Keaton Shawhan
cruzID: kshawhan
Assignment 7

Assignment purpose:

The purpose of this assignment is to utilize huffman trees for the purpose of file compression. Creating these Huffman trees will require the use of min heaps, priority queues, stacks, a byte/bit parser, and character-frequency histograms, which will help build an understanding of how file compression works in C.

heap_help.c:

void swap(Node *n1, Node *n2)

Set n1's pointer to a temp pointer
Set n1 to n2
Set n2 to the temp pointer

uint32_t l_child(uint32_t n)

Return the index (n*2)+1

uint32_t r_child(uint32_t n)

Return the index (n*2)+2

uint32_t parent(uint32_t n)

Return the index (n-1)/2

void up_heap(uint32_t *arr, uint32_t n)

loop while the element index is > 0 and the array's frequency of the index n is less than the array's frequency of the parent function's return value

Swap the parent element and the current element at index n
Update the index n to the parent's index (parent() function)

void down_heap(uint32_t *arr, PriorityQueue *q)

initialize the element index n and the smaller value of the childs
loop while the left child function's return value of the element index is less than the current heap size

check if the right child function's return value is equal to the current heap size if so, update the smaller variable to the left child function's return value

If not, check if the array's value of the left child function's return value of the element index n is less than the array's value of the right child function's return value of the element index n

if so, update the smaller variable to the left child function's return value of the element index n

if not, update the smaller variable to the right child function's return value of the element index n

check if the array's value of element index n is less than the array's value of the smaller variable

If so, break out of the initial loop

swap the addresses of the array's value of the element index n with the array's value of the smaller variable

update the element index n with the smaller variable

node.c:

Create the Node struct

Initialize a variable for the node’s symbol in uint8_t

Initialize a variable for the symbol’s frequency in uint64_t

initialize a the next and previous nodes of type Node

Node *node_create(uint8_t symbol, uint64_t frequency)

Use malloc() to allocate the memory for the node itself

Make sure if the node exists

Initialize the left and right Node types in the struct

Set the symbol and frequency in the struct

Return the Node

void node_delete(Node **n)

Make sure that the node exists (not NULL)

Use the free() function to free the memory allocated for Node n

Set the pointer to the node to NULL

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

Make sure that both the left and right nodes exist (not NULL)

Call node_create() for the creation of the parent node, passing in ‘$’ as the symbol (type uint8_t to convert), and the sum of left->frequency and right->frequency

Make the parent point to the left and right children
Make sure that the node_create didn't return NULL (created correctly)
Return the parent node

void node_print(Node *n)
Make sure that the node itself isn't NULL
Check the type of print to use with iscntrl() and isprint()
Print out the symbol
Print out the frequency

bool node_cmp(Node *n1, Node *n2)
Check that both nodes exist (not NULL)
Check the frequencies of both nodes
Return True if the n1 frequency is greater than the n2

void node_print_sym(Node *n)
Check that the node exists (not NULL)
Check the type of print to use with iscntrl() and isprint()
Only print the node's symbol

pq.c:

Create the PriorityQueue struct
Initialize a capacity variable for the max size
Initialize a size variable for the amount of nodes inside the Priority Queue
Initialize an array of Nodes, don't malloc() yet

PriorityQueue *pq_create(uint32_t capacity)
Use malloc() to allocate the memory for the priority queue itself
Make sure that the priority queue allocated correctly (not NULL)
Use calloc() to allocate the memory for the array of Nodes inside the PriorityQueue struct, using the capacity passed in for the size of the calloc
Make sure that the array of Nodes has allocated correctly
Return the priority queue

void pq_delete(PriorityQueue **q)
Check that the priority queue q actually exists (not NULL)

Free the q
Set the pointer to NULL

bool pq_empty(PriorityQueue *q)
Make sure that the q isn't NULL
Return true if the size variable inside the priority queue struct is 0, else false

bool pq_full(PriorityQueue *q)
Make sure that the q isn't NULL
Return true if the size variable inside the priority queue struct equals the capacity of the queue, else false

uint32_t pq_size(PriorityQueue *q)
Make sure that the q isn't NULL
Return the size variable inside the priority queue struct

bool equeue(PriorityQueue *q, Node *n)
Make sure that the q isn't NULL and the node n isn't NULL(exists)
Check if the pq_full is true (return false)
Put the new node at the size variable of the struct index
Call up_heap() (from heap_help.c file) to fix the node that was just placed at the end of the array
Increment size by 1
Return true once finished

bool dequeue(PriorityQueue *q, Node **n)
Make sure that the q isn't NULL (exists)
Check if the pq_empty is true (return false)
Use the swap() function to switch the element at the top and the last element (index of size-1)
Pop the last element, which was the original root node
Decrease the size by 1
Call down_heap() from the heap_help.c file to re-order the heap and fix the order to make it a valid min-heap again
Return true once finished

void pq_print(PriorityQueue *q)

Check that the priority queue isn't NULL
Print the list of nodes

code.c:

Define the block size (4096)
Define the alphabet size (256)
Define the 32-bit magic number
Define the MAX_CODE_SIZE (alphabet/8)
Define the MAX_TREE_SIZE (3*alphabet-1)

Create the Code struct
Initialize the top variable (uint32_t)
Initialize the bits array of type uint8_t, set size to MAX_CODE_SIZE (maximum number of bytes needed to store any valid code)

Code code_init(void)
Create the Code on the stack
Set the top to 0
Zero out any bits inside the array of bits
Use . instead of -> for the 2 lines above
Return the Code

Uint32_t code_size(Code *c)
Return the top variable inside the Code struct

bool code_empty(Code *c)
Check if the code_size(c) is 0
If so, return true
Else, return false

bool code_full(Code *c)
Check if the code_size(c) is 256 (alphabet)
Return true
Else, return false

bool code_set_bit(Code *c, uint32_t i)
Check that i is less than code_size(c) to make sure that the index is in the range

If it's greater than code_size(c), return false
Find the actual index by moduloing by 8 and dividing by 8
Else, change the index i of the bits stack to 1 by using the bit shifting | operator and return true

bool code_clr_bit(Code *c, uint32_t i)
Check that i is less than code_size(c) to make sure that the index is in the range
If it's greater than code_size(c), return false
Find the actual index by moduloing by 8 and dividing by 8
Else, change the index i of the bits stack to 0 by using the bit shifting & operator and return true

bool code_get_bit(Code *c, uint32_t i)
Use bit shifting & operator on the stacks index i to check if the bit is 1 or 0
Check that i is less than code_size(c) to make sure that the index is in the range or the value isn't 0
If it's greater than code_size(c) or 0, return false
Else, Return true

bool code_push_bit(Code *c, uint8_t bit)
Check if code_full is true, if so return false
Set the bit using code_set_bit at index of top and return true if the bit passed in is 1
Clear the bit using code_clr_bit at index of top and return true if the bit passed in is 0
Increment the top variable in the Code struct by 1 after pushing
Return true

bool code_pop_bit(Code *c, uint8_t *bit)
Check if the code_empty is true, if so return false
Call code_get_bit at the index of top and set the bit pointer passed in to it's value (1/0)
Call code_clr_bit at the index of top-1
Subtract the top variable in the Code struct by 1 after popping
Return true

void code_print(Code *c)
Loop through the size of the code (code_size)
Call code_get_bit and print the result

stack.c:

Create the Stack struct

initialize the top variable
initialize the capacity variable
initialize the list of Nodes items

Stack *stack_create(uint32_t capacity)

Use malloc() to allocate the memory required for the Stack object
Check that the memory has allocated properly
Use calloc() to create the list of Nodes, passing in the capacity
Check that the memory has been allocated properly
Set the top to 1
Set the capacity variable inside the struct to capacity

void stack_delete(Stack **s)

Check that the Stack s isn't NULL
Use free() on the list of Nodes
Set the list of Nodes pointer to NULL
Use free() on the Stack object s
Set the Stack object s to NULL

bool stack_empty(Stack *s)

Check that the stack s isn't NULL
If top is 0, return true
Else, return false

bool stack_full(Stack *s)

Check that the stack isn't NULL
If top equals the capacity of the stack, return True
Else, return false

uint32_t stack_size(Stack *s)

Returns the top variable

bool stack_push(Stack *s, Node *n)

Check that the stack isn't NULL, the node isn't NULL, and that the stack isn't full (stack_full returning true)

If so, return false
Else, push the Node n at index top and return true
Increment 1 to the top

bool stack_pop(Stack *s, Node **n)
Check that the stack isn't NULL and that the stack isn't empty (stack_empty returning true)
If so, return false
Set the node pointer passed in to the node at index top-1
Else, set the node at index top-1 to NULL
Return True and subtract 1 from the top variable

void stack_print(Stack *s)
Check that the stack isn't NULL or empty
Loop through the size of the stack
Call node_print on each of the indices

io.c:

define a static position variable
make a static array for the block
define a static variable for the number of bytes in the static buffer array (both static arrays)
define another static array for the write_code block

int read_bytes(int infile, uint8_t *buf, int nbytes)
Check that the extern variable bytes_read is greater than or equal to 4096 (block size)
If so, return 0
Make a counter/total variable
Loop while the total variable is less than nbytes
Update the total variable by the return value of the read(infile, *buf, nbytes-total variable)
Check if the read() function returns 0, if so, the amount of bytes desired to be read is too large for the amount left in infile
If so, read each byte 1 by 1
Increment the extern variable bytes_read by the total variable
Return the total variable

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

Check that the extern variable bytes_written is greater than 4096 (block size)
If so, return 0
Make a counter/total variable
Loop while the total variable is less than nbytes
Update the total variable by the return value of the write(infile, *buf, nbytes-total variable)
Check if the write() function returns 0, if so, the amount of bytes desired to be written is too large for the amount left in infile
If so, write each byte 1 by 1
Increment the extern variable bytes_written by the total variable
Return the total variable

bool read_bit(int infile, uint8_t *bit)
Check if the current position % the block (4096 bytes)*8 (to make it bits) if it is divisible (equals 0)
Clear the buffer array of bytes
Call read_byte to fill the buffer array of bytes
Check if the current position equals the bytes_read * 8 (to make it bits) extern variable
If so, return false to indicate every bit has been read
Calculate the current bits position
Get the bit of the current position value and set the bit variable passed in to the value
Increment the current position extern variable
If the loop ends and there are still bits to be read, return true

void write_code(int outfile, Code *c)
Loop while the length of the Code c isn't empty, or while code_pop_bit is still true, pulling each bit from the top of the stack until its empty
Check if the static buffer array of bits is full
If it is, call write_bytes() to send the full buffer array of bits directly to the outfile and clear the static buffer array
If not, continually call code_pop_bit() and fill the buffer array with the bit passed through the parameter
Increment the current position extern variable
Once there are no more bits to be read, call flush_codes() to clear the semi-full buffer array

void flush_codes(int outfile)

Call write_bytes() and pass in the current position of the amount of bytes inside the semi-full buffer array as the nbytes parameter

huffman.c:

define a static position variable for the amount of bits in the build_codes array
define a static array to hold the bit batten for build_codes

Node *build_tree(uint64_t hist[static ALPHABET])

Create the priority queue using alphabet as the capacity

Loop through the range of the size of ALPHABET (256)

Check if the value at each index is non-zero (meaning there is at least 1 character is the original histogram)

Make a node with the ascii representation of the index in the histogram as the symbol and the value at the index as the frequency

If it is, enqueue the node into the priority queue

If it's not, just continue

Loop while the priority queue still has 1 node inside it (length of the pq)

Set a left variable to the first pq_dequeue

Set a right variable to the second pq_dequeue

Use node_join to join the two left and right children (2 least frequent) and create a parent node between them

Enqueue the parent node inside the priority queue to join it with later nodes

Set a root variable to the last variable in the priority queue because it has the highest frequency by using pq_dequeue

Return the root

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

Create a Code variable as a counter for the binary pattern of walking the tree

Fill the Code struct with the static array defined above, looping through the position/size static

Check that the Node isn't NULL

Check if the current node has any children

If the node doesn't have any children, then set the output table at the index of the node symbol (ascii value) to the pattern of the binary

If the node has children

First walk to the left and push bit 0 to the temporary Code variable to count the pattern

Call build_tree() again on the left child of the node and the current pattern of the table, meaning you'll start at the left child of the original node instead of the root

Keep recursively calling until the node doesn't have a left child, removing the 0 if it doesn't

Then walk to the right of the current node being looked at as much as you can (or if you can walk left again), pushing 1 to the temporary code variable to continue counting the pattern

Call build_tree() again to push to the right again, adding the current binary code in the recursive call

Remove the 1 if the node doesn't have a right child

void dump_tree(int outfile, Node *root)

Check if the current node is actually present (not NULL)

Recursively call dump_tree, passing in the same outfile and the left child of the current node to go left as much as possible

Recursively call dump_tree, passing in the same outfile and the right child of the current node

Check if the current node doesn't have children (leaf)

If so, Make an empty buffer array of size 2

Then use write_byte(), passing in the buffer array of "L" and the node's symbol, with 2 bytes as nbytes

If the current node isn't a leaf

Use write_byte(), passing in an "I" character an 1 byte to the outfile

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

Create a stack for each of the symbols (stack_create)

Create a for loop to iterate through each of the symbols in the tree_dump array

Check if the character is an L

If it is then make a node for the symbol itself and push it to the node stack

Increment the for loop length counter by 1 to jump over the symbol after the L

Check If the character is an I

If it is, use stack_pop() for the right child first

Then use stack_pop() for the left child

Finally create a parent node between the two children using node_join for the interior node (setting the symbol to a “$” and the frequency to the sum of the two children)

Return the root node of the reconstructed tree

void delete_tree(Node **root)

Check if the root node has children by checking if its NULL

Recursively call delete_tree(), passing in the left node if there is one

Recursively call delete_tree(), passing in the right node if there is one

Delete the node being looked at

Header struct used in encode.c and decode.c:

```c
typedef struct {
    uint32_t magic;       // 32-bit magic number.
    uint16_t permissions; // Input file permissions.
    uint16_t tree_size;   // Emitted tree size in bytes.
    uint64_t file_size;   // Input file size.
} Header;
```

encode.c:

include libraries/files

define options

initialize the global variables from the header file with values
initialize the default values (input and output files)
Create temp file to take the stdin input

parse through each command line character

initialize opt and other variables

loop, checking which option was used and if it's in the list of options

switch

case for h:

Print the help message and return a non-zero code

Case for i:

Set the input file to encode using Huffman coding.

The default input should be set as stdin.

Case for o:

Set the output file to encode using Huffman coding. The default input should be set as stdin.

Case for v:

Set the stat boolean to true for later printing

Create a histogram the size of the alphabet with calloc()

Loop while read_byte() doesn't return zero, passing in 1 for the nbytes

Look at the *buf array from read_byte() and insert the byte into a histogram (or increase the frequency) using the index of the value of buf[0]

Clear the buffer array holding the ascii value of the singular character

If the 0th and 1st index are 0, set both indices to a value of 1

Call build_tree() to build the initial tree, passing in the histogram that was just created

Create a Code type variable for the code table

Call build_codes(), passing in the root node returned from build_tree() and the empty code table

Set the header struct values

Set the magic number to the MAGIC macro (using read_byte() or read_bit(), reading 32 bits (size of the magic number))

Use fchmod() to use the file's return integer to set the permissions of outfile to match the per-missions of infile (fstat() to get infile's) (16 bits from the read_bit() function or 2 bytes from read_byte())

Count how many non-zero indices there are in the histogram to find the amount of unique symbols in the huffman tree

Set the tree_size in the header object to (3*unique symbols)-1

Use the fstats() function to find the size of the infile, setting file_size to this value

Use write_byte() to write the header to the specified outfile

Write the constructed Huffman tree to outfile using dump_tree()

Call write_code() for each code representing the binary pattern for each symbol, printing to the outfile, then call flush_codes() to clear any bits remaining inside the semi-full buffer array

Close the in and out files.

Check if the printing boolean is true

If so, print out the uncompressed file size, the compressed file size, and space saving (100 ×(1 −(compressed size/uncompressed size))).

decode.c:

include libraries/files

define options

initialize the global variables from the header file with values
initialize the default values (input and output files)

parse through each command line character
    initialize opt and other variables
    loop, checking which option was used and if it's in the list of options
        switch
            case for h:
                Print the help message and return a non-zero code
            Case for i:
                Set the input file to decode using Huffman coding. The default input should be set as stdin.
                Case for o:
                    Set the output file to decode using Huffman coding. The default input should be set as stdin.
                Case for v:
                    Set the stat boolean to true for later printing
    Use read_byte() to read in the header from the infile
    Check if the magic number read in is the same as the one defined in defines.h (0xBEEFBBAD)
        If it is, continue
        If not, then print out the help message and return a non-zero number

    Continue to read the data from the header file, use fchmod() to set the permissions for the decode.c's outfile
    Create an array from the size of the tree_size header variable, filling it with the dumped tree itself
    Then call rebuild_tree() and pass in the array of the tree dump and the size of the array (nbytes) into the function
    Call read_bit() on the infile to traverse down the tree one link at a time for each bit that is read
        Print out the symbol at each post-order traversal node of the tree links
    Check if the printing boolean is true

If so, print out the uncompressed file size, the compressed file size, and space saving (100 ×(1 −(compressed size/decompressed size))).

Close the infile and outfile and delete the root node

Makefile

CC = clang
CFLAGS = -Wall -Wextra -Werror -Wpedantic -gdwarf-4 (for debugging)
make all: build all of the executables
make encode: build just the encoder, specifying the required .o files in the definition.
make decode: build just the decoder, specifying the required .o files in the definition.
make clean: removes all files that are compiler generated (.o files).
make spotless: removes all files that are compiler generated (.o files) as well as the executable (decode or encode).
make format: formats all of the source code, including the header files.