William Dai
12 December 2020
CSE-13S

Specification

Assignment 7: The objective of this assignment is to create compression and decompression programs using the LZ78 compression algorithm. When encoding, the first program, encode, will take an input file and compress it, outputting the encoded file an output file. When decoding, the second program, decode, will take an input file and decompress it, outputting the decoded file to an output file.

encode.c: The C file that encodes and compresses a given input file and outputs the result into a given output file. It will also track statistics of the compression.

decode.c: The C file that decodes and decompresses a given input file and outputs the result into a given outpul file. It will also track statistics of the decompression.

trie.c: Function file defining the trie and its implementations.

trie.h: Header file for trie.c.

word.c: Function file defining the word and word table and their implementations.

word.h: Header file for word.c.

io.c: Function file defining the I/O redirection implementations.

io.h: Header file for io.c.

code.h: Definer file for defining various defines.

The primary function of this assignment is to implement the LZ78 compression algorithm to compress and decompress an input file or vice versa through a decompression.

Definitions Used in Specification:

STOP_CODE = 0
EMPTY_CODE = 1
START_CODE = 2
MAX_CODE = Largest int Value
ALPHABET = 256
LSB = Least Significant Bit

William Dai
12 December 2020
CSE-13S

Program Design

Trie (trie.c)

The trie data structure implements a structure that uses nodes to store symbols to make up words as nodes are being traversed, akin to an "advanced" linked list with multiple pointers. When a symbol from a word is passed into the trie, it will check whether any of the word's prior symbols with a trie node line have been created down the trie. If so, it will place the symbol at the end. If not, it will create a new trie node line. This will make a tree of "words" that will efficiently retrieve information. It uses an index to determine where in the tree it will be. The trie data holds 256 (ALPHABET) pointers called the children, signifying the number of ASCII values that exist, and the code to which will signify where a node will go.

Function trie_node_create() creates a trie node. It takes in $index$, the index of code to which it will determine where the node will go in the trie. It will first allocate memory for the trie node, then will set the code of the trie node to $index$. It returns the trie node created.

```
Function trie_node_create(index) {
      Allocate memory to n;
      If n is not NULL {
            Set n->code to index;
            return n;
      }
}
```

Function trie_node_delete() deletes the designated trie node. It takes in n, the trie node to delete. It will free the memory at the trie node.

```
Function trie_node_delete (n) {
      free(n);
}
```

Function trie_create() creates a trie. It does not take any parameters, but rather create a trie node with EMPTY_CODE, signifying the root node. It will do so by calling trie_node_create() with this index. It returns the trie node created.

```
Function trie_node_delete (n) {
      return trie_node_create(EMPTY_CODE);
}
```

William Dai
12 December 2020
CSE-13S

Function trie_delete() deletes the designated trie. It takes in n, the trie to delete. It will do so by, while scanning all of the children pointers (from 0 to ALPHABET), recursively call trie_delete(). If n at each recursion is NULL, return. Otherwise, it will recursively call until it reaches a leaf node, to which it will call trie_node_delete() to delete the designated node.

```
Function trie_delete (n) {
      If n is NULL {
            return;
      }
      For i from 0 to ALPHABET {
            trie_delete(n->children[i]);
      }
      trie_node_delete(n);
}
```

Function trie_reset() resets the designated trie to the root node. It takes in $root$, the trie to be reset. It will go by scanning all of the children pointers (from 0 to ALPHABET). If the children dereference is not NULL, it will call trie_delete() on the designated children.

```
Function trie_reset (n) {
      For i from 0 to ALPHABET {
            If root->children[i] is not NULL {
                  trie_delete(root->children[i]);
            }
      }
}
```

Function trie_step() will step into the next child node of the current node that correlates to the symbol passed in. It takes in n, the parent trie node the current pointer is at, and sym, the symbol to step into, if the symbol is in a child trie node. It will do so by returning the sym if found in the children nodes. If children[sym] doesn't exist it will return NULL by default.

```
Function trie_step(n, sym) {
      return n->children[sym];
}
```

William Dai
12 December 2020
CSE-13S

Word and Word Table (word.c)

This section will implement two concepts: word and word table. The word structure will implement an array of characters to make up a word, with the structure holding symbols of that array, as well as the length of the word. The world table structure is an array that will house the words by pushing words into the word table.

Function word_create() will create a word structure. It takes in $syms$, the "word", or more accurately symbols that will construct the word, and len, the length of the word. It will first allocate memory for the word structure. Then it will set the length to len, followed by allocating memory for the symbol dereference. It will then copy over the symbols onto the word symbol array, then will return the word.

```
Function word_create(syms, len) {
      Allocate memory for w;
      Set w->len to len;
      Allocate memory for w->syms;
      Copy syms onto w->syms;
      return w;
}
```

Function word_create_append() will create a new word based on a word passed in with another symbol appended to the end. It takes in w, the original word, and sym, the symbol to append onto the original word. It will first allocate memory for the new word, then will set the length of the new word to $(w \rightarrow len) + 1$. It will then allocate memory for the new word's symbol dereference. It will then copy over the symbols from w onto the new word, and after will append sym onto the end of the word, then will return the new word.

```
Function word_create(syms, len) {
      Allocate memory for w;
      Set w->len to len;
      Allocate memory for w->syms;
      Copy syms onto w->syms;
      return w;
}
```

Function word_delete() will delete a designated word. It takes in w, the word to delete. It will do so by first freeing the symbol dereference of w, then will free w itself.

```
Function word_delete(w) {
      free(w->syms);
      free(w);
}
```

William Dai
12 December 2020
CSE-13S

Function wt_create() will create a new word table. It will do so by first allocating space for the word table structure, then will initialize the first index of the array (EMPTY_CODE) to a NULL word. It returns the word table.

```
Function wt_create() {
      Allocate memory for wt;
      Set wt[EMPTY_CODE] to word_create(NULL);
      return wt;
}
```

Function wt_delete() will delete a designated word table. It takes in wt, the word table to delete. It will do so by scanning the word table array from EMPTY_CODE to MAX_CODE. If a designated word within the array is not NULL, it call word_delete() on the designated word to delete it. It will free wt itself.

```
Function wt_delete () {
      For i from START_CODE to MAX_CODE {
            If wt[i] is not NULL {
                  word_delete(wt[i]);
            }
            free(wt);
      }
}
```

Function wt_reset() will reset a designated word table to a clean state. It takes in wt, the word table to reset. It will do so by scanning the word table array from EMPTY_CODE to MAX_CODE. If a designated word within the array is not NULL, it will call word_delete() on the designated word. The difference between wt_reset() and wt_delete() is that wt_reset() leaves the word table intact, while wt_delete() deletes the whole structure.

```
Function wt_delete () {
      For i from START_CODE to MAX_CODE {
            If wt[i] is not NULL {
                  word_delete(wt[i]);
            }
      }
}
```

William Dai
12 December 2020
CSE-13S

Input/Output (io.c)

This section implements the input/output used to compress and decompress files using encoding and decoding, respectively. I/O will handle the manipulation of files, including reading and writing headers and bytes, and buffering for reads and writes onto an input file and output file, respectively. In this implementation, it will use 4KB (4096) to buffer. Two buffers will be used, $bitbuf$ and $symbuf$, as well as an index counter for each, $bitindex$ and $symindex$. For the header struct, $FileHeader$, it will have two dereferences for $magic$ and $protection$. Magic is a number that will indicate the file being compressed, and protection will hold the permissions of the original, uncompressed file.

Function read_bytes() will read the amount of bytes within an input file. It takes in $infile$, the input file to read in, buf, the buffer to fill, and to_read, the number of bytes to read. It will have two ints within, $readBytes$ and $total$, both initially set to 0. With these, it will read in $infile$ with $buf + total$, and $to\_read - total$, then will increment $total$ by $readBytes$. It will loop through this until $readBytes$ is less than 0 and $total$ is not equal to to_read.

```
Function read_bytes(infile, buf, to_read) {
      int readBytes equals to 0;
      int total equals to 0;
      do {
            Set readBytes to read(infile, buf + total, to_read − total);
             Increment total by readBytes;
      } while readBytes > 0 and total is not equal to to_read;
      return total;
}
```

Function write_bytes() will write the amount of bytes to an output file. It takes in $outfile$, the output file to write to, buf, the buffer to fill, and to_write, the number of bytes to write. It will have two ints within, $writeBytes$ and $total$, both initially set to 0. With these, it will write to $outfile$ with $buf + total$, and $to\_write - total$, then will increment $total$ by $writeBytes$. It will loop through this until $writeBytes$ is less than 0 and $total$ is not equal to to_write.

```
Function write_bytes(outfile, buf, to_write) {
      int writeBytes equals to 0;
      int total equals to 0;
      do {
            Set writeBytes to read(outfile, buf + total, to_write − total);
            Increment total by writeBytes;
      } while writeBytes > 0 and total is not equal to to_write;
      return total;
}
```

William Dai
12 December 2020
CSE-13S

Function read_header() will read in a file header within an input file. It takes in $infile$, the input file to read in, and $header$, the location of where the header will point to. It will call read with $infile$, $header$, and the size of $FileHeader$.

```
Function read_header(infile, header) {
      read(infile, header, size of FileHeader);
}
```

Function write_header() will write in a file header within an output file. It takes in $outfile$, the output file to write to, and $header$, the location of where the header will write to. It will call write with $outfile$, $header$, and the size of $FileHeader$.

```
Function write_header(outfile, header) {
      write(outfile, header, size of FileHeader);
}
```

Function read_sym() will read in symbols found within an input file. It takes in $infile$, the input file to read in, and sym, the location of where the symbol will be stored. It will have an int end that tracks the current byte and uses the global variables $symbuf$ and $symindex$. It will first check if $symindex$ is 0, and if it is, it will call read_bytes() onto end. It will then set sym to the current $symindex$ in $symbuf$. It will then check if $symindex$ is at 4KB, and if it is, it will reset it back to 0. It will then check if end is at 4KB. If it is, the function will return true, else if $symindex$ is equal to $end + 1$, the function will return true. Otherwise, if the prior checks are not matched, return true regardless.

```
Function read_sym(infile, sym) {
      int end equals to 0;
      If symindex is equal to 0 {
            Set end to read_bytes(infile, symbuf, 4096);
      }
      Set byte to symbuf[symindex++];
      If symindex is equal to 4096 {
            Set symindex to 0;
      }
      If end is equal to 4096 {
            return true;
      }
      Else {
            If symindex is equal to end + 1 {
                  return false;
            }
            Else {
                  return true;
            }
      }
}
```

Function buffer_pair() buffers a pair of a symbol and an index. It takes in $outfile$, the output file to buffer into, $code$, the index, sym, the symbol, and bit_len, the number of bits of the index to buffer in. It will use the global variables $bitbuf$ and $bitindex$ as well. It will first buffer in the bits of $code$ starting from the LSB, then do the same for the bits of sym. If the buffer is full, it will call write_bytes to write onto the output file.

```
Function buffer_pair(outfile, code, sym, bit_len) {
      // code
      For i from 0 to bit_len {
            If LSB is equal to 1 {
                  Set bit of bitbuf at bitindex;
            }
            Else {
                  Clear bit of bitbuf at bitindex;
            }
            Increment bitindex by 1;
            Right shift code by 1 bit;
            If bitindex is equal to 4096 * 8 {
                  write_bytes(outfile, bitbuf, 4096);
            }
      }
      // sym
      For i from 0 to 8 {
            If LSB is equal to 1 {
                  Set bit of bitbuf at bitindex;
            }
            Else {
                  Clear bit of bitbuf at bitindex;
            }
            Increment bitindex by 1;
            Right shift code by 1 bit;
            If bitindex is equal to 4096 * 8 {
                  write_bytes(outfile, bitbuf, 4096);
            }
      }
}
```

William Dai
12 December 2020
CSE-13S

Function read_pairs() reads in an input file's content onto the designated buffer. It takes in $infile$, the input file to read in, $code$, the index, sym, the symbol, and bit_len, the number of bits of the index to read in. It will use the global variables $bitbuf$ and $bitindex$ as well. The pairs are read into the buffer, with $bitindex$ acting as the counter to keep track of the current bit within the buffer. It will read in blocks of bits at a time. Once all the bits within a block are read, the next block is read and so on until all the blocks are read in. The function will return true if there are pairs still needed to be read in, else false if there are no more pairs to be read in.

```
Function read_pairs(outfile, code, sym, bit_len) {
      Set code to 0;
      For i from 0 to bit_len {
            If bitindex is equal to 0 {
                  read_bytes(infile, bitbuf, 4096);
            }
            If the bit of bitbuf at bitindex is set {
                  Set bit of code at i;
            }
            Else {
                  Clear bit of code at i;
            }
            Increment bitindex by 1;
            If bitindex is equal to 4096 * 8 {
                  Set bitindex to 0;
            }
      }
      Set sym to 0;
      For i from 0 to 8 {
            If bitindex is equal to 0 {
                  read_bytes(infile, bitbuf, 4096);
            }
            If the bit of bitbuf at bitindex is set {
                  Set bit of sym at i;
            }
            Else {
                  Clear bit of sym at i;
            }
            Increment bitindex by 1;
            If bitindex is equal to 4096 * 8 {
                  Set bitindex to 0;
            }
      }
      return code != STOP_CODE;
}
```

William Dai
12 December 2020
CSE-13S

Function flush_pairs() will buffer out the remaining pairs within the designated buffer if the buffer still has lingering pairs that did not get pushed out due to the buffer not being completely filled while in buffer_pair(). It takes in $outfile$, the output file to buffer into. It will use the global variables $bitbuf$ and $bitindex$ as well. It will have an int $bytes$ that tracks the necessary number of bytes needed to write out to the output file. It will first change if $bitindex$ is not 0. If it is not, it will check if the modulo from $bitindex$ % 8 is 0. If it is, $bytes$ will be set to $bitindex$ divided by 8. If it is not, it will set $bytes$ to $bitindex$ divided by 8 plus 1. It will then call write_bytes to write into the output file.

```
Function flush_pairs(outfile) {
      int bytes equals to 0;
      If bitindex is not equal to 0 {
            If bitindex % 8 is equal to 0 {
                  bytes = bitindex / 8;
            }
            Else {
                  bytes = (bitindex / 8) + 1;
            }
            write_bytes(outfile, bitbuf, bytes);
      }
}
```

Function buffer_word() buffers a word onto the designated output file. It takes in $outfile$, the output file to buffer into, and w, the word to buffer in. It will use the global variables $bitbuf$ and $bitindex$ as well. It will take each symbol of the word and push it into the buffer. When the buffer is full, it will call write_bytes() to write onto $outfile$.

```
Function buffer_word(outfile, w) {
      For i from 0 to w->len {
            Set symbuf[symindex++] to w->syms[i];
            If symindex is equal to 4096 {
                  write_bytes(outifle, symbuf, 4096);
                  Set symindex to 0;
            }
      }
}
```

Function flush_words() will buffer out the remain words within a buffer onto the designated output file if it is not empty. It takes in $outfile$, the output file to buffer into. It will also use the global variables $symbuf$ and $symindex$ as well. It will check if $symindex$ is not 0. If it is not, it will call write_bytes() to write onto the output file.

```
Function flush_word(outfile) {
      If symindex is not equal to 0 {
            write_bytes(outfile, symbuf, symindex);
      }
}
```

William Dai
12 December 2020
CSE-13S

Main Program Function:

Function bitlength() calculates the bit length of an int value passed in through the use of bit shifting. It takes in i, the int to count. It will have an int inc that counts how many times the bits will be shifted. Whenever i is left shifted, inc will be incremented by 1. i will be left shifted until i is 0, to which it will return inc.

```
Function bitlength(i) {
      int i equal to 0;
      While i is not equal to 0 {
            Left shift i by 1;
            Increment inc by 1;
      }
      return inc;
}
```

There will be some statistics that accompany the encode and decode programs that will keep track of the efficiency of the compression and decompression. These are:

Compressed file size (in bytes)
Uncompressed file size (in bytes)
Compression ratio (percentage)

The compression ratio is calculated by the following formula:

$$Compression\ ratio = 100 * \left[1 - \left(\frac{Compressed\ size}{Uncompressed\ size}\right)\right]$$

William Dai
12 December 2020
CSE-13S

Encode (encode.c)

The main function in the encode.c file will be to take a designated input file, assign it a header for the compression, and then compress the contents of the file with the LZ78 compression algorithm. It will then output the now compressed contents onto the designated output file.

```
Function main() { // For encode.c
      getopt Loop { ... }
      Open infile;
      Open outfile;
      Create header;
      Compress with LZ78 Compression Algorithm;
      Print statistics if requested;
      Close infile;
      Close outfile;
}
```

Decode (decode.c)

The main function in the decode.c file will be to take a designated output file, read the header to ensure the file can be decoded and decompressed, and then decompress the contents of the file with the LZ78 decompression algorithm. It will then output the now decompressed contents onto the designated output file.

```
Function main() { // For decode.c
      getopt Loop { ... }
      Open infile;
      Open outfile;
      Read header;
      Decompress with LZ78 Decompression Algorithm;
      Print statistics if requested;
      Close infile;
      Close outfile;
}
```