Assignment 6 DESIGN.pdf

Julian Chop

March 14, 2023

1 Brief Description of Assignment

For this assignment, we will be implementing two programs, encode and decode, that perform LZ78 compression and decompression.

- encode: compresses any file, text, or binary.
- decode: decompresses any file, text, or binary, that was compressed with encode.

We will also need to implement 2 new ADTs: one fore tries, and another for words. These will be placed in their respective .c files: trie.c and word.c. All of these files will be further explained below.

2 trie.c

TrieNode

TrieNode *trie_node_create(uint16_t code)

This constructor function will create a Trie node. The node's code will be set to code. Each of the children node pointers are NULL using malloc.

```
create TrieNode and allocate memory for children;
set TrieNode code to code;
return created TrieNode;
```

void trie_node_delete(TrieNode *n)

This deconstructor function will take in a TrieNode pointer and free the memory that it's pointing at.

```
free memory of node n;
```

TrieNode *trie_create(void)

This function initializes a trie: a root TrieNode with the code, EMPTY_CODE. It will return a TrieNode pointer, ei the root, if successful. Will return NULL otherwise.

```
create TrieNode and allocate memory for children;
set TrieNode code to EMPTY_CODE;

for sym in ALPHABET:
    set TrieNode children [i] to NULL;

return created TrieNode;
```

void trie_reset(TrieNode *root)

This function will reset a trie to just the root TrieNode. For this function, I can simply call trie_delete which basically does the same thing except it deletes from the given pointer. So for all the children of root, call trie_delete on it.

```
for sym in ALPHABET:
    if root of children[sym] != NULL:
        trie_reset(root children[sym]);
        root children[sym] = NULL;
```

void trie_delete(TrieNode *n)

This function will delete a sub-trie starting from node n.

```
for sym in ALPHABET:
    if n children[sym] != NULL:
        trie_delete(n children[sym]);
        n children[sym] = NULL;

trie_node_delete(n);
```

TrieNode *trie_step(TrieNode *n, uint8_t sym)

This function returns a pointer to the child node representing the symbol sym. If the symbol does not exist, return NULL.

```
return n children[sym];
```

3 word.c

Word *word_create(uint8_t *syms, uint32_t len)

This constructor function for a word makes a word where sysms is the array of symbols Word represents and len if the length of the array. Returns a Word pointer if successful or NULL otherwise.

```
allocate memory for word pointer;
if len == 0:
    word syms = NULL;
    word len = len;
    return created word

allocate memory for syms;

//copy syms arry to word syms
for i in len:
    word syms[i] = syms[i];
word len = len;

return word;
```

Word *word_append_sym(Word *w, uint8_t sym)

Appends sym to the specified Word w.

```
allocate memory for new word pointer

//if new w is empty
if w len == 0:
    allocate memory for new word syms
    set new word[syms] = sym;
    set new word len = 1;
    return new word;

allocating memory for new word with size len + 1;
copy over over w to new word;
append symbol to new word;
return new word;
```

void word_delete(Word *w)

This deconstructor function simply removes Word *w.

```
free w syms with free();
free the word w with free();
```

WordTable *wt_create(void)

This function creates a new WordTable, which is an array of Words.The sie of WordTable is MAX_CODE. A WordTable is initialized with a single Word at index EMPTY_CODE.

```
allocate memory for *wt with calloc with length MAX_CODE;
set wt[EMPTY_CODE] = word_create(NULL, 0);
return new table;
```

void wt_reset(WordTable *wt)

This function resets the given WordTable, wt, to contain just the empty Word.

```
iterate over wt starting at 2, ending at MAX_CODE with i:
    if wt[i] != NULL:
        word_delete(wt[i]);
        wt[i] = NULL;
```

void wt_delete(WorldTable wt)

This function deletes all words and tables and frees up associated memory.

```
for i in MAX_CODE:
    if(wt[i] != NULL):
        word_delete(wt[i]);
        wt[i] = NULL;
free memory of wt;
```

4 io.c

int read_bytes(int infile, uint8_t *buf, int to_read)

This function will help perform reads. This will be called whenever we need to perform a read. Returns the number of bytes read.

```
bytes_read = 0
curr_read = amount of bytes returned from read()
while to_read > 0 AND still stuff to read:
    bytes_read += amount read from infile;
    to_read -= amount read from infile;
    curr_read = amount of bytes return from read();
return bytes_read;
```

int write_bytes(int outfile, uint8_t *buf, int to_write)

This function is basically the same as read_bytes but with write(). Will continue to write until specified bytes or when there is nothing left to be written. Number of bytes written is returned.

```
bytes_written = 0;
curr_written = number of bytes written to outfile
while to_write > 0 AND still stuff to write:
    bytes_read += amount written;
    to_write -= amount written;
    curr_written = number of bytes written to outfile;
return bytes_written;
```

void read_header(int infile, FileHeader *header)

This function reads in sizeof(FileHeader) bytes from the input file. These bytes are then read in the supplied header. This function must also verify the magic number.

```
read file header from infie to header pointer

if in big endian:
    swap bytes in header->magic
    swap bytes in header->protection
check it header->magic = MAGIC
```

void write_header(int outfile, FileHeader *header)

```
if in big endian:
    swap bytes in header->magic;
    swap bytes in header->protection;
write header to outfile with write_bytes;
```

bool read_sym(int infile, uint8_t *sym)

This function writes sizeof(FileHeader) bytes to the output file.

```
make counter for buffer;

if counter = 0:
    try to fill buffer with read_bytes;
    if read bytes returns 0:
        return false;
if counter is equal to BLOCK:
    try to fill buffer with read_bytes;
    if read bytes returns 0:
        return false;
    counter = 0;
set sym pointer = word_buffer[counter];
counter += 1;
```

void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)

This function writes pairs to outfile which is comprised of a code and a symbol. It will use a global variable, bit_index to keep track of bits read as well as a global buffer pair_buffer.

```
writing code to outfile
for i in bitlen:
    grab the ith bit from code;
    write bit into the pair_buffer;
    bit_index += 1;
    if buffer is full:
        run flush_pairs to write pair_buffer to outfile;
        reset bit_index to 0;

//writing sym to outfile
for i in 8:
    grab the ith bit from sym;
    write bit into the pair_buffer;
    bit_index += 1;

    if buffer is full:
        run flush_pairs to write pair_buffer to outfile;
        reset bit_index to 0;
```

void flush_pairs(int outfile)

This function writes out any remaining pairs of symbols and codes to the output file. It will use the global variable bit_index and pair_buffer.

```
write bytes from pair_buffer to outfile using write_bytes;
zero out entire buffer with memset();
```

bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen)

This function reads pairs from the input file. Uses global bit_index and pair_buffer.

```
read bytes from infile into pair_buffer if bit_index = 0;
if read_btyes returns 0:
    return false:

//read code bits to code
for i in bitlen:
    if bit_index == BLOCK;
        read more bytes from infile with read_bytes;
        if read_bytes returns 0:
            return false;
        set bit_index to 0;
    grab the ith bit from buffer and write it to code
    bit_index += 1;

//read sym bits to sym
for i in 8:
    if bit_index == BLOCK;
        read more bytes from infile with read_bytes;
        if read_bytes returns 0:
            return false;
        set bit_index to 0;
    grab the ith bit from buffer and write it to sym
    bit_index += 1;

if code is equal to STOP_CODE:
    return false;

return true;
```

void write_word(int outfile, Word *w)

This function every symbol from w into outfile. Uses the global buffer word_buffer.

```
make a static variable index_counter;
for i in w length:
    word_buffer[index_counter] = w syms[i];
    index_counter += 1;

    //check if the buffer is full
    if index_counter is equal to BLOCK:
        write buffer to outfile with flush_words;
        set index_counter to 0;
```

void flush_words(int outfile)

This function writes out any remaining symbols in the buffer to out file. Uses the global buffer, word_buffer.

```
use write_bytes to write everything from pair_buffer to outfile;
```

5 compression.c

This file will contain the main() for the compression program as well as the command-line options. It will compress given infile and place the output into outfile.

```
open infile with open();
open outfile with open();
write out file heaser to outfile using write_header();
create trie with trie_create() setting root node to EMPTY_CODE;
for symbols in infile:
    curr_sym = read_sym(symbol);
    next_code = trie_step(curr_node, curr_sym);
    if next_node != NULL:
        prev_node = curr_node;
        curr_node = next_node;
    else:
        write_pair curr_node ->code, curr_sym
    if next_node == MAX_CODE:
        trie_reset();
    prev_sym = curr_sym;
write pair (STOP_CODE, 0);
flush_pairs();
close infile and outfile with close();
```

6 decompression.c

This file will contain the main() for the decompression program as well as the command-line options. It will decompress given infile compressed by compression and out into outfile.

```
    open infile with open();
    read file header with read_header() to verify magic number;
    open outfile with open();
    table = wt_create(); //creating new word table
    for pairs in infile:
        curr_code, curr_sym = read_pair();
        table[curr_code] = word denoted by curr_code
        next_code += 1
        if next_code == MAX_CODE:
            wt_reset();
            next_code = START_CODE;
    flush_words();
    close infile and outfile with close();
```