CSCI 340
Data Structures and Algorithms
Spring 2018

Programming Project 4
Lempel-Ziv-Welch Data Compression and Decompression

Due Date: Tuesday, April 17
Points: 40

Many standard file compression programs, including Unix's compress and Windows based PKZip use variations on an algorithm created by Jakob Ziv and Abraham Lempel. Their scheme is based on entering phrases into a dictionary. Each phrase is associated with a small integer (byte), we will call its index. When a phrase is repeated, the index of the dictionary entry is output to the compressed file, rather than the phrase. Later the algorithm was improved by Terry Welch. You often see it referenced as LZW. We will implement a simplified version of the Lempel-Ziv algorithm that only works on text files.

The dictionary is a HashMap of strings and their indices. Initialize your HashMap with the standard ASCII characters with indices 0-127. We will limit the size of our dictionary to 256 entries, so each index will fit into a byte. If the dictionary fills to 25**6** entries **(last index is 255)**, do not add any more to it. (Aside: The dictionary is the primary place we are not following LZW. LZW dictionaries start out with 256 entries and grow to up to 4K. Thus, their indices are each 12 bits.)

The algorithm processes a file one character at a time. Start with `str` (the current string) set to “”.
The next character to be processed is `ch`.

```
if str + ch is in the dictionary
   set str to str + ch
else
   output the index of the str
   add str + ch to the dictionary
   set str to ch
end if
```

```
Repeat the process until you reach the end of the file.
Output the index of the final index.
```

Sample run:
Input: wed we wee web

str	ch	Output:	Changes to the dictionary
""	w	none	none
"w"	e	$77_h=119_{10}$	dictionary.put("we", 128)
"e"	d	$65_h=101_{10}$	dictionary.put("ed", 129)
"d"	" "	$64_h=100_{10}$	dictionary.put("d ", 130)
" "	w	$20_h=32_{10}$	dictionary.put(" w", 131)
"w"	e	none	none
"we"	" "	$80_h=128_{10}$	dictionary.put("we ", 132)
" "	w	none	none
" w"	e	$83_h=131_{10}$	dictionary.put(" we", 133)
"e"	e	$65_h=101_{10}$	dictionary.put("ee", 134)
"e"	" "	$65_h=101_{10}$	dictionary.put("e ",135)
" "	w	none	none
" w"	e	none	none
" we"	b	$85_h=133_{10}$	dictionary.put(" web", 136)
"b"	<eof>	$62_h=98_{10}$	

The original text is 14 characters (bytes) long. If each of the output codes takes a byte of storage, then the compressed representation is only 10 bytes long.

Task 1:
You are to implement the LZW compression algorithm. Have the main method prompt the user for the name of the file. Compress the file, naming the result the same as the original, but appending a `.lzw` suffix. The output file should be in binary. You do not need to do any error checking on the input file. You may assume the file contains only ASCII characters.

Task 2:
Write a Java program that implements LZW decompression. Have the main method prompt the user for the name of the file. The filename **MUST** end with `.lzw`. The resulting output file should be named the same, except strip off the `.lzw` suffix.

Your instructor is not giving you the algorithm for decompression. The algorithm is (almost) straightforward. You read in an index, look it up in the decompression dictionary and output the result. There two tricky parts to decompression.
1. The dictionary is not saved in the compressed file. You must recreate it during decompression. You start by assuming the same **original** dictionary entries as in compression (except with key and value reversed) and then add new entries to that dictionary as you decompress.
2. There is only one special case where the index that you read in is not already in the dictionary. This case occurs with highly repetitive data. Consider: `eeeeeeeeee` (that's 10 e's).
It encodes to be: 101_{10} 128_{10} 129_{10} 130_{10}
where 101_{10} is for the first e, 128_{10} is the code for 2 e's, 129_{10} is the code for 3 e's, and 130_{10} is the code for the final 4 e's. When decoding, however, 128_{10}, 129_{10} and 130_{10} will not yet have been added to the dictionary.

More generally the situation occurs whenever the encoder encounters the input of the form cScSc, where c is a single character, S is a string and cS is already in the dictionary. The encoder outputs the symbol for cS putting new symbol for cSc in the dictionary. Next it sees the cSc in the input and sends the new symbol it just inserted into the dictionary**. If you read a code that is not in the dictionary, you can safely assume that it decodes to be the same as the previous code, with the first letter repeated at its end.**

Help Along the Way
Java's `RandomAccessFile` class makes it easy to read and a write a file one byte at a time. In the course directory on the lab machines there is a directory named `LZW` that contains sample code showing you how to handle byte IO.

The `LZW` directory also contains `HickoryDickoryDock.txt` and `HickoryDickoryDock.txt.lzw`. These are to help you test your code.

Submission:
Only electronic submission is required. Use the submit command to submit LZWCompression.java and LZWDecompression.java. Your instructor will use the following commands to test your code:

```
javac LZWCompression.java
java LZWCompression

javac LZWDecompression.java
java LZWDecompression
```