Design

Overall Design of asgn7.

The program has the following files.

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

For encoding files, the encode.c file used and uses the following header files, trie.h and io.h

For decoding files, the decode.c file used and uses the following header files, word.h and io.h

For this design document, I am going to divide the document into 2 describing the encode design and decode design.

But to understand both the design of io.h needs to be explained.

Io.h

This file does the reading and writing and buffering part of the program.

Pseudo code

Global integers:

Bytes_read and bytes_writen

```
read_bytes(infile, buf, bytes to read)
        Total_read = 0
        Bytes = 0
        While till bytes does not equal zero and total read does not equal to_read
                Bytes = read(infile, total_read + buf, to_read - total_read)
                Total_read += bytes
        Bytes_read += total_read
        Return total_read
write _bytes
        Total_write = 0
        Bytes = 0
        While till bytes does not equal zero and total_write does not equal to_write
                Bytes = read(infile, total_write + buf, to_write - total_write)
                Total_read += bytes
        Bytes_read += total_write
```

```
        Return total_write
read_header(infile, header)
        read_bytes(infile, header, size of header)
read_header(infile, header)
        read_bytes(infile, header, size of header)

Global static variables:
symbuf[BLOCK] and sym_index = 0

Read_sym(infile, sym)
        End = 0
        If sym == 0
                end  = read(infile, symbuf, BLOCK)
        Sym = symbuf[sum_index++]
        If sym index == BLOCK
                Sym_index  = 0
        If end == BLOCK
                Return true
        Else
                If sym_index == end + 1
                        Return false
                Else
                        Return true

Global static variables:
bitbuf[BLOCK] and bit_index = 0

Buffer_pair(outfile, sym, code, bit_len)
        Loops through the code
                If lsb of code = 1
                        Set bit in bitbuf at bit_index
                Else
                        Clear in bitbuf at bit_index
                Bit_index += 1
                Take off lsb from code
                If bit_index  == BLOCK * 8
                        write_bytes(outfile, bitbuf, BLOCK)
                        Bit_index = 0
        Loops through sym
                If lsb of sym = 1
```

```
                Set bit in bitbuf at bit_index
            Else
                Clear in bitbuf at bit_index
            Bit_index += 1
            Take off lsb from sym
            If bit_index  == BLOCK * 8
                write_bytes(outfile, bitbuf, BLOCK)
                Bit_index = 0
    Return
Flush_pairs
    Bytes = 0
    If bit_index does not equal 0
        If bit_index % 8 equals 0
            Bytes = bit_index / 8
        Else
            Bytes = (bit_index / 8) + 1
        write_bytes(outfile, bitbuf, bytes)
    Return

Read_pair
    Code = 0
    Loops through each bit in code
        If bit_index  =  0
            read_bytes(infile, bitbuf, BLOCK)
        If bit in bitbuf at bit_index is set
            Set bit in code
        Else
            Clear bit in code
        Bit_index += 1
        If bit_index = BLOCK * 8
            Bit_index = 0
    Sym = 0
    Loops through each bit in sym
        If bit_index  =  0
            read_bytes(infile, bitbuf, BLOCK)
        If bit in bitbuf at bit_index is set
            Set bit in sym
        Else
            Clear bit in sym
        Bit_index += 1
```

```
                If bit_index = BLOCK * 8
                        Bit_index = 0
        Return true if code != stop_code
buffer_word(outfile, word)
        Goes through each sym in word
                Symbuf[sym_index++] = w->syms[i]
                If sym_index == BLOCK
                        Write_bytes(outfile, symbuf, BLOCK)
                        Sym_index = 0
        Return
Flush_word(outfile)
        if sym_index != 0
                write_bytes(outfile, symbuf, sym_index)
        return
```

Encode.c

So this part of the lab takes in a file and compresses the file into an output file. The compression algorithm was given in the lab manual and I tturned it into a funcion within encode.c, main() calls compress(infile, outfile). The function follows the following pseudo code:

```
COMPRESS(infile, outfile)
 1  root = TRIE_CREATE()
 2  curr_node = root
 3  prev_node = NULL
 4  curr_sym = 0
 5  prev_sym = 0
 6  next_code = START_CODE
 7  while READ_SYM(infile, &curr_sym) is TRUE
 8      next_node = TRIE_STEP(curr_node, curr_sym)
 9      if next_node is not NULL
10          prev_node = curr_node
11          curr_node = next_node
12      else
13          BUFFER_PAIR(outfile, curr_node.code, curr_sym, BIT-LENGTH(next_code))
14          curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
15          curr_node = root
16          next_code = next_code + 1
17      if next_code is MAX_CODE
18          TRIE_RESET(root)
19          curr_node = root
20          next_code = START_CODE
21      prev_sym = curr_sym
22  if curr_node is not root
23      BUFFER_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code))
24      next_code = (next_code + 1) % MAX_CODE
25  BUFFER_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
26  FLUSH_PAIRS(outfile)
```

I also created a function that calculates the the bit length of a number using the following mathematical equation: log2(x) = bit length of x

compress() also uses a trie ADT, which is defined in trie.h library with functions defined in trie.c The Abstract data structure creates nodes and connecting them like a tree and it always starts with a root with index EMPTY_CODE.

Trie.c

Pseudo code

Trie_node_create(code)

- Allocates memory for trie node
- Checks if node is initialized
- Sets node->code to code
- Returns node

trie_node_delete(trienode node)

- free(node)

Return
trie_create(void)
Return trie_node_create(EMPTY_CODE)
trie_reset(trienode root)
Itretes through all of the i children of trie_node
trie_delete(root->children[i])
Return
Trie_delete(trienode n)
Checks if n is initialized
Iterates through each i of children of node n
trie_delete(n->children[i])\
If n is true
trie_node_delete(n)
return
Trie_step(trie_node node, sym)
Return node->chidlren[sym]

Than for the main function within encode.c.
Global variables
Bytes_read
bytes_written
main()
Used getopt to use command line arguments
Getopt loop
If -v
Set stats true
If -i
set std_in false
Set input_file to what is imputed after -i
If -o
set std_out false
Set output_file to what is imputed after -o

If std_in is true
Set infile to STDIN_FILENO
Else
Infile = open(input_file, O_RDONLY)

If std_out is true
Set infile to STDOUT_FILENO

Else
    outfile= open(output_file, O_WRONLY | O_CREATE | O_TRUNC)

Create infile stat struct infile_stats
Using fstat get the state of infile and stat it infile_stats
Using fchmod set the outfile protection to the same of infile protection

Create header for oufile called outfile_header and allocate memory for it
Check if outfile_header is initialized
Set the protection and magic for header
Write the header to the outfile

compress(infile, outfile)

If stats is true
    print "Compressed file size:" bytes_written "bytes"
    print "Uncompressed file size:" bytes_read "bytes"
    Compressed_ratio = 100 * (1 - (bytes_written / bytes_read))
    Print "Compression ratio:" compressed_ratio "%"

free(outfile_header)
close(infile)
close(outfile)

Decode.c

So this part of the lab takes in a file and compresses the file into an output file. The decompression algorithm was given in the lab manual and I turned it into a funcion within decode.c, main () calls decompress(infile, outfile). The function follows the following pseudo code:

```
DECOMPRESS(infile, outfile)
 1  table = WT_CREATE()
 2  curr_sym = 0
 3  curr_code = 0
 4  next_code = START_CODE
 5  while READ_PAIR(infile, &curr_code, &curr_sym, BIT-LENGTH(next_code)) is TRUE
 6        table[next_code] = WORD_APPEND_SYM(table[curr_code], curr_sym)
 7        buffer_word(outfile, table[next_code])
 8        next_code = next_code + 1
 9        if next_code is MAX_CODE
10              WT_RESET(table)
11              next_code = START_CODE
12  FLUSH_WORDS(outfile)
```

I also created a function that calculates the the bit length of a number using the following mathematical equation: log2(x) = bit length of x

decompress() also uses a wordtable ADT, which is defined in word.h library with functions defined in word.c. The abstract is basically a string of symbols

Word.c

Pseudocode

Word_create(syms, len)

Allocates memory for word w
Checks if word w is inititialized
Allocates memory for w->syms
Checks if w->syms is intialized
Sets w->len to len
Iterates through w->syms
Setting w->syms[i] = syms[i]

Return w

```
Word_append(word w, sym)
        If w equal NULL
                Syms = {sym}
                word new_word = word_create(syms, 1)
                Return new_word
        Else
                Word new_word = word_create(w->syms, w-> len)
                Reallocate new_word->syms so that it can store one extra symbol
                Checks if new->word sysm is intialized
                new_word->len += 1
                new_word->syms[new_word->len - 1] = sym
                Return new_word
word_delete(word w)
        free(w->syms)
        free (w)
wt_create(void)
        Wordtable wt = calloc(MAX_CODE, size of word)
        Return wt
Wt_reset(wordtable wt)
        Goes through the word table starting at STARTCODE not at the beginning
                word_delete(wt[i])
        Return
wt_delete(wordtable wt)
        Goest through the whole table starting at from the front
                Check if w is not NULL
                        word_delete(wt[i])
        free(wt)
        Return

Than for the main function within encode.c.
Global variables
Bytes_read
bytes_written

main()
        Used getopt to use command line arguments
        Getopt loop
                If -v
                        Set stats true
                If -i
```

```
            set std_in false
            Set input_file to what is imputed after -i
        If -o
            set std_out false
            Set output_file to what is imputed after -o

If std_in is true
        Set infile to STDIN_FILENO
Else
        Infile = open(input_file, O_RDONLY)

If std_out is true
        Set infile to STDOUT_FILENO
Else
        outfile= open(output_file, O_WRONLY | O_CREATE | O_TRUNC)

Creates and allocates memory for a header for infile called infile_header
read_header(infile, infile_header)

Checks if infile_header->magic equals magic
        If not print statement

Uses fchmod to set the outfile protection the same as infile_header

decompress(infile, outfile)

If stats is true
        print  "Compressed file size:"  bytes_read "bytes"
        print  "Uncompressed file size:"  bytes_written "bytes"
        Compressed_ratio = 100 * (1 - (bytes_read / bytes_written))
        Print "Compression ratio:" compressed_ratio "%"

free(infile_header)
close(infile)
close(outfile)
```