数据结构实验总结报告

李博杰 PB10000603

一、调试过程中遇到哪些问题？

（1）在二叉树的调试中，从广义表生成二叉树的模块花了较多时间调试。

由于一开始设计的广义表的字符串表示没有思考清晰，处理只有一个孩子的节点时发生了混乱。调试之初不以为是设计的问题，从而在代码上花了不少时间调试。

目前的设计是：

```
Tree = Identifier(Node,Node)
Node = Identifier | () | Tree
Identifier = ASCII Character
```

例子：a(b(),f),c(d,e))

这样便消除了歧义，保证只有一个孩子的节点和叶节点的处理中不存在问题。

（2）Huffman 树的调试花了较长时间。Huffman 编码本身并不难处理，麻烦的是输入输出。

①Huffman 编码后的文件是按位存储的，因此需要位运算。

②文件结尾要刷新缓冲区，这里容易引发边界错误。

在实际编程时，首先编写了屏幕输入输出（用 0、1 表示二进制位）的版本，然后再加入二进制文件的读写模块。主要调试时间在后者。

二、要让演示版压缩程序具有实用性，哪些地方有待改进？

（1）压缩文件的最后一字节问题。

压缩文件的最后一字节不一定对齐到字节边界，因此可能有几个多余的 0，而这些多余的 0 可能恰好构成一个 Huffman 编码。解码程序无法获知这个编码是否属于源文件的一部分。因此有的文件解压后末尾可能出现一个多余的字节。

解决方案：

①在压缩文件头部写入源文件的总长度（字节数）。需要四个字节来存储这个信息（假定文件长度不超过 4GB）。

②增加第 257 个字符（在一个字节的 0~255 之外）用于 EOF。对于较长的文件，会造成较大的损耗。

③在压缩文件头写入源文件的总长度%256 的值，需要一个字节。由于最后一个字节存在或不存在会影响文件总长%256 的值，因此可以根据这个值判断整个压缩文件的最后一字节末尾的 0 是否在源文件中存在。

（2）压缩程序的效率问题。

在编写压缩解压程序时

①编写了屏幕输入输出的版本

②将输入输出语句用位运算封装成一次一个字节的文件输入输出版本

③为提高输入输出效率，减少系统调用次数，增加了 8KB 的输入输出缓存窗口

这样一来，每写一位二进制位，就要在内部进行两次函数调用。如果将这些代码合并起来，再针对位运算进行一些优化，显然不利于代码的可读性，但对程序的执行速度将有一定提高。

（3）程序界面更加人性化。

```
Huffman Tree Demo (C) 2011-12-16 boj
Usage: huffman [-c file] [-u file] output_file
  -c Compress file.  e.g. huffman -c test.txt test.huff
  -u Uncompress file.   e.g. huffman -u test.huff test.txt
```

目前的程序提示如上所示。如果要求实用性，可以考虑加入其他人性化的功能。

三、调研常用的压缩算法，对这些算法进行比较分析

（一）无损压缩算法

①**RLE**

RLE 又叫 Run Length Encoding，是一个针对无损压缩的非常简单的算法。它用重复字节和重复的次数来简单描述来代替重复的字节。尽管简单并且对于通常的压缩非常低效，但它有的时候却非常有用（例如，JPEG 就使用它）。

变体 1：重复次数+字符

文本字符串：A A A B B B C C C C D D D D，编码后得到：3 A 3 B 4 C 4 D。

变体 2：特殊字符+重复次数+字符

文本字符串：A A A A A B C C C C B C C C，编码后得到：B B 5 A B B 4 C B B 3 C。编码串的最开始说明特殊字符 B，以后 B 后面跟着的数字就表示出重复的次数。

变体 3：把文本每个字节分组成块，每个字符最多重复 127 次。每个块以一个特殊字节开头。那个特殊字节的第 7 位如果被置位，那么剩下的 7 位数值就是后面的字符的重复次数。如果第 7 位没有被置位，那么剩下 7 位就是后面没有被压缩的字符的数量。例如：文本字符串：A A A A A B C D E F F F。编码后得到：85 A 4 B C D E 83 F（85H= 10000101B、4H= 00000100B、83H= 10000011B）

②**Huffman**

哈夫曼编码是无损压缩当中最好的方法。它使用预先二进制描述来替换每个符号，长度由特殊符号出现的频率决定。常见的符号需要很少的位来表示，而不常见的符号需要很多为来表示。

哈夫曼算法在改变任何符号二进制编码引起少量密集表现方面是最佳的。然而，它并不处理符号的顺序和重复或序号的序列。

③**Rice**

Rice 编码背后的基本思想是尽可能的用较少的位来存储多个字（正像使用哈夫曼编码一样）。实际上， Rice 类似静态的哈夫曼编码（例如，编码不是由实际数据内容的统计信息决定，而是由小的值比高的值常见的假定决定）。编码非常简单：将值 X 用 X 个‘1’位之后跟一个 0 位来表示。

对于由大 word（例如：16 或 32 位）组成的数据和教低的数据值，Rice 编码能够获得较好的压缩比。音频和高动态变化的图像都是这种类型的数据，它们被预处理过（例如 delta 相邻的采样）。

尽管哈夫曼编码处理这种数据是最优的，却由于几个原因而不适合处理这种数据（例如：32 位大小要求 16GB 的柱状图缓冲区来进行哈夫曼树编码）。因此一个比较动态的方式更适合由大 word 组成的数据。

④**LZ77**

在 LZ 压缩算法的背后是使用 RLE 算法用先前出现的相同字节序列的引用来替代。

简单的讲，LZ 算法被认为是字符串匹配的算法。例如：在一段文本中某字符串经常出现，并且可以通过前面文本中出现的字符串指针来表示。当然这个想法的前提是指针应该比

字符串本身要短。

例如，在上一段短语“字符串”经常出现，可以将除第一个字符串之外的所有用第一个字符串引用来表示从而节省一些空间。

一个字符串引用通过下面的方式来表示：

1. 唯一的标记
2. 偏移数量
3. 字符串长度

使用 LZ77 的一个问题是由于算法需要字符串匹配，对于每个输入流的单个字节，每个流中此字节前面的哪个字节都必须被作为字符串的开始从而尽可能的进行字符串匹配，这意味着算法非常慢。

另一个问题是为了最优化压缩而调整字符串引用的表示形式并不容易。例如，必须决定是否所有的引用和非压缩字节应该在压缩流中的字节边界发生。

⑤字典算法

最为简单的压缩算法。把文本中出现频率比较多的单词或词汇组合做成一个对应的字典列表，并用特殊代码来表示这个单词或词汇。

对包含某些特殊名词很多的自然语言文本比较有效，不属于通用压缩算法。

（二）有损压缩算法

一般用于视频、音频、图片等多媒体信息的压缩。

有两种基本的有损压缩机制：

一种是有损变换编解码，首先对图像或者声音进行采样、切成小块、变换到一个新的空间、量化，然后对量化值进行熵编码。

另外一种是预测编解码，先前的数据以及随后解码数据用来预测当前的声音采样或者或者图像帧，预测数据与实际数据之间的误差以及其它一些重现预测的信息进行量化与编码。

有些系统中同时使用这两种技术，变换编解码用于压缩预测步骤产生的误差信号。

有损与无损压缩比较

有损方法的一个优点就是在有些情况下能够获得比任何已知无损方法小得多的文件大小，同时又能满足系统的需要。

有损方法经常用于压缩声音、图像以及视频。有损视频编解码几乎总能达到比音频或者静态图像好得多的压缩率（压缩率是压缩文件与未压缩文件的比值）。音频能够在没有察觉的质量下降情况下实现 10:1 的压缩比，视频能够在稍微观察质量下降的情况下实现如 300:1 这样非常大的压缩比。有损静态图像压缩经常如音频那样能够得到原始大小的 1/10，但是质量下降更加明显，尤其是在仔细观察的时候。

当用户得到有损压缩文件的时候，譬如为了节省下载时间，解压文件与原始文件在数据位的层面上看可能会大相径庭，但是对于多数实用目的来说，人耳或者人眼并不能分辨出二者之间的区别。

一些方法将人体解剖方面的特质考虑进去，例如人眼只能看到一定频率的光线。心理声学模型描述的是声音如何能够在不降低声音感知质量的前提下实现最大的压缩。

参考资料：维基百科