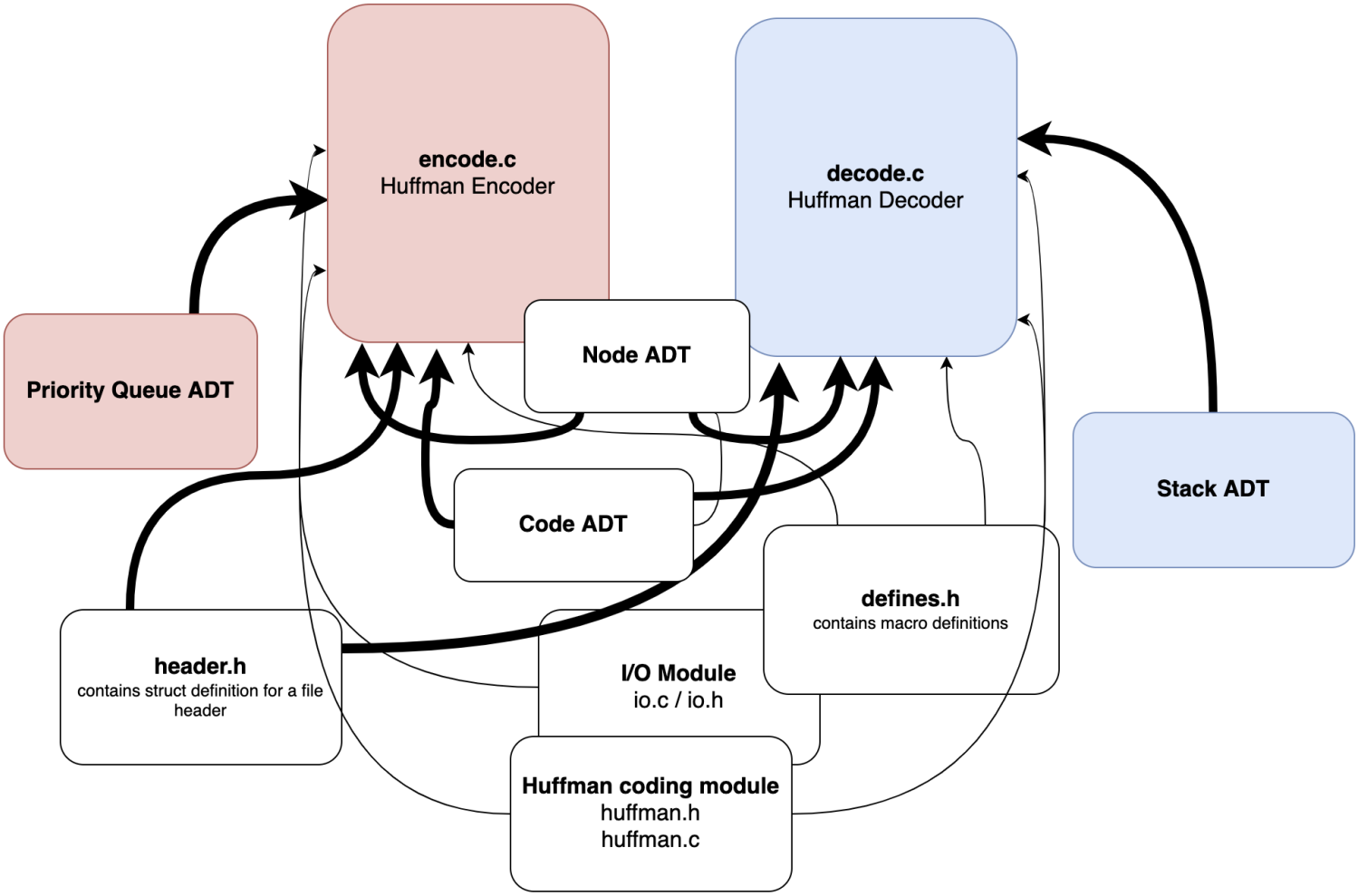
Assignment 6
Design Document

Purpose

Create a two programs: one that can compress a given file using Huffman coding, and one that can decompress such a file.

Layout/Structure

encode.c and decode.c are the main files, the rest of the files are auxiliary



See my collection of pseudocode below ↓↓

encode.c

```
static void write_tree( ... ) {
       // Make sure n exists
        // If this node is a leaf {
            // Then write 'L' + n->symbol to outfile
            // And return
        }
        // Else this node is an interior node
        // So search to the left
        // And search to the right
        // Then Write 'I' to outfile
       // And return
}

int main( ... ) {
    // Parse command-line arguments

    // Create histogram
          // Count the occurrence of every byte read
          // Increment 0 and 255 to ensure a minimum of two elements

    // Construct Huffman tree

    // Initialize code table

    // Build codes

    // Construct a header using struct definition from header.h
    // Get information about infile
    // Calculate number of unique elements
    // Set header properties

    // Write header to outfile

    // Set permissions of outfile

    // Write tree to outfile

    // Read through infile a second time and compress it using code table
    // Seek back to start of infile
        // For every byte read ...
            // Write corresponding code to outfile
    // Flush remaining codes to outfile

    // Print compression statistics if verbose printing was enabled

    // Delete the Huffman tree

    // Free memory allocated for header

    // Close infile and outfile

    // Finished encoding! :)
}
```

huffman.c

```
Node *build_tree( ... ) {
    // Initialize a priority queue
    // Enqueue a node for every symbol in the histogram
    // Rearranging the prioty queue into a Huffman tree
    // While there are two or more nodes in the queue ...
        // Dequeue two nodes
        // Put the joined nodes back in the queue
    // Now there must be only 1 node in the queue
    // This node is the root of the Huffman tree
    // Delete the priority queue
    // Return the root (which contains all the data for the entire Huffman tree)
}

static void traverse( ... ) {
       // Make sure n exists
        // If this node is a leaf ...
            // Then c is the code for this node's symbol
        // Otherwise, this node is an interior node
        // Search to the left
        // Search to the right
    // Left and right child searched so go back up the tree
}

void build_codes( ... ) {
    // Create a place to store codes
    // Traverse tree to build code table
    traverse()
}

Node *rebuild_tree( ... ) {
    // Create a stack to temporarily store nodes
    // Nodes in stack s will be rearranged into a Huffman tree
    // For each byte of the tree ...
        // If byte is 'L' then ...
            // Create a node with the symbol immediately after the 'L'
            // And add it to the stack
            // Increment i to skip over symbol
        } else { // Else then byte is 'I'
            // An 'I' will always have at least two bytes that came before it
            // So we can remove two nodes from the stack
            // Join them together
            // And put the new parent node p back onto the stack
        }
    }
    // At this point there should be one node in stack s
    // This node is the root of the Huffman tree, and the root contains the entire tree
    // Pop the root from stack s
    // At this point we can delete s
    // Return the root node of the Huffman tree
}

void delete_tree( ... ) {
       // Make sure root exists
        // If this node is a leaf ...
        // (If this node has no children ... )
            // Delete node
            // And go back up tree to continue searching
        // Otherwise, this node is an interior node
        // So search to the left
        // Search to the right
        // Delete node
    }
    // When both left and right children have been searched,
    // Go back up tree to continue searching
}
```

io.c

```
int read_bytes( ... ) {
    // Loop calls to read() until nbytes read or no more bytes to read
        // Add number of bytes read to total
        // Adjust how many additional bytes need to be read
    return total bytes read
}

int write_bytes( ... ) {
    // Looping calls to write() until nbytes written or no more bytes to write
        // Add number of bytes written to total
        // Adjust how many additional bytes need to be written
    return total bytes written
}

bool read_bit( ... ) {
    // If the code buffer is empty ...
        // Then fill the code buffer up with code from the infile
        // Set code_top to (the number of bytes that were put into the code buffer)
* 8
        // But if the number of bytes read is zero ...
            // Then return false because there is no more code to be read
    // Now that we have put code from infile into the code buffer ...
    // Let's pass the bit at code_index back through the bit pointer
    // Increment code_index so that next time we come back, we read the next bit
    // If we have reached the end of code_buffer ...
        // Reset code_index and and code_top to zero
    // Assume there are more bytes to read and return true
}

void write_code( ... ) {
    // For every bit in code c ...
        // Get the ith bit of code c ...
        // Add it to code_buffer
        // If the code_buffer is now full ...
            // Write the bits in code_buffer to the outfile
            // Reset code_buffer
}

void flush_codes( ... ) {
    // If code buffer contains bits, let's write them to the outfile
    // First, calculate how many *bytes* of the code buffer should be written
        // If number of bits in code buffer is divisible by 8, simply write all
bytes
        // Otherwise, write an additional byte for the remainder bits
}
```

node.c

```
Node *node_create(...) {
    // Allocate memory for the node using malloc
    / As long as n exists ...
        // Set left and right children to NULL
        // Because a node should not have children when it is first created
        // Set symbol and frequency
    // Now return n
}

void node_delete(Node **n) {
    // If n and *n are not NULL ...
        // Free *n
        // And set *n to NULL
}

Node *node_join(Node *left, Node *right) {
    // Create a new parent node with the given nodes as its children
    // Allocate memory for the parent node
    // If malloc was successful and n is not NULL ...
        // Set left and right children
        // Set symbol
        // Set frequency as the sum of the children's frequencies
    // Now return n
}

void node_print(Node *n) {
     print symbol, frequency, and children
}
```

pq.c

```
PriorityQueue {
    head
    tail
    size
    capacity
    items
}

PriorityQueue *pq_create(uint32_t capacity) {
    // Allocate memory for q using malloc
    // If malloc succeeded and q is not NULL ...
        // Initialize queue properties
        // If calloc failed...
            // Free the memory allocated for q
            // Set pq to null
    // Now return q
}

void pq_delete(PriorityQueue **q) {
    // If q and *q exist and are not already NULL ...
        // Free memory allocated for q
        // Set *q to NULL
}

bool pq_empty(PriorityQueue *q) {
    // Return true if size is 0, false otherwise
}

bool pq_full(PriorityQueue *q) {
    // Return true if size is equal to capacity, else return false
}

uint32_t pq_size(PriorityQueue *q) {
    // Return the size of the priority queue given
}

leftof(uint32_t i, uint32_t capacity) {
    // Returns the position to the left of i
}

rightof(uint32_t i, uint32_t capacity) {
    // Returns the position to the right of i
}

bool enqueue(PriorityQueue *q, Node *n) {
    // Let's enqueue node n into priority queue q
    // First, check if q is full
        // If it is full, return false to indicate failure
    // Otherwise, search from tail to head for a spot to insert n
        // Examine the position to the left
        // If we are at the head, meaning there is no node to the left,
        // Or if the frequency of the node to the left
        // is less than or equal to the frequency of n ...
            // Insert node n at position i
        // Otherwise, shift the node at l to the right
        // And move to the left to continue searching
    }
}

bool dequeue(PriorityQueue *q, Node **n) {
    // If q is empty ...
        // Return false to indicate failure
    // Get the node at the head of the queue and store at the value of n
    // Decrease size by 1 because there is now 1 less node
    // Shift the head over 1 because the node at head has just been removed
    // Finally, return true to indicate success
}

int ndigits(uint64_t x) {
    // Count the number of digits in x
}

void pq_print(PriorityQueue *q) {
    print each node in q
}
```

code.c

```
Code code_init(void) {
    // Initialize properties of code c
}

uint32_t code_size(Code *c) {
    // The top will always reflect the size
}

bool code_empty(Code *c) {
    // Code is only empty if top is zero
}

bool code_full(Code *c) {
    // Code is full if top is equal to max size
}

bool code_push_bit(Code *c, uint8_t bit) {
    // Make sure code is not already full
    // Insert bit in c at index of top
    // Move top up
}

bool code_pop_bit(Code *c, uint8_t *bit) {
    // Make sure code is not empty
    // Move top down
    // Pass the value at index of top into address of bit
}

void code_print(Code *c) {
    // Print each bit in code c
}
```

Old notes from my design process:

```
I just wanna understand how encode is actually gonna work.
i want to be able to visualize how this is supposed to play out before i go about coding it


super basic overall idea:
// get data from infile
// encode and compress data
// write data to outfile

slightly more specific:
// read an input file
// find Huffman encoding for file
// use Huffman encoding to compress file and write compressed version to outfile


more specific set of instructions:
// read input file
// step through each character and compute histogram of file (count number of occurrences of each character)
// construct Huffman tree using histogram and priority queue
// construct a code table. you will need stack of bits to traverse Huffman tree
// traverse tree and dump an encoding of the Huffman tree to a file
// step through each character of the file again. this time, emit each character code to outfile


refined pseudocode:
Create Histogram
     Initialize array of 256 uint64_t values all set to zero
     Read through infile and increment values of histogram accordingly
     Increment the count of element 0 and element 255 by 1

Create Priority Queue
     For each symbol in histogram create node and insert it into priority queue.
     (the symbol and frequency is given by histogram, but what should the left and right node be? i suppose they should be nothing)
     (also, do we construct the priority queue within build_tree(), or before calling it? seems like it should be done within)
     UPDATE: it should be within build tree

construct Huffman Tree from Priority Queue using build_tree()
     while there are two or more nodes in the priority queue …
          dequeue 2 nodes
          the first will be left child node
          the second will be right child node
          join these two nodes together using node_join and add joined node back to priority queue
          (the frequency of the parent node should be the sum of the frequencies of the left and right child nodes)
     eventually, there will only be one node in the queue. this is the root
     return the root node of the tree you just built
     (wait, so the original priority queue we created gets transformed into the Huffman Tree? Seems like it)
     I guess when you return the root you are effectively returning the entire tree,
     because the root contains the addresses of all the locations of data in memory

use build_codes() to construct code table by traversing Huffman Tree. code table is an array of 256 Codes (Code is a stack of bits)
     initialize a new code c using code_init()
     starting at the root of Huffman Tree, perform post-order traversal (recursion?)
          if current node is a leaf ... (base case?)
               current code represents path to this node
               current code is code for this node's symbol
               so add it to code table
               return and go back up
          otherwise current node must be an interior node (recursive case?)
               so push 0 to code and recurse down left link
```

```
// read compressed file
// decompress file
// write decompressed file

// read the emitted (dumped) tree from input file. reconstruct the tree using a stack of nodes
// read the rest of the infile bit-by-bit, traverse down Huffman tree, and write characters to the outfile
```