Assignment 6 Design Document

Zihua Li

CSE 13S: Computer Systems and C Programming

University of California, Santa Cruz

ABSTRACT

A design document of a program which compresses files using the LZ78 compression algorithm and decompresses the compressed files with the corresponding decoder.

Keywords: Lempel-Ziv Compression, C Programming, LZ78

TRIE.C

Each node has 256 pointers, one for each ASCII character.

*trie_node_create(uint16_t code)

This is a constructor for a TrieNode. The node's code is set to code. Children node pointers are set to NULL.

```
Pseudocode
def trie_node_create(code):
    node->code = code
    i = 0
    when i < ALPHABET:
        node->children[i] = NULL
        i += 1
    return code
```

trie_node_delete(TrieNode *n)

This is a destructor for a TrieNode.

```
Pseudocode
def trie_node_delete(n):
    if n == NULL
        return
    i = 0
    when i < ALPHABET:
        trie_node_delete(n->children[i])
        i += 1
    del(n)
```

*trie_create(void)

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

```
Pseudocode
def trie_create():
    root = trie_node_create(EMPTY_CODE)
    if root == NULL:
        return NULL
```

```
    else:
        return root
```

trie_reset(TrieNode *root)

Resets a trie to just the root TrieNode. 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.

```
Pseudocode
def trie_reset(root):
    if root == NULL:
        return
    i = 0
    when i < ALPHABET:
        if root->children[i] != NULL:
            trie_delete(root->children[i])
            root->children[i] = NULL
        i += 1
    i = 0
    when i < MAX_CODE:
        root->code = EMPTY_CODE
        i += 1
```

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. Make sure to set the pointer to the children nodes to NULL after you free them with trie_node_delete().

```
Pseudocode
def trie_delete(n):
    if n == NULL:
        return
    i = 0
    when i < ALPHABET:
        trie_delete(n->children[i])
        n->children[i] = NULL
        i += 1
    trie_node_delete(n)
```

*trie_step(TrieNode *n, uint8_t sym)

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

```
Pseudocode
def trie_step(n, sym):
    if n == NULL:
        return NULL
    return n->children[sym]
```

WORD.C

Decompression needs to use a look-up table for quick code to word translation. This look-up table will be defined as a new struct called a WordTable.

***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.

```
Pseudocode
def word_create(syms, len):
    word = []
    for i in range(len(Word)):
        word.append('')
    if word == NULL:
        return NULL
    word->syms = []
    for i in range(uint8_t):
        word->syms.append('')
    if word->syms == NULL:
        del(word)
        return NULL
    memcpy(word->syms, syms, len * len(uint8_t))
    word->len = len
    return 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 to may be empty. If the above is the case, the new Word should contain only the symbol. Returns the new Word which represents the result of appending.

```
Pseudocode
def word_append_sym(w, sym):
    new_len = w->len + 1
    new_syms = []
    for i in range(new_len * len(uint8_t)):
        new_syms.append('')
    if new_syms == NULL:
        return NULL
    memcpy(new_syms, w->syms, w->len * len(uint8_t))
    new_syms[new_len - 1] = sym
    new_word = word_create(new_syms, new_len)
    del(new_syms)
    return new_word
```

word_delete(Word *w)

Destructor for a Word, w.

```
Pseudocode
def word_delete(w):
    if w != NULL:
        del(w->syms)
        del(w)
```

***wt_create(void)**

Creates a new WordTable, which is an array of Words. A WordTable has a pre-defined size of MAX_CODE, which has the value UINT16_MAX. This is because codes are 16-bit integers. A WordTable is initialized with a single Word at index EMPTY_CODE. This Word represents the empty word, a string of length of zero.

```
Pseudocode
```

```
def wt_create():
    wt = [] #WordTable
    for i in range(len(WordTable) * (MAX_CODE + 1)):
        if wt == NULL:
            return NULL
        empty_word = word_create(NULL, 0)
        if empty_word == NULL:
            del(wt)
            return NULL
        wt[EMPTY_CODE] = empty_word
        i = 0
        when i < EMPTY_CODE:
            wt[i] = NULL
            i += 1
        i = EMPTY_CODE + 1
        when i <= MAX_CODE:
            wt[i] = NULL
            i += 1
        return wt
```

wt_reset(WordTable *wt)

Resets a WordTable, wt, to contain just the empty Word. Make sure all the other words in the table are NULL.

```
Pseudocode
def wt_reset(wt):
    i = 0
    when i <= MAX_CODE:
        if wt[i] != NULL AND i != EMPTY_CODE:
            word_delete(wt[i])
            wt[i] = NULL
```

IO.C

It is also a requirement that your programs, encode and decode, perform efficient I/O. Reads and writes will be done 4KB, or a block, at a time, which implicitly requires that you buffer I/O. Buffering is the act of storing data into a buffer, which you can think of as an array of bytes.

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

This will be a useful helper function to perform reads. As you may know, the read() syscall does not always guarantee that it will read all the bytes specified. For example, a call could be issued to read a block of bytes, but it might only read half a block. So, we write a wrapper function to loop calls to read() until we have either read all the bytes that were specified (to_read), or there are no more bytes to read. The number of bytes that were read are returned. You should use this function whenever you need to perform a read.

```
Pseudocode
def read_bytes(infile, buf, to_read): #buf is a pointer
    saw_end_of_file = 0
    if saw_end_of_file == 1:
        return 0
    already_read = 0
    while already_read < to_read:
        read_this_time = 0
        if infile != 0:
            read_this_time = read(infile, buf, to_read - already_read)
```

```
        else:
            read_time_time = read(infile, buf, 1)
        if read_this_time == -1:
            print(ARGV0, ": read_bytes(): \
                error during read().\n", sep="")
            exit()
        if read_this_time == 0:
            saw_end_of_file = 1
            break
        already_read += read_this_time
        buf += read_this_time
    total_bytes_read += already_read
    return already_read
```

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

This function is very much the same as read_bytes(), except that it is for looping calls to write(). As you may imagine, write() isn't guaranteed to write out all the specified bytes (to_write), and so we loop until we have either written out all the bytes specified, or no bytes were written. The number of bytes written out is returned. You should use this function whenever you need to perform a write.

Pseudocode

```
def write_bytes(outfile, buf, to_write): #buf is a pointer
    already_written = 0
    while already_written < to_write:
        written_this_time = \
            write(outfile, buf, to_write - already_written)
        if written_this_time == -1:
            printf(ARGV0, ": \
                write_bytes(): error during write()\n", sep="")
            exit()
        if written_this_time == 0:
            break
        already_written += written_time_time
    total_bytes_written += already_written
    return already_written
```

read_header(int infile, FileHeader *header)

This 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.

Pseudocode

```
def read_header(infile, header): #header is a pointer
    bytesRead = read_bytes(infile, header, len(FileHeader))
    if bytesRead != len(FileHeader):
        print(ARGV0, ": read_header(): \
            Unable to read full FileHeader.\n", sep="")
        exit()
    if big_endian != NULL:
        header->magic = swap32(header->magic)
        header->protection = swap16(header->protection)
    if header->magic != 0xBAADBAAC:
        printf(ARGV0, ": read_header(): \
            FileHeader magic number is incorrect.\n", sep="")
        exit()
```

write_header(int outfile, FileHeader *header)

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

```
Pseudocode
def write_header(outfile, header): #header is a pointer
    if big_endian() != NULL:
        header->magic = swap32(header->magic)
        header->protection = swap16(header->protection)
    if header->magic != 0xBAADBAAC:
        print(ARGV0, ": write_header(): \
            FileHeader magic number is incorrect.\n", sep="")
        exit()
    fileHeader = header #fileHeader is a pointer
    i = 0
    while i < len(FileHeader):
        cWrPtr++ = fileHeader++ #they are both pointers
        if (cWrPtr == &wruteBuffer[BLOCK]):
            write_bytes(outfile, writeBuffer, BLOCK)
            cWrPtr = writeBuffer
        i += 1
```

read_sym(int infile, uint8_t *sym)

An 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.

```
Pseudocode
def read_sym(infile, sym): #sym is a pointer
    if cRdPtr == eRdPtr:
        int nBytesRead = \
            read_bytes(infile, readBuffer, len(readBuffer))
        cRdPtr = readBuffer
        eRdPtr = cRdPtr + nBytesRead
        if cRdPtr == eRdPtr:
            return false
    sym = cRdPtr++ #they are both pointers
    return true
```

write_bits(int outfile, uint16_t value, int bitlen)

An assistive function to write_pair(), most of write_pair()'s jobs are completed by this function. "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. The bits of the symbol are buffered next, also starting from the LSB. The code buffered has a bit-length of bitlen. The buffer is written out whenever it is filled.

```
Pseudocode
def write_bites(outfile, value, bitlen):
    if optAscii != NULL:
        cWrPtr++ = '' #pointer
        if cWrPtr == &writeBuffer[BLOCK]:
            write_bytes(outfile, writeBuffer, BLOCK)
            cWrPtr = writeBuffer
        while bitlen-- == 1:
            cWrPtr++ = value & 1 ? '1' : '0' #cWrPtr is a pointer
            value >>= 1;
            if cWrPtr == &writeBuffer[BLOCK]:
```

```
                write_bytes(outfile, wrteBuffer, BLOCK)
                cWrPtr = writebuffer
        return
    wrValueMask = 1
    while bitlen-- == 1:
        if !wrByteMsk = 1:
            *cWrPtr++ = wrByteVal
            if cWrPtr == &writeBuffer[BLOCK]:
                write_bytes(outfile, writeBuffer, BLOCK)
                cWrPtr = writeBuffer
            wrByteMsk = 1
            wrByteVal = 0
        if value & wrValueMask:
            wrByteVal |= wrByteMsk
        fflush(stderr)
        wrValueMask <<= 1
        wrByteMsk <<= 1
        fflush(stderr)
```

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

"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. The bits of the symbol are buffered next, also starting from the LSB. The code buffered has a bit-length of bitlen. The buffer is written out whenever it is filled.

```
Pseudocode
def write_pair(outfile, code, sym, bitlen):
    write_bits(outfile, code, bitlen)
    write_bits(outfile, sym, 8)
```

flush_pairs(int outfile)

Writes out any remaining pairs of symbols and codes to the output file.

```
Pseudocode
def flush_pairs(int outfile):
    if !optAscii == 1:
        if wrByteMsk == 1:
            *cWrPtr++ = wrByteVal
            if cWrPtr == &writeBuffer[Block]:
                write_bytes(outfile, writeBuffer, BLOCK)
                cWrPtr = writeBuffer
    write_bytes(outfile, writeBuffer, cWrPtr - writeBuffer)
    cWrPtr = writeBuffer
```

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

I will skip the read_bits() pseudocode, please reference to write_bits(), they are basically the same thing. We're reading a value starting with the least significant bit first, and starting with a value of 0 (as we'll be turning bits on, not off).

```
Pseudocode
def read_pair(infile, code, sym, bitlen): #code and sym are pointers
    read_bits(infile, &value, bitlen)
    pointer(code) = value
    read_bits(infile, &value, 8)
    pointer(sym) = value
    return pointer(code) != STOP_CODE
```

write_word(int outfile, Word *w)

"Writes" a pair to the output file. Each symbol of the Word is placed into a buffer. The buffer is written out when it is filled.

```
Pseudocode
def write_word(outfile, w): #w is Word pointer
    write_bytes(outfile, w->syms, w->len)
```

flush_words(int outfile)

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

```
Pseudocode
def flush_words(outfile):
    write_bytes(outfile, writeBuffer, cWrPtr - writeBuffer)
    cWrPtr = writeBuffer
```

ENCODE.C

Contains the main() function for the encode program.

main(int argc, char *const argv[])

This is the main function.

```
Pseudocode
def main(argc, argv[]):
    user_input = input("menu_message")
    if user_input == a:
        optAscii++
    elif user_input == i:
        inPath = strdup(option_argument)
        # assume it was split from user_input
    elif user_input == o:
        outPath = strdup(option_argument)
        break
    elif user_input == v:
        optVerbose++
        break
    elif user_input == h:
        print("menu_message")
        return
    else:
        print("menu_message")
        return
    #open infile
    inFile = open(inPath)
    #file header
    FileHeader outFileHeader = {0xBAADBAAC, inFileStats.st_mode}
    #open outfile
    outFile = open(outPath)
    #write header
    write_header(outFile, &outFileHeader)
    #create trie
    trieRoot = trie_create()
    curr_node = trieRoot
    #a monotonic counter to keep track of the next available code
    next_code = START_CODE
    #two variables to keep track of the previous trie node
```

```
    #and previously read symbol
    prev_node = NULL
    prev_sym = NULL
    #read all symbols from infile
    curr_sym = NULL
    while read_sym(inFile, &curr_sym):
        next_node = trie_step(curr_node, curr_sym)
        if next_node:
            prev_node = curr_node
            curr_node = next_node
        else:
            write_pair(outFile, curr_node->code, curr_sym, /
                bitLength(next_code))
            curr_node->children[curr_sym] = \
                trie_node_create(next_code)
            curr_node = trieRoot
            next_code = (next_code + 1) % (MAX_CODE + 1)
        if next_code == MAX_CODE:
            next_code = START_CODE
            trie_reset(trieRoot)
        prev_sym = curr_sym
        if curr_node != trieRoot:
            write_pair(outFile, prev_node->code, prev_sym, \
                bitLength(next_code))
            next_code = (next_code + 1) % (MAX_CODE + 1)
        #write the pairr (STOP_CODE, 0) to signal the end
        write_pair(outFile, STOP_CODE, 0, bitLength(next_code))
        #flush pairs
        flush_pairs(outFile)
        #close both files
        close(inFile)
        close(outFile)
        return
```

DECODE.C

Contains the main() function for the decode program.

main(int argc, char *const argv[])

This is the main function.

```
Pseudocode
def main(argc, argv[]):
    user_input = input("menu_message")
    if user_input == a:
        optAscii++
    elif user_input == i:
        inPath = strdup(option_argument)
        # assume it was split from user_input
    elif user_input == o:
        outPath = strdup(option_argument)
        break
    elif user_input == v:
        optVerbose++
        break
    elif user_input == h:
```

```
        print("menu_message")
        return
    else:
        print("menu_message")
        return
    #open infile
    inFile = open(inPath)
    #file header
    FileHeader outFileHeader = {0xBAADBAAC, inFileStats.st_mode}
    #create WordTable
    wordTable = wt_create();
    #write header
    write_header(outFile, &outFileHeader)
    #keep track of the current code and next code
    curr_node = NULL
    next_code = START_CODE
    #read all pairs from infile
    curr_sym = NULL
    while read_pair(inFile, &curr_code, &curr_sym, \
        bitLength(next_code)):
        wordTable[next_code] = \
            word_append_sym(wordTable[curr_code], curr_sym)
        write_word(outfile, wordTable[next_code])
        if (++next_code == MAX_CODE):
            next_code = START_CODE
            wt_reset(wordTable)
    #flush
    flush_words(outFile)
    #close both files
    close(inFile)
    close(outFile)
```

WORKS CITED:

Assignment document provided by Professor.