huffman编码——实验报告

- **班级**：通信2301
- **学号**：U202342641
- **姓名**：陶宇轩

一、编程实验名称与内容概述

- **实验名称**：Huffman编码
- **内容概述**：
  唯一霍夫曼树的构造与验证系统
  - **输入**
    - 从键盘输入一个字符串（长度≤1000），包含大小写字母、数字、标点符号（如!@#$%^&()）及空格。
      注意：字符区分大小写（A与a视为不同字符）。
  - **输出**
    - **字符频率统计表**：按ASCII码升序输出（如！的ASCII为33，排在 0（ASCII 48）之前）。
    - **霍夫曼编码表**：按ASCII码升序输出字符及其编码。
    - **编码结果**：将输入字符串转为二进制编码流（连续输出，无分隔符）。
    - **译码验证**：将编码结果还原为原始字符串，验证一致性。

二、程序设计思路

数据结构

1. **HuffmanNode结构体**
   - `character`: 存储字符
   - `frequency`: 字符出现频率
   - `min_ascii`: 当前子树中最小的ASCII值（用于频率相同时的节点排序）
   - `left`/`right`: 左右子节点指针
2. **优先队列（最小堆）**
   - 使用自定义比较规则：
     - 首先比较频率（小者优先）
     - 频率相同时比较`min_ascii`（ASCII值小者优先）

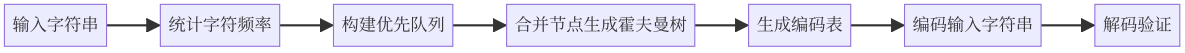
算法步骤

1. **频率统计**
   - 遍历输入字符串，统计每个字符的出现次数，存入`map<char, int>`
2. **构建霍夫曼树**
   - 将所有字符作为叶子节点加入优先队列
   - 循环合并队列中优先级最低的两个节点
     - 取出频率最小的两个节点，按`min_ascii`调整顺序（左子节点ASCII更小）。

        - 创建新节点，频率为两者之和，`min_ascii`取较小值。
    - 直到队列中仅剩一个根节点。
3. **生成编码表**
    - 递归遍历霍夫曼树：
        - 左分支标记为`0`，右分支标记为`1`
        - 叶子节点记录路径编码（单节点树特殊处理为`"0"`）
4. **解码验证**
    - 根据编码字符串遍历霍夫曼树，到达叶子节点时记录字符并重置起点

三、代码说明

流程图



核心函数

1. **`HuffmanTree`构造函数**
    - 初始化优先队列，合并节点时保证左子节点的`min_ascii`更小
    - 处理单节点树（输入字符串全为同一字符）

```
    /**
     * 构造函数：根据频率映射构建霍夫曼树
     * @param freqMap 字符频率映射表
     */
    HuffmanTree(map<char, int>& freqMap) {
        // 使用优先队列（最小堆）存储节点
        priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;

        // 将所有字符作为叶子节点加入优先队列
        for (auto& pair : freqMap) {
            pq.push(new HuffmanNode(pair.first, pair.second,
pair.first));
        }

        // 合并节点直到只剩一个根节点
        while (pq.size() > 1) {
            HuffmanNode* left = pq.top(); pq.pop();
            HuffmanNode* right = pq.top(); pq.pop();

            // 确保左子节点的min_ascii较小（频率相同情况下保持顺序）
            if (left->min_ascii > right->min_ascii)
                swap(left, right);

            // 创建合并后的新节点，频率为子节点之和，min_ascii取较小值
            HuffmanNode* merged = new HuffmanNode('\0',
                                  left->frequency + right->frequency,
                                  min(left->min_ascii,right->min_ascii));
            merged->left = left;
            merged->right = right;
```

```
            pq.push(merged);
        }

        root = pq.top();  // 设置根节点
    }
```

2. `buildCodeTable`递归函数
   - 通过深度优先遍历生成编码，左分支加`0`，右分支加`1`
   - 特殊处理单节点树，直接编码为`"0"`

```
    /**
     * 递归生成霍夫曼编码表
     * @param node 当前遍历的节点
     * @param code 当前路径的二进制编码
     * @param codeTable 存储字符与编码的映射表
     */
    void buildCodeTable(HuffmanNode* node, string code, map<char, string>& codeTable) {
        // 到达叶子节点时记录编码（单节点树需特殊处理）
        if (node->left == nullptr && node->right == nullptr) {
            codeTable[node->character] = code.empty() ? "0" : code;
            return;
        }
        // 左子树路径添加'0'，右子树路径添加'1'
        buildCodeTable(node->left, code + "0", codeTable);
        buildCodeTable(node->right, code + "1", codeTable);
    }
```

3. `decode`解码函数
   - 根据编码逐位遍历霍夫曼树，到达叶子节点后记录字符并重置起点

```
    /**
     * 解码二进制字符串
     * @param encoded 编码后的二进制字符串
     * @return 解码后的原始字符串
     */
    string decode(const string& encoded) {
        string result;
        HuffmanNode* current = root;

        // 处理单节点树的特殊情况（所有字符相同）
        if (root->left == nullptr && root->right == nullptr) {
            return string(encoded.size(), root->character);
        }

        // 遍历编码字符串，根据0/1移动节点
        for (char bit : encoded) {
            current = (bit == '0') ? current->left : current->right;

```

```
            // 到达叶子节点时记录字符并重置遍历起点
            if (current->left == nullptr && current->right == nullptr) {
                result += current->character;
                current = root;
            }
        }
        return result;
    }
};
```

四、运行结果与复杂度分析

运行结果

输入：wfbabfueibvvicae
输出：

```
字符频率统计表:
a 2
b 3
c 1
e 2
f 2
i 2
u 1
v 2
w 1

霍夫曼编码表:
a 0000
b 10
c 1100
e 111
f 010
i 011
u 1101
v 001
w 0001

编码结果:
00010101000001001011011110111000100101111000000111

译码验证结果：一致
```

复杂度分析

- **时间复杂度**：
  - 构建霍夫曼树：`O(n * log n)`（`n`为字符种类数，每次堆操作为`O(log n)`）
  - 生成编码表：`O(n)`（遍历所有节点）
  - 解码：`O(m)`（`m`为编码长度）

- **空间复杂度**：
  - 优先队列：`O(n)`
  - 递归栈深度：`O(log n)`（树的高度）

五、改进方向与心得体会

改进方向

1. **内存管理**：添加析构函数释放动态分配的树节点
2. **优化排序**：探索更高效的节点合并策略

心得体会

1. 通过实现霍夫曼树，深入理解了贪心算法与最优二叉树的构造原理
2. 学习到了优先队列和自定义排序规则的使用