1. 题目：文件压缩

使用哈夫曼编码实现对一份日志文件的无损压缩，并解压缩。
最终实现压缩比：65.38%

2. 整体设计思路

2.1.哈夫曼编码文件压缩基本原理

文件中不同字符出现的频数不同，可能一部分字符高频出现，而一些字符几乎不出现，使得原有的ascii编码字符在存储上有较多冗余。

因此根据文件中字符出现频数，重新给字符编码，为出现次数较多的字符编以较短的编码，再将文件翻译成的01串每8位转为字符输出，即得到压缩文件。

解压时还原哈夫曼树，即还原自定义的新编码，即可得到压缩前的文件。

2.2.使用的一些结构

1. 哈夫曼树的节点：使用结构体存储

   （包含参数：字符chr，该字符频数freq，左孩子指针，右孩子指针，父节点指针）

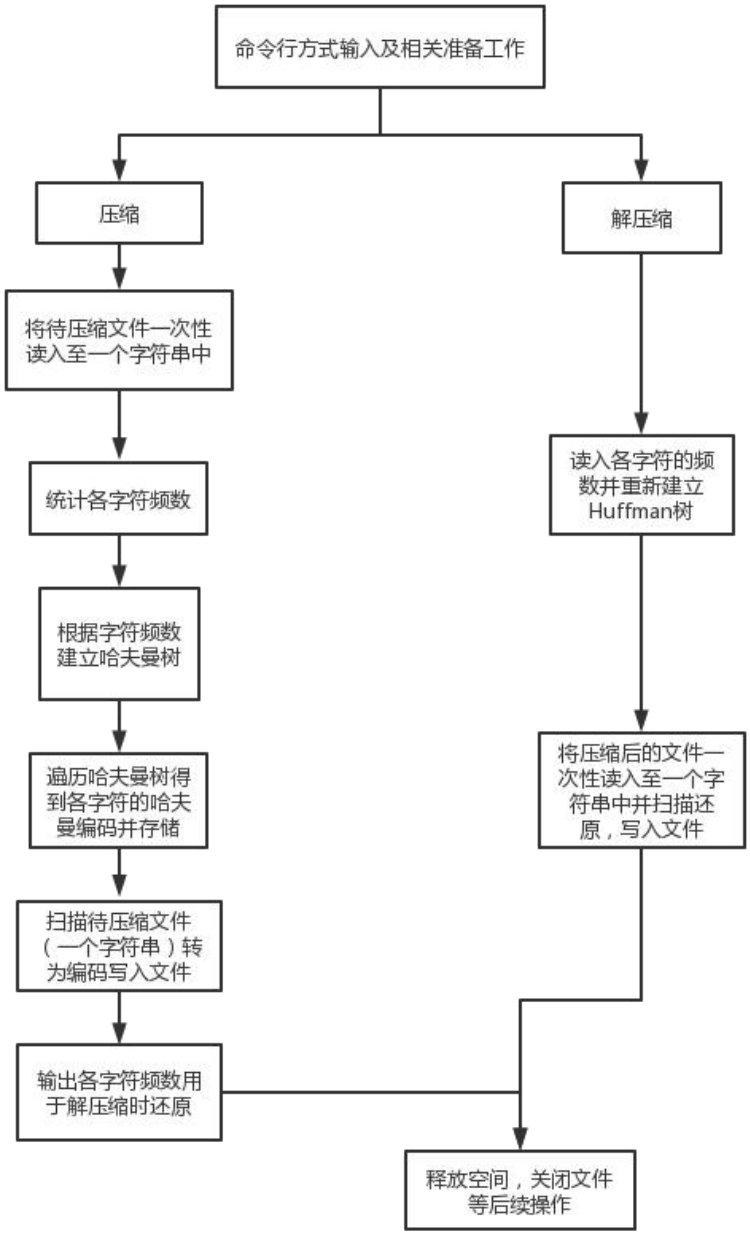
2. 哈夫曼树：优先队列维护，按各字符频数自定义排序

3. 每个字符的编码：哈希表

   字符ascii码作为下标

   哈希表内容为一个结构体，包含该字符的编码长度（int）和编码（bitset）

2.3.程序设计思路



3. 主要功能的实现

3.1. 建立哈夫曼树并得到各字符的编码

统计各字符的频数

--> 将出现过的字符插入优先队列（每个字符用一个节点存储，包含字符和字符的频数）

--> 每次从优先队列中取出堆顶的两个节点（字符频数最小的两个），合成一个新节点，即建立它们的父节点（节点权值为它们的频数之和），并将父节点插入优先队列维护

--> 建树完成后，返回根节点指针，从根节点开始递归遍历哈夫曼树：

用32位bitset记录路径即编码（往左孩子记0，往右孩子记1），到达叶子节点则将编码和编码长度存入哈希表code中，即为该叶子节点所表示的字符的编码

3.2.压缩后写入文件

以8个二进制为单位写入（用8位bitset存储）。

从头到尾扫描待压缩文件即一个字符串，以pos为字符串当前下标，bitnum为在8位bitset中当前写入到的位置，将字符转为哈夫曼编码写入bitset中，满8位就将这8位转为字符输出并清空bitset。

最后输出剩余的不足8位的。

3.3.解压缩还原哈夫曼树

压缩文件时将各字符的频数输出到文件tree中，解压缩时读入tree,调用之前的建树函数重新建立哈夫曼树。

3.4.读入压缩后的文件及还原操作

1.将文件全部读入一个字符串。

2.从头到尾扫描该字符串，对于每个字符：

先将当前字符ascii码转为01串存入8位bitset中；再依次扫描每一位编码，遇0则往左孩子，1则往右孩子，到达叶子结点就输出该路径所表示的字符，即还原了原字符。

此过程中，用tmproot表示在哈夫曼树中当前所处的节点指针，到达叶子节点后就回到根节点，以此实现整体的连贯性。

4. 调试过程碰到的问题及解决方法

4.1.如何存储编码

问题：算法涉及字符与01串的相互转化，如何更合理地存储？

解决：最终使用32位bitset存储每个字符的哈夫曼编码，用8位bitset在写入文件时临时存储，通过bitset的下标访问及.to_ulong等操作实现转化。

4.2.输入和输出的方式

问题：如何输入输出使得既能实现程序又速度较快？

解决：一开始多处使用文件指针移动及fstream::get()函数，速度较慢，后来改为一次性读入到字符串中（对于含空格换行的文件，使用了题面所提供的读入方式，设置文件指针指向开始和结束，一次性读入）。输出时使用put函数输出字符。

4.3.压缩程序运行时间过长

问题：最初的压缩过程需要25秒，时间主要花费在写入文件的过程中

解决过程：

原因1：一开始使用unordered_map储存编码，而写入文件时频繁通过它查找编码，unordered_map速度较慢浪费了时间。

解决：不用unordered_map储存编码,直接用数组储存，下标表示字符的ascii码。

原因2：一开始在写入文件时，采用256位bitset为单位写入，每次都将当前字符的编码从左往右依次存入256位bitset，每次都需要整体左移等操作，满256位在输出，整体过程繁琐速度慢。

解决：先没有动我的写入文件的方法，只是简单改为128位，64位，32位bitset位为单位写入，发现速度略微提升。后来就推翻了原先的写入方式，改为直接8位为单位写入，满8位则输出，并清空bitset,继续存储。最后运行时间提升至11秒。

5. 心得体会

1. 首先理清算法原理和整体思路，构建程序框架，并重点注意用哪些结构来存储和维护，既要做到方便实现，又要考虑速度和内存问题。

2. 在每一个细节处理上尽量优化，减少不必要的浪费，由于文件较大，一些不起眼的细节往往对整体影响很大。比如最初选择使用onordered_map存储编码就是没有必要的，完全可以用速度更快的数组直接实现。

3. 掌握得当的调试方法：本次大作业在调试过程中遇到了许多困难，有时没有理清思路被绕进去（比如该如何写入文件，用什么方式储存编码），整体思维比较混乱浪费了许多时间。这次作业也给了我一次很有益的调试经历，调试时学会分解，分模块调试找问题（比如程序运行时间长，那就先试试时间都花在哪部分了，再到相应部分去找原因），并要始终有整体观念。

附件：源程序

```
#include <fstream>
#include <iostream>
#include<cstdio>
#include <cstdlib>
#include <string>
#include<vector>
#include<queue>
#include<map>
#include <bitset>
using namespace std;

//Huffman 树的结点
struct node {
    long long freq;  //字符出现频数
    unsigned char chr;   //字符
    node* left;  //左孩子
    node* right;  //右孩子
    node* father;  //父节点
    node() {
        chr = 0;
        left = NULL;
        right = NULL;
        father = NULL;
        freq = -1;
    }
};

//优先队列维护结点，按各字符频数从小到大排序
struct cmp {
    bool operator()(const node& n1, const node& n2) {
        return n1.freq > n2.freq;
    }
};
priority_queue <node, vector<node>, cmp> que;

long long cnt[256] = { 0 };  //统计每个字符频数
string content;  //将待压缩的文件一次性读入字符串中

struct CODE {
    int len = 0;  //该字符的编码长度
    bitset<32> cd = 0;  //该字符的编码
};
struct CODE code[256];  //以字符的 ascii 码为下标，储存编码


//建立 Huffman 树
struct node* build_tree()
{
    //将出现过的字符插入优先队列
    for (int i = 0; i < 256; i++)
        if (cnt[i] > 0) {
            struct node* newnode = new(nothrow)(struct node);
            if (newnode == NULL) {
                cout << "wrong" << endl;
                exit(-1);
            }
            newnode->freq = cnt[i];
            newnode->chr = (unsigned char)i;
            que.push(*newnode);
        }
    //建立哈夫曼树
    while (!que.empty()) {
        struct node* top1 = new(nothrow)(struct node);
        struct node* top2 = new(nothrow)(struct node);
```

```
        struct node* fa = new(nothrow)(struct node);  //父节点
        if (top1 == NULL || top2 == NULL || fa == NULL) {
            cout << "wrong" << endl;
            exit(-1);
        }
        *top1 = que.top();
        que.pop();
        if (que.empty())
            return top1;
        *top2 = que.top();
        que.pop();
        top1->father = fa;
        top2->father = fa;
        fa->left = top1;
        fa->right = top2;
        fa->freq = top1->freq + top2->freq;
        que.push(*fa);
    }
    return NULL;
}

//得到各字符的编码
void get_code
(struct node* root, int len, bitset<32> x)
{
    //到达叶子结点
    if (root->left == NULL && root->right == NULL) {
        int i = (int)root->chr;
        code[i].len = len;
        code[i].cd = x;
        return;
    }
    //向左递归
    x[len] = 0;
    get_code(root->left, len + 1, x);
    //向右递归
    x[len] = 1;
    get_code(root->right, len + 1, x);
    return;
}

//回收 Huffman 树空间
void cleartree(node* root)
{
    if (!root)
        return;
    cleartree(root->left);
    cleartree(root->right);
    delete(root);
}


int main(int argc, char* argv[]) {
    //带参主函数输入格式判断
    cout << "Zipper 0.001! Author: root" << endl;
    if (argc != 4) {
        cerr << "Please make sure the number of parameters is correct." << endl;
        return -1;
    }
    if (strcmp(argv[3], "zip") && strcmp(argv[3], "unzip")) {
        cerr << "Unknown parameter!\nCommand list:\nzip/unzip" << endl;
        return -1;
    }

    //文件打开
    ifstream fin(argv[1], ios::binary); // 以二进制方式打开输入文件
    if (!fin) {  // 输出错误信息并退出
        cerr << "Can not open the input file!" << endl;
```

```
    return -1;
}
ofstream fout(argv[2], ios::binary); // 以二进制方式打开输出文件
if (!fout) {  // 输出错误信息并退出
    cerr << "Can not open the output file!" << endl;
    return -1;
}

//无损压缩与解压缩
/*****************无损压缩过程*****************/
if (strcmp(argv[3], "zip") == 0) {
    istreambuf_iterator<char> beg(fin), end; // 设置两个文件指针，指向开始和结束，以 char(一字节) 为步长
    string content(beg, end); // 将文件全部读入 string 字符串

    //统计各字符频数
    int pos = 0;
    while (pos < (int)content.length()) {
        cnt[(int)content[pos]]++;
        pos++;
    }

    //建树与编码
    bitset<32> x = 0;  //储存每个字符的 huffman 编码
    struct node* root = build_tree();   //建立 Huffman 树
    get_code(root, 0, x);   //得到各字符编码

    //以 8 个二进制位为单位写入文件
    bitset<8> n = 0;
    int bitnum = 0;  //当前二进制位存入 bitset 的位置
    pos = 0;  //原文件字符串下标
    while (pos < (int)content.length())
    {
        int asc = (int)content[pos];
        int i = code[asc].len;  //当前字符的 01 串编码长度
        for (int j = 0; j < i; j++) {
            n[bitnum] = code[asc].cd[j];
            bitnum++;
            if (bitnum >= 8) {  //写满 8 位就输出
                fout.put((char)n.to_ulong());
                n = 0;
                bitnum = 0;
            }
        }
        pos++;
    }
    if (bitnum != 0)   //输出最后剩余不足 8 位的
        fout.put((char)n.to_ulong());

    //输出各字符频数用于解压缩时还原
    ofstream outtree("tree.log", ios::out);
    if (!outtree) {
        cerr << "Can not open the output file!" << endl;
        return -1;
    }
    for (int i = 0; i < 256; i++)
        outtree << cnt[i] << " ";
    outtree.close();

    //释放 Huffman 树空间
    cleartree(root);
}

/*****************解压缩过程*****************/
else if (strcmp(argv[3], "unzip") == 0) {
    //读入各字符的频数
    ifstream intree("tree.log", ios::in);
    if (!intree) {
        cerr << "Can not open the output file!" << endl;
```

```
            return -1;
        }
        for (int i = 0; i < 256; i++)
            intree >> cnt[i];
        intree.close();

        //重新建立 Huffman 树
        struct node* root = build_tree();
        if (root == NULL) {
            cerr << "wrong" << endl;
            exit(-1);
        }

        //读入压缩后的文件及还原操作
        istreambuf_iterator<char> beg(fin), end; // 设置两个文件指针，指向开始和结束，以 char(一字节) 为步长
        string s(beg, end); // 将文件全部读入 string 字符串
        int pos = 0;
        struct node* tmproot = root;  //表示当前节点
        while (pos < (int)s.length()) {
            bitset<8> x((unsigned long)s[pos]); //将当前字符转为 01 串
            int i = 0;
            while (i < 8) {
                //遇 0 则往左孩子，1 则往右孩子
                if (x[i] == 0)
                    tmproot = tmproot->left;
                else
                    tmproot = tmproot->right;
                i++;
                //到达叶子结点就输出该路径所表示的字符
                if (tmproot->left == NULL && tmproot->right == NULL) {
                    fout.put(tmproot->chr);
                    tmproot = root;
                }
            }
            pos++;
        }

        //释放 Huffman 树空间
        cleartree(root);
    }

    // 全部操作完文件后关闭文件句柄
    fin.close();
    fout.close();

    cout << "Complete!" << endl;

    return 0;

}
```