

《数据结构与算法》课程设计

指导教师：胡洁、杨东鹤

2023 年 12 月

班级：计算机科学与技术 21（4）班

学号：2021329600006

姓名：陈昊天

目录

| | |
|--------------------------|---|
| 一、需求分析 | 1 |
| 1.1 任务描述 | 1 |
| 1.2 数据处理流程 | 1 |
| 1.3 模块划分 | 1 |
| 1.4 用户交互和操作流程 | 2 |
| 1.5 高级特性 | 2 |
| 二、概要设计 | 2 |
| 2.1 抽象数据类型定义 | 2 |
| 2.2 主程序流程 | 3 |
| 2.3 模块设计与层次调用关系 | 3 |
| 三、详细设计 | 4 |
| 3.1 数据结构设计 | 4 |
| 3.1.1 Node 类 | 4 |
| 3.1.2 Compare 结构 | 4 |
| 3.1.3 minHeap 类型定义 | 5 |
| 3.2 主要模块设计 | 5 |
| 3.2.1 数据读取与频率统计模块 | 5 |
| 3.2.2 哈夫曼树构建模块 | 5 |
| 3.2.3 编码模块 | 6 |
| 3.2.4 译码模块 | 6 |
| 3.2.5 哈夫曼树序列化与反序列化 | 7 |
| 3.3 主函数设计 | 7 |
| 四、调试分析 | 8 |
| 4.1 问题诊断与解决 | 8 |
| 4.2 时空复杂度分析 | 9 |
| 4.3 经验与体会 | 9 |
| 五、用户使用说明 | 9 |

| | |
|-------------------|----|
| 5.1 程序功能 | 9 |
| 5.2 编译和运行 | 10 |
| 5.3 使用指南 | 10 |
| 5.4 注意事项 | 11 |
| 六、测试结果分析与讨论 | 11 |
| 6.1 功能测试用例 | 11 |
| 6.2 边界条件测试 | 13 |
| 6.3 异常情况处理 | 14 |
| 6.4 性能测试 | 15 |
| 七、附录 | 16 |
| 7.1 程序源代码 | 16 |
| 7.2 参考文献 | 23 |

一、需求分析

1.1 任务描述

本项目旨在设计和实现一个基于哈夫曼编码的编/译码系统，用于优化数据传输效率。该系统应能处理文本数据和二进制数据（如图像文件），并具备以下功能：

初始化：设置起始配置和必要参数。

编码：将输入的数据（文本或二进制文件）转换成哈夫曼编码。

译码：将哈夫曼编码转换回原始数据格式。

打印代码文件：输出编码和译码的结果。

1.2 数据处理流程

输入数据：从 `data.txt` 或任意二进制文件读取数据。

编码过程：构建哈夫曼树，将输入数据转换为哈夫曼编码，存储编码结果至 `code.txt`。

保存哈夫曼树：将构建的哈夫曼树存储，以便于译码时使用。

译码过程：读取 `code.txt` 和哈夫曼树，将编码数据还原为原始格式，并在屏幕上显示。

1.3 模块划分

数据读取模块：从文件中读取文本或二进制数据。

频率统计模块：统计数据中各字符/字节的出现频率。

哈夫曼树构建模块：根据频率统计结果构建哈夫曼树。

编码模块：使用哈夫曼树对数据进行编码。

哈夫曼树存储模块：保存哈夫曼树结构，用于译码。

译码模块：读取编码数据和哈夫曼树，进行译码。

输出模块：将译码结果输出到屏幕或文件。

1.4 用户交互和操作流程

用户输入数据文件路径。

系统自动执行编码和译码流程。

用户接收译码后的输出结果。

1.5 高级特性

二进制文件处理：对非文本文件（如图像文件）进行编码和译码。

自定义编码：根据数据类型或用户指定的规则进行优化编码。

二、概要设计

2.1 抽象数据类型定义

Node 类：

属性：

unsigned char data: 字符数据。

unsigned freq: 字符频率。

bool isLeaf: 是叶子节点

HuffmanNode *left, *right: 指向左右子节点的指针。

构造函数：接收字符数据和频率，初始化左右子节点为 nullptr。

compare 结构：

操作符重载：实现优先级队列的比较机制，以频率作为比较依据。

HuffmanPriorityQueue 类型定义：

优先级队列，用于构建哈夫曼树。

2.2 主程序流程

主函数 (main):

读取原始文件，构造哈夫曼树

调用 `encodeHuffman` 函数进行文件编码。

保存编码，序列化哈夫曼树。

读取编码，反序列化哈夫曼树。

调用 `decodeFile` 函数进行文件解码。

比较原始文件和解码后的文件。

2.3 模块设计与层次调用关系

数据读取与频率统计:

函数: `encodeHuffman`

流程: 读取输入文件吗, 统计字符频率, 构建哈夫曼树 (调用 `buildHuffmanTree`)。

哈夫曼树构建:

函数: `buildHuffmanTree`

流程: 以频率为基础构建优先级队列, 合并节点直到只剩下一个节点 (树的根节点) 。

编码过程:

函数: `printCodes`

流程: 遍历哈夫曼树, 生成编码表。

编码数据写入:

函数: `encodeHuffman`

流程: 将编码后的数据写入输出文件。

序列化哈夫曼树:

函数: `serializeHuffmanTree`

流程: 将哈夫曼树结构写入文件。

译码过程:

函数: `decodeHuffman`

流程: 读取编码数据和哈夫曼树文件, 使用哈夫曼树进行译码, 将译码结果写入输出文件。

反序列化哈夫曼树:

函数: `deserializeHuffmanTree`

流程: 从文件中读取并重建哈夫曼树结构。

三、详细设计

3.1 数据结构设计

3.1.1 Node 类

```
struct Node {
    Byte data;
    unsigned frequency;
    bool isLeaf;
    Node *left, *right;
    Node(Byte data, unsigned frequency, bool isLeaf) {
        left = right = nullptr;
        this->data = data;
        this->frequency = frequency;
        this->isLeaf = isLeaf;
    }
};
```

3.1.2 Compare 结构

```
struct Compare {
```

```

    bool operator()(Node* l, Node* r) { return l->frequency >
r->frequency; }
};

```

3.1.3 minHeap 类型定义

```
std::priority_queue<Node*, std::vector<Node*>, Compare> minHeap;
```

3.2 主要模块设计

3.2.1 数据读取与频率统计模块

```

std::vector<Byte> readFile(const std::string& filename) {
    std::ifstream file(filename, std::ios::binary);
    std::vector<Byte>
    bytes((std::istreambuf_iterator<char>(file)),
          std::istreambuf_iterator<char>());
    return bytes;
}
std::unordered_map<Byte, int> freqMap;
for (auto byte : originalData) {
    freqMap[byte]++;
}

```

3.2.2 哈夫曼树构建模块

```

Node* buildHuffmanTree(Byte data[], int freq[], int size) {
    struct Node *left, *right, *top;
    std::priority_queue<Node*, std::vector<Node*>, Compare> minHeap;
    for (int i = 0; i < size; ++i)
        minHeap.push(new Node(data[i], freq[i], true));
    while (minHeap.size() != 1) {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();
        top = new Node('\0', left->frequency + right->frequency,
false);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }
}

```



```

    }
    return minHeap.top();
}

```

3.2.3 编码模块

```

void printCodes(Node* root, std::string str) {
    if (!root) return;
    if (root->isLeaf) codes[root->data] = str;
    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

std::vector<Byte> encodeHuffman(const std::vector<Byte>& bytes) {
    std::string encodedString;
    for (auto byte : bytes) {
        encodedString += codes[byte];
    }
    int lastByteLength = encodedString.size() % 8;
    while (encodedString.size() % 8 != 0) {
        encodedString += "0";
    }
    std::vector<Byte> encodedData;
    for (size_t i = 0; i < encodedString.size(); i += 8) {
        std::string byteString = encodedString.substr(i, 8);
        std::bitset<8> bitset(byteString);
        encodedData.push_back(static_cast<Byte>(bitset.to_ulong()));
    }
    if (lastByteLength == 0) lastByteLength = 8;
    encodedData.push_back(lastByteLength);
    return encodedData;
}

```

3.2.4 译码模块

```

std::vector<Byte> decodeHuffman(const std::vector<Byte>&
    encodedData, Node* root) {
    std::vector<Byte> decodedData;
    Node* current = root;
    int lastByteLength = encodedData.back();
    for (size_t i = 0; i < encodedData.size() - 1; ++i) {
        std::bitset<8> bits(encodedData[i]);
        int limit = (i == encodedData.size() - 2) ? lastByteLength :

```

```

    8;
    for (int j = 7; j >= 8 - limit; --j) {
        current = bits[j] ? current->right : current->left;
        if (current->left == nullptr && current->right == nullptr)
        {
            decodedData.push_back(current->data);
            current = root;
        }
    }
}
return decodedData;
}

```

3.2.5 哈夫曼树序列化与反序列化

```

void serializeHuffmanTree(Node* root, std::ostream& out) {
    if (!root) {
        out.put(0);
        return;
    }
    out.put(1);
    out.put(root->data);
    serializeHuffmanTree(root->left, out);
    serializeHuffmanTree(root->right, out);
}

Node* deserializeHuffmanTree(std::istream& in) {
    if (!in.good()) return nullptr;
    char flag;
    in.get(flag);
    if (flag == 0) return nullptr;
    char data;
    in.get(data);
    Node* node = new Node(data, 0, false);
    node->left = deserializeHuffmanTree(in);
    node->right = deserializeHuffmanTree(in);
    if (!node->left && !node->right) node->isLeaf = true;
    return node;
}

```

3.3 主函数设计

```

int main() {

```

```

std::string filePath;
std::cout << "请输入要编码的文件的相对路径: ";
std::cin >> filePath;
// ...读取原始文件...
// ...统计频率...
Node* root = buildHuffmanTree(data, freq, size);
printCodes(root, "");
auto encodedData = encodeHuffman(originalData);
writeFile("encoded.bin", encodedData);
std::ofstream treeOutFile("tree.txt");
printHuffmanTree(root, treeOutFile, "");
treeOutFile.close();
std::ofstream treeFile("tree.bin", std::ios::binary);
serializeHuffmanTree(root, treeFile);
treeFile.close();
auto encodedFileData = readFile("encoded.bin");
std::ifstream treeFileIn("tree.bin", std::ios::binary);
Node* deserializedRoot = deserializeHuffmanTree(treeFileIn);
treeFileIn.close();
auto decodedData = decodeHuffman(encodedFileData,
deserializedRoot);
writeFile(("decoded" + extension).c_str(), decodedData);
// ...程序输出和内存清理...
return 0;
}

```

四、调试分析

4.1 问题诊断与解决

(1) 内存泄漏问题

在构建哈夫曼树时，使用了 `new` 关键字动态分配内存。初步调试时，注意到未适当释放内存，导致内存泄漏。通过在程序结束前加入递归删除树节点的函数，解决了这个问题。

(2) 文件读写错误

在处理二进制文件时，遇到了文件读写错误。问题在于文件以文本模式打开，而非二进制模式。修改文件流的打开方式为二进制模式(`std::ios::binary`)后解决。

(3) 编码不一致性

在某些情况下，编码后的数据与原始数据不一致。原因是在编码过程中，对最后一个字节的处理不正确。添加了特殊处理以记录最后一个字节的实际长度，解决了这个问题。

4.2 时空复杂度分析

(1) 时间复杂度

构建哈夫曼树的时间复杂度为 $O(n \log n)$ ，其中 n 是不同字符的数量。编码和解码过程的时间复杂度均为 $O(m)$ ，其中 m 是原始数据的长度。

(2) 空间复杂度

哈夫曼树的空间复杂度为 $O(n)$ 。编码和解码过程中需要额外的空间来存储编码字符串和解码结果，取决于原始数据的大小。

4.3 经验与体会

在实施哈夫曼编码系统的过程中，我深入了解了哈夫曼编码的理论和实践应用，特别是利用优先队列来构建哈夫曼树的技术。我发现将理论知识应用于实际问题解决中是一个挑战，同时也是一个极好的学习机会。通过这个项目，我学会了分步测试每个模块的方法，这种方法在调试过程中证明非常有效。每当一个模块完成时，我会对其进行单独测试，确保其功能正常，再将其整合到主程序中。这种逐步集成的方法减少了寻找和修复错误的时间，提高了整体开发效率。

五、用户使用说明

5.1 程序功能

1. 对文件进行哈夫曼编码，实现数据压缩。

2. 将压缩后的数据保存到文件中。
3. 将哈夫曼树结构保存到文件中以供解压缩时使用。
4. 对压缩后的文件进行解压缩，恢复到原始状态。

5.2 编译和运行

由于程序涉及到 macOS 系统调用命令 `open`，必须使用 macOS 方能运行，Windows 与 Linux 需修改源码。确保系统中安装了 C++ 编译器，如 `g++` 或 `clang`，并且可以使用 C++11 或更高版本的标准。

在终端或命令提示符中，导航到程序文件的目录，并执行以下命令来编译程序：

```
g++ -std=c++11 -o huffman_coding [程序文件名].cpp
```

编译成功后，通过以下命令运行程序：

```
./huffman_coding
```

5.3 使用指南

输入文件路径：程序会提示输入要进行编码的文件的相对路径。确保文件路径正确无误。

文件压缩：程序将读取指定的文件，统计字符频率，构建哈夫曼树，并进行哈夫曼编码。编码后的数据会保存到名为 `encoded.bin` 的文件中。

哈夫曼树保存：哈夫曼树的结构会被保存在两个文件中。一份是可读的文本形式（`tree.txt`），另一份是序列化后的二进制形式（`tree.bin`），用于解码。

文件解压缩：程序接着读取 `encoded.bin` 和 `tree.bin` 文件，利用反序列化的哈夫曼树对数据进行解码，恢复原始文件。解码后的文件将保存为 `decoded[原始文件扩展名]`。

查看结果：程序将尝试自动打开原始文件、哈夫曼树可读形式文件和解码后的文件。可以手动查看这些文件以验证编解码过程。

5.4 注意事项

1. 确保输入文件的路径是正确的，并且文件存在于系统中。
2. 由于哈夫曼编码是无损压缩，解码后的文件应当与原始文件完全相同。
3. 运行程序时，请确保有足够的权限访问和修改指定的文件和目录。

六、测试结果分析与讨论

6.1 功能测试用例

测试用例 1：小型文件测试

描述：使用一个小的文本文件进行测试，检查编码和解码是否正确执行。

预期结果：解码后的文件与原始文件内容完全一致。

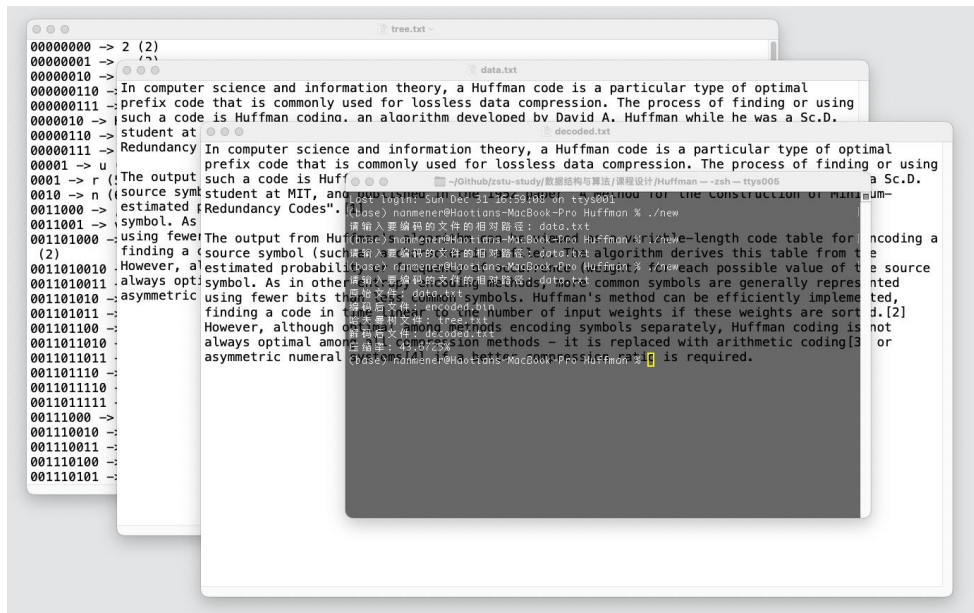


图 6-1 小型文件测试

测试用例 2：大型文件测试

描述：使用一个较大的文件（如图片或视频文件）进行测试，以检查程序在处理大型文件时的性能和准确性。

预期结果：解码后的文件应与原始文件完全相同，没有数据丢失或损坏。

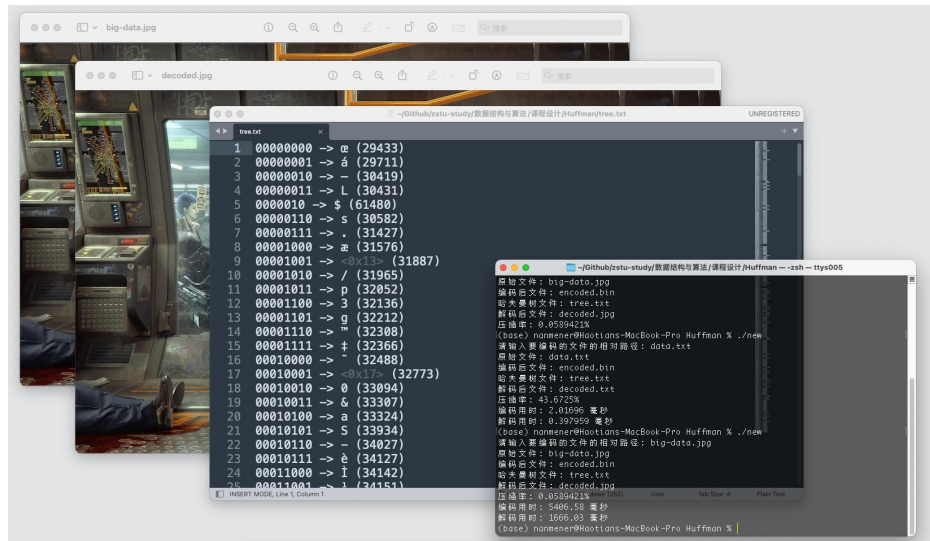


图 6-2 大型文件测试

测试用例 3：不同文件格式测试

描述：测试不同类型的文件（如.txt、.jpg、.mov 等），以确保程序对不同文件格式的兼容性。

预期结果：所有类型的文件都能被正确编码和解码。

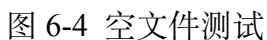


图 6-3 MOV 文件测试

测试用例 4: 空文件测试

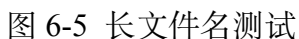
描述：输入一个空文件。

预期结果: 程序应该能够处理空文件, 不产生错误。



描述: 测试具有非常长文件名的文件。

预期结果: 文件名的长度不应影响编解码过程。



6.3 异常情况处理

测试用例 6: 不存在的文件路径

描述: 输入一个不存在的文件路径。

预期结果: 程序应该能够优雅地处理文件不存在的情况, 给出明确的错误信息。

A terminal window titled '~/Github/zstu-study/数据结构与算法/课程设计/Huffman -- zsh -- ttys005'. The user runs a series of commands: 'rm blank.txt', 'touch blan.txt' (note the typo), './new', and then enters 'blank.txt' when prompted. The program outputs an error: 'ERROR: 无法打开文件 'blank.txt' 以进行读取。可能是文件不存在或权限不足。'. The user then runs 'touch blank.txt', './new', and enters 'blank.txt'. The program outputs another error: 'ERROR: 文件 'blank.txt' 不存在或为空。'. Finally, the user runs './new' and enters '123'. The program outputs: 'ERROR: 无法打开文件 '123' 以进行读取。可能是文件不存在或权限不足。'.

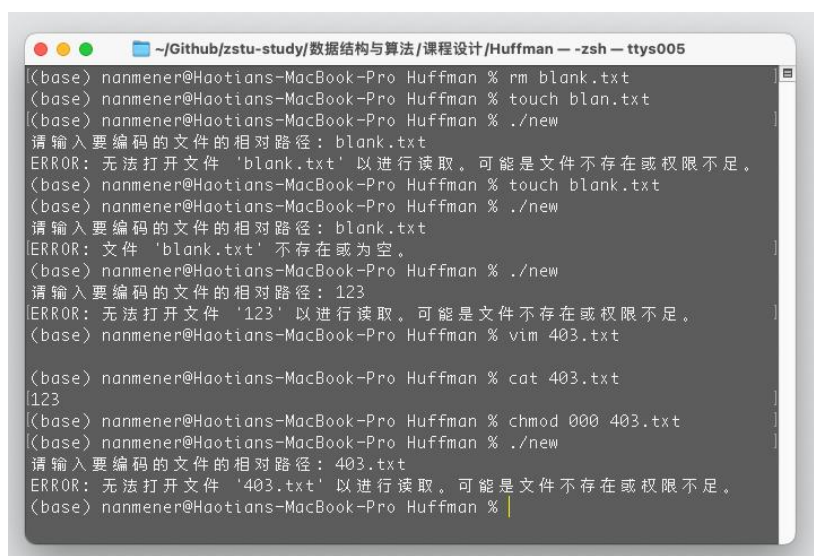
```
(base) nanmener@Haotians-MacBook-Pro Huffman % rm blank.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % touch blan.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: blank.txt
ERROR: 无法打开文件 'blank.txt' 以进行读取。可能是文件不存在或权限不足。
(base) nanmener@Haotians-MacBook-Pro Huffman % touch blank.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: blank.txt
ERROR: 文件 'blank.txt' 不存在或为空。
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: 123
ERROR: 无法打开文件 '123' 以进行读取。可能是文件不存在或权限不足。
(base) nanmener@Haotians-MacBook-Pro Huffman %
```

图 6-6 不存在文件测试

测试用例 7: 文件读写权限测试

描述: 使用一个程序没有读写权限的文件进行测试。

预期结果: 程序应该能够处理权限问题, 并给出适当的错误信息。

A terminal window titled '~/Github/zstu-study/数据结构与算法/课程设计/Huffman -- zsh -- ttys005'. The user runs: 'rm blank.txt', 'touch blan.txt' (typo), './new', and enters 'blank.txt'. The program outputs an error: 'ERROR: 无法打开文件 'blank.txt' 以进行读取。可能是文件不存在或权限不足。'. The user runs 'touch blank.txt', './new', and enters 'blank.txt'. The program outputs: 'ERROR: 文件 'blank.txt' 不存在或为空。'. The user runs './new' and enters '123'. The program outputs: 'ERROR: 无法打开文件 '123' 以进行读取。可能是文件不存在或权限不足。'. The user then runs 'vim 403.txt', 'cat 403.txt' (which outputs '123'), 'chmod 000 403.txt', and './new'. When prompted with '403.txt', the program outputs: 'ERROR: 无法打开文件 '403.txt' 以进行读取。可能是文件不存在或权限不足。'.

```
(base) nanmener@Haotians-MacBook-Pro Huffman % rm blank.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % touch blan.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: blank.txt
ERROR: 无法打开文件 'blank.txt' 以进行读取。可能是文件不存在或权限不足。
(base) nanmener@Haotians-MacBook-Pro Huffman % touch blank.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: blank.txt
ERROR: 文件 'blank.txt' 不存在或为空。
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: 123
ERROR: 无法打开文件 '123' 以进行读取。可能是文件不存在或权限不足。
(base) nanmener@Haotians-MacBook-Pro Huffman % vim 403.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % cat 403.txt
123
(base) nanmener@Haotians-MacBook-Pro Huffman % chmod 000 403.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: 403.txt
ERROR: 无法打开文件 '403.txt' 以进行读取。可能是文件不存在或权限不足。
(base) nanmener@Haotians-MacBook-Pro Huffman %
```

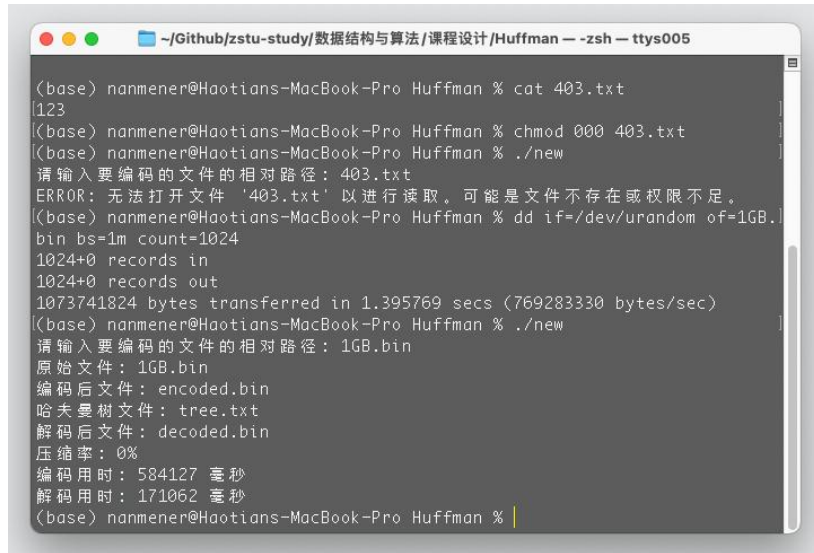
图 6-7 文件读写权限测试

6.4 性能测试

测试用例 8：大文件的处理速度

描述：测试程序处理大文件（如数 GB 大小的文件）的时间。

预期结果：程序应能在合理的时间内完成大文件的编码和解码过程。



```
(base) nanmener@Haotians-MacBook-Pro Huffman % cat 403.txt
123
(base) nanmener@Haotians-MacBook-Pro Huffman % chmod 000 403.txt
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: 403.txt
ERROR: 无法打开文件 '403.txt' 以进行读取, 可能是文件不存在或权限不足。
(base) nanmener@Haotians-MacBook-Pro Huffman % dd if=/dev/urandom of=1GB.
bin bs=1m count=1024
1024+0 records in
1024+0 records out
1073741824 bytes transferred in 1.395769 secs (769283330 bytes/sec)
(base) nanmener@Haotians-MacBook-Pro Huffman % ./new
请输入要编码的文件的相对路径: 1GB.bin
原始文件: 1GB.bin
编码后文件: encoded.bin
哈夫曼树文件: tree.txt
解码后文件: decoded.bin
压缩率: 0%
编码用时: 584127 毫秒
解码用时: 171062 毫秒
(base) nanmener@Haotians-MacBook-Pro Huffman %
```

图 6-8 大文件处理测试

测试用例 9：内存使用情况

描述：监视编码和解码大文件时的内存使用情况。

预期结果：程序应有效地管理内存，防止过度消耗或泄漏。

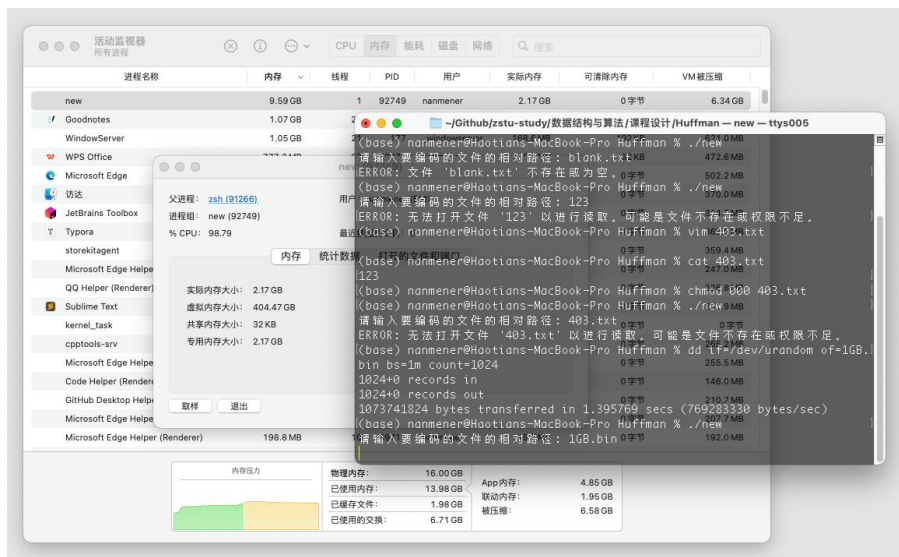


图 6-9 内存使用情况

七、附录

7.1 程序源代码

```
#include <bitset>
#include <chrono>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <iterator>
#include <queue>
#include <string>
#include <unordered_map>
#include <vector>

typedef unsigned char Byte;

// 存储哈夫曼编码的映射
std::unordered_map<Byte, std::string> codes; // codes[字节]=哈夫曼
编码

// 哈夫曼树 节点
struct Node {
    Byte data;           // 存储 1 字节
    unsigned frequency; // 频率
    bool isLeaf;         // 是叶子节点
    Node *left, *right;  // 左右节点

    Node(Byte data, unsigned frequency, bool isLeaf) {
        left = right = nullptr;
        this->data = data;
        this->frequency = frequency;
        this->isLeaf = isLeaf;
    }
};

// 小根堆 频率升序排序
struct Compare {
    bool operator()(Node* l, Node* r) { return l->frequency >
r->frequency; }
};
```

```

// 生成哈夫曼编码
void printCodes(Node* root, std::string str) {
    if (!root) return;

    if (root->isLeaf) codes[root->data] = str;

    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

// 构建哈夫曼树
Node* buildHuffmanTree(Byte data[], int freq[], int size) {
    struct Node *left, *right, *top;
    std::priority_queue<Node*, std::vector<Node*>, Compare> minHeap;
    for (int i = 0; i < size; ++i) // 创建节点 加入堆
        minHeap.push(new Node(data[i], freq[i], true));

    while (minHeap.size() != 1) {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();

        // 频率相加 data=0
        top = new Node('\0', left->frequency + right->frequency,
false);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }
    return minHeap.top();
}

// 读取文件到字节数组
std::vector<Byte> readFile(const std::string& filename) {
    std::ifstream file(filename, std::ios::binary);
    // 检查文件是否成功打开
    if (!file) {
        std::cerr << "ERROR: 无法打开文件 '" << filename
        << "' 以进行读取。可能是文件不存在或权限不足." << std::endl;
        return std::vector<Byte>(); // 返回空数组
    }
    std::vector<Byte>

```

```

bytes((std::istreambuf_iterator<char>(file)),
      std::istreambuf_iterator<char>());
    if (bytes.empty()) {
        std::cerr << "ERROR: 文件 '" << filename << "' 为空。" <<
std::endl;
        return std::vector<Byte>(); // 返回空数组
    }
    return bytes;
}

// 将字节数组写入文件
void writeFile(const std::string& filename, const std::vector<Byte>&
bytes) {
    std::ofstream file(filename, std::ios::binary);
    // unsigned char* 强制转换为 const char*
    file.write(reinterpret_cast<const          char*>(&bytes[0]),
bytes.size());
}

// 累加编码数据 连接哈夫曼编码
std::vector<Byte> encodeHuffman(const std::vector<Byte>& bytes) {
    std::string encodedString;
    for (auto byte : bytes) {
        encodedString += codes[byte];
    }

    int lastByteLength = encodedString.size() % 8;
    while (encodedString.size() % 8 != 0) {
        encodedString += "0"; // 补齐长度为 8 的倍数
    }

    // 将字符串转换为字节
    std::vector<Byte> encodedData;
    for (size_t i = 0; i < encodedString.size(); i += 8) {
        std::string byteString = encodedString.substr(i, 8);

        // string->bitset
        std::bitset<8> bitset(byteString);
        // bitset->ulong->uchar

encodedData.push_back(static_cast<Byte>(bitset.to_ulong()));
    }

    // 若最后一个字节完整 设置为 8

```

```

    if (lastByteLength == 0) lastByteLength = 8;
    // 添加表示最后字节长度的字节 记录信息以便解码
    encodedData.push_back(lastByteLength);

    return encodedData;
}

// 解码数据
std::vector<Byte> decodeHuffman(const std::vector<Byte>&
    encodedData,
                                Node* root) {
    std::vector<Byte> decodedData;
    Node* current = root;

    // 获取最后字节的实际长度
    int lastByteLength = encodedData.back();

    // 遍历除最后一个字节之外的所有字节
    for (size_t i = 0; i < encodedData.size() - 1; i++) {
        std::bitset<8> bits(encodedData[i]);
        // 如果最后的字节, 只处理 lastByteLength 位, 否则处理 8 位
        int limit = (i == encodedData.size() - 2) ? lastByteLength :
8;
        for (int j = 7; j >= 8 - limit; --j) { // 从最高位(左边)开始
            current = bits[j] ? current->right : current->left; // 1
            // 0 往左
            if (current->left == nullptr && current->right == nullptr)
            {
                // 到达叶子结点
                decodedData.push_back(current->data);
                current = root; // 重置根节点
            }
        }
    }
    return decodedData;
}

// 序列化哈夫曼树
void serializeHuffmanTree(Node* root, std::ostream& out) {
    if (!root) { // 叶子节点的子节点 nullptr
        out.put(0);
        return;
    }
}

```

```

// '1'表示节点, 后跟数据
out.put(1);
out.put(root->data);

serializeHuffmanTree(root->left, out);
serializeHuffmanTree(root->right, out);
}

// 反序列化哈夫曼树
Node* deserializeHuffmanTree(std::istream& in) {
    if (!in.good()) return nullptr; // 检查输入流 in 有效

    char flag;
    in.get(flag);
    if (flag == 0) return nullptr; // 空节点

    char data;
    in.get(data);

    Node* node = new Node(data, 0, false); // 不需要频率
    node->left = deserializeHuffmanTree(in);
    node->right = deserializeHuffmanTree(in);

    if (!node->left && !node->right) node->isLeaf = true;

    return node;
}

// 打印哈夫曼树到文件
void printHuffmanTree(Node* root, std::ostream& out, std::string path)
{
    if (!root) return;

    if (root->isLeaf) {
        out << path << " -> " << root->data << " (" << root->frequency
<< ")\n";
    } else {
        printHuffmanTree(root->left, out, path + "0");
        printHuffmanTree(root->right, out, path + "1");
    }
}

int main() {
    std::string filePath;

```

```
std::cout << "请输入要编码的文件的相对路径: ";
std::cin >> filePath;

// 读取原始文件
auto originalData = readFile(filePath);

if (originalData.empty()) return 0;

// 提取文件后缀
std::string extension;
std::string::size_type idx = filePath.rfind('.');
if (idx != std::string::npos && idx !=
filePath.find_last_of("/\\")) {
    extension = filePath.substr(idx); // 包括.的扩展名
}

// 统计频率 Byte=unsigned char=1B
std::unordered_map<Byte, int> freqMap;
for (auto byte : originalData) {
    freqMap[byte]++;
}

// 构建哈夫曼树
int size = freqMap.size();
Byte* data = new Byte[size];
int* freq = new int[size];
int i = 0;
for (auto& pair : freqMap) {
    data[i] = pair.first;
    freq[i] = pair.second;
    i++;
}
Node* root = buildHuffmanTree(data, freq, size);
printCodes(root, "");

// 测量编码时间
auto startEncode = std::chrono::high_resolution_clock::now();
// 进行哈夫曼编码
auto encodedData = encodeHuffman(originalData);
auto endEncode = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> encodeDuration =
    endEncode - startEncode;

writeFile("encoded.bin", encodedData);
```



```
// 打印哈夫曼树的可读形式到文件
std::ofstream treeOutFile("tree.txt");
printHuffmanTree(root, treeOutFile, "");
treeOutFile.close();

// 序列化哈夫曼树 写入文件
std::ofstream treeFile("tree.bin", std::ios::binary);
serializeHuffmanTree(root, treeFile);
treeFile.close();

// 读取编码文件
auto encodedFileData = readFile("encoded.bin");

// 读取并反序列化哈夫曼树
std::ifstream treeFileIn("tree.bin", std::ios::binary);
Node* deserializedRoot = deserializeHuffmanTree(treeFileIn);
treeFileIn.close();

// 测量解码时间
auto startDecode = std::chrono::high_resolution_clock::now();
// 用反序列化的哈夫曼树解码
auto decodedData = decodeHuffman(encodedFileData,
deserializedRoot);
auto endDecode = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> decodeDuration =
    endDecode - startDecode;

writeFile(("decoded" + extension).c_str(), decodedData);

// 计算压缩率
double originalSize = originalData.size();
double compressedSize = encodedData.size();
double compressionRate =
    (originalSize - compressedSize) / originalSize * 100;
// if (compressionRate < 1e-3) compressionRate = 0;

std::cout << "原始文件: " << filePath << std::endl;
std::cout << "编码后文件: encoded.bin" << std::endl;
std::cout << "哈夫曼树文件: tree.bin" << std::endl;
std::cout << "哈夫曼树可读形式: tree.txt" << std::endl;
std::cout << "解码后文件: decoded" << extension << std::endl;
std::cout << "压缩率: " << compressionRate << "%" << std::endl;
std::cout << "编码用时: " << encodeDuration.count() << " 毫秒" <<
```

```
std::endl;
    std::cout << "解码用时: " << decodeDuration.count() << " 毫秒" <<
std::endl;

    // macOS 系统调用
    system(("open " + filePath).c_str()); // 原始文件
    system("open tree.txt");              // 哈夫曼树的可读形式
    system(("open decoded" + extension).c_str()); // 解码后的文件

    delete[] data;
    delete[] freq;

    return 0;
}
```

7.2 参考文献

- [1] Moffat A. Huffman coding[J]. ACM Computing Surveys (CSUR), 2019, 52(4): 1-35.
- [2] Katona G, Nemetz O. Huffman codes and self-information[J]. IEEE Transactions on Information Theory, 1976, 22(3): 337-340.
- [3] 朱怀宏, 吴楠, 夏黎春. 利用优化哈夫曼编码进行数据压缩的探索[J]. 微机发展, 2002, 12(5): 1-6.
- [4] 王群芳. 哈夫曼编码的另一种实现算法[J]. 安徽教育学院学报, 2006, 24(6): 36-38.
- [5] 田端财, 殷晓丽. 基于哈夫曼编码的图像压缩技术研究[J]. 科技资讯, 2009 (8): 29-30.