学生学号		实验课成绩	

武漢理工大學

学 生 实 验 报 告 书

实验课程名称 _____

开课学院 _____

指导教师姓名 _____

学生姓名 _____

学生专业班级 _____

2024 — 2025 学年 第 2 学期

实验课程名称：数据结构与算法综合实验

实验项目名称	二叉树与哈夫曼图片压缩			实验成绩	
实验者		专业班级		组别	
同组者				实验日期	2025 年 2 月 27 日

第一部分：实验分析与设计（可加页）

一、实验内容描述（问题域描述）

使用 Huffman 压缩算法，对一幅 BMP 格式的图片文件进行压缩。图片文件名为“Pic.bmp”,压缩后保存为“Pic.bmp.huf”文件。使用 VS Studio 作为开发工具，开发一个控制台程序，使用 Huffman 压缩算法对图片文件“Pic.bmp”进行压缩。

具体要求如下：

(1) 读取原文件，统计权值：

运行程序，输入文件名。

以“Pic.bmp”文件为例。若文件存放在 F 盘根目录下，输入文件完整路径“F： \Pic.bmp”。按回车结束。以字节流的方式，只读打“Pic.bmp”文件。

逐字节读取文件，统计文件中 256 种字节重复的次数，保存到一个数组中 int weight[256]中。

(2) 生成 Huffman 树

根据（1）中统计的结果，构建 Huffman 树。定义一个结构体来记录每个节点的权值、父节点、左孩子和右孩子。使用结构体数组来存储这个 Huffman 树。

(3) 生成 Huffman 编码

遍历（2）中生成的 Huffman 树，记录 256 个叶子节点的 Huffman 编码，保存在字符串数组中。

(4) 压缩原文件

使用 Huffman 编码对原文件中的字节重新编码，获得压缩后的文件数据。

(5) 保存压缩文件

将编码过的数据，保存到文件“Pic.bmp.huf”中。

二、实验基本原理与设计

本节分为 A、B 两部分，分别是**压缩**和**解压缩**部分的相关算法设计与流程分析。

A. 压缩部分的算法设计

1. 构造哈夫曼树、哈夫曼编码

实验中，先读取 BMP 文件的内部字节（一共有 256 种），并统计其重复的次数作为权值。根据这一权值信息构造哈夫曼树，进而构造出哈夫曼编码。

读入权值数组的算法实现如下。最终构造出了权值数组 weights（长度为 256）。

```
string filename;
cout << "====== Huffman 文件压缩 ======" << endl;
cout << "请输入文件名：";
cin >> filename;
```

```
ifstream file;
file.open(filename, ios::binary);

// 统计并输出文件大小
int filesize = (int) filesystem::file_size(filename);
cout << "文件大小：" << filesize << "B" << endl;

int ch;
int weights[bits] = {0};
while ((ch = file.get()) != EOF) {
    weights[ch]++;
}
```

其中，bits 为一宏，它指定的是该文件中所包含的不同字节的**种类数**。在本实验中，规定 bits 的值为 256，此后不再发生变化。

压缩的基础是基于权值数组构造出哈夫曼树。根据数据结构的相关知识，可以写出相关的算法。首先，定义哈夫曼树的节点结构。在本实验中通过 C++ Class 实现这一点。

```
class HuffmanNode {
public:
    int weight;
    int parent;
    int lchild;
    int rchild;

    explicit HuffmanNode(int w) : weight(w), parent(-1), lchild(-1), rchild(-1) {};
};
```

该类的相关属性解释如下：

- ✧ weight：当前节点所对应的权值
- ✧ parent：当前节点的父节点在哈夫曼树的顺序结构表示中的下标
- ✧ lchild、rchild：当前节点的左（右）孩子在哈夫曼树的顺序结构表示中的下标

约定下标从 0 开始。

n 个权值所构建出的哈夫曼树一共有 2n-1 个节点。本报告中使用顺序结构来表示哈夫曼树。首先构造一个长度为 2n-1（在本实验中，是 2*bits-1）的 HuffmanNode*数组，然后在其之上做修改。构造哈夫曼树的相关代码实现如下：

```
void buildHuffmanTree(HuffmanNode *tree[], int weights[], int size) {
    for (int i = 0; i < size; i++) tree[i] = new HuffmanNode(weights[i]);
    // 注意：哈夫曼树的最后一个节点下标为 size*2-2。
    for (int i = size; i < size * 2 - 1; i++) tree[i] = new HuffmanNode(0);

    for (int i = size; i < size * 2 - 1; i++) {
        auto targets = select(tree, i - 1);
```

```
            tree[i]->weight = tree[targets.first]->weight + tree[targets.second]->weight;
            tree[i]->lchild = targets.first;
            tree[i]->rchild = targets.second;
            tree[targets.first]->parent = i;
            tree[targets.second]->parent = i;
        }
}
```

其中 select 函数返回的是哈夫曼树 0~n-1 位中最小的两个节点的下标，使用 std::pair<int, int> 接纳。

构造出哈夫曼树以后，就可以根据每一个权值在哈夫曼树中的位置构造一个哈夫曼编码。此时约定向左为 0，向右为 1。考虑将编码存储在一个长度为 256 的 std::string 数组中。哈夫曼编码函数实现如下：

```
void genHuffmanCode(string code[], HuffmanNode *tree[], int size) {
    for (int i = 0; i < size; i++) {
            string c;
            int current = i;
            int ptr = tree[current]->parent;

            while (ptr != -1) {
                if (tree[ptr]->lchild == current) c.insert(0, "0");
                else if (tree[ptr]->rchild == current) c.insert(0, "1");

                current = ptr;
                ptr = tree[ptr]->parent;
            }

            code[i] = c;
    }
}
```

由于此算法的构造过程是从叶子节点向上回溯实现的，因此用到了 c.insert(0, "0"或者"1")。这是为了在当前哈夫曼编码的**开头**插入编码的比特内容，确保顺序是正确的。

2. 利用哈夫曼编码对 BMP 文件内容进行映射和压缩

得到了每一种字节所对应的哈夫曼编码以后，我们可以对文件的内容一一进行映射。也就是说，如果文件原本的比特内容是这样的（一个数字表示一字节）：

```
1 3 57 43 29 57
```

就可以将其映射到对应的哈夫曼编码：

```
0001 10000 0101001 11011 1101 0101001
```

由于我们先前的操作从一开始就将字节与下标进行了一一映射，因此只需访问我们构造出的哈夫曼编码数组的对应下标，就可以得到正确的哈夫曼编码。例如上面的例子中的哈夫曼编码就可以用下面的下标访问来表示：

```
code[1] code[3] code[57] code[43] code[29] code[57]
```

我们将最终构造出的只含 0 和 1 的字符串命名为 bitstream（字节流），根据以上分析就可以编写出以下构造 bitstream 的具体方式：

```
ifstream in;
in.open(filename, ios::binary);
int ch;
string bitstream;

while ((ch = in.get()) != EOF) {
    bitstream += code[ch];
}
```

得到的 bitstream 就是每一个字节映射到哈夫曼编码的结果。然而，这样形成的新文件似乎还会让得到的文件变大，并没有起到“压缩”的效果。

为了达到压缩的效果，我们可以通过逐 8 个读取 bitstream 字符串中的 0 与 1，将这一个 8 位字符串看成是一个**八位二进制数**，进而将其压缩为一个 0~255 范围内的字节。这一过程也是可逆的，可为后续解压做铺垫。由此分析，最好的情况下我们可将编码后的文件压缩到原有的 1/8。

实现这一算法的过程中，我们需要首先定义一个 cursor（指标、指针），用于指示当前我们在字符串中所处的位置，默认是 0。通过 cursor 这一参数，可调用 std::string 的 substr 方法来提取正确的子串。每读取 8 位后，就将 cursor 后移 8 位，由此循环往复，直到可读字符数不满 8 位，就取剩余的所有字符。这一点可通过一个 min 函数来实现：

```
int len = (int) bitstream.length();
int cursor = 0;

// 如果没有读取到末尾
while (cursor < len) {
    // 8 和 len-cursor 中取小者，来达到要么读 8 位，要么把剩下的<8 位全部读完的效果
    int substrLen = min(8, len - cursor);
    string sub = bitstream.substr(cursor, substrLen);
    // ...
    cursor += 8;
}
```

于是我们就实现了逐 8 个字符读取 bitstream 字符串。接下来要做的是两件事情，一是将该字符串转换为其对应的 8bit 数字表示（str2byte 过程），二是将其写入到文件的正确位置。

第一步的实现有两种途径，一是根据逻辑推理，针对字符串的每一位展开分析，对一个基数（如 0x00）进行多步位运算（与、或、移位等），转化成其正确的数字对应，较为麻烦。这种方法的一个可能的实现如下：

```
char str2byte(string s) {
    char b = 0x00;
    for (int i = 0; i < 8; i++) {
        b = b << 1;
        if (s[i] == '1') {
            b = b | 0x01;
```

```
        }
    }

    return b;
}
```

第二种途径是借助 STL 中自带的 atoi、stoi 等常见 AtoB 函数来直接转换。本文使用这种方法，实现的 str2byte 函数如下：

```
unsigned char str2byte(string s) {
    while (s.size() < 8) s += "0";
    return (unsigned char) stoi(s, nullptr, 2);
}
```

需要注意的是，C++中的 std::byte 并不是基础类型，此处用 8 位无符号字符型（unsigned char）来代替，其用途是相同的。另外，为了后续解压能够正确处理哈夫曼编码，我们需要添加**末尾补零**机制，这一点将在后面的解压算法部分中详细解释。

该函数实现了将一个二进制数，如 00010001，转化为一个 0~255 范围内的整数 17。

至于写入文件这一步，可通过 fstream 所带的 write 方法来实现：

```
void writeChar(unsigned char s, ofstream &out) {
    unsigned char cc[1];
    cc[0] = s;
    out.write(reinterpret_cast<const char *>(cc), sizeof(cc));
}
```

由于 write 方法只支持 char 类型数组的写入，而我们只需要写入单个字符，所以此处我们借助一个数组来写入单个字符。此外，由于并不支持 unsigned char*，这里用到 reinterpret_cast 来进行一个无损的类型转换，从而确保后续解压过程中能够正确读出我们存储的无符号字符型。

这两部分实现后，将相关代码添加到上面的 while 循环中，我们就得到了将哈夫曼编码后的文件压缩后写入压缩文件（.buf）中的算法：

```
while (cursor < len) {
    int substrLen = min(8, len - cursor);
    string sub = bitstream.substr(cursor, substrLen);
    writeChar(str2byte(sub), out);
    compressedSize += sizeof(unsigned char); // 1
    cursor += 8;
}
```

最后的 compressSize 的值就是压缩以后的文件字节数。

B. 解压缩部分的算法设计

1. 解压缩的要求

如果一个文件压缩算法是可逆、无损的，其应当确保压缩后的文件内的信息足以原样还原出原文件的所有字节信息。A 部分压缩算法所使用的哈夫曼编码和 8 位字符串压缩算法都是可逆、无损的，因此解压缩算法的要求自然是能够 100%还原出原文件的内容。

是否还原出了原文件的内容可通过（初步）对比大小、diff 或者 cmp 指令（类 Unix 系统自带）以及计算 SHA256 checksum 来进行验证。

2. 文件头的设计与写入读取过程

为了确保解压后的文件与原文件完全相同，我们需要先确保其大小完全相同。因此有必要在压缩时将原文件的大小信息写入文件的开头，这并不会增加压缩文件多少大小，却十分重要。

我们还需要在解压缩时重新构造出压缩时出现的一模一样的哈夫曼树。我们注意到哈夫曼树构造的参数只有三个，其中需要外部提供的只有权值数组。所以我们也需要将 weights 数组写入到文件头中。

由于只有两种数据，考虑用**顺序**写入/读取的方式存储在文件最开头的方式。这样，只要写入和读取的顺序不变，就可以确保写入和读取到的数据的一致性。

在压缩过程写入 str2byte 结果之前，先写入以上提到的两种数据：

```
for (int i = 0; i < kinds; i++) {
    out << weights[i] << " "; //  写入权值数组内容
}

int len = (int) bitstream.length();

out << len << "\n"; //  写入 bitstream 的长度
```

这样，文件中的内容先是按顺序存储权值数组的每一个元素，然后是一个数字用来表示 bitstream 的长度。在这里权值数组的长度已知（即 bits=256），不需要特别记录。

在解压缩读取压缩文件内容时，按照顺序读取并吃掉最后一个换行符即可。

```
int weights[kinds];
int bitstream_length;

for (int i = 0; i < kinds; i++) {
    in >> weights[i];
}

in >> bitstream_length;

in.get(); //  吃掉换行符
```

3. 哈夫曼编码的解压缩

在上面我们得到了会用到的两个重要参数：权值数组和哈夫曼编码后的 bitstream 长度。权值数组使得我们可以直接构造出一模一样的哈夫曼树，因此在解压缩代码的开头我们就可以完成这一步骤：

```
HuffmanNode *tree[kinds * 2];
buildHuffmanTree(tree, weights, kinds);
```

其中 kinds 作为参数传入，值为 256；weights 就是我们读入的权值数组。

接下来，我们需要将文件中的字节反编码为哈夫曼编码的内容，也就是将每一个字节展开为一个 8 位二进制数（作为字符串）。将此函数取名为 byte2str。

这一点亦可以通过 STL 中的常用函数来实现：

```
string byte2str(unsigned char s) {
    return bitset<8>(s).to_string();
}
```

bitset<8>(s) 将传入的无符号字符型看作是长为 8 的位集，调用 to_string 后就得到了我们在压缩算法中编码之前的字符串内容。此函数可以将 0~255 内的字符型，如 32，转换为形如 "00010000" 的八位字符串。

逐个读取文件字节，并将其展开为八位二进制数字符串，最后拼接在一起，就还原成了压缩前的哈夫曼编码内容（下文为了简洁，称之为**全文哈夫曼编码**）。这一步实质上是实现了我们在压缩算法中，对哈夫曼编码的压缩的解压缩。

4. 哈夫曼编码的逆映射

在得到全文哈夫曼编码后，注意到这一编码的内容是每个字节的哈夫曼编码按照顺序从前向后排序构成的。也就是说，对于文件内容

```
1 3 57 43 29 57
```

其对应的哈夫曼编码为：

```
0001 10000 0101001 11011 1101 0101001
```

实际的 bitstream 则移除了其中的空隙，变为：

```
0001100000101001110111101010 1001
```

这正是我们现在得到的全文哈夫曼编码。既然有顺序，我们就可以考虑是否可以直接将其还原为其对应的字节。但是如果没有自带的空隙，应该如何区分他们之间的分割线呢？

注意到每一个哈夫曼编码的对象是**叶子节点**。很显然，上文中构造哈夫曼编码的算法是可逆的：构造算法中从叶子节点出发，依次向上抵达根节点，从而产生一个哈夫曼编码。为了逆转这个过程，我们只需从根节点开始沿着哈夫曼编码中 0 和 1 的指示依次向左、右分支移动，到达叶子节点停止，然后重新回到根节点进行下一步即可。

又因为每一个叶子节点所处的下标恰好与其对应的字节的数量表示相对应，所以在抵达叶子节点后我们可以很方便地知道该编码所对应的字节种类（也就是内容）。同时，得益于我们构造哈夫曼树时所采用的顺序存储结构，HuffmanNode 数组的最后一个节点（下标为 2n-2）**恰好为根节点**，因此获取根节点也是易如反掌的事情。

于是我们无需任何“空格”，只需判断是否到达了叶子节点就可以准确地解析出当前的字节内容。相关算法实现如下：

```
int ptr = kinds * 2 - 2;
int decompressed_size = 0;

for (char bit: bitStream) {
    ptr = bit == '0' ? tree[ptr]->lchild : tree[ptr]->rchild;

    if (tree[ptr]->lchild == -1 && tree[ptr]->rchild == -1) {
        writeChar((unsigned char) ptr, out);
        decompressed_size++;
        ptr = kinds * 2 - 2;
    }
}
```

其中 ptr = kinds * 2 - 2 就是将当前的操作对象指向根节点。最终得到的 decompressed_size 便是解压缩以后的文件大小。

5. 确保解压缩文件的正确性的手段

上述过程在逻辑上是可行的，但是还需要进行一些保障。这也正是我们要在文件头中存储额外信息的原因。除了权值数组以外，我们存储了 bitstream 的长度，因此我们可以在解压缩文件算法之前、解压缩哈夫曼编码算法（得到 bitstream）之后，将 bitstream 长度进行限制：

```
if (bitStream.size() > bitstream_length) bitStream = bitStream.substr(0, bitstream_length);
```

另外，上文中提到的在对全文哈夫曼编码进行逐 8 位压缩用到的这一函数 byte2str 中，末尾补零的机制也是为了压缩以后的字节能够正确对应到哈夫曼编码上。更多解释会在调试部分进行。

三、主要仪器设备及耗材

1．PC 机

2．C++集成环境

第二部分：实验调试与结果分析（可加页）

一、调试过程（包括调试方法描述、实验数据记录，实验现象记录，实验过程发现的问题等）

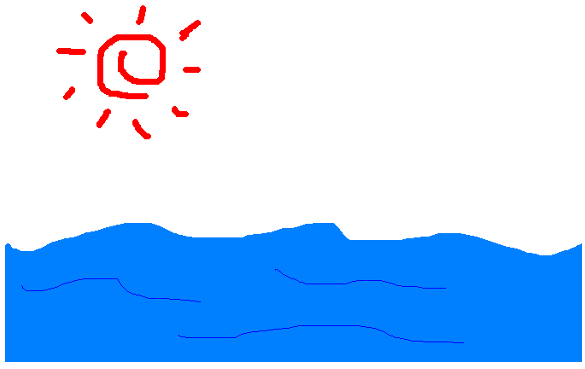
实验过程发现的问题

1. 哈夫曼树的构造

通过向哈夫曼树构造算法输入权值数组 (5,29,7,8,14,23,3,11)，可以看到能够构造出正确的哈夫曼树数组。

下标	权值	双亲节点下标	左孩子下标	右孩子下标
0	5	8	-1	-1
1	29	13	-1	-1
2	7	9	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	8	-1	-1
7	11	10	-1	-1
8	8	10	6	0
9	15	11	2	3
10	19	12	8	7
11	29	13	4	9
12	42	14	10	5
13	58	14	1	11
14	100	-1	12	13

文件在调试过程中使用的图片为 pic.bmp：



pic.bmp 是一个普通的 BMP 文件，其特点是可压缩性较高，因为其中包含了大量的纯色（相同的信息）。通过哈夫曼编码的结果就可以反映出这一点。

将文件读取并根据字节种类构造权值数组的相关算法对接后，对 pic.bmp 中的 256 种字节，按权值构造哈夫曼树，得到的部分结果如下表所示。

下标	权值	双亲节点下标	左孩子下标	右孩子下标
0	76980	509	-1	-1
1	261	493	-1	-1
2	37	420	-1	-1

		...		
309	16	364	272	273
310	16	365	274	275
311	17	371	276	8
		...		
508	75110	509	507	128
509	152090	510	508	0
510	649318	-1	509	255

注：可能因为不是原图而有数据上的出入，但压缩/解压缩过程是可行的。

可见哈夫曼编码的算法可行。

2. str2byte 与 byte2str 的一个问题的解决
与 byte2str 的后补零机制的解释

假设在压缩过程中生成的全文哈夫曼编码长度为 259=256+3。这表示在逐 8 位压缩过程中，会有多余的 3 位，且不能将这 3 位直接抛弃。stoi 函数的默认行为是将这三位数字直接转换为其对应的二进制表示，即 101 转换为 5。在这一过程中隐含了一个前补零的操作，可以相当于是将 101 看作了 8 位二进制数 00000101。

在解压缩过程中，由于手动规定了 byte2str 函数返回的字符串长度，最终得到的 bitstream 一定是 8 的倍数，称之为**带冗余的全文哈夫曼编码**。在当前假设中我们得到的长度为 256+8=264，其中最后八位正是 00000101（因为 5=00000101）。

这会导致解压缩文件进行到这一部分时，多走了前面多出的五个 0 的分支，导致解压出现错误（经过测试，这样解压出来的文件仍然可以打开，但是兼容性大打折扣）。如果我们通过 bitstream 长度的信息，用 substr 将带冗余的全文哈夫曼编码的长度限制为与压缩前的全文哈夫曼编码长度相同，更是会直接舍弃掉一些处于末尾的正确信息。

因此，正确的实现应该是在 str2byte 时进行末尾补零，即

```
while (s.size() < 8) s += "0";
```

在这个例子中，我们得到的带冗余全文哈夫曼编码的末 8 位会变成 10100000，通过 substr 截取后转换为原有的 101，从而能够正确解析出相应的字节，达成文件的无损解压。

二、实验结果及分析（包括结果描述、实验现象分析、影响因素讨论、综合分析和结论等）
通过向程序输入 pic.bmp 的文件路径，可以得到以下结果：

```
文件大小：649318B
raw bitstream size:917867
压缩文件大小：114734B
压缩率：17.6699%
output bitstream size:917872
解压缩文件大小：649318B
```

其中输出的 bitstream 大小 917872 较大，是 917867 补 0 后的 bitstream 长度：
917872=8*114734=917867+5
这恰好印证了上文中提到过的全文哈夫曼编码与带冗余的全文哈夫曼编码之间的关系。

解压缩后的文件与原文件相同性检验

对压缩和解压缩产生的两个文件 pic.bmp.buf 和 pic.bmp.dec.bmp 进行分析，它们之间的大小关系如下图和下表所示：

文件名	大小
pic.bmp	649318 字节
pic.bmp.buf	114734 字节
pic.bmp.dec.bmp	649318 字节

利用 Unix 指令 od 输出的压缩前文件和解压缩后文件的字节（按 16 进制，-h）：

od -h pic.bmp

```
0000000    4d42    e866    0009    0000    0000    0036    0000    0028
0000020    0000    0242    0000    fe8a    ffff    0001    0018    0000
0000040    0000    e544    0009    1625    0000    1625    0000    0000
0000060    0000    0000    0000    ffff    ffff    ffff    ffff    ffff
                                  ......
2360560    0080    80ff    ff00    0080    80ff    ff00    0080    80ff
2360600    ff00    0080    80ff    ff00    0080    80ff    ff00    0080
2360620    80ff    ff00    0080    80ff    ff00    ffff    0000    ffff
2360640    ffff    ffff    ffff    ffff    ffff    ffff    ffff    ffff
                                  *
                         2364140    ffff    ffff    0000
```

od -h pic.bmp.dec.bmp

```
0000000    4d42    e866    0009    0000    0000    0036    0000    0028
0000020    0000    0242    0000    fe8a    ffff    0001    0018    0000
0000040    0000    e544    0009    1625    0000    1625    0000    0000
0000060    0000    0000    0000    ffff    ffff    ffff    ffff    ffff
                                  ......
2360560    0080    80ff    ff00    0080    80ff    ff00    0080    80ff
2360600    ff00    0080    80ff    ff00    0080    80ff    ff00    0080
2360620    80ff    ff00    0080    80ff    ff00    ffff    0000    ffff
2360640    ffff    ffff    ffff    ffff    ffff    ffff    ffff    ffff
                                  *
                         2364140    ffff    ffff    0000
```

利用 Unix 指令 diff 判定一致性：

指令：diff pic.bmp pic.bmp.dec.bmp && echo $?
输出：0（表示没有问题发生）

计算文件 SHA256 checksum：

指令：shasum -a 256 pic.bmp pic.bmp.dec.bmp

输出：

3528ce9c76723b5570a5044d9c71a05a43a2d00b0b1c6f819f347529c3cc83f8 pic.bmp

3528ce9c76723b5570a5044d9c71a05a43a2d00b0b1c6f819f347529c3cc83f8 pic.bmp.dec.bmp

第三部分：实验小结、建议及体会

一、实验小结

本实验成功实现了基于 Huffman 算法的 BMP 图片压缩与解压缩功能，验证了算法的正确性与可行性。压缩后文件大小显著减小（原文件 649KB→压缩文件 114KB），解压后文件与原文件内容完全一致（通过文件大小、二进制字节比对与 SHA256 校验），算法实现无损性。

Huffman 编码对重复字节较多的文件（如 BMP 纯色图片）压缩效果显著，压缩率约为 17.7%。但对于复杂图像或已压缩格式，压缩率可能降低（经过测试，一些 JPG 风景照片压缩率均为 95%以上）。

二、改进建议

1. 优先队列优化建树：当前使用顺序数组构建 Huffman 树时间复杂度较高（$O(n^2)$），可改用优先队列（$O(n \log n)$）提升效率。
2. 位操作直接处理：将哈夫曼编码直接按位写入缓冲区，避免生成字符串再转换，减少内存占用与时间开销。

三、实验体会

1. 通过本实验深入理解了 Huffman 编码的核心原理，如权值统计、最优前缀码构建等，并掌握了如何将抽象算法转化为具体代码实现。同时认识到实际工程中文件 IO 等底层细节，与纯理论分析存在显著差异。
2. 边界条件处理：如文件末尾不足 8 位时的补零策略、位流长度的精确截断，需通过多次测试验证逻辑正确性。
3. 模块化设计：将统计、建树、编码、压缩等步骤封装为独立函数，提升代码可维护性。

本实验不仅巩固了数据结构的核心知识，复习了 C++文件处理的流程，更培养了工程实践与问题解决能力，为后续复杂系统开发积累了宝贵经验。