数据结构与算法设计（C++描述）

专题报告

实验名称： **压缩软件**

	姓名	学号	专业
小组成员	[redacted]	[redacted]	电子信息工程（徐特立英才班）
	[redacted]	[redacted]	电子信息工程（徐特立英才班）
	[redacted]	[redacted]	电子信息工程（徐特立英才班）
	[redacted]	[redacted]	电子信息工程（徐特立英才班）
	[redacted]	[redacted]	电子信息工程（徐特立英才班）
任课教师：	[redacted]	备注：无	

版本

版本号	描述/改动	日期	责任人
1	初版	[redacted]	[redacted]
2	进一步完善与修改	[redacted]	[redacted]

一、基本情况

题目内容	设计并实现一款基于哈夫曼编码的文件压缩软件，实现文件的无损压缩与解压。通过分析文件中的字符频率，建立字符集频率表并由此构建哈夫曼树对源文件进行编码，将源文件压缩为更小的二进制文件；同时支持将压缩后的文件解压缩为原始文件。
题目要求及约束条件	我们设计的压缩软件可以从源文件中读取数据，并统计各字符出现的频率生成字符集频率表。 基于字符集频率表构建哈夫曼树对源文件进行编码，生成.huff 格式文件。 利用构建的哈夫曼树对.huff 格式文件进行解压，还原为解压文件。 设计的压缩软件支持输入源文件为任意格式，并且解压文件与压缩文件相比无失真。 压缩文件需要自定义文件头存储关键信息（存储频率表、编码位数等元数据），解压时需读取文件头重建哈夫曼树。
成员分工	[redacted]

二、设计与实现

1、设计思想

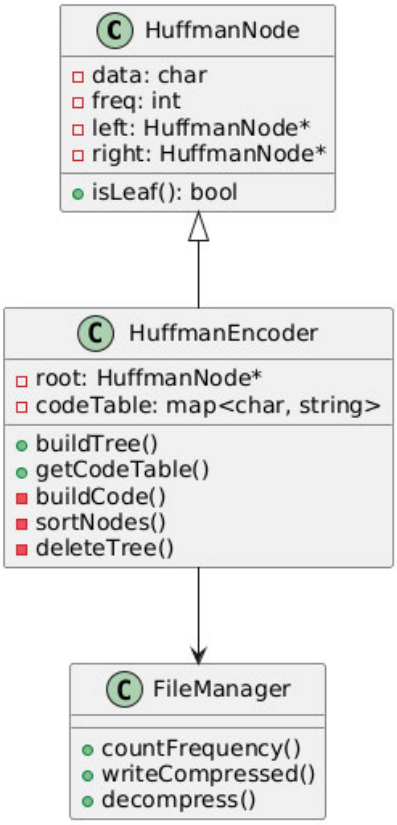
在这个信息高速发展的时代，我们需要存储并传输大量的文件。在占用存储空间较大的文件中通常有大量冗余，因而我们可以对其进行编码实现文件压缩。在我们设计的压缩软件中，我们通过哈夫曼编码实现对任意格式源文件的压缩和解压。在压缩过程中首先从源文件中读取数据并建立字符集频率表，利用建立的字符集频率表完成哈夫曼树的构建并对源文件进行编码，最后将字符种类数和编码表写入文件头，压缩数据按照每八位码符号一字节写入.huff 文件正文实现对源文件的压缩。在解压过程中，首先通过读取文件头重建解码映射表，再根据前缀码的特性通过持续匹配编码完成对压缩文件的解压。我们设计的压缩软件还可以对源文件为空，指针异常等情况进行处理，更能满足实际情况的需求。

2、类结构

1. 哈夫曼树节点(HuffmanNode)类：表示哈夫曼树的单个节点，存储该节点对应的字符数据、出现频率和指向其左右孩子节点的指针，并提供了叶子结点的判断方法。
2. 哈夫曼编码器(HuffmanEncoder)类：构建哈夫曼树并对每个叶子节点的字符数据进行哈夫曼编码。存储构造的哈夫曼树的根节点以及编码表，提供了获取源文件的编码表，构建哈夫曼树并进行哈夫曼编码的方法。
3. 文件管理(FileManager)类：封装了统计字符频率、写入压缩文件、解压缩文件等函数，

使代码的逻辑结构更加清晰。

这些类结构的使用提升了代码的封装性，有效保护了私有数据成员中的核心数据。并且具有严格的内存管理机制，有效防止内存泄漏并提高内存利用效率。

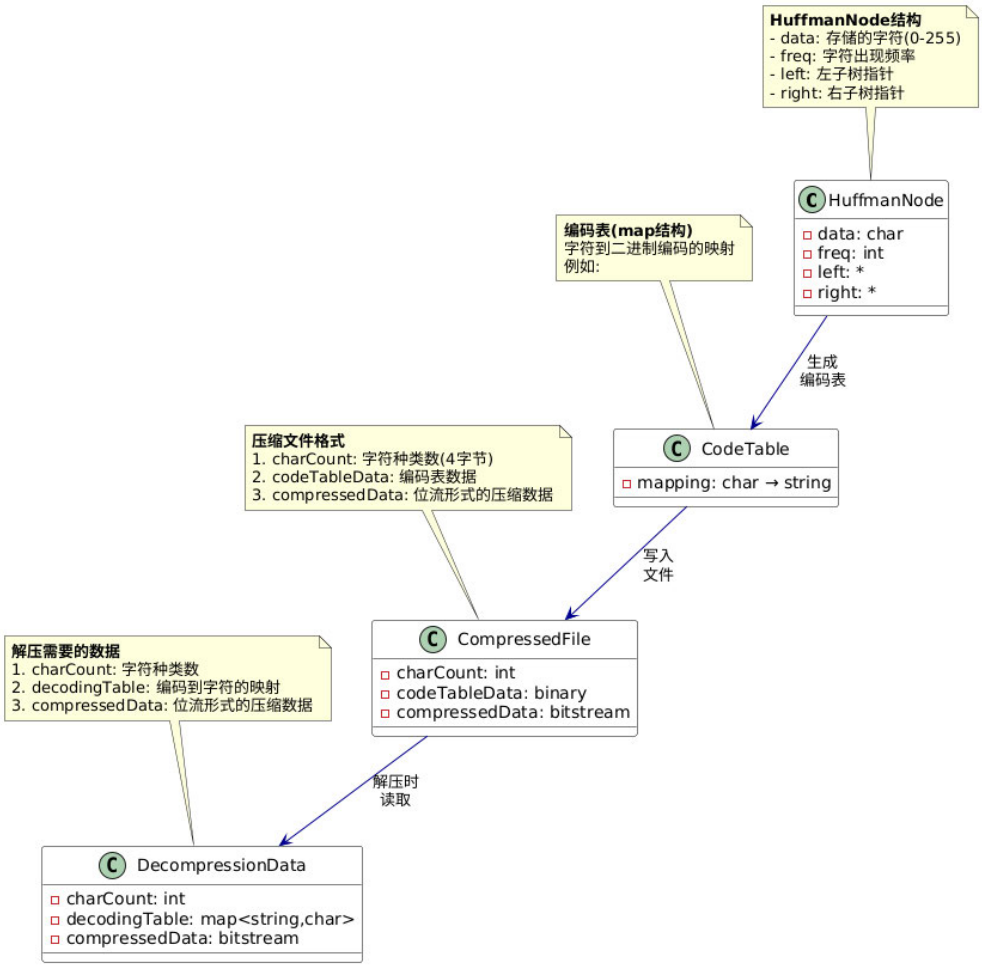


3、主要数据结构

1. 哈夫曼树：程序通过构建的字符集频率表建立哈夫曼树。哈夫曼树通过每个节点的左右孩子指针构成二叉树，并且每个叶子结点存储源文件中的字符数据及其出现频率。哈夫曼树是带权路径长度最小的二叉树，利用构建的哈夫曼树我们可以对源文件进行哈夫曼编码。
2. map 容器：map 是 STL（中文标准模板库）的一个关联容器，可以将任何基本类型映射到任何基本类型。在本程序中通过 map 容器构建了编码表、频率表和解码表，分别实现了源文件字符到哈夫曼编码字符串、源文件字符到出现频率以及哈夫曼编码字符串到源文件字符的映射。
3. 节点指针数组：在我们设计的压缩软件中，我们处理的源文件不同字符数一般不超过 256 个，因此构建的哈夫曼树需要 511 个节点。我们使用指针数组来存放指向这些节点的指针，使用这种顺序存储结构也有利于接下来哈夫曼树的构建。

使用这些数据结构有以下优点：

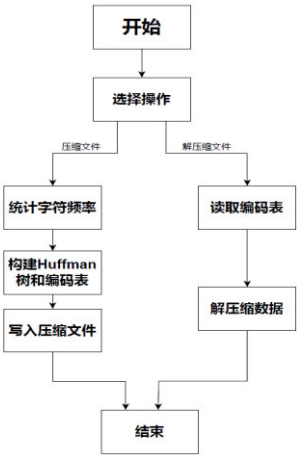
1. 由哈夫曼树的结构可以直接使用递归的方法构造最优前缀码，并且采用二叉链表的存储方式使内存利用更高效。
2. map 容器按键值对的方式建立映射关系，使代码逻辑更加清晰；并且键从小到大的顺序自动排序，有利于程序的调试和测试。map 容器无需预先分配内存，只存储实际出现的字符，有效提升内存利用率，并且时间复杂度为 O(log n)，查询和更新高效。
3. 节点指针数组的使用避免了动态内存分配，有利于提升访问速度，并且更易于编程实现。



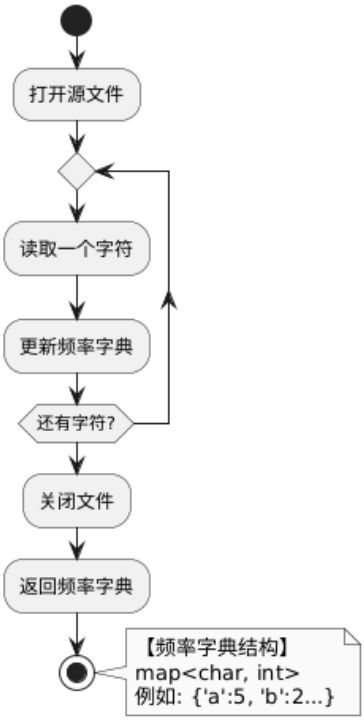
4、算法设计

1. 主程序流程：首先进入压缩软件的开始界面，显示提示信息。用户通过键盘输入数字 1 进行压缩文件，输入 2 则进行解压缩，若输入错误会提示用户重新输入。当压缩文件时，用户输入含有后缀名的源文件，并输入压缩后的.huff 文件的名称，若文件打开失败会提示错误信息。当源文件正常打开后程序会读取文件中的字符并建立字符集频率表，接下来程序根据建立的字符集频率表构建哈夫曼树并对源文件进行哈夫曼编码，最后将字符种类数、编码表和压缩数据写入.huff 文件完成压缩。当解压文件时，用户输入.huff 文件并输入含有后缀名的解压缩后的文件名称，若文件打开失败会提示错误信息。当.huff 文件正常打开后，程序会重建解码表并由此将.huff 文件中的编码还原为源文件中的字符，实现解压缩操作。

主函数流程图如下：

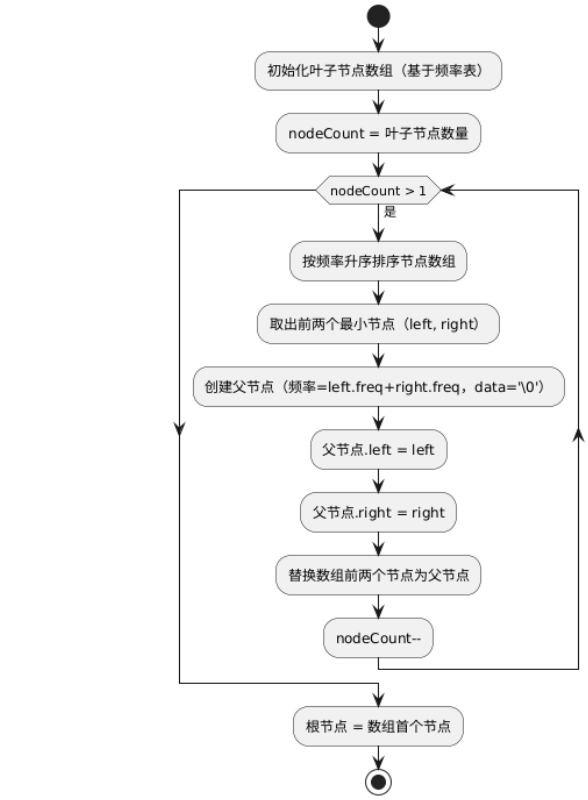


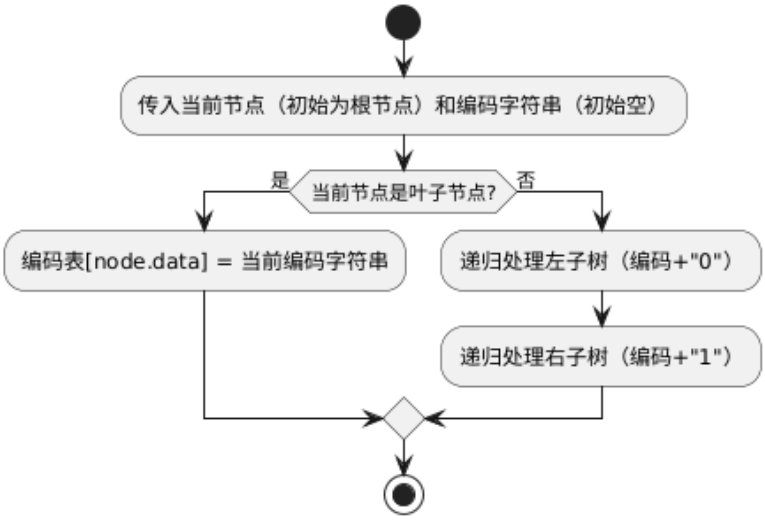
2. 统计字符频率：通过 countFrequency 函数实现，这里使用 map 容器实现字符到出现频率的映射。打开源文件后，程序逐一读取字符，并将该字符所对应的频率加一，读取完毕后关闭文件。



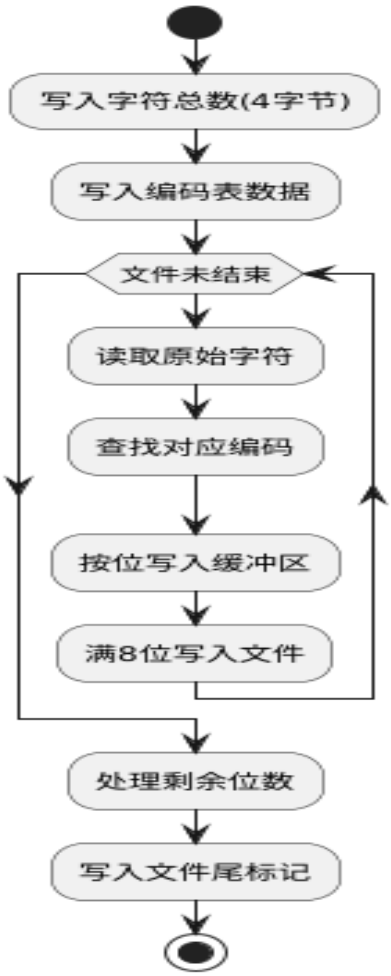
3. 字符哈夫曼编码生成：通过 buildCode 数实现。该算法是在哈夫曼树构建完成后，采取从根节点开始的深度优先遍历，使用递归的方法实现。当根节点为空时结束递归，当根节点不为空时，对于左子树路径编码追加“0”后向下递归，对于右子树路径编码追加“1”后向下递归。当到达叶子结点时，将字符和对应的路径编码存入编码表。
4. 构建哈夫曼树：首先根据建立的字符集频率表，为每个字符创建叶子结点，形成只有根结点的二叉树，令其权值为出现频率，得到森林。在森林中选取两棵根结点权值最小的树分别作为左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和。然后在森林中删除这两棵树，同时将新得到的二叉树加入森林中。重复上述步骤，直到森林中只剩一棵树为止，这棵树即为所构建的哈夫曼树。

构建 Huffman 树的流程图如下：

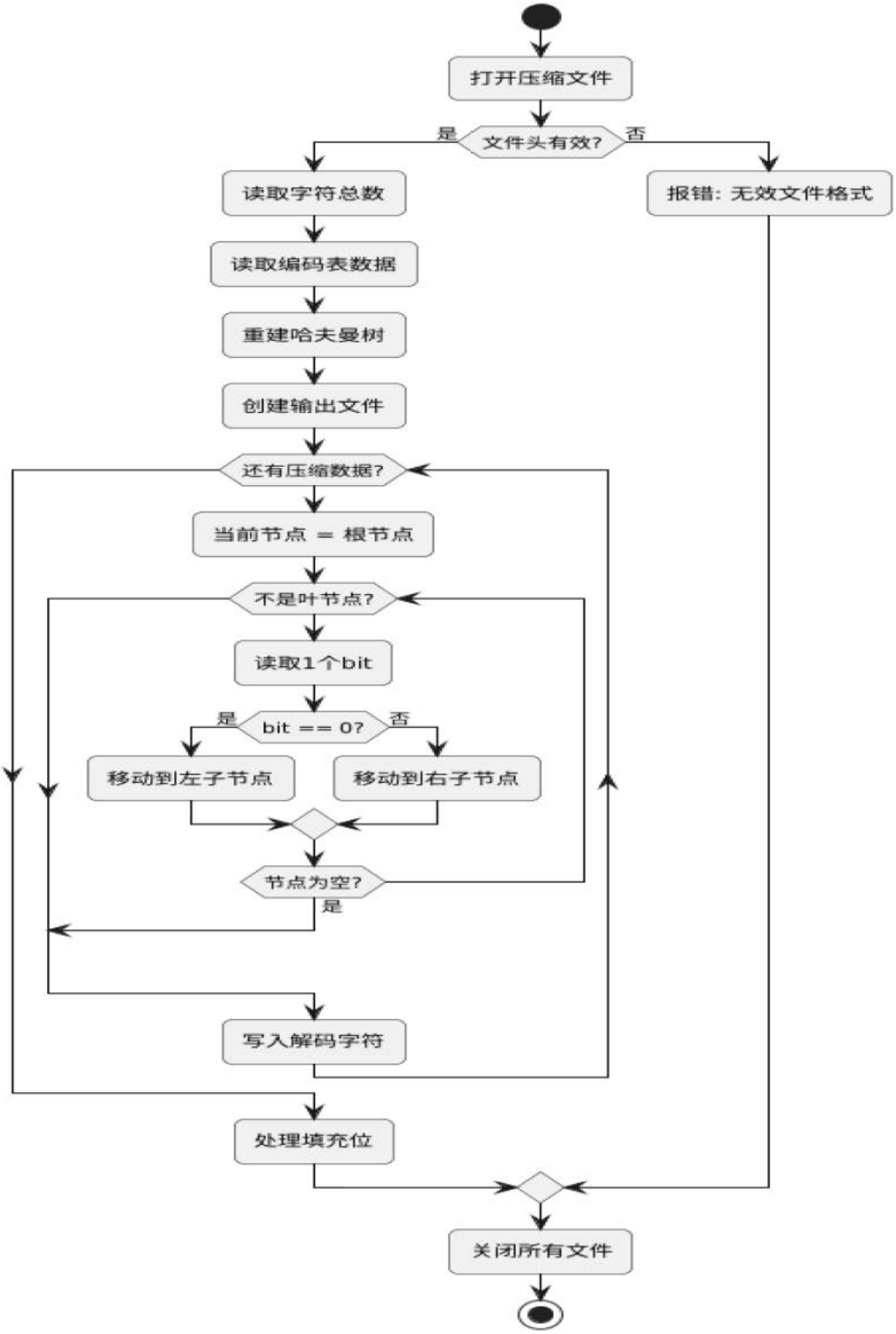




5. 写入压缩文件：通过 writeCompressed 函数实现。首先以二进制模式打开源文件，若为空文件自动返回。接下来将字符种类数和编码表写入头文件，方便以后得解压缩操作。然后逐个读取源文件中的字符信息并查找其哈夫曼编码，在存储时使用位缓冲机制：首先建立 8 位缓冲区，然后遍历编码的每个位并左移缓冲区将当前位添加进来。当缓冲区满八位时就写入文件并重置缓冲区，不满八位时补零后写入文件，压缩完成后关闭源文件。
6. 实现文件压缩的原理：通过对字符使用 0 和 1 码元进行哈夫曼编码，使出现概率大的字符对应短码，出现概率小的字符对应长码。在完成编码生成压缩文件时，将每个字符对应的二进制哈夫曼编码按照八位一个字节写入文件，这样得到的压缩文件字节数比源文件要少。



7. 解压文件：通过 decompress 函数实现。首先以二进制模式打开.huff 文件，若为空文件自动返回。接下来读取文件头的内容，重建解码表，得到编码字符串到原字符的映射。在解压缩时，逐字节读取压缩数据，根据哈夫曼编码是前缀码的特性，进行前缀匹配解码，匹配成功则写入字符，若匹配失败则重新读取压缩数据后再进行匹配。重复以上操作，直至完成解压缩为止。



5、核心代码展示

1.Huffman 树节点类

```
class HuffmanNode {
private:
    char data;        // 存储字符（0 - 255）
```

```
    int freq;         // 字符出现频率
    HuffmanNode* left;   // 左子节点指针
    HuffmanNode* right;  // 右子节点指针

public:
    friend class HuffmanEncoder; // 友元类以便访问私有成员

    // 构造函数
    HuffmanNode(char d, int f)
        : data(d), freq(f), left(nullptr), right(nullptr) {}
    // 判断是否为叶子节点（没有子节点）
    bool isLeaf() const {
        return !left && !right;
    }
};
```

代码定义了 Huffman 树的节点结构，包含字符数据、字符出现频率以及左右子节点指针。isLeaf 方法用于判断节点是否为叶子节点。

2.哈夫曼树构建

```
void buildTree(const map<char, int>& freqMap) {
    const int MAX_NODES = 512; // 最大节点数（256 个字符需要 511 个节点）
    HuffmanNode* nodes[MAX_NODES] = {nullptr}; // 初始化指针数组
    int nodeCount = 0;
    // 如果频率表为空，清空相关数据
    if (freqMap.empty()) {
        root = nullptr;
        codeTable.clear();
        return;
    }
    // 第一步：创建所有叶子节点
    for (map<char, int>::const_iterator it = freqMap.begin();
         it != freqMap.end(); ++it) {
        nodes[nodeCount++] = new HuffmanNode(it->first, it->second);
    }
    // 第二步：构建 Huffman 树
    while (nodeCount > 1) {
        // 按频率排序节点
        sortNodes(nodes, nodeCount);
        // 取两个最小节点
        HuffmanNode* left = nodes[0];
        HuffmanNode* right = nodes[1];
        // 创建新的父节点
        HuffmanNode* parent = new HuffmanNode('\0', left->freq + right->freq);
        parent->left = left;
        parent->right = right;
        // 更新节点数组
```

```
            nodes[0] = parent;  // 新节点占据第一个位置
            for (int i = 1; i < nodeCount-1; ++i) {
                nodes[i] = nodes[i+1]; // 节点前移
            }
            nodeCount--; // 总节点数减1
        }
        root = nodes[0];  // 最后一个节点为根节点
        buildCode(root, ""); // 生成编码表
    }
```

根据字符频率表构建 Huffman 树，并生成对应的 Huffman 编码表。首先创建所有叶子节点，然后不断合并频率最小的两个节点，直到只剩下一个根节点。最后调用 buildCode 方法生成编码表。

3.编码生成

```
void buildCode(HuffmanNode* node, string code) {
    if (!node) return;
    if (node->isLeaf()) {
        codeTable[node->data] = code;  // 存储叶节点编码
        return;
    }
    // 左0 右1 的编码规则
    buildCode(node->left, code + "0");
    buildCode(node->right, code + "1");
}
```

递归遍历哈夫曼树，为每个字符生成二进制编码（左路径为 0，右路径为 1）。

4. 字符频率计算

```
static map<char, int> countFrequency(const string& filename) {
    ifstream fin(filename, ios::binary);
    if (!fin) {
        cerr << "无法打开文件 " << filename << endl;
        exit(1);
    }
    map<char, int> freqMap;
    char ch;
    while (fin.get(ch)) {  // 逐个字符读取
        freqMap[ch]++;     // 统计频率
    }
    fin.close();
    return freqMap;
}
```

统计输入文件中每个字符的出现频率，返回一个字符到频率的映射表。通过二进制模式打开文件，逐个字符读取并更新频率表。

5.压缩文件写入

```
static void writeCompressed(const string& inputFile,
```

```
                          const string& outputFile,
                          const map<char, string>& codeTable) {
    ifstream fin(inputFile, ios::binary);
    ofstream fout(outputFile, ios::binary);
    // 写入字符数量（4 字节）
    int charCount = codeTable.size();
    fout.write(reinterpret_cast<const char*>(&charCount), sizeof(int));
    // 如果文件为空，关闭文件并返回
    if (charCount == 0) {
        fin.close();
        fout.close();
        return;
    }
    // 写入字符 + 编码长度 + 编码数据
    for (map<char, string>::const_iterator it = codeTable.begin();
         it != codeTable.end(); ++it) {
        fout.put(it->first);          // 写入字符（1 字节）
        unsigned char len = it->second.length();
        fout.put(len);                // 写入编码长度（1 字节）
        fout.write(it->second.c_str(), len); // 写入编码
    }
    // 压缩数据
    unsigned char buffer = 0;  // 位缓冲区
    int bitCount = 0;          // 当前缓冲区位数
    char ch;

    while (fin.get(ch)) {
        const string& code = codeTable.at(ch);
        for (size_t i = 0; i < code.size(); ++i) {
            buffer = (buffer << 1) | (code[i] - '0'); // 填充缓冲区
            if (++bitCount == 8) {  // 缓冲区满
                fout.put(buffer);
                buffer = bitCount = 0;
            }
        }
    }
    // 处理剩余位数，不足 8 位补 0
    if (bitCount > 0) {
        buffer <<= (8 - bitCount); // 填充剩余位
        fout.put(buffer);
    }
    fin.close();
    fout.close();
}
```

将输入文件进行压缩并写入输出文件。首先写入字符数量和编码表信息，然后将输入文件中的字符转换为对应的 Huffman 编码，按位填充到缓冲区，按照每 8 位编码一个字节写入输出

文件。最后处理剩余的不足 8 位的编码，补 0 后写入文件，由此实现对源文件的压缩。

6.文件解压算法

```
static void decompress(const string& inputFile, const string& outputFile) {
    ifstream fin(inputFile, ios::binary);
    ofstream fout(outputFile, ios::binary);
    // 读取字符数量
    int charCount;
    fin.read(reinterpret_cast<char*>(&charCount), sizeof(int));
    // 如果文件为空，关闭文件并返回
    if (charCount == 0) {
        fin.close();
        fout.close();
        return;
    }
    // 解析编码表，存储到解码映射中
    map<string, char> decodeMap;
    for (int i = 0; i < charCount; ++i) {
        char data;
        unsigned char codeLen;
        fin.get(data);
        fin.read(reinterpret_cast<char*>(&codeLen), 1);

        string code;
        code.resize(codeLen);
        fin.read(&code[0], codeLen);  // 读取编码

        decodeMap[code] = data;  // 建立映射关系
    }
    // 解压缩数据
    string bitStr;    // 累积的二进制字符串
    char byte;
    while (fin.get(byte)) {
        // 将字节转换为8 位二进制字符串
        for (int i = 7; i >= 0; --i) {
            bitStr += (byte & (1 << i)) ? '1' : '0';
        }
        // 匹配编码并写入输出文件
        bool found;
        do {
            found = false;
            for (map<string, char>::const_iterator it = decodeMap.begin();
                 it != decodeMap.end(); ++it) {
                if (bitStr.substr(0, it->first.size()) == it->first) {
                    fout.put(it->second);  // 写入解码后的字符
                    bitStr = bitStr.substr(it->first.size()); // 移除已匹配的编码
```

```
                found = true;
                break;
            }
        }
    } while (found && !bitStr.empty());
}
fin.close();
fout.close();
}
```

首先读取字符数量和编码表信息，建立解码映射。然后逐字节读取压缩文件，将字节转换为二进制字符串，不断匹配解码映射中的编码，将解码后的字符写入输出文件。

三、测试与结论

1、测试环境与数据

环境：Windows 11，Visual Studio 2022
数据：压缩对象为图片“1 组-灰度图.bmp”和英文文本“novel1.txt”。

2、测试用例

1. 打开程序界面：

图 1：程序开始界面

2. 压缩文件，压缩后数据存储至”1 组-灰度图.huff”。

图 2：压缩文件

3. 对 1 组-灰度图.huff 进行解压缩，得到“2 组-灰度图.bmp”。

图 3：解压文件

3、测试结论

源文件大小为 8254 字节，压缩后变为 6844 字节，压缩率 1.206，说明我们设计的压缩软件可以有效压缩图片。

源文件和解压文件的对比如下图：

图 4：源文件（左）解压文件（右）

按照同样的步骤对英文文本“novel1.txt”进行压缩，源文件大小为 523110 字节，压缩后的“novel1.huff”文件变为 275269 字节，压缩比为 1.900，说明我们设计的压缩软件可以有效压缩英文文本，并且源文件和解压文件的部分文本内容对比如下图：

He was an old man who fished alone in a skiff in the Gulf Stream and he had gone eighty-four days now without taking a fish. In the first forty days a boy had been with him. But after forty days without a fish the boy′ s parents had told him that the old man was now definitely and finally salao, which is the worst form of unlucky, and the boy had gone at their orders in another boat which caught three good fish the first week. It made the boy sad to see the old man come in each day with his skiff empty and he always went down to help him carry either the coiled lines or the gaff and harpoon and the sail that was furled around the mast. The sail was patched with flour sacks and, furled, it looked like the flag of permanent defeat.

He was an old man who fished alone in a skiff in the Gulf Stream and he had gone eighty-four days now without taking a fish. In the first forty days a boy had been with him. But after forty days without a fish the boy′ s parents had told him that the old man was now definitely and finally salao, which is the worst form of unlucky, and the boy had gone at their orders in another boat which caught three good fish the first week. It made the boy sad to see the old man come in each day with his skiff empty and he always went down to help him carry either the coiled lines or the gaff and harpoon and the sail that was furled around the mast. The sail was patched with flour sacks and, furled, it looked like the flag of permanent defeat.

图 4：源文本（左）解压文本（右）

通过以上的对比，源文件在经过压缩解压后不会发生失真，符合压缩软件的设计要求。

四、总结与思考

1、题目难点要点

1. 哈夫曼树的构建：读取文件得到字符集频率表并建立哈夫曼树，由此得到哈夫曼编码。
2. 文件的读取和写入：压缩文件头结构的设计，通过合理的编码存储方式实现文件压缩。
3. 内存资源的管理：树节点递归与释放，二叉链表中指针的精确管理，防止内存泄漏。

2、本组工作特点

1. 哈夫曼编码压缩文件的实现：实现了完整的哈夫曼压缩解压流程，包括频率统计、树构建、编码生成、压缩和解压。
2. Map 容器的应用：本程序采用 map 容器建立了编码表、频率表和解码表,构建了各种映射关系，便于数据的快速读取、查找和修改等操作。
3. 文件格式设计及其读写操作：通过在压缩文件中添加头文件部分，使得解压时无需依赖外部存储的编码表等额外信息。采用位缓冲机制处理变长编码写入，提高压缩效率。

3、本组改进方向

1. 对源文件字符种类个数有限制：本程序最多只能处理含有 256 个字符的文件，对于字符个数较多的大文件无法正常压缩，可以进一步改善使其支持更多字符种类。
2. 压缩率进一步优化：支持动态哈夫曼编码，并增加 LZ77 预处理。
3. 解压缩性能优化：解压缩使用前缀匹配法效率较低，可以考虑重构哈夫曼树进行解码。