# 需求分析

编写一个压缩软件，能使用两种压缩算法，反馈压缩率，并提示用户使用较高的那种算法，能够使用两种算法进行压缩。识别压缩文件的原始压缩方法是否是该压缩算法，若是能够解压。

# 概要和详细设计

程序功能：

根据用户输入的路径对指定的文件进行压缩/解压，压缩时会先反馈两种压缩算法的压缩率，提示用户使用较高压缩率的算法。并且能够判断已压缩文件的原始压缩算法是否是那两种算法之一，若是能够解压。

概要设计：

我采用的是Huffman和lz77两种压缩算法，分别定义了结构体Huffman和Lz77处理这两种不同的压缩和解压问题。

(首先，我们规定这里的字符指的是读入的byte，共0~255 256种不同字符)

Huffman:首先统计出所有在文件中出现的字符的出现次数，根据每个单字符的出现次数为叶子建立哈夫曼树，从根节点出发遍历一遍树上结点找出每个字符对应的编码，然后输出压缩后的文件，它由映射表和原文件的压缩内容两部分组成。

映射表：开头2byte表示出现的字符个数byteAppeaer，后面接着byteAppear个元组组成映射表，每个元组第一个byte表示对应的字符bi，第二个表示映射后的码长li，而后就是长度为li的01串组成的码。

原文件的压缩内容则比较简单，将原来的字符换成他对应的码即可。

输出的结果是以bit为单位的，但是要输出到文件中，起码是以byte为单位，所以还需要一定的转换。

做法是：先在最后得到的bit串末尾加个1，然后不能整除8则补0.

解压的方法也很简单，先取前16位求出byteAppear，然后建立反向映射表，Huffman编码又是前缀码，映射表的结束位置后一个字符作为第一个原文件字符串的首字符，后移结束位置，第一个出现在映射表中的串对应的字符就是我们要找的字符，然后再后移位置指针，直到走到文件尾（最后一个1）为止。

Lz77:将文件中在之前已经出现过的部分用二元组(beg,len)(即出现地址，长度替代)，此外为了解码方便，用特殊字符放在每个二元组之前作为标记。

解码时从前向后遍历，出现特殊字符，就将后两位用他们对应的字串替代即可。

由于文件较大，可能beg或len会大于255，所以采用拼接的方法将他们8位8位拼接，用sign后面的数的前4bit表示共几个byte组成的beg,后4bit表示几个byte组成len.

除此之外，每次压缩我还将原文件的后缀也保存在了压缩后的结果中，这样解压时可以直接还原到原来的文件，而压缩后的文件名就是原文件文件名。

我的感受：

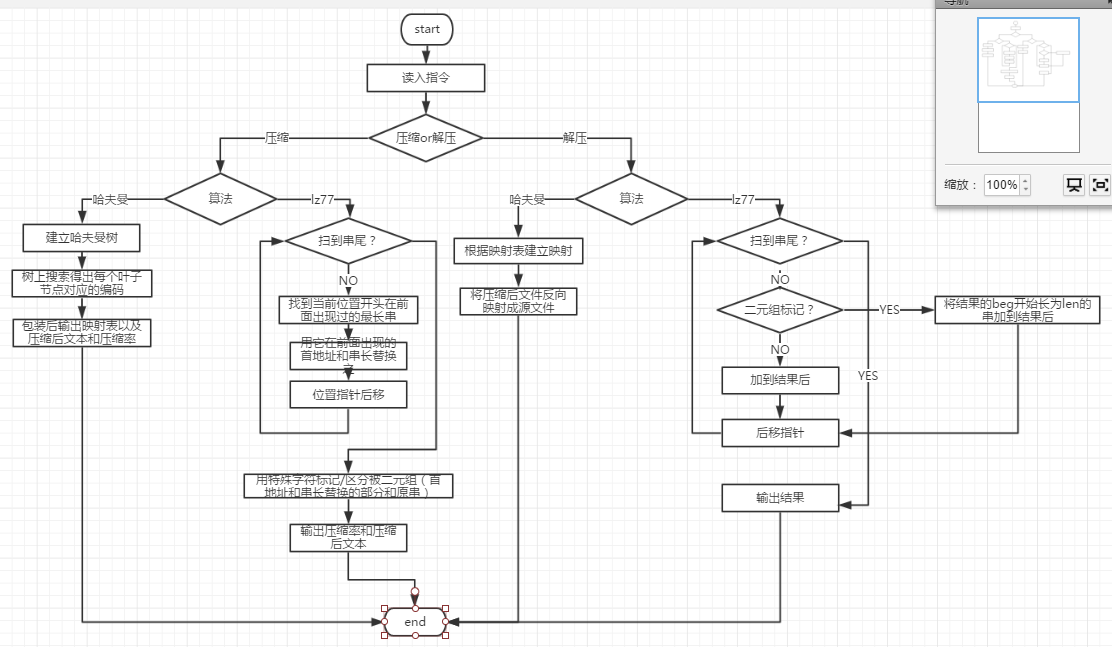
之前用c++写，各种问题，后来发现用java实现应该效果更佳。以后还是应该先根据想法挑出合适的语言再动手。

主要问题出在与文件交互上，java有byte流还有专门byte数据类型，用Huffman编码时会方便的多，而用c++先转化为char再读入后，由于前32个控制字符的存在，输出的结果和再次读入的结果会完全不相同！！！血的教训。

最明显的例子就是ascll为0对应的字符再读入会被当成ascll码为32的空格。

而且我发现把串暂时存储进string和直接输出竟然也会有差异，用二进制输入输出也失败了，可能这不是c++擅长的事情。

# 程序结构



# 代码解释

Line 19 to 36：定义类Debug用于调试。

先解释一些两个类都有的函数方法：

Int byteToInt()将byte类型转换成Int，直接强制类型转换的结果不符合我的要求，因为byte是带符号的，有些浪费位数，所以把<0的部分+256，相当于当作unsigned byte使用。

compressName\decompressName返回压缩后的文件名和解压后的文件名，我的设定是除了最后一个.开始的标注文件类型的标识，其他部分都保持不变。Huffman类中compressName就将最后一个”.”后面的内容改成huf，而Lz77则改成lz，decompressName就改回去。

Line 37 to 291:处理Huffman编码的相关问题。

先看函数compress():

buildTree()是建立Huffman树和映射,output（）是输出映射表及压缩后文本，output返回压缩后文件名。

buildTree():

首先统计每个字符出现次数在cnt[]中，然后以他们为叶子建树。

树上结点用结构体TNode表示，id是每个结点的唯一标号，lid是左孩子标号，rid是右孩子标号（-1表示没有），val是（叶子）的对应字符，fre是（该节点所有叶子的总出现频率）。

首先将所有叶子加入优先队列(Q),优先级定义为fre越小越高，这样每次取出优先队列的前两小fre,合并后的结点的fre等于他们fre之和加入队列，直到仅一个结点为止，就将树建好了。在此过程中，最后一个加入Q的元素一定就是根节点，用root记录每个结点标号，最后结束时root就是根结点标号。

Void dfs(int now,string s)now就是结点标号，s是对应的码，若当前结点是叶子（val!=-1）那么就将他代表的字符val对应的码设置为s.否则就搜索他的左子结点，s+’0’，再搜索右孩子，s+’1’;

String output():输出结果，返回压缩后文件名。

Void setB()将要输出的内容（映射表+原文件）先转换成boolen数组，因为huffman编码后的码长长度不一，所以干脆都变成boolen让他重新组合。

byteToBit\stringToBit都是将byte\string转换成对应的boolen加在数组b后面

Byte[] setOut():将转换后的boolen数组再重新组合成byte数组以便输出。

Int compressSize():跟compress()类似，但是只返回压缩后文件大小，也省略了无关的操作。

Line 235 to 291:void decompress():

Void setD()将输入转换为boolen数组。

Line 251 to 264:首先取前两位得到byteAppear,然后取出对每个匹配，取前8个bit得到原来的byte,然后再取8个bit得到串长，然后把后面串长个bit取出就是该byte对应的码，将他们加入到map中。

对于后面还没有处理的bit都是原来的文件了，根据建立的映射表来映射即可得到原来的文件。

Line 184 to 265:处理lz77算法的相关问题

Line 322 to 338:

定义结构体LNode。从前向后遍历文件，若在前面出现，将该部分在之前的出现的首位置和长度设置LNode的beg和len，并将其加入arraylist<LNode> v;否则将该串存入LNode.c 再加入v，且LNode对应的beg置为-1用于标记。

Line 404 to 432:

遍历v，对于beg=-1的直接接在压缩后的串后面，否则先添加特殊标记字符，再将beg和len接在原串后。这里考虑到beg和len可能特别的大，一个byte可能不够用，我就决定多用几位将他们接起来。紧接标记字符后面的那个byte标记前4位和后4位分别标记用于记录beg和len的字节个数。另外标记字符选用0，而将原串中的0改为00两位。

这样解压时先读到0，再看后以为，若是0，则只是一个byte 0,否则说明后面是beg和len，把他们拼接起来得到最终的beg和len，取字串即可。（这个sign的使用是借用了转义字符的思想）

# 测试

运行程序根据提示输入合理，且文件不是很大，都是可以压缩和解压的，不分文件类型。

初测不超100KB都能够在较短时间内实现压缩。