LZ78 Compression & Decompression
DESIGN DOCUMENT

CSE 13S, Professor Long

Alexander Ragland, adraglan@ucsc.edu, #1858717

encode.c - Contains main() function for encode program

1. Open infile with open()
    a. Error handle
    b. Stdin default
2. For outfile
    a. Check that first thing is file header (ADT)
    b. Check that magic number in header is 0xBAADBAAC
    c. Store file size and protection bit mask with fstat()
3. Open outfile with open()
    a. Check that permissions and protection bits are the same as file header
    b. Error handle
    c. Stdout default
4. Write filled out file header to outfile with write_header()
    a. Write out the struct itself
5. Do compression

```
Compress(infile, outfile)
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)
        next_node = trie_step(curr_node, curr_sym)
        if next_node != NULL
                prev_node = curr_node
                curr_node = next_node
        else
                write_pair(outfile, curr_node.code, curr_symb, 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 == MAX_CODE
                trie_reset(root)
                curr_node = root
                next_code = start_code
        prev_sym = curr_sym
```

```
if curr_node != root
        write_pair(outfile, prev_node.code, prev_sym, bit-length(next_code))
        next_code = (next_code + 1) % MAX_CODE
write_pair(outfile, STOP_CODE, 0, bit-length(next_code))
flush_pairs(outfile)
close(infile)
close(outfile)
```

decode.c - Contains main() function for decode program

1. Open infile with open
    a. Error handle
    b. Default stdin
2. Read file header with read_header()
    a. If magic number is verified then proceed
3. Open outfile with open()
    a. Protection bits for outfile should match file header you just reader
    b. Error handle
    c. Default stdout
4. Decompress

```
Decompress(infile, outfile)
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))
        table[next_code] = word_append_sym(table[curr_code], curr_sym)
        write_word(outfile, table[next_code])
        next_code = next_code + 1
        if next_code == MAX_CODE
                wt_reset(table)
                next_code = START_CODE
flush_words(outfile)
close(infile)
close(outfile)
```

trie.c - Source file for Trie ADT

- TrieNode ***trie_node_create(**uint16_t **code)**
    - Constructor for TrieNode
    - Each child node pointers set to NULL

- void **trie_node_delete(**TrieNode ***n)**
    - Destructor for TrieNode
    - Single pointer passed here

- TrieNode ***trie_create(void)**
    - Initialize a trie
        - A root TrieNode with the code EMPTY_CODE
        - If successful
            - Return root (TrieNode *)
        - Else
            - Return NULL

- void **trie_reset(**TrieNode ***root)**
    - Reset a trie to the root TrieNode
        - Make sure each of root's children nodes are NULL

- void **trie_delete(**TrieNode ***n)**
    - Deletes a sub-trie starting from the trie rooted at node n
    - Recursively call each of n's children
        - Free child with trie_node_delete()
        - Set children node pointer to NULL

- TrieNode ***trie_step(**TrieNode ***n,** uint8_t **sym)**
    - If symbol exists
        - Return pointer to child node representing the symbol
    - Else
        - Return Null

word.c - Source file for Word ADT

- struct **Word** {uint8_t ***syms;** uint32_t **len;};**

- typedef **Word * WordTable**

- Word ***word_create(**uint8_t ***syms,** uint32_t **len)**
    - Constructor for word where sysms is the array of symbols a Word represents
    - Length of the array of symbols is given by len
    - If successful
        - Return Word *
    - Else
        - Return Null

- Word ***word_append_sym(**Word ***w,** uint8_t **sym)**
    - Constructs a new word from Word w, appended with symbol sym
    - If the Word specified to append is empty
        - The new Word should only contain the symbol
    - Returns the new Word which represents the result of appending

- void **word_delete(**Word ***w)**
    - Destructor for a Word, w
    - Single pointer used here for simplicity

- WordTable ***wt_create(void)**
    - Creates a new WordTable, an array of Words
    - WordTable has size of MAX_CODE (UINT16_MAX)
    - Initialize with a single Word at index EMPTY_CODE
        - This Word represents the empty word (string of length 0)

- void **wt_reset(**WordTable ***wt)**
    - Resets a WordTable, wt, to contain only the empty Word
    - Make sure all other words in table are NULL

io.c - Source file for I/O module

- struct **FileHeader** {uint32_t **magic;** uint16_t **protection;};**

- int **read_bytes(**int **infile,** uint8_t ***buf,** int **to_read)**
    - Helper function to perform reads
    - Loop read() until we have read all specified bytes (to_read) or there are no more bytes to read
    - Return number of bytes read

- int **write_bytes(**int **outfile,** uint8_t ***buf,** int **to_write)**
    - Helper function to perform writes
    - Loop write() until we have written all specified bytes (to_write) or no bytes were written
    - Return number of bytes written

- void **read_header(**int **infile,** FileHeader ***header)**
    - Reads sizeof(FileHeader) bytes from infile
    - These bytes are read into the supplied header
    - Endianness is swapped if byte order isn't little endian
    - Must also verify magic number

- void **write_header(**int **outfile,** FileHeader ***header)**
    - Writes sizeof(FileHeader) bytes to outfile
    - 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)**
    - Index keeps track of currently read symbol in buffer
    - After all symbols are processed, another block is read
    - If less than a block is read
        - End of the buffer is updated
    - If there are symbols to be read
        - Return true
    - Else
        - Return false

- void **write_pair(**int **outfile,** uint16_t **code,** uint8_t **sym,** int **bitlen)**
    - "Writes" a pair to outfile
    - A pair is a code and a symbol

    - The bits of the code are buffered, starting from LSB
    - Then, the bits of the symbol are buffered, starting from LSB
    - The code buffered has a bit-length of bitlen
    - The buffer is written out whenever it is filled

- void **flush_pairs(**int **outfile)**
    - Uses write_pair() for any remaining pairs of symbols and codes to outfile

- bool **read_pair(**int **infile,** uint16_t ***code,** uint8_t ***sym,** int **bitlen)**
    - "Reads" a pair (code and symbol) from infile
    - The "read" code is placed in the pointer to code (ex. *code = val)
    - The "read" symbol is placed in the pointer to sym (ex. *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
        - If there are pairs left to read in buffer (read code is not STOP_CODE)
            - Return true
        - Else
            - Return false

- void **write_word(**int **outfile,** Word ***w)**
    - "Writes" a pair to outfile
    - 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 outfile
    - Uses fstat() and fchmod() to make sure outfile has the same protection bits as the original infile
    - Note that
        - All reads and writes in this program must use read() and write()
        - Must use open() and close() to get file descriptors
        - All reads and writes must use two static 4KB uint8_t arrays as buffers
            - One for binary pairs (have an index/variable to keep track of current byte or bit processed)
            - One for characters (have an index/variable to keep track of current byte or bit processed)