数据结构 ： 哈夫曼编码压缩、解压文件 实验报告

毛子恒 李臻 张梓靖

2020 年 11 月 26 日

小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211310	姓名：李臻	学号：2019211458	分工：测试 文档
班级：2019211308	姓名：张梓靖	学号：2019211379	分工：可视化 文档

目录

1 需求分析

1.1 题目描述

对于一个指定的文件，应用哈夫曼编码压缩文件，或者对本程序生成的压缩文件进行解压。

1.2 输入描述

程序从给定的二进制文件中读入数据。

1.3 输出描述

程序向给定的二进制文件中输出结果，向标准输出中输出提示。
压缩文件分为若干个部分，每个部分包含如下信息：

1. 原字符串中包含的不同字符的个数；
2. 哈夫曼树中每个节点存储的字符（非叶结点不存储字符）、结点左右孩子的序号；
3. 原字符串的长度、压缩后位串的长度；
4. 编码后的位串。

输出分为五种情况：

1. 输入合法，程序正常运行结束，此时结果存储在给定的二进制文件中，并在标准输出中打印提示信息。
2. 输入不合法，程序向标准输出打印错误信息，并且要求重新输入或异常退出。
3. 程序发生运行时错误，比如内存分配失败、指定的输入/输出文件无法打开。此时程序没有输出。

关于更多细节请参考第 5 节。

1.4 样例输入输出

由于压缩后的二进制文件难以阅读，故通过转换程序（testing/bit2string.c) 将该压缩文件转化成容易阅读的形式展现在本实验报告中，将不同数据以空格或换行符分隔，原本按每 8 位存储的位串转化成 01 字符串。

1.4.1 样例输入输出 1

压缩。
【输入】(samples/sample1.in)

```
aaababcd
```

【输出】(samples/sample1.out)
('#' 后的内容为注释)

```
4 #出现的不同字符的个数/哈夫曼树叶结点的个数
c #一个叶结点，其中存储字符c，没有左右孩子
d
b
```

```
a
1 2 #一个非叶结点，不存储字符，左右孩子分别是1、2
3 5
4 6
8 2 #原串的长度，压缩后字符串的长度（2个字节）
0001001011011100 #压缩后的位串
```

1.4.2 样例输入输出 2

压缩。

【输入】(samples/sample2.in)

```
!#$""$#$"$$"!!"!$$#"$#"$"$!$#$!"!$!#"#!""#"!!!#"##"!$$$"#"$"$"$#!$$"$#!"#$##!!"#"#!!##!#$"!""$!"!#$$
```

【输出】(samples/sample2.out)

```
4
#
!
"
$
1 2
3 4
5 6
100 25
0100111010110011101111100101100111110010110010111011011100110110011101001000011010001001010100100000100111111110001011101110110001111111011000110001100000101100010000010100000100111001101011011001001111
```

1.4.3 样例输入输出 3

压缩。

【输入】(samples/sample3.in)

```
1>8#2$'&'-).1%2/1:)/,8;2!3>(/69",#&7;>#$</%'=-=>+2#5&>!)5;(<96&;>0,7,;:/.)7$$/%5-(67933<(';==4464!12
```

【输出】(samples/sample3.out)

```
29
"
+
0
.
:
8
-
5
9
<
!
4
```

```
3
%
)
'
&
$
#
=
7
6
1
,
(
2
>
;
/
1 2
3 30
4 5
6 31
7 8
9 10
11 12
13 14
15 32
16 17
18 19
20 21
22 23
24 25
26 33
27 34
28 35
29 36
37 38
39 40
41 42
43 44
45 46
47 48
49 50
51 52
53 54
55 56
100 60
11111010000110110110010110101100011001110000101010110101110111111010100101000111111011111011010000000001100110001010010101000100000110001111001110001111000001101111001111010110010011011110100111110001010111000111000101011100010000111110010110110101111001010010010101100101101100001011110111011110111011001011001000011100000111010000011010111110000
```

```
10111010110111011101011010100010101010110101000011111011101011101010010100011110001110000110111001110010011110011101001110010111110010000
```

1.4.4 样例输入输出 4

解压。(对应样例 1)

【输入】

```
4
c
d
b
a
1 2
3 5
4 6
8 2
00010010110111100
```

【输出】

```
aaababcd
```

1.4.5 样例输入输出 5

解压。(对应样例 2)

【输入】

```
4
#
!
"
$
1 2
3 4
5 6
100 25
0100111010110011101111100101100111110010110010111011011100110110011101001000011010001001010100100000100111111110001011101110110001111111011000110001100000101100010000010100000100111001101011011001001111
```

【输出】

```
!#$""$#$"$$"!!"!$$#"$#"$"$!$#$!"!$!#"#!""#"!!!#"##"!$$$"#"$"$"$#!$$"$#!"#$##!!"#"#!!##!#$"!""$!"!#$$
```

1.5 程序功能

对于压缩过程，程序构造哈夫曼树并且对文件进行压缩，将需要的解码信息和压缩后的位串保存到压缩文件中。

对于解压过程，程序从压缩文件中读入解码信息和位串，并利用哈夫曼树进行解压。

2 概要设计

2.1 问题解决的思路

构造哈夫曼树进行压缩、解压过程。

2.2 哈夫曼编码的设计

```
// 数据对象
typedef struct node
{
    char data;
    int cnt;
    unsigned char parent, lc, rc; // 哈夫曼树双亲、孩子结点的下标
} Node;

/*
 * 操作：编码
 * 前件：inputFile和outputFile分别指向输入、输出文件
 * 后件：将编码结果输出到outputFile指向的文件中
 */
void encode(FILE * inputFile, FILE * outputFile);

/*
 * 操作：解码
 * 前件：inputFile和outputFile分别指向输入、输出文件
 * 后件：将解码结果输出到outputFile指向的文件中
 */
void decode(FILE * inputFile, FILE * outputFile);
```

2.3 主程序的流程

1. 获取输入，判断是否合法
2. 如果合法，则调用压缩/解压函数

2.4 各程序模块之间的层次关系

程序模块层次关系图如图 1。

图 1: 程序模块层次关系

3 详细设计

3.1 哈夫曼编码的实现

哈夫曼编码的设计中基本操作的伪代码算法如下：

```
void encode(FILE * inputFile, FILE * outputFile)
{
    while (true)
    {
        从inputFile读入长度最长为1024字符串
        如果读到文件结束，退出循环
        cnt数组用来记录各个字符出现的次数，将其初始化
        统计各个字符出现次数（权值）
        按照权值从大到小对cnt数组排序
        从前往后，找到字符串的字符集的大小，设为tot
        创建tot个哈夫曼树叶结点（哈夫曼树采用顺序存储，权值小的字符在前）
        创建tot-1个非叶节点：
        t1 <- 1 目前没有双亲节点的权值最小的叶节点的下标
        t2 <- tot + 1 目前没有双亲节点的权值最小的非叶节点的下标
        for (int i = tot + 1 to 2 * tot + 1)
        {
            在t1、t1+1、t2、t2+2四个结点中选取权值最小的两个结点
            创建新的结点，左孩子和右孩子为刚刚选取的节点
            更新两个结点的双亲以及相应的指针
        }
        将tot写入文件
        将哈夫曼树叶节点存储的字符依次写入outputFile
        将哈夫曼树非叶节点的左右孩子结点的下标依次写入outputFile
        将原字符串的长度写入outputFile
        for (int i = 1 to tot)
        {
            从叶结点开始，逆向求每个字符的哈夫曼编码，作为'0'/'1'字符串写入tempString
            在huffmanCode数组（编码表）中分配位置，拷贝编码
        }
        将原字符串按照编码表编码，将编码结果写入tempString字符串中
        将tempString字符串的长度补全到8的倍数
        将tempString字符串每8个字符压缩到一个字节，写入到outputString中
        将编码后的字符串长度写入outputFile
        将outputString写入outputFile
        释放huffmanCode的空间
    }
}
void decode(FILE * inputFile, FILE * outputFile)
{
    while (true)
    {
        从inputFile读入tot
        如果读到文件结束，退出循环
        读入tot个叶节点的信息，复原哈夫曼树
        读入tot-1个非叶节点的信息，复原哈夫曼树
        读入原字符串的长度和编码后字符串的长度
        读入编码后的字符串inputString
        cur <- 2 * tot - 1 指针，表示当前在哈夫曼树中的结点下标
```

```
49        for (int i = 0 to n-1)
50        {
51            for (int j = 0 to 7)
52            {
53                获取当前位的值temp
54                if (temp)
55                    cur指向当前结点的右孩子
56                else
57                    cur指向当前结点的左孩子
58                if (找到了叶节点)
59                {
60                    将解码出的字符写入到outputFile
61                    if (解码完了所有字符)
62                        break;
63                    cur恢复指向根结点
64                }
65            }
66        }
67    }
68 }
```

3.2 函数的调用关系图

函数调用关系图如图 2。

图 2: 函数调用关系图

4 调试分析报告

4.1 调试过程中遇到的问题和思考

由于对二进制文件的输入、输出不熟悉，曾尝试使用 printf 和 scanf 函数进行输入输出，了解相关知识后改为调用 fread 和 fwrite 函数。

在解码的过程中 feof 函数经常出现探测文件末尾不准确，故改为判断 fread 函数是否为 0 来结束循环。

在构造哈夫曼树的过程中由于判断条件过于复杂，出现错误；改为了更简洁的判断条件。

在写入到压缩文件之前，压缩码需要以'0'/'1' 字符串的形式存储，否则调用各种字符串函数会出现错误。

编码以 8 位压缩到一个字节当中，写入和读取的时候注意高低位的顺序。

4.2 设计实现的回顾讨论

编码是每次从二进制文件中读入 1024 个字节，创建编码表并且压缩。

由于哈夫曼树结点数较少，所以存储下标时采用 unsigned char 类型，写入压缩文件时可以节省空间。

对于没有出现过的字符显然不需要编码。调用库函数 qsort 对权值进行排序。

哈夫曼树采取顺序存储结构，前 n 个结点为叶节点，之后 $n-1$ 个结点为非叶节点，每次构造新的非叶节点时只需要比较叶节点中权值最小的两个结点和非叶节点中权值最小的两个结点，从中挑出权值最小的两个结点即可，使得查找结点的时间复杂度降低到 $O(1)$

压缩后的串为一个 01 串，将其每 8 位压缩到一个字节并写入压缩文件。

对于只有一种字符的情况，由于无法建立哈夫曼树，所以压缩文件中只存储 tot、该字符、原字符串的长度。编码和解码是均对这种情况特殊处理。

4.3 算法复杂度分析

encode,decode 函数的复杂度为 $O(len)$，其中 len 为字符串长度。

主函数的时间复杂度为 $O(1)$，整体时间复杂度为 $O(len)$。

4.4 改进设想的经验和体会

4.4.1 改进 1

可以通过进一步压缩位（比如去掉一定为 0 的某些位）来使得压缩率更高，但是效果不明显。

5 用户使用说明

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main.c huffman.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后将数据写入到指定文件中，可以通过标准输入选择选项，也可以通过执行可执行文件时传递参数来选择选项：

1. 如果执行可执行文件时没有传递参数，则可以根据标准输入的提示选择压缩/解压功能，以及指定输入和输出文件的位置
2. 执行可执行文件时传递的参数：第一个参数为 1 或 2，表示选择压缩/解压，之后两个参数分别为输入文件的路径和输出文件的路径，例如，执行

   ```
   ./main 1 ./input.in ./output.out
   ```

   表示执行压缩功能，输入文件为本目录下的 input.in，输出文件为本目录下的 output.out

通过标准输出打印提示信息。如果输入合法并且程序正常运行结束，主函数返回值为 0。

6 测试结果

测试环节分为三个步骤。

6.1 测试第一部分

对 1.4 节给出的样例进行测试。

6.2 测试第二部分

测试边界条件。

【输入】(samples/sample4.in)

(空文件)

【输出】(samples/sample4.out)

(空文件)

【输入】(samples/sample5.in)

文件内容为 1000000 个字符 a

【输出】(samples/sample5.out)

压缩结果为一个 8873 字节的二进制文件。

【输入】

上一个例子的输出。

【输出】(samples/sample6.out)

解压与 sample5.in 相同。

6.3 测试第三部分

测试在 macOS Big Sur 11.0.1 下进行。

在 $len <= 10$，$len <= 100$，$n <= 1000000$ 的范围下分别随机生成 1000 组测试数据，进行压缩和解压操作，并且比对解压结果和原始输入是否相同。

3000 组数据中没有出现错误。

数据生成程序 (testing/data.cpp) 如下：

```cpp
#include <bits/stdc++.h>
using namespace std;

int main(int argc, char *argv[])
{
    srand(time(0));
    FILE *file = fopen("./raw.in", "wb");
    int n = atoi(argv[1]), m = atoi(argv[2]);
    for (int i = 1; i <= n; ++i)
        fprintf(file, "%c", rand() % m + 32);
    fclose(file);
    return 0;
}
```

传入两个参数，第一个参数是字符串的长度，第二个参数是字符集的大小。

比对脚本 (testing/chk.sh) 如下：

```sh
for i in {1..100}
do
    sleep 1
    ./data 100000 5
    ./main 1 raw.in encode.out
    ./main 2 encode.out decode.out
    if ! diff raw.in decode.out
```

```
    then
        break
    fi
    echo "Correct"
done
```

7 可视化

随机生成若干组数据，由三元组表实现和二维数组实现分别计算，比对运行时间，并且使用 JavaScript 将结果可视化。

比对脚本 (testing/timecount.py) 如下：

```
import os
import json
import time
a = []
b = []
for i in range(100):
    os.system("./data >in.in")
    starttime = time.time()
    os.system("./main <in.in >out.out")
    endtime = time.time()
    a.append(endtime-starttime)
    starttime = time.time()
    os.system("./test <in.in >out1.out")
    endtime = time.time()
    b.append(endtime-starttime)
    print(i)

json_str = json.dumps([a, b])
with open("80%result.json", mode="w") as file:
    file.write(json_str)
```

数据规模如下：

$n = 1000$，20% 的元素是 0、50% 的元素是 0、80% 的元素是 0 的数据各 100 组，共 300 组数据。

7.0.1 实现细节

利用已有的时间数据，使用 Highcharts 库绘制二维柱状图，比较两种算法的运行时间差距。

7.0.2 用户使用说明

使用现代浏览器打开 Chart/index.html，即可看到柱状图，将鼠标指针放到某个柱上可以看到对应的值。

7.0.3 示例

见图 3。

图 3: 可视化示例

7.0.4 结论

可以看出在规模较大的数据中，相比二维数组的实现，三元组表的实现在绝大多数情况下都有明显的性能优势，并且当矩阵越稀疏时性能优势越明显。