Mylo Lynch

Darrel Long

Cse 13s

3/1/23

Lz78

Deliverables:

encode.c: contains the main() function of the encode program

-v:Print Compression Statistics To Stderr.

-i <input>:Specify input to compress (stdin by default)

-o <output>:Specify Output Of Compressed Input(stdout by default)

Pseudo code analysis:

COMPRESS(*infile*, *outfile*)

1 *root* = TRIE_CREATE() // create the tree
2 *curr_node* = *root* //*set current node to root node*
3 *prev_node* = NULL //no previous node since its the beginning of the code
4 *curr_sym* = 0 //current symbol is empty
5 *prev_sym* = 0 //no previous symbol
6 *next_code* = START_CODE //start code is the first useable code (empty_code is first)
7 **while** READ_SYM(*infile*, &*curr_sym*) **is** TRUE //condition, while loop is happening while being read
8 *next_node* = TRIE_STEP(*curr_node*, *curr_sym*) //step through the trie to encode the next word

```
9       if next_node is not NULL  //if not null then update prev and current node to the current node and next node respectively
10              prev_node = curr_node
11              curr_node = next_node
12      else // word is complete write pair and create new trie node, current is set to root and next_code +=1
13              WRITE_PAIR(outfile, curr_node.code, curr_sym, BIT-LENGTH(next_code))
14              curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
15              curr_node = root
16              next_code = next_code + 1
17      if next_code is MAX_CODE //if max is reached reset the tree from root and make next the start code
18              TRIE_RESET(root)
19              curr_node = root
20              next_code = START_CODE
21      prev_sym = curr_sym// within while loop after conditional statements, always set the previous symbol to the current symbol before the next iteration
22  if curr_node is not root
23      WRITE_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code)) //encode pair
24      next_code = (next_code+1)%MAX_CODE //returns value between 0 and max code
25  WRITE_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code)) //write and flush
26  FLUSH_PAIRS(outfile)
```

decode.c:contains the main() function for the decode program.

-v:Print Decompression Statistics To Stderr.

-i <input>:Specify Input To Decompress(stdin by default)

-o <output>:Specify Output Of Decompressed Input(stdout by default)

Pseudocode analysis:

DECOMPRESS(*infile*, *outfile*)

table = WT_CREATE() //create word table

curr_sym = 0 //0 = empty code

curr_code = 0 //start at empty code

next_code = START_CODE //next code = 1 = start code = first letter/word

while READ_PAIR(*infile*, &*curr_code*, &*curr_sym*, BIT-LENGTH(*next_code*)) **is** TRUE

//while readable

//use next code as index and append the current code/symbol

table[*next_code*] = WORD_APPEND_SYM(*table*[*curr_code*], *curr_sym*)
WRITE_WORD(*outfile*, *table*[*next_code*]) //write decoded word

next_code = *next_code* + 1 //next code +=1 to iterate through the encoded values

if *next_code* **is** MAX_CODE //if at the end reset

WT_RESET(*table*)

next_code = START_CODE

FLUSH_WORDS(*outfile*) //clear words

trie.c: the source file for the Trie ADT.

Create trie node struct with fields for trienode children and for the code for an ASCII character

TrieNode *trie_node_create(uint16_t code)

Creates a trie node, the node's code is whatever is in the parameter, children node pointers should be NULL

void trie_node_delete(TrieNode *n)

Deletes a trie node

TrieNode *trie_create(void)

Creates trie, returns pointer for the root if successful and NULL if not

void trie_reset(TrieNode *root)

Reset when full

void trie_delete(TrieNode *n)

Deletes subtrie that starts from parameter root node n

TrieNode *trie_step(TrieNode *n, uint8_t sym)

Will be called repeatedly to check if a word exists within the array of nodes

trie.h: the header file for the Trie ADT. **DO NOT EDIT**

word.c:the source file for the Word ADT.

WORD STRUCT AND DEFINE ARRAY OF WORDS AS WORDTABLE

Word *word_create(uint8_t *syms, uint32_t len)

Makes word. Parameter syms is an array of symbols the word represents. Length of array is len.

Return NULL if unsuccessful

Word *word_append_sym(Word *w, uint8_t sym)

Appends a symbol to an existing word if word exists to make new word, if empty word is only the symbol to be appended. Return the new word with the symbol appended

void word_delete(Word *w)

Deletes word (free memory clear variables etc.)

WordTable *wt_create(void)

Creates array of words of size MAX_CODE (UINT16_MAX) with empty word initialized (string of length 0. When encoding, pairs with code 0 will only represent the symbol it is paired with)

void wt_reset(WordTable *wt)

Resets word table to only having empty word

word.h:the header file for the Word ADT. **DO NOT EDIT.**

io.c:the source file for the I/O module.

In and out operations. Define file header struct with a magic number and protection bit mask. Swap endianness as needed for interoperability

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

Wrapper loops calls to read() until all specified (or all) bytes

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

Like read but looping calls to write

void read_header(int infile, FileHeader *header)

reads in sizeof(FileHeader) bytes from the input file

void write_header(int outfile, FileHeader *header)

Writes sizeof(FileHeader) bytes to the output file

bool read_sym(int infile, uint8_t *sym)

Reads symbol and uses index to keep track of where its at in the buffer returns true if there are symbols to be read

If less than a block is read end of buffer is updated

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

Writes code symbol pair to outfile buffered from least significant bit

void flush_pairs(int outfile)

Writes out any remaining pairs to outfile

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

Returns true if there are pairs left to read (i.e. if its not stop code) in the buffer, else false.

void write_word(int outfile, Word *w)

Places word in buffer, buffer is written when full

void flush_words(int outfile)

Writes out any remaining symbols in the buffer to the outfile.

Io.h:the header file for the I/O module. DO NOT EDIT.

endian.h: the header file for the endianness module. DO NOT EDIT.

code.h:theheaderfilecontainingmacrosforreservedcodes.DO NOT EDIT.

Makefile

CC = clang
CFLAGS = -Wall -Wextra -Werror -Wpedantic **$(**shell **pkg-config --cflags gmp)**
LFLAGS = **$(**shell **pkg-config)**
TARGETS =
OBJS =

all:

```
TARGETS: encode.o decode.o io.o code.o
	$(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)

%.o: %.c
	$(CC) $(CFLAGS) -c $<

clean:
	rm -f $(TARGETS) $(OBJS)

format:
	clang-format -i -style=file *.[ch]
```