Assignment 4 : LZW Compression

1 Introduction

LZW is a famous compression algorithm. Here, we assign a 16-bit code for sequences of ASCII characters. First go through `https://en.wikipedia.org/wiki/Lempel-Ziv-Welch` for the overview of the algorithm.

2 Initialization and Updation of the dictionary

Before starting compression, initialize the dictionary as follows

ASCII	16-bit code
0000000 'NUL'	0000000000000000
0000001 'SOH'	0000000000000001
⋮	⋮
1111111 'DEL'	0000000001111111

When a new character sequence is encountered, assign the **'least available'** 16-bit code to that sequence and add it to the dictionary.

2.1 Example LZW encoding

Consider the following character sequence

`aaabbbbbbaabaaba`

Dictionary is initialized as above. Pointer starts at the beginning of the sequence. The longest available sequence starting at the pointer is `'a'`. Its 16-bit code is `'0000000001100001'`. Now for the character sequence `'aa'`, we assign the lowest available 16-bit code i.e., `'0000000010000000'`. Now the pointer is after the first character. Longest available sequence in the dictionary now is `'aa'`. Its code is `'0000000010000000'`. Add the sequence `'aab'` to the dictionary with code `'0000000010000001'`. Pointer is currently after third character. Longest available character sequence is `'b'`. Its 16-bit code is `'0000000001100010'`. Similarly, after finding the longest prefix "p" of the unencoded part of the string that is in the dictionary, if there is a next character "c" left in the string, then "pc" ("pc" is the prefix string "p" followed by the character "c") is assigned the next code and inserted into the dicitionary. The final dictionary looks as given below. Note that there is no need to store the dictionary for decoding back the original dictionary. It would be recreated as you go on decode each 16-bit string from the compressed file.

Character Sequence	16-bit code
'NUL'	0000000000000000
'SOH'	0000000000000001
⋮	⋮
'DEL'	0000000001111111
aa	0000000010000000
aab	0000000010000001
bb	0000000010000010
bbb	0000000010000011
bbba	0000000010000100
aaba	0000000010000101

Hence, the complete code for the sequence `'aaabbbbbbaabaaba'` is

```
0000000001100001 0000000010000000 0000000001100010 0000000010000010 0000000010000011
0000000010000001 0000000010000101
```

We shall now see how the above code is decoded: We initialize the dictionary as given above. `0000000001100001` corresponds to the character `'a'`. Hence, the first character is `'a'`. Now we add this and the next character (We still don't know what the next character is!) to the dictionary along with the least available 16-bit code. Next 16-bit code is `0000000010000000`. We know that sequence corresponding to this is `'a'` followed by the character at second place. Now, we can infer that this sequence is `'aa'`. Hence, the text till third place is `'aaa'`. Now, we add `'aa'` followed by the character in 4th place(We don't know what it is yet!) to the dictionary. Next code is `0000000001100010` which corresponds to `'b'`. Thus, the sequence to be added in the previous step was `'aab'`. We can proceed like this and construct the entire original sequence.

Also, observe that using a 16-bit code, the dictionary can store only 2^{16} character sequences. When the dictionary becomes full, don't add new sequences to the dictionary and proceed with that dictionary till the end.

2.2 Implementing dictionary

Use **hashing** to implement dictionary. Use a hash function made **by yourself**. Experiment with various hash-functions. Use quadratic probing to handle collisions.

3 Writing 16-bit code to a file

You will compress the files which contain only ASCII characters. To print the 16-bit code corresponding to `'aa'` i.e., `'0000000001000000'`, use something like below

```
OutputStream os = new FileOutputStream(file);
bytes code[2];
code[0] = 0; // 0 is the signed 8-bit value of 00000000
code[1] = -128; // -128 is the signed 8-bit value of 1000000
os.write(code);
```

The above code will write the two bytes to the file.

4 Deliverables

You need to submit following two files one for compression and another for decompression.

- `Compress.java`
  There will be two input arguments. `args[0]` is the file name of the file that is to be compressed. `args[1]` is the file name where you have to write the compressed encoding.
- `Decompress.java`
  There will be two input arguments. `args[0]` is the file name of the file containing compressed encoding. `args[1]` is the file name where you have to write the decompressed output.

Make sure that your submissions work when we execute the following commands

```
$ javac Compress.java
$ java Compress original_file compressed_file
```

The above should take the file by the name `'original_file'` and generate a file named `'compressed_file'`. Similarly for `Decompress.java`.