Design for Assignment 8

Adrian Murillo

December 2019

1 Introduction

In this assignment we were tasked on creating a LWZ algorithm. For this we had to handle the decompression and compression aspects of the program and implement different ADTs for helping with the processes. Most of the code was given to us by Prof Darrel Long from CSE 13S of UCSC. In this file I will explain breifly how to implement the algorithms.

2 Trie ADT

The trie that we had to implement was purposed for handling the words that we read from the input txt file during the compression algorithm. It worked similarly to a binary tree ADT, whcih made it easy to program. Like a binary tree, the trie has a root node which spread out into multiple nodes, but in this case we set the number of children to 256, one for every ASCII character. The root node was initialized as well as all of its children, which hold the assigned ASCII code for the word located in that node.

The functions were pretty straight forward, so I won't go into detail in this paper. The following functions are included in Word.h:

```
TrieNode *trie_node_create(uint16_t code);

void trie_node_delete(TrieNode *n);

TrieNode *trie_create();

void trie_reset ( TrieNode * root ) ;

void trie_delete(TrieNode *n);

TrieNode *trie_step(TrieNode *n, uint8_t sym);
```

3 Word ADT

The word ADT was also pretty simple in its implementation. In this ADT implementation we also specified for a struct called WordTable, in which we can store the words. The purpose of the Word and WordTable objects was to implement during the decompression algorithm to hold all of the words that we found and place them in the word table in the position according to their ASCII code. In this part we used a WordTable instead of the trie since we didn't need to come up with a fancy data type to find the words that we already used since every word was assigned to an ASCII code starting at 256, which can be accessed easily within an array.

The word structs work as strings, but the only difference is that they are not NULL terminated, which allowed us to plug them into the word buffer and output them to the chosen file. How I would have coded this project if I were to design it from scratch, and not from the class's template, is by using normal char*, and just use strlen to calculate their length. To avoid the problem of the NULL termination, we just traverse the string until we reach strlen()-1. This would give us the possibility of using the append() and strlen() functions for the decompression algorithm.

The functions included in this file are:

```
Word *word_create(uint8_t *word, uint64_t word_len);

void word_delete(Word *w);

WordTable *wt_create();

void wt_reset(WordTable *wt);

void wt_delete(WordTable *wt);

void word_print(Word *w);

void wt_print(WordTable *wt);
```

4 I/O

This section from the code for me was the most challenging one, as we had to work with each byte as a bit vector, handle buffers, and fit all of its functions to the program that we were creating.

4.1 Code Buffer

For the code buffer we needed to use a bit of bitvector logic to introduce each set of bits into the bytes of the buffer. I decided to use the buffer as a static üint8_t array which allowed me to use it between calls without losing the values

within it. To maximize the use of my buffer I filled it up to the brim every time and then flushed it to the output file. This allowed for more straight forward calculations using bit_total.

To buffer the code I first see where the bits are going, then split them accordingly to the different bytes that they are going into. Using bitwise operations I set the individual bits as the input code, without losing any bits in the process. If the buffer is filled I just flush the buffer and use the second portion of the bits to fill the first byte, instead of the next one.

4.2 Word buffer

The word buffer was a bit simpler than the code buffer. I just had to one by one add the bytes to the buffer and when it got full I just flush the buffer and set the first place of the buffer to the next char.

4.3 Next code and Next char

For next char it was really simple, just access the byte of the char buffer that holds the current char and return it. For the next code I had to play a little with bitwise operations. Splitting the code into two was really helpful, since each code was located in the curr code position and the curr code+1 position. The value of bit_total%8 and bit_len allowed us to know the points were the split happened between two bytes. Using logical shifts using these values allowed us to "isolate" the codes from the code buffer and return only the bits that we need.

5 Compression and Decompression

This was the most complicated aspect of the assignment, but thankfully the Professor provided us with the pseudocode for designing this algorithms. The only challenge with this is turnng the almost two pages of pseudocode into c language and fix all of the memory and access issue for each part of the code.