Deeraj Gurram

dgurram@ucsc.edu

7 November 2021

CSE 13S Fall 2021

Assignment 5 - Huffman Coding

Design Document

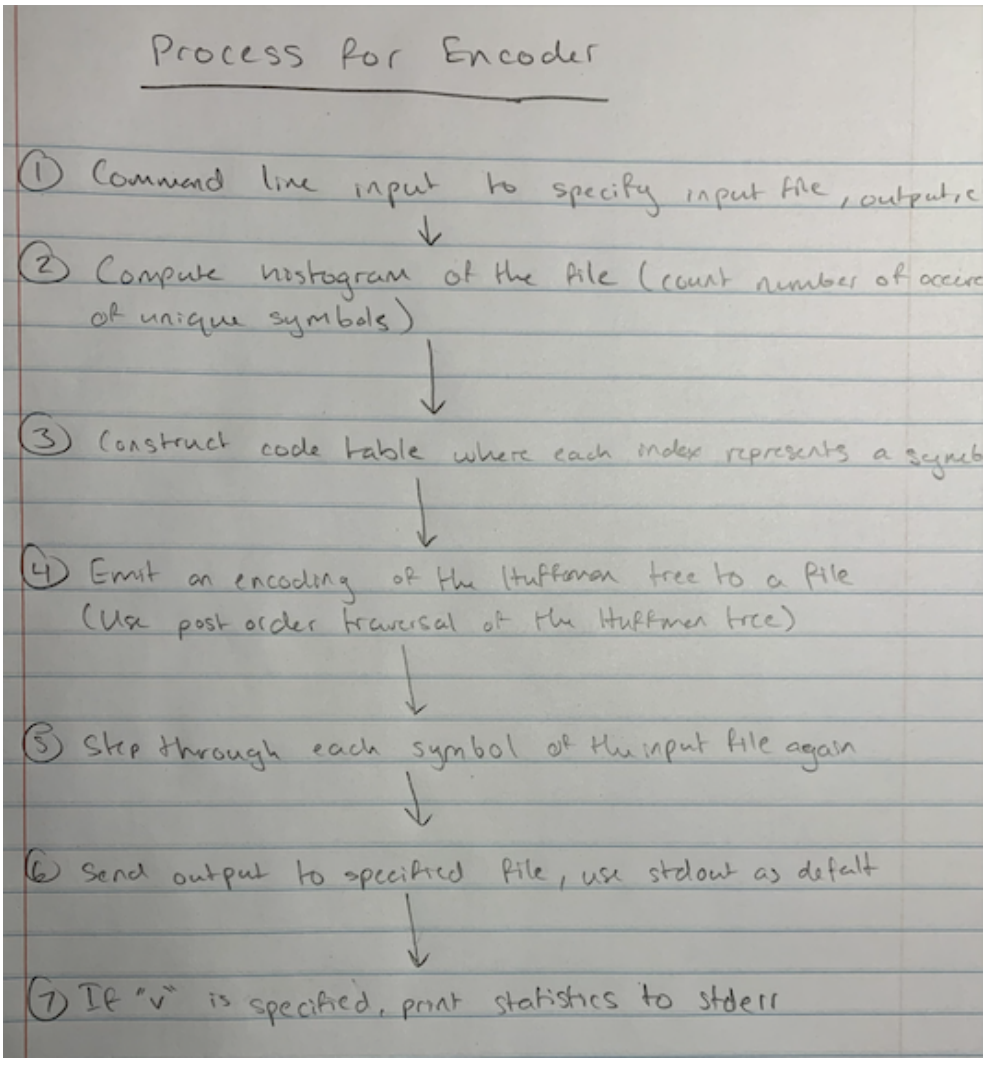
Description of the Program:

In this assignment, I'll be creating a Huffman encoder that reads file input, finds the Huffman encoding of its contents, and uses the encoder to compress the file. Included in the program files are a variety of functions necessary to read input, create a node ADT, create a Huffman encoder, utilize a priority queue ADT, traverse the Huffman tree using a stack ADT, as well as decode the encoded file using the stack ADT and recursion. Also included in the encoder program are command line options that take in command line input to specify different variables as well as a decoder program that similarly takes in command line input for various options.

Description of the Encoder:

The encoder will serve to read the file input, find the Huffman encoding of its contents, and use the encoding to compress the file. The encoder program, aptly name encode.c, supports any combination of the command line options “hiov”, where “h” prints the help message and program purpose, “i’ specifies the input file that must be encoded, “o” specifies the output file where the compressed input is stored, and “v” prints the compression statistics to stderr. The compression statistics include information about the uncompressed file size, compressed file size, and space saving which is calculated using the following formula:

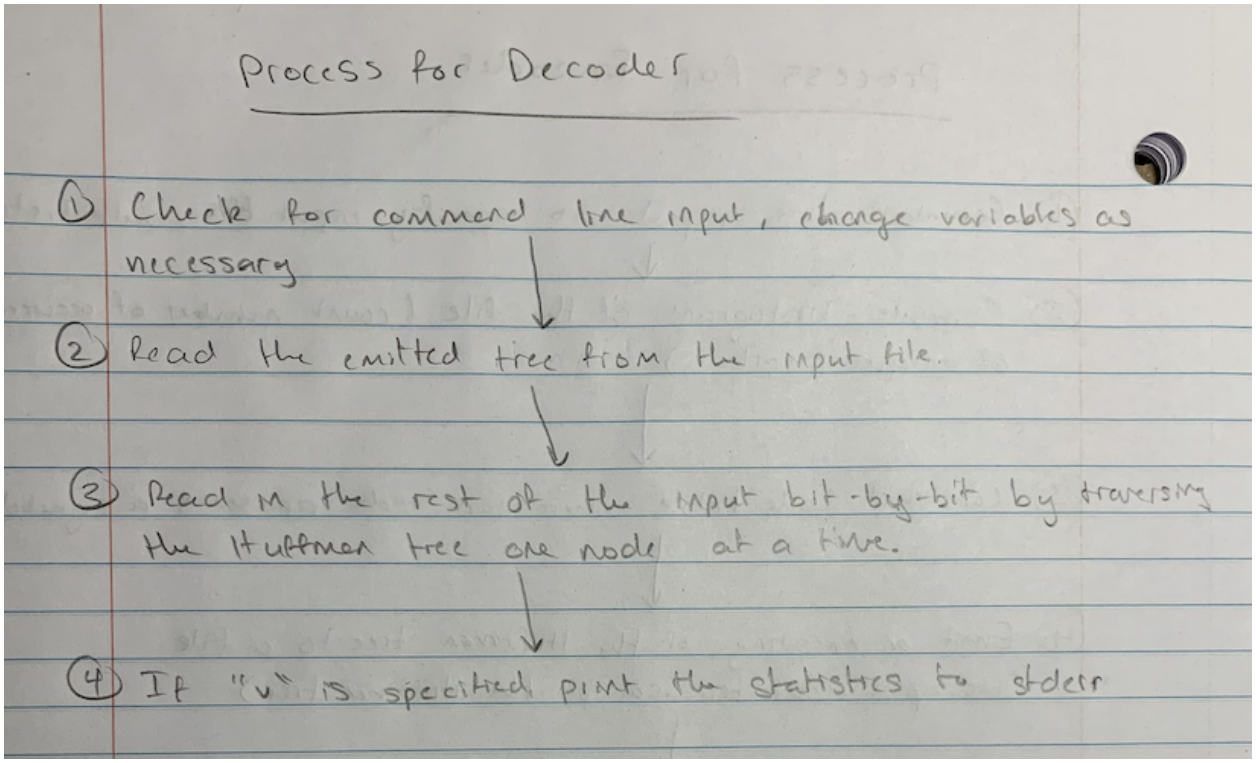
$$100 \times (1 - (\text{compressed size} / \text{uncompressed size}))$$



Description of the Decoder:

The decoder will serve to read in the compressed input file and decompress it, so it returns to its original uncompressed size. The decoder program named decode.c will support the same command line options as encode.c, though they will do mildly different things. The option “h” will print out the help message and program purpose, “i” will specify the compressed input file, “o” will specify the output file to write the decompressed input into, and “v” will print the decompression statistics to stderr. These statistics include the compressed file size, decompressed file size, and space saving which will be calculated using the following formula:

$$100 \times (1 - (\text{compressed size} / \text{decompressed size}))$$



Deliverables:

1. node.c
    - Node *node_create(uint8_t symbol, uint64_t frequency) is the constructor for a node. It sets the node's symbol as symbol and the frequency as frequency
    - void node_delete(Node **n) is the destructor for a node. It frees the allocated memory for a node and sets the pointer to NULL.
    - Node *node_join(Node *left, Node *right) joins the left child node and right child node, and returns a pointer to a created parent node.
    - void node_print(Node *n) is a debug function to verify that the nodes are created and joined properly.
2. pq.c
    - PriorityQueue *pq_create(uint32_t capacity) is the constructor for the priority queue. It sets the queue's max capacity as specified by capacity
    - void pq_delete(PriorityQueue **q) is the destructor for the priority queue. It frees the allocated memory for a priority queue and sets the pointer to NULL.
    - bool pq_empty(PriorityQueue *q) returns true if the priority queue is empty and false otherwise.
    - bool pq_full(PriorityQueue *q) returns true if the priority queue is full and false otherwise.
    - uint32_t pq_size(PriorityQueue *q) returns the number of items in the priority queue.
    - bool enqueue(PriorityQueue *q, Node *n) Enqueues a node into the priority queue.

- bool dequeue(PriorityQueue *q, Node **n) Dequeues a node from the priority queue and pushes it back through the double pointer n.
- void pq_print(PriorityQueue *q) prints the priority queue.

3. code.c
    - Code code_init(void) creates a new Code on the stack and sets the top to 0 while zeroing out the array of bits.
    - uint32_t code_size(Code *c) returns the size of the Code.
    - bool code_empty(Code *c) returns true if Code is empty and false otherwise.
    - bool code_full(Code *c) returns true if Code is full and false otherwise.
    - bool code_set_bit(Code *c, uint32_t i) sets the bit at index i in the Code to 1.
    - bool code_clr_bit(Code *c, uint32_t i) clears the bit at index i and sets it to 0.
    - bool code_get_bit(Code *c, uint32_t i) gets the bit at index i in the Code.
    - bool code_push_bit(Code *c, uint8_t bit) pushes a bit onto the Code.
    - bool code_pop_bit(Code *c, uint8_t *bit) Pops a bit off of the Code.
    - void code_print(Code *c) helps verify whether or not bits are pushed and popped correctly in a Code.
4. io.c
    - int read_bytes(int infile, uint8_t *buf, int nbytes) reads the exact number of bits in the input file and stores it into the byte buffer buf.
    - int write_bytes(int outfile, uint8_t *buf, int nbytes) loops calls to write() until all the bits in buffer buf have been written to the outfile or no bytes were written.
    - bool read_bit(int infile, uint8_t *bit) maintains a static buffer of bytes and an index into the buffer that tracks which bit to return using the pointer bit. The buffer will store BLOCK number of bytes as defined in defines.h.

- void write_code(int outfile, Code *c) buffers each bit in the code.c into the buffer. When the buffer of BLOCK bytes is filled, the contents of the buffer will be written to the outfile.
- void flush_codes(int outfile) writes out any leftover, buffered bits.

5. stack.c
    - Stack *stack_create(uint32_t capacity) is the constructor for the stack. The max number of nodes the stack can hold is specified by capacity.
    - void stack_delete(Stack **s) is the destructor for the stack. Frees the allocated memory of a stack and sets the pointer to NULL.
    - bool stack_empty(Stack *s) returns true if the stack is empty and false otherwise.
    - bool stack_full(Stack *s) returns true if the stack is full and false otherwise.
    - uint32_t stack_size(Stack *s) returns the number of nodes in the stack.
    - bool stack_push(Stack *s, Node *n) pushes a node onto the stack.
    - bool stack_pop(Stack *s, Node **n) pops a node from the stack.
    - void stack_print(Stack *s) prints the contents of the stack.
6. huffman.c
    - Node *build_tree(uint64_t hist[static ALPHABET]) constructs a Huffman tree given a computed histogram.
    - void build_codes(Node *root, Code table[static ALPHABET]) populates a code table and builds the code for each symbol in the Huffman tree.
    - void dump_tree(int outfile, Node *root) conducts a post-order traversal of the Huffman tree rooted at root and writes to the outfile.
    - Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes]) reconstructs a Huffman tree given the post-order tree dump stored in the array tree-dump.

- void delete_tree(Node **root) is the destructor fo the Huffman tree. Requires a post-order traversal to free all the nodes. Sets the pointers to NULL after freeing the allocated memory.

7. encode.c
    - Functions unknown - I haven't coded this program file yet
    - The purpose of this file is to encode the given input file using Huffman encoding and store the result in a compressed input file
8. decode.c
    - Functions unknown - I haven't coded this program file yet
    - The purpose of this file is to decode the given compressed input file and store the result in an output file.
9. node.h
    - This fill will contain the node ADT interface.
10. pq.h
    - This file will contain the priority queue ADT interface.
11. code.h
    - This file will contain the code ADT interface.
12. io.h
    - This file will contain the I/O module interface.
13. stack.h
    - This file will contain the stack ADT interface.
14. huffman.h
    - This file will contain the Huffman coding module interface.
15. defines.h
    - This file will contain the macro definitions used throughout the assignment
16. header.h

- This file will contain the struct definition for a file header.

17. Makefile
    - Builds the encoder and decoder as well as all program files, removes files that are compiler generated, and formats all source code including header files.
18. README.md
    - Gives a brief description of the program and Makefile and also lists the command line options that the program accepts. Also notes any errors or bugs found in the code and gives credit to sources of code or help.
19. DESIGN.pdf
    - Provides a description of the assignment and program files, the deliverables, and pseudocode for each of the .c files. Also describes the design and design process for the program.

Pseudocode:

node.c

Node *node_create(uint8_t symbol, uint64_t frequency)

Allocate memory to a Node n of size Node

Set the node's symbol as the argument symbol

Set the node's frequency as the argument frequency

Return the newly created Node

void node_delete(Node **n)

Free the allocated memory of the Node n

Set the pointer n equal to NULL

Node *node_join(Node *left, Node *right)

Set a Node called parent equal to the symbol '$' and the added frequencies of left and right

// This is the frequency of the parent node

Return the newly created parent node

void node_print(Node *n)

Print the node's symbol

Print the node's frequency

// This will allow us to verify that the left and right node's frequencies have been properly

// added. Printing the parent node's symbol will make sure they are '$' as specified

pq.c

Initialize a struct with members

size

capacity

A Node called items

PriorityQueue *pq_create(uint32_t capacity)

Allocate memory to a PriorityQueue q of size PriorityQueue

Allocate memory to a Node items of size Node

Set the PriorityQueue's capacity to the argument capacity

Set the PriorityQueue's size to 0

Return the newly created PriorityQueue

// This creates the PriorityQueue and allocates the appropriate memory

void pq_delete(PriorityQueue **q)

Free the array of Nodes items

Free the PriorityQueue q

Set the pointer of q to NULL

// This properly deletes the PriorityQueue and frees the allocated memory

bool pq_empty(PriorityQueue *q)

Check if q's size is equal to 0

    Return true

Else

    Return false

// This returns true if the PriorityQueue is empty

bool pq_full(PriorityQueue *q)

Check if q's size is equal to q's capacity

Return true

Else

Return false

// This returns true if the PriorityQueue is full

uint32_t pq_size(PriorityQueue *q)

Return q's size

// Returns the PriorityQueue's size

uint32_t min_child(PriorityQueue *q, uint32_t first, uint32_t last)

Initialize a variable called left to hold the left child's value and set it equal to 2 * first

Initialize a variable called right to hold the right child's value and set it equal to left + 1

Check if right is less than or equal to last and the frequency of q's (right child - 1) is less than the frequency of q's (left child - 1)

Return the variable right

// This allows us to return the largest values and place them at the top

// of the heap

Else

Return the variable left

void fix_heap(PriorityQueue *q, uint32_t first, uint32_t last)

Initialize a boolean found and set it equal to false

Initialize a variable mother and set it equal to argument first

Initialize a Node temp

Initialize a variable least and set it equal to function min_child(q, mother, last)

Create while loop to swap elements in the tree based on if mother is less than or equal to least, make sure mother is less than or equal to last divided by 2, and that found is false

Check if the frequency of q's (mother - 1) is greater than the frequency of q's (least - 1)

Set temp equal to q's items of mother - 1

Set q's items of mother - 1 equal to q's items of least - 1

Set q's items of least - 1 equal to temp

Set mother equal to least

Set least equal to min_child(q, mother, last)

// This checks the next children in the array items

Else

Set found equal to true

bool enqueue(PriorityQueue *q, Node *n)

Initialize a variable k

Initialize a variable parent

Check if q is full

Return false

Else

Set q's items of q's size equal to Node n

// If the array of items is empty, just set the 0th element to Node n

Increment q's size by 1

Set k equal to q's size

Create a while loop where k must be greater than 1

Set variable parent equal to k / 2

Call the function fix_heap(q, parent, q's size)

Set k equal to parent

Return true

// This function adds a Node n to the array items and fixes the heap so the PriorityQueue

// is properly sorted

bool dequeue(PriorityQueue *q, Node **n)

Initialize a Node temp2

Initialize a variable k

Initialize a variable parent

Check if the PriorityQueue is empty

Return false

Set the pointer to n equal to the first element of q's items array

// This stores the dequeued element in the pointer

Set temp2 equal to q's items of element 0

Set q's items of element 0 equal to q's items of size - 1

Set q's items of size - 1 equal to temp2

// This effectively swaps the values of the first and last elements in the items array

Decrement q's size by 1

Set variable k equal to q's size

Create a while loop where k must be greater than 1

Set variable parent equal to k / 2

Call the function fix_heap(q, parent, q's size)

Set k equal to parent

Return true

void pq_print(PriorityQueue *q)

Create a for loop where the iterator starts at 0 and ends when it reaches q's size

Print the frequency of each item inside of the items array

code.c

Code code_init(void)

Initialize a Code c

Set c's top value equal to 0

Create a for loop to go through the struct member bits where the iterator starts at 0 and stops when it reaches the top value of c

Set the i'th value of c's array of bits to 0

Return c

uint32_t code_size(Code *c)

Return the top value of Code c

// The top value indicates the length of the array which holds the number of pushed bits

bool code_empty(Code *c)

Check if the top value of Code c is equal to 0

Return true

Else

Return false

// If the top value of the Code c is 0, then there aren't any bits inside of array bits

bool code_full(Code *c)

Check if the top value of Code c is equal to ALPHABET

// ALPHABET is the number of ASCII characters and is defined in defines.h

Return true

Else

Return false

// This function checks if the Code c's bits array is full

bool code_set_bit(Code *c, uint32_t i)

Check if argument i is less than 0, or greater than c's top value * 8

Return false

Else

Set c's bits value of element i divided by 8 equal to the number 1 right shifted by the argument i modded by 8

Return true

// This function sets all the bits in array bits equal to 1 if the iterator is within the

// acceptable range

bool code_clr_bit(Code *c, uint32_t i)

Check if argument i is less than 0, or greater than c's top value * 8

Return false

Else

Set c's bits value of element i divided by 8 equal to the The opposite of (number 1 right shifted by the argument i modded by 8)

Return true

// This function sets all the bits in array bits equal to 0 if the iterator is within the

// acceptable range

bool code_get_bit(Code *c, uint32_t i)

Check if argument i is less than 0, or greater than ALPHABET, or the number 1 right

shifted by the argument i modded by 8 equals 0

Return false

Else check if the number 1 right shifted by the argument i modded by 8 equals 1

Return true

Return false

// This function checks if the specified bits array element i is equal to 1

bool code_push_bit(Code *c, uint8_t bit)

Check if the top value of Code c is equal to ALPHABET

Return false

Else check if argument bit is greater than 0

Call the function code_set_bit(c, c's top value)

Increment c's top value by 1

Else check if the bit value equals 0

Call the function code_clr_bit(c, c's top value)

Increment c's top value by 1

Return true

// This function pushes a bit into the array bits if the array isn't already full

bool code_pop_bit(Code *c, uint8_t *bit)

Check if the top value of Code c is equal to 0

Return false

Else

Decrement c's top value by 1

Set the pointer to bit equal to code_get_bit(c, c's top value)

Return true

// This function pops a bit from the array bits and stores it in the memory address of

// argument bit

void code_print(Code *c)

Print c's top value

Create a for loop to go through the array bits, set i equal to 0, where i is less than the top value of Code c, increment in steps of 1

Print out the value of c's bits[i]

// This function verifies whether the or not the bits are pushed and popped from the Code

// c correctly

io.c

Create an extern integer named bytes_read

Create an extern integer named bytes_written

Initialize a static integer index equal to 1

Initialize a static integer end equal to -1

Create a static int buffer of size BLOCK

int read_bytes(int infile, uint8_t *buf, int nbytes)

Initialize a variable bytes to 0

Set variable bytes_read to 0

Create a do while loop

Set variable bytes equal to read(argument infile, argument buf + bytes_read, argument nbytes - bytes_read)

Increment bytes_read by bytes

The loop should end when the value of bytes equals 0 or less

Return the variable bytes_read

// This function reads bytes into the buffer until there aren't any bytes to read remaining // in the infile

int write_bytes(int outfile, uint8_t *buf, int nbytes)

Initialize a variable bytes to 0

Set variable bytes_written to 0

Create a do while loop

Set variable bytes equal to read(argument infile, argument buf +bytes_written, argument nbytes - bytes_written)

Increment bytes_written by bytes

The loop should end when the value of bytes equals 0 or less

Return the variable bytes_written

// This function writes bytes from the buffer into the outfile until there aren't any bytes to

// write remaining in the buffer

bool read_bit(int infile, uint8_t *bit)

Check if the index equals 0

Set integer bytes equal to the function read_bytes(infile, buffer, BLOCK)

Check if the variable bytes is less than BLOCK

Set the variable end equal to (bytes * 8) + 1

Set the pointer to bit equal to the (index / 8)th element in buffer that is bitwise ANDed

with the number 1 right shifted by (the index modded by 8)

Increment the index by 1

Check if the index equals BLOCK * 8

Set the index to 0

Return index doesn't equal end

// This function returns a bit out of the buffer

void write_code(int outfile, Code *c)

Create a for loop where the iterator starts at 0 and stops when it reaches the

MAX_CODE_SIZE

Initialize a variable bit and set it equal to code_get_bit(c, iterator)

Check if bit equals 1

Call the function code_set_bit(c, iterator)

Else

Call the function code_clr_bit(c, iterator)

Increment the index by 1

Check if index doesn't equal end

Call the function flush_codes(outfile)

// This function writes the bits of the Code c to the outfile

void flush_codes(int outfile)

Create an integer nbytes

Check if index is greater than 0

Set nbytes equal to ((index / 8) * (index modded by 8 doesn't equal 0)

Call the function write_bytes(outfile, buffer, nbytes)

// This function writes out any remaining buffered bits

stack.c

Create a struct for Stack with members

top

capacity

A Node called items

// These members will be used to see/manipulate the top element of the stack, check the

// size of the stack, and go through the items Node to manipulate the next value of the

// stack

Stack *stack_create(uint32_t capacity)

Allocate memory to a Stack s of size Stack

Check if s is true

Set the struct member top equal to 0

Set the struct member capacity equal to the argument capacity

Set the struct member items equal to a dynamically allocated memory of size capacity to

a Node *

Check if the struct member items is not true

Free the allocated memory of stack s

Set s equal to NULL

// This will free the allocated memory of Node items

Return the stack s

// This function is used to create a Stack of size Stack as well as a Node of size Node,

// unless the Node items doesn't exist, in which case the allocated memory is freed

void stack_delete(Stack **s)

Check if the pointer to s and the Node items are true

Free the memory allocation of the struct member items

Free the memory allocation of *s

Set *s equal to NULL

Return

// This function deletes the stack by freeing the allocated memory of the Node, then

// freeing the allocated memory of the Stack s before setting s to NULL

bool stack_empty(Stack *s)

Check if the top of the Stack s is equal to 0

Return true

Else

Return false

// This function checks if the stack is empty, useful in other functions

bool stack_full(Stack *s)

Check if the top value of the Stack s is equal to the maximum capacity of s

Return true

Else

Return false

// This function checks if the stack is full, useful in other functions

uint32_t stack_size(Stack *s)

Return the top value of the Stack s

// Returns the size of the stack

bool stack_push(Stack *s, Node *n)

Check if the top value of the Stack s is equal to the maximum capacity of s

Return false

Else

Set the Stack's top value inside of the Stack's Node items equal to n

Increment the Stack's top value by 1

Return true

// This function first checks if the stack is already full, then executes the rest of the code

// if it isn't full yet. The rest of the code pushes a value specified in the arguments to the

// Node items inside of the Stack s

bool stack_pop(Stack *s, Node **n)

Check if the Stack's top value is equal to 0

Return false

Else

Decrement the Stack's top value by 1

Set the pointer in memory of n equal to the Stack's top value inside of the Stack's

Node items

Return true

// This function first checks if the stack is empty, then executes the rest of the code

// if it isn't empty. The rest of the code pops a value specified in the arguments from the

// Node items inside of the Stack s and stores it inside of the memory address of n

void stack_print(Stack *s)

Print the stack's capacity

Print the stack's top value

Create a for loop to go through the top values of Stack s, set i equal to 0, where i is less than the top value of s, increment in steps of 1

Print the i'th value of the Node item's frequency inside of the stack to the outfile

// This function debugs the Stack s by printing its contents

huffman.c

Create a static Code c

Create a static boolean init and set it equal to false

Node *build_tree(uint64_t hist[static ALPHABET])

Create a Node n

Initialize an integer count and set it equal to 0

Create a for loop where the iterator starts at 0 and is less than ALPHABET

Check if hist[iterator] is greater than 0

Increment the count variable by 1

// This will serve to initialize A PriorityQueue of size count

Create a PriorityQueue q of size count

Create a for loop where the iterator starts at 0 and is less than ALPHABET

Set n equal to function node_create(iterator, hist[iterator])

Enqueue the node n onto q

Create a while loop where the size of q must be greater than 1

Dequeue the node n from q

Set Node left equal to Node n

Dequeue the node n from q

Set Node right equal to Node n

Set the Node parent equal to the function node_join(left, right)

Enqueue the parent node to q

Dequeue the Node n

Set Node root equal to Node n

Return the root

// This function constructs a Huffman tree given a computed histogram.

void build_codes(Node *root, Code table[static ALPHABET])

Create an integer bit and set it equal to 0

Check if boolean init equals false

Set c equal to code_init()

Set init equal to true

Check if root doesn't equal NULL

Check if root's left node doesn't equal NULL and root's right node doesn't equal NULL

Set the element root's symbol of the table equal to c

Else

Call the function code_push_bit(c, 0)

Call the function build_codes(root's left node, table of size ALPHABET)

Call the function code_pop_bit(c, bit)

Call the function code_push_bit(c, 1)

Call the function build_codes(root's right node, table of size ALPHABET)

Call the function code_pop_bit(c, bit)

// This function populates a code table with each element being a symbol

void dump_tree(int outfile, Node *root)

Initialize an integer l and set it equal to 'L'

Initialize an integer i and set it equal to 'I'

Check if root

// Recursively call cump_tree on the left and right nodes

dump_tree(outfile, root's left node)

dump_tree(outfile, root's right node)

Check if root's left node equals NULL and root's right node equals NULL

Call the function write_bytes(outfile, l, 1)

Call the function write_bytes(outfile, the root's symbol, 1)

Else

Call the function write_bytes(outfile, i, 1)

Node *rebuild_tree(uint16_t nbytes, uint8_t tree[static nbytes])

Create a Stack s with capacity nbytes

Create a for loop to go through all the nbytes

Check if the current element of the tree equals 'L'

Create a Node leaf equal to node_create(tree[iterator + 1], 0)

Push the leaf onto the stack s

Else check if the current element of the tree equals 'I'

Create a Node interior

Pop the interior value from the stack s

Create a Node right and set it equal to interior

Pop the interior node from the stack s

Create a Node left and set it equal to interior

Create a Node parent and set it equal to node_join(left, right)

Push the parent node onto the stack

Create a node root

Pop the value of root from the stack s

Return the root

void delete_tree(Node **root)

Check if root

// Recursively call delete_tree on the left and right nodes to free up the allocated

// memory for all the nodes in the tree

delete_tree(root's left node)

delete_tree(root's right node)

Call the function node_delete(root)

encode.c

void usage(char *exec)

Use the fprintf function with arguments (stderr, the help message, exec)

// This allows us to print the error message and change the name of the file inside the

// help message

int main(int argc, char **argv)

Create an integer opt and set it equal to 0

Create an integer infile

Create an integer outfile

Create a while loop to go through all command line inputs that were specified, the loop should make sure (opt equals getopt() which has arguments argc, argv, OPTIONS) and that this whole thing doesn't equal -1. The -1 signals that all command line options have been read

Create a switch statement with argument opt

Case 'i'

Check if the (infile equals the function open(optarg, read | trunc | creat))

== -1

Print the error message to the stderr

Return an EXIT_FAILURE

Case 'o'

Check if the (infile equals the function open(optarg, write | trunc | creat))

== -1

Print the error message to the stderr

Return an EXIT_FAILURE

Case 'v'

Case ‘h’

Call the function usage(argv[0])

Return an EXIT_SUCCESS

Initialize an array of 64 bit integers calle hist of size ALPHABET

Initialize a 32 bit integer called unique_syms and set it equal to 0

Initialize an 8 bit integer array named buffer of size BLOCK and initialize the elements to 0

Initialize an integer named bytes_read

Increment the first element of hist by 1

Increment the last element of hist by 1

// This while loop reads through the input file and properly populates the histogram

Create a while loop where bytes_read equals read_bytes of infile, buffer, BLOCK and must be greater than 0

Create a for loop where the iterator starts at 0 and must be less than bytes_read

Increment the unique_sums by 1

Increment hist[buffer[i]] by 1

// Set the file permissions for the outfile

Call the struct stat and call this instance header

Call the function fstat(infile, header)

Call the function fchmod(outfile, header’s mode)

// These functions duplicates the permissions of the infile to the outfile

Initialize the struct members of Header h to all 0

Set the struct member magic equal to MAGIC

Set the struct member permissions equal to header's permissions

Set the struct member tree_size to 3 * unique_syms - 1

Set the struct member file_size to the header's size

Call the function write_bytes(outfile, h, the size of the header h)

// Now we need to build the tree using the populated histogram

Create a Node root and set it equal to function build_tree(hist)

//Initialize a code table

Create a Code called table of size ALPHABET and initialize the array elements to 0

// Build the Code table

Call the function build_codes(root, table)

// Dump the tree

Call the function dump_tree(outfile, root)

// Write the codes

Call the function lseek(infile, 0, SEEK_SET)

Create a while loop where bytes_read equals read_bytes of infile, buffer, BLOCK and must be greater than 0

    Create a for loop where the iterator starts at 0 and must be less than bytes_read

        Call the function write_code(outfile, table of size buffer of i)

// Flush the remaining bits from the code table

Call the function flush_codes(outfile)

// Print the compression statistics

Check if statistics is true

    Print the bytes_read

    Print the bytes_written

    Print the formula 100×(1−(compressed size/uncompressed size))

// Delete the tree and free the allocated memory

Call the function delete_tree(root)

Close the infile and the outfile

decode.c

Create an integer opt and set it equal to 0

Create an integer infile

Create an integer outfile

Create a while loop to go through all command line inputs that were specified, the loop should make sure (opt equals getopt() which has arguments argc, argv, OPTIONS) and that this whole thing doesn't equal -1. The -1 signals that all command line options have been read

Create a switch statement with argument opt

Case 'i'

Check if the (infile equals the function open(optarg, read | trunc | creat))

== -1

Print the error message to the stderr

Return an EXIT_FAILURE

Case 'o'

Check if the (infile equals the function open(optarg, write | trunc | creat))

== -1

Print the error message to the stderr

Return an EXIT_FAILURE

Case 'v'

Case 'h'

Call the function usage(argv[0])

Return an EXIT_SUCCESS

// Set the header permissions for the outfile

Initialize the struct members of Header h to 0

Call the function read_bytes(infile, h, size of the header h)

Check if the struct member magic doesn't equal MAGIC

    Print an error message to stderr

    Close the infile

    Close the outfile

    Exit the code

Call the function fchmod(outfile, header h's permissions)

// Print the decompression statistics

Check if statistics equals true

    Print the bytes_read

    Print the bytes_written

    Print the formula 100×(1−(compressed size/uncompressed size))

Credit:

1. Used the provided C code and pseudocode from the asgn5.pdf from Dr. Long
2. Used the assignment and program file descriptions from the asgn5.pdf from Dr. Long
3. Got the code for read_bytes and write_bytes from TA Eugene's section
4. Got the code for most of read_bit from TA Eugene's section
5. Used the pseudocode for the while loop in enqueue from TA Eugene's section
6. Used coding ideas from TA Christian Ocon's section to create struct members in pq.c
7. Used the bitwise shifts and operators code for code_set_bit, code_clr_bit, and code_get_bit from the example provided by Dr. Long in the CSE 13S Code Comments bv8.h file
8. Used the stack.c code from my asgn 4 stack.c. I used the python pseudocode from Dr. Long in asgn4 to create the functions.
9. Used TA Eugene's pseudocode for build_tree in huffman.c from his section
10. Based my build_codes function off of Dr. Long's python pseudocode
11. Based my dump_tree function off of Dr. Long's python pseudocode
12. Based the structure and setup of the command line options off of Dr. Long's tsp.c in asgn4
13. Got the idea to use O_TRUNC and O_CREAT from Elmer in Discord
14. Used code from TA Eric to populate the histogram, set the file permissions, and build the codes
15. Copied the error messages and help messages from the resources repo