DESIGN FOR ASGN7
Lempel-Ziv Compression

TASK:
Implement two programs called encode and decode which performs LZ78 compression and decompression, respectively.

Encode should compress any file, text, or binary.
Decode can decompress any file, text, or binary that was compressed by encode.

Both should compress/decompress using variable bit-length codes.

Both should perform read and writes in efficient blocks of 4KB.

Use the Trie ADT to store words during the compression process using codes.

Use the Word ADT to loop up quick code to word translations using a word table.

Use an I/O module to perform efficient reads and writes of code pairs, bytes, etc. in 4KB blocks.
Account for the file header of a file so that the magic number (0x8badbeef) serves as an identifier for files compressed using encode.
Check for the identifier in the decoding process.

Keep tracks of symbols read for the size of the uncompressed file and the bits written for the compressed file size.
Print them if statistics are specified.

Pseudocode

```
trie_node_create(code)
  malloc memory for a trienode
  set its code to input code
  set all 256 children to null
  return the trienode

trie_node_delete(node)
  free the specific input node

trie_create()
  create the root node with empty code
  using trie_node_create
  return root node

trie_reset(root node)
  iterate through all the children of the root node,
  including their children (recursion with trie_delete) and delete them
  as you set them back to the null

trie_delete(node)
  iterate through all the children of the node, including their
  children (recursion) and delete them as you set them back to null
  afterwards, trie_node_delete input node

trie_step(node, sym)
  return the specific symbol in a node's children array

tn_print(node)
  iterate through the node's children and print each character
  that is not null
```

```
word_create(syms, len)
  malloc memory for word
  check if there is a nulls syms, if so, return an empty word with 0 length

  otherwise, malloc memory for syms, assign the length, then set the input
  syms to the word's syms

  return the word

word_append_sym(word, sym)
  malloc a new word and set its length to one more than input word's

  copy the memory from the input's syms to the new word's syms

  set the last sym of the new word's syms to the input sym

  return the new word

word_delete(word)
  free the syms and the word itself

wt_create()
  calloc memory for an array of words (word table)

  set the first word in the word table as an empty word

  return the word table

wt_reset(word table)
  iterate through the wt and call word_delete on each word
  set each word to null

wt_delete(word table)
  iterate through the wt and call word_delete on each word, setting them to null
  free the wt at the end
```

Pseudocode

```
read_bytes(infile, buffer, bytes to read)
  while there are still bytes to be read and number of bytes read does not equal the number to read
    set bytes read to number of bytes read by read()
    increment total bytes read by bytes_read
    decrement cnt by bytes_read

  return total

write_bytes(outfile, buffer, bytes to write)
  while are still bytes to write and the total does not equal the bytes to write
    set bytes written to number of bytes written by write()
    increment total bytes written by bytes_written
    decrement cnt by bytes_written

  return total

read_header(infile, fileheader)
  read_bytes on file header

write_header(outfile, fileheader)
  write_bytes on file header
  increment total_bits by number of bytes in header

read_sym(infile, sym)
  if at first index of sym buffer.
    set end to number of bytes read by read_bytes on sym buff

  set the input sym to the sym in sym buffer
  increment sym index and total_syms

  if sym index is 4096
    return true
  else,
    if sym index is equal to end + 1
      return false
    else,
      return true

buffer_pair(outfile, code, sym, bit length)
  increment total bits by 8 + bit length

  buffer the bits of the symbol from LSB, counting for variable bit length

  then buffer the bits of the index

  when buffer is full, write it to the outfile using write_bytes

  set bit index to 0

flush_pairs(outfile)
  if not the first bit index

  if bit is divisible 8, divide them by 8 to get n_bytes
  else, round up n_bytes

  write_bytes(outfile, bit buffer, n_bytes)

read_pair(infile, code, sym, bit length)
  read in the bits of the symbol, counting for variable bit length

  read in the bits of the index

  if the bit index is 0, read_bytes of the infile to the bit buffer in 4KB blocks

  return true if the current code is not the last one (STOP_CODE)

buffer_word(outfile, word)
  increment the total syms by the length of the word

  iterating through the sym buffer, set it equal to the syms of the input word

  when sym index is 4096, write_bytes to outfile from sym buffer in 4KB block
  then set sym index back to 0

flush_words(outfile)
  if sym index is not 0,
    write_bytes to outfile from sym buffer with leftover syms
```

Pseudocode

Header file for code.h and LZ78 compression/decompression algorithms were provided by Asgn7 lab manual

```c
#ifndef __CODE_H__
#define __CODE_H__

#include <inttypes.h>

#define STOP_CODE   0               // Signals end of decoding/decoding.
#define EMPTY_CODE  1               // Code denoting the empty Word.
#define START_CODE  2               // Starting code of new Words.
#define MAX_CODE    UINT16_MAX      // Maximum code.

#endif
```

Compression Algorithm

```
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) is TRUE
    next_node = TRIE_STEP(curr_node, curr_sym)
    if next_node is not NULL
        prev_node = curr_node
        curr_node = next_node
    else
        BUFFER_PAIR(outfile, curr_node.code, curr_sym, BIT-LENGTH(next_code))
        curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
        curr_node = root
        next_code = next_code + 1
    if next_code is MAX_CODE
        TRIE_RESET(root)
        curr_node = root
        next_code = START_CODE
    prev_sym = curr_sym
if curr_node is not root
    BUFFER_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code))
    next_code = (next_code + 1) % MAX_CODE
BUFFER_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
FLUSH_PAIRS(outfile)
```

Decompression Algorithm

```
table = WT_CREATE()
curr_sym = 0
curr_code = 0
next_code = START_CODE
while READ_PAIR(infile, &curr_code, &curr_sym, BIT-LENGTH(next_code)) is TR
    table[next_code] = WORD_APPEND_SYM(table[curr_code], curr_sym)
    buffer_word(outfile, table[next_code])
    next_code = next_code + 1
    if next_code is MAX_CODE
        WT_RESET(table)
        next_code = START_CODE
FLUSH_WORDS(outfile)
```

Pseudocode

ENCODE

```
int main(command line arguments)
  while getopt parses through arguments
    if -v, print statistics
    if -i, set input file name to optarg (default STDIN)
    if -o, set output file name to optarg (default STDOUT)

  allocate memory for file header, setting it with the appropriate magic number
  and protection

  write the header to the outfile

  call the LZ78 compression algorithm on the input file
    will create trie struct as it buffers pairs of code and syms to
    outfile

  close any files that were opened and free trie and file header structs

  if stats were specified, print compressed file size (total_bits / 8), uncompressed
  file size (total_syms), and compression ratio

  return 0;
```

DECODE

```
int main(command line arguments)
  while getopt parses through arguments
    if -v, print statistics
    if -i, set input file name to optarg (default STDIN)
    if -o, set output file name to optarg (default STDOUT)

  allocate memory for file header, setting it with the appropriate magic number
  and protection

  write the header to the outfile

  call the LZ78 decompression algorithm on the input file
    will create a word table struct to read pairs of codes and syms as it translates
    and writes appropriate characters to outfile

  close any files that were opened and free word table and file header structs

  if stats were specified, print compressed file size (total_bits / 8), uncompressed
  file size (total_syms), and compression ratio

  return 0;
```

Design Process:

The TrieNode and Word ADTs were easy to implement as they had several similarities to our linked list ADT from our previous lab. I was able to use many of same concepts regarding pointers and memory allocation to successfully create functions and change the structs after creating them. I ran into a problem both the trie and word structs would hold random data after calling the delete and reset functions for both structs, but I realized after deleting them, I needed to set them back to NULL to fully 0 them out, much like how calloc does when you allocate some memory.

The hardest part for me when I was creating the functions in io.c was how to manage the variable bit lengths. After some explanation and pseudocode in lab sections, I realized I had to manipulate the bits when managing the code and symbol pairs, making sure I handle them in a specific order as I read or write each one.

Both algorithms were relatively easy to implement in C and worked the very first time during my initial tests.

I realized we shouldn't open the test encode and decode files with vim as it would slightly change its contents; instead, I learned that I should be using xxd to view its data and then cmp them between the example files and mine.

Sources:

Compression/Decompression algorithms were provided by Asgn7 lab manual

Functions in io.c were derived from pseudocode in lab sections (TA: Oran)