Design Document

1. Trie.c
    a. Trie_node_create:
        i. Input: index
        ii. Output: Trienode
        iii. Function:
            1. Create a node size of trinode.
            2. Set code of nodes to index
            3. Set all children of node to null
    b. Trie_node_delete
        i. Input: Trienode
        ii. Output: void
        iii. Function:
            1. Frees Trienode.
    c. Trie_create:
        i. Input: void
        ii. Output: node
        iii. Function:
            1. Create a node size of trienode
            2. Set all children of node to null
            3. Set the code of node to empty_code

    d. Trie_reset:
        i. Input: root node
        ii. output : voide
        iii. Function:
            1. Recursively frees all children and children of root node.
            2. Keep root node.
    e. Trie_delete:
        i. input : node n
        ii. Output: none
        iii. Function:
            1. Recursively frees all children and children of root node.
    f. Trie_step:
        i. Input: node n and symbol
        ii. Output: return trienode
        iii. Function:
            1. Checks if symbol is any codes of node n children.
            2. Return the node if yes else return null.

2. Word.c

a. Word_Create:
   i. Input: symbol and len
   ii. Output: word
   iii. Function:
      1. Create a word size of word
      2. Create word array of symbols size of len input
      3. Copy symbol to word array
      4. Set word len to input len
b. Word_append_sym:
   i. Input: word w and symbol
   ii. Output: word
   iii. Function:
      1. Create word size of word
      2. Set length of new word table of input wordtable length + 1
      3. Create a word array of symbols of size w-> len + 1;
      4. Copy symbol to word array
      5. Add symbol to word table array
c. Word_delete:
   i. Input: word
   ii. Output: node
   iii. Function:
      1. Deletes word_table
      2. Frees array of word_table.

d. wt_create:
   i. Input: void
   ii. output : wordtable
   iii. Function:
      1. Create word table size of Max_code
      2. Set_table at index 0 to empty string

e. wt_reset:
   i. input : wt
   ii. Output: none
   iii. Function:
      1. frees all words table except first index.
f. wr_delete:
   i. input : wt
   ii. Output: none
   iii. Function:
      1. frees all words table.

3. Io.c

a. read_bytes:
    i. Input: file, buffer, nits to read
    ii. Output: total number of bytes read.
    iii. Function:
        1. Read bytes from file to buffer until buffer is full or file is empty.

b. Write_bytes:
    i. Input: file, buffer, nits to write
    ii. Output: total number of bytes read.
    iii. Function:
        1. Writes bytes from file to buffer until buffer is empty or file is empty.
c. Read_Header:
    i. Input: file to read, header
    ii. output : none
    iii. Function:
        1. Read file header
d. Read_sym:
    i. Input: file to read, bytes
    ii. output : returns true if there are symbols to read
    iii. Function:
        1. Reads a symbol from the input file.
        2. The "read" symbol is placed into the pointer to sym (pass by reference).
        3. In reality, a block of symbols is read into a buffer.
        4. An index keeps track of the currently read symbol in the buffer.
        5. Once all symbols are processed, another block is read.
        6. If less than a block is read, the end of the buffer is updated.
        7. Returns true if there are symbols to be read, false otherwise.
e. Write header:
    i. input : file to write to, header
    ii. Output: none
    iii. Function:
        1. Write file header
f. buffer_pair:
    i. input : file to write to, code, symbol, bit length of code
    ii. Output: node
    iii. Function:
        1. Writes bits from buffer array to output file from lsb to rsb.
        2. Only does when buffer is full.
g. Flush_pair
    i. input : file to write to
    ii. Output: node
    iii. Function:
        1. Empty buffer and write number of bytes left in buffer.

h. Read_pair:
    i. Input: outfile
    ii. Output: True if there are pairs left to read, false otherwise.
    iii. Function:
        1. "Reads" a pair (symbol and index) from the input file.
        2. The "read" symbol is placed in the pointer to sym (pass by reference).
        3. The "read" index is placed in the pointer to index (pass by reference).
        4. In reality, a block of pairs is read into a buffer.
        5. An index keeps track of the current bit in the buffer.
        6. Once all bits have been processed, another block is read.
        7. The first 8 bits of the pair constitute the symbol, starting from the LSB.
        8. The next bit_len bits constitutes the index, starting from the the LSB.
        9. Returns true if there are pairs left to read in the buffer, else false.
        10. There are pairs left to read if the read index is not STOP_INDEX.

i. Buffer_word:
    i. Input: outfile, word
    ii. Output: none
    iii. Function:
        1. Buffers a Word, or more specifically, the symbols of a Word.
        2. Each symbol of the Word is placed into a buffer.
        3. The buffer is written out when it is filled.

j. Flush_words:
    i. Input: outfile
    ii. Output: none
    iii. Functin:
        1. Writes out any remaining symbols in the buffer.

Encode.c

1. Sudo Code provided

1. Open `infile` with `open()`. If an error occurs, print a helpful message and exit with a status code indicating that an error occurred. `infile` should be `stdin` if an input file wasn't specified.
2. The first thing in `outfile` must be the file header, as defined in the file `io.h`. The magic number in the header must be `0x8badbeef`. The file size and the protection bit mask you will obtain using `fstat()`. See the man page on it for details.
3. Open `outfile` using `open()`. The permissions for `outfile` should match the protection bits as set in your file header. Any errors with opening `outfile` should be handled like with `infile`. `outfile` should be `stdout` if an output file wasn't specified.
4. Write the filled out file header to `outfile` using `write_header()`. This means writing out the `struct` itself to the file, as described in the comment block of the function.
5. Create a trie. The trie initially has no children and consists solely of the root. The code stored by this root trie node should be `EMPTY_CODE` to denote the empty word. You will need to make a copy of the root node and use the copy to step through the trie to check for existing prefixes. This root node copy will be referred to as `curr_node`. The reason a copy is needed is that you will eventually need to reset whatever trie node you've stepped to back to the top of the trie, so using a copy lets you use the root node as a base to return to.

6. You will need a monotonic counter to keep track of the next available code. This counter should start at `START_CODE`, as defined in the supplied `code.h` file. The counter should be a `uint16_t` since the codes used are unsigned 16-bit integers. This will be referred to as `next_code`.
7. You will also need two variables to keep track of the previous trie node and previously read symbol. We will refer to these as `prev_node` and `prev_sym`, respectively.
8. Use `read_sym()` in a loop to read in all the symbols from `infile`. Your loop should break when `read_sym()` returns false. For each symbol read in, call it `curr_sym`, perform the following:
   (a) Set `next_node` to be `trie_step(curr_node, curr_sym)`, stepping down from the current node to the currently read symbol.
   (b) If `next_node` is not `NULL`, that means we have seen the current prefix. Set `prev_node` to be `curr_node` and then `curr_node` to be `next_node`.
   (c) Else, since `next_node` is `NULL`, we know we have not encountered the current prefix. We buffer the pair (`curr_node->code`, `curr_sym`), where the bit-length of the buffered code is the bit-length of `next_code`. We now add the current prefix to the trie. Let `curr_node->children[curr_sym]` be a new trie node whose code is `next_code`. Reset `curr_node` to point at the root of the trie and increment the value of `next_code`.
   (d) Check if `next_code` is equal to `MAX_CODE`. If it is, use `trie_reset()` to reset the trie to just having the root node. This reset is necessary since we have a finite number of codes.
   (e) Update `prev_sym` to be `curr_sym`.
9. After processing all the characters in `infile`, check if `curr_node` points to the root trie node. If it does not, it means we were still matching a prefix. Buffer the pair (`prev_node->code`, `prev_sym`). The bit-length of the code buffered should be the bit-length of `next_code`. Make sure to increment `next_code` and that it stays within the limit of `MAX_CODE`. Hint: use the modulo operator.
10. Buffer the pair (`STOP_CODE`, 0) to signal the end of compressed output. Again, the bit-length of code buffered should be the bit-length of `next_code`.
11. Make sure to use `flush_pairs()` to flush any unwritten, buffered pairs.
12. Use `close()` to close `infile` and `outfile`.

Decode.c

7 Decompression

The following steps for decompression will refer to the input file to decompress as `infile` and the uncompressed output file as `outfile`.

1. Open `infile` with `open()`. If an error occurs, print a helpful message and exit with a status code indicating that an error occurred. `infile` should be `stdin` if an input file wasn't specified.
2. Read in the file header with `read_header()`, which also verifies the magic number. If the magic number is verified then decompression is good to go and you now have a header which contains the original protection bit mask.
3. Open `outfile` using `open()`. The permissions for `outfile` should match the protection bits as set in your file header that you just read. Any errors with opening `outfile` should be handled like with `infile`. `outfile` should be `stdout` if an output file wasn't specified.
4. Create a new word table with `wt_create()` and make sure each of its entries are set to `NULL`. Initialize the table to have just the empty word, a word of length 0, at the index `EMPTY_CODE`. We will refer to this table as `table`.
5. You will need two `uint16_t` to keep track of the current code and next code. These will be referred to as `curr_code` and `next_code`, respectively. `next_code` should be initialized as `START_CODE` and functions exactly the same as the monotonic counter used during compression, which was also called `next_code`.
6. Use `read_pair()` in a loop to read all the pairs from `infile`. We will refer to the code and symbol from each read pair as `curr_code` and `curr_sym`, respectively. The bit-length of the code to read is the bit-length of `next_code`. The loop breaks when the code read is `STOP_CODE`. For each read pair, perform the following:
   (a) As seen in the decompression example, we will need to append the read symbol with the word denoted by the read code and add the result to `table` at the index `next_code`. The word denoted by the read code is stored in `table[curr_code]`. We will append `table[curr_code]` and `curr_sym` using `word_append_sym()`.
   (b) Buffer the word that we just constructed and added to the table with `buffer_word()`. This word should have been stored in `table[next_code]`.
   (c) Increment `next_code` and check if it equals `MAX_CODE`. If it has, reset the table using `wt_reset()` and set `next_code` to be `START_CODE`. This mimics the resetting of the trie during compression.
7. Flush any buffered words using `flush_words()`.
8. Close `infile` and `outfile` with `close()`.