Deeraj Gurram

dgurram@ucsc.edu

28 October 2021

CSE 13S Fall 2021

Assignment 5 - Huffman Coding

Design Document

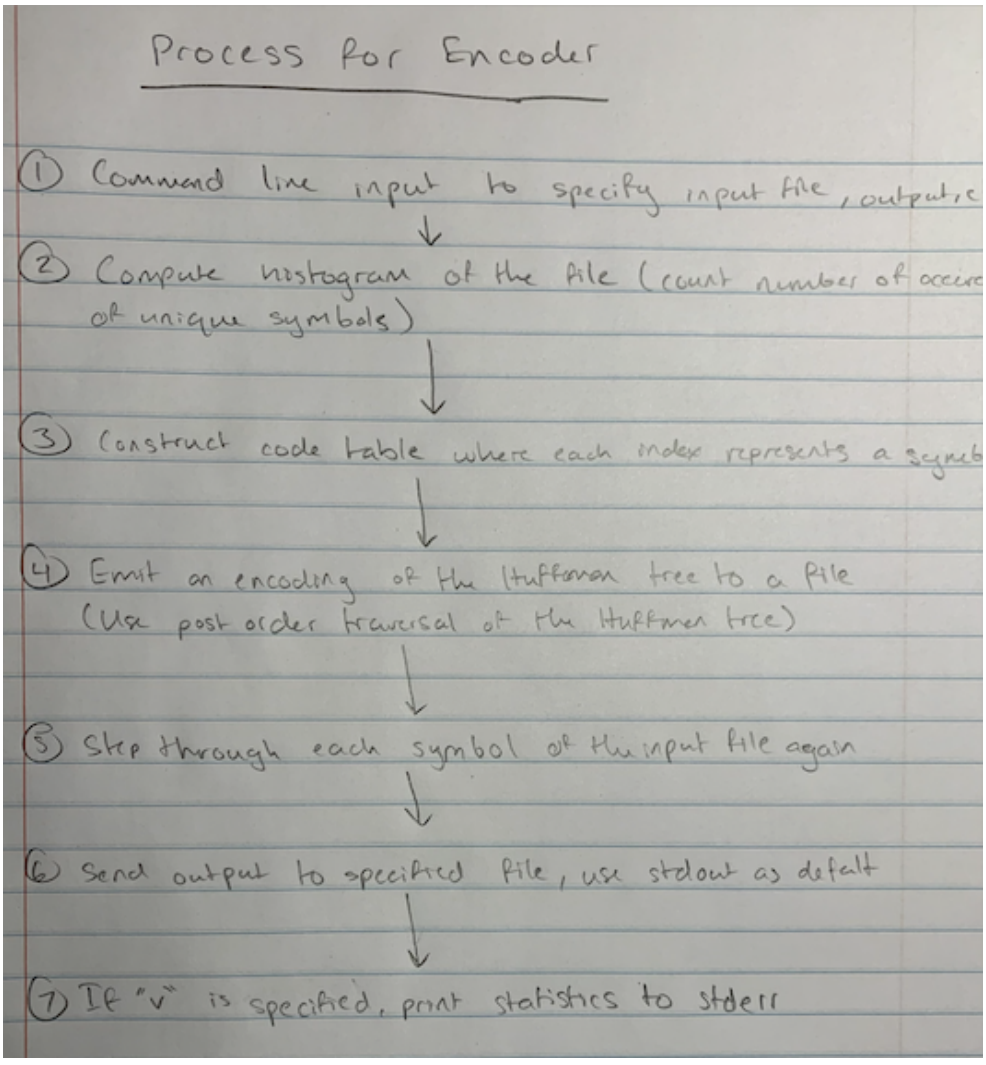
Description of the Program:

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:

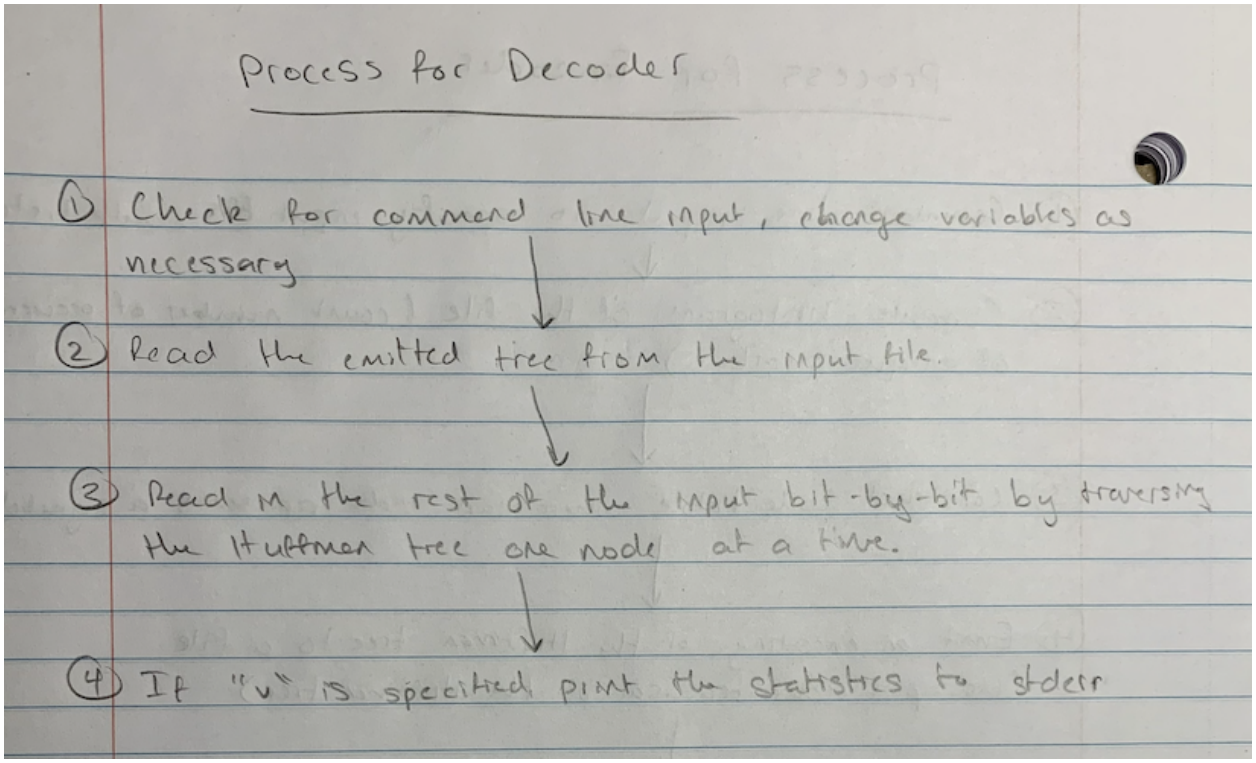
$$100 \times (1 - (\text{compressed size} / \text{uncompressed size}))$$



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 ot not bits are pushed and pooped 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

Create a struct Node with members

*left

*right

symbol

frequency

// These members will be used to traverse the Huffman tree in the left and right child,

// represent the symbols in the text, and display the frequency of the symbols

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

void node_delete(Node **n)

Check if the pointer to n, pointer to n's left child, and pointer to n's right child exist

Free the allocated memory of the Node n's left child

Free the allocated memory of the Node n's right child

Free the allocated memory of the Node n

Set the pointer n equal to NULL

// We need to free the left and right child because they are also Nodes due to the nature

// of a tree

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

Create a Node called parent

Add the frequencies of *left and *right and set it equal to the parent Node's frequency

// This is the frequency of the parent node

Set the parent node's symbol to '$'

void node_print(Node *n)

Print the left node's members

Print the right node's members

Print the parent node's members

// 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

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 32 bit integer

Check if the struct member items is not true

Free the allocated memory of stack s

Set stack 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

Check the Stack's top value inside of the Stack's Node items and set it 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)

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 items inside of the stack to the outfile

Print a comma and space

Print a newline at the end of the outfile

// This function debugs the Stack s by printing its contents to an outfile that is verifiable

code.c

Create a struct for Code with members

top

bits[MAX_CODE_SIZE]

// where MAX_CODE_SIZE is defined in defines.h

// These members will be used to look at the top value of a struct Code, and to manage

// the bits of the Code input

Code code_init(void)

Set the struct member top equal to 0

Create a for loop to go through the struct member bits

Set the i'th value of the array to 0

Return Code 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)

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

Check if i isn't within the range

Return false

Else

Set bits[i] equal to 1

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)

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

Check if i isn't within the range

Return false

Else

Set bits[i] equal to 0

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 isn't within the acceptable range

Return false

Else

Check if bits[i] equals 1

Return true

// 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

Set bits[top + 1] equal to argument bit

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

Set the pointer to bit equal to bits[top - 1]

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)

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 bits[i]

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

pq.c

Haven’t implemented any of the functions yet

io.c

Haven’t implemented any of the functions yet

huffman.c

Haven’t implemented any of the functions yet

encode.c

Haven’t implemented any of the code yet

decode.c

Haven’t implemented any of the code yet

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. Copied the error messages and help messages from the resources repo
4. Used the stack.c code from my assignment 4 stack.c