TrieNode

TrieNode *trie_node_create(uint16_t code)
    calloc () memory for node
    n->code = code
    Returns node n

Void trie_node_delete(TrieNode *n)
    Frees node n
    Sets n to NULL

TrieNode *trie_create(void)
    Creates the trie which is the first empty code node
    returns trie_node_create(EMPTY_CODE)

Void trie_delete(TrieNode *n)
    Loop through 256 children
    Recursive function to set all children to null
    Returns nothing
    After loop, frees node

Void trie_reset(TrieNode *root)
    Same as trie_delete except does not delete first node
    Only deletes the children

TrieNode *trie_step(TrieNode *n, uint8_t sym)
    Returns the pointer to child representing symbol
    n->children[sym]
    Else returns NULL

Word Tables

Word *word_create(uint8_t *syms, uint32_t len)
    calloc () memory for word
    w->len=len
    calloc() memory for *syms for the input length
    Copies input syms into w->syms
    If it fails return NULL
    Else returns Word *

Word *word_append_sym(Word *w, uint8_t sym)
    Reallocates of syms with len+1
    Returns the word

Void word_delete(Word *w)
    Frees syms
    Sets syms to NULL
    Frees w and sets it to NULL

WordTable *wt_create(void)
    Creates memory for an array of words
    Calloc(MAX_CODE, sizeof(word *))
    Creates a word with input empty_code and length 0
    Returns wt

Void wt_delete(WordTable *wt)
    Loops from empty code to MAX_CODE
    Recursive function to delete everything in wt
    Set each wt[index] to NULL
    After loop frees word table (wt)

Void wt_reset(WordTable *wt)
    Same as wt_delete
    Except start loop at START_CODE to skip deleting EMPTY_CODE
    Don't free wt at the end of loop

I/O

Int read_bytes(int infile, uint8_t *buf, int to_read)
    Helper function to perform reads
    int counter to keep track of how many bytes you have read
    Use the read function to read in bytes
    Keep looping until all specified bytes were read
    Increment the counter to include read bytes

Int write_bytes(int outfile, uint8_t *buf, int to_write)
    Helper function to perform writes
    Same as read function just use the write() function
    Loop until specified bytes written

Void read_header(int infile, FileHeader *header)

Use read_bytes function for up to the sizeof(header)
Use endian function from h file to determine of endian
If not little endian convert the magic number and protection to little endian
Check if the magic number matches

Void write_header(int outfile, FileHeader *header)

Check endianness at beginning and swap if needed
Similar to read_header just use the write_bytes function
Write out the bytes for only the header
So write_bytes(infile, header, sizeof(Fileheader))

Bool read_sym(int infile, uint8_t *sym)

Have a buffer for symbol
Create an index to keep track of block size
Read_bytes for the block_size
Loop until the index equals the block size
If bytes_read != BLOCK
Then increment the end by 1 and loop back

Void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)

Have a buffer to use for write_pair and read_pair
If the code is big_endian then swap it
Loop through the length of width and mask the buffer with input of the bytes
If the bytes are equal to the BLOCK
Then write_bytes and reset the buffer

Repeat the same process for writing symbols

Void flush_pairs(int outfile)

Writes the remaining bytes to the outifle (symbols and codes)

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

Similar to wite_pair
Just use read_bytes instead
Check endianness and swap if necessary

Void write_word(int outfile, Word *w)

Loops through the length of the word

Buff[i] = w->syms[i]

If the index equals the block

Write the bytes for the length of the block

Reset the index

Void flush_words(int outfile)

Use write_bytes to write remaining words from outfile to outfile

For compression and decompression, pseudo code is provided

Use getopt to get command line options