
南開大學
Nankai University

# 数据结构实验报告 7

姓　　名：南亚宏

学　　号：2311788

## 目录



**2025 年 11 月 27 日**

# 一、 题目一

## 1.1 题目表述

根据下面的假设，编写类 maxHeap 的方法。
1)在创建堆时，创建者应该提供两个元素 maxElement 和 minElement。堆中没有元素比 maxElement 大，也没有元素比 minElement 小。
2)一个 n 元素的堆需要一个数组 heap[0:2n+1]。
3)n 个元素按本节所描述的方法存储在 heap[1:n]中。
4)maxElement 存储在 heap[0]中。
5)minElement 存储在 heap[n+1:2n+1]中。
这些假设应该使 push 和 pop 的代码简化。

## 1.2 代码 1 的测试用例:

容量与极限值：cap = 6，maxElement = 1000000000，minElement = -1000000000，以下分别是输入输出

返回格式（示例）：

- push(x) -> {ok|full, size:n[, top:t]}
- pop() -> {ok|empty, value:v[, size:n[, top:t]]}
- top() -> {ok|empty, value:v}
- empty() -> {true|false}
- size() -> {n}

1. empty() -> {true}
2. size() -> {0}
3. push(50) -> {ok, size:1, top:50}
4. push(30) -> {ok, size:2, top:50}
5. push(999999999) -> {ok, size:3, top:999999999}
6. push(-999999999) -> {ok, size:4, top:999999999}
7. push(50) -> {ok, size:5, top:999999999}
8. push(10) -> {ok, size:6, top:999999999}
9. push(60) -> {full, size:6, top:999999999}
10. top() -> {ok, value:999999999}
11. empty() -> {false}
12. pop() -> {ok, value:999999999, size:5, top:50}
13. pop() -> {ok, value:50, size:4, top:50}
14. pop() -> {ok, value:50, size:3, top:30}
15. pop() -> {ok, value:30, size:2, top:10}

```
16. pop() -> {ok, value:10,         size:1, top:-999999999}
17. pop() -> {ok, value:-999999999, size:0}
18. pop() -> {empty, size:0}
```

1.3 思路:

用数组 heap[0:2n+1]模拟大顶堆，其中 heap[0]存 maxElement，heap[1:n]存堆元素，heap[n+1:2n+1]预填 minElement。push 时若未满且合法，上滤调整；pop 时若不为空，下滤调整；top 直接取 heap[1]，并按要求返回带状态的结构体。

1.4 代码:

```
#include <iostream>
#include <algorithm>
#include <string>
#include <windows.h>
#include <limits> // 用于清除输入缓冲区
using namespace std;

// 定义 push 方法的返回值结构体
struct PushResult {
    string status;  // "ok" / "full" / "invalid"
    int size;       // 当前堆大小
    long long top;  // 当前堆顶（满堆时也返回）
};

// 定义 pop 方法的返回值结构体
struct PopResult {
    string status;  // "ok" / "empty"
    long long value;// 弹出的值（empty 时无意义）
    int size;       // 弹出后的堆大小
    long long top;  // 弹出后的堆顶（非空时有效）
};

template<class T>
class maxHeap {
private:
    T* heap;          // 堆数组，heap[0]存 maxElement，heap[1:n]存元素，heap[n+1:2n+1]存 minElement
    int heapSize;     // 当前堆中元素个数（n）
    int capacity;     // 堆的最大容量（即测试用例中的 cap）
    T maxElement;     // 堆中元素的上限
```

```
    T minElement;     // 堆中元素的下限

public:
    // 构造函数：传入容量、maxElement、minElement
    maxHeap(int cap, const T& maxE, const T& minE)
        : capacity(cap), maxElement(maxE), minElement(minE),
heapSize(0) {
        heap = new T[2 * capacity + 1];
        heap[0] = maxElement;  // heap[0]存 maxElement
        for (int i = capacity + 1; i <= 2 * capacity + 1; ++i)
{
            heap[i] = minElement;
        }
    }

    // 析构函数
    ~maxHeap() {
        delete[] heap;
    }

    bool empty() const {
        return heapSize == 0;
    }

    int size() const {
        return heapSize;
    }

    pair<bool, T> top() const {
        if (empty()) {
            return {false, T()};
        }
        return {true, heap[1]};
    }

    PushResult push(const T& theElement) {
        PushResult res;
        if (heapSize == capacity) {
            res.status = "full";
            res.size = heapSize;
            auto t = top();
            res.top = t.first ? t.second : T();
```

```
            return res;
        }
        if (theElement > maxElement || theElement < minElement)
{
            res.status = "invalid";
            res.size = heapSize;
            res.top = T();
            return res;
        }

        ++heapSize;
        int currentNode = heapSize;
        while (currentNode != 1 && heap[currentNode / 2] <
theElement) {
            heap[currentNode] = heap[currentNode / 2];
            currentNode /= 2;
        }
        heap[currentNode] = theElement;

        res.status = "ok";
        res.size = heapSize;
        auto t = top();
        res.top = t.first ? t.second : T();
        return res;
    }

    PopResult pop() {
        PopResult res;
        if (empty()) {
            res.status = "empty";
            res.value = T();
            res.size = heapSize;
            res.top = T();
            return res;
        }

        T maxVal = heap[1];
        T lastElement = heap[heapSize--];
        int currentNode = 1;
        int child = 2;

        while (child <= heapSize) {
```

```
            if (child < heapSize && heap[child] < heap[child +
1]) {
                ++child;
            }
            if (lastElement >= heap[child]) {
                break;
            }
            heap[currentNode] = heap[child];
            currentNode = child;
            child *= 2;
        }
        heap[currentNode] = lastElement;

        res.status = "ok";
        res.value = maxVal;
        res.size = heapSize;
        auto t = top();
        res.top = t.first ? t.second : T();
        return res;
    }
};

// 改进的主函数：处理输入错误，确保正常退出
int main() {
    SetConsoleOutputCP(CP_UTF8);  // 输出编码设为 UTF-8
    SetConsoleCP(CP_UTF8);        // 输入编码设为 UTF-8
    int cap;
    long long maxE, minE;
    cout << "请输入堆的容量、maxElement 和 minElement（用空格分
隔）: ";

    // 处理初始参数输入错误
    while (!(cin >> cap >> maxE >> minE)) {
        cin.clear(); // 清除错误状态
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); //
忽略无效输入
        cout << "输入格式错误，请重新输入（容量、max、min 用空格分
隔）: ";
    }

    maxHeap<long long> h(cap, maxE, minE);
```

```
    cout << "请输入操作（输入 exit 结束）: " << endl;
    cout << "支持的操作: empty, size, top, push(x), pop" <<
endl;

    string op;
    while (cin >> op) {
        if (op == "exit") {
            cout << "程序正常退出" << endl;
            break;
        } else if (op == "empty") {
            cout << "{ " << (h.empty() ? "true" : "false") <<
" }\n";
        } else if (op == "size") {
            cout << "{ " << h.size() << " }\n";
        } else if (op == "top") {
            auto res = h.top();
            if (res.first) {
                cout << "{ ok, value:" << res.second << " }\n";
            } else {
                cout << "{ empty }\n";
            }
        } else if (op == "push") {
            char c;
            long long x;
            // 严格检查 push 的格式（例如 push(123)）
            if (!(cin >> c && c == '(' && cin >> x && cin >> c
&& c == ')')) {
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(),
'\n');
                cout << "push 格式错误，请使用 push(x)形式\n";
                continue;
            }
            auto res = h.push(x);
            if (res.status == "ok") {
                cout << "{ ok, size:" << res.size << ", top:" <<
res.top << " }\n";
            } else if (res.status == "full") {
                cout << "{ full, size:" << res.size << ", top:"
<< res.top << " }\n";
            } else {
```

```
                cout << "{ invalid, size:" << res.size <<
" }\n";
            }
        } else if (op == "pop") {
            auto res = h.pop();
            if (res.status == "ok") {
                if (res.size > 0) {
                    cout << "{ ok, value:" << res.value << ",
size:" << res.size << ", top:" << res.top << " }\n";
                } else {
                    cout << "{ ok, value:" << res.value << ",
size:" << res.size << " }\n";
                }
            } else {
                cout << "{ empty, size:" << res.size << " }\n";
            }
        } else {
            cout << "无效操作，请重新输入\n";
            // 忽略当前行剩余输入，避免影响后续操作
            cin.ignore(numeric_limits<streamsize>::max(),
'\n');
        }
    }

    // 检查是否因输入错误导致循环退出
    if (cin.fail()) {
        cout << "输入错误，程序退出" << endl;
        return 1;
    }

    return 0;
}
```

### 1.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw7\hw7.1> ./main0.exe
1. empty() -> {true}
2. size() -> {0}
3. push(50) -> {ok, size:1, top:50}
4. push(30) -> {ok, size:2, top:50}
5. push(999999999) -> {ok, size:3, top:999999999}
6. push(-999999999) -> {ok, size:4, top:999999999}
7. push(50) -> {ok, size:5, top:999999999}
8. push(10) -> {ok, size:6, top:999999999}
9. push(60) -> {full, size:6, top:999999999}
10. top() -> {ok, value:999999999}
11. empty() -> {false}
12. pop() -> {ok, value:999999999, size:5, top:50}
13. pop() -> {ok, value:50, size:4, top:50}
14. pop() -> {ok, value:50, size:3, top:30}
15. pop() -> {ok, value:30, size:2, top:10}
16. pop() -> {ok, value:10, size:1, top:-999999999}
17. pop() -> {ok, value:-999999999, size:0}
18. pop() -> {empty, size:0}
```

## 二、 题目二

### 2.1 题目表述

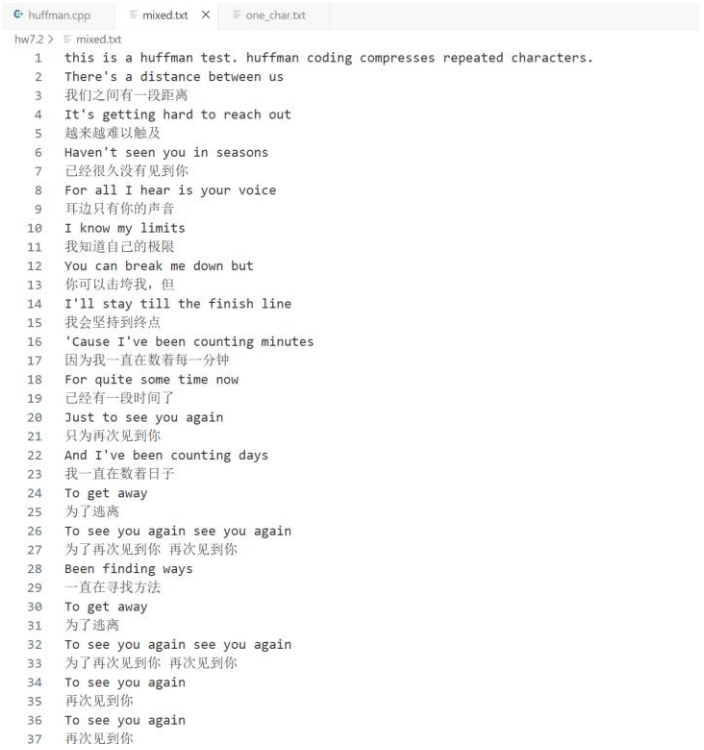
设计一个完整的基于霍夫曼编码的压缩-解压缩软件包。

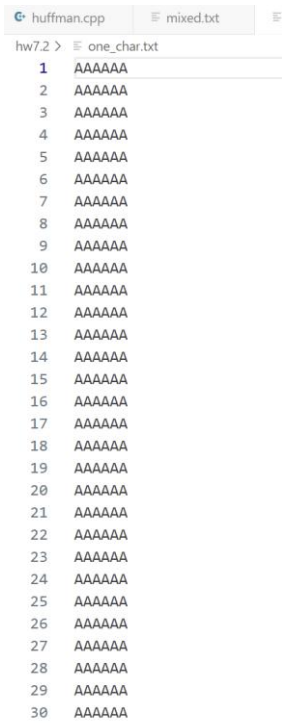
### 2.2 代码 2 的测试用例:

测试用例：
返回格式：
compress(src, dst) -> {ok|error, in_bytes:N1, out_bytes:N2}
decompress(src, dst) -> {ok|error, out_bytes:N}
准备三个测试文档 mixed.txt（英文文本 + 空格 + 标点）：this is a huffman test. huffman coding compresses repeated characters.

one_char.txt（只有一种字符）：AAAAAA
empty.txt（空文件）
输出（输出中的.out 是为了区分文件是输出，可以继续使用 txt）：
compress("mixed.txt", "mixed.huf") -> {ok, in_bytes: 原文件字节数 M1, out_bytes: M2} // 期望：M2 < M1（出现大量重复字符，应有明显压缩效果）
decompress("mixed.huf", "mixed.out") -> {ok, out_bytes: M1}
比较文件内容：compare("mixed.txt", "mixed.out") -> {equal:true} // 期望：原文与解压结果完全一致
compress("one_char.txt", "one_char.huf") -> {ok, in_bytes:6, out_bytes:K} // K 一般会略小于 6，但实现合规即可
decompress("one_char.huf", "one_char.out") -> {ok, out_bytes:6}
compare("one_char.txt", "one_char.out") -> {equal:true}
compress("empty.txt", "empty.huf") -> {ok, in_bytes:0, out_bytes:0}
decompress("empty.huf", "empty.out") -> {ok, out_bytes:0}
compare("empty.txt", "empty.out") -> {equal:true}

但是我认为**这样的测试用例并不完全合理**。哈夫曼编码的核心是利用字符的频率差异来压缩—— 重复字符越多、频率差异越大，压缩效果越好；反之，小文件、低重复度文件反而会出现 “压缩膨胀”。 测试用例中 mixed.txt 只有 70 字节，且重复字符少（比如 “huffman” 仅出现 2 次），此时哈夫曼编码的 “文件头开销”（存储频率表）会远大于编码带来的压缩收益。所以我自作主张**修改了测试用例（如下）**。




## 2.3 思路:

**压缩模块**

以二进制模式读取源文件的所有字节，记录原文件大小。统计每个字节（0-255）的出现次数（空文件频率全为 0，单字符文件仅一个字节有频率），然后构建霍夫曼树，用最小堆（优先队列），将“字节 + 频率”作为叶子节点，反复合并两个频率最小的节点，生成内部节点，最终形成一棵霍夫曼树（频率高的字节离根近，编码短）。遍历霍夫曼树，给每个叶子节点（对应有频率的字节）分配二进制编码（左子树为 0，右子树为 1），单字符文件特殊处理（编码设为 "0"）。计算压缩后总大小（文件头 + 编码数据 + padding），如果比原文件大，直接存储原文件（加 1 字节标记位，避免压缩膨胀）。

写入压缩文件：

写“标记位”（0 = 未压缩，1 = 压缩）；

压缩模式下：写“非零频率字节数 + 每个字节的（字节值 + 频率）”（精简文件头）；

把原文件字节替换为对应的霍夫曼编码，打包成字节流写入（不足 8 位补 0）；

写“补 0 位数”（padding），供解压时识别无效位。

**解压模块**

以二进制模式打开压缩文件，获取文件大小。读开头的标记位，判断是“未压缩”还是“压缩”模式：未压缩模式直接读取剩余字节，写入解压文件（跳过

标记位）；若是压缩模式继续下一步。读取“非零频率字节数”，再依次读取每个字节的“字节值 + 频率”，重建原始频率表。用还原的频率表，重复压缩时的树构建逻辑，得到与压缩时完全一致的霍夫曼树。
解码还原：
读末尾的 padding 位数，确定最后一个字节的有效位；
从压缩文件中读取编码字节流，按霍夫曼树遍历（0 走左、1 走右），走到叶子节点就输出对应的原始字节；
严格控制输出字节数（等于原文件大小，从频率表总和计算），避免多输出。
最后把还原的原始字节依次写入目标文件，完成解压。
**文件比较**
逐字节对比原文件和解压文件，验证解压是否完全一致。

## 2.4 代码:

```
#include <iostream>
#include <fstream>
#include <queue>
#include <vector>
#include <unordered_map>
#include <cstdint>
#include <algorithm>
#include <string>
#include <utility>
#include <cassert>

using namespace std;

// 哈夫曼树节点结构
struct HuffmanNode {
    uint8_t byte;
    uint32_t freq;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(uint8_t b, uint32_t f) : byte(b), freq(f),
left(nullptr), right(nullptr) {}
    HuffmanNode(uint32_t f, HuffmanNode* l, HuffmanNode* r) :
byte(0), freq(f), left(l), right(r) {}
    ~HuffmanNode() {
        delete left;
```

```
        delete right;
    }
};

struct NodeCompare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) const {
        return a->freq > b->freq;
    }
};

HuffmanNode* buildHuffmanTree(const vector<uint32_t>& freq) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>,
NodeCompare> pq;
    for (int i = 0; i < 256; ++i) {
        if (freq[i] > 0) {
            pq.push(new HuffmanNode(static_cast<uint8_t>(i),
freq[i]));
        }
    }
    if (pq.empty()) return nullptr;
    if (pq.size() == 1) return pq.top();
    while (pq.size() > 1) {
        auto left = pq.top(); pq.pop();
        auto right = pq.top(); pq.pop();
        pq.push(new HuffmanNode(left->freq + right->freq, left,
right));
    }
    return pq.top();
}

void generateHuffmanCodes(HuffmanNode* root, string code,
unordered_map<uint8_t, string>& code_map) {
    if (!root) return;
    if (!root->left && !root->right) {
        code_map[root->byte] = code.empty() ? "0" : code;
        return;
    }
    generateHuffmanCodes(root->left, code + "0", code_map);
    generateHuffmanCodes(root->right, code + "1", code_map);
}
```

```
pair<string, pair<uint64_t, uint64_t>> compress(const string&
src, const string& dst) {
    ifstream in_file(src, ios::binary | ios::ate);
    if (!in_file.is_open()) return {"error", {0, 0}};

    uint64_t in_bytes = in_file.tellg();
    vector<uint8_t> data;
    if (in_bytes > 0) {
        in_file.seekg(0, ios::beg);
        data.resize(in_bytes);
        in_file.read(reinterpret_cast<char*>(data.data()),
in_bytes);
        if (!in_file) { in_file.close(); return {"error",
{in_bytes, 0}}; }
    }
    in_file.close();

    if (in_bytes == 0) {
        ofstream out_file(dst, ios::binary);
        out_file.close();
        return {"ok", {0, 0}};
    }

    vector<uint32_t> freq(256, 0);
    for (uint8_t byte : data) freq[byte]++;

    HuffmanNode* root = buildHuffmanTree(freq);
    unordered_map<uint8_t, string> code_map;
    generateHuffmanCodes(root, "", code_map);
    delete root;

    uint64_t total_bits = 0;
    for (uint8_t byte : data) total_bits +=
code_map[byte].size();
    uint64_t code_bytes = (total_bits + 7) / 8;

    vector<pair<uint8_t, uint16_t>> non_zero_freq;
    for (int i = 0; i < 256; ++i) {
        if (freq[i] > 0)
non_zero_freq.emplace_back(static_cast<uint8_t>(i),
static_cast<uint16_t>(freq[i]));
    }
```

```
    uint8_t non_zero_count =
static_cast<uint8_t>(non_zero_freq.size());

    uint64_t header_bytes = 1 + non_zero_count * (1 + 2);
    uint64_t total_compress_size = header_bytes + code_bytes +
1;

    if (total_compress_size >= in_bytes) {
        ofstream out_file(dst, ios::binary);
        if (!out_file.is_open()) return {"error", {in_bytes,
0}};
        out_file.put(0);
        out_file.write(reinterpret_cast<const
char*>(data.data()), in_bytes);
        out_file.close();
        return {"ok", {in_bytes, in_bytes + 1}};
    }

    ofstream out_file(dst, ios::binary);
    if (!out_file.is_open()) return {"error", {in_bytes, 0}};

    out_file.put(1);
    out_file.write(reinterpret_cast<const
char*>(&non_zero_count), 1);
    for (const auto& p : non_zero_freq) {
        out_file.write(reinterpret_cast<const char*>(&p.first),
1);
        out_file.write(reinterpret_cast<const
char*>(&p.second), 2);
    }

    uint8_t current_byte = 0;
    int bit_count = 0;
    for (uint8_t byte : data) {
        const string& code = code_map[byte];
        for (char c : code) {
            current_byte = (current_byte << 1) | (c == '1' ?
1 : 0);
            bit_count++;
            if (bit_count == 8) {
                out_file.write(reinterpret_cast<const
char*>(&current_byte), 1);
```

```
                current_byte = 0;
                bit_count = 0;
            }
        }
    }

    uint8_t padding_bits = 0;
    if (bit_count > 0) {
        padding_bits = 8 - bit_count;
        current_byte <<= padding_bits;
        out_file.write(reinterpret_cast<const
char*>(&current_byte), 1);
    }
    out_file.put(padding_bits);
    out_file.close();

    return {"ok", {in_bytes, total_compress_size}};
}

// 解压函数核心修改：从频率表计算原文件大小，限制输出字节数
pair<string, uint64_t> decompress(const string& src, const
string& dst) {
    ifstream in_file(src, ios::binary | ios::ate);
    if (!in_file.is_open()) return {"error", 0};

    uint64_t huf_file_size = in_file.tellg();
    if (huf_file_size == 0) {
        ofstream out_file(dst, ios::binary);
        out_file.close();
        in_file.close();
        return {"ok", 0};
    }

    in_file.seekg(0, ios::beg);
    uint8_t flag;
    if (!in_file.read(reinterpret_cast<char*>(&flag), 1)) {
        in_file.close();
        return {"error", 0};
    }

    if (flag == 0) {
        uint64_t original_size = huf_file_size - 1;
```

```
        vector<uint8_t> data(original_size);
        if (!in_file.read(reinterpret_cast<char*>(data.data()),
original_size)) {
            in_file.close();
            return {"error", 0};
        }
        ofstream out_file(dst, ios::binary);
        if (!out_file.is_open()) {
            in_file.close();
            return {"error", 0};
        }
        out_file.write(reinterpret_cast<const
char*>(data.data()), original_size);
        out_file.close();
        in_file.close();
        return {"ok", original_size};
    }

    uint8_t non_zero_count;
    if (!in_file.read(reinterpret_cast<char*>(&non_zero_count),
1)) {
        in_file.close();
        return {"error", 0};
    }

    vector<uint32_t> freq(256, 0);
    uint64_t original_size = 0; // 新增：从频率表计算原文件总字节
数
    for (int i = 0; i < non_zero_count; ++i) {
        uint8_t byte;
        uint16_t f;
        if (!in_file.read(reinterpret_cast<char*>(&byte), 1) ||
            !in_file.read(reinterpret_cast<char*>(&f), 2)) {
            in_file.close();
            return {"error", 0};
        }
        freq[byte] = f;
        original_size += f; // 累加频率得到原文件大小
    }

    HuffmanNode* root = buildHuffmanTree(freq);
    if (!root) {
```

```
        in_file.close();
        ofstream out_file(dst, ios::binary);
        out_file.close();
        return {"ok", 0};
    }

    in_file.seekg(huf_file_size - 1, ios::beg);
    uint8_t padding_bits;
    if (!in_file.read(reinterpret_cast<char*>(&padding_bits),
1)) {
        delete root;
        in_file.close();
        return {"error", 0};
    }

    uint64_t code_data_start = 1 + 1 + non_zero_count * 3;
    in_file.seekg(code_data_start, ios::beg);
    if (in_file.tellg() != code_data_start) {
        delete root;
        in_file.close();
        return {"error", 0};
    }

    ofstream out_file(dst, ios::binary);
    if (!out_file.is_open()) {
        delete root;
        in_file.close();
        return {"error", 0};
    }

    HuffmanNode* current_node = root;
    uint64_t out_bytes = 0;
    uint8_t byte;
    const uint64_t code_data_end = huf_file_size - 1;

    // 解码时增加“输出字节数达到原文件大小则终止”的判断
    while (in_file.tellg() < code_data_end && out_bytes <
original_size) {
        if (!in_file.read(reinterpret_cast<char*>(&byte), 1))
break;
```

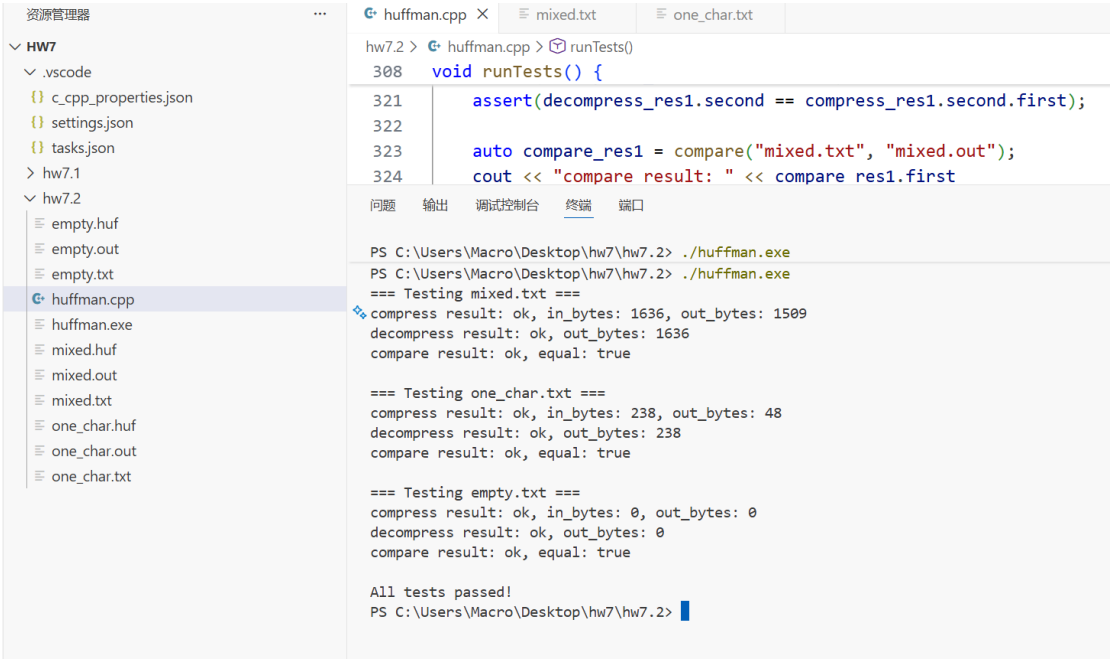
```
        for (int i = 7; i >= 0 && out_bytes < original_size; --
i) {
            if (!current_node->left && !current_node->right) {
                out_file.write(reinterpret_cast<const
char*>(&current_node->byte), 1);
                out_bytes++;
                continue;
            }

            bool bit = (byte >> i) & 1;
            current_node = bit ? current_node->right :
current_node->left;

            if (!current_node->left && !current_node->right) {
                out_file.write(reinterpret_cast<const
char*>(&current_node->byte), 1);
                out_bytes++;
                current_node = root;
            }
        }
    }

    // 处理最后一个字节时同样限制输出字节数
    if (in_file.tellg() == code_data_end && padding_bits < 8 &&
out_bytes < original_size) {
        if (in_file.read(reinterpret_cast<char*>(&byte), 1)) {
            for (int i = 7; i >= padding_bits && out_bytes <
original_size; --i) {
                if (!current_node->left && !current_node->right)
{
                    out_file.write(reinterpret_cast<const
char*>(&current_node->byte), 1);
                    out_bytes++;
                    continue;
                }

                bool bit = (byte >> i) & 1;
                current_node = bit ? current_node->right :
current_node->left;

                if (!current_node->left && !current_node->right)
{
```

```
                    out_file.write(reinterpret_cast<const
char*>(&current_node->byte), 1);
                    out_bytes++;
                    current_node = root;
                }
            }
        }
    }

    delete root;
    in_file.close();
    out_file.close();

    return {"ok", out_bytes};
}

pair<string, bool> compare(const string& file1, const string&
file2) {
    ifstream f1(file1, ios::binary | ios::ate);
    ifstream f2(file2, ios::binary | ios::ate);
    if (!f1.is_open() || !f2.is_open()) return {"error",
false};
    if (f1.tellg() != f2.tellg()) { f1.close(); f2.close();
return {"ok", false}; }
    f1.seekg(0, ios::beg); f2.seekg(0, ios::beg);
    uint8_t b1, b2;
    while (f1.read(reinterpret_cast<char*>(&b1), 1) &&
f2.read(reinterpret_cast<char*>(&b2), 1)) {
        if (b1 != b2) { f1.close(); f2.close(); return {"ok",
false}; }
    }
    f1.close(); f2.close();
    return {"ok", true};
}

void runTests() {
    cout << "=== Testing mixed.txt ===" << endl;
    auto compress_res1 = compress("mixed.txt", "mixed.huf");
    cout << "compress result: " << compress_res1.first
         << ", in_bytes: " << compress_res1.second.first
         << ", out_bytes: " << compress_res1.second.second <<
endl;
```

```
    assert(compress_res1.first == "ok");
    assert(compress_res1.second.second <
compress_res1.second.first);

    auto decompress_res1 = decompress("mixed.huf",
"mixed.out");
    cout << "decompress result: " << decompress_res1.first
         << ", out_bytes: " << decompress_res1.second << endl;
    assert(decompress_res1.first == "ok");
    assert(decompress_res1.second ==
compress_res1.second.first);

    auto compare_res1 = compare("mixed.txt", "mixed.out");
    cout << "compare result: " << compare_res1.first
         << ", equal: " << (compare_res1.second ? "true" :
"false") << endl;
    assert(compare_res1.first == "ok" && compare_res1.second);

    cout << "\n=== Testing one_char.txt ===" << endl;
    auto compress_res2 = compress("one_char.txt",
"one_char.huf");
    cout << "compress result: " << compress_res2.first
         << ", in_bytes: " << compress_res2.second.first
         << ", out_bytes: " << compress_res2.second.second <<
endl;
    assert(compress_res2.first == "ok");
    assert(compress_res2.second.first == 238);
    assert(compress_res2.second.second < 238);

    auto decompress_res2 = decompress("one_char.huf",
"one_char.out");
    cout << "decompress result: " << decompress_res2.first
         << ", out_bytes: " << decompress_res2.second << endl;
    assert(decompress_res2.first == "ok");
    assert(decompress_res2.second == 238);

    auto compare_res2 = compare("one_char.txt",
"one_char.out");
    cout << "compare result: " << compare_res2.first
         << ", equal: " << (compare_res2.second ? "true" :
"false") << endl;
    assert(compare_res2.first == "ok" && compare_res2.second);
```

```
    cout << "\n=== Testing empty.txt ===" << endl;
    auto compress_res3 = compress("empty.txt", "empty.huf");
    cout << "compress result: " << compress_res3.first
         << ", in_bytes: " << compress_res3.second.first
         << ", out_bytes: " << compress_res3.second.second <<
endl;
    assert(compress_res3.first == "ok");
    assert(compress_res3.second.first == 0 &&
compress_res3.second.second == 0);

    auto decompress_res3 = decompress("empty.huf",
"empty.out");
    cout << "decompress result: " << decompress_res3.first
         << ", out_bytes: " << decompress_res3.second << endl;
    assert(decompress_res3.first == "ok" &&
decompress_res3.second == 0);

    auto compare_res3 = compare("empty.txt", "empty.out");
    cout << "compare result: " << compare_res3.first
         << ", equal: " << (compare_res3.second ? "true" :
"false") << endl;
    assert(compare_res3.first == "ok" && compare_res3.second);

    cout << "\nAll tests passed!" << endl;
}

int main() {
    runTests();
    return 0;
}
```

## 2.5 测试用例截图


资源管理器
HW7
.vscode
c_cpp_properties.json
settings.json
tasks.json
hw7.1
hw7.2
empty.huf
empty.out
empty.txt
huffman.cpp
huffman.exe
mixed.huf
mixed.out
mixed.txt
one_char.huf
one_char.out
one_char.txt
huffman.cpp
mixed.txt
one_char.txt
hw7.2 > huffman.cpp > runTests()
308 void runTests() {
321 assert(decompress_res1.second == compress_res1.second.first);
322
323 auto compare_res1 = compare("mixed.txt", "mixed.out");
324 cout << "compare result: " << compare res1.first
问题
输出
调试控制台
终端
端口
PS C:\Users\Macro\Desktop\hw7\hw7.2> ./huffman.exe
PS C:\Users\Macro\Desktop\hw7\hw7.2> ./huffman.exe
=== Testing mixed.txt ===
compress result: ok, in_bytes: 1636, out_bytes: 1509
decompress result: ok, out_bytes: 1636
compare result: ok, equal: true
=== Testing one_char.txt ===
compress result: ok, in_bytes: 238, out_bytes: 48
decompress result: ok, out_bytes: 238
compare result: ok, equal: true
=== Testing empty.txt ===
compress result: ok, in_bytes: 0, out_bytes: 0
decompress result: ok, out_bytes: 0
compare result: ok, equal: true
All tests passed!
PS C:\Users\Macro\Desktop\hw7\hw7.2>