《文件压缩》大作业报告

班级：智能交通与车辆 14 班

学号：2152402

姓名：段婷婷

完成日期：2022.4.20

目录

一. 功能描述与设计思路

1. 功能描述:

使用**哈夫曼编码算法**对文件进行**无损压缩。**

使用 cmdline 方式读取参数，参数格式为 {压缩文件名} {输出文件名} {压缩指令 (zip/unzip}

对所提供的 ser.log 文件，文件压缩率及压缩时间如下：

```
C:\Users\DTTTTTTT\Desktop\文件压缩\文件压缩大作业\文件压缩>文件压缩.exe ser.log ser_compressed.log zip
文件压缩率：65.3977%
程序运行时间：6.098秒

C:\Users\DTTTTTTT\Desktop\文件压缩\文件压缩大作业\文件压缩>文件压缩.exe ser_compressed.log ser_decompressed.log unzip
程序运行时间：2.771秒
```

2. 设计思路:

（1）ZIP 中：哈夫曼树的建立

第一步，读入待压缩文档中所有的字符，并且统计各个字符出现的次数，存储在 unordered_map<char,int>类型的 cnt 中，由此可以对任意字符方便地获取其出现次数。

```
while (!infile.eof()) { //统计所有字符出现的次数
    char c;
    //infile >> c; 注意到此种读入会忽略空格与换行!
    c = infile.get();
    if (infile.eof())break; //此句的必要性与eof()的特性有关
    if (cnt.find(c) == cnt.end()) cnt[c] = 1;
    else cnt[c]++;
}
```

第二步，对所有的字符创建哈夫曼结点（它们将成为哈夫曼树的叶结点），并且 push 进优先队列 que 中。具体来说，定义一个 unordered_map<char,int> 类型的迭代器 it 来遍历 cnt，建立对应的结点（使**用动态申请内存**），并且放入 priority_queue<Huffman_node*, vector<Huffman_node*>, cmp> 类型的 que

中。其中 cmp 需要自定义结构体，以确定 Huffman_node*类型在优先队列中的比较方式，即出现次数越少，优先级越高。

```
unordered_map<char, int>::iterator it; //定义一个unordered map<char,int>类型的迭代器用来遍历 cnt
for (it = cnt.begin();it != cnt.end();it++) { //创建所有字符的Huffman结点并且加入优先队列
    char c = it->first;
    int freq = it->second;
    Huffman_node* nd = new Huffman_node;
    nd->c = c;
    nd->freq = freq;
    nd->num = ++node_num;
    que.push(nd);
}
```

```
priority_queue<Huffman_node*, vector<Huffman_node*>, cmp>que; //构建Huffman时所需的对结点排序
```

```
struct cmp {
    bool operator() (const Huffman_node* x, const Huffman_node* y) {
        return x->freq > y->freq;
    }
};
```

第三步，每次取出队首两个结点，建立新结点，并将取出的两个结点作为新结点的左右孩子，新结点的 freq 值为左右孩子 freq 值之和，最后将新结点 push 入 que。直到 que 中只剩下一个结点，哈夫曼树建立完毕，最后剩下的结点即为根结点。

```
while (que.size() > 1) {
    Huffman_node* n1, * n2; //取出优先队列队首的两个元素
    n1 = que.top();
    que.pop();
    n2 = que.top();
    que.pop();

    Huffman_node* nd = new Huffman_node; //创建新结点：n1 n2的父结点
    nd->freq = n1->freq + n2->freq;
    nd->left_child = n1;
    nd->right_child = n2;
    nd->num = ++node_num;

    que.push(nd);
}

root = que.top(); //最后剩下的结点是根结点
```

（2）ZIP 中：每个字符的编码字符串（01 串）的得到

此步骤通过函数 traverse_tree(Huffman_node*)遍历哈夫曼树完成。在函数中定义一个 static string 类型的 str，表示当前结点的编码字符串。当下一个去左孩子结点时，在 str 后加上”0”；当下一个去右孩子结点时，在 str 后加上”1”。注意此处访问完之后需要对 str 进行回溯。若当前结点是叶结点，则记录对应字符的编码字符串，存储至 unordered_map<char,string>类型的 ans 中。

```
void traverse_tree(Huffman_node* t) { //遍历Huffman tree来找到每个字符（叶结点）对应的
    static string str;
    int len = str.length();
    if (t->left_child != NULL) {
        str = str + "0";
        traverse_tree(t->left_child);
        str.erase(len);
    }
    if (t->right_child != NULL) {
        str = str + "1";
        traverse_tree(t->right_child);
        str.erase(len);
    }
    if (is_leaf(t)) ans[t->c] = str;
}
```

（3）ZIP 中：压缩文件最终的得到

首先需要输出哈夫曼树，因为解压缩时需要重构哈夫曼树，用函数 void output_tree(Huffman_node* t,char* fn)实现。具体来说，先遍历子结点，再输出当前结点，保证输出一个结点时，它的左右子结点都已输出，便于解压缩时重构 Huffman tree。输出的信息包括结点是否为叶结点（一个 bool 类型），结点编号（一个 int 类型），若为叶结点，则输出对应字符（一个 char 类型），否则输出左右子结点编号（两个 int 类型，若无相应子结点则输出 0）。

```
if (t->left_child != NULL)
    output_tree(t->left_child, fn);
if (t->right_child != NULL)
    output_tree(t->right_child, fn);

//先遍历，再输出，保证输出一个结点时，它的左右子结点都已输出，便于解压缩时重构Huffman tree
fl.open(fn, ios::binary | ios::app);
if (!fl) {
    cout << "Error opening file" << endl;
    exit(1);
}

if (is_leaf(t)) { //叶结点：首先输出一个代表叶结点的 1，接着输出编号与叶结点对应字符
    bool leaf = 1;
    fl.write((char*)&leaf, sizeof(bool));
    fl.write((char*)&t->num, sizeof(int));
    fl.write((char*)&t->c, sizeof(char));
}
else { //非叶结点：首先输出一个表示非叶节点的 0，接着输出编号与其左右子结点编号（无子结点即用0表
    bool leaf = 0;
    int child = 0;

    fl.write((char*)&leaf, sizeof(bool));
    fl.write((char*)&t->num, sizeof(int));
    if (t->left_child == NULL)
        fl.write((char*)&child, sizeof(int));
    else
        fl.write((char*)&t->left_child->num, sizeof(int));
    if (t->right_child == NULL)
        fl.write((char*)&child, sizeof(int));
    else
        fl.write((char*)&t->right_child->num, sizeof(int));
}
```

然后输出待压缩文件的编码字符：再次读取待压缩文件，对于当前字符 c，将其编码字符串 ans[c]加入答案字符串 ans_str。此处还需要对 ans_str 进行进一步处理以达到压缩的目的：

将 ans_str 每八位作为一个二进制数用一个 unsigned char 表示，对于最后

不满八位的，需要单独处理（具体在【二.在实验过程中遇到的困难及解决办法】中阐述）。

```
string ans_str;
while (!infile.eof()) {
    char c;
    c = infile.get();
    ans_str += ans[c];
    while (ans_str.length() >= 8) {
        unsigned char ansc = 0;
        for (int i = 0;i < 8;i++)
            ansc = ansc << 1 | (ans_str[i] - '0');
        outfile.write((char*)&ansc, sizeof(unsigned char));
        ans_str.erase(0, 8);
    }
}
if (ans_str.length()) {
    unsigned char ansc = 0;
    for (int i = 0;i < ans_str.length();i++)
        ansc = ansc << 1 | (ans_str[i] - '0');
    outfile.write((char*)&ansc, sizeof(unsigned char));
}
```

此外，为了便于解码，还输出了：哈夫曼结点的总数目 node_num，输出的 unsigned char 类型的总数目 tot_ansc，原文本对应的 01 串总长度除以 8 所得的余数 left_num 到压缩文件中。

（4）UNZIP 中：哈夫曼树的重构

因为输出描述结点父子关系的方式为输出左右子结点的编号，所以需要建立编号与对应结点的关系。此处需要用到一个unordered_map<int,Huffman_node*>类型的 mp2。由于在输出时已经保证一个结点输出时，它的孩子结点一定已经输出，所以可以直接建立当前结点与其孩子结点的关系：

```
if (n1 != 0)nd->left_child = mp2[n1];
if (n2 != 0)nd->right_child = mp2[n2];
```

```
infile.read((char*)&node_num, sizeof(int));  //开始重构Huffman tree
while (node_num--) {
    Huffman_node* nd = new Huffman_node;
    if (node_num == 0) { //最后一个是根结点!
        root = nd;
    }
    bool leaf;
    infile.read((char*)&leaf, sizeof(bool));
    infile.read((char*)&nd->num, sizeof(int));
    mp2[nd->num] = nd;
    if (leaf) {
        infile.read((char*)&nd->c, sizeof(char));
    }
    else {
        int n1 = 0, n2 = 0;
        infile.read((char*)&n1, sizeof(int));
        infile.read((char*)&n2, sizeof(int));
        if (n1 != 0)nd->left_child = mp2[n1];
        if (n2 != 0)nd->right_child = mp2[n2];
    }
}
```

(5) UNZIP 中：读取原文本对应的编码并且解码

首先读入 unsigned char 类型的总数以及原文本对应的 01 串总长度除以 8 所得的余数。

然后依次读取 unsigned char 类型的 c。将 c 存储 bitset<8>可以方便的获得它的二进制表示。所以，每读入一个 c，就可以从当前位置开始走 8 步。如果走到叶结点，则输出对应字符，并且位置回到根结点。

```
long long tot_ansc, left_num; //开始读取原文本对应的压缩编码
infile.read((char*)&tot_ansc, sizeof(long long));
infile.read((char*)&left_num, sizeof(long long));

Huffman_node* cur = root;
while (tot_ansc--) {
    unsigned char c;
    infile.read((char*)&c, sizeof(unsigned char));
    bitset<8> bit(c);

    for (int i = 7;i >= 0;i--) {//倒序!
        if (bit[i] == 0) //注意是0而非'0'!
            cur = cur->left_child;
        else
            cur = cur->right_child;

        if (is_leaf(cur)) {
            outfile << cur->c;
            cur = root;
        }
    }

}
```

对于最后一个 unsigned char 类型的字符，需要对其单独处理，因为它包含的 8 位并不是都代表原文本的压缩信息。

```
if (left_num != 0) { //最后一个单独处理
    unsigned char c;
    infile.read((char*)&c, sizeof(unsigned char));
    bitset<8> bit(c);
    for (int i = 7 - left_num;i >= 0;i--) {
        if (bit[i] == 0)
            cur = cur->left_child;
        else
            cur = cur->right_child;
        if (is_leaf(cur)) {
            outfile << cur->c;
            cur = root;
        }
    }

}
```

二．在实验过程中遇到的困难及解决方法

1．**困难**：建立哈夫曼结点的优先队列中存储的是 Huffman_node*类型，如何表示其优先顺序？

解决方法：手写一个 cmp 结构体，具体如下：

```
struct cmp {
    bool operator() (const Huffman_node* x, const Huffman_node* y) {
        return x->freq > y->freq;
    }
};
```

注意到优先队列默认为大根堆，所以此处应该 return x->freq > y->freq，如此才能得到按照 freq 值从小到大排列的优先队列。

2. **困难：**直接输出了原文本对应的压缩字符串（01 串），发现文件变得更大了，如何进一步处理？

解决方法：注意到直接输出 01 串时，每一个 0 或 1 都是 char 类型，占了一个字节的空间，若**将每长度为 8 的 01 串作为二进制数存入 unsigned char 类型**，则一个 0 或 1 只占一个比特的空间，所占空间缩减到原来的八分之一。

3. **困难：**处理原文本对应的压缩字符串 (01 串) 时，每八个用一个 unsigned char 类型的字符表示，剩下的总数不满 8 的 01 串如何处理？

解决方法：假设剩下的 01 串长 len，记录这个长度并且输出到压缩文件中，将剩下的 01 串仍然用 unsigned char 类型的 ansc 表示，但是注意到此 ansc 中并不是 8 位全部表示压缩信息，而是只有 0~(len-1)位含压缩信息。为了再解压缩的时候能够清楚何时到了最后一个 unsigned char 类型的字符，另外再统计一个 tot_ansc 表示输出 unsigned char 类型的总数减去 1，即除去最后一个单独处理的。

```
if (ans_str.length()) {
    unsigned char ansc = 0;
    for (int i = 0;i < ans_str.length();i++)
        ansc = ansc << 1 | (ans_str[i] - '0');
    outfile.write((char*)&ansc, sizeof(unsigned char));
}
```

```
if (left_num != 0) { //最后一个单独处理
    unsigned char c;
    infile.read((char*)&c, sizeof(unsigned char));
    bitset<8> bit(c);
    for (int i = 7 - left_num;i >= 0;i--) {
        if (bit[i] == 0)
            cur = cur->left_child;
        else
            cur = cur->right_child;
        if (is_leaf(cur)) {
            outfile << cur->c;
            cur = root;
        }
    }

}
```

4. **困难**：使用何种方式读写文件？

解决方法：尝试了 get 和流的方式，都无法很好的处理读压缩文件的需求。翻了谭书，发现**二进制文件读写的 read 和 write 超级好用！** 可以任意选择类型读写，这样就可以完全不用考虑是否中间需要间隔或换行等等。

5. **困难**：如何重构哈夫曼树？

解决方法：**对哈夫曼结点增加一个编号 num 变量**，以此来联系父子结点关系。在输出压缩文件时，对每个非叶结点输出其编号 num，左孩子的编号和右孩子的编号，并且，要保证在该结点输出前，其孩子结点已输出。在解压缩时，每次读到一个结点的信息时，创建新结点，并且建立它的编号与结点的关系（存入 unordered_map<int,Huffman_node*>mp2 中），这样一来，读到其左右孩子的编号，便可以知道指向其左右孩子结点的指针，直接赋给 nd->left_child 和 nd->right_child 就好。其中还有一些细节就不赘述了。

6. **困难**：在解压缩时如何方便地得到一个 unsigned char 字符的二进制表示？

解决方法：用 **bitset!** 可以直接实现转化。

```
unsigned char c;
infile.read((char*)&c, sizeof(unsigned char));
bitset<8> bit(c);

for (int i = 7;i >= 0;i--) {//倒序!
    if (bit[i] == 0) //注意是0而非'0'!
        cur = cur->left_child;
    else
        cur = cur->right_child;

    if (is_leaf(cur)) {
        outfile << cur->c;
        cur = root;
    }
}
```

三．在此次作业中学会的新知识小结

1. 当然时**哈夫曼编码算法**来压缩文件，以上都在讲这个，下面说一点其他细节的知识。

2. 带参主函数

我们都知道，在编写 C/C++ 程序时，主函数 `main()` 是特别重要的，因为如果缺少这个函数的实现，我们的程序就无法编译。就像C/C++ 中的所有其他函数一样，主函数 `main()` 函数也能够接受参数。向主函数 `main()` 传递参数与向其他函数传递参数的区别在于，在前一种情况下，我们必须通过命令行传递参数，因为主函数 `main()` 本身是驱动函数，所以没有其他函数能够调用它并向它传递参数。

`argv` 和 `argc` 是 C 和 C++ 中向 `main()` 传递命令行参数的方式。按照惯例，这两个变量被命名为 `argc`（*argument count*, 参数计数）和 `argv`（*argument vector*, 参数向量），但它们可以被赋予任何有效的标识符——例如 `int main(int num_args, char** arg_strings)` 也是有效的。`argc` 是 `argv` 所指向的字符串的数量，在程序中一般表示 1 加上参数的数量。

如果你不打算处理命令行参数，也可以完全省略它们，就像我们所习惯的那样，写成 `int main()` 的形式。

学习了带参主函数并且在作业中用到它来传递输入输出文件名，以及 zip/unzip 的指令。

```cpp
int main(int argc, char** argv) {
    clock_t starttime, endtime;
    starttime = clock();

    if (argc != 4) {
        cerr << "Please make sure the number of parameters is correct." << endl;
        return -1;
    }
    if (strcmp(argv[3], "zip") == 0) {
        ZIP(argv);
    }
    else if (strcmp(argv[3], "unzip") == 0) {
        UNZIP(argv);
    }
    else {
        cerr << "Unknown parameter!\nCommand list:\nzip/unzip" << endl;
        return -1;
    }
```

```
C:\Users\DTTTTTTT\Desktop\文件压缩\文件压缩大作业\文件压缩>文件压缩.exe ser.log ser_compressed.log zip
文件压缩率：65.3977%
程序运行时间：6.098秒

C:\Users\DTTTTTTT\Desktop\文件压缩\文件压缩大作业\文件压缩>文件压缩.exe ser_compressed.log ser_decompressed.log unzip
程序运行时间：2.771秒
```

3. 优先队列中元素为指针时的重载运算符方式：

写一个 cmp 的结构体，具体见下：

```cpp
struct cmp {
    bool operator() (const Huffman_node* x, const Huffman_node* y) {
        return x->freq > y->freq;
    }
};
```

4. 用二进制方式打开文件，并且用 **read** 和 **write** 读写二进制文件

```cpp
ifstream infile(argv[1], ios::binary); // 以二进制方式打开文件
ofstream outfile(argv[2], ios::binary);
```

```cpp
infile.read((char*)&leaf, sizeof(bool));
infile.read((char*)&nd->num, sizeof(int));
```

```
outfile.write((char*)&tot_ansc, sizeof(long long));
outfile.write((char*)&left_num, sizeof(long long));
```

四．心得体会

1. 注意细节

（1）打开文件后，检查文件是否成功打开

（2）打开的文件都要及时关闭

（3）申请的动态内存都要及时处理：

结束操作后，需要 delete 掉所有的 Huffman 结点!

（4）变量名要取得容易理解，可能不如直接叫 x、y、z 简洁，但是在一个稍有些复杂的程序里，按照意义取名真的很重要！这样思路也会更加清晰。

2. 保持清醒的头脑，不要焦躁

这个作业一共写了 3 天，在前一天半里比较急躁，没有一步步来，遇到问题没有一步步分析，**归根结底是想跳过一步步踏实走而存在着侥幸心理，幻想着会不会直接过了**，最后甚至有些崩溃。重新整理状态，开始慢慢 debug、想处理方案、向 TA 和同学请教，一步一步地解决一个个小问题，一点点耐心地排查 bug，才终于完成作业。

所以！写这样的程序的时候，不如慢慢来，理清思路，耐心冷静，进度也许不那么快，但是很稳定，最后花的时间反而会少些。

3. 及时和同学讨论、向 TA 请教、翻书学习非常重要

在遇到瓶颈想不通时，和同学讨论讨论非常能够开阔思路！在这次作业中，关于如何处理原文本 01 串最后不满 8 位的零碎部分，我想了很久也没有一个清晰的思路。本来是想请教一个同学，但他也不会，于是我们一起讨论了一下，结果很快想出了好几种办法。这样大抵是因为，和人交流的过程，其实也是理清自己思路的过程。

在遇到一些知识性问题是，向 TA 请教能够事半功倍。在读压缩文件时我最

初用的是 get()，遇到了读到中间会中止的问题，浪费了大量的时间。最后请教了 TA，一句话就解答了我的问题，因为压缩文件不是标准形式了，应该考虑其他方式。

在寻找其他方式时，在从来没有翻开过的谭书里找到了最符合我需求的方式！**有时候这些并不是在网上搜索容易得到的知识，而是需要翻阅系统性的教材才能获取的。**

五. 源代码

```
#include<iostream>
#include<fstream>
#include<cstring>
#include<queue>
#include<time.h>
#include<bitset>
#include<unordered_map>

using namespace std;

int node_num; //Huffman tree 的结点总数

struct Huffman_node { //定义 Huffman 结点
 char c = '\0';
 int freq = 0, num = 0;
 Huffman_node* left_child = NULL, * right_child = NULL;
 /*friend bool operator <(Huffman_node x, Huffman_node y) {   //定义该结构体的大小比较
```

```
18.     return x.freq < y.freq;   //不过由于此题中需要比较的是指向该结构体的指针，所以不采用这种方式
19.     }*/
20.     }*root;
21.
22.     struct cmp {
23.     bool operator() (const Huffman_node* x, const Huffman_node* y) {
24.     return x->freq > y->freq;
25.     }
26.     };
27.
28.     unordered_map<char, int>cnt; //cnt[c]: 字符 c 在待压缩文件中出现的次数
29.     unordered_map<char, string>ans; //ans[c]: 字符 c 对应的 01 串
30.     priority_queue<Huffman_node*, vector<Huffman_node*>, cmp>que; //构建 Huffman 时所需的对结点排序的优先队列
31.
32.     long getFileSize(const char* strFileName) //得到文件大小
33.     {
34.     std::ifstream in(strFileName, ios::binary);
35.     if (!in.is_open()) return 0;
36.
37.     in.seekg(0, std::ios_base::end);
38.     std::streampos sp = in.tellg();
39.     return sp;
40.     }
41.
42.     bool is_leaf(Huffman_node* t) {
```

```
43.     return (t->left_child == NULL) && (t->right_child == NULL);
44.   }
45.
46.   void output_tree(Huffman_node* t, char* fn) { //输出 Huffman tree 到 fn 文件中
47.     ofstream fl;
48.     if (t == root) {
49.       fl.open(fn,ios::binary); //二进制打开文件
50.       if (!fl) { //检查文件是否成功打开
51.         cout << "Error opening file" << endl;
52.         exit(1);
53.       }
54.
55.       fl.write((char*)&node_num, sizeof(int));
56.       fl.close();
57.     }
58.
59.     if (t->left_child != NULL)
60.       output_tree(t->left_child, fn);
61.     if (t->right_child != NULL)
62.       output_tree(t->right_child, fn);
63.
64.     //先遍历，再输出，保证输出一个结点时，它的左右子结点都已输出，便于解压缩时重构 Huffman tree
65.     fl.open(fn, ios::binary | ios::app);
66.     if (!fl) {
```

```
67.     cout << "Error opening file" << endl;
68.     exit(1);
69.   }
70.
71.   if (is_leaf(t)) { //叶结点：首先输出一个代表叶结点的 1，接着输出编号与叶结点对应字符
72.     bool leaf = 1;
73.     fl.write((char*)&leaf, sizeof(bool));
74.     fl.write((char*)&t->num, sizeof(int));
75.     fl.write((char*)&t->c, sizeof(char));
76.   }
77.   else { //非叶结点：首先输出一个表示非叶节点的 0, 接着输出编号与其左右子结点编号（无子结点即用 0 表示）
78.     bool leaf = 0;
79.     int child = 0;
80.
81.     fl.write((char*)&leaf, sizeof(bool));
82.     fl.write((char*)&t->num, sizeof(int));
83.     if (t->left_child == NULL)
84.       fl.write((char*)&child, sizeof(int));
85.     else
86.       fl.write((char*)&t->left_child->num, sizeof(int));
87.     if (t->right_child == NULL)
88.       fl.write((char*)&child, sizeof(int));
89.     else
```

```
90.         fl.write((char*)&t->right_child->num, sizeof(int));
91.     }
92.
93.     fl.close(); //及时关闭文件!
94. }
95.
96. void traverse_tree(Huffman_node* t) { //遍历 Huffman tree 来找到每个字符（叶结点）对应的 01 串并存进 map ans 中
97.     static string str;
98.     int len = str.length();
99.     if (t->left_child != NULL) {
100.        str = str + "0";
101.        traverse_tree(t->left_child);
102.        str.erase(len);
103.    }
104.    if (t->right_child != NULL) {
105.        str = str + "1";
106.        traverse_tree(t->right_child);
107.        str.erase(len);
108.    }
109.    if (is_leaf(t)) ans[t->c] = str;
110. }
111.
112. void delete_tree(Huffman_node * t) {
113.     if (t->left_child != NULL)
```

```
114.     delete_tree(t->left_child);
115.   if (t->right_child != NULL)
116.     delete_tree(t->right_child);
117.   delete t;
118. }
119.
120. void ZIP(char** argv) {
121.   ifstream infile(argv[1], ios::binary);
122.   if (!infile) { //检查文件是否成功打开
123.     cerr << "Error opening file" << endl;
124.     exit(1);
125.   }
126.
127.   while (!infile.eof()) { //统计所有字符出现的次数
128.     char c;
129.     //infile >> c;  注意到此种读入会忽略空格与换行!
130.     c = infile.get();
131.     if (infile.eof())break; //此句的必要性与 eof()的特性有关
132.     if (cnt.find(c) == cnt.end()) cnt[c] = 1;
133.     else cnt[c]++;
134.   }
135.
136.   infile.close();
137.
```

```
138.  unordered_map<char, int>::iterator it; //定义一个 unordered map<char,int>类型的迭代器用来遍历 cnt
139.  for (it = cnt.begin();it != cnt.end();it++) { //创建所有字符的 Huffman 结点并且加入优先队列
140.      char c = it->first;
141.      int freq = it->second;
142.      Huffman_node* nd = new Huffman_node;
143.      nd->c = c;
144.      nd->freq = freq;
145.      nd->num = ++node_num;
146.      que.push(nd);
147.  }
148.
149.  while (que.size() > 1) {
150.      Huffman_node* n1, * n2; //取出优先队列队首的两个元素
151.      n1 = que.top();
152.      que.pop();
153.      n2 = que.top();
154.      que.pop();
155.
156.      Huffman_node* nd = new Huffman_node; //创建新结点：n1 n2 的父结点
157.      nd->freq = n1->freq + n2->freq;
158.      nd->left_child = n1;
159.      nd->right_child = n2;
160.      nd->num = ++node_num;
```

```
161.
162.     que.push(nd);
163.   }
164.
165.   root = que.top(); //最后剩下的结点是根结点
166.
167.   output_tree(root, argv[2]); //输出 Huffman tree
168.   traverse_tree(root); ////遍历 Huffman tree 来找到每个字符（叶结点）对应的 01 串并存进 map ans 中
169.
170.   long long tot_ansc = 0, left_num=0; //left_ansc: 输出 unsigned char 类型的总数-1 ; left_num:原文件对应的 01 串除以 8
      得到的余数
171.   infile.open(argv[1], ios::binary);
172.   ofstream outfile(argv[2], ios::binary | ios::app);
173.
174.   while (!infile.eof()) {
175.     char c;
176.     //infile >> c;  注意到此种读入会忽略空格与换行!
177.     c = infile.get();
178.     left_num += ans[c].length();
179.     tot_ansc += left_num / 8;
180.     left_num %= 8;
181.   }
182.   outfile.write((char*)&tot_ansc, sizeof(long long));
183.   outfile.write((char*)&left_num, sizeof(long long));
184.   infile.close();
```

```
185.
186.  infile.open(argv[1], ios::binary);
187.  if (!infile) {
188.   cerr << "Can not open the input file!" << endl; // 输出错误信息并退出
189.   return;
190.  }
191.  string ans_str;
192.  while (!infile.eof()) {
193.   char c;
194.   c = infile.get();
195.   ans_str += ans[c];
196.   while (ans_str.length() >= 8) {
197.    unsigned char ansc = 0;
198.    for (int i = 0;i < 8;i++)
199.     ansc = ansc << 1 | (ans_str[i] - '0');
200.    outfile.write((char*)&ansc, sizeof(unsigned char));
201.    ans_str.erase(0, 8);
202.   }
203.  }
204.  if (ans_str.length()) {
205.   unsigned char ansc = 0;
206.   for (int i = 0;i < ans_str.length();i++)
207.    ansc = ansc << 1 | (ans_str[i] - '0');
```

```
208.     outfile.write((char*)&ansc, sizeof(unsigned char));
209.     }
210.
211.     infile.close();
212.     outfile.close();
213.     delete_tree(root); //清理动态申请的内存!
214.
215.     cerr << "文件压缩率：" << 1.0 * getFileSize(argv[2]) / getFileSize(argv[1]) * 100 << "%" << endl;
216. }
217.
218. unordered_map<int, Huffman_node*>mp2; // mp2[num]: num 对应的 Huffman 结点
219.
220. void UNZIP(char** argv) {
221.     ifstream infile(argv[1], ios::binary); // 以二进制方式打开文件
222.     ofstream outfile(argv[2], ios::binary);
223.     if ((!infile) || (!outfile)) {
224.         cerr << "Can not open the input file!" << endl; // 输出错误信息并退出
225.         return;
226.     }
227.
228.     infile.read((char*)&node_num, sizeof(int));  //开始重构 Huffman tree
229.     while (node_num--) {
230.         Huffman_node* nd = new Huffman_node;
231.         if (node_num == 0) { //最后一个是根结点!
232.             root = nd;
```

```
233.     }
234.     bool leaf;
235.     infile.read((char*)&leaf, sizeof(bool));
236.     infile.read((char*)&nd->num, sizeof(int));
237.     mp2[nd->num] = nd;
238.     if (leaf) {
239.         infile.read((char*)&nd->c, sizeof(char));
240.     }
241.     else {
242.         int n1 = 0, n2 = 0;
243.         infile.read((char*)&n1, sizeof(int));
244.         infile.read((char*)&n2, sizeof(int));
245.         if (n1 != 0)nd->left_child = mp2[n1];
246.         if (n2 != 0)nd->right_child = mp2[n2];
247.     }
248. }
249.
250. long long tot_ansc, left_num; //开始读取原文本对应的压缩编码
251. infile.read((char*)&tot_ansc, sizeof(long long));
252. infile.read((char*)&left_num, sizeof(long long));
253.
254. Huffman_node* cur = root;
255. while (tot_ansc--) {
```

```
256.     unsigned char c;
257.     infile.read((char*)&c, sizeof(unsigned char));
258.     bitset<8> bit(c);
259.
260.     for (int i = 7;i >= 0;i--) {//倒序!
261.      if (bit[i] == 0) //注意是 0 而非'0'!
262.       cur = cur->left_child;
263.      else
264.       cur = cur->right_child;
265.
266.      if (is_leaf(cur)) {
267.       outfile << cur->c;
268.       cur = root;
269.      }
270.     }
271.
272.   }
273.   if (left_num != 0) { //最后一个单独处理
274.     unsigned char c;
275.     infile.read((char*)&c, sizeof(unsigned char));
276.     bitset<8> bit(c);
277.     for (int i = 7 - left_num;i >= 0;i--) {
278.      if (bit[i] == 0)
279.       cur = cur->left_child;
```

```
280.     else
281.         cur = cur->right_child;
282.     if (is_leaf(cur)) {
283.         outfile << cur->c;
284.         cur = root;
285.     }
286.   }
287.
288.   }
289.
290.   infile.close();
291.   outfile.close();
292.   delete_tree(root);//清理动态申请的内存!
293.  }
294.
295.  int main(int argc, char** argv) {
296.   clock_t starttime, endtime;
297.   starttime = clock();
298.
299.   if (argc != 4) {
300.     cerr << "Please make sure the number of parameters is correct." << endl;
301.     return -1;
302.   }
303.   if (strcmp(argv[3], "zip") == 0) {
```

```
304.     ZIP(argv);
305.     }
306.     else if (strcmp(argv[3], "unzip") == 0) {
307.     UNZIP(argv);
308.     }
309.     else {
310.     cerr << "Unknown parameter!\nCommand list:\nzip/unzip" << endl;
311.     return -1;
312.     }
313.
314.     endtime = clock();
315.     cerr << "程序运行时间：" << 1.00 * (endtime - starttime) / CLOCKS_PER_SEC << "秒" << endl;
316.     return 0;
317. }
318. /*
319. g++ 文件压缩.cpp -o 文件压缩
320. 文件压缩.exe ser.log ser_compressed.log zip
321. 文件压缩.exe ser_compressed.log ser_decompressed.log unzip
322. */
```