Michael Nguyen
mnguy181@ucsc.edu
3/04/2021

CSE13S Winter 2021
Assignment 7: Lempel-Ziv Compression
Design Document

PURPOSE

The purpose of this programming assignment is to learn about data compression. Compressed data yields many benefits due to its reduced size since it allows data to be transferred faster and utilizes a lesser amount of storage space. In this program, we will be programming a lossless data compression algorithm, the LZ78 compression and decompression algorithms. In other words, this program will be split into two programs called encode.c (performs LZ78 compression) and decode.c (that performs the LZ78 decompression). The requirements of these programs as listed in the lab document are as follows:

1. Encode should be able to compress any file, text or binary.
2. Decode should be able to decompress any file, text or binary (that was compressed with encode)
3. Both programs can operate on both little and big endian systems.
4. Both programs use variable bit-length codes.
5. Both perform read and write operations in efficient blocks of 4 kilobytes.
6. Both encode and decode must interoperate with provided binaries - not just my code.

In order to successfully complete this assignment, the program will also need to include several new ADTs, tries ADT and another ADT for words. Furthermore, the program will also contain a source file for the I/O module and a header file for the endianness module. The general idea on how to implement these modules and ADTs will be included later in the Descriptions portion of my design document. The encode and decode programs should support several different command-line options including:

Encode:

-v: Print compression statistics to stderr
-i <input>: specifies input to compress (by default is stdin)
-o <output>: specifies output of compressed input (by default is stdout)

Decode:

-v: Print decompression statistics to stderr
-i <input>: specifies input to decompress (by default is stdin)
-o <output>: specifies output of decompressed input (by default is stdout)

These two programs will print out informative statistics about the compression or decompression that is performed including the compressed file size, uncompressed file size, and the amount of space saved (in which the formula for how to calculate has been provided on the lab document as 100*(1 - [compressed size/uncompressed size])).

Please do note that for this assignment, I will be in charge of implementing encode.c, decode.c, trie.c (the source file for the Trie ADT), word.c (source file for the Word ADT), and io.c (the source file for the I/O module). The header files, trie.h, word.h, io.h, endian.h, and code.h have

been provided for me in the CSE13s resources repository along with example binaries. In addition, pseudocode for compression and decompression have been provided in the Appendix portion of the lab document.

DESCRIPTION

To begin this programming assignment, TA Eugene recommended that I implement the program in the following order: I/O, Trie ADT, Word ADT, then the encode and decode files. Thus, the description portion of this design document will also be done in this order. The reasoning is that the I/O module is by far the hardest portion of this assignment, and thus, I should spend more time testing this module to ensure I have it done properly before moving onto the ADTs.

To ensure that encode and decode perform efficient I/O, I will be required to implement an I/O module in io.c. In this module, it is specified that reads and writes will be done 4 kilobytes or blocks at a time requiring the use of buffer I/O. As defined in the lab document, buffering means to store data into a buffer or an array of bytes.

The file header contains the magic number for our program (defined as 0xBAADBAAC) and the protection bit mask for the original file. The magic number field, magic, serves as an identifier for compressed files and is used by decode to decompress files with the correct magic number. Thus, the struct definition is as follows as supplied in the lab document:

```
Struct FileHeader {
        Uint32_t magic;
        Uint16_t protection;
};
```

Below is the pseudocode that will be used to implement the functions for the I/O module. Please do note that this pseudocode was developed with the help of TA Eugene from his lab section on 3/02/2021 along with TA Gabe's lab section on 3/03/2021. While the pseudocode is not perfect, it is my general thought process on how I would approach each function.

```
Int read_bytes(int infile, uint8_t *buf, int to_read)
        Initialize an int num_bytes variable to keep track of the number of bytes read
        While loop() that ends when num_bytes is >= to_read OR when read() returns a 0
                Set num_bytes equal to the amount of bytes read by read()
                Subtract num_bytes from to_read
                Call read() on the index of the *buf at num_bytes (indicating the starting point)
        Return the number of bytes read

Int write_bytes(int outfile, uint8_t *buf, int to_write)
        Initialize an int num_bytes variable to keep track of number of bytes written
        While loop() that ends when num_bytes is >= to_write OR when write() returns a 0
                Set num_bytes equal to the amount of bytes written by write()
                Subtract num_bytes from to_write
                Call write() on the index of the *buf at num_bytes (indicating the starting point)
        Return the number of bytes written
```

Void read_header(int infile, FileHeader *header)
    If the header.magic is != 0xBAADBAAC
        Return
    If (isBig()) ← checks if the endianness isn't little endian
        swap32() the header's magic int
        swap16() the header's protection mask
    Call read() on the infile, sizeof(FileHeader)

Void write_header(int outfile, FileHeader *header)
    If (isBig())
        swap32() the header's magic int
        swap16() the header's protection mask
    Call write() on the outfile, sizeof(fileheader)

Bool read_sym(int infile, uint8_t *sym)
    Initialize a count variable to keep track of the index of the currently read symbol
    If the count is 0
        Call read_bytes on sym and BLOCK
    Sym is set to symbols [BLOCK]
    Increment count variable
    Set a temp variable equal to the number of bytes read from read_bytes(symbol, BLOCK)
    If the temp variable is less than the BLOCK
        Add 1 to the end of the buffer
    If the temp variable is equal to count
        Return false
    Return true

Void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)
    For loop() that iterates from 0 to the bitlen, incrementing by 1
        If the index of the bit is equal to BLOCK + 8
            write_bytes() buffer array and BLOCK
        Check if the index of the bit of the code set
            If it isn't, set the bit of index bit
        Else
            Clear the bit
    For loop() that iterates from 0 to 8 in the sym array
        If the index of the bit of the sym is not set
            Set the bit of the sym array
        Else
            Clear the bit of the sym array

Void flush_pairs(int outfile)
    Call write_pair() on any of the remaining pairs of symbols and codes

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

```
        For loop() that iterates from 0 to the bitlen, incrementing by 1
                If the index of the bit is equal to BLOCk + 8
                        read_bytes() buffer array and BLOCK
                Check if the index of the bit of the code is set
                        Set the bit of the index bit
                Else
                        Clear the bit
        For loop() that iterates from 0 to 8 in the sym array
                If the index of the bit of the sym is not set
                        Set the bit of the sym array
                Else
                        Clear the bit of the sym array

Void write_word(int outfile Word *w)
        Call write_pair() on the buffer
        If the buffer is filled
                Print the buffer and write to outfile

Void flush_words (int outfile)
        Call write_pairs() on any remaining symbols in the buffer to the outfile
```

The following pseudocode showcases how I would approach the WordTable ADT:
The struct definition of the word table is as follows as defined in the lab document:

```
Struct Word {
        Uint8_t *syms;
        Uint32_t len;
};
Define an array of Words as a WordTable

Word *word_create(uint8_t *syms, uint32_t len)
        Allocate memory using calloc() for a Word struct
        Check if calloc() failed
                Return Null
        Set the Word's syms to syms
        Set the Word's len to len
        Return the pointer to word

Word *word_append_sym(Word *w, uint8_t sym)
        Call word_create() on w.sym + sym, len
        Return the new word pointer

Void word_delete(Word *w)
        Free the pointer
        Set the pointer to null
```

```
WordTable *wt_create(void)
    Initialize word * wordtable, size of MAX_CODE (uint16_max -1)
    Initialize at index 0, empty_code

Void wt_reset(WordTable *wt)
    For loop that iterates through the wordtable array wt
        Call word_delete on each word
```

Below contains my implementation and how I would approach the Trie ADT. Please do note that this pseudocode was developed with the help of TA Eugene during his lab section.
The struct definition of the Trie ADT is as follows as defined in the lab document:

```
Struct TrieNode {
    TrieNode *children[ALPHABET]
    Uint16_t code
}

TrieNode *trie_node_create(uint16_t code)
    Dynamically allocate memory for the struct TrieNode using calloc()
    Set the trienode's code to code
    For loop that iterates through each child node
        Set them to null
    Return a pointer to the trie_node

Void trie_node_delete(TrieNode *n)
    Free the pointer to the node
    Set to null

TrieNode *trie_create(void)
    Initialize a trie using trie_node_create with empty_code
    Return the trieNode pointer
    Otherwise
        Return null

Void trie_reset(TrieNode *root)
    trie_delete() called on the root

Void trie_delete(TrieNode *n)
    For node c in n.children
        trie_delete(c)
        Set c to null
    trie_node_delete(n)

TrieNode *trie_step(TrieNode *n, uint8_t sym)
    For loop that iterates through n's children nodes
```

```
            If the node.code = sym
                    Return the node
    Return null
```

For the encode and decode main() functions, pseudocode has been supplied in the Appendix portion of the lab document, so I felt that there was no need to include it in my design document.