西 安 邮 电 大 学

(计算机学院)

数据结构课程设计报告

题　目：哈夫曼编/译码器

专业名称：计算机科学与技术

班　　级：计科 2203

学生姓名：路亮

学号（8 位）：04221096

指导教师：闫青丽

设计起止时间：2023 年 12 月 25 日—2023 年 12 月 29 日

一. 设计目的

利用哈夫曼编码进行信息通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码，在接收端将传来的数据进行译码（复原）。试为这样的信息收发站写一个哈夫曼的编/译码器。

二. 设计内容

建立哈夫曼树：读入文件(*.souce)，统计文件中字符出现的频度，并以这些字符的频度作为权值，建立哈夫曼树。

编码：利用已建立好的哈夫曼树，获得各个字符的哈夫曼编码，并对正文进行编码，然后输出编码结果，并存入文件(*.code)中。

译码：利用已建立好的哈夫曼树将文件(*.code)中的代码进行译码，并输出译码结果，并存入文件(*.decode)中。

利用位操作，实现文件的压缩与解压。

三. 概要设计

1. 功能模块图

```
Course_Design/
├── DecodeOnHaffmanCode/
│   ├── decode.h
│   └── decode.c
├── EncodeOnHaffmanTree/
│   ├── encode.h
│   └── encode.c
├── Helps/
│   └── help.h
│   └── help.c
├── HuffmanTreeBuild/
│   ├── build.c
│   └── build.h
├── Test/
│   ├── 1.pdf
│   ├── 2.pptx
│   ├── 3.png
│   ├── 4.mp3
│   ├── 5.mp4
│   └── 6.txt
├── main.c
├── script.sh
├── README.md
└── Makefile
```

图 1 项目目录

图 2 功能模块图

2. 各个模块详细的功能描述

1）哈夫曼编码

哈夫曼编码模块，它的功能是统计一个文件中的字节 Byte 类型以及各个类型占的比例，这样就能得到哈夫曼树中的结点的权值。得到权值之后，根据权值进行排序，然后利用优先队列构建哈夫曼树。

构建好哈夫曼树之后，利用深度优先搜索遍历所有的叶子结点，得到每一个叶子结点的哈夫曼编码。

2）文件压缩

这里以二进制读取和写入文件。得到哈夫曼编码之后，就可以对整个文件进行压缩了。首先需要向 *.code 文件中写入魔数（magic number），我设计的这个魔数的值为 0xdeabeef，这样可以标记 *.code 文件是一个压缩文件（加密文件）。

之后需要向这个文件中写入哈夫曼编码，哈夫曼编码位于一个数组之中，要将其依次写入到 *.code 文件中，这里是先写入哈夫曼编码的大小，然后再写入哈夫曼编码。然后还要得到被压缩文件的文件大小信息，再将其写入到 *.code 文件中。

接着开始对文件进行编码并将编码写入到 *.code 文件之中。二进制读取被压缩的文件 *.source，拿到字节 A 之后，在哈夫曼编码中搜索字节 A 的哈夫曼编码。搜到哈夫曼编码后将其放到一个缓冲区 buffer 中，等到其凑够一个字节之后，写入到 *.code 文件之中。

可能会存在一种状况，那就是最后一个哈夫曼编码不能构成一个字节，这时候，需要就

计算出需要补的比特的数量，这个数量小于等于 7，这里的设计是补比特 0。

3）文件解压

这里以二进制读取文件。首先检查魔数，检查待解压的文件是否是经过哈夫曼编码的文件。接着读出哈夫曼编码的大小，然后循环读出哈夫曼编码信息，得到哈夫曼编码数组，类似于数据结构的反序列化。

拿到哈夫曼编码后，就要开始解密文件了。先读出文件的大小，然后设置一个缓冲区，将读出来的字节 A 存到缓冲区中。由于哈夫曼编码是前缀码，因此只要从这个字节缓冲区的最高位按位读取一个二进制串 001010，再将其转化成数组，这个数组就是字符串类型的二进制串。

利用这个字符串在哈夫曼编码中搜索这个字符串的值，这个值就是字节 A，然后将其写入到文件 *.decode 中。写完后，立即清空字符串数组，往里面继续填充二进制数据，重复搜索、写入过程，一直到被写入文件的大小为压缩文件中写入的文件的大小为止。然后终止整个写入过程，关闭文件，完成文件的解压。

4）帮助函数

帮助函数主要是实现一些信息打印、输出的函数，帮助用户学习使用这个命令行工具。

四．详细设计

1．功能函数的调用关系图

1）哈夫曼编码

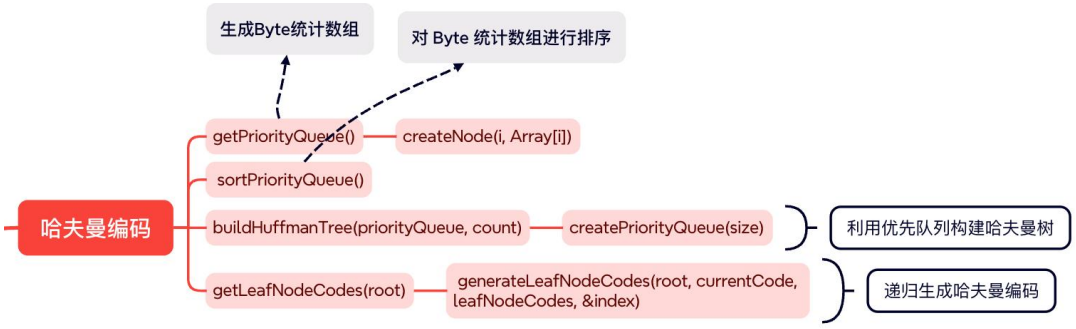

图 3 哈夫曼编码函数调用图

2）文件压缩

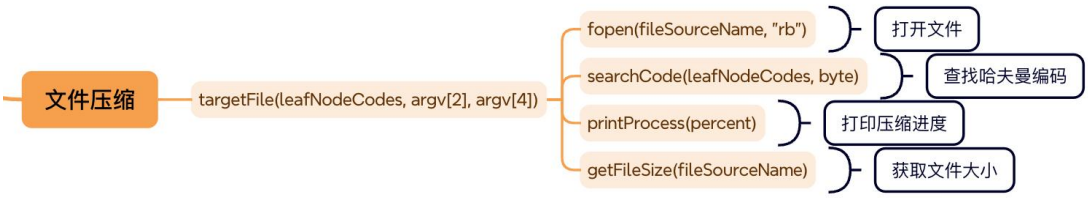

图 4 文件压缩调用图

3）文件解压

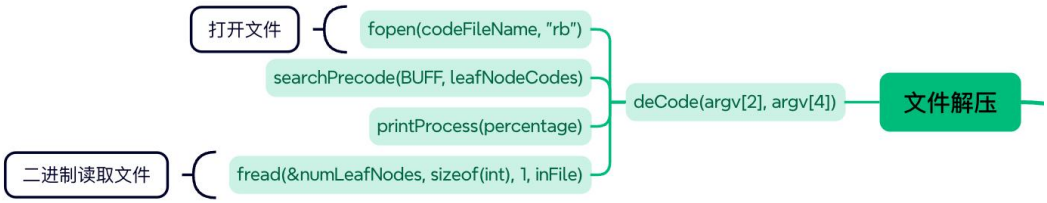

图 4 文件解压函数调用图

2. 各功能函数的数据流程图

1）哈夫曼编码

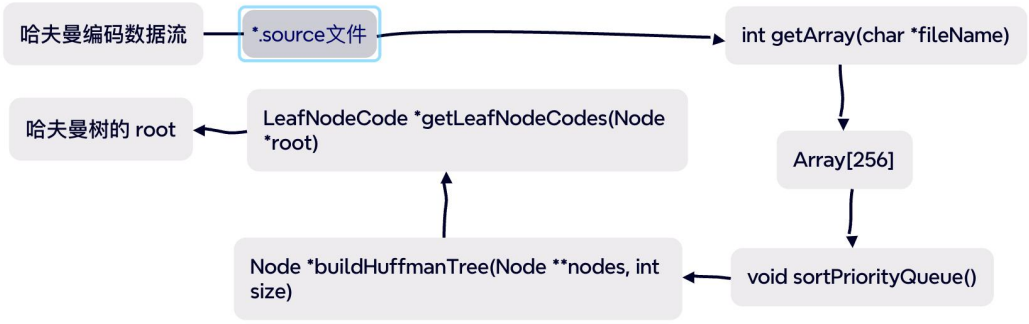

图 5 哈夫曼编码数据流程图

2）文件压缩

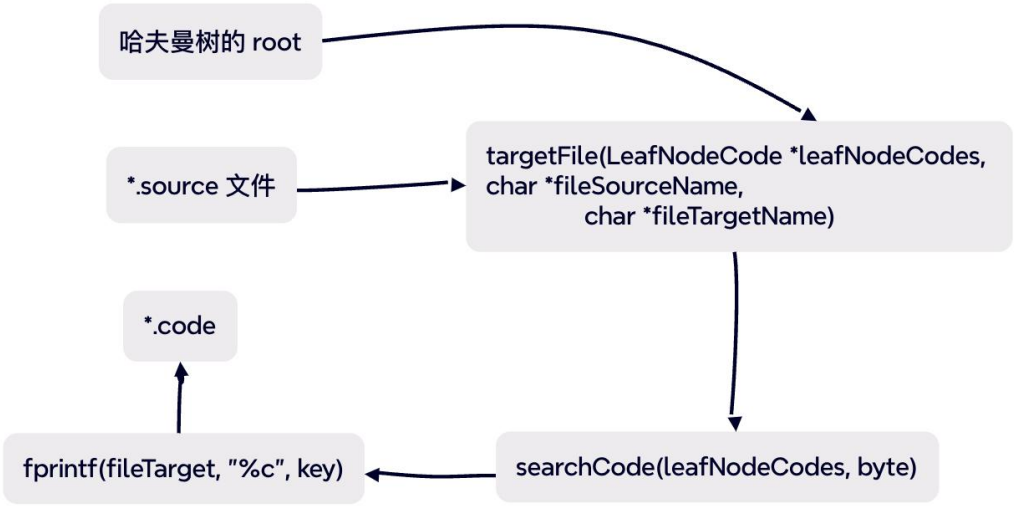

图 6 文件压缩数据流程图

3）文件解压

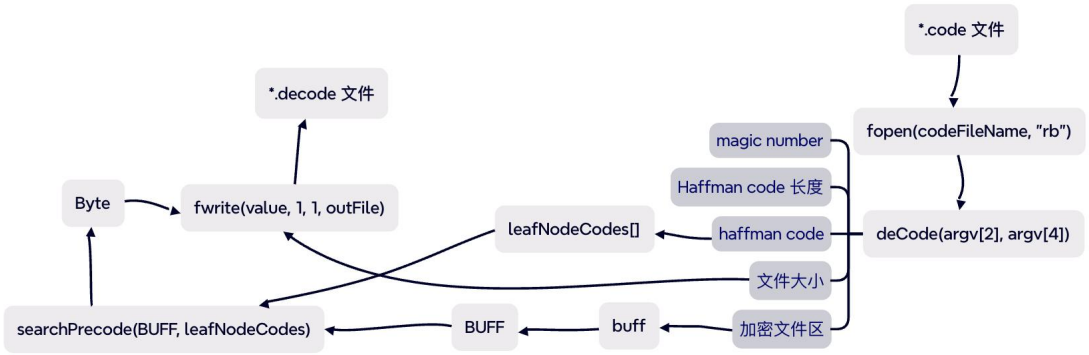

图 7 文件解压数据流程图

3. 重点设计及编码

1）哈夫曼编码

这里的重点设计就是哈夫曼树的构建了，哈夫曼编码就要依赖对此树进行递归遍历（先根遍历）。构建哈夫曼树要用到一个优先队列的数据结构，这个数据结构有两个函数：Node *pop(PriorityQueue *queue)、void insert(PriorityQueue *queue, Node *node) 。前者会每次会从优先队列中弹出一个权值最小的结点，后者会在这个队列中插入一个结点，插入后队列依然会保持有序。

```
// insert node to the priorityQueue
void insert(PriorityQueue *queue, Node *node) {
    int i = queue->size;
    while (i > 0 && node->freq < queue->nodes[(i - 1) / 2]->freq) {
        Queue->nodes[i] = queue->nodes[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    queue->nodes[i] = node;
    queue->size++;
}

// pop the smallest freq node
```

```
Node *pop(PriorityQueue *queue) {
    if (queue->size <= 0) {
        return NULL;
    }
    Node *minNode = queue->nodes[0];
    queue->size--;

    Node *lastNode = queue->nodes[queue->size];
    int i = 0;
    while (i * 2 + 1 < queue->size) {
        int child = i * 2 + 1;
        if (child + 1 < queue->size &&
        queue->nodes[child + 1]->freq < queue->nodes[child]->freq) {
        child++;
    }
    if (lastNode->freq <= queue->nodes[child]->freq) {
        break;
    }
    queue->nodes[i] = queue->nodes[child];
        i = child;
    }
    queue->nodes[i] = lastNode;

    return minNode;
}
```

以上两个个函数就是*pop(PriorityQueue *queue)、void insert(PriorityQueue *queue, Node *node)的实现。接下来就要实现哈夫曼树的构建了，构建的思路就是不断地 pop()两个结点，权值相加之后，得到一个母结点，然后将母结点的左、右指针分别指向这两个结点。再将其insert()到优先队列中。当优先队列剩一个结点时，那么这个结点就是哈夫曼树的根节点，然后将其 pop()之后，返回。我们就得到了哈夫曼树：

```
// create Haffman tree
Node *buildHuffmanTree(Node **nodes, int size) {
    if (size <= 0) {
```

```
        return NULL;
    }

// with a PriorityQueue
    PriorityQueue *queue = createPriorityQueue(size);
    for (int i = 0; i < size; i++) {
        insert(queue, nodes[i]);
    }

// tree building...
    while (queue->size > 1) {
        Node *left = pop(queue);
        Node *right = pop(queue);
        Node *parent = createNode('$', left->freq + right->freq);
        parent->left = left;
        parent->right = right;
        insert(queue, parent);
    }

    Node *root = pop(queue);
    free(queue->nodes);
    free(queue);

    return root;
}
```

以上 Node *buildHuffmanTree(Node **nodes, int size)的实现，接下来就是遍历哈夫曼树构建哈夫曼叶子结点的编码了：

```
// generate Haffman code
void generateLeafNodeCodes(Node *node, char *currentCode,
    LeafNodeCode *leafNodeCodes, int *index) {
    if (node == NULL) {
        return;
    }
```

```
        // go left +0
        char leftCode[256];
        strcpy(leftCode, currentCode);
        strcat(leftCode, "0");
        generateLeafNodeCodes(node->left, leftCode, leafNodeCodes, index);

        // go right +1
        char rightCode[256];
        strcpy(rightCode, currentCode);
        strcat(rightCode, "1");
        generateLeafNodeCodes(node->right, rightCode, leafNodeCodes, index);

        if (node->left == NULL && node->right == NULL) {
            // find the leaf node, save the complete code
            leafNodeCodes[*index].data = node->data;
            strcpy(leafNodeCodes[*index].code, currentCode);
            (*index)++;
        }
}
```

我们将这些编码存储起来，就得到了哈夫曼树叶子结点的编码数组：

```
// create LeafNodeCode[]
LeafNodeCode *getLeafNodeCodes(Node *root) {
        numLeafNodes = count;

        LeafNodeCode *leafNodeCodes =
        (LeafNodeCode *)malloc(numLeafNodes * sizeof(LeafNodeCode));

        int index = 0;
        char currentCode[256] = "";

        // generate leaf code
        generateLeafNodeCodes(root, currentCode, leafNodeCodes, &index);
```

```
    return leafNodeCodes;
}
```

至此，我们完成了哈夫曼编码，并且将字节 Byte 的哈夫曼编码放到了一个数组中，接着就可以利用它对文件进行加密（压缩了）。

2）文件压缩

这里的重点设计就是，如何编码，并且能确保正确解压。首先现将压缩文件特殊标志魔数（magic number）写入文件，这是文件最开头的 4 个字节，它标识了生成的文件是利用哈夫曼编码压缩的文件。接着写入哈夫曼编码数组的大小信息，这指明了哈夫曼编码数组边界，之后就是对源文件的编码了。

对于编码，由于文件系统操作是以字节 Byte 为单位的，因此必须要确保每次写入的都是一个字节。换句话说，拿到字节 A 的编码信息后，要将其转化成二进制 bits，然后将二进制 bits 凑成一个字节后，再写入到文件。

这里棘手的点是，必须设置一个缓冲区，将哈夫曼编码的第一位，放到缓冲区的最低位，然后通过左移，不断让缓冲区从最低位向最高位增长，当期增长到 8 时，就立即利用掩码，拿到最后一个字节，然后将其写入到文件。之后将缓冲区清空，并且复位。

```
while ((byte = fgetc(fileSource)) != EOF) {
    char *haffman_code = searchCode(leafNodeCodes, byte);
    lenOfHaffmanCode = strlen(haffman_code);

    for (int i = 0; i < lenOfHaffmanCode; i++) {
        if (haffman_code[i] == '1') {
        buffer ^= (uint32_t)1;
    } else {
        buffer ^= (uint32_t)0;
    }
    lengthOfBUffer++;

    if (lengthOfBUffer == 8) {

        // get the lowest Byte
```

```
            // write the Byte to the file
            unsigned char key = buffer & MASKCODE;
            fprintf(fileTarget, "%c", key);
            lengthOfBUffer -= 8;
            // reset
            buffer = 0;
            // show the process
            double percent = (Nwrite++) / (sizeOfFile * 1.0);
            printProcess(percent);
        }
        buffer <<= 1;
        }
}
```

所以有一个小问题，那就是当最后一个二进制 bits 构不成一个字节时，需要将其补成一个字节，这里的设计是补 0。以下为整个压缩加密设计的代码：

```
// zip
int targetFile(LeafNodeCode *leafNodeCodes, char *fileSourceName,
char *fileTargetName) {
        FILE *fileSource = fopen(fileSourceName, "rb");
        FILE *fileTarget = fopen(fileTargetName, "wb");
        if (fileSource == NULL || fileTarget == NULL) {
            perror("zip failed!\n");
            return -1;
        }
        // info
        printf("file: %s is zipping now!\n", fileSourceName);

        // magic number
        char magicNumber[] = {0xde, 0xad, 0xbe, 0xef}; // magic number: 0xdeadbeef
        fwrite(magicNumber, sizeof(char), sizeof(magicNumber), fileTarget);
```

```
// Haffman Code
fwrite(&numLeafNodes, sizeof(int), 1, fileTarget);
for (int i = 0; i < numLeafNodes; ++i) {
        fwrite(&leafNodeCodes[i], sizeof(LeafNodeCode), 1, fileTarget);
}

// the size of the file
long sizeOfFile = getFileSize(fileSourceName);
fwrite(&sizeOfFile, sizeof(long), 1, fileTarget);

// file code
long Nwrite = 0; // how many Bytes already been written in the file
uint32_t buffer = 0; // buffer for 4 Bytes
int lengthOfBUffer = 0; // length of buffer
int byte; // Byte from the source file
int lenOfHaffmanCode = 0;
while ((byte = fgetc(fileSource)) != EOF) {
        char *haffman_code = searchCode(leafNodeCodes, byte);
        lenOfHaffmanCode = strlen(haffman_code);

        for (int i = 0; i < lenOfHaffmanCode; i++) {
                if (haffman_code[i] == '1') {
                        buffer ^= (uint32_t)1;
                } else {
                        buffer ^= (uint32_t)0;
                }
                lengthOfBUffer++;

                if (lengthOfBUffer == 8) {

                        // get the lowest Byte
                        // write the Byte to the file
                        unsigned char key = buffer & MASKCODE;
                        fprintf(fileTarget, "%c", key);
```

```
                    lengthOfBUffer -= 8;

                    // show the process
                    double percent = (Nwrite++) / (sizeOfFile * 1.0);
                    printProcess(percent);
                }
                buffer <<= 1;
            }
        }
        // check
        if (lengthOfBUffer == 0) {
        unsigned char key = ZERO;
        fprintf(fileTarget, "%c", key);
        } else {
            // recover
            buffer >>= 1;
            int numsOfBit0 = 8 - lengthOfBUffer;
            buffer <<= numsOfBit0;

            // write last code
            unsigned char key = buffer & MASKCODE;
            fprintf(fileTarget, "%c", key);
        }
        fclose(fileSource);
        fclose(fileTarget);
        printf("file: %s zipped successed!\n", fileSourceName);

        // print info:
        long sizeOfSource = getFileSize(fileSourceName);
        long sizeOfTarget = getFileSize(fileTargetName);

        printf("the rate of the zip is %.5f\n", sizeOfTarget / (sizeOfSource * 1.0));
        return 0;
}
```

3）文件解压

文件解压的过程和压缩加密过程刚好相反。首先要检查魔数（magic number），看其是否是可以被解压的文件。然后读出 Haffman 数组的大小，接着一直将哈夫曼编码读完，类似于反序列化过程。之后读取原文件的大小，通过这个大小可以和已经写入到目标文件的字节数作对比，一旦写入的字节数和文件大小相等时，直接退出，完成解密。

这里的难点就是仍然需要设置一个缓冲区，不过这个缓冲区是用来将读出来的字节，从高位拿到 bit，将其放到 buff 字符数组下标最低处。

```
while (1) {
    n = fread(&buffer, sizeof(buffer), 1,inFile); // load 1 Byte once a time
    if (n < 1) {
        // end of the file
        break;
    }
    for (i = 0; i < 8; i++) {
        result = buffer & (1 << (7 - i));
        if (result == 0) {
            BUFF[len++] = '0';
        } else {
            BUFF[len++] = '1';
        }
        value = searchPrecode(BUFF, leafNodeCodes);
        if (value == NULL) {
            continue;
        } else {
            // write in
            if (fwrite(value, 1, 1, outFile) <= 0) {
                perror("unzip error");
                return;
            } else {
                // recover
                Nwrite++;
                memset(BUFF, '\0', 16);
```

```
                len = 0;
            }
        }
        double percentage = Nwrite / (sizeOfFile * 1.0);
        printProcess(percentage);

        if (Nwrite == sizeOfFile) {
            goto finished;
        }
    }
}
```

由于哈夫曼编码是前缀码，因此可以在当缓冲器每增加一个长度时，同时搜索哈夫曼编码对应的字节值，如果找到了将其写入文件，如果没有找到继续增长缓冲区。以下为整个解码过程：

```
// unzip
void deCode(char *codeFileName, char *decodeFileName) {
    FILE *inFile = fopen(codeFileName, "rb");
    FILE *outFile = fopen(decodeFileName, "wb");
    if (inFile == NULL || outFile == NULL) {
        perror("File open error");
        return;
    }

    // check magic number
    uint32_t magicNo = 0;
    fread(&magicNo, sizeof(uint32_t), 1,inFile);
    // Read magic number at the begining of the file first
    if (ntohl(magicNo) != 0xdeadbeef) {
        printf("the decoding file is not a zipped file.\n");
        return;
    }

    // Haffman Code
    int numLeafNodes = 0;
```

```
fread(&numLeafNodes, sizeof(int), 1, inFile); // Read number of nodes first

LeafNodeCode *leafNodeCodes =
(LeafNodeCode *)malloc(numLeafNodes * sizeof(LeafNodeCode));
if (leafNodeCodes == NULL) {
        perror("Memory allocation error");
        fclose(inFile);
        return;
}

for (int i = 0; i < numLeafNodes; ++i) {
        fread(&leafNodeCodes[i], sizeof(LeafNodeCode), 1, inFile);
}

long sizeOfFile = 0; // the size of file
fread(&sizeOfFile, sizeof(sizeOfFile), 1, inFile);

// coded file
uint8_t buffer = 0; // buffer for 1 Bytes
char BUFF[16] = {0}; // Code cache
int len = 0; // the length of BUFF
char *value = 0; // HaffmanCode to Byte
int result = 0; // bit
int Nwrite = 0; // already written in the file
int n = 0; // nums of fread()
int i = 0; // a pointer

printf("file: %s is unzipping!\n",
codeFileName); // some info of the unzipping file

while (1) {
        n = fread(&buffer, sizeof(buffer), 1,inFile); // load 1 Byte once a time
        if (n < 1) {
                // end of the file
```

```
            break;
        }
        for (i = 0; i < 8; i++) {
            result = buffer & (1 << (7 - i));
            if (result == 0) {
                BUFF[len++] = '0';
            } else {
                BUFF[len++] = '1';
            }
            value = searchPrecode(BUFF, leafNodeCodes);
            if (value == NULL) {
                continue;
            } else {
                // write in
                if (fwrite(value, 1, 1, outFile) <= 0) {
                    perror("unzip error");
                    return;
                } else {
                    // recover
                    Nwrite++;
                    memset(BUFF, '\0', 16);
                    len = 0;
                }
            }
            double percentage = Nwrite / (sizeOfFile * 1.0);
            printProcess(percentage);

            if (Nwrite == sizeOfFile) {
                goto finished;
            }
        }
    }

    fclose(inFile);
```

```
    fclose(outFile);

finished:
    printf("file: %s unzip successed!\n", codeFileName);
    fclose(inFile);
    fclose(outFile);
}
```

五. 测试数据及运行结果

1. 正常测试数据和运行结果

本项目采用的脚本自动测试化测试，利用 shell 脚本，对 5 组数据进行了测试，测试结果如下：

```
    file: ./Test/1.pdf is zipping now!
    file: ./Test/1.pdf zipped successed!================================        ] 92.30%
    the rate of the zip is 0.99944
    file: ./Test/2.pptx is zipping now!
    file: ./Test/2.pptx zipped successed!                                       ] 0.00%
    the rate of the zip is 1.01594
    file: ./Test/3.png is zipping now!
    file: ./Test/3.png zipped successed!                                        ] 0.00%
    the rate of the zip is 1.00278
    file: ./Test/4.mp3 is zipping now!
    file: ./Test/4.mp3 zipped successed!                                        ] 0.00%
    the rate of the zip is 0.99846
    file: ./Test/5.mp4 is zipping now!
    file: ./Test/5.mp4 zipped successed!================================        ]
94.03%
    the rate of the zip is 0.99829
    file: ./Test/6.txt is zipping now!
    file: ./Test/6.txt zipped successed!                                        ] 0.00%
    the rate of the zip is 0.60989
```

file: ./Test/zipfile1.code is unzipping!

file: ./Test/zipfile1.code unzip successed!==========] 92.83%

file: ./Test/zipfile2.code is unzipping!

file: ./Test/zipfile2.code unzip successed!] 29.04%

file: ./Test/zipfile3.code is unzipping!

file: ./Test/zipfile3.code unzip successed!==========] 89.76%

file: ./Test/zipfile4.code is unzipping!

file: ./Test/zipfile4.code unzip successed!==========] 95.39%

file: ./Test/zipfile5.code is unzipping!

file: ./Test/zipfile5.code unzip successed!==========] 99.71%

file: ./Test/zipfile6.code is unzipping!

file: ./Test/zipfile6.code unzip successed!==========] 91.03%

测试完后，对文件进行考察：

******@shenjian Test % ls -sl

total 174320

```
 2944 -rw-r--r--  1 ******  staff     1504136 Dec 27 20:04 1.pdf
 4224 -rw-r--r--  1 ******  staff     1504136 Dec 27 21:05 11decode.pdf
  416 -rw-r--r--  1 ******  staff      212411 Dec 27 20:04 2.pptx
  512 -rw-r--r--  1 ******  staff      212411 Dec 27 21:05 22decode.pptx
 2976 -rw-r--r--  1 ******  staff     1520556 Dec 27 20:04 3.png
 4224 -rw-r--r--  1 ******  staff     1520556 Dec 27 21:05 33decode.png
 2808 -rw-r--r--  1 ******  staff     1434980 Dec 27 20:04 4.mp3
 4224 -rw-r--r--  1 ******  staff     1434980 Dec 27 21:05 44decode.mp3
40984 -rw-r--r--  1 ******  staff    20982444 Dec 27 20:04 5.mp4
41088 -rw-r--r--  1 ******  staff    20982444 Dec 27 21:07 55decode.mp4
 5272 -rw-r--r--  1 ******  staff     2696142 Dec 27 20:04 6.txt
 6152 -rw-r--r--  1 ******  staff     2696142 Dec 27 21:08 66decode.txt
 4224 -rw-r--r--  1 ******  staff     1503296 Dec 27 21:05 zipfile1.code
  512 -rw-r--r--  1 ******  staff      215797 Dec 27 21:05 zipfile2.code
 4224 -rw-r--r--  1 ******  staff     1524788 Dec 27 21:05 zipfile3.code
 4224 -rw-r--r--  1 ******  staff     1432764 Dec 27 21:05 zipfile4.code
41088 -rw-r--r--  1 ******  staff    20946471 Dec 27 21:05 zipfile5.code
 4224 -rw-r--r--  1 ******  staff     1644344 Dec 27 21:05 zipfile6.code
```

可以看到，一共测试了 6 个文件，包括 pdf、pptx、png、mp3、mp4、txt 文件，其中 txt 文件的压缩效率最好，可以达到 60%，而其它文件由于本身就是压缩文件，所以哈夫曼编码的压缩算法就行不通了，因为这些字节的权值相当接近，导致哈夫曼编码几乎都是一个字节长，因此基本上没有可压缩空间。

2. 异常测试数据及运行结果

本压缩解压程序可以满足所有文件的压缩和解压，没有任何异常测试可以提供。

六. 调试情况，设计技巧及体会

1. 改进方案

打印出帮助信息，可使用命令：haff -h，它会输出帮助信息：

```
HAFF(1)                          User Commands                          HAFF(1)

NAME
       haff - compress or decompress files using Huffman coding

SYNOPSIS
       haff [-h] [-z source_file_name -o compressed_file_name]
       haff [-h] [-u compressed_file_name -o decompressed_file_name]

DESCRIPTION
       haff is a command-line tool to compress or decompress files using Huffman coding.

OPTIONS
       -h      Print this help message
       -z      Compress the source file
       -u      Decompress the compressed file
       -o      Specify output file name
```

EXAMPLES

Compress:

haff -z source_file_name -o compressed_file_name

Decompress:

haff -u compressed_file_name -o decompressed_file_name

SEE ALSO

haff(2), haff(3)

压缩文件可以使用命令：haff -z *.source -o *.code

解压文件可以使用命令：haff -u *.code -o *.decode

由于压缩和加压缩都是计算密集型，因此可以考虑采用并行计算，提高效率，因此改进的思路就是将一个大文件分割成多个临时文件，然后对其加密，加密完成后，可以等待其它压缩线程，等其它完成后再拼接成一个完整的压缩文件。对于解压，由于哈夫曼编码是一个前缀码，如果分割的不合适，那么将会得到错误的目标文件。因此在压缩的时候，必须向文件的分割处写入特定的标记字节，以便解压分割文件时识别。

由于现在的 CPU 都是多核处理器，因此采用并行计算，可以大大提高解压和压缩效率。另外，还可以再哈夫曼编码的时候，采用两个字节一对而不是一个字节构成一个编码对象，这样有望提高压缩率以及压缩速度。

2. 体会

这个哈夫曼编/译码器的设计是一个复杂而充实的挑战，实现了数据压缩与解压的基本功能。在这个过程中，我学到了很多关于算法设计、文件处理以及系统优化的知识和技巧。这里是一些详细的心得体会：

算法设计的重要性：

实现哈夫曼编码和解码的核心是建立哈夫曼树和进行前缀编码。深入理解和掌握哈夫曼树的构建过程对于正确生成有效编码是至关重要的。在设计过程中，对节点的处理、优先队列的应用和树的遍历等都是核心算法部分。

文件处理与数据流：

处理文件的读取和写入是这个项目的关键。对文件操作的正确性、有效性以及错误处理是保证程序稳健性的关键所在。考虑到大文件的处理，逐步读取与写入可以有效避免内存爆炸。同时，了解文件的二进制结构，掌握二进制数据和字符数据的转换也是必备技能。

错误处理与异常情况：

考虑到各种可能的异常情况是非常关键的。文件不存在、权限问题、数据损坏等各种可能的错误都需要有相应的处理方式，比如合适的错误提示、安全的文件操作等。

算法效率与优化：

哈夫曼编码的特性让其对数据压缩非常高效。然而，对于某些文件类型，由于其本身的特点，哈夫曼编码并不能带来太大的压缩比。这也提醒了我在算法选择和优化上需要充分考虑应用场景和数据特性。

用户体验与命令行工具：

设计一个易于使用的命令行工具对于用户体验至关重要。提供清晰的帮助信息、合理的命令参数解析以及友好的交互提示都可以提高工具的易用性。

团队协作与版本管理：

如果是团队协作的项目，使用版本管理工具（如 Git）是非常重要的。合理的分工合作、代码审查和注释文档编写都能够提高项目的开发效率和质量。

这个项目不仅仅是对算法知识的应用，更是对系统设计和软件工程的综合考验。通过这个过程，我深刻体会到了解决问题的方法和流程，也更加熟悉了如何处理复杂项目中的挑战。

七．参考文献

1. 王曙燕（主编）. (2019). 数据结构与算法. 北京：高等教育出版社. (ISBN 978-7-04-052437-6)

2. 严蔚敏，吴伟民. (1997). 数据结构：C 语言版本. 北京：清华大学出版社. (ISBN 978-7-302-02368-5)