Objective

1. Compressing a file in order to reduce the memory of the file.
2. Learning how to use trie node and WordTables
3. Learning to use buffers
4. Using File Descriptors
5. Decoding a file to put it back to its original state

High-Level Descriptions\

Compression:

Getting Users input for the inputfile and outputfile and if none is entered, we use stdin and stdout. Opening the files as a file descriptor and reading from the input file in order to get the symbols and the code for where the symbol would be placed. Placing them into bufffers as binary and printing the ascii values of it. There would be a lot of nonsense printed out but it would be compress because of the way that binary values were formed. By it's next code's length or by 8 bits.

Decompression:

Getting Users input for the inputfile and outputfile and if none is entered, we use stdin and stdout. Opening the files as a file descriptor and reading from the input file in order to get the symbols and the code for where the symbol would be placed. The code and symbols are read through binary and changed to ascii values to put them back into their original forms. The output of the compression should be the same as the input of the compression.

Word.c

Stores the symbols of the ascii values and the length of the symbol pointers.

Word Constructor

```
Word *word_create(uint8_t symbol pointer, length) {
        Creates new word and malloc the size of the Word struct
        w->syms = symbol pointer
        w->len = entered length
        Return new word
}
```

Appending the symbols within the word

```
Word *word_append_sym(word, new sym) {
        If word doesnot exist call word_create with len 1 and the new sym
        If Word exists but len is zero, change the len to one and create a new word with the new length and the new symbol
        If Word exists, append the new symbol to the symbol pointer and reallocated memory for the new symbol to enter and increment the length
        Return the new word
}
```

```
Deconstructor for Word
word_delete() {
        free(symbol pointer)
        free(word)
        Word and symbol pointer = null
}

Wordtable creation. Wordtable is just a array of Words.

wt_create() {
        Emptysym = Null
        Create an empty word with len 0
        Create a wordtable with size of WordTable times MAX_CODE -1
        Insert the empty word in index EMPTY_CODE
        Return newwordtable
}

Leaves only the empty word
wt_reset(wordtable) {
        While(wordtable[empty_code + 1 = i] != NULL) {
                word_delete(wt[i])
                i++
        }
        free(wordtable)
}
Deletes the word table
wt_delete(wordtable)
        While(wordtable[empty_code + 1 = i] != NULL) {
                word_delete(wt[i])
                i++
        }
        free(wordtable)
}

Trie.c
Constructor for a single trie node
TrieNode *trie_node_create(code) {
        Newtrienode with malloc of sizeof(trienode)
        newtrienode->code = code
        Sets all the children of the trie node to NULL
        Return the trie node
}
```

```
Deletes a single trienode
trie_node_delete(TrieNode *n) {
        Sets all the children of the trienodes to NULL
        free(n)
        N = NULL
}

Constructor for a trie
TireNode *trie_create() {
        TrieNode *roottrienode = tire_node_create(empty_code)
        Return roottrienode
}

Reseting the trie node to just the root
Void trie_reset(TrieNode *root) {
        Find the children nodes that aren't null and free them
}

Deletes the trie
trie_delete(TrieNode *n) {
        Find the children nodes that aren't null and free them
        Frees the trienode itself
}

Finds the TrieNode holding the index sym
TrieNode *trie_step(TrieNode *n, uint8_t sym) {
        Returns the children that holds index sym
        If not found return NULL
}

Io.c

Initialize global variables of buffers for syms and pairs

Read_header (infile, header) {
        Reads the fileheader from infile
        If magic of the fileheader doesn't match print an error message
        Swaps the endian if it is in big endian
}

Write_header (outfile, header) {
        Checks if its in big endians if it is swap the values of the fileheader
        Write the file header onto the output file.
```

```
}

read_sym(infiile, *sym) {
        Reads a block of 4kb
        Stores a byte into the sym and returns it.
        If the buffer gets full, reset the buffer
}

buffer_pair(outfile, code, sym, bitlen) {
        Stores the code in binary and the sym in binary.
        Code is the length of the bitlen
        Insert zeros near the end if needed.
        Write the binaries from the least significant bits on to the outfile when the buffer is at 4kb
        After storing the binaries into the buffer for pairs
        If the buffer gets full, reset it
}

flush_pair(outfile) {
        Writes the rest of the buffer for the pairs
}

buffer_word(outfile, word) {
        Stores the symbols of the word into the buffer for symbols and once its al 4kb write it on
to the outfile.

}

Flush_words(outfile) {
        Writes the remainder of the buffer for symbols onto the outfile
}
//Code by DDEL
```

```c
#ifndef __CODE_H__
#define __CODE_H__

#include <inttypes.h>

#define STOP_CODE   0                // Signals end of decoding/decoding.
#define EMPTY_CODE  1                // Code denoting the empty Word.
#define START_CODE  2                // Starting code of new Words.
#define MAX_CODE    UINT16_MAX       // Maximum code.

#endif
```

code.h

// Code by DDEI

canonical. Here is another module specifically for handling endianness.

```
#ifndef __ENDIAN_H__
#define __ENDIAN_H__

#include <inttypes.h>
#include <stdbool.h>

//
// Checks if the order of bytes on the system is big endian.
//
static inline bool is_big(void) {
  union {
    uint8_t bytes[2];
    uint16_t word;
  } test;
  test.word = 0xFF00;
  return test.bytes[0];
}

//
// Checks if the order of bytes on the system is little endian.
//
static inline bool is_little(void) {
  return !is_big();
}

//
// Swaps the endianness of a uint16_t.
//
// x:  The uint16_t.
//
```



```
static inline uint16_t swap16(uint16_t x) {
  uint16_t result = 0;
  result |= (x & 0x00FF) << 8;
  result |= (x & 0xFF00) >> 8;
  return result;
}

//
// Swaps the endianness of a uint32_t.
//
// x:  The uint32_t.
//
static inline uint32_t swap32(uint32_t x) {
  uint32_t result = 0;
  result |= (x & 0x000000FF) << 24;
  result |= (x & 0x0000FF00) << 8;
  result |= (x & 0x00FF0000) >> 8;
  result |= (x & 0xFF000000) >> 24;
  return result;
}

//
// Swaps the endianness of a uint64_t.
//
// x:  The uint64_t.
//
static inline uint64_t swap64(uint64_t x) {
  uint64_t result = 0;
  result |= (x & 0x00000000000000FF) << 56;
  result |= (x & 0x000000000000FF00) << 40;
  result |= (x & 0x0000000000FF0000) << 24;
  result |= (x & 0x00000000FF000000) << 8;
  result |= (x & 0x000000FF00000000) >> 8;
  result |= (x & 0x0000FF0000000000) >> 24;
  result |= (x & 0x00FF000000000000) >> 40;
  result |= (x & 0xFF00000000000000) >> 56;
  return result;
}
```

Encode.c

Main(argc, *8argv) {
	Switch statement in order to get the user input for infile and outfile or to print the compression stats.

	Open the input and output
	If not inputed, get stdin and stdout
	Sets up the fileheader for the output and writes it into the outputfile
	Compression (inputfile, outputfile)
	Prints the stats of the compressions if the flag was inputted.
	Close inputfile and outputfile
	free(filehead)

}
Gets the length of the code
bit_length(code) {
	If code is greater than or equal to 1, length is the floor of log base 2 of code +1
	Else length = 1;
}

decode.c

Main(argc, *8argv) {
	Switch statement in order to get the user input for infile and outfile or to print the compression stats.

	Open the input and output
	If not inputed, get stdin and stdout
	Reads the filehead from the input and checks if the magic and protection matches up else throw an error
	decompression (inputfile, outputfile)
	Prints the stats of the decompressions if the flag was inputted.
	Close inputfile and outputfile
	free(filehead)

}
Gets the length of the code

bit_length(code) {
    If code is greater than or equal to 1, length is the floor of log base 2 of code +1
    Else length = 1;
}

8.2 Decompression

```
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)
```

//code by DDEL

8.1 Compression

```
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)
```