**CSL201: Assignment 5**

**Hashing for LZW**

**Due Date : October 27, 2013**

Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries. We could just transmit each word as an 18 bit number. But it is not a good solution, as there would be too many bits to represent any text document. So we are looking for a way to build the dictionary adaptively, so that we can compress any text message.

In this assignment, you will learn about the famous Lempel-Ziv-Welch (LZW) compression algorithm. Most compression algorithms including UNIX *compress* are based on the LZW algorithm. The LZW algorithm compresses a given text by figuring out patterns which appear very often in the text. The basic idea is to replace patterns which appear many times by a much shorter code. Here is the algorithm.

**LZW Compression**

The LZW compression method maps strings of text characters to numeric codes. To begin with, all characters that may occur in the text are assigned a code.

w = NIL;

 while ( read a character k )

 {

 if wk exists in the dictionary

 w = wk;

 else

 add wk to the dictionary;

 output the code for w;

 w = k;

 }

 For example, suppose the text file to be compressed is the string :

**aaabbbbbbaabaaba**

It is known beforehand that the given string is composed of the characters `a' and `b'. So initially `a' is assigned the code 0 and `b' is assigned the code 1.

The mapping between character strings and their codes is stored in a dictionary. Each dictionary entry has two fields : "key" and "code". The character string represented by "code" is stored in the field "key". The initial dictionary for our example is given by the first two columns of the table below (i.e., codes "0" and "1") :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| code  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| key  | a  | b  | aa  | aab  | bb  | bbb  | bbba  | aaba  |

Beginning with the dictionary initialized as above, the LZW compressor repeatedly finds the longest prefix "p" *of the unencoded part of the input file* that is in the dictionary and outputs its code. If there is a next character "c" in the input file, then "pc" is assigned the next code ("pc" is the prefix string "p" followed by the character "c"). “pc” and its code are inserted into the dictionary. This strategy is called the LZW rule.

Let us now try this rule on our example string.

The longest prefix of the input that is in the dictionary is "a".

Its code,0, is the output. The longest prefix of remaining text is the string "aa" which is assigned the code 2 and entered into the dictionary.

"aa" is the longest prefix of the remaining string that is already in the dictionary. Its code, 2, is outputted; and the string "aab" is assigned the code 3 and entered into the dictionary.

Following the output of the code 2, the code for "b" is output; "bb" is assigned code 4 and entered into the dictionary. Then the code for "bb" is output, and "bbb" is entered into the table with code 5.

 Next the code 5 is output, and "bbba" is entered with code 6. Then the code 3 is output for "aab", and "aaba" is entered into the dictionary with code 7.

 Finally the code 7 is output for the remaining string "aaba". Our sample string is encoded as the string 0214537.

Your assignment is to implement the LZW algorithm. You can assume that the input text contains English characters (capital and small), space, and full stop. You need to write two programs - one for compressing an input text, and the other to uncompress the compressed text to the original text. To implement the dictionary of code and key, you will use a hash table. Use your own hash function, but you must use quadratic probing to resolve collisions. Your program should accept a text file , compress it, and indicate the amount of compression.