高级程序语言设计–实验报告

日志压缩与解压

报告名称：实验报告–日志压缩与解压

班级：国 03

学号：2253156

姓名：闫浩扬

完成日期：2023 年 6 月 6 日

一．功能描述

编写了一个日志压缩与解压小程序，基于 Huffman 编码，实现文本文件的压缩与解压，。
压缩率在 60%左右，压缩与解压时间随文本大小变化。
示例 ser.log 文件压缩需要大约 8.7s，解压大约 4.9s,压缩率 65%左右。
将示例日志文件*12 后，压缩需要大约 104s，解压需要 58s 左右。

二．设计思路

（1）算法的选择

LZ77 算法的优点：

高压缩比：LZ77 算法通过利用重复片段的方式实现数据压缩，特别适合处理具有大量重复内容的数据，例如文本文件或者包含重复信息的日志文件。

快速解压缩：由于 LZ77 算法使用了字典和偏移量来表示重复片段，解压缩时可以通过简单的替换操作进行快速还原。

Huffman 算法的优点：

高效的压缩率：Huffman 算法通过将频率较高的字符编码为较短的比特串，可以实现较高的压缩比，特别适合处理字符频率分布较为不均匀的数据。

灵活性：Huffman 算法可以根据具体数据的频率分布进行自适应编码，可以针对不同数据集实现最优的压缩效果。

主要考虑 LZ77 和 Huffman 算法，观察日志文件重复内容较多，采用 LZ77 肯定可以实现较高压缩率，同时思考想要满足不同类型文本内容的处理，Huffman 根据字符频率编码也存在一定优点。查询参考资料发现可以将二者结合，LZ77 算法主要用于识别和消除重复片段，而 Huffman 编码用于将字符或符号映射为可变长度的编码。通过先使用 LZ77 算法进行初步的压缩，然后再使用 Huffman 编码对剩余的数据进行编码，可以进一步减小数据的体积。LZ77 算法提供了重复片段的信息，Huffman 编码则将这些信息以更紧凑的方式进行编码，使得压缩率进一步提高。

但是在实践中，由于时间等问题，存在许多 bug 未解决，先放弃 LZ77 算法，采用 Huffman 算法。

（2）算法的学习

Huffman 压缩的步骤：

1. 统计字符频率：首先需要对待压缩的数据进行扫描，并统计每个字符出现的频率。可以使用哈希表、数组或其他数据结构来记录字符频率。

2. 构建哈夫曼树：根据字符频率构建哈夫曼树。哈夫曼树是一种特殊的二叉树，其中每个叶节点代表一个字符，内部节点代表字符频率的合并。构建哈夫曼树的常见方法是使用优先队列来选择频率最低的两个节点，然后合并它们创建一个新的节点，直到最终构建出哈夫曼树。

3. 生成哈夫曼编码：通过遍历哈夫曼树，可以为每个字符生成对应的哈夫曼编码。在遍历过程中，左

分支表示编码位为 0，右分支表示编码位为 1。从根节点到每个叶节点的路径即为该字符的哈夫曼编码。将字符与其对应的哈夫曼编码建立映射关系，可以使用哈希表或其他数据结构存储这个映射关系。

4. 进行压缩：将原始数据替换为对应的哈夫曼编码。遍历原始数据，将每个字符替换为它的哈夫曼编码。压缩后的数据长度会变短，因为使用了较短的编码表示频率较高的字符。

5. 写入文件：将压缩后的数据写入文件。通常需要将哈夫曼编码映射表（字符和对应的编码）也写入文件，以便解压时使用。

解压的过程与压缩过程相反：

1. 读取压缩文件：从压缩文件中读取压缩后的数据和哈夫曼编码映射表。

2. 解析哈夫曼编码：根据读取到的哈夫曼编码映射表，将压缩后的数据解析为原始数据。从根节点开始，根据每个编码位的取值（0 或 1）依次遍历哈夫曼树，直到到达叶节点。将到达的叶节点对应的字符输出，并继续解析下一个编码位，直到解析完所有的编码位。

3. 写入文件：将解压后的原始数据写入文件。

但 Huffman 算法存在一些问题，如压缩和解压缩速度慢，由于哈夫曼编码是基于字符频率的构建的，构建哈夫曼树和生成编码的过程涉及频繁的排序和比较操作,这些操作可能会导致压缩和解压缩的速度相对较慢，特别是在处理大量数据时。所以考虑使用一些优化措施，例如引入 STL 容器,vector,priority_queue.

优先队列 priority_queue：

标准的队列遵从严格的先进先出,优先队列并不遵从标准的先进先出,而是对每一个数据赋予一个权值，根据当前队列权值的状态进行排序，使得权值最大（或最小）的永远排在队列的最前面，让优先级高的排在队列前面,优先出队。

priority_queue<T, Container, Compare>

A）T 就是 Type 为数据类型

b）Container 是容器类型，（Container 必须是用数组实现的容器，比如 vector

c）Compare 是比较方法，对于自定义类型，需要我们手动进行比较运算符的重载。与 sort 直接 Bool 一个函数来进行比较的简单方法不同，Compare 需要使用结构体的运算符重载完成。

（3）算法的实现

1. 创建哈夫曼树节点

```
struct Node {     //哈夫曼树节点
    char ch;      //节点字符
    int freq;     //字符频率
    Node* left;   //左子节点
    Node* right; //右子节点

    Node(char ch, int freq) : ch(ch), freq(freq), left(nullptr), right(nullptr) {};
};
```

定义一个结构体，用于存放哈夫曼树的节点，节点内包括：

char ch：表示节点所存储的字符。

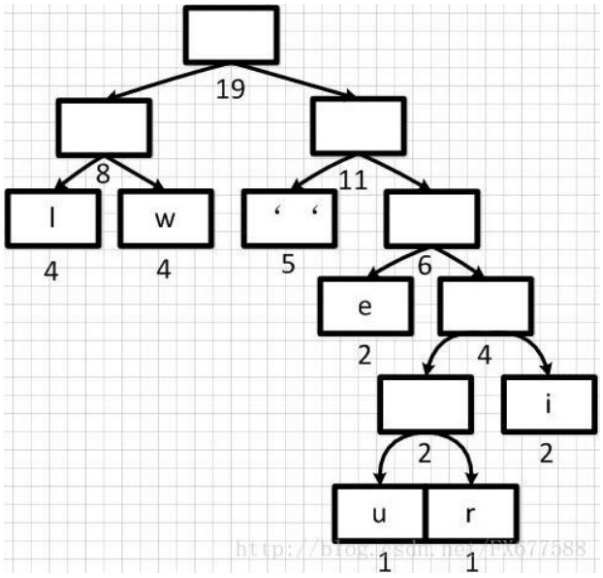
int freq：表示字符的频率或出现次数。

Node* left：指向左子节点的指针。

Node* right：指向右子节点的指针。

该结构体通过构造函数进行初始化，构造函数接受两个参数：字符 ch 和频率 freq，并将它们赋值给相应的成员变量。left 和 right 成员变量被初始化为 nullptr，表示初始时没有左子节点和右子节点。

2. 构建哈夫曼树




```
// 创建优先级队列，用于构建哈夫曼树
priority_queue<Node*, vector<Node*>, Compare> pq;
for (const auto& pair : freq) {
    pq.push(new Node(pair.first, pair.second));
}

// 构建哈夫曼树
while (pq.size() != 1) {
    Node* left = pq.top();
    pq.pop();
    Node* right = pq.top();
    pq.pop();
    int sum = left->freq + right->freq;
    Node* parent = new Node('\0', sum);
    parent->left = left;
    parent->right = right;
    pq.push(parent);
}
Node* root = pq.top(); //根
```

优先级队列（priority_queue）构建哈夫曼树。

priority_queue<Node*, vector<Node*>, Compare>，它的元素类型是 Node*，元素的优先级由 Compare 决定。Compare 是自定义的比较函数对象（functor），通过重载 operator()实现，用于比较两个节点指针之间的优先级。

优先级队列默认以最大优先级的顺序排列元素，即优先级高的元素排在队列的前面。在这里，Compare 赋予频率较小的节点具有较高的优先级。因此，在优先级队列中，频率较小的节点会排在前面，频率较大的节点会排在后面。

通过将节点指针按照频率的优先级插入优先级队列中，可以实现构建哈夫曼树的过程。优先级队列会根据节点的频率自动调整元素的顺序，保证频率较小的节点始终在前面，可以极大提高效率。这样，每次从队列中取出频率最小的两个节点，将它们合并为一个新的节点，然后将新节点插入队列。这个过程将重复执行，直到队列中只剩下一个节点，即哈夫曼树的根节点。

3. 实现哈夫曼编码

使用函数 Huffman_encode 生成哈夫曼编码，传入指向哈夫曼树根节点的指针 root 和一个存储字符编码的向

量 huffmanCode 作为参数。

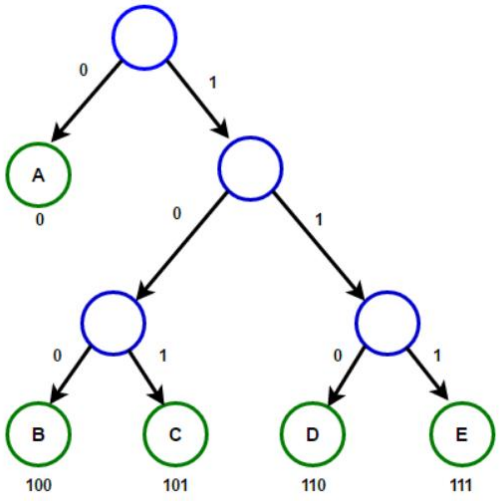
函数通过使用广度优先搜索（BFS）的方式遍历哈夫曼树的每个节点，从根节点开始，每次将节点和对应的编码（由上层节点的编码加上'0'或'1'得到）存储在队列 q 中。

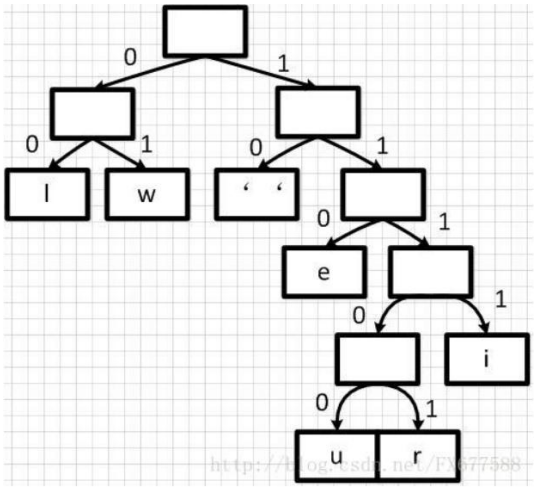
在每一次循环中，从队列中取出一个节点和它的编码。如果该节点是叶子节点（无左子节点和右子节点），则将节点的字符和对应的编码存储在 huffmanCode 向量中。这样就得到了字符和对应的哈夫曼编码的映射。

接下来，如果节点有左子节点，则将左子节点和当前编码加上 0 的结果放入队列 q 中。如果节点有右子节点，则将右子节点和当前编码加上 1 的结果放入队列 q 中。继续遍历下一层的节点，直到队列为空，完成了对整个哈夫曼树的遍历。

最终，huffmanCode 中存储了每个字符和对应的哈夫曼编码，可以通过该向量进行字符的编码和解码操作。

4. 实现哈夫曼压缩





打开输入文件和输出文件，并创建一个存储字符频率的向量 freq。读取输入文件，逐个字符计算字符频率。创建一个优先级队列 pq，根据 freq 的频率构建哈夫曼树。

使用 Huffman_encode 函数生成哈夫曼编码。该函数利用 BFS 算法遍历哈夫曼树，为每个叶子节点生成对应的哈夫曼编码，并存储在 huffmanCode 向量中。

将字符频率表写入输出文件。首先写入频率表的大小，然后逐个写入每个字符及其对应的频率。

重置输入文件指针，将其指针位置设置为文件开头。

压缩数据。逐个读取输入文件中的字符，根据 huffmanCode 中的编码表，将字符转换为对应的哈夫曼编码，并将结果保存在 compressedData 字符串中。

将压缩后的二进制数据转换为字节。根据字节对齐的要求，在 compressedData 的末尾添加足够的零位，使其长度成为 8 的倍数。然后将每个字节按照从左到右的顺序转换为对应的整数值，并将结果写入输出文件。

最后释放哈夫曼树的内存，避免内存泄漏。

5. 实现哈夫曼解压

根据压缩文件中存储的字符频率和编码信息，通过构建哈夫曼树来解码压缩的数据，从而还原原始的数据。

打开输入和输出文件流，并获取输入文件的大小。从输入文件中读取字符频率信息（字符和对应的频率）。根据字符频率信息，构建哈夫曼树。使用优先级队列来构建哈夫曼树，其中节点的权重为字符的频率，读取填充位的数量。

准备开始解压数据，创建一个指向当前节点的指针 current，初始指向根节点。从输入文件中读取每个字节，并对其进行解码。每个字节的每个位被依次解码，根据位的值，指针 current 在哈夫曼树中向左子节点或右子节点移动。当指针 current 达到叶子节点时，将叶子节点对应的字符写入输出文件，并将指针 current 重置为根节点。

同时，在解压过程中，计算解压进度，并显示进度条，以提供反馈给用户。解压完成后，根据填充位的数量，移除输出文件的填充位。释放哈夫曼树的内存空间，关闭输入和输出文件流。

6. **显示进度条**

```
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* Huffman解压开始 *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
[============>                                                          ]  16%
```

7. **Main 函数传递参数**

int argc, char **argv 用于运行时，把命令行参数传入主程序。

argc -- 命令行参数总个数，包括 可执行程序名。

argv[i] -- 第 i 个参数。

argv[0] -- 可执行程序名。

```
Usage: exe fileIN fileOUT [zip]|[unzip]
```

c

exe 文件为 argv[0] 原文件为 argv[1] 输出文件为 argv[2] zip/unzip 操作为 argv[3]

利用 strcmp(argv[3], "zip")==0 选择压缩还是解压。

三. 在实验过程中遇到的问题及解决方法

1. 优先队列出错

用了 priority_queue 容器。当时考虑到在哈夫曼中要每次挑选两个频率最小（即出现次数最小，我那个 hNode 里的 value 是出现的次数），网上查询发现了 priority_queue 容器，优先队列每次都会弹出队列中权值最高的元素，这个特性无疑是实现哈夫曼算法的最佳选择。然而因为第一次用 priority_queue 容器，结果出了不少问题，经过搜索尝试，成功解决，也学到了很多东西。

2. 哈夫曼编码错误

没有考虑到编码冲突，哈夫曼编码要求编码不具有前缀码性质，即一个字符的编码不是另一个字符编码的前缀。如果在构建编码树或生成编码时出现了冲突，即某个字符的编码是另一个字符编码的前缀，将导致解码错误。因此，在生成编码时，需要确保没有冲突出现。 解决办法：通过二进制填充来解决。通过在编码中插入额外的比特位来消除冲突，使用一种特殊的填充位，在解码过程中，当遇到填充位时，就知道该填充位不是任何字符的编码，而是用于消除冲突的标记。

在解码时，当遇到填充位时，可以根据填充位的位置和规定的标记来确定编码的结束，并继续解码下一个字符。通过使用二进制填充，哈夫曼编码可以保持前缀编码的特性，同时消除编码冲突，确保解码的准确性。

3. 解压与压缩速度慢

初步推测为频繁排序以及节点合并操作导致的时间消耗，考虑使用优先队列。

优先队列可以确保频率最低的节点始终位于队列的前部，从而使频率最低的节点能够快速地被获取和合并。在压缩阶段，可以快速地构建哈夫曼编码树，并生成字符的编码表。这样，在将原始数据映射为对应的编码时，可以通过查找编码表中的映射关系，实现高效的压缩。在解压阶段，同样可以利用优先队列，快速地根据编码表重建哈夫曼编码树。解码过程中，只需要按照编码的比特位逐步遍历哈夫曼树，直到找到对应的字符，从而实现快速的解压。

4. 文件大小的获取

在 C++中，使用 fstream 打开同一个文件并同时进行文件指针的移动可能会导致干扰。这是因为 fstream 对象维护了一个文件指针，用于跟踪当前读写位置。如果使用两个独立的 fstream 对象在同一时间同时移动文件指针，它们可能会相互干扰，导致位置的不确定性或数据损坏。干扰的具体情况取决于你进行文件指针移动的方式以及操作的顺序。

解决办法：使用一个 fstream 对象，记录下当前位置，进行指针移动获取文件大小以后返回到初始位置。

四．心得体会

1. 比较不同算法之间的优缺点，以及每个算法的适用情况。以及不同的算法之间可以进行组合来达到优势互补，实现更好的效果。

2. 学会对算法进行优化，通过搜索参考资料，发现了更高效的优化措施，如利用 STL 容器中的优先级队列进行权重排序，极大地提高了程序效率。

3. 学习了 main 函数传递参数的实际用途，可以通过命令行将参数传递给程序，来实现不同的功能以及参数传递。

4. 数据结构的选择：选择合适的数据结构对于实现算法至关重要。在哈夫曼编码中，使用可以高效地存储字符频率，而优先级队列则可以方便地构建哈夫曼树。使用了 vector<pair<char, int>> freq 来记录字符频率。每个 pair 中的第一个元素表示字符，第二个元素表示该字符出现的频率。通过遍历文件中的字符，并检查 freq 向量中是否已经存在相应字符的记录，可以更新字符的频率或添加新的字符记录。这种方式可以有效地记录每个字符的频率信息。

5. 进度显示的优化：在进行长时间的文件压缩和解压缩过程中，添加进度条可以提供用户更友好的反馈。然而，频繁刷新进度条会影响程序性能，需要适当控制进度显示的更新频率，

避免性能下降。在添加进度条后压缩与解压耗时确实增加，但是也提高了视觉反馈，让用户得知程序运行的状态。

6. 异常处理的重要性：在文件操作、内存分配和算法执行过程中，可能会出现各种异常情况。要有必要的异常处理机制，避免潜在的错误和问题。

7. 模块化设计，规范命名，提高代码的可读性和可维护性。在编写程序时，有意识地将各个功能分割，将相关功能的函数放在独立的 cpp 文件中，头文件放置函数及其他数据结构的声明，单独的 cpp 文件放置函数及数据结构的定义及实现，main 函数中调用这些函数。可维护性强，代码结构更加清晰和模块化。

五．源代码

FileHead.h

```
#pragma once
#include<iostream>
#include<fstream>
#include<ctime>
#include<iomanip>
#include<string>
#include <queue>
using namespace std;


//获取文件大小
int get_osize(ofstream& ff);
int get_isize(ifstream& ff);
int getfilesize(const string file);

//Huffman
struct Node; //声明 哈夫曼节点
struct Compare;//声明 比较左右
void Huffman_encode(Node* root, vector<pair<char, string>>& huffmanCode);              //哈夫曼编码
void freeHuffmanTree(Node* root);                                                      //释放哈夫曼
树
int huffman_compress(const std::string& input, const std::string& output,int FileSize);//哈夫曼压缩
函数
void huffman_decompress(const std::string& input, const std::string& output);          //哈夫曼解压
函数
```

Huffman.cpp

```
#include"FileHead.h"

struct Node {    //哈夫曼树节点
    char ch;     //节点字符
    int freq;    //字符频率
    Node* left;  //左子节点
    Node* right; //右子节点

    Node(char ch, int freq) : ch(ch), freq(freq), left(nullptr), right(nullptr) {};
};

struct Compare {   //用于优先队列比较
    bool operator()(Node* left, Node* right) {
        return left->freq > right->freq;
    }
};

int lastProgress = 0;
bool flag = 1;
// 进度条显示函数
void showProgressBar(double progress) {
    progress *= 100;
    if (int(progress) == lastProgress)
        flag = 0;
    else
        flag = 1;
    if (flag==0) {
        return;
    }
    else {
        int barWidth = 70;
        cout << "[";
        int pos = (int)(barWidth * progress/100);
        for (int i = 0; i < barWidth; ++i) {
            if (i < pos) cout << "=";
            else if (i == pos) cout << ">";
            else cout << " ";
        }
        cout << "] " << setw(3) << static_cast<int>(progress*1.0) << "%\r";
        cout.flush();
        lastProgress = (int)progress;
    }
}

int get_osize(ofstream& ff)
{
    streampos currentPosition = ff.tellp();
    ff.seekp(0, ios::end);
    int size = (int)ff.tellp();
    ff.seekp(currentPosition);
```

```
    return size;
}
int get_isize(ifstream& ff)
{
    streampos currentPosition = ff.tellg();
    ff.seekg(0, ios::end);
    int size = (int)ff.tellg();
    ff.seekg(currentPosition);
    return size;
}
int getfilesize(const string file)
{
    ifstream fp(file, ios::binary);
    if (fp.is_open()) {
        fp.seekg(0, ios::end);
        int fileSize = (int)fp.tellg();
        fp.close();
        return fileSize;
    }
    return -1;
}
// 生成哈夫曼编码
void Huffman_encode(Node* root, vector<pair<char, string>>& huffmanCode) {
    if (root == nullptr) {
        return;
    }

    queue<pair<Node*, string>> q;
    q.push({ root, "" });

    while (!q.empty()) {
        Node* node = q.front().first;
        string code = q.front().second;
        q.pop();

        if (!node->left && !node->right) {
            huffmanCode.push_back({ node->ch, code });
        }

        if (node->left) {
            q.push({ node->left, code + "0" });
        }
        if (node->right) {
            q.push({ node->right, code + "1" });
        }
    }
}


// 释放哈夫曼树
void freeHuffmanTree(Node* root) {
    if (root == nullptr) {
        return;
    }
```

```
    freeHuffmanTree(root->left);
    freeHuffmanTree(root->right);
    delete root;
}

// 压缩函数

int huffman_compress(const string& input, const string& output, int FileSize) {
    ifstream inFile(input, ios::binary);
    ofstream outFile(output, ios::binary);

    vector<pair<char, int>> freq;

    // 读取文件并计算字符频率
    char ch;
    while (inFile.get(ch)) {
        bool found = false;
        for (size_t i = 0; i < freq.size(); ++i) {
            if (freq[i].first == ch) {
                freq[i].second++;
                found = true;
                break;
            }
        }
        if (!found) {
            freq.push_back({ ch, 1 });
        }
    }


    // 优先队列 构建哈夫曼树
    priority_queue<Node*, vector<Node*>, Compare> pq;
    int freqSize = (int)freq.size();
    for (int i = 0; i < freqSize; ++i) {
        pq.push(new Node(freq[i].first, freq[i].second));
    }

    // 构建哈夫曼树
    while (pq.size() != 1) {
        Node* left = pq.top();
        pq.pop();
        Node* right = pq.top();
        pq.pop();
        int sum = left->freq + right->freq;
        Node* parent = new Node('\0', sum);
        parent->left = left;
        parent->right = right;
        pq.push(parent);
    }
    Node* root = pq.top(); //根

    // 生成哈夫曼编码
    vector<pair<char, string>> huffmanCode;
    Huffman_encode(root, huffmanCode);
```

```
// 写入字符频率表
outFile << freq.size() << "\n";
int freqSize0 = (int)freq.size();
for (int i = 0; i < freqSize0; ++i) {
    outFile << freq[i].first << " " << freq[i].second << "\n";
}


// 重置文件指针
inFile.clear();
inFile.seekg(0);

// 压缩数据
string compressedData;
size_t huffmanCodeSize = huffmanCode.size();
while (inFile.get(ch)) {
    for (size_t i = 0; i < huffmanCodeSize; ++i) {
        if (huffmanCode[i].first == ch) {
            compressedData += huffmanCode[i].second;
            break;
        }
    }
    // 显示进度条
    double progress = static_cast<double>(inFile.tellg()) / FileSize;
    showProgressBar(progress);
}


// 将压缩后的二进制数据转换为字节
int padding = 8 - (compressedData.length() % 8);
compressedData += string(padding, '0');
outFile << padding << "\n";

uint8_t byte = 0;
int bitCount = 0;
for (size_t i = 0; i < compressedData.length(); ++i) {
    byte <<= 1;
    if (compressedData[i] == '1') {
        byte |= 1;
    }
    ++bitCount;
    if (bitCount == 8) {
        outFile.put(byte);
        byte = 0;
        bitCount = 0;
    }
}

// 释放哈夫曼树
freeHuffmanTree(root);

int size1 = get_osize(outFile);
inFile.close();
```

```
    outFile.close();
    return size1;
}


// 解压函数
void huffman_decompress(const string& input, const string& output) {
    int FileSize = getfilesize(input);
    ifstream inFile(input, ios::binary);
    ofstream outFile(output, ios::binary);

    size_t numChars;
    inFile >> numChars;

    // 读取字符频率
    vector<pair<char, int>> freq;
    char ch;
    int f;
    for (size_t i = 0; i < numChars; ++i) {
        inFile.get(ch); // 读取空格
        inFile.get(ch);
        inFile >> f;
        freq.push_back(make_pair(ch, f));
    }

    // 优先队列 构建哈夫曼树
    priority_queue<Node*, vector<Node*>, Compare> pq;

    for (size_t i = 0; i < freq.size(); ++i) {
        pq.push(new Node(freq[i].first, freq[i].second));
    }

     // 构建哈夫曼树
    while (pq.size() != 1) {
        Node* left = pq.top();
        pq.pop();
        Node* right = pq.top();
        pq.pop();

        int sum = left->freq + right->freq;
        Node* parent = new Node('\0', sum);
        parent->left = left;
        parent->right = right;

        pq.push(parent);
    }

    Node* root = pq.top();

    // 读取填充位
    int padding;
    inFile >> padding;
    inFile.get(ch); // 读取换行符
```

```
        // 解压数据
        Node* current = root;
        while (inFile.get(ch)) {
            uint8_t bits = static_cast<uint8_t>(ch);
            for (int i = 0; i < 8; ++i) {
                current = (bits & (1 << (7 - i))) ? current->right : current->left;
                if (!current->left && !current->right) {
                    outFile.put(current->ch);
                    current = root;
                }
            }
            // 计算进度并显示进度条
            double progress = static_cast<double>(inFile.tellg()) / FileSize;
            showProgressBar(progress);
        }

        // 移除填充位
        outFile.seekp(-padding, ios::end);
        outFile.write("", 1);

        // 释放哈夫曼树
        freeHuffmanTree(root);

        inFile.close();
        outFile.close();
    }
```

main.cpp

```
    #include"FileHead.h"

    int main(int argc, char* argv[]) {
        cout << "+-------------------------------------------------------------------------------+"
<< endl;
        cout << "|                                                                               |"
<< endl;
        cout << "|                          Huffman 解压缩小程序 V1.0          Author : @Root      |"
<< endl;
        cout << "|                                                                               |"
<< endl;
        cout << "|                  Usage: exe fileIN fileOUT [zip]|[unzip]                      |" <<
endl;
        cout << "+-------------------------------------------------------------------------------+"
<< endl;
        clock_t ta, tb;
        int size0, size1;
        double t;
        if (argc != 4) {
            std::cerr << "用法说明: " << argv[0] << " <输入文件> <输出文件> [zip|unzip]" << std::endl;
            return 1;
        }
```

```
        if (strcmp(argv[3], "zip")==0) {
            cout << "-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* Huffman 压缩开始 *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-" <<
endl;
            ta = clock();
            size0 = getfilesize(argv[1]);
            size1 = huffman_compress(argv[1], argv[2],size0);
            tb = clock();
            cout << endl << endl;
            cout << "原始文件大小为：" << setw(10) << setiosflags(ios::right) << size0 << "字节" << endl;
            cout << "压缩文件大小为：" << setw(10) << setiosflags(ios::right) << size1 << "字节" << endl;
            cout << "文件压缩比率为：" << setw(11) << setiosflags(ios::right) << 100.00 * size1 /
(double)size0 << "%" << endl;
            cout << "压缩用时：" << setw(16) << setiosflags(ios::right) << (t = ((double)tb - (double)ta))
/ CLOCKS_PER_SEC << "秒" << endl << endl;
            cout << "-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* Huffman 压缩成功 *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-" <<
endl;
        }
        else if (strcmp(argv[3], "unzip")==0) {
            cout << "-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* Huffman 解压开始 *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-" <<
endl;
            ta = clock();
            huffman_decompress(argv[1], argv[2]);
            tb = clock();
            size0 = getfilesize(argv[1]);
            size1 = getfilesize(argv[2]);
            cout << endl <<endl << "压缩文件大小为：" << setw(10) << setiosflags(ios::right) << size0 <<
"字节" << endl;
            cout << "解压文件大小为：" << setw(10) << setiosflags(ios::right) << size1 << "字节" << endl;
            cout << "解压用时：" << setw(16) << setiosflags(ios::right) << (t = ((double)tb - (double)ta))
/ CLOCKS_PER_SEC << "秒" << endl << endl;
            cout << "-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* Huffman 解压完成 *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-" <<
endl;
        }
        else {
            std::cerr << "非法用法: " << argv[3] << std::endl;
            std::cerr << "用法说明: " << argv[0] << " <输入文件> <输出文件> [zip|unzip]" << std::endl;
            return 1;
        }

        return 0;
    }
```