Name: Jacob Gurien
cruzID: jgurien

General Purpose:

The purpose of this program is to implement a huffman coding program that can encode and decode a string of characters using a huffman tree structure.

node.c

struct Node

This structure will provide the central architecture for the node structure, which will contain a left node, a right node, a symbol represented in bit format, and a uint64 representing the symbol's frequency.

node_create

This function will create a node structure by calling malloc on the node. Then, it will check if the node exists, and if it does, it will initialize its left node and right node to null, and set its frequency to the passed-through symbol and its symbol to its passed through symbol. Then, it will return the node.

node_delete

This function will free the pointer to the node structure and set its value to null, if the pointer to the node exists.

node_join

This function will create a parent node. First, it will call node_create on the node, and then check if it isn't null. Also, inside of the node_create, it will have the parameters of the symbol ('$') and the left frequency + the right frequency. Node_create will also only be called only if the node's right and left nodes are not null. Finally, if all of the cases are valid, then it will return its node.

node_cmp

This function will return true if the left node's frequency is greater than the right node's frequency.

node_print_sym:

This function will only print the symbol of the node. I will also print control characters or non-printable symbols.

node_print

This function will print the node. To do this, it will check if the node exists, and if it does, then it will print that it exists. Then, it will check if its left and right nodes exist, and if they do, it will print that they exist. If they don't exist, I will print that they weren't successfully joined. I will also print the frequency and symbol to debug, as well as control or non-printable symbols.

node_helper.c

l_child

This function will return the left child of the function by calling 2 times the number passed in plus 1.

r_child

This function will return the right child of the function by calling 2 times the number passed in plus 2.

parent

This function will return the parent by returning the number passed in minus 1 and then dividing it by 2.

swap

This function will take two nodes to swap. First, I will set a temp node to the first node, and then I will set the first node to the second node. Then, I will set the second node to the first node.

up_heap

This function will order the heap array once, given a specific value n. To start, I will first check if the passed in index (n) is greater than zero and if the array pointed to frequency (arr[n->frequency]) is less than the array of index parent of index pointed to frequency (arr[parent(n->frequency)]). If this is true, then I will swap the two arrays, and update the index value to the parent of the index.

down_heap

The purpose of this function will rebuild (fix) the heap. This function will start by initializing two variables: a uint32 representing the smaller variable and a variable (n) representing the number to insert inside of the left/right/parent child functions. Then, I will initialize a for-loop that will check if the left child of n is less than the heap size. If this is true, then I will first check if the right child equals the heap size, and if it does, then I will set the smaller variable to the left child. Then, I will check if the passed-through array of the left child is less than the passed-through array of the right child. If this is true, then the smaller variable is set to the left child. If not, then the smaller variable is set to the right child variable. Then, I will check if the array of n (arr[n]) is less than the array of smaller arr[smaller], and if it is, then I will break. Finally, at the end, I will set n to the smaller number which will conclude this function.

pq.c

struct PriorityQueue

This structure will initialize a node array (Node **) and a uint32 capacity variable to represent the capacity of the priority queue.

pq_create

To start, I will initialize a priority queue using malloc. Then, I will set the queue's capacity to the capacity set through. Then, I will initialize a node array by calling calloc on it. Then, if it is allocated right, I will set the capacity to the field passed in and the size to 0.

pq_delete

In this function, I will check if the priority queue exists. If it does, then I will free the priority queue pointer, its list, and set the priority queue's pointer value to null.

pq_empty

This function will return true if the priority queue's size is equal to 0, and false if it is greater than 0.

pq_full

This function will return true if the size is equal to the capacity, and false otherwise.

pq_size

This function will return the size of the priority queue using its passed-in size.

enqueue

This function will insert an element into a priority queue. First, I will check if the priority queue is full, and if it is, then I will return false. If it isn't full, I will put the node into the array, with the size as its index. Then, I will call up_heap to sort the heap into a valid heap, and if this finishes correctly, then I will return true, showing that the node was properly inserted into the heap.

dequeue

This function will pop the root element from the heap structure, and reorganize and fix the heap. To start, I will first check if the priority queue is empty, and if it is, I will return false. Then, I will swap the first node element and the last node element with each other, in order to allow for the heap to become valid after one heap fix call. Once the values have been swapped, I will pop the last element of the node array, and subtract the size by 1. Then, I will call my down_heap again to rebuild and fix the heap. Once all of this happens, I will then return true.

pq_print

To start this function, I will first loop through the priority queue's size, and print each index of the array.

code.c

struct code

This function will initialize two variables: the first being a uint32 number representing the top, and the other representing a uint8 bit array that can hold MAX_CODE_SIZE (256/8).

code_init

In this function, I will set a code object. Then, I will loop through the code pointed uint8 array and set each value to 0 (c.arr[i] = 0). Then, I will set the code dot the top uint32 number to 0 (c.top = 0). Then, I will return the code object.

code_size

This function returns the value of top, representing the number of pushed bits.

code_empty

This function will return true if the code structure pointed top is 0. Else, it will return false.

code_full

This function will check if the code size is equal to 256, and if it is, return true. Else, return false.

code_set_bit

First, I will check if the bit that wants to be set is in range, and if it is, continue. Else, I will return false. Then, I will first grab the bit's location and position by dividing the bit index by 8 and modulo by 8. Then, I will do bitshift OR on the

stack at location index and 1 left shift by the position index. If this passes, it will return true.

code_clr_bit

First, I will check if the bit that wants to be cleared is in range, and if it is, continue, else, it will return false. Then, I will grab the bit's location and position by dividing the bit index by 8 and modulo by 8. Then, I will do bitshift AND to clear the bit's value by using the location in the array and its position. If this passes, it will return true.

code_get_bit

First, I will check if the bit that wants to be cleared is in range, and if it is, continue, else, it will return false. Then, I will grab the bit's location and position by dividing the bit index by 8 and modulo by 8. Then, I will do bitwise AND on the array's location right shifted by the position and 1. Then, it will return the bit's value.

code_push_bit

First, I will check if code_full is true, and if it is, I will return false. If it isn't, I will check if the passed-through bit is 1. If it is, I will call code_set_bit onto the top bit, and if it isn't, I will call code_clr_bit on the top bit. Then, I will increment the top variable, and return true.

code_pop_bit

First, I will check if code_empty is true, and if it is, I will return false. If it isn't, I will call code_get_bit on the top bit - 1, and I will clear the top bit using code_clr_bit. Then, I will decrease the top variable value by 1, and return true.

code_print

In this function, I will loop through code_size and print the output of code_get_bit for each bit in the range.

io.c

read_bytes:

To start, I will initialize two static variables: one representing the number of bytes being read with the read() call, and another one being incremented by this value, representing the total bytes. Then, I can start my program. First, I will check if the number of bytes is less than or equal to 0. If it is, then I will return -1. However, if this isn't true, then I will read in more bytes using the read() call, and increment this value to another variable to track the total number of bytes read. Then, I will check if this value is 0, and if it is, I will return -1. If this isn't true, then I will recursively call read_bytes with the nbytes field being subtracted each time by the number of bytes read in that function call. Finally, once all of these steps finish, I will return the total number of bytes read, as well as setting the global bytes_read variable to this number.

write_bytes:

To start, I will initialize two static variables: one representing the number of bytes being read with the write() call, and another one being incremented by this value, representing the total bytes. Then, I can start my program. First, I will check if the number of bytes is less than or equal to 0. If it is, then I will return -1. However, if this isn't true, then I will write in more bytes using the write() call, and increment this value to another variable to track the total number of bytes written. Then, I will check if this value is 0, and if it is, I will return -1. If this isn't true, then I will recursively call write_bytes with the nbytes field being subtracted each time by the number of bytes read in that function call. Finally, once all of these steps finish, I will return the total number of bytes written, as well as setting the global bytes_written variable to this number.

read_bit

This function will return a bit that is being read, in a block of 4096 bytes. To do this, I first have to make sure that the block is fully created, meaning that its value is a full block (4096 or size of the line), and nothing more. To do this check, I will also have to initialize a local variable that represents the position, or index, of the bit that I am setting. Then, I will use bitshift AND to make sure that the position is in a valid position (or block). If it is, then the block is full, and a valid

block can be read. Then, I will loop through the block, and set each index of the buffer array (another external variable) to zero, as to dole out all values inside. Then, I will call read_bytes on this buffer. Then, I will initialize another check, which will see if the bytes that have been read are equal to 0. If this is true, then I will return false inside the function. If this doesn't happen, then I can get the current bit index's position and location in the array, by performing two calculations to get this, which are similar to what I did in my bit functions in code.c. Then, I will perform bitshifting to get the current bit value of these indices, and then, I will set the bit pointer's value to this bit found. Finally, I will increment the bit's index by 1, and return true.

write_code

This function will write code using a code structure. To do this, I will first initialize a while-loop where I check if the pop bit is true (there are still bits in the stack). Then, I will get the bit's current position and location by using two modulo and division operators. Then, I will check if the bit value is 0, and if it is, I will set it to 1. If it is one, then I will clear it. These will both be using bit shifting. Then, I will increment the current index value by 1, since a bit will have either been set or cleared prior to this. Then, I will check if the block size is a valid block, and if it is, I will call write_bytes. Then after doing this, I will loop through the buffer array that was populated with write_bytes, and set each index value to 0, and exit the function.

flush_codes

This function will flush out (increment) the remaining bytes. To do this, I will call write_bytes, with the parameters being the outfile, the write code buffer array, and the block size. However, if the buffer's position is odd, I will write an extra byte to account for the lost byte that happens in floor division.

stack.c

struct Stack

This structure will take two uint32 variables, a top variable and a capacity variable. I will also initialize a node list.

stack_create

This function will initialize a stack. First, I will call malloc on the stack structure to make my stack. Then, I will make sure that the stack exists, and if it does, I will continue. Then, I will set the stack pointed capacity to the capacity value passed through. I will also initialize the stack pointed to the top value to 0. Then, I will call calloc on the list of nodes, with the size of them being the capacity. Then, I will return the stack structure.

stack_delete

First, I will check if the stack pointer isn't null, and if it exists, then I will free the list of nodes, as well as the stack pointer itself. Then, I will set the stack pointer to null.

stack_empty

This function will return true if the stack pointed to the top equals 0, else, false.

stack_full

This function will return true if the stack pointed to the top equals the stack pointed to the capacity, else, false.

stack_size

This function will return the stack pointed to the top.

stack_push

First, I will check if stack_full returns true. If it does, then I will return false. If it doesn't return true, I will set the array with index of stack pointed top to the node pointer. Then, I will add the stack pointed top value by 1, and return true.

stack_pop

First, I will check if stack_empty returns true. If it does, then I will return false. If it doesn't return true, I will set the node pointer to the array with index of stack

pointed top. Then, I will set the top of the node array to null, and subtract the stack pointed top value by 1, and return true. Also, all of these functions will only execute if the stack pointer exists.

stack_print

I will loop in the range of 0 to the top of the stack and call node_print on each node index.

huffman.c

build_tree

To first do this, I will construct a priority queue inside of the function, with its capacity being the alphabet macro. Then, I will loop through the ascii values contained inside of the inherited alphabet table. Then, I will make sure that each value stored at the index is a non-zero one, and if this is true, then I will make a node and enqueue it. Then, I will initialize a while-loop that will check if the size of the priority queue is greater than 1. If it is, then I will create two nodes, a left and right node - each equal to the dequeued value of the priority queue. Then, I will create a parent node that will join the left and right nodes, and finally, I will call enqueue again with the parent node being passed through. Once this while-loop completes, I will set the root node equal to the final dequeued value in the queue, and return it.

build_codes

This function will build the code table. First, I will initialize a code object at the top. Then, I will check if the node that is passed through is valid (not null), and if it is, I will then check if the node's left and right value are valid. If they are, then I will set the table's node symbol to the initialized code table. However, if the node passed through was invalid (null), then I will push a bit on the 0th index of the code object, and recurse back through build, but passing through the left node as the node parameter. This is to recurse through the furthest left sides of the tree. Then, after this recursion call, I will pop the top bit off of the code structure. Then, I will do the same thing, but on the right side of the tree. To do this, I will push a bit on the 1st index of the table, with me then sequentially recursing on the right side of the tree (right node). Finally, I will pop the top bit off of the code

structure, and end the code there. Also just to clarify, this recursive function's purpose is to continue traversing the tree until every node has been traveled/reached, where it will then set the table's value to that final code value.

dump_tree

This program will conduct post-order traversal (left, right, node) to dump the tree and record its process (a string of text). To do this, I will first check if the root value passed in is valid. If it is, then I will recurse (travel) the tree on the left and right roots. Then, I will check if the left root and the right root is invalid (meaning that the current root is a leaf), and if it is, then I will write the character 'L,' standing for leaf, and its symbol. If this check isn't true, I will write 'I,' meaning interior node.

rebuild_tree

In this function, I will rebuild the tree and return the root node using post-order traversal. First, I will initialize a stack object. Then, I will initialize a while-loop that will run until all of the dump symbols have been processed. Inside of the loop, I will check if the current symbol is 'L,' (leaf), and if it is, then I will insert this symbol into a node by creating it, and then pushing it onto the stack object. However, if the current symbol is 'I,' (interior node), then I will pop the stack for the right and left children (nodes), and create a parent node with these values. This will continue until there is only one node left on the stack, which will be the root node, hence rebuilding the tree.

delete_tree

This function will delete the tree. To start, I will check if the node array passed through is valid, and if it is, I will check if the left and right nodes are valid. If the left is valid, I will call delete_tree recursively on the left node, and vice-versa for the right node. Then, once this completes, I should only have the root node remaining, which I will then manually delete by calling node_delete.

encode.c

To start this function, I will first initialize a temporary array with a size of 0. Its purpose will be to store one byte. Then, I will read in bytes into this array, and input each byte

value into that index of the histogram. Then, once all possible values have been inputted into the histogram (read_bytes returns 0), then the loop will break. Also, inside of the loop, I will reset the index after storing it in the histogram array to 0, so there are not any memory leaks. Then, I will create two symbols for non-zero count variables, which will help us build our tree. Basically, I will implement this by checking if the symbol's count in the histogram is 0, and if it is, I will set it to 1. Next, I will create an empty array that implements a code structure (Code arr[]). Then, I will call build_tree to build the huffman tree. Next, I will construct a code table by calling build_codes, while utilizing the code array that we used above. Then, I will initialize all passed through variables (magic, permissions, tree_size, file_size) to their specified values that we want, to a structure. To do this, I will set magic to the magic number macro that is defined in defines.h. Then, I will retrieve the infile's permissions by using fstat, with me then specifying the outfile's permissions to match those with fchmod. Next, I will retrieve all unique values (non-zero values) by traversing to the histogram array and incrementing a count variable by each unique index. Then, I will obtain the file_size parameter by using fstat to find the size of the infile. Then, I will write this structure to an outfile using write_bytes. Then, I will call dump_tree to write the huffman tree to outfile. Next, I will call write_code to write each symbol in infile to outfile. Finally, I will write out any remaining bytes in the buffer with flush_codes, and close my infile, outfile and exit.

decode.c

To start this function, I will first read in the header structure from infile and then verify its magic number. To do this, I will compare the inherited magic number with the defined one in defines.h, and if they match, then I will continue. If they don't, I will print a helpful error message. Next, I will look at the permission message value inherited from the header structure, and store those values. Then, I will use those values with fchmod to set the proper permissions. Then, I will create an array that is tree_size long (a value inherited from the header structure), and repopulate the tree by calling rebuild_tree to obtain the root node. Next, I will run through the infile one bit at a time by calling read_bit. If the bit's value is 0, I will traverse through the left side of the tree, and if the value is 1, I will traverse through the right side. If the symbol that I find reads 'L," or leaf, then I will write the node's symbol to outfile. After this, I will reset its position to root. I will repeat this process until the amount of decoded symbols match the file_size parameter, and once that is done, I will close all files and quit.

Makefile

This file will be my file used to compile my program. To start, I will create 5 phony targets: encode, decode, all, clean, and spotless. To make encode, I will only generate .o files for all derived files used to encode. I will do the same thing for decode as well. To make all, I will derive all files used. Make spotless will remove all .o files and derived files as well, whereas make clean will remove only .o files. Also, I will add make format, which will format my code, and make scan-build, which will search my code for more strict and niche errors to debug. Once running this file, my program will compile.