Roleen Ferrer
Professor Maxwell Dunne
CSE 13S - Computer Systems and C Programming

Assignment 7 - Lempel-Ziv Compression

In this lab, I will be implementing Abstract Data Types such as Tries and WordTables, coupled with the usage of I/O to parse through certain files and compress or decompress their sizes. Compressing the data will reduce the number of bits that it is needed to represent it, thus making it faster to transfer and also more storage efficient. I will be using Lempel-Ziv compression in order to compress data, which is an algorithm that reduces data into repeated patterns using pairs which are composed of a code and a symbol. The purpose of this lab is to do as follows:

1. *encode* can compress any file, text, or binary.
2. *Decode* can decompress any file, text, or binary, that was compressed with *encode*.
3. Both will use variable bit-length codes
4. Both will perform read and writes in efficient blocks of 4KB.

Implementation

Main (encode.c)

This source file contains the implementation for the main function of encode.c. It also contains the function compress() to encode a file. Main uses getopt() to receive inputs from the command line. The commands are as follows:

1. -v : display compression statistics
2. -i + infile : Specify which file to compress (default = stdin)
3. -o + outfile : Specify output of compressed infile (default = stdout)

The functions in encode.c are:

- int main(int argc, char **argv)
    - This function is main, which will accept user commands and edit the infile and outfile permissions
    - After taking in commands from user, create a FileHeader struct to store the header
    - Create a stat struct to retrieve the protection from the infile
    - Call fstat() on the infile to obtain the status of protection
    - Set the header's protection

        - header.protection = header_protection.st_mode
    - Call fchmod() to change the mode (protection) of the outfile
    - Call write_header() to write the header to the outfile
    - Call compress(infile, outfile) to compress the infile and write the compressed file to outfile
    - Check if the -v command is flagged, if it is, print the statistics of compression
        - The compressed file size is the total number of symbols that were read in
        - The uncompressed file size is the total number of bits that were buffered and written, including header
        - The compression ratio uses the formula 100 (* (1 - total_syms / total_bits))
    - Close both the infile and outfile, freeing memory
- void compress(int infile, int outfile)
    - This function will compress an infile to outfile. The pseudocode for the design behind this function is in the lab document, but essentially, it follows this flow:
    - Create a TrieNode struct using trie_create()
    - Create other TrieNode structs that will store the current node and the previous node
    - Declare uint8_t variables that will store the current symbol and previous symbol
    - Declare a uint16_t variable that will store the next code at START_CODE (2)
    - Create a while loop that will run while (read_sym(infile, &curr_sym) == true)
        - Create a Trienode struct using trie_step(curr_node, curr_sym)
        - Create a conditional that will check if next_node is NULL
            - If it is, set prev_node to curr_node and curr_node to next_node
        - else,
            - buffer_pair(outfile,curr_node->code,curr_sym,(uint16_t)log2(next_code) + 1)
            - Set curr_node->children[curr_sym] to another TrieNode using trie_node_create (next_code)
            - Set the current node to 1
            - Increment next_code by 1

        - Create a conditional that will check if next_code is MAX_CODE (uint16_t)
            - If it is, call trie_reset() to reset the root
            - Set curr_node to the root
            - Set next_code to START_CODE (2)
    - Create a conditional that will check if curr_node != root
        - If it does, call:
            - buffer_pair(outfile,prev_node->code,prev_sym,(uint16_t)log2(next_code) + 1
            - Set next_code to (next_code + 1) % MAX_CODE
    - Call buffer_pair(outfile,STOP_CODE,0,(uint16_t)log2(next_code)+1
    - Call flush_pairs() to flush the pairs of the outfile
    - Call trie_delete() to free memory from the TrieNode

Main (decode.c)

This source file contains the implementation for the main function of decode.c. It also contains the function decompress() to encode a file. Main uses getopt() to receive inputs from the command line. The commands are as follows:

1. -v : display compression statistics
2. -i + infile : Specify which file to decompress (default = stdin)
3. -o + outfile : Specify output of decompressed infile (default = stdout)

The functions in decode.c are:

- int main(int argc, char **argv)
    - This function is main, which will accept user commands and edit the infile and outfile permissions
    - After taking commands from user create a FileHeader struct to store the header
    - Call read_header(infile, &header) to read the header form the infile
    - Call fchmod(outfile, header.protection) to change the mode of the outfile
    - Call decompress(infile, outfile) to decompress the infile and write the decompressed file to outfile
    - Check if the -v command is flagged, if it is, print the statistics of compression

        - The compressed file size is the total number of bits that were read in, including the header
        - The uncompressed file size is the total number of symbols that were buffered and written out
        - The compression ratio uses the formula 100 (* (1 - total_bits / total_syms))
    - Close both the infile and outfile, freeing memory
- void decompress(int infile, int outfile)
    - This function will decompress an infile to outfile. The pseudocode for the design behind this function is in the lab document, but essentially, it follows this flow:
    - Create a WordTable struct using wt_create()
    - Declare a uint8_t variable that will store the curr_sym
    - Declare uin16_t variables that will store the curr_code at 0 and next_code at START_CODE (2)
    - Create a while loop using read_pair() == true
        - while (read_pair(infile,&curr_code,&curr_sym,(uint16_t)log2(next_code)+1)
        - Set the table[next_code] to word_append_sym(table[curr_code],curr_sym)
        - Call buffer_word(outfile, table[next_code]) to buffer the current word at table indexed at next_code
        - Increment next_code by 1
        - Create a conditional that will check if next_code == MAX_CODE (uint16_t)
            - if it is, call wt_reset() on the created WordTable
            - Set next_code to START_CODE (2)
        - Flush the words from the outfile using flush_words
        - Delete the WordTable to free memory

Tries (trie.c)

This source file contains the Trie ADT. A Trie checks for prefixes of words in a very efficient way, as each node represents a symbol of ASCII characters. Using this type of method will help efficiently compress and store words.

```
struct TrieNode {
  TrieNode *children[ALPHABET];
  uint16_t code;
};
```

The functions that are implemented are as follows:

- TrieNode *trie_node_create(uint16_t code)
    - This function is the constructor for a TrieNode
    - Malloc enough memory in order to properly create the TrieNode
    - Create a for loop that will iterate through its children through the amount of ASCII characters (256) to set them as NULL
    - Set the TrieNode "code" member to code
    - Return the TrieNode that was just created
- void trie_node_delete(TrieNode *n)
    - This function is the destructor for a TrieNode
    - Free the TrieNode to gain back the memory it used
- TrieNode *trie_create(void) {
    - This function will initialize a root of a TrieNode with the code EMPTY_CODE (1)
    - Call and return trie_node_crete(EMPTY_CODE)
- void trie_reset(TrieNode *root)
    - This function will reset a Trie to just the root TrieNode
    - Create a for loop that will iterate through all of the children
        - Check if there is a valid element in the current index
        - if (root->children[i])
            - If there is an element in the index, call trie_delete
- void trie_delete(TrieNode *n)
    - This function will delete a sub-Trie starting from the sub-Trie's root
    - Check if the TrieNode is valid
        - if it isn't exit function

    - Create a for loop that will iterate through the 256 ASCII characters
        - Call trie_delete() to delete elements in children
    - Delete the TrieNode itself by calling trie_node_delete
- TrieNode *trie_step(TrieNode *n, uint8_t sym)
    - This function will return a pointer to the child TrieNode representing the symbol sym
    - Return the current element at index children[sym]

WordTables (word.c)

This source file contains the Word/WordTable ADT. Each index of the WordTable will house a Word, which will store byte arrays of uint8_t. Creating this ADT allows for a quick lookup when using decompression.

```
typedef struct Word {
  uint8_t *syms
  uint32_t len
} Word;

typedef Word *WordTable;
```

The following functions are as follows:

- Word *word_create(uint8_t *syms, uint64_t len)
    - This function is a constructor for a word
    - Allocate memory to the word using malloc()
    - Allocate memory to the word's symbols (w->syms) using malloc()
        - (uint8_t *)malloc(sizeof(uint8_t) * len + 1)
    - Set the word's length to the given length
    - Copy the given symbols to the word using a for loop
        - for (i - len)
            - w->syms[i] = syms[i]
    - Return the Word
- Word *word_append_sym(Word *w, uint8_t sym)

    - This function will construct a new Word from the specified Word appended with a symbol
    - Allocate memory for a temporary variable that will store syms
    - Set the temp syms to w->syms using a for loop
    - Create a new Word create using (temp, w->len + 1)
    - Set the last index of the new Word to the sym to be appended
    - Free the memory of the temporary variable
    - Return the new Word with the previous syms and the newly appended one
- void word_delete(Word *w)
    - This function is a destructor for a Word
    - Free the memory allocated from w->syms
    - Free the Word
- WordTable *wt_create(void)
    - This function will create a new WordTable, which houses an array of Words
    - Create a WordTable using calloc(MAX_CODE, sizeof(Word *))
        - Using calloc will initialize the WordTable
    - Initialize the WordTable at index EMPTY_CODE (1)
        - (Word *)calloc(1, sizeof(Word))
    - Return the WordTable
- void wt_reset(WordTable *wt)
    - This function will reset a WordTable to having just the empty word
    - Create a for loop that will iterate through i = START_CODE through MAX_CODE
        - Call word_delete(wt[i])
- wt_delete(WordTable *wt)
    - This function will delete an entire WordTable
    - Create a for loop that will iterate through i = 0 - MAX_CODE
        - Check if the current index of WordTable is NULL
            - if it is, call word_delete(wt[i])
    - Free the memory of the WordTable using free()

File I/O (io.c)

This source file stores the functions in replicating efficient I/O of files. These reads and writes to files will be done in one BLOCK at a time (4kb).

```
extern uint64_t total_syms;
extern uint64_ total_bits;

static uint8_t bytebuff[4096];
static uint16_t bytebuffindex = 0;

static uint8_t pairbuff[4096];
static uint16_t pairbuffindex= 0;

typedef struct FileHeader {
  uint32_t magic;
  uint16_t protection;
} FileHeader;
```

The functions are as follows:

- int read_bytes(int infile, uint8_t *buf, int to_read)
    - This function will loop to read the specified number of bytes
    - Declare variables to store the read bytes and the total bytes
    - Create a do while loop
        - while (rbytes > 0)
        - Set rbytes to read(infile,buf + total, to_read - total)
        - Increment total by rbytes
    - For statistical purposes, set the total_bits to the total number of bytes
    - Return the total number of bytes
- int read_bytes(int infile, uint8_t *buf, int to_read)
    - This function will loop to write the specified number of bytes
    - Declare variables to store the write bytes and the total bytes
    - Create a do while loop

        - while (wbytes > 0)
        - Set wbytes to read(outfile,buf + total, to_write - total)
        - Increment total by wbytes
    - For statistical purposes, set the total_syms to the total number of bytes
    - Return the total number of bytes
- void read_header(int infile, FileHeader *header)
    - This function will read in the sizeof(FileHeader) bytes from the input file
    - Call the function read_bytes() to read the header
        - read_bytes(infile, (uint8_t *)header, sizeof(FileHeader))
- void write_header(int outfile, FileHeader *header)
    - This function will write in the sizeof(FileHeader) bytes from the input file
    - Call the function write_bytes() to write the header
        - write_bytes(outfile, (uint8_t *)header, sizeof(FileHeader))
- bool read_sym(int infile, uint8_t *sym)
    - This function will read in a symbol from the input file. The function will return true if there are more symbols to read and false if there isn't
    - Declare a variable to signal the end of the BLOCK
    - Create a conditional that will check if the bytebuffindex is 0
        - if it is, set end to read_bytes(infile, bytebuff, 4096)
    - Set sym to the current index of bytebuff
    - Create a conditional that will check if bytebuffindex is 4096
        - if it is, set the bytebuffindex to 0
    - Create a conditional that will check if end == 4096 (BLOCK)
        - if it is, return true
        - otherwise, return false if bytebuffindex == end + 1
- void buffer_pair(int outfile, uint16_t code, uint8_t sym, uint8_t bitlen)
    - This function will buffer a pair, which comprises of a code and a symbol
    - Create a for loop that will iterate through code from 0 - bitlen
        - Create a conditional to check if the code is set
            - if (code & 1) == 1
            - if it is, set the bit, else, clear the bit

        - Increment the pairbuffindex by 1
        - Shift the code right by 1
        - Check if pairbuffindex is BLOCK * 8
            - if it is, call write_bytes()
            - Reset the pairbuffindex
    - Create a for loop that will iterate through sym from 0 - 8
        - Create a conditional to check if the sym is set
            - if ((sym & 1 == 1)
            - if it is, set the bit, else clear the bit
        - Increment the pairbuff index by 1
        - Shift the sym right by 1
        - Check if pairbuffindex is BLOCK * 8
            - if it is, call write_bytes()
            - Reset the pairbuffindex
- void flush_pairs(int outfile)
    - This function will write out any remaining pairs of symbols and codes to the output file
    - Declare a variable to store to_write bytes
    - Create a conditional that will convert the pairbuffindex to the correct byte
    - Call write_bytes(outfile, pairbuff, bytes)
    - Reset the pairbuffindex to 0
- bool read_pair(int infile, uint16_t *code, uint8_t *sym, uint8_t bitlen)
    - This function will read code and symbol pairs form the input file
    - Have the read code in the pointer to code
        - *code = 0
    - Create a for loop that iterates through code from i = 0 - bitlen
        - Check if the pairbuffindex is 0 and call read_bytes()
        - Create a conditional that will check if the current bit at pairbuff index is set
            - if it is, set the code bit
                - *code |= (1 << (i % 16)

                - else, clear the bit
                    - *code &= ~(1 << (i % 16)
            - Check if pairbuffindex is BLOCK * 8
                - if it is, reset the index
    - Have the sym code in the pointer to sym
        - *sym = 0
    - Create a for loop that iterates through i = 0 - 8
        - Check if the pairbuffindex is 0 and call read_bytes()
        - Create a conditional that will check if the current bit at pairbuff index is set
            - if it is, set the sym bit
                - *sym |= (1 << (i % 8)
            - else, clear the bit
                - *sym &= ~(1 << (i % 8)
        - Check if pairbuffindex is BLOCK * 8
            - if it is, reset the index
    - Return the boolean value of *code != STOP_CODE (0)
- void buffer_word(int outfile, Word *w)
    - This function will buffer a Word
    - Create a for loop that will iterate through i = 1 - w->len +1
        - Set the bytebuff at bytebuffindex to the symbol of the word at index i
            - bytebuff[bytebuffindex] = w->syms[i]
        - Increment the bytebuffindex by 1
        - Check if bytebuffindex == BLOCK and write_bytes() and reset bytebuffindex
- void flush_words(int outfile)
    - This function will write out any remaining symbols in the buffer
    - Create a conditional that will check if the bytebuf index != 0
        - if true, call write_bytes(outfile, bytebuffer, bytebuf index)