Washington State University Vancouver

Systems Programming - CS 360

Assignment 1 - Due: 11:59AM Feburary 5

Instructor:
Ben McCamish

January 22, 2020

Overall Assignment - 100 points

Your task is to write an implementation of the Lempel-Ziv-Welch (LZW) compression/decompression algorithms in C99 for execution in the ENCS laboratory's Linux environment. Although you will find pseudo-code for these algorithms below, you may want to further familiarize yourself with LZW and its history by following this link to its Wikipedia page: https://en.wikipedia.org/wiki/Lempel–Ziv–Welch.

Your program will be organized as a static linkable library with a pre-determined interface. Object-oriented techniques will be applied to implement an object-style interface in C, as well as among your internal components.

Details

After unpacking the template you will find a subdirectory named `lzwLib`, this directory will contain the source files, object files, header files and Makefiles associated with your implementation.

The `lzwLib` subdirectory contains another subdirectory: `include`, where the header files are located. The header file `lzw.h` describes the interface you must implement and cannot be modified. The structure and contents of the remaining header files you can regard as a starting suggestion that you can use, but with your own additions and modifications.

Ultimately, you will submit a `.zip` file containing the `lzwLib` directory and its contents, such that its Makefile will build the `lzwLib.a` library. Your code will be linked with the instructor's test bed to run/test/evaluate your implementation. Of course, you will want to link it your own test bed (main program) to do your own testing.

`Makefiles` are already implemented to build the library and can be used as is, unless you modify the template structure for your implementation. Your submission must contain a `Makefile` in the `lzwLib` directory that builds the target `lzwlib.a` from your source files. A clean target must also be present. All targets should accurately list their dependencies. Failure to comply with the Makefile requirements may result in a score of 0 for the assignment.

Testing

You are expected to create and run your own tests of your `lzwlib.a` library to verify its correct functioning. You will want to run it with both text and binary files, large and small. This program should be straightforward to test. If you encode data, and then decode it, the result should be identical byte-for-byte to the original data. You can use the Unix `diff` utility for comparing files on a byte-for-byte basis. Also, if you want to delve into the contents of binary files, `hexdump` may be useful, but lets hope you don't have to do that! You will probably want to start debugging with small text files to observe the algorithm's proper functioning. The Linux debugger `gdb`, and its graphical interface `ddd`, may be valuable tools during your code development.

I have also posted a (Linux) binary executable containing my solution to this assignment on the website. To obtain a definition of the program's command line options, invoke the program in the shell in one of these two ways:

```
$ ./lzw
$ ./lzw -help
```

It will produce a "usage" message explaining how to compress/decompress files with my implementation. Your implementation should (as in must!) correctly decompress data compressed by my implementation, and vice-versa. If such is not the case, you have a problem in your code. Also, turning on various levels of debugging in my executable may help you gain a better understanding of the LZW algorithm and how it is applied.

Dynamic Memory Management

Undoubtedly, you will be using the `stdlib.h` functions `malloc()` and `free()`, and perhaps others, to build persistent data structures in the heap. Your program must not "leak" heap memory. You will want to use the Linux `valgrind` tool to find, identify, and remedy such problems.

Style

- You are expected to use proper indention. Adopt a readable naming convention and adhere to it. Keep your procedures short and avoid repetitious code resulting from indiscriminate "cutting and pasting". Add comments where needed to explain your code, but don't just repeat the obvious. Avoid "magic numbers" and instead define macro constants with meaningful names. The overall goal is code that is readable and understandable by both you and a proficient C programmer.
- Other than a global "debug" flag, there is no need for global variables, so do not define any. Your modules should be fully re-entrant. For example, if multiple dictionaries were to be used simultaneously, your code should function correctly, imposing no restrictions. It is permissible to use static variables and procedures in your modules, but nevertheless maintaining reentrancy.
- Use the compile flags present in `lzwLib/Makefile`. These will ensure you get the maximum benefit from the compiler's analysis. Correct your code to eliminate all compiler warnings (and, of course, errors).
- Maintain an object-oriented style. Code using a module/object should not directly access or modify data inside the object's struct. Instead, access and modification should be through interface methods, which are declared and described in the module's header file. You will likely want to add methods to the module/header files I supply. If you contemplate making wholesale changes to the module design I supply, be sure you are making improvements and not just avoiding techniques you are uncomfortable with.
- I encourage you to make use of assertions (`#include <assert.h>`) wherever you can. These can document and test the assumptions you make within your code. It's much better to crash immediately when your assumptions are violated, than have to dig through the downstream code to figure out what went wrong and where.
- **Important:** Your code must be entirely of your own making. It must not be sourced from the net or the product of another party. You can use any code I give you (i.e. the template files), but everything else must come from you!

Some Tips and Organization

You may want to start by assuming that all symbol "codes" have a width of 16 bits (two 8 bit bytes). This implies that when you read a "code" you will be merely reading two bytes in succession, concatenating the second byte read to the first, to obtain one 16-bit "code". Similarly, when you write a "code" you will merely write two bytes, the most significant byte of the 16 bit "code" first, followed by the least significant byte second.

Fixing the size of a symbol "code" at 16 bits will make the implementation of the template's `BitStream` module trivial, however, the resulting data compression will not be very good. If you test your compression/decompression against my posted executable, its default values for `startingBits` and `maximumBits` are both 16. With these defaults, my executable should decompress your compressed data and vice-versa.

To calibrate you with respect to the size of the resulting program (library), a minimalist solution (no debug facilities or serious optimization) could take as little as 150 lines of code added to the template's $\tilde{1}00$ lines of .c files. Of course, a few lines must be added to the header files to provide data content to the struct definitions. With reasonable debug facilities and self-checking, the code you add to complete this assignment should be in the approximate range of 200 – 400 lines. This does not include your test bed or main procedure code, which will not be submitted.

Once you have the program working with default widths, you may then implement varying size symbol "codes". This will require a rethinking and rewriting the internals of your `BitStream` implementation. Also,your `lzwEncode` and `lzwDecode` procedures will now need to increase the size (width in bits) of new codes when necessary and stop increasing the size (and stop defining new codes) if the maximum width is attained.

Your `BitStream` implementation must continue to write out the most significant bytes of each "code" first, continuing through the least significant bytes. Reading "codes" will then, of course, involve reading bytes from the input where the most significant bits/bytes of each "code" will be read before the less significant bits/bytes. Note that there must be no extra bits or bytes in the compressed data; codes will be "back-to-back". The debug features available in my posted executable may be helpful in determining exactly what is being written and read. A good grasp of shifting and masking (using `&, |, ^, ~, <<, >>`) will be needed.

When testing against my executable, you will need to set command line arguments startingBits and maximumBits to the values you are using to obtain interoperable results.

Algorithms

Algorithm 1 LZW Encoding

Initialize dictionary D and insert characters 0-225 as single character sequences, each with their own code set to their respective character value.
$nextCode \leftarrow 256$
Create new Sequence W containing the first byte of data.
while there is still data to be read **do**
    $C \leftarrow$ Next byte of data from input
    Create new Sequence X using W appended with C
    if $X \in D$ **then**
        $W \leftarrow X$
    else
        find W in D
        Output code assigned to sequence W
        if more codes are permitted **then**
            Insert X into D, assigning $nextCode$ as its code
            $nextCode \leftarrow nextCode + 1$
        Create new sequence W with just character C
Find W in D and output its assigned code

Algorithm 2 LZW Decoding

Table T is a table of Sequences, indexed by an integer code. The table must be large enough to hold as many codes as are permitted by the maximum code bit width. Initialize table T with entries 0 through 255 with each holding a single character Sequence, one entry for each character 0 through 255, respectively.
$previousCode \leftarrow$ first code read from input
while there are more codes to be read **do**
    $currentCode \leftarrow$ next code from input
    if $currentCode \in T$ **then**
        $C \leftarrow$ first character of $T[currentCode]$
    else
        $C \leftarrow$ first character of $T[previousCode]$
    if T is not full **then**
        $W \leftarrow$ new sequence using $T[previousCode]$ appended with C
        add W at next index in T
    output Sequence $T[currentCode]$
    $previousCode \leftarrow currentCode$

What to turn in (in a zip on Blackboard):

- `.zip` of your entire directory need to compile the library. Do not include your testbed.
- **Note:** You should not need to provide a readme with the submission as your Makefile and interface should function seamlessly with mine.