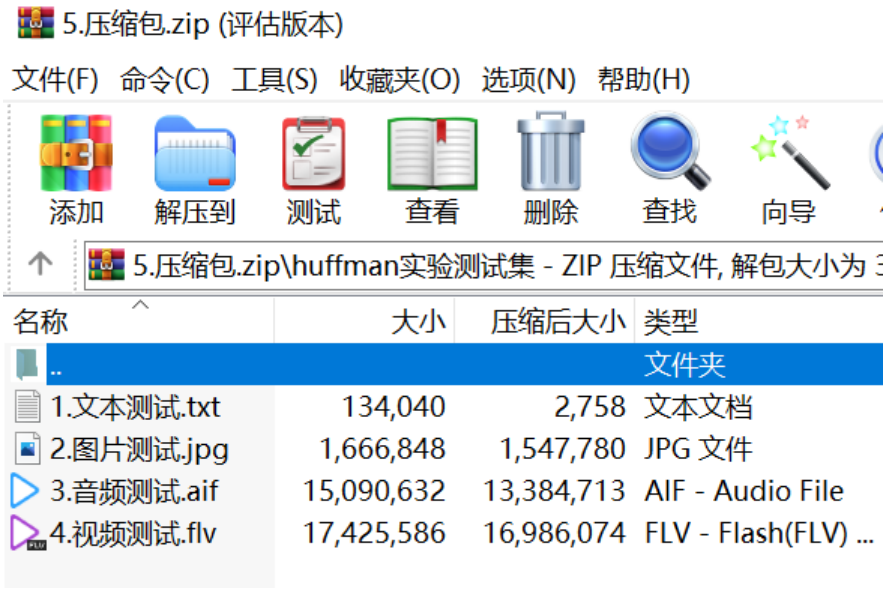
3. 遇到的 BUG 以及处理方法

- 出现了一直在读文件不停止的现象。原因在于需要以二进制形式读入
- 由于在写之前将整个大的任务进行了拆分，在写每个小任务时不断调试分析，所以并没有其他较大的 BUG 出现。

五、代码测试

- 首先进行压缩的测试：命令行输入指令，**第一个文件为要读入的原文件，第二个文件为输出文件。**测试的文件为一个压缩文件，里面包含了文本文件、图片、音频文件、视频文件。

  测试文件如下图：



执行情况如下图：

```
PS C:\Users\86198\Desktop\数据结构\Huffman\Finished\Huffman_final\Debug>
.\Huffman_final.exe .\5.压缩包.zip out.zip
========HuffmanCoding========
=====E:对文件编码============
=====D:对文件解码============
=====P:打印Huffman树=========
=====Q:退出编码系统==========
请输入指令: E
编码已完成!
```

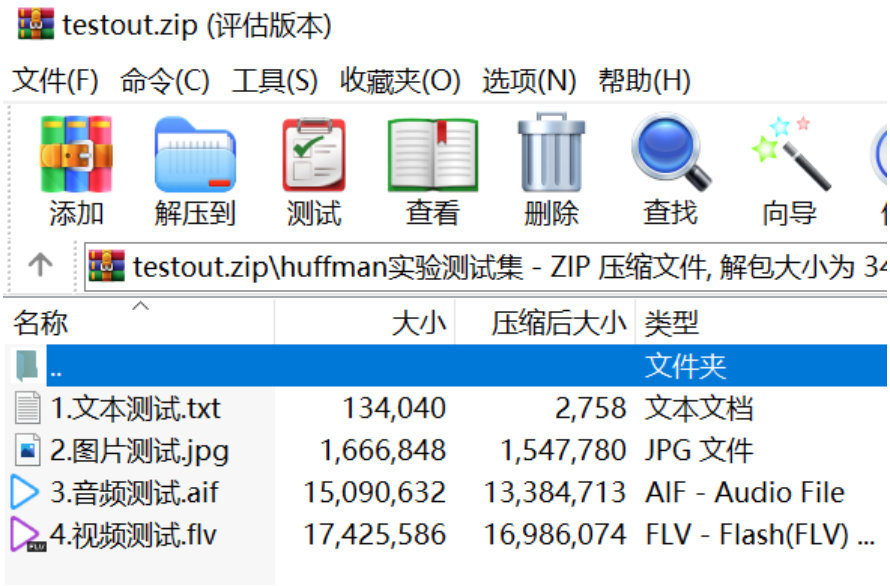
获得了压缩文件 out.zip

- 再将压缩文件进行解压缩，输出文件名为 testout.zip。命令中，**第一个文件为要读入的压缩文件文件，第二个文件为输出文件**

  执行情况如下图：

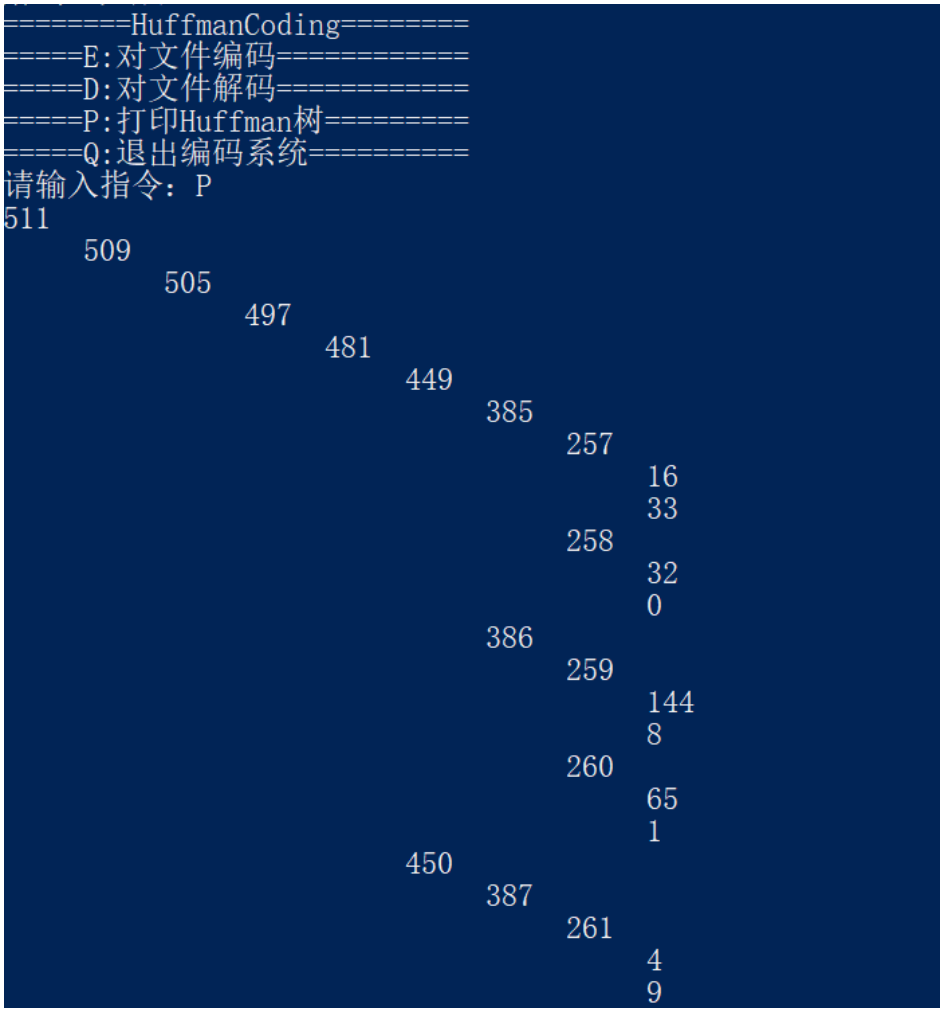
```
PS C:\Users\86198\Desktop\数据结构\Huffman\Finished\Huffman_final\Debug>
 .\Huffman_final.exe out.zip testout.zip
========HuffmanCoding========
=====E:对文件编码============
=====D:对文件解码============
=====P:打印Huffman树=========
=====Q:退出编码系统==========
请输入指令: D
解码已完成!
```

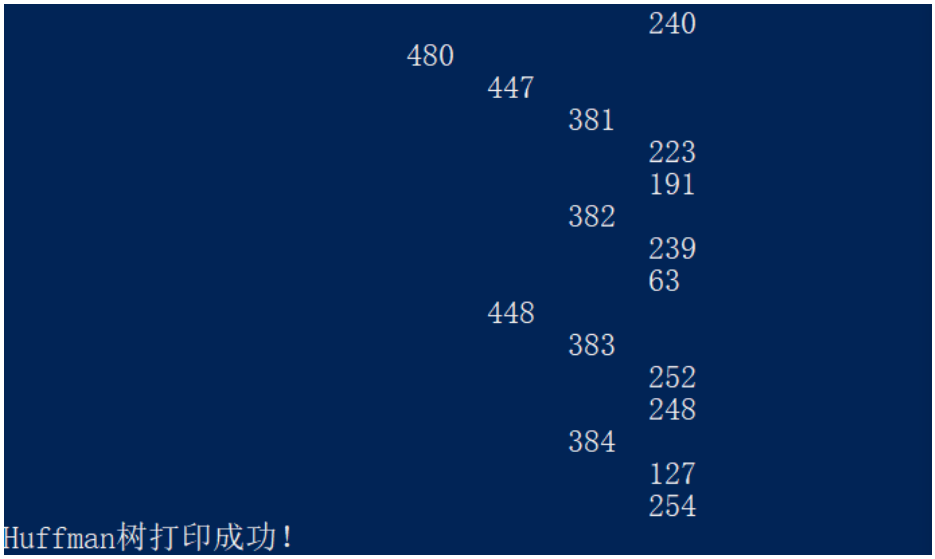
  解压后的文件如下图：



将压缩包解压后，文件均可正常打开

- 进行 Huffman 树的打印测试：（Huffman 树较大，仅截图一部分）





六、实验总结

1. 本次实验，最大的挑战在于文件的读入和输出，具体挑战有三个：
    - 如何读取命令行指令，然后根据命令行输入的文件路径和名称，对相应文件进行操作
    - 如何正确读入和输出二进制文件
    - 如何将附加信息存储到压缩文件中，解压缩时如何分离和处理压缩文件的不同信息

    所以通过本次实验，我得到的最多的练习就是有关文件的操作。
2. 通过建立和遍历 Huffman 树，我对于树和栈有了更深一步的理解
3. 单比特的 Huffman 压缩效率并不高，对于纯文本文件的压缩还可以，但对于音频和视频的压缩效果就不理想，主要原因是音频视频的比特几乎使用到了全部的情况。

七、附录

Huffman_final.cpp