Marian Zlateva
mzlateva@ucsc.edu
3/14/2021

CSE13s Fall 2020
Assignment 7: Lempel-Ziv Compression

Description:

In this lab, I wrote up a program that compresses files and another program to work alongside it to decompress those files

Flags for both encoder.c and decoder.c:

-v	Print decompression statistics to stderr
-i:	input file
-o:	output file
-h	print help information

Files:

- encode.c :
  - contains the main() function for the encode program.
- decode.c :
  - contains the main() function for the decode program.
- trie.c :
  - the source file for the Trie ADT.
- trie.h :
  - the header file for the Trie ADT.
- word.c :
  - the source file for the Word ADT.
- word.h :
  - the header file for the Word ADT.
- io.c :
  - the source file for the I/O module.
- io.h :
  - the header file for the I/O module.
- endian.h :
  - the header file for the endianness module.
- code.h :
  - the header file containing macros for reserved codes.
- bit_tools.c
  - the source file for many bit manipulation functions.
- bit_tools.h
  - the header file for the bit tools module.

Functions:

encode.c

int main()

- ➢ does LZ78 Compression

- ➔ parses the command line options as described as the start of the file
- ➔ checks for file errors
- ➔ prints help statistics if -h flag or invalid flag is passed and exits with error code 1
- ➔ make output file permission same as input file permission
  - ➞ get permission status of infile
  - ➞ set permission status of outfile to infile's permission status
- ➔ writes the header to the compressed file with magic set to MAGIC (from io.h) and protection set to statbuf.st_mode using fchmod (make sure to call fix_endianess() to fix endianness if needed)
- ➔ writes the header to the compressed file
- ➔ follows the LZ78 compression pseudocode presented in the assignment document and below
- ➔ prints the statistics if needed
  - ➞ compressed file size
  - ➞ uncompressed file size
  - ➞ space saving -> 100 * (1 - (comp_fs/ uncomp_fs));
- ➔ closes the files and exits

Given LZ78 Compression Algorithm Pseudocode

```
root = TRIE _ CREATE ()
curr_node = root
prev_node = NULL
curr_sym = 0
prev_sym = 0
next_code = START _ CODE
while READ _ SYM (infile, &curr_sym) is TRUE
        next_node = TRIE _ STEP (curr_node, curr_sym)
        if next_node is not NULL
                prev_node = curr_node
                curr_node = next_node
        else
                WRITE _ PAIR (outfile, curr_node. code, curr_sym, BIT -
                LENGTH (next_code))
                curr_node. children[curr_sym] = TRIE _ NODE _ CREATE
                (next_code)
                curr_node = root
                next_code = next_code + 1
```

```
if next_code is MAX _ CODE
        TRIE _ RESET (root)
        curr_node = root
        next_code = START _ CODE
        prev_sym = curr_sym
if curr_node is not root
        WRITE _ PAIR (outfile, prev_node. code, prev_sym, BIT - LENGTH
        (next_code))
        next_code = (next_code + 1) % MAX _ CODE
WRITE _ PAIR (outfile, STOP _ CODE , 0, BIT - LENGTH (next_code))
FLUSH _ PAIRS (outfile)
```

fix_endianess(FileHeader *h)

- ➢ fixes the values inside a file header depending if a computer is big or little endian

- ➔ if system is big endian, swap the endianness of header values

decode.c

int main()

- ➢ does LZ78 decompression

- ➔ parses the command line options as described as the start of the file
- ➔ checks for file errors
- ➔ prints help statistics if -h flag or invalid flag is passed and exits with error code 1
- ➔ make output file permission same as input file permission
    - ➞ get permission status of infile
    - ➞ set permission status of outfile to infile's permission status
- ➔ read file header
- ➔ call fix_endianess()
- ➔ make sure header's magic is same as MAGIC (from io.h)
- ➔ follows the LZ78 decompression pseudocode presented in the assignment document and below
- ➔ prints the statistics if needed
    - ➞ compressed file size
    - ➞ uncompressed file size
    - ➞ space saving -> 100 * (1 - (comp_fs/ uncomp_fs));
- ➔ closes the files and exits

Given LZ78 Decompression Algorithm Pseudocode

```
table = WT _ CREATE ()
curr_sym = 0
curr_code = 0
next_code = START _ CODE
while READ _ PAIR (infile, &curr_code, &curr_sym, BIT - LENGTH (next_code)) is
TRUE
        table[next_code] = WORD _ APPEND _ SYM (table[curr_code],
        curr_sym)
        WRITE _ WORD (outfile, table[next_code])
        next_code = next_code + 1
if next_code is MAX _ CODE
        WT _ RESET (table)
        next_code = START _ CODE
FLUSH _ WORDS (outfile)
```

fix_endianess(FileHeader *h)

- fixes the values inside a file header depending if a computer is big or little endian

- if system is big endian, swap the endianness of header values

io.c

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

- reads a specified amount of bytes from a text file into a buffer

- because read() does not guarantee to read all of the bytes specified, we need to call it in a loop until no more bytes to read
  - increment total bytes read by the bytes just read
  - decrement the # of bytes that need to be read in by the # of bytes we just read
  - moves the buffer pointer so that we don't rewrite the bytes we just read when we call the loop again

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

- writes a specified amount of bytes into a text file from a buffer

- because write() does not guarantee to write all of the bytes specified, we need to call it in a loop until no more bytes to write

- increment total bytes read by the bytes just written
- decrement the # of bytes that need to be written by the # of bytes we just wrote
- moves the buffer pointer so that we don't rewrite the bytes we just wrote when we call the loop again

void read_header(int infile, FileHeader *header)

➢ reads a header from an infile and stores it inside a pointer

➔ reads the magic and protection data into a FileHeader ADT
➔ increase total bits processed by the amount of bits in a file header

void write_header(int outfile, FileHeader *header)

➢ writes a header to the outfile

➔ writes the magic and protection data from a FileHeader ADT, into the outfile
➔ increase total bits processed by the amount of bits in a file header

bool read_sym(int infile, uint8_t *sym)

➢ reads the next symbol from the infile and sets it to the passed in sym pointer

➔ if symbuf is empty, fill it
  - end = the amount of bytes available in the symbuf + 1
➔ set sym to the next sym in the symbuf
➔ if symbuf is full, empty it
➔ sym_index == end, there are no more bytes left to read in the file
  - return false
➔ return true

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

➢ writes a code and a symbol to an outfile in a pair

➔ if computer is big endian, swap the code's bits
➔ copies each bit in 'code' to 'bitbuf'
  - copy the bit
  - if the buffer is filled up, empty it into the outfile to allow more bits to be read in
➔ copies each bit in 'sym' to 'bitbuf' so that code and sym are next to eachother
  - copy the bit

- ➞ if the buffer is filled up, empty it into the outfile to allow more bits to be read in
- ➔ zero's out the rest of the byte so that we don't accidently print out wrong numbers
- ➔ increase total bits by the amount of bits written

void flush_pairs(int outfile)

- ➢ empties bitbuf into the outfile

- ➔ write_bytes(outfile, bitbuf, bytes(bit_index));
- ➔ set bit_index to 0

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

- ➢ create a code that will eventually overwrite the passed in 'code'

- ➔ reads a code and a symbol from an infile
- ➔ copies 'bitlen' amount of bits from infile to 'bitbuf'
  - ➞ if bitbuf is empty, fill it up
  - ➞ if there are no more bytes to read and we haven't read 'bitlen' amount of bits, abort the program bc the file was corrupted
- ➔ copy the bit at bit_index of 'bitbuf' to 'read_code'
- ➔ if the end of 'bitbuf' is reached, prepare to overwrite 'bitbuf' with the next bytes to be read by setting bit_index to 0
- ➔ if computer is big endian, swap the code's bits
- ➔ set the code parameter to the read in code
- ➔ create a symbol that will eventually overwrite the passed in 'sym'
- ➔ if bitbuf is empty, fill it up
- ➔ if there are no more bytes to read and we haven't read 'bitlen' amount of bits, abort the program bc the file was corrupted
- ➔ copy the bit at bit_index of 'bitbuf' to 'sym'
- ➔ if the end of 'bitbuf' is reached, prepare to overwrite 'bitbuf' with the next bytes to be read
- ➔ set the sym parameter to the read in sym
- ➔ increase total bits by the amount of bits read
- ➔ if read_in_code == STOP_CODE, there are no more pairs left to read after this one
  - ➞ return false
- ➔ return true bc there are more pairs left to read after this one

void write_word(int outfile, Word *w)

- ➢ writes a word to a file

- ➔ iterate through the symbols in 'w'
  - ➞ fill up the buf with the 'sym's
  - ➞ increment total syms processed
  - ➞ empty the buf if it gets filled up

void flush_words(int outfile)

- ➢ empties all the words currently in the symbuf to a file

- ➔ call write_bytes(outfile, symbuf, sym_index);
- ➔ set sym_index to 0

trie.c

TrieNode *trie_node_create(uint16_t code)

- ➢ the constructor for a trie node

- ➔ create a trienode n
- ➔ set n->code to code
- ➔ set all of n's children to NULL
- ➔ return n

void trie_node_delete(TrieNode *n)

- ➢ frees the memory allocated to a trie node

- ➔ free(n)

void trie_reset(TrieNode *root)

- ➢ deletes all children nodes to a root node

- ➔ iterate through all the children of the current node
  - ➞ delete each child's subtree using trie_delete(child)

void trie_delete(TrieNode *n)

- ➢ recursively deletes a root node and its children

- ➔ iterate through all the children of the current node
  - ➞ delete each child's subtree using trie_delete()

➔ delete the root node

TrieNode *trie_step(TrieNode *n, uint8_t sym)

➢ get a pointer to the child node representing sym

➔ return n->children[sym]

word.c

Word *word_create(uint8_t *syms, uint32_t len)

➢ the constructor for a trie node

➔ allocate memory for a word in the heap
➔ allocate memory for w->syms in the heap

Word *word_append_sym(Word *w, uint8_t sym)

➢ creates a new word with a symbol appended to the end of the passed in word

➔ create new word (new_word)
➔ allocate enough size for the old word + enough size for the new sym
➔ copy the syms from 'w' to 'new_word'
➔ set the last spot of 'new_word' to 'sym'

void word_delete(Word *w)

➢ frees the memory allocated to a word

➔ free(w->syms)
➔ free(w)

WordTable *wt_create(void)

➢ creates an array of words with an empty word initialized at index EMPTY_CODE

➔ allocate space for an array of words in the heap
➔ set wt[EMPTY_CODE] to word_create(NULL,0);

void wt_delete_words(WordTable *wt)

- ➢ sets all the words in a word table to NULL except the first word which is an empty word

- ➔ deletes all of the words in the table
- ➔ creates the empty word in the first spot of the table

void wt_delete(WordTable *wt)

- ➢ deletes all the words in the table

- ➔ call wt_delete_words(wt);
- ➔ free(wt)

bit_tools.c

uint8_t bitlen(uint16_t code)

- ➢ gets the bit length of a code

- ➔ shifts the bits in code to the right by 1 until code = 0
  - ➞ increments a counter to count how many times it performs this operation

uint32_t bytes(uint32_t bits)

- ➢ gets the amount of bytes needed for an amount of bits

- ➔ if bits can perfectly fit into a number of bytes, then return bits/8
- ➔ if bits doesn't perfectly fit into a number of bytes, allocate 1 extra byte so that the remaining bits can have space

uint8_t bt_8_get_bit(uint8_t byte, uint8_t index)

- ➢ gets a bit from a byte

- ➔ make sure that the index is in the range of the amount of bits in a byte
- ➔ make a byte with a 1 shifted 'index' spots in (put a 1 at spot 'index')
- ➔ multiply the 'mask' and 'byte' bytes together so that all bits that aren't in spot 'index' must be 0s
- ➔ move the bit to the least significant bit spot in the byte and return it

void bt_8_set_bit(uint8_t *byte, uint8_t index)

- ➢ gets a bit from a byte

- ➔ make sure that the index is in the range of the amount of bits in a byte
- ➔ make a byte with a 1 shifted 'index' spots in (put a 1 at spot 'index')
- ➔ set the byte to be the union of 'word' and 'mask'

void bt_8_clr_bit(uint8_t *byte, uint8_t index)

- ➢ sets a bit to 0 in a byte

- ➔ make sure that the index is in the range of the amount of bits in a byte
- ➔ make a byte with a 1 shifted 'index' spots in (put a 1 at spot 'index')
- ➔ multiply the 'mask' and 'byte' bytes together so that all bits that aren't in spot 'index' stay the same while spot 'index' turns to a 0

uint8_t bt_16_get_bit(uint8_t word, uint8_t index)

- ➢ gets a bit from a word

- ➔ make sure that the index is in the range of the amount of bits in a word
- ➔ make a word with a 1 shifted 'index' spots in (put a 1 at spot 'index')
- ➔ multiply the 'mask' and 'word' words together so that all bits that aren't in spot 'index' must be 0s
- ➔ move the bit to the least significant bit spot in the word and return it

void bt_16_set_bit(uint8_t *word, uint8_t index)

- ➢ gets a bit from a word

- ➔ make sure that the index is in the range of the amount of bits in a word
- ➔ make a word with a 1 shifted 'index' spots in (put a 1 at spot 'index')
- ➔ set the word to be the union of 'word' and 'mask'

uint8_t bt_buf_get_bit(uint8_t *bytes, uint32_t bit_index)

- ➢ gets a bit from an array of bytes

- ➔ return bt_8_get_bit(bytes[bit_index/8], bit_index%8)

void bt_buf_set_bit(uint8_t *bytes, uint32_t bit_index)

- ➢ sets a bit to 1 in an array of bytes

➔ bt_8_set_bit(&bytes[bit_index/8], bit_index%8)

void bt_buf_clr_bit(uint8_t *bytes, uint32_t bit_index)

- ➢ sets a bit to 1 in an array of bytes

➔ bt_8_clr_bit(&bytes[bit_index/8], bit_index%8)

Lossy compression – compress but lose data

Lossless compression – compress without losing data

LZ78 thought proccers

STOP_CODE=0

EMPTY _CODE=1

word	code
""	EmptyCode
"a"	start_code
"b"	3
"ab"	4

encode: "abab"

current word → cw = ""

cw=cw+"a" = "a" (add "a")

we added an entry so we will reset the cur word to the empty word

cw = ""

cw = cw + "b"

cw = ""

cw = cw + "a" = "a"

don't add "a" b/c its already in the dictionary

cw = cw + "b" = "ab"

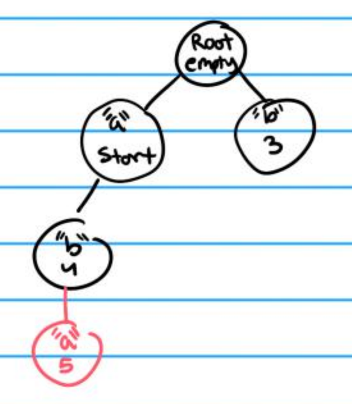
compressed text:

(Empty_code, "a") → | "" + "a" = "a"
(Empty_code, "b") → | "" + "b" = "b"
(start_code, "b") → | "a" + "b" = "ab"

} "abab"

Tree method

do it like the prev example but put it in a tree



decode "abababa"

(Empty, "a")
(Empty, "b")
(Start, "b")
(4, "a")

TrieNode

children [Alphebet] → indexing is O(1) b/c we use ASKii as array index

recursive → trie_reset ( ):
- deletes everything but the root Node
- itterate through children of root & call trie delete()

none recursive → trie_delete(node)
- itterate through children notes:
  - if child ≠ Null
    - trie_detete(child)
    - child = Null
  - trie_node_delete(n)

Word Tables

Word wordtable ← array of words

↑ not struct b/c there are no parts

word_append_sym (Word *syms, uint32_t len)

ex syms = "ab" len= "c"

return "abc"

I/O

make sure majic # matches bfr reading

Buffering

- one for reads & one for writes

Buffer - an array of bytes

Simulation

File: Hello world

u8 symbols [Block] = [H|E|L|L|O| |W|O|R|L|D]
↑ index

```
read_sym (*sym)
    if index ==0 {
        read_bytes (symbols, BLOCK)
    }
    *sym = symbols [index]
    index +=1

    if (index == BLOCK){
        index=0
    }
```

```
read_bytes(symbols, block ) ← returns # of read bytes
    n= 5
    if 5< BLOCK{
        endofBuffer =n
    }
    if endOfBuffer == index
        return false
```

→ BLOCK =5

0	1	2	3	4	5
h	e	l	l	o	

print out all read in symbols

```
while (read_sym (stdin, &sym) {
    print (sym)
}
```

```
read_sym( *sym)
    if(index ==0){
        read_bytes (symbols ,BLOCK)
    }
    *sym=symbols [index]
    index+=1
```

Writing pairs

u8 bitand codes [BLOCK]

int bit index = 0

bit Index=0 ↓

bits AndCode [0] =MSB 0000 0000LSB

bits AndCode [1] =MSB 0000 0000LSB

bits AndCode [2] =MSB 0000 0000LSB

bits AndCode [3] =MSB 0000 0000LSB

```
write-pair(outfile,code,sym,bitlen) {     ← sym = symbol
    for (i=0; i<bitlen; i++){
        if(bit at spot "i" of code ==1){
            set the bitindex of bits & code
        else
            clr the bitindex of bits & code

    for (i=0; i<8; i++)
        is the i th bit of sym set?
    if buffer is filled, then value of
        if bitIndex == BLOCK*8
            write_bytes (bitAndCode, BLOCK)
```

Ex) code =13

buffer 5 bits → bitlen =5

MSB 0000 1101 LSB
7654 3210

bits AndCode [0] =MSB 0000 1101 LSB

↑ buffer =5 so we stop here