William Santosa
wsantosa@ucsc.edu
14 March 2021

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

DESCRIPTION

Storing large amounts of data has always been a problem for Computer Scientists. IBM's report on Big Data in 2011 states that "we create 2.5 quintillion bytes of data" every day and that number has only increased since then. As such, it is imperative that as up-and-coming Computer Scientists we learn methods of compressing data in order to more quickly transfer and easily store bits, which results in freeing up storage and increasing network capacity. There are two categories of data compression algorithms: lossy and lossless. The former loses some data but retains most of it, and the latter doesn't lose any. In this lab, we aim to implement some of these algorithms in C ourselves.

Files:

- code.h
  - Header file containing macros for Reserved Codes
- endian.h
  - Header file for Endianness module
- io.{c, h}
  - Source and header file for I/O ADT
- word.{c, h}
  - Source and header file for Word ADT
- trie.{c, h}
  - Source and header file for Trie ADT
- decode.c
  - Contains main() for Decode program
- encode.c
  - Contains main() for Encode program
- Makefile
  - Runs program and creates two executable files called Encode and Decode
- README.md
  - Information about building, running, and options of the program
- DESIGN.pdf
  - Describes purpose, covers the layout, clear description of program parts, pseudo code, and contains the pre lab questions.

<u>TOP LEVEL DESIGN / PSEUDOCODE</u>

<u>trie.c</u>

```
#define ALPHABET 256

struct TrieNode {
   TrieNode *children[ALPHABET]; // Array of children TrieNode pointers
   uint16_t code;
};

TrieNode *trie_node_create(uint16_t code){
   Initialize TrieNode TN with malloc
   Initialize children with malloc
   Set TN code to code
   Return TN
}

void trie_node_delete(TrieNode *n){
   Check NULL
   Free children
   Free TN
   Free pointer
}

TrieNode *trie_create(void){
   Initialize Trienode TN with trie_node_create(EMPTY_CODE)
   Return TN
}

void trie_reset(TrieNode *root){
   Just use trie_delete on all of root's children
}

void trie_delete(TrieNode *n){
   Save node n
   Parse children till null:
      Use trie_node_delete to delete
      Move back up
      Continues until n
   Trie_node_delete n
}

TrieNode *trie_step(TrieNode *n, uint8_t sym){
   Convert sym to uint16 number
   Initialize TrieNode TN to NULL
   Parse through TN's children:
      If found, set TN to node
   Return NULL
}
```

<u>word.c</u>

```
struct Word {
    uint8_t *syms;
    uint32_t len;
};

typedef Word *WordTable; // Array of Words

Word *word_create(uint8_t *syms, uint32_t len){
    Initialize word w;
    Set w syms to sysm
    Set w len to len
    Return WT
}

Word *word_append_sym(Word *w, uint8_t sym){
    Add sym to word w
    Return result
}

void word_delete(Word *w){
    Free syms
    Free word
    Set w to null
}

WordTable *wt_create(void){
    Initialize WorldTable WT with malloc of size MAX_CODE;
    Index 0 of WT W set to EMPTY_CODE
    Return WT
}

void wt_reset(WordTable *wt){
    Set everything in WT to null
    Set index 0 to EMPTY_CODE
}
```

```
void wt_delete(WordTable *wt){
   For loop through all WT and delete using word_delete
}
```

<u>io.c</u>

```
#define BLOCK 4096 // 4KB blocks.
#define MAGIC 0xBAADBAAC // Unique encoder/decoder magic number.

extern uint64_t total_syms; // To count the symbols processed.
extern uint64_t total_bits; // To count the bits processed.

struct FileHeader {
   uint32_t magic;
   uint16_t protection;
};

int read_bytes(int infile, uint8_t *buf, int to_read){
   Initialize bytes, total bytes, and bytes to read
   While loop with read using infile, buf and total, total to read > 0:
      If EOF exit loop
      If bytes == -1 print to stderr
      Increase total bytes by bytes
      Decrease total to read by bytes
}

int write_bytes(int outfile, uint8_t *buf, int to_write){
   Same as above but with write
}

void read_header(int infile, FileHeader *header){
   Use read_bytes
   Check endianness
}

void write_header(int outfile, FileHeader *header){
   Check endianness
   Use write_bytes
}

bool read_sym(int infile, uint8_t *sym){
   Initialize bytes to 0
   If sindex is zero:
        Set bytes to read_bytes
    If bytes < BLOCK
        Set end to bytes plus 1
    Set *sym to sbuf[symbol_index]
    Increment index
    Check for block == sindex
    Check if end and return false if end, true otherwise
}

void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen){
    Same as above but do write
}

void flush_pairs(int outfile){
    If bindex > 0:
        Write bytes
        Set bindex to 0
}

bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen){
    Tempcode to 0
    For loop from 0 to bitlen:
        If bindex ==0;
            Read bytes
        Set and clear bits using pbuf
        Increment bindex
        If end set bindex to 0
    Repeat for sym
}

void write_word(int outfile, Word *w){
    For loop from 0 to w->len:
        Set sbuf sindex to w->syms[i]
        Increment sindex
    If end:
        Write bytes
        Set sindex to 0
}

void flush_words(int outfile){
    Same as flush_pairs
}
```

<u>encode.c</u>

Lab Document

<u>decode.c</u>

Lab Document

NOTE : Will definitely be altered (Listed below)

<u>DESIGN PROCESS / MODIFICATIONS</u>

- Created helpers functions to get and set bits
  - Get, set, clear bits
  - 8 bit, 16 bit, and for the buffer as well
    - Modded everything by 8 bit = WRONG
    - Mod by the amount of bits it has
- Had to cast in encode.c because header was not type uint8_t
- Helper function to get amount of bytes needed to represent a number of bits
- Helper function to get the numbers of bits required to represent a number (bit-length)
- Could write to outfile using flush_words, less code needed that way
-