DS2024_PJ中期文档

姓名：丘俊

学号：23307130289

项目进度简述

1. **已完成的项目需求**：搭好核心框架，解决了哈夫曼树的构造和序列化、反序列化等

问题，实现了单文件压缩与解压。
2. **未完成的项目需求**：实现多文件压缩、设置压缩密码、用户交互。
3. **接下来的工期计划**：12.1前实现多文件压缩、设置压缩密码、用户交互。

中期文档问题

(1) 哈夫曼编码算法是否总能保证最优压缩？如果不一定，请举例说明在哪些情况下它可能不是最优的。

否。一般来说，哈夫曼编码能够确保生成的编码具有最小的平均码字长度，从而实现最优的压缩效果。从其原理可知，其最优性依赖于以下几个假设：

1. **符号独立性**： 每个符号的出现是独立的。
2. **符号概率已知**： 符号的出现概率是已知且精确的。
3. **符号频率事先确定**： 所有符号的出现频率在编码过程中是静态且已知的。

于是，如果文本中某个字符总是附在另一字符的后面，我们可以通过特殊处理优化此种情况，使得哈夫曼编码不是最优的。又如，从自然语言处理角度而言，英语中的字母出现并非完全独立，字母“q”几乎总是跟随“u”，利用上下文关系的编码方法（如上下文自适应编码）能够更有效地压缩数据，而哈夫曼编码由于忽略了上下文信息，无法达到相同的压缩效率。

(2) 如何根据文件的字节流，构建哈夫曼树？

遍历文件的字节流，使用'map<char, int>'记录每个字符出现的次数，计算相应频率，构建哈夫曼树。

(3) 对于一组频率已经给定的字符，如果哈夫曼树已经构建完成，是否能快速找到一个新的字符的编码，而不重新构建整棵树？如何实现？

不能不重新构建整棵树。由于新增字符后，所有字符的频率都会被改变，必须重建整棵树。（如果树中不存储字符频率而是存储字符频数的话，虽然大部分字符的频数不改变，但仍然有必要重建整棵树，因为每一个非叶结点都存储了以它为根的树下所有叶结点的字符的频数之和，于是大部分非叶结点都需要更新，且找不到有效低开销方法对其进行部分更新，不如重建。）

(4) 假设对一个文件进行了哈夫曼编码压缩，如何通过编码表和压缩后的字节流，准确还原出原始文件内容？

建立一个`map<string, char>`来存储编码表，对压缩后的字节流进行遍历，先选取第一个字符加入字符串`s`，如果`map.has(s)`，则输出`map.get(s)`，否则，不断选取下一个字符加入`s`，直

到`map.has(s)`为止。输出了`map.get(s)`后，将`s`重置为`""`，不断读取字节流后面的内容，直到读取到文件尾。

(5) 构建哈夫曼树的过程中，如何每次高效、便捷地选出出现频率最低的两个节点?

维护一个优先队列，优先队列可以在 **O(log n)** 的时间复杂度内找到并删除最小的元素。

(6) 如何完成文件夹的压缩并保留内部文件名等信息的一致性?

在压缩包头部记录信息来保存这些内容，下面是一个可能的例子。

```
{
  "directories": ["root", "root/subdir1", "root/subdir2"],
  "files": [
    {"path": "root/subdir1/file1.txt", "size": 1024},
    {"path": "root/subdir1/file2.txt", "size": 2048},
    {"path": "root/subdir2/file3.txt", "size": 4096}
  ]
}
```

(7) 如果需要对大量的小文件进行压缩，而不是单个大文件，哈夫曼编码的效率如何？是否有优化的空间?

对于大量的小文件的压缩，哈夫曼编码的效率可能并不理想，原因主要有以下几点：

1. **频率分布不明显**：哈夫曼编码依赖于字符频率分布来生成最优编码，对于大文件，字符的频率差异较大，可以利用这一点生成较短的编码。但对于大量的小文件，由于文件的规模较小，字符频率往往不那么明显，哈夫曼编码的压缩效果就会下降。
2. **构建编码树的开销**：每次处理一个文件时，都需要重新构建哈夫曼编码树，这对于大量小文件来说会产生较大的开销。每个文件可能都需要一次频率统计和树的构建过程，增加了计算时间和内存开销。
3. **小文件的压缩率较低**：对于小文件，哈夫曼编码可能生成的编码较长，或者压缩效率不高。原因是每个文件中的信息量不足以形成显著的频率差异，这导致压缩后的数据并不会比原文件小很多，甚至可能因为编码表的开销而增加文件大小。

优化空间：

1. **全局频率表**：

- 可以通过对所有小文件的字符频率进行全局统计，构建一个统一的哈夫曼编码表。这样，就不需要为每个文件都单独构建哈夫曼树。对于多个小文件共用同一编码表，可以提高压缩效率并减少计算和存储开销。

2. **块级压缩**：
   - 将多个小文件合并成一个较大的数据块，再进行压缩。这样可以通过合并文件间的频率信息来生成更为紧凑的编码，提升压缩率。
3. **编码表优化**：
   - 可以尝试压缩哈夫曼编码表本身，特别是在大量小文件中，如果编码表过大，会增加压缩开销。对于文件间相似度较高的场景，可以使用更紧凑的编码方式。

(8) 于文档中附上目前代码完成情况的主体部分，并做简略的说明。

文件字符计数器

输入文件路径，返回map，map中记录了每个字符出现的次数。

```
std::map<char, int> Counter::getCountMap(const std::string& filename) {
    char character;
    std::ifstream inputFile(filename, std::ios::in);
    std::map<char, int> countMap;

    while (inputFile.get(character)) {
        countMap[character]++;
    }

    inputFile.close();

    return countMap;
}
```

Huffman树结点结构

Huffman树结点成员变量如下

```
    char data;
    HuffmanTreeNode *leftChild, *rightChild;
    int count;
```

`data`为结点对应的字符，如果结点非叶子结点，则`data`填入`\0`，`count`为该字符在文本中出现的次数。

Huffman树结点序列化算法

每个结点序列化后的格式为：`(字符,次数,左子节点,右子结点)`，如果本结点不是Huffman树的叶子结点，则`字符`处不填任何内容。如果结点的左子结点为空，则`左子节点`处填入`()`，右子结点同理。如，文本`abbbccccc`的序列化哈夫曼树为`(,9,(,4,(a,1,(),()),(b,3,(),())),(c,5,(),()))`。

由于字符'('、')'和"会和格式产生冲突，故需作转义处理，字符为这三个时，转为′'、′'和'\'存储。

代码如下，通过递归构造序列化后的字符串。

```
std::string HuffmanTreeNode::serialize() const {
        std::string data_s = std::string(1, data);
        if (data == '(' || data == ')' || data == '\\')
                data_s = '\\' + data_s;
        return "(" + data_s + "," + std::to_string(count) + "," + serialize(leftChil
d) + "," + serialize(rightChild) + ")";
}

std::string HuffmanTreeNode::serialize(HuffmanTreeNode *node) {
        if (node == nullptr)
                return "()";
        return node->serialize();
}
```

Huffman树结点反序列化算法

根据上面所述格式进行反序列化。关键是合法输入字符串`s`中，`s[0]`必然是`(`，`s[1]`可能是字符，也可能因为该结点非叶子结点而不填入任何东西，导致`s[1]`也可能是`,`，在这里做一个特判。`count`的值必然是`s`中第一个逗号与第二个逗号之间的那个数字。而对于左子树和右子树的构造，则用一个计数器计算括号的层数。左子树是第一个完整括号包围的内容，右子树是第二个完整括号包围的内容，并且第一个括号出现的位置必然在`s[3]`之后。同时，还需对′'、′'和'\'作转义处理。代码如下。

```
HuffmanTreeNode * HuffmanTreeNode::deserialize(const std::string &s) {
        if (s[0] != '(')
                throw std::domain_error("Invalid input when deserializing HuffmanTreeN
ode.");
        if (s[1] == ')')
                return nullptr;

        char data;
        if (s[1] == ',')
                data = '\0';
        else if (s[1] == '\\')
                data = s[2];
        else data = s[1];

        int count = extractNumberBetweenCommas(s);
        std::string left_s = extractIthParenthese(s.substr(4), 1);
        std::string right_s = extractIthParenthese(s.substr(4), 2);
        return new HuffmanTreeNode(data, count, deserialize(left_s), deserialize(righ
t_s));
}

int extractNumberBetweenCommas(const std::string& input) {
    size_t firstComma = input.find(',');
    if (firstComma == std::string::npos) {
        throw std::domain_error("First comma not found.");
    }

    size_t secondComma = input.find(',', firstComma + 1);
    if (secondComma == std::string::npos) {
        throw std::domain_error("Second comma not found.");
    }

    std::string numberStr = input.substr(firstComma + 1, secondComma - firstComma -
1);
    std::istringstream numberStream(numberStr);
    int result;
    if (!(numberStream >> result)) {
        throw std::domain_error("Failed to convert the string between commas to an int
eger.");
```

```
    }

    return result;
}

std::string extractIthParenthese(const std::string& str, int i) {
        std::string result;
        int count = 1;
        int pare_count = 0;
        bool recording = false;
        for (size_t j = 0; j < str.size(); j++) {
                bool flag = j == 0 || str[j - 1] != '\\';
                if (flag) {
                        if (str[j] == '(') {
                                pare_count ++;
                                if (count == i)
                                        recording = true;
                        }
                        if (str[j] == ')') {
                                pare_count --;
                                if (pare_count == 0) {
                                        if (recording) {
                                                result += str[j];
                                                return result;
                                        }
                                        count ++;
                                }
                        }
                }
                if (recording)
                        result += str[j];
        }
        throw std::domain_error("Cannot find " + std::to_string(i) + "th parenthes
e.");
}
```

Huffman树的序列化与反序列化

Huffman树的序列化与反序列化比较简单，就是对根结点的序列化与反序列化。

Huffman树的构造

输入一个`map<char, int>`，根据它构造Huffman树，利用STL标准库的优先队列定义一个最小堆编写Huffman构造算法。

此种方法需要定义Huffman结点之间的大小，因此重载运算符`>`、`>=`、`==`、`<`、`<=`。

```
bool HuffmanTreeNode::operator<(const HuffmanTreeNode &other) const {
        return count < other.count;
}

bool HuffmanTreeNode::operator>(const HuffmanTreeNode &other) const {
        return count > other.count;
}

bool HuffmanTreeNode::operator==(const HuffmanTreeNode &other) const {
        return count == other.count;
}

bool HuffmanTreeNode::operator<=(const HuffmanTreeNode &other) const {
        return count <= other.count;
}

bool HuffmanTreeNode::operator>=(const HuffmanTreeNode &other) const {
        return count >= other.count;
}

HuffmanTree::HuffmanTree(const std::map<char, int> &m) {
    struct compare {
        bool operator()(HuffmanTreeNode *a, HuffmanTreeNode *b) {
            return *a > *b;
        }
    };
    std::priority_queue<HuffmanTreeNode *, std::vector<HuffmanTreeNode *>, compare> pr
i_queue;
    for (auto iter = m.begin(); iter != m.end(); iter ++) {
        HuffmanTreeNode *node = new HuffmanTreeNode(iter->first, iter->second);
        pri_queue.push(node);
    }

    while (pri_queue.size() > 1) {
        HuffmanTreeNode *node1 = pri_queue.top();
        pri_queue.pop();
        HuffmanTreeNode *node2 = pri_queue.top();
        pri_queue.pop();
        HuffmanTreeNode *parent = new HuffmanTreeNode('\0', node1->count + node2->coun
```

```
t);
        parent->leftChild = node1;
        parent->rightChild = node2;
        pri_queue.push(parent);
    }

    root = pri_queue.top();
}
```

Huffman树中字符对应Huffman编码的提取

使用遍历递归的方式计算Huffman编码。在递归过程中传递的`std::map<std::string, char>`是引用类型，保证了对同一个`map`进行操作。

```
std::map<std::string, char> & HuffmanTreeNode::writeCharMap(const HuffmanTreeNode *nod
e, std::string s, std::map<std::string, char> &m) {
        if (node == nullptr)
                return m;

        if (node->data != '\0')
                m[s] = node->data;

        else {
                writeCharMap(node->leftChild, s + "0", m);
                writeCharMap(node->rightChild, s + "1", m);
        }

        return m;
}

std::map<std::string, char> HuffmanTree::getCharMap() const {
        std::map<std::string, char> m;
        HuffmanTreeNode::writeCharMap(root, "", m);
        return m;
}
```

压缩单个文件

压缩步骤如下：

1. 统计字符频率并构建哈夫曼树
    - 首先，遍历源文件中的每一个字符，统计每个字符出现的频率。基于这些频率信息，构建哈夫曼树。哈夫曼树是一种用于数据压缩的二叉树，它能为每个字符分配一个唯一的二进制编码，频率高的字符使用较短的编码，频率低的字符使用较长的编码，从而达到压缩数据的目的。
2. 序列化哈夫曼树并存储
    - 构建完成的哈夫曼树需要被序列化，以便在压缩文件中保存树的结构信息。序列化后的哈夫曼树被写入到压缩文件的开头部分，这样在解压时可以还原出原始的哈夫曼树结构。
3. 转换源文件为哈夫曼编码
    - 使用生成的哈夫曼编码表，将源文件中的每个字符替换为对应的哈夫曼编码位串。将这些位串按顺序拼接起来，形成一个连续的比特流，并将其写入到压缩文件中。
4. 字节对齐处理
    - 为了确保压缩文件的比特流能够被正确解析，需要将比特流的长度调整为8的倍数。如果比特流的长度不是8的倍数，则在末尾添加适量的填充零位，使其达到8位对齐。填充的位数会被记录下来，以便在解压时正确去除这些多余的填充位。

```
void HuffmanZip::compress(const std::string& inputPath, const std::string& outputPat
h) {
    std::map<char, int> countMap = Counter::getCountMap(inputPath);
    HuffmanTree huffmanTree(countMap);
    std::map<char, std::string> codeMap = huffmanTree.getCodeMap();

    std::ifstream inFile(inputPath, std::ios::binary);
    std::ofstream outFile(outputPath, std::ios::binary);

    std::string serializedTree = huffmanTree.serialize();
    uint32_t treeSize = serializedTree.size();

    treeSize = htonl(treeSize);
    outFile.write(reinterpret_cast<char*>(&treeSize), sizeof(treeSize));

    outFile.write(serializedTree.c_str(), serializedTree.size());

    std::string bitStream;
    char ch;
    while (inFile.get(ch)) {
        bitStream += codeMap[ch];
    }

    size_t padding = 8 - (bitStream.size() % 8);
    if (padding != 8) {
        bitStream.append(padding, '0');
    } else {
        padding = 0;
    }

    outFile.put(static_cast<char>(padding));

    for (size_t i = 0; i < bitStream.size(); i += 8) {
        std::bitset<8> bits(bitStream.substr(i, 8));
        char byte = static_cast<char>(bits.to_ulong());
        outFile.put(byte);
    }

    inFile.close();
```

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

解压单个文件

解压步骤如下：

1. 读取并反序列化哈夫曼树
    - 从压缩文件中读取序列化的哈夫曼树数据，并将其反序列化还原为原始的哈夫曼树结构。这一步骤确保了解压过程中可以正确地将比特流转换回原始字符。
2. 读取填充信息并提取比特流
    - 接着，读取压缩文件中存储的填充位数信息。这些填充位是为了实现字节对齐而添加的多余零位。在去除这些填充位之后，剩余的比特流即为实际的哈夫曼编码位串。
3. 使用哈夫曼树解码比特流
    - 利用还原后的哈夫曼树，从比特流中逐位遍历并进行解码。每当遍历到一个叶子节点时，意味着找到了一个完整的原始字符，将其写入到输出文件中。继续从树的根节点开始下一次解码，直到整个比特流被完全解析为原始数据。

```
void HuffmanZip::decompress(const std::string& inputPath, const std::string& outputPat
h) {
    std::ifstream inFile(inputPath, std::ios::binary);
    if (!inFile)
        throw std::domain_error("Unable to open input file.");

    uint32_t treeSize;
    inFile.read(reinterpret_cast<char*>(&treeSize), sizeof(treeSize));
    treeSize = ntohl(treeSize);

    std::string serializedTree(treeSize, '\0');
    inFile.read(&serializedTree[0], treeSize);
    HuffmanTree tree = HuffmanTree(serializedTree);
    HuffmanTreeNode* root = tree.root;

    char paddingChar;
    inFile.get(paddingChar);
    size_t padding = static_cast<unsigned char>(paddingChar);

    std::string bitStream;
    char ch;
    while (inFile.get(ch)) {
        std::bitset<8> bits(static_cast<unsigned long long>(static_cast<unsigned cha
r>(ch)));
        bitStream += bits.to_string();
    }

    if (padding > 0 && padding < 8) {
        bitStream = bitStream.substr(0, bitStream.size() - padding);
    }

    std::ofstream outFile(outputPath, std::ios::out | std::ios::binary);
    if (!outFile)
        throw std::domain_error("Unable to open output file.");

    HuffmanTreeNode* currentNode = root;
    for (char bit : bitStream) {
        currentNode = (bit == '0') ? currentNode->leftChild : currentNode->rightChild;
```

```
        if (!currentNode->leftChild && !currentNode->rightChild) {
            outFile.put(currentNode->data);
            currentNode = root;
        }
    }

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