Assignment 6 Design Document

Program Purpose

The program must perform compression using the Huffman algorithm. The guiding principle is that the symbols that appear with higher frequencies are encoded using a small number of bits, while symbols that appear with lower frequencies can use a higher number of bits. This allows the overall content to be compressed to a smaller size.

Originally proposed by Huffman, the program first creates a histogram, noting the frequency of each symbol in the text. This histogram is used to create a tree that we call the Huffman tree. As described above, the logic used in the construction of the tree is that the symbols that appear with higher frequencies are leaf nodes that have the shortest path to the root nodes. As the frequency reduces, the distance to the root node increases. This allows us to generate codes for each symbol, which are then used to encode the text in a compressed format.

The decode process reverses this logic. It recreates the Huffman tree from the information stored in the file. Each code that follows can then be replaced by the symbol to get the original file back.

Program Requirements

Encoder

1. Read through infile to construct a histogram.
2. Increment the count of element 0 and element 255 by one in the histogram.
3. Construct the Huffman tree using a priority queue.
4. Construct a code table by traversing the Huffman tree.
5. Construct a header, initializing the fields magic, permissions, tree_size, and file_size
6. Write the constructed header to outfile.
7. Write the constructed Huffman tree to outfile using dump_tree().
8. Starting at the beginning of infile , write the corresponding code for each symbol to outfile with write_code(). When finished with all the symbols, make sure to flush any remaining buffered codes with flush_codes().
9. Close infile and outfile.

Decoder

1. Read in the header from infile and verify the magic number.
2. Set the permissions for outfile using fchmod().
3. Read the dumped tree from infile into an array that is tree_size bytes long. Then, reconstruct the Huffman tree using rebuild_tree().
4. Read infile one bit at a time using read_bit().
5. Close infile and outfile.

Program Pseudocode

encode.c

1. Include header files

2. Usage Function
    - Input parameters:
        - exec_name: char *: Name of the program
    - Return: void

```
void usage(char *exec_name) {
        Print the usage message.
        return
}
```

3. Create the frequency table for the input file.
    - The file is read as bytes, and each byte is used as an index into the histogram. Each time the byte is encountered in the file, the frequency is incremented by 1.
    - To ensure that there are at least 2 nodes in the tree that's created from this histogram, the first and the last frequency is incremented by 1.
    - Input parameters:
        - infile: char *: Input file

        - h: uint64_t *: Pointer to the histogram
    - Return: void

```
void create_histogram(char *infile, uint64_t *h) {
    uint8_t buf[BLOCK]
    int num_bytes_read

    h[0] += 1
    h[ALPHABET-1] += 1
```

- ❖ Read the infile using read_bytes(). Use the byte value as an index to the histogram and increment the frequency by 1.

```
    While read_bytes() != 0
        Use the byte value as an index i to increment h[i] by 1.
}
```

4. The main function
    - Input parameters:
        - argc: int: Number of input arguments
        - argv: char **: The input arguments
    - Return: int: 0 in case of success, non-zero for failure

```
int main(int argc, char **argv) {
```

```
Parse the input options.
Obtain permissions for the input file using fstat.
Construct a histogram using create_histogram().
Build a Huffman tree using build_tree().

Construct a code table using build_codes().
Create a header for the output file.
    Set the magic number for the header.
    Set the permissions field for the header.
    Determine the tree size as (3*hist_size -1) and set the tree_size field of the header to this value.
    Set the file_size field to the size of the input file.
Write the header to the output file.
Go to the beginning of the input file, and re-read the file using read_bytes.
For each byte, identify the code created, and add it to the output buffer.
Once the buffer is full (block size), write to the output file.
Call flush_codes() to flush any codes that are not yet written to the output file.
If verbose == true
    Identify the size of the input and output files.
    Print this information, along with compression gain/loss percentage.
Close the input and output files.
Free the space allocated for the Huffman Tree.
```

```
}
```

<u>decode.c</u>

1. Include header files

2. Usage Function
    - Input parameters:
        - exec_name: char *: Name of the program
    - Return: void

```
void usage(char *exec_name) {
        Print usage message
        return
}
```

3. The main function
    - Input parameters:
        - argc: int: Number of input arguments
        - argv: char **: The input arguments
    - Return: int: 0 in case of success, non-zero for failure

```
int main(int argc, char **argv) {
        Parse the input options.
        Check that the input file exists, is readable, and the header field is correct.
```

```
    Set permissions using fchmod.
    Read the dumped tree from infile into a tree_size bytes long array.
    Reconstruct the Huffman tree using rebuild_tree.
  ❖ Read infile one bit at a time using read_bit().
    num_decoded_symbols = 0
    while number of decoded symbols < file_size
        If bit_value == 0
            Walk down to the left child of the current node.
        else if bit_value == 1
            Walk down to the right child of the current node.
        if leaf node
            Write leaf node's symbol to outfile.
            Reset the current node back to the root of the tree.
        Increment num_decoded_symbols.
    Close infile and outfile.
}
```

<u>**pq.c**</u>

1. Include pq.h and node.h.

```
struct PriorityQueue {
    Node **n
    uint32_t size
    uint32_t capacity
}
```

2. Constructor function for the priority queue
    - Input parameters:
        - capacity: uint32_t: Maximum capacity of the queue.
    - Return: PriorityQueue *: Pointer to the pq created

```
PriorityQueue *pq_create(uint32_t capacity) {
    Allocate memory for pq.
    q->n = Allocate memory for capacity * (Node pointers)
    q->size = 0
    q->capacity = capacity

    return q
}
```

3. Destructor function for the priority queue
    - Input parameters:
        - q: PriorityQueue **: Pointer to the pq to be deleted
    - Return: void

```
void pq_delete(PriorityQueue **q) {
    Free the memory.
    Set the pointer to NULL.
}
```

4. Check if the queue is empty or not.
    - Input parameters:
        - q: PriorityQueue *: Ptr to the queue to check
    - Return: bool: true if the queue is empty. False otherwise

```
bool pq_empty(PriorityQueue *q) {
    if q->size == 0
        return true
    return false
}
```

5. Check if the queue is full or not.
    - Input parameters:
        - q: PriorityQueue *: Ptr to the queue to check
    - Return: bool: true if the queue is full. False otherwise

```
bool pq_full(PriorityQueue *q) {
    if q->size == q->capacity
        return true
    return false
}
```

6. Return the current size of the queue.
    - Input parameters:
        - q: PriorityQueue *: Ptr to the queue to check
    - Return: uint32_t: Current size of the queue

```
uint32_t pq_size(PriorityQueue *q) {
    Return q->size
}
```

7. Enqueue a node in the pq.
    - Input parameters:
        - q: PriorityQueue *: Ptr to the queue to be updated
        - n: Node *: Ptr to the node to be added
    - Return: bool: False if the queue is full before addition. True otherwise

```
bool enqueue(PriorityQueue *q, Node *n) {
  ❖ Use insertion sort to enqueue the new node.
    If queue is full
        Return false.


    while q->nodes[i]->frequency < n->frequency
        Move nodes to the right.
    Insert the new node.
    Increment size of the queue by 1.
```

}

8. This dequeues a node from the pq.
   - The node dequeued has the highest priority of all the nodes in the pq.
   - Input parameters:
     - q: PriorityQueue *: Ptr to the queue to be updated
     - n: Node **: Will contain the ptr to the node that's dequeued.
   - Return: bool: false if the queue is empty before dequeue. True otherwise

```
bool dequeue(PriorityQueue *q, Node **n) {
    If queue is empty
        Return false.
    Dequeue the node with highest priority (lowest frequency).
    Reduce the size of the queue by 1.
}
```

9. Debug function to print the pq.
   - Input parameters:
     - q: PriorityQueue *: Ptr to the queue to be printed
   - Return: void

```
void pq_print(PriorityQueue *q) {
    for (uint32_t i = 0; i < q->size; i++) {
        printf("%u\t%ld\n", q->nodes[i]->symbol, q->nodes[i]->frequency)
    }
```

}

huffman.c

1. Include huffman.h and pq.h.

2. Construct a Huffman tree, given a computed histogram.
    - Input parameters
        - hist: uint64_t[]: Histogram of size ALPHABET to be used for the tree
    - Return: Node *: Pointer to the root node of the tree.

```
Node *build_tree(uint64_t hist[static ALPHABET]) {
    Construct Huffman tree using a priority queue by calling build_tree().
    For each symbol histogram where frequency > 0
        Create a node.
        Insert the node in the priority queue.
    while (number of nodes in queue >= 2)
        Dequeue a node. Call it the left node.
        Dequeue another node. Call it the right node.
        Join the nodes using node_join.
        Enqueue the joined parent node.
        parent_node->frequency = l->frequency + r->frequency
    Only one node is left in the priority queue.
    This is the root of the Huffman tree.
    Pop and return node.
}
```

3. Populates a code table, building the code for each symbol in the Huffman tree.
    - Input parameters:
        - root: Node *: Root node of the Huffman tree.
        - table: Code []: Code table
    - Return: void

```
void build_codes(Node *root, Code table[static ALPHABET]) {
    If the node is a leaf node
        Add the symbol to the code table.
    else
        Push 0 to the code c.
        Make a recursive call to build_codes with root->left.
        Pop a bit from the code.
        Push 1 to the code c.
        Make a recursive call to build_codes with root->right.
        Pop a bit from the code.
    return
}
```

4. Dumps the contents of the tree into a file. Leaf nodes are represented with the symbol L, followed by the symbol itself, and the interior nodes are represented with the symbol I.
    - Input parameters:
        - outfile: int: File descriptor of the output file
        - root: Node *: Root node of the Huffman tree.
    - Returns: void

```
void dump_tree(int outfile, Node *root) {
    Make a recursive call with root->left.
    Make a recursive call with root->right.
    If leaf node
        Write "L" to the outfile.
        Write the symbol of the node to outfile.
    Else // Interior Node
        Write I to the outfile.
}
```

5. Rebuilds the tree from the data read from the encoded file.
    - Any time 'L' is encountered, the next byte is a leaf node symbol.
    - 'I' represents an interior node that joins two leaf nodes.
    - Input parameters:
        - nbytes: uint16_t: Buffer size
        - tree: uint8_t []: Buffer to build the tree
    - Returns: Node *: Pointer to the root of the node

```
Node *rebuild_tree(uint16_t nbytes, uint8_t tree[static nbytes]) {
    Create a stack to help build the tree.
    Iterate over the contents tree_dump from 0 to nbytes.
    if content == 'L'                              (next element is leaf node)
        Create a new node with node_create().
        Push the created node onto the stack.
    else if content == 'I'                         (Interior node)
        Pop the stack once to get the right child.
        Pop again to get the right child.
        Join the left and right nodes with node_join.
        Push the joined parent node on the stack.
    Pop the stack to obtain the root node.
    Free the stack memory using stack_delete.
    Return the root node.
```

6. Delete the tree to free memory using postorder traversal
    - Input parameters:
        - root: Node **: Root * for the tree to be deleted
    - Returns: void

```
void delete_tree(Node **root) {
    Make a recursive call with root->left.
    Make a recursive call with root->right.
    Delete the node by calling node_delete().
}
```

7. Debug function to print the tree.
    - Input parameters:
        - root: Node **: Root * for the tree to be deleted
    - Returns: void

```
void print_tree(Node *node) {
        if (node == NULL) {
                return
        }
        if (node->left != NULL) {
                print_tree(node->left)
        }
        if (node->right != NULL) {
                print_tree(node->right)
        }
        printf("%c\n", node->symbol)
        return
}
```

node.c

1. Include node.h.

2. Node constructor function.
    - Allocates memory for a node, and sets the value of the symbol and frequency.
    - The left and right children are set to NULL, to indicate a leaf node.
    - Input parameters:
        - symbol: uint8_t: Set the symbol field of the node to this value
        - frequency: uint64_t: Set the frequency field of the node to this value
    - Returns: Node *: Pointer to the node that's created

```
Node *node_create(uint8_t symbol, uint64_t frequency) {
    Node *node = Allocate memory
    node->symbol = symbol
    node->frequency = frequency
    node->left = NULL
    node->right = NULL

    return node
}
```

3. Node destructor function.
    - Frees memory, and sets the pointer to null.
    - Input parameters:
        - n: Node **: Ptr to pointer to the node to be destroyed
    - Returns: void

```
void node_delete(Node **n) {
    Free the memory.
    Set the pointer to null.
}
```

4. Create a parent node, assigning the two input nodes as its children
    - Input parameters:
        - left: Node *: Node that will be the left child
        - right: Node *: Node that will be the right child
    - Returns: Node *: Pointer to the parent node

```
Node *node_join(Node *left, Node *right) {
    Node *parent = Allocate memory

    parent->left = left
    parent->right = right
    parent->frequency = left->frequency + right->frequency

    return parent
}
```

5. Debug function to verify that the nodes are created and joined correctly
    - Input parameters:
        - n: Node *: Pointer to the node to print/verify

- Returns: void

```
void node_print(Node *n) {
    printf("%ld\n", n->frequency)
}
```

code.c

1. Include code.h.

2. Constructor function.
    - Creates a new code on the stack.
        - Input parameters: None
    - Returns: Code: Returns the initialized Code

```
Code code_init(void) {
    Create a new code on the stack.
    Set the top to 0.
    zero out the array of bits.
}
```

3. Return the size of the Code
    - Input parameters:
        - c: Code *: Code whose size needs to be returned

    - Returns: uint32_t: Size of the code

```
uint32_t code_size(Code *c) {
    return (number of bits pushed onto the Code)
}
```

4. Check if the code is empty
    - Input parameters:
        - c: Code *: Code to check
    - Returns: bool: true if Code is empty, false otherwise

```
bool code_empty(Code *c) {
    Return true if c->top = 0 (and false otherwise).
}
```

5. Check if the code is full
    - Input parameters:
        - c: Code *: Code to check
    - Returns: bool: true if Code is full, false otherwise

```
bool code_full(Code *c) {
    Return true if c->top = ALPHABET. False otherwise
}
```

6. Set a specific bit in the Code to 1
    - Input parameters

        - c: Code *: Code that will be updated
        - i: uint32_t: Index that will be updated
    - Returns: bool: false if index is out of range, true otherwise

```
bool code_set_bit(Code *c, uint32_t i) {
    if (i >= ALPHABET) {
        return false
```

- ❖ Right shift by 3 is the same as division by 8, and an "AND" with mask 0x7 is the same as the remainder with 8.
- ❖ Since we have an array of 32 uint8_t elements, this should set the desired bit to 1.

  Use bitwise OR with the above logic to set the desired bit to 1.

```
}
```

7. Set a specific bit in the Code to 0.
    - Input parameters
        - c: Code *: Code that will be updated
        - i: uint32_t: Index that will be updated
    - Returns: bool: false if index is out of range, true otherwise

```
bool code_clr_bit(Code *c, uint32_t i) {
    Use similar logic as the above function. However, to clear the bit, we will
    need to use bitwise AND.
}
```

8. Get the bit at the specified index in the given Code

- Input parameters
  - c: Code *: Code that will be read
  - i: uint32_t: Index to read
- Returns: bool: true if bit i is equal to 1, false otherwise

```
bool code_get_bit(Code *c, uint32_t i) {
    Again, use a similar logic as above, but check the value to see if it 0.
        Return false, if bit value is 0.
    Return true.
}
```

9. Push a bit onto the Code
   - Input parameters:
     - c: Code *: Code to push the bit to
     - bit: uint8_t *: bit to push
   - Returns: bool: false if the Code is full prior to pushing, true otherwise

```
bool code_push_bit(Code *c, uint8_t bit) {
    If code is full
        Return false.
    If bit == 0
        Clear the bit using code_clr_bit().
    Else
        Set the bit using code_set_bit().
    Update the top of the stack.
}
```

10. Pop a bit off the Code
    - Input parameters:
        - c: Code *: Code to pop the bit from
        - bit: uint8_t *: Popped bit
    - Returns: bool: false if the Code is empty prior to popping, true otherwise

```
bool code_pop_bit(Code *c, uint8_t *bit) {
    If code is empty
        Return false.
    Get the code using code_get_bit, and check the return value.
    If return value == false
        *bit = 0
    Else
        *bit = 1
    Update the top of the stack.
}
```

11. Debug function to verify if bits are pushed onto and popped off a Code correctly.
    - Input parameters:
        - c: Code *: Code to be printed.
    - Returns: void

```
void code_print(Code *c) {
        Print the bits in the stack.
}
```

io.c

1. Include io.h and code.h.

2. Read the contents from infile.
    - We create a wrapper around the read() system call that loops till the desired number of bytes (nbytes) are read.
    - Input parameters:
        - infile: int: File descriptor of the file to be read
        - buf: uint8_t *: Buffer containing the read bytes
        - nbytes: int: Number of bytes to be read
    - Returns: int: Number of bytes read

```
int read_bytes(int infile, uint8_t *buf, int nbytes) {
        ssize_t bytes_read = 0
        size_t bufSize = BLOCK * sizeof(uint8_t)
```

❖ Initialize the buffer to 0 to ensure that if the amount of data read is less than the buffer size, histogram is not corrupted.

```
        memset(buf, 0, bufSize)
        while (bytes_read < nbytes) {
                Use read to read nbytes number of bytes in the buffer
                if (num_bytes == 0)
                    ❖ End of file
                      break

                bytes_read += num_bytes
        }
        return bytes_read
}
```

3. Write the contents to outfile.
    - We create a wrapper around the write() system call, that loops till the desired number of bytes (nbytes) are written.
    - Input parameters:
        - outfile: int: File descriptor of the file to be written
        - buf: uint8_t *: Buffer containing the bytes to be written
        - nbytes: int: Number of bytes to be written
    - Returns: int: Number of bytes written

```
int write_bytes(int outfile, uint8_t *buf, int nbytes) {
        ssize_t bytes_written = 0
        while bytes_written < nbytes
        Use write() to write to the outfile.
        if (num_bytes == 0) {
            ❖ End of file
              break
        }
        bytes_written += num_bytes
    }
    Return bytes_written.
}
```

4. Read a block of bytes from infile into a buffer, and dole out one bit at a time, till all bits have been doled out.
    - Input parameters:
        - infile: int: File descriptor of the file to be read
        - bit: uint8_t *: doled out bit
    - Returns: bool: false if there are no more bits to be read, true otherwise.

```
bool read_bit(int infile, uint8_t *bit) {
    If the buffer is empty or there are no more bits to read
        Read upto BLOCK number of bits in the buffer.
    Use static bit and byte indexes. Use bit mathematics as in code_set_bit().
    and code_clr_bit() to identify the bit to be returned.
    Increment bit index by 1 module 8.
    Once the bit index is reset to 0, increment byte_index by 1.
}
```

5. Write bits from Code c into a buffer. Once the buffer is full, it will be written to the outfile.
    - Input parameters:
        - outfile: int: File descriptor of the file to be written
        - c: Code *: Code to be written to the buffer
    - Returns: void

```
void write_code(int outfile, Code *c) {
    For i = 0 to code_size
        Get the ith code bit using code_get_bit.
        Set the appropriate bit in code_buf based on the value.
        If the buffer is full
            Write it to the outfile, and reset it and the index.
}
```

6. Write out any leftover, buffered bits. Extra bits in the last bytes are zeroed out before the code is flushed.
    - Input parameters:
        - outfile: int
    - Returns: void

```
void flush_codes(int outfile) {
    The code buffer is already created by the write_code fn.
    Write it to outfile using write_bytes().
}
```

stack.c

1. Include stack.h

```
struct Stack {
    uint32_t top
    uint32_t capacity
    Node **items
};
```

2. Stack Constructor
    - The capacity is the maximum number of nodes that the stack can hold.
    - Input parameters:
        - capacity: uint32_t: Max number of nodes in the stack
    - Returns: Stack *: Pointer to the stack created

```
Stack *stack_create(uint32_t capacity) {
    Stack *s = Allocate memory for a stack
    s->items = Allocate memory for capacity node pointers
    s->top = 0
    return s
}
```

3. Stack Destructor.
    - Input parameters:
        - s: Stack **: Stack to be deleted
    - Return: void

```
void stack_delete(Stack **s) {
    for (uint32_t i = 0; i < (*s)->top; i++) {
        Free memory for the nodes.
    }
    Free memory for items.                    (ptr to ptr)
    Free memory for the stack and set the ptr to NULL.
}
```

4. Check if the stack is empty.
    - Input parameters:
        - s: Stack *: Stack to be checked
    - Returns: bool: true if the stack is empty. False otherwise

```
bool stack_empty(Stack *s) {
    if (s->top == 0) {
        return true
    }
    Return false.
}
```

5. Check if the stack is full.
    - Input parameters:
        - s: Stack *: Stack to be checked
    - Return: bool: True if the stack is full, and false otherwise.

```
bool stack_full(Stack *s) {
    if (s->top == s->capacity-1) {
        Return true.
    }
    Return false.
}
```

6. Return the size of the stack
    - Input parameters:
        - s: Stack *: Stack whose size is to be returned
    - Returns: uint32_t: Size of the stack

```
uint32_t stack_size(Stack *s) {
    return(s->top)
}
```

7. Push a node onto a stack.

- Input parameters:
    - s: Stack *: Stack to which the node must be pushed.
    - n: Node *: Node to be pushed
- Returns: bool: false if the stack is full before push, true otherwise

```
bool stack_push(Stack *s, Node *n) {
    if (stack_full(s)) {
        Return false.
    }
    s->items[s->top] = n
    s->top += 1
    Return true.
}
```

8. Pop a node from the stack.
    - Input parameters:
        - s: Stack *: Stack from which the node must be popped..
        - n: Node **: Node that's popped
    - Returns: bool: false if the stack is empty before push, true otherwise

```
bool stack_pop(Stack *s, Node **n) {
    if (stack_empty(s)) {
        Return false.
    }
    n = &(s->items[s->top])
    s->top--
    Return true.
}
```

9. Debug function to print the contents of a stack.
    - Input parameters:
        - s: Stack *: Stack to be printed.
    - Returns: void

```
void stack_print(Stack *s) {
    For i from 0 to s->top
        Print the symbol for s->items[i].
}
```