《实验十一：哈夫曼压缩、解压缩算法（编译码器）》

（一）问题描述

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，要求在发送端通过一个编码系统对传输数据预先编码（压缩）；在接收端将传来的数据进行译码（解压缩复原）。试为这样的通信站编写一个哈夫曼编译码系统---哈夫曼压缩/解压缩算法。

基本要求：

1）通信内容可以是任意的多媒体文件；

2）自己设定字符大小，统计该文件中不同字符的种类（字符集、个数）、出现频率（在该文件中）；

3）构建相应的哈夫曼树，并给出各个字符的哈夫曼编码；

4）对原文件进行哈夫曼压缩编码形成新的压缩后文件（包括哈夫曼树）；

5）编写解压缩算法对压缩后文件进行解码还原成原文件。

（二）解决思路

利用哈夫曼编码对文件进行压缩与解压，首先要扫描原文件，统计每类字符的频率（即出现的次数），然后根据字符频率建立哈夫曼树，接着根据哈夫曼树生成哈夫曼编码。再次扫描原文件，每次读取 8bits，根据“字符——编码”表，匹配编码，并将编码存入压缩文件，同时存入各个字符的频率。解压时，先读取字符频率，重新建立哈夫曼树，生成哈夫曼编码，然后读取压缩文件的编码，匹配编码表找到对应字符，存入文件，完成解压。

（三）数据结构

1、哈夫曼树：即带权路径长度 WPL 最小的二叉树。利用哈夫曼树生成的哈夫曼编码，既满足前缀编码的条件，译码时不会产生二义性，又能保证文件的总编码长度最短。

2、堆：满足下列条件的一棵完全二叉树——对于任意一个非终结点，其关键值≥（或≤）其任意一个子节点的关键值。建立哈夫曼树的过程中，每次都要从森林中取根结点权值最小的两棵树，作为左右子树构造一棵新的二叉树，故可用小顶堆对森林进行排序。

（四）算法分析

①文件读写：为了能够处理任何格式的文件，采用二进制方式进行读写。以一个无符号字符（unsigned char）的长度 8 位作为处理单元，最多有 256（0-255）种组合，即 256 类字符。（之所以选择无符号类型，是因为在统计字符频率时，可以把字符作为数组下标，从而能够对数组元素进行随机访问）。

②哈夫曼树的建立：要建立哈夫曼树，首先需要知道文件中每一个不同字符的出现频率。将

一个 8bits 的数据作为一个字符编码（共有 256 种字符），并建立一个叶结点数组（leaves），数组下标对应不同的字符编码，这样可以实现随机访问，提高效率。每读取一个 8bits 的数据，就将对应数组元素的权重加一。文件读取完毕后，将每个叶结点的内存地址赋值给森林数组（tree），每次将森林中根结点权重最小的两棵树取出，作为左右子树构造一棵新的二叉树，其权重为左右子树根结点的权重之和，并将左子树根结点中存储的字符设置为‘0’，右子树根结点中存储的字符设置为‘1’。然后将两棵树从森林中删除，并将新的树加入森林。不断重复以上操作，直至森林中只剩下一棵树，该树即为哈夫曼树。在比较两棵树根结点的权重大小时，可以建立一个小顶堆，堆顶元素即为根结点权重最小（非零）的树，每次将其取出后，需要对小顶堆重新调整。

③哈夫曼编码：哈夫曼树建好后，依次从每一个叶结点出发直至根结点，自底而上将结点中存储的‘0’‘1’字符拷贝至 ch_temp 数组中临时存储，然后统计字符串总长度，动态分配内存，将字符串反向拷贝至分配的内存中，再把内存地址拷贝给 Code 数组（其下标对应于原始的 8bits 字符编码）。

④文件压缩：先将原文件的长度以及字符频率写入压缩文件中，然后再以二进制只读模式打开原文件，每次读取一个 8 位的无符号字符，并将其作为数组下标，访问存储在 Code 数组中的编码信息。由于编码长度不定，故需要一个编码缓存，待编码满足 8 位时才写入，文件结束时缓存中可能不足 8 位，在后面补 0，凑足 8 位写入。此外，由于编码的每一位都是以字符形式保存的，占用空间很大，不可以直接写入压缩文件，故需要转为二进制形式写入。可以利用 C 语言的位操作（与、或、移位）来实现，每匹配一位，用“或”操作存入低位，并左移一位，为下一位腾出空间，依次循环，满足 8 位就写入一次。

⑤文件解压：解压时，先从压缩文件中读取原文件长度及各字符的频率，重新建立哈夫曼树。然后从树的根结点出发，先从压缩文件中读取一个 8 位的无符号字符，由字符高位到低位依次进行判断，若为 0 则进入左子树，反之进入右子树。若判断到字符的最低位时仍未进入叶结点，则继续读取一个 8 位的无符号字符，重复上述操作，直至到达叶结点，然后将叶结点对应的编码写入解压文件，并重新由哈夫曼树的根结点出发，继续对字符的下一位进行判断，并进入相应的子树。（注意，由于压缩文件是二进制文件，无法用 EOF 来判断文件的结束，故这里采用原文件长度来控制解压文件的写入）

（五）运行结果

程序说明：

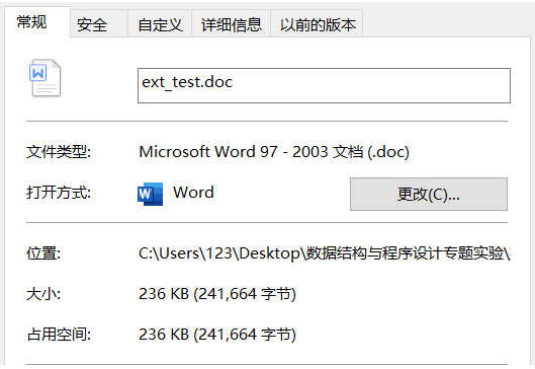
输入待处理的文件时必须输入全称（包括文件扩展名），且文件名不能超过 50 个字节的大小，另外待处理文件必须跟编译后的 exe 应用程序处于同一级目录下。压缩后的文件会在原文件名后面加上.huf 的后缀，将.huf 文件解压后，会去掉.huf 后缀，并在文件名前加上 ext_的前缀。

结果截图：

控制台操作界面

原文件

压缩文件



解压文件

结论分析：

对于纯文本文件（如 txt、doc 格式等），哈夫曼编码的压缩率接近 50%，如上面的运行结果所示，原文件的大小为 236KB，压缩后文件大小为 132KB；而对于其它文件（如 xlsx、pptx、pdf、png 等格式），哈夫曼编码的压缩率则接近于 100%，甚至比原文件还大。这是由于程序将 8bits 数据作为一个原始编码，而在非纯文本文件中，各种原始编码出现的频率大致相同，哈夫曼编码对比原始编码长度并无明显缩短，且压缩文件中还需写入文件长度及各原始编码的频率等信息，故无法对文件进行有效压缩。

（六）反思总结

通过本次实验，我对树这种数据结构有了更加深入的了解，掌握了如何建立哈夫曼树、哈夫曼编码的生成、筛选法建堆及堆的调整，还学习了文件的读取与写入操作，掌握了 fopen、fclose、fread、fwrite 等文件读写函数的使用。总之，本次实验让我收益匪浅。

（七）源程序

```
//哈夫曼压缩、解压缩算法（编译码器）
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define SIZE 256

//树结点
typedef struct tree_node {
    char code;
    unsigned long weight;
    struct tree_node *parent;
    struct tree_node *left;
    struct tree_node *right;
} node;

//函数原型
char menu(void);
bool compress(void);
bool extract(void);
void Huffman_Tree(node *leaves);
char **encoder(node *leaves);
void free_tree(node *leaves);
void SiftDown(node **tree, int n, int i);

int main(void) {
    char choice = menu();
    while (choice != 'c') {
        switch (choice) {
            case 'a':          //压缩
                compress();
                break;
            case 'b':          //解压
                extract();
                break;
            default:
                puts("Error!!!");
        }
        choice = menu();
    }
    return 0;
}
```

```
//主界面
char menu(void) {
    printf("****************************************\n\n");
    printf("     基于哈夫曼编码的文件压缩程序\n\n");
    printf("      (a)压缩          (b)解压\n");
    printf("      (c)退出\n\n");
    printf("****************************************\n\n");
    printf("功能选择：");
    char choice = getchar();
    fflush(stdin);                  //清空输入流
    while (!strchr("abcABC", choice)) {
        printf("\n输入有误！请重新输入：");
        choice = getchar();
        fflush(stdin);              //清空输入流
    }
    if (choice < 'a')
        choice -= 'A' - 'a';        //大写字母转小写

    return choice;
}

//压缩文件
bool compress(void) {
    char in_file[50];
    printf("\n待压缩文件名：");
    gets(in_file);
    fflush(stdin);

    FILE *fp = fopen(in_file, "rb");          //以二进制只读模式打开文件
    if (fp == NULL) {
        printf("\n文件不存在！\n\n");
        return false;
    }

    printf("\n压缩中……\n\n");

    node leaves[SIZE];                        //创建叶结点
    for (int i = 0; i < SIZE; i++) {
        leaves[i].weight = 0;                 //初始化叶结点
        leaves[i].parent = NULL;
        leaves[i].left = NULL;
        leaves[i].right = NULL;
    }
```

```
unsigned long file_len = 0;             //原文件长度
unsigned char temp;
fread(&temp, sizeof(unsigned char), 1, fp);          //读取8bit
while (!feof(fp)) {
    leaves[temp].weight++;                           //统计字符频率
    file_len++;                                      //统计原文件长度
    fread(&temp, sizeof(unsigned char), 1, fp);      //读取8bit
}

fclose(fp);           //关闭文件

Huffman_Tree(leaves);                   //建立哈夫曼树
char **Code = encoder(leaves);          //生成哈夫曼编码
free_tree(leaves);                      //释放树占用的内存

char out_file[54];                      //生成的压缩文件名
strcpy(out_file, in_file);
strcat(out_file, ".huf");

FILE *fp1 = fopen(in_file, "rb");       //以二进制只读模式打开文件
FILE *fp2 = fopen(out_file, "wb");      //以二进制只写模式打开文件

fwrite(&file_len, sizeof(unsigned long), 1, fp2);        //写入文件长度

//写入字符频率
for (int i = 0; i < SIZE; i++)
    fwrite(&(leaves[i].weight), sizeof(unsigned long), 1, fp2);

//写入哈夫曼编码
char code_buf[SIZE + 8] = "\0";                          //哈夫曼编码缓存
while (!feof(fp1)) {
    fread(&temp, sizeof(unsigned char), 1, fp1);         //读取8bit
    strcat(code_buf, Code[temp]);                        //编码字符写入缓存区
    while (strlen(code_buf) >= 8) {
        temp = '\0';
        for (int i = 0; i < 8; i++) {
            temp <<= 1;
            if (code_buf[i] == '1')
                temp |= 1;
        }
        fwrite(&temp, sizeof(unsigned char), 1, fp2);    //写入8bit
        strcpy(code_buf, code_buf + 8);                  //去除已处理的前8位
    }
}
```

```
    //处理最后不足8bit编码
    int length = strlen(code_buf);
    if (length > 0) {
        temp = '\0';
        for (int i = 0; i < length; i++) {
            temp <<= 1;
            if (code_buf[i] == '1')
                temp |= 1;
        }
        temp <<= 8 - length;
        fwrite(&temp, sizeof(unsigned char), 1, fp2);
    }

    fclose(fp1);      //关闭文件
    fclose(fp2);

    for (int i = 0; i < SIZE; i++)          //释放编码字符占用的内存
        free(Code[i]);
    free(Code);

    printf("压缩完成！\n\n");
    return true;
}

//解压文件
bool extract(void) {
    char in_file[54];                       //待解压文件名
    printf("\n待解压文件名：");
    gets(in_file);
    fflush(stdin);

    FILE *fp1 = fopen(in_file, "rb");       //以二进制只读模式打开文件
    if (fp1 == NULL) {
        printf("\n文件不存在！\n\n");
        return false;
    }

    printf("\n解压中……\n\n");

    char out_file[54] = "ext_";             //生成的解压文件名
    strncat(out_file, in_file, strlen(in_file) - 4);

    FILE *fp2 = fopen(out_file, "wb");      //以二进制只写模式打开文件
```

```
unsigned long file_len;                 //读取文件长度
fread(&file_len, sizeof(unsigned long), 1, fp1);

node leaves[SIZE];
for (int i = 0; i < SIZE; i++) {        //读取字符频率
    leaves[i].parent = NULL;            //初始化叶结点
    leaves[i].left = NULL;
    leaves[i].right = NULL;
    fread(&(leaves[i].weight), sizeof(unsigned long), 1, fp1);
}

Huffman_Tree(leaves);                   //建立哈夫曼树
node *root;                             //寻找根结点
for (int i = 0; i < SIZE; i++) {
    if (leaves[i].weight) {             //频率为0的字符不在哈夫曼树中
        root = &(leaves[i]);            //无法通过其寻找根结点
        break;
    }
}
while (root->parent)
    root = root->parent;

unsigned long written_len = 0;          //写入的编码个数
unsigned char temp;
node *leaf = root;
while (file_len != written_len) {       //二进制文件无法用feof判断文件结束
    fread(&temp, sizeof(unsigned char), 1, fp1);
    for (int i = 0; i < 8; i++) {
        leaf = (temp & 128) ? leaf->right : leaf->left;
        temp <<= 1;
        if (!leaf->left) {              //从根结点遍历至叶结点
            unsigned char ch = leaf - leaves;               //确定编码
            fwrite(&ch, sizeof(unsigned char), 1, fp2);     //写入8bit
            written_len++;
            if (file_len == written_len)
                break;
            leaf = root;
        }
    }
}

fclose(fp1);            //关闭文件
fclose(fp2);
free_tree(leaves);      //释放树占用的内存
```

```
    printf("解压完成！\n\n");
    return true;
}

//构建哈夫曼树
void Huffman_Tree(node *leaves) {
    node *tree[SIZE];                    //建立森林
    for (int i = 0; i < SIZE; i++)
        tree[i] = &(leaves[i]);
    //筛选法建立小顶堆
    for (int i = SIZE / 2 - 1; i >= 0; i--)      //从最后一个内部结点开始
        SiftDown(tree, SIZE, i);

    for (int count = SIZE - 1; count > 0; count--) {
        while (tree[0]->weight == 0) {
            tree[0] = tree[count];
            SiftDown(tree, count, 0); //调整小顶堆
            count--;
        }
        node *temp1 = tree[0];     //权重最小的元素（非零）

        tree[0] = tree[count];
        SiftDown(tree, count, 0);      //调整小顶堆

        while (tree[0]->weight == 0) {
            tree[0] = tree[count];
            SiftDown(tree, count, 0); //调整小顶堆
            count--;
        }
        node *temp2 = tree[0];     //权重次小的元素（非零）

        node *root = (node *)malloc(sizeof(node));
        root->weight = temp1->weight + temp2->weight;
        root->parent = NULL;
        root->left = temp1;            //生成子树
        root->right = temp2;
        temp1->parent = temp2->parent = root;
        temp1->code = '0';
        temp2->code = '1';

        tree[0] = root;
        SiftDown(tree, count, 0);      //调整小顶堆
    }
}
```

```
//筛选法调整堆
void SiftDown(node **tree, int n, int i) {
    node *temp = tree[i];
    while (2 * i + 1 < n) {            //不断下沉
        int index = 2 * i + 1;
        if (index < n - 1 && tree[index]->weight > tree[index + 1]->weight)
            index++;                   //取左右子结点中的较小者
        if (temp->weight <= tree[index]->weight) break;      //终止下沉
        else { tree[i] = tree[index]; i = index; }
    }
    tree[i] = temp;
}

//哈夫曼编码
char **encoder(node *leaves) {
    char **Code = (char **)malloc(SIZE * sizeof(char *));
    for (int i = 0; i < SIZE; i++) {
        node *temp = &leaves[i];       //从叶结点向上读取编码
        int j = 0;
        char ch_temp[SIZE];            //临时存储反向编码
        while (temp->parent) {
            ch_temp[j] = temp->code;
            j++;
            temp = temp->parent;
        }
        //获取正向编码
        char *code = (char *)malloc((j + 1) * sizeof(char));
        int m = 0;
        for (int k = j - 1; k >= 0; k--) code[m++] = ch_temp[k];
        code[m] = '\0';
        Code[i] = code;
    }
    return Code;
}

//销毁哈夫曼树
void free_tree(node *leaves) {
    for (int i = 0; i < SIZE; i++) {
        node *temp1 = leaves[i].parent; node *temp2;
        while (temp1) {
            temp2 = temp1->parent; free(temp1); temp1 = temp2;
        }
    }
}
```