

河南工业大学

课程设计

用哈夫曼算法压缩文件

File Compression with Huffman Algorithm

课程设计名称: 数据结构课程设计

专 业 班 级: 计算机 1303 班

学 生 姓 名: 田劲锋

学 号: 201316920311

指 导 教 师: 白 浩

课程设计时间: 2014 年 5 月 12 日 - 2014 年 5 月 18 日

计算机专业课程设计任务书

学生姓名	田劲锋	专业班级	计算机 1303 班	学 号	201316920311
题 目	用哈夫曼算法压缩文件				
课题性质	B. 工程技术研究		课题来源	D. 自拟课题	
指导教师	白浩		同组姓名	无	
主要内容	<p>探究 Huffman 编码算法；</p> <p>利用 Huffman 算法来压缩和解压文件。</p>				
任务要求	<p>明确算法的实现步骤，表现为伪代码和图形；</p> <p>使用高级编程语言 C 实现算法；</p> <p>测试算法的正确性，测试程序的健壮性。</p>				
参考文献	<p>算法导论（第三版）. Thomas H.Cormen 等. 机械工业出版社: 2012</p> <p>算法精解（C语言描述）. Kyle Loudon. 机械工业出版社: 2012</p> <p>Introduction to Data Compression. Guy E. Blelloch. 2013</p> <p>Data Compression. Debra A. Lelewer, Daniel S. Hirschberg. 1987</p> <p>LaTeX 2_ε科技排版指南. 邓建松, 彭冉冉, 陈长松. 科学出版社: 2001</p>				
审查意见	<p>指导教师签字:</p> <p>教研室主任签字: _____ 年 月 日</p>				

说明：本表由指导教师填写，由教研室主任审核后下达给选题学生，装订在设计（论文）首页

目录

课程设计任务书	1
1 需求分析	3
1.1 前缀码	3
1.2 构造 Huffman 编码	4
1.3 编码和解码	5
2 概要设计	6
2.1 程序运行逻辑	6
2.2 接口定义	7
2.3 位操作	7
2.4 符号表	8
2.5 Huffman 树	9
3 运行环境	11
4 开发环境	11
5 详细设计	11
5.1 声明文件 <code>huffman.h</code>	11
5.2 算法实现 <code>huffman.c</code>	12
5.3 主体逻辑 <code>huff.c</code>	20
6 调试分析	22
7 测试结果	23
参考文献	25
心得体会	26
成绩评价表	27

1 需求分析

一个压缩解压实用程序，要读取输入文件，压缩之后写出输出文件，并可以把压缩文件解压还原成原来的文件。由此我们要实现这两个功能：

- 对文件进行编码
- 对文件进行解码

1.1 前缀码

假定我们希望压缩一个十万字符的文件。表 1 给出了文件中所出现的字符和它们的出现频率。

	a	b	c	d	e	f
频率（千次）	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
变长编码	0	101	100	111	1101	1100

表 1: 一个字符编码问题。一个 100 000 个字符的文件，只包含 a-f 6 个不同字符，频率如此。如果为每个字符指定一个 3 位的码字，则文件编码长度为 300 000 位。但如果使用变长编码，就可以下降到 224 000 位。

本来，如果采用定长编码，这个文件需要 300 000 个二进制位(bit) 来表示。换用表 1 给出的变长编码之后，则只需要

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224000 \text{ bit}$$

就能表示该文件，相比节约了 20% 的空间。实际上，可以证明，这就是该文件的最优字符编码。

这里我们只考虑前缀码，即没有任何字符是其他码字的前缀。任何二进制字符码的编码过程都很简单，只要将每个字符的码字连接起来即可完成文件编码。前缀码的作用是简化解码过程。由于没有码字是其他码字的前缀，编码文件的开始码字是无歧义的。可以简单地识别出开始码字，将其转换回原字符，然后对剩余部分重复此过程即可。

1.2 构造 Huffman 编码

哈夫曼编码 (Huffman coding), 又译霍夫曼编码、哈弗曼编码、赫夫曼编码, 是一种古老的压缩算法, 它是一种基于最小冗余编码的压缩算法 [1]。最小冗余编码是指, 如果知道一组数据中符号出现的频率, 就可以用相应的位数来表示符号从而减少数据需要的空间。David A. Huffman [2] 设计这样一个贪心算法来构造最优前缀码。

下面给出该算法的伪代码。假定 C 是一个 n 个字符的集合, 其中每个字符 $c \in C$ 都是一个对象, 其属性 $c.freq$ 表示字符的出现频率, 事先已经计算好。算法自底向上构造出对应最优编码的二叉树 T 。它从 n 个叶节点开始, 执行 $n - 1$ 次合并操作, 创建出最终的二叉树。 Q 是一个有序存储结构, 每次取出其中 $freq$ 属性最小的两个对象进行合并。合并之时, 得到的新对象的频率则设为原来两个对象频率之和。

```
HUFFMAN( $C$ )
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      为  $z$  分配存储空间
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$ 
```

算法运行速度取决于 Q 的实现。一般地情况下, 使用 Fibonacci 堆可以得到

$$O(1) + O(n - 1) \cdot (2O(\lg n) + O(1)) = O(n \lg n)$$

的时间复杂度, 但相应的实现比较复杂, 可以成为另一个课题讨论¹。这里简化问题, 使用一个数组存储, 每次插入之后进行一次增序排序 (使用快速排序), 每次取出最前面两个元素即可, 而时间复杂度就退化成了

$$O(1) + O(n - 1) \cdot (2O(1) + O(n \lg n)) = O(n^2 \lg n)$$

¹实际上, 使用 van Emde Boas 树还可以进一步优化成 $O(n \lg \lg n)$, 这里不再讨论。

实际上，由于 n 是字符集数量，这个值一般是个常数，为 256。在实际运行中，这样小的规模中， $O(n \lg n)$ 的优势并不比 $O(n^2 \lg n)$ 好多少。这并不是一个严重的问题，相对于庞大的数据量来说，符号数造成的影响小到可以忽略。

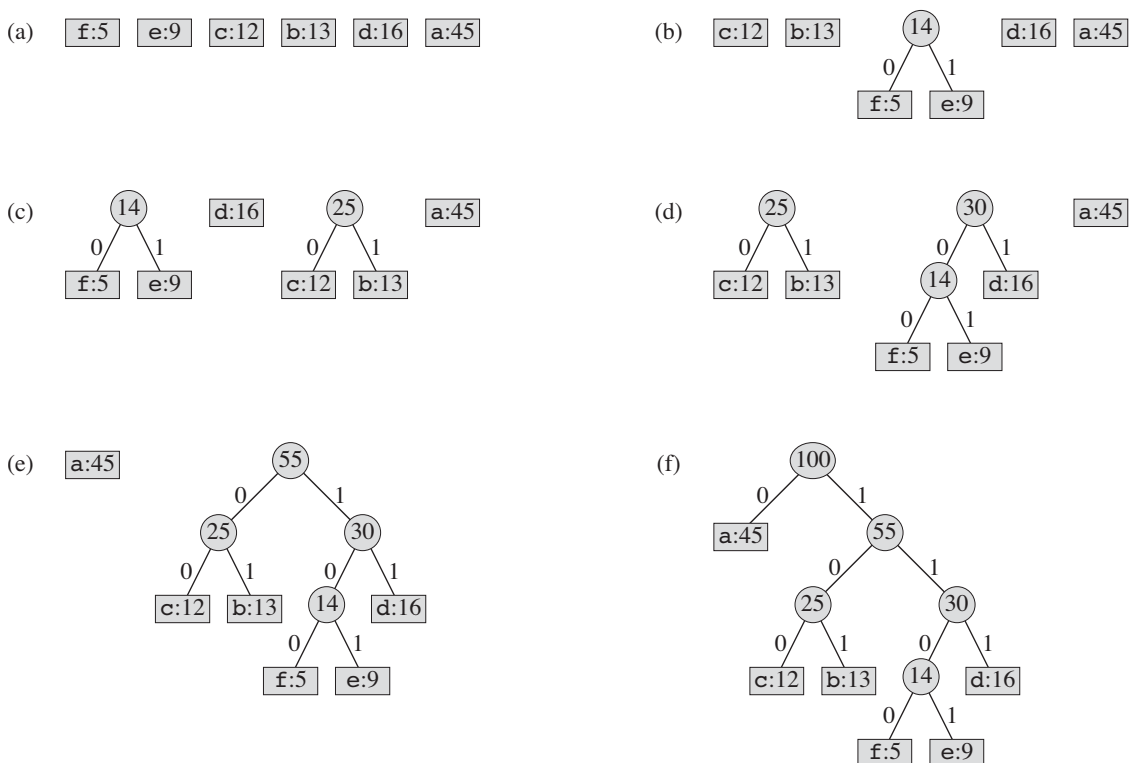


图 1: 对表 1 给出的频率执行 Huffman 算法的过程。每一部分显示了 Q 中的内容，且已有递增序。每个步骤中，频率最低的两棵树进行合并。叶结点用矩形表示，包含一个字符及其频率。内部结点用圆圈表示，包含其孩子结点频率之和。结点指向左孩子的边标记为 0，指向右孩子的边标记为 1。一个字母的码字对应从根到其叶结点的路径上的边标记序列。(a) 初始集合有 $n = 6$ 个结点，每个结点对应一个字母。(b)–(e) 为中间步骤。(f) 是最终的编码树。(参考 [3])

对前文给出的例子，Huffman 算法的执行过程如图 1 所示。字母表包含 6 个字母所以初始 $n = 6$ ，需要 5 个合并步骤构造二叉树。最终的二叉树表示最优前缀码。每个字母的码字为根节点到该字母叶结点的简单路径上边标签的序列。

1.3 编码和解码

由于所有文件都是二进制存放的，所以我这里不讨论文件编码²问题，将中文字符和西文字符都看做字节，减少问题的复杂度。由于我们不安排编码，所以压

²通常的文件编码最小单位都是字节 (byte)，包括 ASCII 和 GB 定长编码和 UTF-8 变长编码，这些都是用作通常存储的非压缩编码。

缩解压后文件编码如原文一致。这样理论上也可以压缩图片、音乐甚至视频等非文本文件。所以我们以二进制方式读入写出文件。

编码时，读入文件，从头到尾扫描一遍，统计每个字符（符号）的出现次数（频率）。然后，依据该频率统计结果，构造一棵 Huffman 树，并依此树生成一个编码表。写出该编码表。再次扫描文件，根据 Huffman 树编码每个字符，并写出到文件。

解码时，读入文件开始部分的编码表，依此构建 Huffman 树。接着读入剩余字符，根据 Huffman 树解码每个字符，并写出到文件。

2 概要设计

2.1 程序运行逻辑

我们首先考虑程序运行的主要逻辑。当键入程序名运行程序时，应该给出帮助信息，告诉用户应该怎样使用本程序。假定运行程序名为 `huff`。

考虑到要压缩文件并输出，至少我们需要两个参数即<输入文件>和<输出文件>。所以，要压缩一个文件1到文件2，键入命令 `huff 文件1 文件2` 即可。

对于解压文件，可以直接加一个参数 `-u`。比如把文件2解压成文件3，应当键入命令 `huff 文件2 文件3 -u`。

我们还可以提供如下选项：

- `-i` 输入文件
- `-o` 输出文件
- `-u` 解压缩
- `-z` 压缩（默认）
- `-h` 显示本帮助
- `-v` 显示版本信息

使用 `getopt()` 函数即可处理这些参数。

默认情况下，对于正常完成的操作，不应该输出提示信息，直接结束即可。对于异常情况，则输出错误信息并结束程序，返回错误码。

2.2 接口定义

huffman_encode_file

```
int huffman_encode_file(FILE *in, FILE *out);
```

返回 成功返回 0

描述 读入文件 *in*，编码之后输出到 *out* 中。首先获取输入文件中符号出现频率，依此调用 Huffman 算法生成编码表。再次扫描文件，使用之前生成的编码表来编码，每读入一个位就进行对照编码表进行编码，满一个字节就写出，最后不满补 0。

huffman_decode_file

```
int huffman_decode_file(FILE *in, FILE *out);
```

返回 成功返回 0

描述 读入文件 *in*，解码之后输出到 *out* 中。首先读入编码表，构建对应的 Huffman 树。接着按位读入后面的数据，不断查询 Huffman 树知道叶结点，输出解码结果，不断重复到文件直到末尾。

2.3 位操作

code
len: unsigned long
bits: unsigned char *

定义数据类型 *code* 用来存储位数据，其 *len* 表示编码位长，*bits* 是指向存储编码值得指针，一个 *bits*[] 表示八位编码。

bit_len_byte

```
unsigned long bit_len_byte(unsigned long len);
```

返回 字节长度

描述 把位的长度转换成字节的长度，不足 8 位则进位。

get_bit

```
unsigned char get_bit(unsigned char *bits, unsigned long i);
```

返回 0 或 1

描述 获取 *bits* 的第 *i* 位二进制位。

reversc_bits

```
void reversc_bits(unsigned char *bits, unsigned long len);
```


返回 无

描述 把长度为 *len* 的 *bits* 依二进制位反转。

new_code

```
code * new_code(const node *leaf);
```

返回 指向生成的编码结构体的指针

描述 从叶结点 *leaf* 沿 Huffman 树回溯到根结点，逆序生成该叶结点对应符号的 Huffman 编码序列。

free_code

```
void free_code(code *p);
```

返回 无

描述 销毁编码 *p*。

2.4 符号表

定义符号表类型：

- 符号频率表 `symfreq[]` 是一个长为 256 的结点数组类型，主要用来存储符号出现频率以构建 Huffman 树；
- 符号编码表 `symcode[]` 是一个长为 256 的编码数组类型，用来存储符号对应的编码。

init_freq

```
void init_freq(symfreq *sf);
```

返回 无

描述 初始化符号频率表 *sf*。

get_symfreq

```
unsigned int get_symfreq(symfreq *sf, FILE *in);
```

返回 文件 *in* 中的符号总数

描述 读入文件 *in* 中所有符号，统计符号总数，并把每个符号出现次数保存到符号频率表 *sf* 中。

write_code_table

```
int write_code_table(FILE *out, symcode *sc, uint32_t syms);
```

返回 成功返回 0

描述 向文件 *out* 中写出符号编码表 *sc*。写出的具体格式如下：

1. (4 字节) 编码大小 n , 即 sc 的大小, 以网络字节顺序写出
2. (4 字节) 待编码字节数 $syms$, 即文件大小, 以网络字节顺序写出
3. 符号的具体编码 $code[1..n]$, 每个 $code[i]$ 的编码格式为:
 - (a) (1 字节) 符号, 即原字符本身
 - (b) (1 字节) 位长, 即下面的编码位长度, 单位是位
 - (c) 具体编码位数据。如果编码不是 8 的倍数的话, 最后一字节会有用以补 0 的多余位

read_code_table

```
node * read_code_table(FILE *in, unsigned int *pdb);
```

返回 从文件中构建的 Huffman 树

描述 从文件 `in` 中读入符号编码表。以网络字节顺序, 首先读入符号总数 $count$, 然后读入待解码的符号总数 pdb 。接着依据规则读入符号编码表, 依此构建 Huffman 树, 并把文件偏移设置到符号编码表末尾。

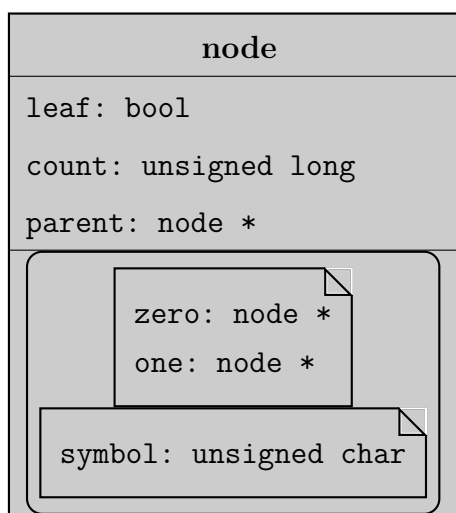
free_encoder

```
void free_encoder(symcode *sc);
```

返回 无

描述 销毁符号编码表 sc 。

2.5 Huffman 树



定义数据类型 `node` 用来存储 Huffman 树的结点。其 `leaf` 标识是否为叶结点, 如果是叶结点, 联合体用 `symbol` 存储该结点表示的符号; 如果不是, 则使用含有 `zero` 和 `one` 指针域的结构体来存储左右子树的地址。`count` 存储该结点的频率 (频数), 指针 `parent` 指向该结点的父亲。

new_leaf_node

`node * new_leaf_node(unsigned char symbol);`

返回 生成的结点地址

描述 生成一个叶结点，其保存了符号 *symbol*。

new_node

`node * new_node(unsigned long count, node *zero, node *one);`

返回 生成的结点地址

描述 生成一个普通结点，其保存了频数 *count*，左右孩子分别是 *zero* 和 *one*。

sfcmp

`int sfcmp(const void *p1, const void *p2);`

返回 正数表示 $p1 > p2$ ，零表示相等，负数表示 $p2 < p1$

描述 对符号频率表进行排序的比较函数，用于 `qsort()`。规则是低频率在前，高频率在后，频率为 0 即频率为空的在末尾。

calc_code

`symcode * calc_code(symfreq *sf);`

返回 已编码的数组指针

描述 从符号频率表 *sf* 建立 Huffman 树的主算法。首先获取符号数 *n*，然后对频率进行增序排序。接着构建 Huffman 树，每次取出并删除 *sf* 最前面两个元素，合并成新的结点并加入 *sf*，调用 `qsort()` 进行增序排序以维护有序性，如此进行 $n - 1$ 次即可构建成功。最后调用 `build_symcode()` 生成符号编码表并返回。

build_symcode

`void build_symcode(node *root, symcode *sc);`

返回 无

描述 以 *root* 为根遍历以构建好的 Huffman 树，建立符号编码表 *sc*。

free_tree

`void free_tree(node *root);`

返回 无

描述 从根结点 *root* 开始递归销毁这棵 Huffman 树。

3 运行环境

理论上，该程序没有硬件和软件限制。只要是现代计算机，有为该平台移植的 GNU C 编译器和网络库即可。为了显示运行结果，可能还需要一个终端实用程序。

对于 Linux 系统，安装了 GNU C 编译器即可，并包含网络库 `<netinet/in.h>`。编译时，进入程序目录，输入 `make` 即可生成 `huff` 二进制可执行文件。

对于其他类 UNIX 系统，我没有进行测试，但只要支持 POSIX 标准，理论上是可以编译运行的。

对于 Windows 系统，安装了 MinGW 编译器即可编译该程序，要求编译器支持 ANSI C，并包含网络库 `<winsock2.h>`。编译时，进入程序目录，输入 `make WIN32=TRUE` 即可生成 `huff.exe` 二进制可执行文件。注意由于源文件本身是 UTF-8 编码的，所以编译之后运行会产生乱码。这时可以使用文本编辑器打开 `huff.c` 文件，保存成 GBK 编码格式，然后再进行编译。由于本文重点不在文字编码，所以不再进行详细讨论。

4 开发环境

为跨平台考虑，我在两个平台上进行开发和调试，见表 2。

	主作业平台	副作业平台
操作系统	Microsoft Windows 8.1	Arch Linux (内核 3.13.7-1)
系统类型	x64	x86_64 (虚拟机)
编辑器	Sublime Text 2	Vim 7.4
编译器	MinGW32 GCC 4.5.2	GCC 4.8.2

表 2: 我的开发环境

5 详细设计

5.1 声明文件 `huffman.h`

```
1 /*
2  * 文件 : huffman.h
3  * 描述 : Huffman 压缩接口定义
4  */
5
```

```

6  #ifndef HUFFMAN_H
7  #define HUFFMAN_H
8
9  #include <stdio.h>
10 #include <stdint.h>
11
12 typedef enum {false, true} bool;
13
14 int huffman_encode_file(FILE *in, FILE *out);
15 int huffman_decode_file(FILE *in, FILE *out);
16
17 #endif

```

5.2 算法实现 huffman.c

```

1  /*
2   * 文件 : huffman.c
3   * 描述 : Huffman 压缩算法实现
4   */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <assert.h>
10
11 #include "huffman.h"
12
13 #ifdef WIN32
14 #include <winsock2.h>
15 #include <malloc.h>
16 #else
17 #include <netinet/in.h>
18 #endif
19
20 typedef struct node_ {
21     bool leaf;
22     unsigned long count;
23     struct node_ *parent;
24     union {
25         struct {
26             struct node_ *zero, *one;
27         };
28         unsigned char symbol;
29     };
30 } node;
31
32 typedef struct code_ {
33     /* 编码位长 */
34     unsigned long len;
35     /* 一个 bits[] 表示八位编码 */
36     unsigned char *bits;
37 } code;
38

```

```

39  /* 位长度到转成字节长度 */
40  static unsigned long bit_len_byte(unsigned long len)
41  {
42      return len / 8 + (len % 8 ? 1 : 0);
43  }
44
45  /* 返回第 i 位 */
46  static unsigned char get_bit(unsigned char *bits, unsigned long i)
47  {
48      return (bits[i / 8] >> i % 8) & 1;
49  }
50
51  /* 反转各位 */
52  static void reversc_bits(unsigned char *bits, unsigned long len)
53  {
54      unsigned long lbytes = bit_len_byte(len);
55      unsigned char *tmp = (unsigned char *) alloca(lbytes);
56      unsigned long c_bit;
57      long c_byte = 0;
58      memset(tmp, 0, lbytes);
59      for (c_bit = 0; c_bit < len; ++c_bit) {
60          unsigned int bitpos = c_bit % 8;
61
62          if (c_bit > 0 && c_bit % 8 == 0)
63              ++c_byte;
64
65          tmp[c_byte] |= (get_bit(bits, len - c_bit - 1) << bitpos);
66      }
67      memcpy(bits, tmp, lbytes);
68  }
69
70  /* 对叶节点生成编码 */
71  static code *new_code(const node *leaf)
72  {
73      /* 向上移动到根节点, 反转各位 */
74      unsigned long len = 0;
75      unsigned char *bits = NULL;
76      code *p;
77      while (leaf && leaf->parent) {
78          node *parent = leaf->parent;
79          unsigned char c_bit = (unsigned char) (len % 8);
80          unsigned long c_byte = len / 8;
81          /* 分配一个新字节 */
82          if (c_bit == 0) {
83              size_t new_size = c_byte + 1;
84              bits = (unsigned char *) realloc(bits, new_size);
85              bits[new_size - 1] = 0; /* 初始化新字节 */
86          }
87          /* 只用把 1 加入, 因为默认是 0 */
88          if (leaf == parent->one)
89              bits[c_byte] |= 1 << c_bit;
90          ++len;
91          leaf = parent;
92      }
93      if (bits)

```

```

94     reversc_bits(bits, len);
95     p = (code *) malloc(sizeof(code));
96     p->len = len;
97     p->bits = bits;
98     return p;
99 }
100
101 #define MAX_SYMBOLS 256
102 /* 符号频率表 */
103 typedef node *symfreq[MAX_SYMBOLS];
104 /* 符号编码表 */
105 typedef code *symcode[MAX_SYMBOLS];
106
107 static node *new_leaf_node(unsigned char symbol)
108 {
109     node *p = (node *) malloc(sizeof(node));
110     p->leaf = true;
111     p->symbol = symbol;
112     p->count = 0;
113     p->parent = 0;
114     return p;
115 }
116
117 static node *new_node(unsigned long count, node *zero, node *one)
118 {
119     node *p = (node *) malloc(sizeof(node));
120     p->leaf = false;
121     p->count = count;
122     p->zero = zero;
123     p->one = one;
124     p->parent = 0;
125     return p;
126 }
127
128 static void free_tree(node *root)
129 {
130     if (root == NULL)
131         return;
132     if (!root->leaf) {
133         free_tree(root->zero);
134         free_tree(root->one);
135     }
136     free(root);
137 }
138
139 static void free_code(code *p)
140 {
141     free(p->bits);
142     free(p);
143 }
144
145 static void free_encoder(symcode *sc)
146 {
147     unsigned long i;
148     for (i = 0; i < MAX_SYMBOLS; ++i) {

```

```

149     code *p = (*sc)[i];
150     if (p)
151         free_code(p);
152 }
153 free(sc);
154 }
155
156 static void init_freq(symfreq *sf)
157 {
158     memset(*sf, 0, sizeof(symfreq));
159 }
160
161 static unsigned int get_symfreq(symfreq *sf, FILE *in)
162 {
163     int c;
164     unsigned int tot = 0;
165     /* 初始化频率表 */
166     init_freq(sf);
167     /* 计算文件中每个符号的出现次数 */
168     while ((c = fgetc(in)) != EOF) {
169         unsigned char uc = c;
170         if (!(*sf)[uc])
171             (*sf)[uc] = new_leaf_node(uc);
172         ++(*sf)[uc]->count;
173         ++tot;
174     }
175     return tot;
176 }
177
178 /* 排序比较函数，低频率在前，空在末尾 */
179 static int sfcmp(const void *p1, const void *p2)
180 {
181     const node *hn1 = *(const node **) p1;
182     const node *hn2 = *(const node **) p2;
183     if (hn1 == NULL && hn2 == NULL)
184         return 0;
185     if (hn1 == NULL)
186         return 1;
187     if (hn2 == NULL)
188         return -1;
189     if (hn1->count > hn2->count)
190         return 1;
191     else if (hn1->count < hn2->count)
192         return -1;
193     return 0;
194 }
195
196 /* 调试用：输出频率表 */
197 static void print_freq(symfreq *sf)
198 {
199     size_t i;
200     for (i = 0; i < MAX_SYMBOLS; ++i) {
201         if ((*sf)[i])
202             printf("%d, %ld\n", (*sf)[i]->symbol, (*sf)[i]->count);
203         else

```



```

204     printf("NULL\n");
205 }
206 }
207
208 /* 遍历子树建立符号编码表 */
209 static void build_symcode(node *root, symcode *sc)
210 {
211     if (root == NULL)
212         return;
213     if (root->leaf)
214         (*sc)[root->symbol] = new_code(root);
215     else {
216         build_symcode(root->zero, sc);
217         build_symcode(root->one, sc);
218     }
219 }
220
221 /* 返回编码数组 */
222 static symcode *calc_code(symfreq *sf)
223 {
224     unsigned int i = 0;
225     unsigned int n = 0;
226     node *m1 = NULL, *m2 = NULL;
227     symcode *sc = NULL;
228     /* 对频率进行增序排序 */
229     qsort((*sf), MAX_SYMBOLS, sizeof((*sf)[0]), sfcmp);
230     /* 获取符号数 */
231     for (n = 0; n < MAX_SYMBOLS && (*sf)[n]; ++n);
232     /* 构造 Huffman 树 */
233     for (i = 0; i < n - 1; ++i) {
234         /* 令 m1 和 m2 是最小的两棵树 */
235         m1 = (*sf)[0];
236         m2 = (*sf)[1];
237         /* 合并成新树 {m1, m2} */
238         (*sf)[0] = m1->parent = m2->parent =
239             new_node(m1->count + m2->count, m1, m2);
240         (*sf)[1] = NULL;
241         /* 对符号频率表排序 (伪优先队列) */
242         qsort((*sf), n, sizeof((*sf)[0]), sfcmp);
243     }
244     /* 从树建立编码表 */
245     sc = (symcode *) malloc(sizeof(symcode));
246     memset(sc, 0, sizeof(symcode));
247     build_symcode((*sf)[0], sc);
248     return sc;
249 }
250
251 /* 写出编码表。格式如下:
252  * * 4 字节编码大小 n
253  * * 4 字节已编码字节数
254  * * 编码 [1..n] , 每个编码 [i] 的格式为:
255  * * 1 字节符号, 1 字节位长, 编码字节
256  * * 如果编码不是 8 的倍数的话, 最后一字节可能会有多余位
257  */
258 static int write_code_table(FILE *out, symcode *sc, uint32_t syms)

```

```

259 {
260     uint32_t i, count = 0;
261     /* 确定 sc 中的符号数 */
262     for (i = 0; i < MAX_SYMBOLS; ++i) {
263         if ((*sc)[i])
264             ++count;
265     }
266     /* 写出符号数 */
267     i = htonl(count);
268     if (fwrite(&i, sizeof(i), 1, out) != 1)
269         return 1;
270     /* 写出待编码字节数 */
271     syms = htonl(syms);
272     if (fwrite(&syms, sizeof(syms), 1, out) != 1)
273         return 1;
274     /* 写出符号表 */
275     for (i = 0; i < MAX_SYMBOLS; ++i) {
276         code *p = (*sc)[i];
277         if (p) {
278             unsigned int lbytes;
279             /* 写出 1 字节符号 */
280             fputc((unsigned char) i, out);
281             /* 写出 1 字节位长 */
282             fputc(p->len, out);
283             /* 写出编码字节 */
284             lbytes = bit_len_byte(p->len);
285             if (fwrite(p->bits, 1, lbytes, out) != lbytes)
286                 return 1;
287         }
288     }
289     return 0;
290 }
291
292 /* 读入编码表, 返回构建的 Huffman 树 */
293 static node *read_code_table(FILE *in, unsigned int *pdb)
294 {
295     node *root = new_node(0, NULL, NULL);
296     uint32_t count;
297     /* 读入符号数 */
298     if (fread(&count, sizeof(count), 1, in) != 1) {
299         free_tree(root);
300         return NULL;
301     }
302     count = ntohl(count);
303     /* 读入当前分支编码数 */
304     if (fread(pdb, sizeof(*pdb), 1, in) != 1) {
305         free_tree(root);
306         return NULL;
307     }
308     *pdb = ntohl(*pdb);
309     /* 读入符号表 */
310     while (count-- > 0) {
311         int c;
312         unsigned int c_bit;
313         unsigned char symbol;

```

```

314     unsigned char len;
315     unsigned char lbytes;
316     unsigned char *bytes;
317     node *p = root;
318     if ((c = fgetc(in)) == EOF) {
319         free_tree(root);
320         return NULL;
321     }
322     symbol = (unsigned char) c;
323     if ((c = fgetc(in)) == EOF) {
324         free_tree(root);
325         return NULL;
326     }
327     len = (unsigned char) c;
328     lbytes = (unsigned char) bit_len_byte(len);
329     bytes = (unsigned char *) malloc(lbytes);
330     if (fread(bytes, 1, lbytes, in) != lbytes) {
331         free(bytes);
332         free_tree(root);
333         return NULL;
334     }
335     /* 依据当前位加入 Huffman 树 */
336     for (c_bit = 0; c_bit < len; ++c_bit) {
337         if (get_bit(bytes, c_bit)) {
338             if (p->one == NULL) {
339                 p->one = c_bit == (unsigned char) (len - 1)
340                     ? new_leaf_node(symbol)
341                     : new_node(0, NULL, NULL);
342                 p->one->parent = p;
343             }
344             p = p->one;
345         } else {
346             if (p->zero == NULL) {
347                 p->zero = c_bit == (unsigned char) (len - 1)
348                     ? new_leaf_node(symbol)
349                     : new_node(0, NULL, NULL);
350                 p->zero->parent = p;
351             }
352             p = p->zero;
353         }
354     }
355     free(bytes);
356 }
357 return root;
358 }
359
360 /* 对文件编码 */
361 static int do_file_encode(FILE *in, FILE *out, symcode *sc)
362 {
363     unsigned char c_byte = 0;
364     unsigned char c_bit = 0;
365     int c;
366     while ((c = fgetc(in)) != EOF) {
367         unsigned char uc = (unsigned char) c;
368         code *cod = (*sc)[uc];

```

```

369     unsigned long i;
370     for (i = 0; i < cod->len; ++i) {
371         /* 添加位到当前字节 */
372         c_byte |= get_bit(cod->bits, i) << c_bit;
373         /* 若字节已满则写出并重置 */
374         if (++c_bit == 8) {
375             fputc(c_byte, out);
376             c_byte = 0;
377             c_bit = 0;
378         }
379     }
380 }
381 /* 还有剩余数据的话直接输出 */
382 if (c_bit > 0)
383     fputc(c_byte, out);
384 return 0;
385 }
386
387 /* 把 in 压缩成 out */
388 int huffman_encode_file(FILE *in, FILE *out)
389 {
390     symfreq sf;
391     symcode *sc;
392     node *root = NULL;
393     int rc;
394     unsigned int syms;
395     /* 获取输入文件中符号出现频率 */
396     syms = get_symfreq(&sf, in);
397     /* 生成编码表 */
398     sc = calc_code(&sf);
399     root = sf[0];
400     /* 再次扫描文件, 使用之前生成的编码表来编码 */
401     rewind(in);
402     rc = write_code_table(out, sc, syms);
403     if (rc == 0)
404         rc = do_file_encode(in, out, sc);
405     /* 释放空间 */
406     free_tree(root);
407     free_encoder(sc);
408     return rc;
409 }
410
411 /* 把 out 解压成 in */
412 int huffman_decode_file(FILE *in, FILE *out)
413 {
414     node *root, *p;
415     int c;
416     unsigned int data_count;
417     /* 读入编码表 */
418     root = read_code_table(in, &data_count);
419     if (!root)
420         return 1;
421     /* 解码文件 */
422     p = root;
423     while (data_count > 0 && (c = fgetc(in)) != EOF) {

```

```

424     unsigned char byte = (unsigned char) c;
425     unsigned char mask = 1;
426     while (data_count > 0 && mask) {
427         p = byte & mask ? p->one : p->zero;
428         mask <<= 1;
429         if (p->leaf) {
430             fputc(p->symbol, out);
431             p = root;
432             --data_count;
433         }
434     }
435 }
436 free_tree(root);
437 return 0;
438 }

```

5.3 主体逻辑 huff.c

```

1  /*
2   * 文件 : huff.c
3   * 描述 : Huffman 压缩程序运行主逻辑
4   */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <assert.h>
11
12 #include "huffman.h"
13
14 #ifdef WIN32
15 #include <malloc.h>
16 extern int getopt(int, char **, char *);
17 extern char *optarg;
18 #else
19 #include <unistd.h>
20 #endif
21
22 static void version(FILE * out)
23 {
24     fputs("Huffman 压缩解压程序\n", out);
25 }
26
27 static void usage(const char *name, FILE * out)
28 {
29     fprintf(out,
30         "用法:  %s [< 输入文件 >] [< 输出文件 >] [ 选项 ]\n"
31         "样例:  %s 1 2          \t# 压缩文件 1 为 2\n"
32         "        %s 2 3 -u       \t# 解压文件 2 为 3\n"
33         "        %s -i1 -o2 -z \t# 压缩文件 1 为 2\n"
34         "选项:  \n"
35         "-i \t 输入文件\n"
36         "-o \t 输出文件\n"

```

```

37     "-u \t 解压缩\n"
38     "-z \t 压缩 (默认)\n"
39     "-h \t 显示本帮助\n"
40     "-v \t 显示版本信息\n",
41     name, name, name, name);
42 }
43
44 int main(int argc, char **argv)
45 {
46     bool compress = true;
47     int opt;
48     const char *file_in = NULL, *file_out = NULL;
49     FILE *in = stdin;
50     FILE *out = stdout;
51
52     /* 参数不足时打印用法 */
53     if (argc < 3) {
54         usage(argv[0], stdout);
55         return 0;
56     }
57
58     /* 默认将前两个参数作为文件名 */
59     file_in = argv[1];
60     file_out = argv[2];
61
62     /* 获取短命令行参数 */
63     while ((opt = getopt(argc, argv, "i:o:uzhv")) != -1) {
64         switch (opt) {
65             case 'i': /* 输入文件 */
66                 file_in = optarg;
67                 break;
68             case 'o': /* 输出文件 */
69                 file_out = optarg;
70                 break;
71             case 'z': /* 压缩 */
72                 compress = true;
73                 break;
74             case 'u': /* 解压 */
75                 compress = false;
76                 break;
77             case 'h': /* 帮助 */
78                 usage(argv[0], stdout);
79                 return 0;
80             case 'v': /* 关于 */
81                 version(stdout);
82                 return 0;
83             default: /* 参数错误 */
84                 usage(argv[0], stderr);
85                 return 1;
86         }
87     }
88
89     /* 打开输入文件 */
90     in = fopen(file_in, "rb");
91     if (!in) {

```

```

92     fprintf(stderr, "无法打开输入文件 '%s': %s\n",
93             file_in, strerror(errno));
94     return 1;
95 }
96
97 /* 打开输出文件 */
98 out = fopen(file_out, "wb");
99 if (!out) {
100     fprintf(stderr, "无法打开输出文件 '%s': %s\n",
101             file_out, strerror(errno));
102     return 1;
103 }
104
105 /* 主程序返回压缩解压函数的返回值 */
106 return compress ?
107     huffman_encode_file(in, out) : huffman_decode_file(in, out);
108 }

```

6 调试分析

起初程序解压结果一直和原文不一致，经检查发现是在内存中和硬盘中字节存储顺序不一致，有必要采取统一编码方式。使用网络库是为了使用函数 `ntohl()` 和 `htonl()`，在网络字节顺序和本地字节顺序之间进行转换。之所以这样，是因为本地操作系统的二进制存储结构可能不尽相同（从大到小或从小到大），为了保证读写二进制文件时不发生错误，需要执行相应的转换保证字节编码方式一致。这两个函数在 Linux 和 Windows 下分别定义在 `<netinet/in.h>` 和 `<winsock2.h>` 中，需要编译时指明调用的库。

还有一个问题在于，如果对一个非本程序压缩的文件进行解压的话，将会产生错误。可以采用标识文件头来解决，这里省略。

Huffman 编码 [2] 是最优的字符压缩编码，但不是最优的压缩编码。如果在一个文件中，各种符号出现的频率接近一致，那么 Huffman 编码并不能有效地减少文件体积，反而由于引入了编码表可能会导致文件变大，这点在非文本文件压缩中体现得尤为明显。甚至把一个文件进行多重压缩之后，文件体积会越来越大。当然还原文件还是可以正常工作的，不过这显然违背了压缩的原则——减小文件体积。所以在日常应用中，我们不使用 Huffman 编码。相应地，会采用 LZ77[4], LZ78[5], LZW[6], ZIP, RAR, LZMA 等算法，这些都是我们常见的压缩格式，实践证明这些也是有效的压缩算法。

7 测试结果

拿到源代码之后参照第 3 节运行环境进行编译，得到对应平台的二进制可执行文件之后，就可以进行测试了。

首先是在 Windows 平台上，编译之后生成 huff.exe，在当前目录下键入程序名称 huff 即可显示帮助信息，如图 2：



```
管理员: C:\Windows\system32\cmd.exe
D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>ls
LICENSE Makefile huff.c huff.exe huffman.c huffman.h

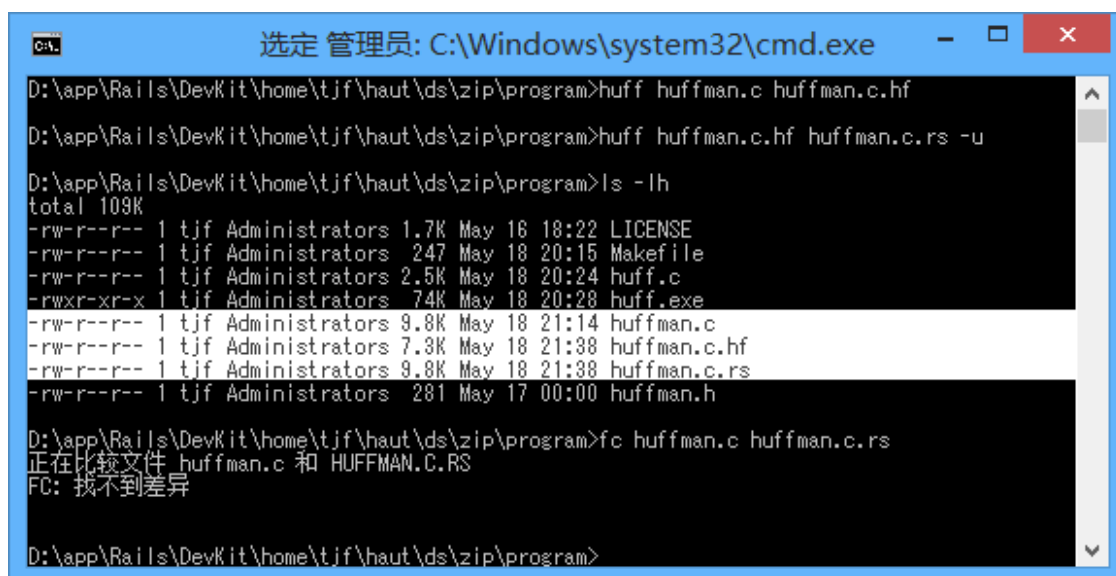
D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>huff
用法: huff [ < 输入文件 > ] [ < 输出文件 > ] [ 选项 ]
样例: huff 1 2 # 压缩文件 1 为 2
      huff 2 3 -u # 解压文件 2 为 3
      huff -i1 -o2 -z # 压缩文件 1 为 2

选项:
-i 输入文件
-o 输出文件
-u 解压缩
-z 压缩 (默认)
-h 显示本帮助
-v 显示版本信息

D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>
```

图 2

如图 3。我们尝试把编写的源代码文件 huffman.c 压缩成 huffman.c.hf，然后把 huffman.c.hf 解压成 huffman.c.rs。将原来的 huffman.c 和 huffman.c.rs 进行比较，发现没有差异，说明压缩算法没有发生数据损失。查看一下文件大小，发现压缩文件的体积确实变小了。这里，9.8K 的源文件压缩成了 7.3K 的压缩文件。



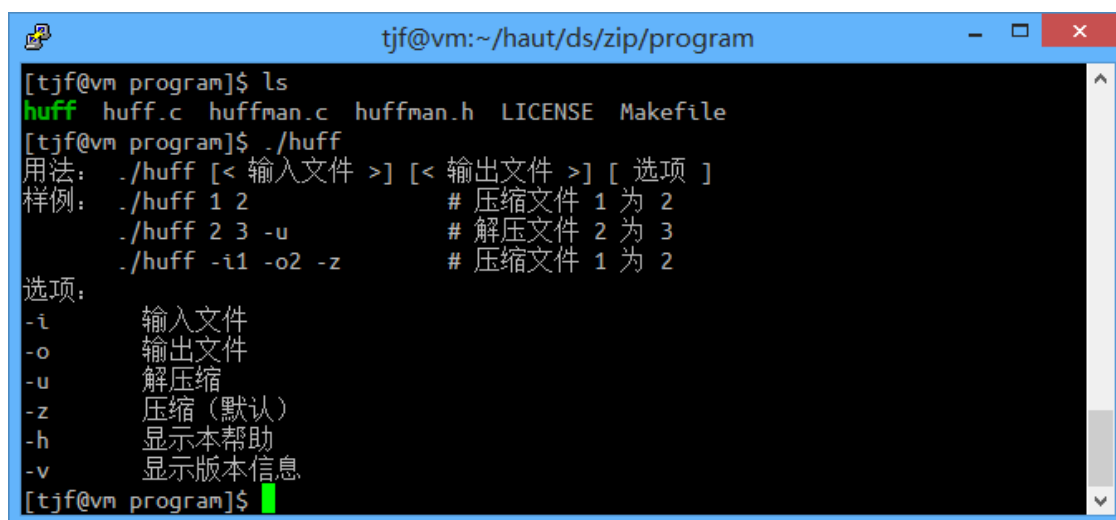
```
选定 管理员: C:\Windows\system32\cmd.exe
D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>huff huffman.c huffman.c.hf
D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>huff huffman.c.hf huffman.c.rs -u
D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>ls -lh
total 109K
-rw-r--r-- 1 tjf Administrators 1.7K May 16 18:22 LICENSE
-rw-r--r-- 1 tjf Administrators 247 May 18 20:15 Makefile
-rw-r--r-- 1 tjf Administrators 2.5K May 18 20:24 huff.c
-rwxr-xr-x 1 tjf Administrators 74K May 18 20:28 huff.exe
-rw-r--r-- 1 tjf Administrators 9.8K May 18 21:14 huffman.c
-rw-r--r-- 1 tjf Administrators 7.3K May 18 21:38 huffman.c.hf
-rw-r--r-- 1 tjf Administrators 9.8K May 18 21:38 huffman.c.rs
-rw-r--r-- 1 tjf Administrators 281 May 17 00:00 huffman.h

D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>fc huffman.c huffman.c.rs
正在比较文件 huffman.c 和 HUFFMAN.C.RS
FC: 找不到差异

D:\app\Rails\DevKit\home\tjf\haut\ds\zip\program>
```

图 3

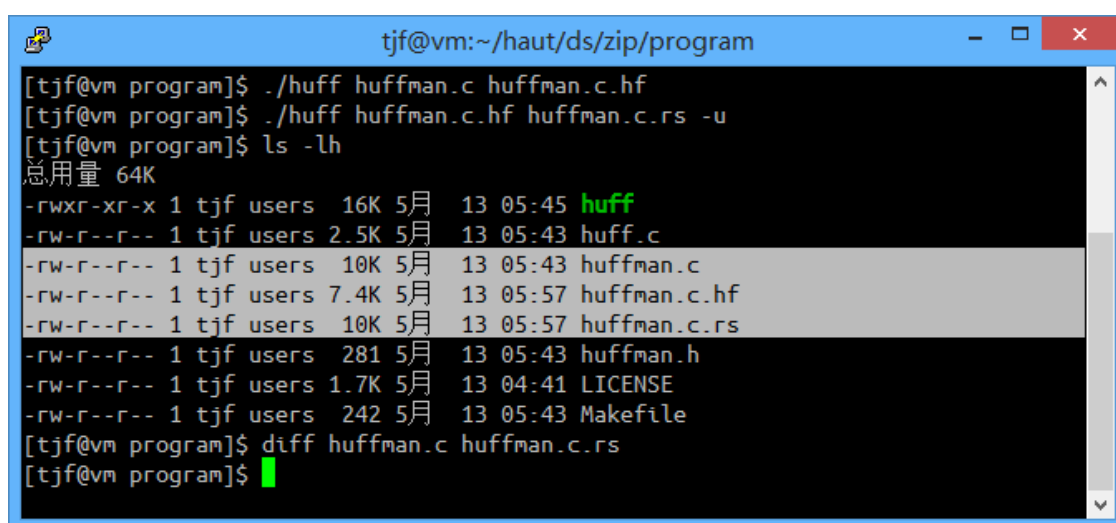
接着我们在 Linux 下进行测试。这里我使用 Putty 以 SSH 连接到虚拟机中的操作系统，执行我需要的命令。图 4 显示了键入程序路径 `./huff` 显示的帮助信息，我们的程序正确地输出了提示信息。



```
tjf@vm:~/haut/ds/zip/program
[tjf@vm program]$ ls
huff huff.c huffman.c huffman.h LICENSE Makefile
[tjf@vm program]$ ./huff
用法: ./huff [< 输入文件 >] [< 输出文件 >] [选项]
样例: ./huff 1 2          # 压缩文件 1 为 2
      ./huff 2 3 -u       # 解压文件 2 为 3
      ./huff -i1 -o2 -z   # 压缩文件 1 为 2
选项:
-i      输入文件
-o      输出文件
-u      解压缩
-z      压缩 (默认)
-h      显示本帮助
-v      显示版本信息
[tjf@vm program]$
```

图 4

如图 5，做相同的测试。由于操作系统存储方式的差异，文件大小有点不一样。但是算法依然是正确的，程序依然产生了正确的输出。



```
tjf@vm:~/haut/ds/zip/program
[tjf@vm program]$ ./huff huffman.c huffman.c.hf
[tjf@vm program]$ ./huff huffman.c.hf huffman.c.rs -u
[tjf@vm program]$ ls -lh
总用量 64K
-rwxr-xr-x 1 tjf users 16K 5月 13 05:45 huff
-rw-r--r-- 1 tjf users 2.5K 5月 13 05:43 huff.c
-rw-r--r-- 1 tjf users 10K 5月 13 05:43 huffman.c
-rw-r--r-- 1 tjf users 7.4K 5月 13 05:57 huffman.c.hf
-rw-r--r-- 1 tjf users 10K 5月 13 05:57 huffman.c.rs
-rw-r--r-- 1 tjf users 281 5月 13 05:43 huffman.h
-rw-r--r-- 1 tjf users 1.7K 5月 13 04:41 LICENSE
-rw-r--r-- 1 tjf users 242 5月 13 05:43 Makefile
[tjf@vm program]$ diff huffman.c huffman.c.rs
[tjf@vm program]$
```

图 5

可以看到，我们的程序实现了预定目标，工作良好。

参考文献

- [1] K. Loudon, 算法精解 (C 语言描述). 北京: 机械工业出版社, 2012.
- [2] D. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, pp. 1098–1101, Sept 1952.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, 算法导论. 北京: 机械工业出版社, 2012.
- [4] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *Information Theory, IEEE Transactions on*, vol. 23, pp. 337–343, May 1977.
- [5] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *Information Theory, IEEE Transactions on*, vol. 24, pp. 530–536, Sep 1978.
- [6] T. Welch, “A technique for high-performance data compression,” *Computer*, vol. 17, pp. 8–19, June 1984.
- [7] D. R. Richardson, “A huffman coding library and command line interface to the library.” <https://github.com/drichardson/huffman>.
- [8] D. A. Lelewer and D. S. Hirschberg, “Data compression,” *ACM Computing Surveys (CSUR)*, vol. 19, no. 3, pp. 261–296, 1987.
- [9] G. E. Blelloch, “Introduction to data compression,” 2001.
- [10] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. San Francisco: Morgan Kaufmann Publishing, 1999.
- [11] D. E. Knuth, *The Art Of Computer Programming*. Pearson Education, 1968–2011.
- [12] 高德纳, 计算机程序设计艺术. 北京: 国防工业出版社, 1992–2010.
- [13] 邓建松, 彭冉冉, and 陈长松, *L^AT_EX 2_ε 科技排版指南*. 北京: 科学出版社, 2001.

心得体会

信息科学与工程学院课程设计成绩评价表

课程名称：数据结构课程设计

设计题目：用哈夫曼算法压缩文件

专业：计算机

班级：1303 班

姓名：田劲锋

学号：201316920311

序号	评审项目	分 数	满分标准说明
1	内容		思路清晰；语言表达准确，概念清楚，论点正确；实验方法科学，分析归纳合理；结论严谨，设计有应用价值。任务饱满，做了大量的工作。
2	创新		内容新颖，题目能反映新技术，对前人工作有改进或突破，或有独特见解
3	完整性 实用性		整体构思合理，理论依据充分，设计完整，实用性强
4	数据准确 可靠		数据准确，公式推导正确
5	规范性		设计格式、绘图、图纸、实验数据、标准的运用等符合有关标准和规定
6	纪律性		能很好的遵守各项纪律，设计过程认真；
7	答辩		准备工作充分，回答问题有理论依据，基本概念清楚。主要问题回答简明准确。在规定的时间内作完报告。
总分			
综合意见	<div style="text-align: right;">指导教师：_____ 年 月 日</div>		