Assignment 6 DESIGN.pdf

Roman Luo

March 12, 2023

1 Description of Program

Two programs called `encode` and `decode` which perform LZ78 compression and decompression, respectively. The requirements for your programs are as follows:

1. `encode` can compress any file, text or binary.
2. `decode` can decompress any file, text or binary, that was compressed with `encode`.
3. Both operate on both little and big endian systems. Interoperability is required.
4. Both use variable bit-length codes.
5. Both perform read and writes in efficient blocks of 4KB.
6. Both `encode` and `decode` must interoperate with the provided binaries – not just your code.

2 Files to be included in directory "asgn6":

- Major files
  1. `encode.c`: contains the `main()` function for the `encode` program.
  2. `decode.c`: contains the `main()` function for the `decode` program.
  3. `trie.c`: the source file for the Trie ADT.
  4. `trie.h`: the header file for the Trie ADT. You must not modify this file.
  5. `word.c`: the source file for the Word ADT.
  6. `word.h`: the header file for the Word ADT. You must not modify this file.
  7. `io.c`: the source file for the I/O module.
  8. `io.h`: the header file for the I/O module. You must not modify this file.
  9. `endian.h`: the header file for the endianness module. You must not modify this file.
  10. `code.h`: the header file containing macros for reserved codes. You must not modify this file.
- Minor files
  1. Makefile
  2. README.md
  3. DESIGN.pdf
  4. WRITEUP.pdf
  5. Working encode and decode programs, along with test files, and other source files will be supplied in the resources repository.

3 Pseudocode / trie.c:

3.1 trie node create

```
// Constructor for a TrieNode
TrieNode *trie_node_create(uint16_t index) {
 allocate memory for for TrieNode using malloc with sizeof itself
 if allocated {
        The node's code is set to code.
        for i in range 256 (ALPHABET)
            Make sure each of the children node pointers are NULL
    }
 return the node;
}
```

3.2 trie node delete

```
// Destructor for a TrieNode.
void trie_node_delete(TrieNode *n) {
    if (n is NULL) {
        return the program
    }
 free n
}
```

3.3 trie create

```
TrieNode *trie_create(void) {
 // Initializes a trie: a root TrieNode with the code EMPTY_CODE.
 set TrieNode *root to trie_node_create(EMPTY_CODE)
 // Returns the root, a TrieNode *, if successful, NULL otherwise.
 if (root is NULL) {
        return NULL;
    } else {
     return the root created;
    }
}
```

3.4 trie reset

```
// Resets a trie to just the root TrieNode.
void trie_reset(TrieNode *root) {
 // Since we are working with finite codes,
 // eventually we will arrive at the end of the available codes (MAX_CODE).
 // At that point, we must reset the trie by deleting its children
 // so that we can continue compressing/decompressing the file.
 // Make sure that each of the root's children nodes are NULL.
 if (root is NULL) {
        return;
    }
    for i in range 256 (ALPHABET) {
        use trie_delete() with root->children[i];
        then set root->children[i] to NULL;
    }
}
```

3.5 trie delete

```
// Deletes a sub-trie starting from the trie rooted at node n.
void trie_delete(TrieNode *n) {
 // This will require recursive calls on each of n’s children.
 if (n is NULL) {
        return;
    }
    for i in range 256 (ALPHABET) {
         recursive calls trie_delete() with n->children[i]
    }
    // Make sure to set the pointer to the children nodes to NULL
    // after you free them with trie_node_delete()
    call trie_node_delete(n);
}
```

3.6 trie step

```
// Returns a pointer to the child node reprsenting the symbol sym.
TrieNode *trie_step(TrieNode *n, uint8_t sym){
 // If the symbol doesn’t exist, NULL is returned.

 if (n is NULL) {
        return NULL;
    }
    return n->children[sym];
}
```

4 Pseudocode / word.c:

4.1 word create

```
// Constructor for a word where sysms is the array of symbols a Word represents.
Word *word_create(uint8_t *syms, uint32_t len) {
    malloc the sizeof Word and set it to w;
    // This function returns a Word * if successful or NULL otherwise.
    if (w) {
        // The length of the array of symbols is given by len.
        calloc with len + 1 and sizeof(uint8_t) to w->syms
        set w->len to len;
        return w;
    } else {
        return NULL
    }
}
```

4.2 word append sym

```
Word *word_append_sym(Word *w, uint8_t sym) {
    set new_len to w->len + 1;
    // uint8_t new_sym = w->syms + sym;
    // call word_create
    Word *new_w = (Word *) malloc(sizeof(Word));
    if (new_w) {
        // The length of the array of symbols is given by len.
        allocate new_w->syms with (uint8_t *) calloc(new_len, sizeof(uint8_t));
        // copy chars in syms over
        for in range w->len {
           set new_w->syms[i] to w->syms[i];
        }
        add the new sym to new_w->syms[new_len - 1]
        update the new_w->len to new_len;
    }
    return new_w;
}
```

4.3 word delete

```
// Destructor for a Word, w.
void word_delete(Word *w) {
    free the word->syms
    free the word
}
```

4.4 wt create

```
// Creates a new WordTable, which is an array of Words.
WordTable *wt_create(void) {
    // allocate memory for word table
    // A WordTable has a pre-defined size of MAX_CODE, which has the value UINT16_MAX.
    Create WordTable *wt by calloc with MAX_CODE, sizeof(Word)
    // call to word create
    // A WordTable is initialized with a single Word at index EMPTY_CODE.
    // represents the empty word, a string of length of zero.
    set wt[EMPTY_CODE] to word_create(NULL, 0);
    return wt;
}
```

4.5 wt reset

```
// Resets a WordTable, wt, to contain just the empty Word.
void wt_reset(WordTable *wt) {
 // Make sure all the other words in the table are NULL.
 for in range of MAX_CODE starting from START_CODE {
        if (wt[i] is not NULL) {
            set wt[i] to NULL;
        }
    }
}
```

4.6 wt delete

```
// Destructor for a Word, w.
void wt_delete(WordTable *wt) {
 for in range of MAX_CODE starting from EMPTY_CODE {
        if (wt[i] is not NULL) {
            call word_delete(wt[i]);
        }
    }
    free memory allocated for the WordTable wt;
}
```

5 Pseudocode / io.c:

5.1 read bytes

```
int read_bytes(int infile, uint8_t *buf, int to_read) {
    if to_read is 0, return 0;

    keep track bytes read, init to 0
    keep track of bytes read on latest read call to 0

    while (bytes read = read(infile, buffer, to_read - bytes read so far)) {
        total bytes read so far += bytes read
        break once reaach to_read amouts of bytes
    }
    return total bytes read
}
```

5.2 write bytes

```
int write_bytes(int outfile, uint8_t *buf, int to_write) {
    if to_read is 0, return 0;

    keep track bytes wrote, init to 0
    keep track of bytes wrote on latest read call to 0

    while (bytes read = write(infile, buffer, to_read - bytes read so far)) {
        total bytes wrote so far += bytes wrote
        break once reaach to_write amouts of bytes
    }
    return total bytes wrote
}
```

5.3 read header

```
void read_header(int infile, FileHeader *header) {
    // This reads in sizeof(FileHeader) bytes from the input file.
    // These bytes are read into the supplied header.
    set bytes_read by calling read_bytes(infile, (uint8_t *)header, sizeof(FileHeader));
    if (bytes_read does not equal sizeof(FileHeader)) {
        return;
    }
    // Endianness is swapped if byte order isn't little endian.
    if (is in big_endian format) {
        swap using 32(header->magic);
        swap using 16(header->protection);
    }
    // Along with reading the header, it must verify the magic number.
    if (header->magic does not equal MAGIC) {
        return;
    }
}
```

5.4 write header

```
void write_header(int outfile, FileHeader *header) {
    // Endianness is swapped if byte order isn't little endian.
    if (is in big_endian format) {
        swap using 32(header->magic);
        swap using 16(header->protection);
    }
    // Writes sizeof(FileHeader) bytes to the output file.
    // These bytes are from the supplied header.
    set bytes_wrote by calling write_bytes(outfile, (uint8_t *)header, sizeof(FileHeader));
    if (bytes_wrote does not equal sizeof(FileHeader)) {
        return;
    }
}
```

5.5 read sym

```
bool read_sym(int infile, uint8_t *sym) {
    // maintain a global buffer (an array) of BLOCK bytes.
    // uint8_t sym_buffer[BLOCK];
    // An index keeps track of the currently read symbol in the buffer.
    // static int sym_buffer_index = 0;

    // if no more bytes in the buffer
    if (sym_buffer_index is greater sym_buffer_index_end) {
        // call read_bytes to refill the buffer with fresh data
        get bytes_read with read_bytes(infile, sym_buffer, BLOCK);
        // If this call fails then you cannot read a symbol and should return false.
        if (bytes_read is 0) {
            return false;
        }
        reset sym_buffer_index to 0;
        set sym_buffer_index_end to bytes_read;
    }
    // Read one symbol from infile into *sym.
    set the sym we read from infile buffer sym_buffer[sym_buffer_index] to the *sym
    // update some counter
    update sym_buffer_index by adding 1;
    update total_syms by adding 1;

    return true;
}
```

5.6 write pair

```
void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen) {
    // uint8_t pair_buffer[BLOCK];
    // pair_buffer_index = 0;
    // \Writes" a pair to outfile. In reality, the pair is buffered.
    // A pair is comprised of a code and a symbol.

    // The bits of the code are buffered first, starting from the LSB.
    for i in bitlen {
        // checks if a pair_buffer is full
        if (write_pair_buffer_index is 8 * BLOCK) {
            write the pair_buffer to oufile
            Set buffer to all zeros. Varun
            reset index to 0
        }
        // checks if the current bit of code is set
        if (code & (1 << i)) {
            // set the corresponding bit in the buffer
            // write_pair_buffer_index / 8 calcuates buffer index
            // write_pair_buffer_index % 8 calcuates the amount to set for specific bit
            set pair_buffer[write_pair_buffer_index / 8] to using | (1 << write_pair_buffer_index % 8
        }
        add 1 to index
    }
    // The bits of the symbol are buffered next, also starting from the LSB.
    for in range of 8 {
        // checks if a pair_buffer is full
        if (write_pair_buffer_index is 8 * BLOCK) {
            write the pair_buffer to oufile
            Set buffer to all zeros. Varun
            reset index to 0
        }
        // checks if the current bit of sym is set
        if (sym & (1 << i)) {
            // set the corresponding bit in the buffer
            // write_pair_buffer_index / 8 calcuates buffer index
            // write_pair_buffer_index % 8 calcuates the amount to set for specific bit
            set pair_buffer[write_pair_buffer_index / 8] to using | (1 << write_pair_buffer_index % 8
        }
        add 1 to index
    }
    // The code buffered has a bit-length of bitlen. The buffer is written out whenever it is filled.
    update the total_bits with bitlen + 8;
}
```

5.7 flush pair

```
void flush_pairs(int outfile) {
    // calculate number of bytes needed to write out remaining bits
    set remaining_bytes to write_pair_buffer_index / 8
    plus 1 if write_pair_buffer_index % 8 is 0

    write remaining bytes to outfile
    reset buffer and buffer index
}
```

5.8 read pair

```
bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen) {
    initialize int bytes_read;

    if (read_pair_buffer_index is 0) {
        reads BLOCK bytes from the input file
        if no bytes were read {
            return false;
        }
    }

    set *code to 0;
    // The bits of the code are buffered first, starting from the LSB.
    for in range of bitlen {
        // checks if a pair_buffer is full
        if (read_pair_buffer_index is 8 * BLOCK) {
            read BLOCK bytes from infile to pair_buffer
            and set bytes_read
            if (bytes_read is 0) {
                return false;
            }
            // reset index to 0
        }
        // return (x->v[k / BITS_PER_UNIT] >> k % BITS_PER_UNIT) & 0x1;
        if (extracts the value of the least significant bit of the shifted byte & (uint8_t) 1) {
            sets the ith bit of code to 1 with | operator
        }
        add 1 to read_pair_buffer_index
    }

    *sym = 0;
    // The bits of the symbol are buffered next, also starting from the LSB.
    for in range 8 {
        if (read_pair_buffer_index == 8 * BLOCK) {
            read BLOCK bytes from infile to pair_buffer
            and set bytes_read
            if (bytes_read is 0) {
                return false;
            }
            read_pair_buffer_index = 0;
        }
        // return (x->v[k / BITS_PER_UNIT] >> k % BITS_PER_UNIT) & 0x1;
        if (extracts the value of the least significant bit of the shifted byte & (uint8_t) 1) {
            sets the ith bit of sym to 1 with | operator
        }
        add 1 to read_pair_buffer_index
    }

    // Update the bit counters
    add bitlen + 8 to total_bits

    // Returns true if there are pairs left to read in the buffer, else false.
    // There are pairs left to read if the read code is not STOP_CODE.
    check if code is not equal to STOP_CODE
    return the result
}
```

5.9 write word

```
void write_word(int outfile, Word *w) {
    for i in range of w->len {
        // Each symbol of the Word is placed into a buffer.
        set sym_buffer[sym_buffer_index] to w->syms[i];
        add 1 to sym_buffer_index
        // The buffer is written out when it is filled.
        if (sym_buffer_index is BLOCK) {
            write_bytes sym_buffer to with BLOCK len
            set sym_buffer_index to 0;
        }
        add 1 to total_syms
    }
}
```

5.10 flush words

```
void flush_words(int outfile) {
    // calculate number of bytes needed to write out remaining symbols
    set remaining_bytes to sym_buffer_index;

    // write remaining bytes to outfile
    write sym_buffer to outfile with remaining_bytes

    reset buffer and buffer index
}
```

6 Pseudocode / encode, decode:

6.1 encode

```
    // this function takes a uint16 and returns its bit length
    int bit_len(uint16_t n) {
        set count to 0;
        while (n is not 0) {
            add 1 to count
            set n to iself right shift 1;
        }
        return count;
    }

    Parse command-line options using getopt() and handle them accordingly.
    While
        case 'i': infile_name = optarg; break;
        case 'o': outfile_name = optarg; break;
        case 'v': verbose = 1; break;
        case 'h': printf("%s", help_message); return 1;

    if a file name is given for infile {
        set infile_descriptor with open(infile_name, O_RDONLY);
        print error if failed
    }

    declare a FileHeader infile_header;
    initialize infile_header.magic to 0;
    initialize infile_header.protection to 0;

    set infile_header.magic to MAGIC;
    declare struct stat protection_bits;
    set fstat(infile_descriptor, &protection_bits);
    set infile_header.protection to protection_bits.st_mode;

    if a name is given for outfile {
        set outfile_descriptor with open(outfile_name, O_WRONLY | O_CREAT)
        this also create the file if it does not exist
        print error message if failed
    }
    set file permission with fchmod(outfile_descriptor, infile_header.protection);

    write_header(outfile_descriptor, &infile_header);

   C OMPRESS(infile, outfile)
```

```
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)

if (verbose) {
    Compressed file size is
        (total_bits / 8 + (total_bits % 8 ? 1 : 0)) + sizeof(FileHeader));
    Uncompressed file size is:
        total sym
    compression_percentage is:
    100.0 * (1.0 - Compressed file size / Uncompressed file size);

}

// 12. Use close() to close infile and outfile.
trie_delete(root);
close(infile_descriptor);
close(outfile_descriptor);
```

6.2 decode

```
// this function takes a uint16 and returns its bit length
int bit_len(uint16_t n) {
    set count to 0;
    while (n is not 0) {
        add 1 to count
        set n to iself right shift 1;
    }
    return count;
}
```

```
 Parse command-line options using getopt() and handle them accordingly.
 While
     case 'i': infile_name = optarg; break;
     case 'o': outfile_name = optarg; break;
     case 'v': verbose = 1; break;
     case 'h': printf("%s", help_message); return 1;

 if a file name is given for infile {
     set infile_descriptor with open(infile_name, O_RDONLY);
     print error if failed
 }

 declare a FileHeader infile_header;
 read the header from infile to infile_header using read_header

 if a name is given for outfile {
     set outfile_descriptor with open(outfile_name, O_WRONLY | O_CREAT)
     this also create the file if it does not exist
     print error message if failed
 }
 set file permission with fchmod(outfile_descriptor, infile_header.protection);

 write_header(outfile_descriptor, &infile_header);

C OMPRESS(infile, outfile)


 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)

 if (verbose) {
     Compressed file size is
         (total_bits / 8 + (total_bits % 8 ? 1 : 0)) + sizeof(FileHeader));
     Uncompressed file size is:
         total sym
     compression_percentage is:
     100.0 * (1.0 - Compressed file size / Uncompressed file size);

 }

 // 12. Use close() to close infile and outfile.
 trie_delete(root);
 close(infile_descriptor);
 close(outfile_descriptor);
```

7 Pseudocode / Makefile:

7.1 Makefile

- `CC = clang` must be specified.
- `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified.
- `pkg-config` to locate compilation and include flags for the GMP library must be used.
- `make` must build the `encode`, `decode` executables, 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.
- `make format` should format all your source code, including the header files.