Zhuo Xiao
CSE13S-Fall 2020, UCSC

Assignment 7
Lemple-Ziv Compression

Design

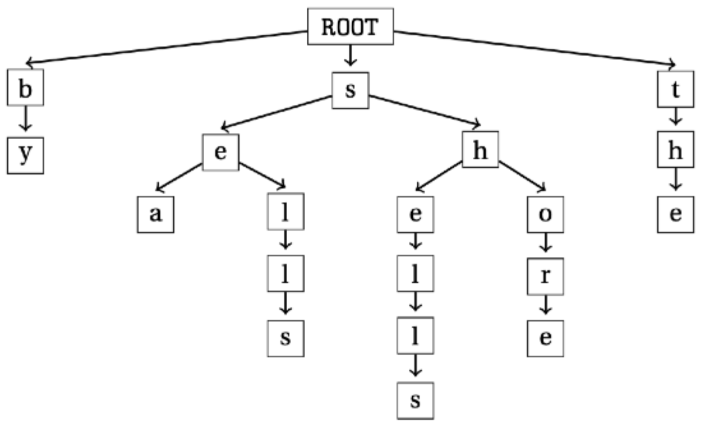
In the program, I have constructed two C program solutions working for compressing and decompressing data. I utilize Lemple-Ziv Compression this time, in order to represent repeated patterns in data with using pairs which are each comprised of a code and a symbol while a code is an unsigned 16-bit integer and a symbol is a 8-bit ASCII character.

In order to achieve this goal, I used I/O, Tries, and WordTable.

Pseudocode are given below:

1. *trie.c:*

   The header file of trie.c is provided by Professor Max Dunne and all CSE 13S staffs. There are 6 functions inside: trie_node_create(), trie_node_delete(), trie_create(), trie_reset(), trie_delete() and trie_step(). The diagram below is what I derived from the lab manual, and shows how graphically trie looks like.



```
trie_node_create(index) :

        tr = malloc(TrieNode)

        tr->code = index

        return tr


trie_create() :

        n = trie_node_create(EMPTY_CODE)

        return n
```

```
trie_node_delete(TrieNode *n) :

        free(n)


trie_reset(TrieNode *root) :

        for i in range(ALPHABET) :

                trie_delete(node->children[i])


tire_delete(TireNode *n):

        trie_reset(n);

        trie_node_delete(n)

trie_step(TireNode *n, uint8_t sym):

        TireNode *n = n->children[sym]

        return n
```

2. *word.c:*

   The header file of trie.c is also provided by Professor Max Dunne and all CSE 13S staffs. This is a look-up table. There are 6 functions inside: word_create(uint8_t *sym, uint32_t len), word_append_sym(Word* w, uint8_t sym), word_delete(Word *w), wt_create(), wt_reset(WordTable *wt) and wt_delete(WordTable *wt).

```
word_create(uint8_t *sym, uint32_t len):

        word = malloc(Word)
        word.sym = malloc(uint8_t * len)

        return word;

word_append_sym(Word* w, uint8_t sym):

        word = malloc(Word)
        word.sym = MAKE_COPY(word.symbol)

        word.len = w.len + 1                        //len + 1 for append wt

        append(new_word.symbols, symbol)            //append the word
        return word
```

```
word_delete(Word *w):
    free(w.sym)
    free(w)

wt_create():
    wt = malloc(MAX_SIZE * WordTable)
    wt[EMPTY_CODE] = word_create(NULL, 0)
    return wt

wt_reset(WordTable *wt):
    for i in range(START_CODE, MAX_SIZE):
        word_delete(wt[i])

wt_delete(WordTable *wt):
    reset(wt)
    delete(wt.head)
    free(wt)
```

3. *io.c:*

   This is one of the most difficult part of this assignment since there is much more conceptually tough. In this program, reads and writes will be done *4KB*, or a block. The magic number field, magic, serves as a unique identifier for files compressed by encode. decode should only be able to decompress files which have the correct magic number. This magic number is *0x8badbeef*. *read_header()* reads in the header and verifies the magic number. At the meanwhile, All reads and writes in this program are done by using the system calls read() and write(). Also, I have utilized two uint8_t* (size of 4096) as buffers for sym and bit.

```
read_bytes(int infile, uint8_t *buf, int to_read):
    while(read_len < to_read)        //still no enough
        read_len + read(infile, buf, still_need_bit)
        read_len += len;
    return read_len

write_bytes(int infile, FileHeader *header):
    while(read_len < to_read)        //still no enough
        read_len + write(infile, buf, still_need_bit)
        read_len += len;
    return read_len

read_header(int infile, FileHeader * header):
    len = read_bytes(infile, header, sizeof FileHeader)
    return if len != sizeof FileHeader

write_header(int infile, FileHeader * header):
    len = write_bytes(infile, header, sizeof FileHeader)
    return if len != sizeof FileHeader

bool read_sym(int infile, uint8_t *sym):
    if sym_buffer != EMPTY
        symbol = symbol_buffer[sym_index]
        return true
```

(continued in next page)

```
            total_sym = read_bytes(4096)
           if(total_sym > 0):
                sym_index = 0;
                return true;
        return false;


void buffer_pair(int outfile, uint16_t code, uint8_t sym, uint8_t bit_len):
        len = pack(symbol, code) // pack data to bits buffer
        pack(bit_buffer, len)
        return if all packed
        write_bytes(file, bit_buffer, 4096)
        pack(bits_buffer, len)


void flush_pair(int outfile):
        if no bit:
                return;
        if bit_buffer != EMPTY:
                write_bytes(file, bits_buffer, (bit_index -1) /8 + 1)


bool read_pair(int outfile, uint16_t code, uint8_t sym, uint8_t bit_len)
        pair = unpack(bit_buffer)
        return pair if unpack
        bits_buffer = read_bytes(outfile, 4096)
        return NULL if bit_buffer == EMPTY
        rp = unpack(bit_buffer)
        return rp
```

```
void buffer_word(int outfile, Word *w):

    buffer(symbol_buffer, word.sym)

    return if index < remain_bytes(full)
    write_bytes(outfile, sym_buffer, 4096)

    return if bytes != BLOCK
    buffer(sym_buffer, word.sym)

    memset()


void flush_word(int outfile):

    if sym_buffer != EMPTY

        write_bytes(outfile, sym_buffer, remain_bytes)

        return if bytes != BLOCK

    return
```

Design Process:

- Ahead of all, based on pseudocode given by CSE13S staffs, all informations are straight and clear. However, since the slight delay from the instruction, there is little hard to fully comprehend the concept of this project. But the situation becomes much better after the second week's lectures and sections.
- Tries and word.c are slight straight forward and easily to capture after I was told and gain some instruction code from Piazza by TAs, which helps a lot.
- But at a later time, I was stuck in the io.c, which is obviously the most difficult part in the course if I could say like this. Gaining ideas from piazza's post and some online tutorials helps me a lot.
- And finally, there is no memory leak in UNIX as I found so far, but there is only one weird occur when I ran the command *make infer* and that is in the given codes for decode.c.