文件压缩与关键字检索

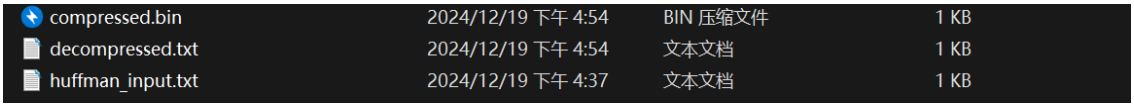
注意👀：在补充的要求中，要求对中文文档进行搜索🔍

So,以下的代码是可以对 中文英文进行统一的搜索😄

> **在报告之前，先展示基本的效果**

- **input.txt**　　采用UTF-16LE的编码格式，为了C语言能够按照2字节处理中英文

- **compress.bin**　　为实际的二进制压缩文件



- **decompress.txt**　为从.bin文件中提取出来的中英文字符

转换编码格式 🤯

- 代码

```c
int main()
{
    // 初始化基本的哈夫曼树
    inihuffman();

    // 打开文档,在不同的函数中，基本上都要反复操作文件，所以可以在局部函数中，创建一个行的文件描述符
    FILE* input = fopen("D:/huffman_input.txt", "r");
    FILE* compressed = fopen("D:/compressed.bin", "wb");
    FILE* decompressed = fopen("D:/decompressed.txt", "w");

    // 得到初始的叶子节点，统计字符频率
    characterCount(input);

    // 构建哈夫曼树
    build_huffman_tree();

    // 生成哈夫曼编码
    generate_huffman_codes();

    // 根据写出的哈夫曼编码，进行对文件的压缩，写入二进制文件,输出压缩率
    compress_in_file(compressed, input);

    // 根据文件进行解压操作
    decompress_to_file();

    // 搜索字符
    search_in_binfile();

    fclose(input);
    fclose(compressed);
    fclose(decompressed);
    return 0;
}
```

- **效果　　编码😖 +压缩率😚+中英文通配符搜索🔍**（出现次数，位置.....）

```
'近'哈夫曼编码为0110111
'进'哈夫曼编码为1111000
'释'哈夫曼编码为0111000
'重'哈夫曼编码为0111001
'量'哈夫曼编码为0111010
'阐'哈夫曼编码为0111011
'面'哈夫曼编码为1111001
'领'哈夫曼编码为1111010
'题'哈夫曼编码为1010110
'高'哈夫曼编码为10001
--------------------------------------
input->364 bytes
compress->148 bytes

so...压缩率为0.41
--------------------------------------
请输入要搜索的中英文字符
中国
--------------------------------------

从压缩文件提取出的字符串是：人才是中国式现代化的基础性、战略性支撑。推进教育科技人才一体化对全面提升高校党建工作水平、推动高校党建高质量发展提出了新要求新使命。全国高校以政治建设为统领、实党对高校工作的全面领导、坚持党的创新理论武装是党的思想建设的根本任务。持续推进实施习近平新时代中国特色思想和中国特色哲学等重大课题研究。以体系化学理化研究阐释帮助师生更好掌握正确的立场、观点、方法。

一共已找到 3 处，位置为4  128  135

F:\n\use\x64\Debug\use.exe (进程 38464)已退出，代码为 0。
```

实验一🩹：文件压缩

1. **由于一开始就考虑到了中文的情况，所以将文件采用了UTF-16LE的编码格式保存**
2. 初始化和构造哈夫曼树

```
// 初始化基本的哈夫曼树
inihuffman();
```

```
void inihuffman()
{
    for (int i = 0; i < MAX_NODE; i++)
    {
        huffman_tree[i].weight = 0;
        huffman_tree[i].l_kid = -1;
        huffman_tree[i].r_kid = -1;
        huffman_tree[i].parent = -1;
    }
    printf("初始化完毕\n");
}
```

3. **必须先计算出权值，才能着手构建整个哈夫曼树**

```
// 得到初始的叶子节点，统计字符频率
characterCount(input);
```

```
all_ch_infile_num //记录实际哈夫曼存储数组🌳边界的范围

fread(&bom, sizeof(short), 1, input);//一次文件读入两个字节，因为utf-16LE编码都是一个中文或者英文字符，都需要两个字节

wchar_t freq[MAX_leaf] //这里存了一张很大很大很大很大！！的数组，直接以所有的编码的码点数值作为下标，这样读入字符的对应频
// 率分布式存储，牺牲空间的高效的影射方法
```

```
int all_ch_infile_num = 0;
void characterCount(FILE* input)
{
    wchar_t ch = 0;
    for (int i = 0; i <= MAX_LEAF; i++)
    {
        freq[i] = 0;
    }
    short bom;
    fread(&bom, sizeof(short), 1, input);

    while ((fread(&ch, sizeof(short), 1, input)) >= 1)
    {
        freq[ch]++;          // 增加对应字符的频率
        all_ch_infile_num++; // 统计所有的字符总数
        ch = 0;
    }
    printf("--------%d\n", all_ch_infile_num);
}
```

4. **造哈夫曼 🌳 是基于我们的结构体数组，并且分为俩步骤**

```
// 构建哈夫曼树
build_huffman_tree();
```

- 遍历创建节点并进行合理的赋值

```
cur_count_num = 0;                  // 设置为0，后面会变
for (int i = 0; i < MAX_LEAF; i++) // 统计字符频率，这个的范围比较大，拿到了所有可能的字符
{
    if (freq[i] > 0)
    {
        huffman_tree[cur_count_num].weight = freq[i]; // 统计字符到分布式的数组里面
        huf_index_map_char[cur_count_num] = i;        // 记录这是什么字符
        cur_count_num++;                              // 大数组前几个都只压入了出现过的的叶子
    }
    // 哈哈，最后一个哈夫曼大数组的cur_count_num的位置没有东西,放第一个非叶子合成节点
}
```

- 统计总共参与建立哈夫曼 🌳 的有效叶节点个数：true_leaf_num(这个之后不会再变)，并且通过显而易见的固定的合并次数: true_leaf_num - 1，循环找到最小权值的两个节点，逐步合并。这意味着我们在哈夫曼的结构体数组中，不断创造非叶子节点，同时延伸数组的有效边界cur_count_num;

```
ture_leaf_num = cur_count_num; // 这才是正宗的基于不为零的频率表而进行生成的过程

for (int k = 1; k <= ture_leaf_num - 1; k++) // 必须进行 num_leaf -1 轮合并
{
    // 找出权重最小的两个节点
    int min1 = -1, min2 = -1; // 用于存储最小和第2小滴节点的索引
    for (int i = 0; i < cur_count_num; i++)
    {
        if (huffman_tree[i].parent == -1)
        { // 找到父节点为 -1 的有效节点
            // 如果min1还没有值或当前节点的权重更小，更新min1和min2
            if (min1 == -1 || huffman_tree[i].weight < huffman_tree[min1].weight)
            {
                min2 = min1; // 将min1的值赋给min2
                min1 = i;    // 更新min1为当前节点
            }
            // 否则，如果min2没有值或当前节点的权重大于min1但小于min2，更新min2
            else if (min2 == -1 || huffman_tree[i].weight < huffman_tree[min2].weight)
            {
                min2 = i; // 更新min2为当前节点
            }
        }
    }
    //printf("找到的最小：%d，权重：%d，此小：%d,权重：%d\n", min1, huffman_tree[min1].weight, min2, huffman_tree[min2].weight);

    // 创建父节点,在我们之前统计的位置的后面继续塞入实际有意义的哈夫曼节点
    huffman_tree[cur_count_num].weight = huffman_tree[min1].weight + huffman_tree[min2].weight;
    huffman_tree[cur_count_num].parent = -1;
    huffman_tree[cur_count_num].l_kid = min1;
    huffman_tree[cur_count_num].r_kid = min2;
    // 更新父节点
    huffman_tree[min1].parent = cur_count_num;
    huffman_tree[min2].parent = cur_count_num;
    // 更新大数组的末尾
    cur_count_num++;
}
// 整个树建立成功
```

这里是最重要的部分： 🌿

如何由一棵树，从叶子到根逆行，得到哈夫曼编码01序列（数字01）->>

- **为每个字符，维护一个01长度数组**

```
void generate_huffman_codes()
{
    for (int i = 0; i < ture_leaf_num; i++)
    {
        code_length[i] = 0;
    }
    // char huffman_codes[MAX_LEAF + 10][MAX_BIT]; // 每个字符的哈夫曼编码
    // int code_length[MAX_LEAF + 10]; // 每个字符的哈夫曼编码长度，注意这里没有未出现的字符
    // 树已经建立好了，所以其实只是对cur_count_num才有建立的意义，其余的频率为0的节点我就没有去建立了
```

- **构造一个可以移动的指针，从叶子到根回溯，并且，不断往int数组里面，按顺序写入01，到时候再做处理**，注意该节点编码长度也发生变化++
- **// 麻烦所有的节点，无论非叶子，只要在树上就要一视同仁，都要编码**！！！！！

```
    for (int i = 0; i < cur_count_num; i++) // 麻烦所有的节点，无论叶子，只要在树上就要一视同仁
    {
        // printf("此时是i = %d节点\n", i);
        int temp_parent;
        int temp_kid = i;
        if (huffman_tree[i].parent != -1)
        {
            for (temp_parent = huffman_tree[i].parent; temp_parent != -1; temp_kid = temp_parent, temp_parent = huffman_tree[temp_parent].parent)
            {
                // printf("temp_parent = %d\n", temp_parent);
                if (temp_kid == huffman_tree[temp_parent].l_kid)
                {
                    temp_code[i][code_length[i]++] = 0;
                    // printf("是左子\n");
                }
                else if (temp_kid == huffman_tree[temp_parent].r_kid)
                {
                    temp_code[i][code_length[i]++] = 1;
                    // printf("是右子\n");
                }
            }
        }
```

将01倒置，同时又要将01直接影射转换为字符串 -->>

```
    for (int j = 1; j <= code_length[i]; j++) // 有多少个编码
    {
        huffman_codes[i][code_length[i] - j] = '0' + temp_code[i][j - 1]; // 逆向结束，用'0'偏移，形成字符
    }
    huffman_codes[i][code_length[i]] = '\0'; // C字符串结尾
```

压缩01串串-----》》.bin 文件

```
// 根据写出的哈夫曼编码，进行对文件的压缩，写入二进制文件，输出压缩率
compress_in_file(compressed, input);
```

```
//文件永远都是持久化的工具，所以我们对文件的读写操作，必须通过一个缓冲区
//注意，操作缓冲区的有效01bit,需要我们用bit_count不断移动控制！！！
    wchar_t ch = 0;
    unsigned long bit_buffer = 0;          // 用于所有字符的二进制数据的缓冲区----->提供比特空间即可
    int bit_count = 0;                     // 缓冲区的有效01bit，用bit_count跟踪
```

- 经验：习惯调节文件指针的位置

```
// 进行文件的读入之前，必须把原来的文件表的pos改为开头0
fseek(input, 0, SEEK_SET);
```

- 位操作才是真的高难度

```
//计算机里面没有直接控制比特位的操作，必须进行特定的位操作公式
// 通过|（加法）和 <<移位（把后头的01给不断移到高位），实现了二进制位串的改变，仿佛像水流，从Buffer的低位到高位漫延
bit_buffer = (bit_buffer << 1) | (huffman_codes[ch_index][i] - '0');
//当最后编码填充总数不足8，我们直接填满，(8 - bit_count)，就不一个一个移动了，直接补到开头
    if (bit_count != 0)
    {
        bit_buffer = (bit_buffer << (8 - bit_count));
        fwrite(&bit_buffer, sizeof(char), 1, compress);
    }
```

- 总体思路：对每一个编码字符的每一个01串，都要充分的写进文件里，缓冲区以8bit为单位不断写进.bin文件里，写满了就异步刷新，重置bit_count

```
    int ch_lenth = 0;
    int ch_index; // 这个不能做循环变量
    int ch_temp_index;
    for (ch_temp_index = 0; ch_temp_index < ture_leaf_num; ch_temp_index++)
    {
        if (huf_index_map_char[ch_temp_index] == ch) // 得知此时读入的字符，处于哈夫曼编码的哪个下标的位置
        {
            ch_lenth = code_length[ch_temp_index]; // 重点是每一种字符所在的哈夫曼大数组的index下标要找到
            ch_index = ch_temp_index;
            // printf("%c下标为%d:%s\n", ch, ch_temp_index, huffman_codes[ch_temp_index]);
        }
    } // 得到读入的某一个字符的长度，也可以知道编码

    // 看好了，告诉你如何把字符中的01，经过层层转化，一个一个当做二进制输入到缓冲区中！
    // printf("这里是不断的注入%c:\n", ch);
    // printf("开始%d个有效bit\n", bit_count);
    // printf(":%s\n", huffman_codes[ch_index]);
    for (int i = 0; i < ch_lenth; i++)
    {
        // 不断把哈夫曼表01字符串以某种形式，变成逐步变长的01流，看好了，，，，，，，，，，，，，，，，，，，，，，，，，，，，这里其实是和上面一样，默认假设
        // printf("bit_buffer  =  %u\n", (bit_buffer));
        // printf("(huffman_codes[%d][%d])  =   %u\n", ch_index, i, (huffman_codes[ch_index][i]));
        // printf("(huffman_codes[%d][%d] - '0')  =   %u\n", ch_index, i, (huffman_codes[ch_index][i] - '0'));
        bit_buffer = (bit_buffer << 1) | (huffman_codes[ch_index][i] - '0'); // 通过|和 <<移位，实现了二进制位串的改变
        bit_count++;
        if (bit_count == 8)
        {                                                   // 积累捕获到了8个位就直接把32位中的低8位写入
            fwrite(&bit_buffer, sizeof(char), 1, compress); // 这个从小端序机器写入底层二进制的方法，你以前估计搞不出
            // fputc(bit_buffer, compress);  //也是一种策略，但是会有自动截断低八位
            bit_buffer = 0;
            bit_count = 0;
        }
    }
    ch = 0;
}
// 你在缓冲区的使用中，前头刚刷新，又要从最低一位开始，可能有部分的数据没有凑齐8bits，没法顶到8bits的高位，失去连续，则人工补齐后面0000
if (bit_count != 0)
{
    bit_buffer = (bit_buffer << (8 - bit_count));
    fwrite(&bit_buffer, sizeof(char), 1, compress);
}
```

----------》写出压缩率

- 最后，用文件指针快速统计文件大小,写出压缩率

```
    printf("-------------------------------------");

    fseek(input, 0, SEEK_END);
    printf("\ninput->%ld bytes\n", ftell(input));

    fseek(compress, 0, SEEK_END);
    printf("compress->%ld bytes\n", ftell(compress));

    printf("\nso...压缩率为%.2lf\n", ftell(compress) * 1.0 / ftell(input));
    printf("-------------------------------------\n");
}
```

文件解码

- 思路：文件解码和压缩字符进文件一样，不过是通过构建一个缓冲区，把文件里面的01流一块块读出来，通过这些01流，控制我们反复从哈夫曼树的树顶，向下移动，到达叶子节点后，在把该叶子节点对应的中英文字符写进.txt文档中
- 为此，我们得有一个指针（下标），遍历整个哈夫曼树（其实就是在哈夫曼数组里面跳来跳去），然后，对于每一个下标，我们应该提前构建一个辅助数组（wchar_t huf_index_map_char[MAX_LEAF]），帮助我们从下标直接影射到这个中英文字符的编码数值。
-

```
FILE* compressed = fopen("D:/compressed.bin", "rb");
FILE* decompressed = fopen("D:/decompressed.txt", "w");
unsigned bit_buffer = 0;
int bit_count = 0;
int root;
root = get_root_index(); // 找到哈夫曼树的树根
int cur_huffnode = root;
```

-

```
while (now_decode_num < all_ch_infile_num)
{
    // buf的消耗补充，知道读取结束
    if (bit_count == 0)
    {
        // printf("here1\n");
        unsigned num = fread(&bit_buffer, sizeof(char), 1, compressed);
        bit_count = 8; // 有效位置满血
        if (num == 0)
        {
            break;
        } // 跳出
    }

    // 在缓冲区中捕获每一个有效的位，并从哈夫曼树的root逐步下移，到达叶子节点，则更新，并且写入，路径是实时更新的
    int now = (bit_buffer >> (bit_count - 1)) & 1; //&是对位的处理的必要步骤
    bit_count--;                                   // 有效的位的个数减少
    // 用拿到的位移动
    if (now == 1)
    {
        //printf("now == 1\n");
        cur_huffnode = huffman_tree[cur_huffnode].r_kid;
    }
    else if (now == 0)
    {
        //printf("now == 0\n");
        cur_huffnode = huffman_tree[cur_huffnode].l_kid;
    }

    // 检查移动结果
    if (huffman_tree[cur_huffnode].r_kid == -1 && huffman_tree[cur_huffnode].l_kid == -1)
    {
        wchar_t ch = huf_index_map_char[cur_huffnode];
        //printf("%c\n", ch);
        fwrite(&ch, sizeof(short), 1, decompressed);
        //fputc(ch, decompressed); // 写入新字符
        now_decode_num++;        // 目前还没有复刻完原文件中所有的字符
        cur_huffnode = root; // 重新从头开始移动
    }
}
```

实验二：关键字检索

检索：在bin中经过缓冲区提取后，进行检索

- 先哈夫曼解码，形成字符串，在内存中进行查询🔍

```
while (now_decode_num < all_ch_infile_num)
{
    // buf的消耗补充，知道读取结束
    if (bit_count == 0)
    {
        // printf("here1\n");
        unsigned num = fread(&bit_buffer, sizeof(char), 1, compressed);
        bit_count = 8; // 有效位置满血
        if (num == 0)
        {
            break;
        } // 跳出
    }
    int now = (bit_buffer >> (bit_count - 1)) & 1; //&是对位的处理的必要步骤
    bit_count--;                                   // 有效的位的个数减少
    // 用拿到的位移动
    if (now == 1)
    {
        //printf("now == 1\n");
        cur_huffnode = huffman_tree[cur_huffnode].r_kid;
    }
    else if (now == 0)
    {
        //printf("now == 0\n");
        cur_huffnode = huffman_tree[cur_huffnode].l_kid;
    }
    // 检查移动结果，这里才要改一下
    if (huffman_tree[cur_huffnode].r_kid == -1 && huffman_tree[cur_huffnode].l_kid == -1)
    {
        // printf("here2\n");
        // printf("here2\n");
        setlocale(LC_ALL, "zh_CN");  // 设置中文环境
        // 在这里处理需要本地化的部分
        wchar_t ch = huf_index_map_char[cur_huffnode];
        //wprintf(L"%lc\n", ch);
        file_str[now_decode_num] = ch;
        setlocale(LC_ALL, "C");  // 恢复到默认的C环境（POSIX标准
        // 在这里处理不需要本地化的部分

        now_decode_num++; // 目前还没有复刻完原文件中所有的字符
        cur_huffnode = root; // 重新从头开始移动
    }
}//此时已经完全解析完了file中的字符串，并转移动内存中
```

- 选择合适的查询方式，进行高效的统计

```
//开始正式的查询
int user_str_len = 0;
int file_str_len = now_decode_num;
for (int i = 0; user_str[i] != 0; i++) {
    user_str_len++;
}
//设置查找变量
int file_i = 0;
int user_i = 0;
int find_cnt = 0;

while (file_i < file_str_len) {
    if (user_str[user_i] == file_str[file_i])
    {
        user_i++;
        file_i++;
        if (user_i == user_str_len) {
            user_i= 0;
            user_trace[find_cnt] = file_i - user_str_len;
            find_cnt++;
        }
    }
    else if(user_str[user_i] != file_str[file_i])
    {
        file_i = file_i - (user_i - 1);
        user_i = 0;
    }
}

if (find_cnt > 0) {
    wprintf(L"\n一共已找到 %d 处，位置为",find_cnt);
    for (int i = 0; i < find_cnt; i++) {
        wprintf(L"%d  ", user_trace[i] + 1);
    }
    putwchar(L'\n');

}
else {
    wprintf(L"该文档未找到");
}
```

采用BM算法来高效检索

```
void search_in_binfile(FILE *compressed) {
    printf("请输入要搜索的英文字符: ");
    char user;
    scanf(" %c", &user);  // 使用 " %c" 以处理换行符

    // 先判断，后搜索
    int find = 0;
    int user_index;
    for (int i = 0; i < ture_leaf_num; i++) {
        if (user == huf_index_map_char[i]) {
```

```
            find = 1;
            user_index = i;
            break;  // 找到字符后，跳出循环
        }
    }

    if (find == 0) {
        printf("文档未搜索到该字符\n");
        return;
    }

    // 正式开始搜索
    fseek(compressed, 0, SEEK_SET);  // 重置文件指针
    char buffer[1024];               // 用来缓存读取的字节数据
    size_t bytes_read;

    // 在压缩文件中搜索字符
    while ((bytes_read = fread(buffer, 1, sizeof(buffer), compressed)) > 0) {
        for (size_t i = 0; i < bytes_read; i++) {
            // 在每个字节中逐位进行搜索
            for (int bit_index = 7; bit_index >= 0; bit_index--) {
                // 通过位运算获取当前比特位的值
                int bit = (buffer[i] >> bit_index) & 1;

                // 如果当前比特匹配目标字符的对应位，继续进行下一个比特的匹配
                if (bit == 1) {
                    // 检查是否已经找到完整字符
                    if (user_index == i) {
                        // 输出结果
                        printf("找到字符 %c，在字节 %zu 位置。\n", user, i);
                        return;
                    }
                }
            }
        }
    }

    printf("未找到该字符\n");
}
```

分析查询错误的可能性，和效率的关系

在查询操作中，错误的发生频率和查询的效率是密切相关的

- 提高查询效率降低错误率
    - 在高效的字符串检索算法中，通常会减少无效比较和不必要的检查，这意味着 **错误发生的可能性较低。**例如，在 **Boyer-Moore** 算法中，若存在不匹配的字符，算法通过跳跃式地移动模式，从而避免了不必要的字符比较，这不仅提高了效率，也减少了错误发生的可能性。
- 较高的错误率可能导致较低的效率
    - 如果算法在处理过程中无法正确地进行匹配，或者误报了错误的匹配，那么需要进行额外的步骤来纠正错误，这会 **增加额外的处理时间**，从而影响查询效率。例如，如果一个字符串匹配

算法存在较高的错误率，它可能会在每个查询中进行额外的回溯、重试或回滚操作，从而降低整体查询效率。

分析压缩率的影响因素，并给出提高的方法

影响因素：

- 不同的压缩算法在压缩效率、速度、压缩率方面有所不同。每种算法的压缩原理不同，因此压缩结果也会有所差异。
    - **Huffman 编码**：对于数据中频繁出现的字符，Huffman 编码通过较短的编码来代替高频字符，得到高效压缩。
    - **LZ77 和 LZ78**：这两种算法依赖于数据中出现的重复模式，通过查找和替换重复的数据来进行压缩，适合于含有很多重复数据的文件。
    - **BZIP2 和 LZMA**：这些算法通过更复杂的压缩策略，通常可以提供更高的压缩率，但也需要更多的计算资源。
- 我们的.txt的自身的编码方式也会对效率产生影响

提高方法：

- **选择合适的压缩算法**：根据数据的特点选择适当的压缩算法。对于大量重复的文本文件，**Huffman 编码** 和 **LZ77/LZ78** 是常见选择；而对于更复杂的文本或文件，可以选择 **BZIP2** 或 **LZMA** 等算法来提高压缩率。
- **混合算法**：结合多种压缩算法，例如使用 **LZ77** 进行初步压缩后，再使用 **Huffman 编码** 来进一步优化压缩率。