Assignment 6 Design

Lempel-Ziv Compression

Arnav Nepal

March 13, 2023

1 Introduction

In this assignment we will be implementing the 1978 Lempel-Ziv data compression algorithm, otherwise known as LZ78. LZ78 uses a dictionary to map keywords to a "code", or index. Keywords are first input into the dictionary as single characters. When a repeat character in a text file is encountered, the dictionary stores is as a pair of the index of the repeated character and the new character. It repeats this, performing the same action any time it runs into a repeated character or sequence of characters. In order to make our program we will need to design and implements a host of abstract data types (ADT's), including a trie ADT and word table ADT. The files we will be implementing include:

- encode.c
- decode.c
- trie.c
- word.c
- io.c
- Makefile

2 Design and Psuedocode

This program requires us to implement 3 abstract data types: a trie module, a word table module, and an input/output module.

2.1 Tries: trie.c

A trie, a play on "re-*trie*-val", is a tree data structure. Also known as a prefix tree, each node in a trie contains a symbol and a pointer to n child nodes, where n is the number of letters in the alphabet being used. As we are using ASCII, $n = 256$. We will be using the trie data structure during compression to store words.

```
//reference for TrieNode struct
struct TrieNode {
    TrieNode *children[ALPHABET];
    uint16_t code;
};

//ALPHABET is defined as 256

TrieNode *trie_node_create(uint16_t index):
    allocate memory for trienode struct *node
    set node->code = index
    for i in range (ALPHABET):
        set node->children[i] to NULL
    return node

//destrctor for trie nodes. only a single free() required
//here since struct elements don't need memory here
void trie_node_delete(TrieNode *n);
    free(n)
    set n = NULL

//simple function, just needs to create a node
//and set code to EMPTY_CODE
TrieNode *trie_create(void);
    TrieNode *root = trie_node_create(EMPTY_CODE)
    if root != NULL:
        return root
    else:
        return NULL

//destructor for trie. this function first checks to see if a given node
//holds any child nodes. If it does, it recursively calls iteslf. If no
//child nodes are held, the end of a branch has been reached, and thus it
//is safe to start deleting nodes. This function can clear a trie or any
//subtrie within a trie
void trie_delete(TrieNode *n);
    for i in range(ALPHABET):
        if n->children[i] == NULL:
            continue
        else:
            trie_delete(n->children[i]) //recursive call
            n->children[i] = NULL
    trie_node_delete(n)

//simple function once trie_delete made. calls trie_delete
```

```
//on every child node of the root node
void trie_reset(TrieNode *root):
    for i in range(ALPHABET):
        if root->children[i] == NULL:
            continue
        else:
            trie_delete(root->children[i])
            set root->children[i] to NULL

TrieNode *trie_step(TrieNode *n, uint8_t sym);
    for i in range (ALPHABET):
        if struct pointed to in index i holds sym:
            return n->children[i]
    return NULL
```

Changes made: I did not need to make many changes to my final design of trie.c. I only made small edits like setting the pointers to null after freeing them for trie_reset and trie_delete, which I had forgotten

2.2 Words and WordTables: word.c

While compression can be accomplished using a trie data structure, decompression requires a look-up table for efficiency. Thus we will need to implement the Word Table ADT, which is a look-up table that contains a Word Table struct in each index. Each struct will contain words stored in a byte array (an array of uint8_t) and the length of that array.

```
//references for word struct
typedef struct Word {
    uint8_t *syms;
    uint32_t len;
} Word;

//definition of WordTable type
typedef Word * WordTable;

//constructor for individual word struct
//contains word in an array and length in
//uint32_t
Word *word_create(uint8_t *syms, uint32_t len):
    calloc memory of struct word using sizeof(word) as "word"
    set word->syms to point to "syms"
    set word->len to len
    if syms equals NULL:
        return NULL
    else:
        return arr

//in order to better understand this functions, let
//w->len equal 10 as an example:
Word *word_append_sym(Word *w, uint8_t sym):
    new string new_syms[w->len+2] //need to account for extra char + null char
    create new word "new_word" with word_create(new_syms, w->len+2):
    for i in range (w->len):
        //len 10, iterate over 0 - 9
        new_word->syms[i] = w->syms[i]
    new_word->syms[w->len] = sym //at index 10, sets new char
    new_word->syms[w->len + 1] = '\0' //null char
    set new_word->len = w->len + 1
    return new_word

//simple delete function, only requires one call to free
void word_delete(Word *w):
    free(w)
    set w = NULL

//A WordTable is an array of words. So we only need to call calloc
//for a WordTable * (since WordTable is typedef'd the expanded form
//would be Word **, which initializes an array of pointers, which
//we want).
WordTable *wt_create(void):
    calloc memory for an array of words of size (MAX_CODE) as "wt"
    create new empty word at wt[EMPTY_CODE]
    for i in range (START_CODE, MAX_CODE):
        set index in wt to null
    return wt

//same code as wt_delete, except it doesn't
//free the first index or the array itself
void wt_reset(WordTable *wt):
    for i in range (1, MAXCODE):
        if wt[i] is NOT null:
            free(wt[i])
            wt[i] = NULL

//deconstructor. Needs to free all
//memory within WordTable and then
//free the WordTable itself
void wt_delete(WordTable *wt):
    for I in range (MAX_CODE):
        if wt[i] is NOT null:
```

```
        free(wt[i])
        wt[i] = NULL
    free(wt)
    wt = NULL
```

Changes in final version: I did not need to make significant changes from my initial design in my final design of word.c. The biggest change I needed to make was setting each child of wt to null in wt_create(), which I had forgotten to do. I made a few edits in other functions, but nothing major

2.3 File I/O: io.c

In this program, we will also need to implement our own input/output (I/O) functions for parsing raw and compressed data for use with our compressor and decompressor. We will need to buffer I/O to minimize system calls, as system calls are very slow and inefficient. Our code also needs to be interoperable between different machines. This means we will have to keep endianness in mind.

Note: ALL BUFFFERS should be statically declared as their values will need to remain after the end of the functions

```
/*
//functions below are low confidence / very early draft designs. I will update
//these functions as I go and write, test my code
*/
//begin references for FileHeader struct
typedef struct FileHeader {
    uint32_t magic;
    uint16_t protection;
} FileHeader;

#define BLOCK 4096
#define MAGIC 0xBAADBAAC

extern uint64_t total_syms; // To count the symbols processed.
extern uint64_t total_bits; // To count the bits processed.
//end reference. Begin Design:

//reads data from a file or input source
int read_bytes(int infile, uint8_t *buf, int to_read):
    int r
    int sum //holds total number of bytes actually read over function life
    while loop:
        r = read(infile, buf, to_read)
        sum += r
        if (sum of r == to_read): //sum == to_read when all bytes read
```

```
            break

int write_bytes(int outfile, uint8_t *buf, int to_write):
    int w
    int sum
    while loop:
        w = write(outfile, buf, and to_write)
        sum += w
        if (sum of w == to_write): //sum == to_write when all bytes written
            break

//function to read header. calloc allocates space for uint32_t
//and uint16_t, so as long as the file header contains those in
//a row read will copy it correctly into header
void read_header(int infile, FileHeader *header):
    read_bytes(infile, header, sizeof(FileHeader))
    if header is big endian:
            swap endianness


void write_header(int outfile, FileHeader *header):
    if header is big endian:
        swap endianness
    write_bytes(outfile, header, sizeof(FileHeader))

bool read_sym(int infile, uint8_t *sym):
    static int r
    static int index
    static array buffer

    index = index % BLOCK //ensures index never exceeds BLOCK
    if (byte_index at 0){
        new buffer needed, read BLOCK bytes into buffer
    }
    if index < r: //ensures index does not exceed bytes read
        sym = buffer[index]
        index ++
        return true;
    else:
        return false

void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen):
    static int bit_index
    static int byte_index = bit_index/8
    if byte_index is equal to BLOCK:
        reset all index values and flush_pairs
    for i in range bitlen;
        set bit_index in write_buffer[byte_index] to bit i in code, starting from
            LSB
        increment bit_index
        byte_index = bit_index/8 //increments byte when bit reaches multiples of 8
        if byte_index is equal to BLOCK:
            reset all index values and flush_pairs

    for i in range 8:
        set bit_index in write_buffer[byte_index] to bit i in sym, starting from
            LSB
        increment bit_index
        byte_index = bit_index/8 //increments byte when bit reaches multiples of 8
        if byte_index is equal to BLOCK:
            reset all index values and flush_pairs
    update total_bits to value of bit_index


void flush_pairs(int outfile):
    write ⌈total_bits/8⌉ to outfile from write_buffer

bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen):
    static int bit_index
    static int byte_index = bit_index/8

    //we need to reset code and sym so they don't contain previous garbage
    *code = 0
    *sym = 0
    if byte_index is equal to BLOCK:
        reset all index values and read another BLOCK into read_buffer
    for i in range bitlen;
        set bit_index in code to bit i in read_buffer[byte_index], starting from
            LSB
        increment bit_index
        byte_index = bit_index/8 //increments byte when bit reaches multiples of 8
        if byte_index is equal to BLOCK:
            reset all index values and read another block

    for i in range 8:
        set bit_index in sym to bit i in read_buffer[byte_index], starting from LSB
        increment bit_index
        byte_index = bit_index/8 //increments byte when bit reaches multiples of 8
        if byte_index is equal to BLOCK:
            reset all index values and read_another block
```

```
//writes each symbol in w to outfile
void write_word(int outfile, Word *w):
    static int byte_index
    for i in range (w->len):
        word_buffer[byte_index] = w->syms[i]
        byte_index++
        total_syms++
        if byte_index is equal to BLOCK:
            flush words and reset byte_index to 0



void flush_words(int outfile):
    write ⌈total_syms/8⌉ to outfile from word_buffer
```

Changes made in final version: The bulk of the changes in my final design from my initial were in io.c. I overly simplified some functions in here while overly complicating others. First, I changed the header functions so they used the already made read_bytes() and write_bytes() functions instead of read() and write(), since it was redundant. On the other hand, my design of read_sym() was overly simplistic. I only included a single call to read to read 1 byte into the sym variable. This was not good since we need to maintain a buffer. Therefore, I updated the design for read_sym() to include a buffer. I also updated flush words once I realized we had to use global buffers, which I was not aware of at the time of the initial draft.

My most severe changes were in read_pair() and write_pair(), where my initial design was wildly off. I had neglected to use a buffer, and made them just make individual read and write symbols. I updated both functions so that they used a buffer, and corrected the process of copying and writing the code and sym, since we need to process both reading and writing bit by bit. I also updated the desing of write_word, which was similarly simplistic (and neglected to really utilize a buffer).