Assignment 7:

Through this assignment, we will be able to implement two programs: encode and decode, that are responsible for the LZ78 compression and decompression, respectively. Compression uses the trie ADT whereas decompression uses word tables; decompression is only done if the input file is compressed with this program or another one that uses the same "magic" number to encode it.

Pseudocode:

IO: -> referenced Eugene and Sahiti's pseudocode
bytes (helper function):
    gets bytes of bits passed in
read_bytes:
    while read still has bytes to read
        read from infile into buffer
write_bytes:
    while write still has bytes to write from buf
        write to the outfile
read_header:
    reads in sizeof Fileheader bytes from infile and stores in header pointer
    checks endianness
        swaps endianness for magic and protection if !little endian
write_header:
    checks endianness
        swaps endianness for magic and protection if !little endian
    writes size of FileHeader bytes to outfile from header pointer
read_sym:
    read block size of bytes from infile into symbuf until eof
        if one full block is not read at once->update end of buffer
write_pair:
    for code (& bitlen(next_code)) passed into function
        set corresponding bit in bit buffer if == 1
        else clear bit
    if bitbuf == full(block*8)
        write block size of bit buffer to outfile
    for sym passed in
        set next eight bits in bit buffer following code that was set if == 1
        else clear bit
    if bitbuf == full(block*8)

write block size of bit buffer to outfile

flush_pair:
if bit index != 0
write bit index number of bytes from bit buffer to outfile
read_pair:
if bit buffer is empty
read in block number of bytes
for bitlen of code
if the bit is set in bit buffer == 1
set in respective bit in temp code
else
clear bit in temp code
if bit index == block *8
reset bit buffer
set pointer code to temp code
do same process for symbols
check endianness-> if !little -> swap
if temp code = STOP_CODE
return false because no more pairs left
else-> true
write_word:
write block size of sym buffer to outfile
flush_words:
if sym_index is !0
write sym index bytes to outfile

TRIE: ->referenced Sahiti's pseudocode
trie_node _create:
calloc for sizeof trie node
zero out all children of trie node
set code passed in to trie node->code
return pointer to node
trie_node_delete:
free(node)
trie_create:
create root node of EMPTY_CODE
return root
trie_reset:
delete all children of root node

retain root node

trie_delete:
delete all children of root node + root node
trie_step:
returns trie node representing sym passed in if it exists

WORD: ->referred to Gabriel and Sahiti's pseudocode
word_create:
calloc sizeof word
syms = calloc len, sizeof u8
memcpy
set len to word->len
word_append_sym:
reallocs mem for new word from word
appends sym passed in to new word->syms[length]
returns new word
word_delete:
frees syms
frees word
wt_create:
calloc MAX_CODE sizeof Word* //bc this means word table
initialize table[EMPTY_CODE] = empty string
return table
wt_reset:
free all indices of table EXCEPT table[EMPTY_CODE]
set indices to NULL
wt_delete:
free all indices of table
set indices to NULL
free table
set table to NULL

ENCODE: —> from asgn doc & Eugene's pseudocode
Compress(infile, outfile)

getopt options(i:o:v)
verbose -> calculations

file permissions

```
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) == true)
//while there are still symbols to be read from the file
        next_node = trie_step(curr_node, curr_sym)
        //sets next node to pointer to child node representing symbol
        if next_node !NULL
        //if next node is not NULL(symbol exists)
                prev_node = curr_node
                //set previous node to current node
                curr_node = next_node
                //set current node to next node
        else //if next node is NULL(symbol does not exist -> trie_step returns NULL)
                write_pair(outfile, curr_node.code, curr_sym, bit-length(next_code))
                //writes new pair for next node to outfile
                curr_node.children[curr_sym] = trie_node_create(next_code)
                //indexing children of curr node with curr_sym to create next code
                curr_node = root
                next_code = next_code+1
        if next_code is MAX_CODE
        //if next_code == MAX_CODE
                trie_reset(root)
                //resets trie to just root
                curr_node = root
                //reset curr_node to root
                next_code = START_CODE
                //reset next code to START_CODE
        prev_sym = curr_sym
if curr_node != root
        write_pair(outfile, prev_node.code, prev_sym, bit-length(next_code))
        //writes new pair to outfile if curr node != root
        next_code = (next_code+1)%MAX_CODE
        //increments next_code while staying within limit of MAX_CODE
write_pair(outfile, STOP_CODE, 0, bit-length(next_code))
```

```
//writes pair (STOP_CODE, 0) to signify end of compressed output
flush_pairs(outfile)
//writes out any remaining pairs of symbols or code to outfile
```

DECODE: —> from asgn doc and Eugene's pseudocode
Decompress(infile, outfile)

```
getopt options i:o:v
check magic number from header
        if != magic number
                break->stderr message
table = wt_create()
//creates a new word table
curr_sym = 0
curr_code = 0
next_code = START_CODE
while (read_pair(infile, &curr_code, &curr_sym, bit-length(next_code)) == true)
//while there are still pairs to be read from infile
        table[next_code] = word_append_sym(table[curr_node], curr_sym)
        //appends new word with symbol to table at the index next code
        write_code(outfile, table[next_code])
        //writes code to table at index next_code
        next_code = next_code + 1
        //increments next_code
        if next_code is MAX_CODE
        //if next code hits MAX_CODE
                wt_reset(table)
                //resets word table
                next_code = START_CODE
                //resets next code to START_CODE
        flush_words(outfile)
        //writes out any remaining symbols in buffer to outfile
```