Acer Cristea
Cruzid: acristea
Winter 2023 - CSE 13S
Dr. Veenstra

Assignment 6 DESIGN

Description of Program

In this assignment, we will be creating two programs. The first is encode which can perform LZ78 compression and can compress any file, test, or binary. The second program is decode, which can perform LZ78 decompression and can decompress any file, text or binary that was compressed with encode. Both of these programs operate on little and big-endian systems where interoperability is required. Both programs use variable bit-length codes and perform reads and write in efficient blocks of 4KB.

NOTE:
You will need to implement some new ADTs for this assignment: an ADT for tries and an ADT for words. In addition to these new ADTS, you will need to be concerned about variable-length codes, I/O, and endianness.

Encode - Pseudocode is given

Your encode program should accept the following command-line options:

- -v: Print the compression statistics to stderr.
    - In the format:
        - Compressed file size: X bytes
        - Uncompressed file size: X bytes
        - Space saving: XX.XX%
- -i: specifies the output file to encrypt (default: stdout).
- -o: specifies the file containing the public key (default: ss.pub).

The program should follow these steps:

1. Open infile with open(). If an error occurs, print a helpful message and exit with a status code indicating that an error occurred.

2. The first thing in outfile must be the file header, as defined in the file io.h. The magic number in the header must be 0xBAADBAAC. The file size and the protection bit mask you will obtain using fstat().

3. Open outfile using open(). The permissions for outfile should match the protection bits as set in your file header. Any errors with opening outfile should be handled like with infile.

4. Write the filled out file header to outfile using write_header(). This means writing out the struct itself to the file, as described in the comment block of the function.

5. Create a trie. The trie initially has no children and consists solely of the root. The code stored by this root trie node should be EMPTY_CODE to denote the empty word. You will need to make a copy of the root node and use the copy to step through the trie to check for existing prefixes. This root node copy will be referred to as curr_node.

6. You will need a monotonic counter to keep track of the next available code. This counter should start at START_CODE, as defined in the supplied code.h file. The counter should be a uint16_t since the codes used are unsigned 16-bit integers. This will be referred to as next_code.

7. You will also need two variables to keep track of the previous trie node and previously read symbol. We will refer to these as prev_node and prev_sym, respectively.

8. Use read_sym() in a loop to read in all the symbols from infile. Your loop should break when read_sym() returns false. For each symbol read in, call it curr_sym, perform the following:
    a. Set next_node to be trie_step(curr_node, curr_sym), stepping down from the current node to the currently read symbol.
    b. If next_node is not NULL, that means we have seen the current prefix. Set prev_node to be curr_node and then curr_node to be next_node.
    c. Else, since next_node is NULL, write the pair (curr_node->code, curr_sym), where the bit-length of the written code is the bit-length of next_code. We now add the current prefix to the trie. Let curr_node->children[curr_sym] be a new trie node whose code is next_code. Reset curr_node to point at the root of the trie and increment the value of next_code.
    d. Check if next_code is equal to MAX_CODE. If it is, use trie_reset() to reset the trie to just having the root node. This reset is necessary since we have a finite number of codes.
    e. Update prev_sym to be curr_sym.

9. After processing all the characters in infile, check if curr_node points to the root trie node. If it does not, it means we were still matching a prefix. Write the pair (prev_node->code, prev_sym). The bit-length of the code written should be the bit-length

of next_code. Make sure to increment next_code and that it stays within the limit of MAX_CODE. Hint: use the modulo operator.

10. Write the pair (STOP_CODE, 0) to signal the end of compressed output. Again, the bit-length of code written should be the bit-length of next_code.

11. Make sure to use flush_pairs() to flush any unwritten, buffered pairs. Remember, calls to write_pair() end up buffering them under the hood. So, we have to remember to flush the contents of our buffer. 12. Use close() to close infile and outfile.

Decode - Pseudocode is given

Your decryptor program should accept the following command-line options:

- -v: Print decompression statistics to stderr
    - In the format:
        - Compressed file size: X bytes
        - Uncompressed file size:
        - X bytes Space saving: XX.XX%
- -i <input>: Specify input to decompress (stdin by default)
- -o <output>: Specify output of decompressed input (stdout by default)

The program should follow these steps:

1. Open infile with open(). If an error occurs, print a helpful message and exit with a status code indicating that an error occurred. infile should be stdin if an input file wasn't specified.

2. Read in the file header with read_header(), which also verifies the magic number. If the magic number is verified then decompression is good to go and you now have a header which contains the original protection bit mask.

3. Open outfile using open(). The permissions for outfile should match the protection bits as set in your file header that you just read. Any errors with opening outfile should be handled like with infile. outfile should be stdout if an output file wasn't specified.

4. Create a new word table with wt_create() and make sure each of its entries are set to NULL. Initialize the table to have just the empty word, a word of length 0, at the index EMPTY_CODE. We will refer to this table as table.

5. You will need two uint16_t to keep track of the current code and next code. These will be referred to as curr_code and next_code, respectively. next_code should be initialized as START_CODE and functions exactly the same as the monotonic counter used during compression, which was also called next_code.

6. Use read_pair() in a loop to read all the pairs from infile. We will refer to the code and symbol from each read pair as curr_code and curr_sym, respectively. The bit-length of the code to read is the bit-length of next_code. The loop breaks when the code read is STOP_CODE. For each read pair, perform the following:
    a. As seen in the decompression example, we will need to append the read symbol with the word denoted by the read code and add the result to table at the index next_code. The word denoted by the read code is stored in table[curr_code]. We will append table[curr_code] and curr_sym using word_append_sym().
    b. Write the word that we just constructed and added to the table with write_word(). This word should have been stored in table[next_code].
    c. Increment next_code and check if it equals MAX_CODE. If it has, reset the table using wt_reset() and set next_code to be START_CODE. This mimics the resetting of the trie during compression.

7. Flush any buffered words using flush_words(). Like with write_pair(), write_word() buffers words under the hood, so we have to remember to flush the contents of our buffer.

8. Close infile and outfile with close()

The Deliverables
(files that need to be included in the directory “asgn6”)

1. encode.c contains the implementation and main() function for the encode program.

2. decode.c contains the implementation and main() function for the decode program.

3. trie.c contains the source file for the Trie ADT.

4. trie.h contains the header file for trie.c, CANNOT be modified.

5. word.c contains the source file for the Word ADT.

6. word.h contains the header file for word.c, CANNOT be modified.

7. io.c contains the source file for the I/O module.

8. io.h contains the header file for the I/O module, CANNOT be modified.

9. endian.h contains the header file for the endianness module, CANNOT be modified.

10. code.h contains the header file containing macros for reserved codes, CANNOT be modified.

11. Makefile:
    - CC = clang must be specified.
    - CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified.
    - make must build the encode and decode as should make all.
    - make encode should build only the encode program
    - make decode should build only the decode program
    - make clean must remove all files that are compiler generated.
    - Programs should have no memory leaks

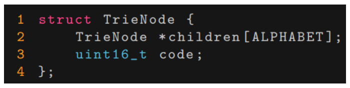
12. README.md: This must use proper Markdown syntax. It must describe how to use your program and Makefile. It should also list and explain any command-line options that your program accepts. Any false positives reported by scan-build should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.

13. DESIGN.pdf: What I am writing right now.

14. WRITEUP.pdf: This document must be a proper PDF. This writeup document must include everything you learned from this assignment. Make sure to mention everything in detail while being as precise as possible. How well you explain all the lessons you have learned in this assignment will be really important here.

Pseudocode / Structure

*Code will be spread out across multiple files

Step 1: Building the trie tree  `struct TrieNode {` `TrieNode *children[ALPHABET];` `uint16_t code;` `};`	The TrieNode struct will have the two fields shown above. Each trie node has an array of 256 pointers to trie nodes as children, one for each ASCII character.

The code field stores the 16-bit code for the word that ends with the trie node containing the code.

TrieNode *trie_node_create(uint16_t code)

- Constructor for a TrieNode. Each of the children's pointers are NULL

void trie_node_delete(TrieNode *n)

- Destructor for a TrieNode, takes one pointer

TrieNode *trie_create(void)

- Initalizes a trie: a root TrieNode with the code EMPTY_CODE. Returns the root, a TrieNode *, if successful, NULL otherwise

void trie_reset(TrieNode *root)

- Resets a trie to just the root TrieNode. There are fnite codes so eventually we get to MAX_CODE. Reset by deleting the children, and making them NULL

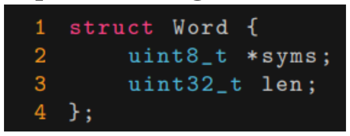
void trie_delete (TrieNode *n)

- Deletes a sub-trie starting from the trie rooted at node n. This will require recursive calls on each of n's children.
- Use trie_node_delete(), then set children to NULL

TrieNode *trie_step(TrieNode *n, uint8_t sym)

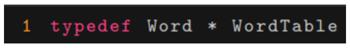
- Returns a pointer to the child node representing the symbol sym. If it doesn't exist, NULL is returned

Step 2: Building Word



```
struct Word {
    uint8_t *syms;
    uint32_t len;
};
```

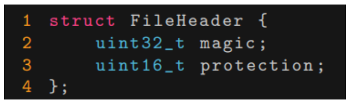
A Word holds an array of symbols, syms, stored as bytes in an array. The length of the array storing the symbols a Word represents is stored in len.



```
typedef Word * WordTable
```

To make things easier to reason about, we can define an array of Words as a WordTable.

	Word *word_create(uint8_t *syms, uint32_t len) - Constructor for a word where sysms is the array of symbols a Word represents. The length of the array of symbols is given by len. This function returns a Word * if successful or NULL otherwise. Word *word_append_sym(Word *w, uint8_t sym) - Constructs a new Word from the specified Word, w, appended with a symbol, sym. The Word specified to append tomay be empty. If that's the case Word should only contain the symbol. Returns the new Word, result of appending. void word_delete(Word *w) - Destructer for Word w. Single pointer is used WordTable *wt_create(void) - Creates a new WorldTable, an array of words. It has a pre-defined size MAX_CODE with the value UINT16_MAX. Codes are 16-bit integers. - Initialized with a single Word at index EMPTY_CODE. This Word represents the empty word, a string of length 0. void wt_reset(WorldTable *wt) - Resets a WorldTable, wt, to contain just the empty Word. All other words in table are NULL

Step 3: I/O
 	

`struct FileHeader {`
`    uint32_t magic;`
`    uint16_t protection;`
`};`
	- Calculate the minimum number of bits needed to represent any integer x ≥ 1 using the formula (floor of log_2 (x)) + 1. The edge case is 0, whose bit length is 1.
	- Note that only codes have variable bit-lengths: symbols in a pair are always 8 bits
	- Struct definition for the file header, which contains the magic number for the program and the protection bit mask of the original file. Magic is a unique identifier for encode. Decode should only

work if it has the correct magic number. (0xBAADBAAC)

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

- Loop that calls read until all bytes specified were read or there are no more bytes. Number of bytes read is returned.

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

- Loop that calls write until all bytes specified were written or until no bytes left. Returns number of bytes written

void read_header(int infile, FileHeader *header)

- Reads in sizeof (FileHeader) bytes from the input file. These bytes are read into the supplied header. Endianness is swapped if byte order isn't little endian. Along with reading the header, it must verify the magic number.

void write_header(int outfile, FileHeader *header)

- Writes sizeof(FileHeader) bytes to output file. Bytes are from the supplied header. Endianness is swapped if byte order isn't little endian.

bool read_sym(int infile, uint8_t *sym)

- Index keeps track of the currently read symbol in the buffer. Once all symbols are processed , another block is read. If less than a block is read, the end of the buffer is updated. Returns true if there are symbols to be read, false otherwise.

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

- Writes a pair to outfile, comprised of a code and a symbol. Bits of the code are buffered first, starting from LSB. Bits of the symbol buffered next, starting from LSB. Code buffered has a bit-len of bitlen. Buffer is written out whenever filles.

	void flush_pairs(int outfile) - Writes out any remaining pairs of symbols and codes to output file Bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen) - Reads a pair from the input file. The read code placed into pointer to code. Read sym to pointer sym. Once all the bits have been processed, another block is read. First bits are the code starting from LSB and the last 8 are the symbol, starting from LSB. - Returns true if there are pairs left to read in the buffer, else false. void write_word(int outfile, Word *w) - Writes a pair to output file void flush_words(int outfile) - Writes out any remaining symbols in the buffer to the outfile. - make use of fstat() and fchmod(). - All reads and writes in this program must be done using the system calls read() and write(), which means that you must use the system calls open() and close() to get your file descriptors. You will want to use two static 4KB uint8_t arrays to serve as buffers: one to store binary pairs and the other to store characters. Each of these buffers should have an index, or a variable, to keep track of the current byte or bit that has been processed.
Step 4: Creating the Makefile	Since no Makefile is provided, we need to make it ourselves to be able to compile and run our files. 1. Make sure to use CFLAG and CLANG