Assignment 6 Design

Lempel-Ziv Compression

Arnav Nepal

March 3, 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
    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])

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
```

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 "arr"
    set struct_word->syms to point to "syms"
    if syms OR arr equal 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, sym):
    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

//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 "arr"
    set length of arr[EMPTY_CODE] to zero
    return &arr

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

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
    while loop:
        r = read(infile, buf, to_read)
        if (r == 0 OR r == to_read): //read returns 0 after reaching EOF
            break

int write_bytes(int outfile, uint8_t *buf, int to_write):
    int w
    while loop:
        w = write(outfile, buf, and to_write)
        if (w == 0 OR w == to_write): //write returns 0 after end of data
            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):
    int r = 1
    while r is not zero: //runs until r equals 0
        r = read(infile, header, sizeof(FileHeader))
        if header is big endian:
            swap endianness
```

```
void write_header(int outfile, FileHeader *header):
    int w = 1
    if header is big endian:
        swap endianness
    while w is not zero:
        w = write(outfile, header, sizeof(FileHeader))

bool read_sym(int infile, uint8_t *sym):
    if (read_bytes(infile, sym, sizeof(uint8_T))):
        total_syms++
        return true
    else:
        return false

void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen):
    calloc memory for a array of uint16_t as "arr"
    int t
    while:
        for i in range(bitlen) with step 2:
            t = i + 1
            arr[0] = code
            arr[1] = sym
        write(outfile, arr, bitlen)


void flush_pairs(int outfile):
    //how does this flush without accepting arguments?
    //I am not able to design this and flush_words() right now.
    //I will need to play around with the read() and write()
    //functions properly first before I am able to do so

bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen):
    if (!read(infile, code, sizeof(uint16_t)):
        return false
    if (!read(infile, sym, sizeof(uint8_t)):
        return false
    return true

void write_word(int outfile, Word *w):
    buffer[BLOCK]
    for i in range (BLOCK):
        buffer[i] = w->syms[i]
    write(outfile, buffer, BLOCK)
```

```
void flush_words(int outfile):
    //how does this flush without accepting arguments?
```
