Assignment 6 pdf:

Description:

In this assignment, I will be creating an application that will compress text files using Lempel-Ziv 78 algorithm. The way the algorithm works is by having a list that is 2^16-1 units long. These will encode repeating patterns and simplify the number of bits that are required to let say store the ASCII character “a” by giving it a smaller number and then reusing it in the encoding. The table will be created by using a trie which is a tree. There will be two main programs that will encode the program and decode the program. The encoding executable will compress the file and write to a new file, and the decoding executable will decompress it into a human-readable format. Lempel-Ziv 78 is a lossless compression algorithm and therefore the final decompressed file should be indistinguishable from the original file.

Files to include:

- encode.c : contains the main() function for the encode program.
- decode.c : contains the main() function for the decode program.
- trie.c: the source file for the Trie ADT.
- trie.h: the header file for the Trie ADT.
- word.c: the source file for the Word ADT.
- word.h: the header file for the Word ADT.
- io.c: the source file for the I/O module
- io.h: the header file for the I/O module.
- endian.h: the header file for the endianness module.
- code.h: the header file containing macros for reserved codes.

trie.c:

- TrieNode *trie_node_create(uint16_t code)
    - Constructor for a TrieNode. The node’s code is set to code. Make sure each of the children node pointers are NULL.
- void trie_node_delete(TrieNode *n)
    - Destructor for a TrieNode. Note that only a single pointer is passed here. The destructors you have written in the past have taken double pointers in order to NULL the pointer by dereferencing it.
- TrieNode *trie_create(void)
    - Initializes a trie: a root TrieNode with the code EMPTY_CODE. Returns the root, a TrieNode *, if successful, NULL otherwise
- void trie_reset(TrieNode *root)

    - Resets a trie to just the root TrieNode. Since we are working with finite codes, eventually we will arrive at the end of the available codes (MAX_CODE). At that point, we must reset the trie by deleting its children so that we can continue compressing/decompressing the file.
- void trie_delete(TrieNode *n)
    - Deletes a sub-trie starting from the trie rooted at node n. This will require recursive calls on each of n's children. Make sure to set the pointer to the children nodes to NULL after you free them with trie_node_delete().
- TrieNode *trie_step(TrieNode *n, uint8_t sym)
    - Returns a pointer to the child node reprsenting the symbol sym. If the symbol doesn't exist, NULL is returned.

word.c:

- Word *word_create(uint8_t *syms, uint32_t len)
    - Constructor for a word where sysms is the array of symbols a Word represents. The length of the array of symbols is given by len. This function returns a Word * if successful or NULL otherwise.
- Word *word_append_sym(Word *w, uint8_t sym)
    - Constructs a new Word from the specified Word, w, appended with a symbol, sym. The Word specified to append to may be empty. If the above is the case, the new Word should contain only the symbol. Returns the new Word which represents the result of appending
- void word_delete(Word *w)
    - Destructor for a Word, w. Like with trie_node_create() in §4.1.3, a single pointer is used here to reduce the complexity of memory management, thus reducing the chances of having memory-related errors
- WordTable *wt_create(void)
    - Creates a new WordTable, which is an array of Words. A WordTable has a pre-defined size of MAX_CODE, which has the value UINT16_MAX. This is because codes are 16-bit integers. A WordTable is initialized with a single Word at index EMPTY_CODE. This Word represents the empty word, a string of length of zero.
- void wt_reset(WordTable *wt)
    - Resets a WordTable, wt, to contain just the empty Word. Make sure all the other words in the table are NULL

io.c:

- int read_bytes(int infile, uint8_t *buf, int to_read)
    - This will be a useful helper function to perform reads. As you may know, the read() syscall does not always guarantee that it will read all the bytes specified. For example, a call could be issued to read a block of bytes, but it might only

read half a block. So, we write a wrapper function to loop calls to read() until we have either read all the bytes that were specified (to_read), or there are no more bytes to read. The number of bytes that were read are returned

- int write_bytes(int outfile, uint8_t *buf, int to_write)
    - This function is very much the same as read_bytes(), except that it is for looping calls to write(). As you may imagine, write() isn't guaranteed to write out all the specified bytes (to_write), and so we loop until we have either written out all the bytes specified, or no bytes were written. The number of bytes written out is returned.
- void read_header(int infile, FileHeader *header)
    - This reads in sizeof(FileHeader) bytes from the input file. These bytes are read into the supplied header. Endianness is swapped if byte order isn't little endian. Along with reading the header, it must verify the magic number.
- void write_header(int outfile, FileHeader *header)
    - Writes sizeof(FileHeader) bytes to the output file. These bytes are from the supplied header. Endianness is swapped if byte order isn't little endian
- bool read_sym(int infile, uint8_t *sym)
    - An index keeps track of the currently read symbol in the buffer. Once all symbols are processed , another block is read. If less than a block is read , the end of the buffer is updated. Returns true if there are symbols to be read , false otherwise.
- void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)
    - "Writes" a pair to outfile. In reality, the pair is buffered. A pair is comprised of a code and a symbol. The bits of the code are buffered first, starting from the LSB. The bits of the symbol are buffered next, also starting from the LSB. The code buffered has a bit-length of bitlen. The buffer is written out whenever it is filled.
- void flush_pairs(int outfile)
    - Writes out any remaining pairs of symbols and codes to the output file.
- bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen)
    - Reads" a pair (code and symbol) from the input file. The "read" code is placed in the pointer to code (e.g. *code = val) The "read" symbol is placed in the pointer to sym (e.g. *sym = val). In reality, a block of pairs is read into a buffer. An index keeps track of the current bit in the buffer. Once all bits have been processed, another block is read. The first bitlen bits are the code, starting from the LSB. The last 8 bits of the pair are the symbol, starting from the LSB. Returns true if there are pairs left to read in the buffer, else false. There are pairs left to read if the read code is not STOP_CODE.
- void write_word(int outfile, Word *w)
    - "Writes" a pair to the output file. Each symbol of the Word is placed into a buffer. The buffer is written out when it is filled.
- void flush_words(int outfile)
    - Writes out any remaining symbols in the buffer to the outfile. Note that the output file in which you write to must have the same protection bits as the original file. Like in assignment 4, you will make use of fstat() and fchmod(). All reads and writes in this program must be done using the system calls read() and write(),

which means that you must use the system calls open() and close() to get your file descriptors. As stated earlier, all reads and writes must be performed in efficient blocks of 4KB

encode.c :

- 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
- 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 0xBAADBAAC. The file size and the protection bit mask you will obtain using fstat(). See the man page on it for details.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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
    - Set next_node to be trie_step(curr_node, curr_sym), stepping down from the current node to the currently read symbol.
    - 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
    - Else, since next_node is NULL, we know we have not encountered the current prefix. We write the pair (curr_node->code, curr_sym), where the bit-length of the

written 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.
    - 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.
    - Update prev_sym to be curr_sym
- 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. Write the pair (prev_node->code, prev_sym). The bit-length of the code written 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.
- Write the pair (STOP_CODE, 0) to signal the end of compressed output. Again, the bit-length of code written should be the bit-length of next_code.
- Make sure to use flush_pairs() to flush any unwritten, buffered pairs. Remember, calls to write_pair() end up buffering them under the hood. So, we have to remember to flush the contents of our buffer.
- Use close() to close infile and outfile.

decode.c :

- 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.
- 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.
- 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.
- 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.
- 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.
- 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:

    - 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().
    - Write the word that we just constructed and added to the table with write_word(). This word should have been stored in table[next_code].
    - 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.
- Flush any buffered words using flush_words(). Like with write_pair(), write_word() buffers words under the hood, so we have to remember to flush the contents of our buffer.
- Close infile and outfile with close()