Nico Vitagliano

Cruz ID: nvitagli

General Purpose -

This lab's purpose is to code a working lossless compressor using huffman encoding and decoding. It will break down bytes into working tables, trees, nodes, and codes and then be able interact with both the main files. The user can pass in files to be compressed and decompressed at will.

Makefile -

Inside the makefile I will start by specifying my compile format as clang. Then I will specify the flags used: -Wall -Wpedantic -Werror -Wextra. Under that will be "all" with the target of the main files. Under that will be the main files and the other 'o' files to compile all of the files in the directory. With that I will specify all of the o files for the compile and then make all of the flags work with the c files. Then after all of that I will make clean and specify all o files. Spotless will be under that with all of the o files and the executables. Then I will do format with the c files and scan build with clean and clang.

Node -

Struct:

Inside this function I will declare the vars and pointers for the left child, the right child, the symbol, and the frequency.

Create:

I will start by creating the function using Node *node_create(uint8_t symbol, uint64_t frequency). Then inside I will allocate memory and check to make sure it is allocated properly. Then I will set the nodes symbol and frequency and the vars value that was passed through into the function. Afterwards, I will set the pointers to null and return the node.

Delete:

I will start by creating the function using void node_delete(Node **n). Then inside I will check if the node is null or not. If not, I will free the pointer and set it to null.

Join:

I will start by creating the function using Node *node_join(Node *left, Node *right). Then inside I will create a new node. I will check if the node was created properly and if so I will set the left child of the new node to the left one passed in and the same for the right side. Then for the symbol I will set it to the “$” and the frequency to the sum of the two frequencies of the nodes passed in. I will then return this node.

Print:

I will start by creating the function using void node_print(Node *n). Then inside I will print the values of all of the object’s variables. For the printing of the symbol I will use iscntrl() and isprint() to identify which format to use. If the symbol is a control character then I will print using 0x%02"PRIx8 and if it isn’t I will just print the character.

Compare:

I will start by creating a function using bool node_cmp(node 1 node 2). Then inside I will return node1 > node2.

Print system:

I will start by creating a function using void node_print_sys(node). Then inside I will print the symbol of the node. For the printing of the symbol I will use iscntrl() and isprint() to identify which format to use. If the symbol is a control character then I will print using 0x%02"PRIx8 and if it isn't I will just print the character.

Heap:

Swap: (two pointers)

Inside I will make a temp equal to the first pointer and then make the first equal to the second and the second equal to the temp.

Left: (index passed in)

Inside I will find the left node using the index times 2 plus one and return that.

Right: (index passed in)

Inside I will find the right node using 2 times the index plus 2 and return that.

Parent: (index passed in)

Inside I will return (index - 2) divided by 2.

Up heap: (index and array passed in)

Inside I will loop while the index is greater than 0 and the array index frequency value is less than the parent. Inside I will use my swap function on the two nodes and make the index equal to the parent.

Down Heap: (array and size) (the children are the index return of the children calls)

Inside I will create temp vars to hold the index and the larger value. Then I will loop while the left child is less than the size. Inside the loop I will check if the right child equals the size. If so I will make the larger temp equal to the left child. Otherwise I will

check if the right child index of the array is bigger than the left index of the array and if so I will make the larger var equal to the left. Otherwise, the right child will be larger. Outside of these checks I will see if the current index is smaller than the larger temp and if so I will break. Then swap the larger and the current index. Also, make the index the larger var.

Priority Queues -

Struct:

Inside I will create a var to hold the size of the array, a var that is the array, and the var that holds the capacity of the object.

Create:

I will start by creating the function using PriorityQueue *pq_create(uint32_t capacity). Then inside I will allocate the memory for the PQ using malloc and check to make sure it was done correctly. Then I will allocate memory for the array using calloc and check that it was done correctly. Afterwards, I will set the object var of capacity to the passed in value and then set the size to zero. Then I will return the PQ.

Delete:

I will start by creating the function using void pq_delete(PriorityQueue **q). Then inside I will check if the pointer is null if not then I will check if the pointer of the array is null. If they are not I will free both and set them both to null.

Empty:

I will start by creating the function using bool pq_empty(PriorityQueue *q). Then inside I will check if the pointer is null and if not I will return true if the size is zero and false if not.

Full:

I will start by creating the function using bool pq_full(PriorityQueue *q). Then inside I will check if the pointer is null and if not I will check if the size equals capacity. If it does not then I will return false otherwise I will return true.

Size:

I will start by creating the function using uint32_t pq_size(PriorityQueue *q). Then inside I will check that the pointer is not null. If it is good then I will return the size of the object.

Enqueue:

I will start by creating the function using bool enqueue(PriorityQueue *q, Node *n). Then inside I will check to make sure the q pointer is not null. Then I will check if it is at maximum capacity using size and capacity. If it is, I will return false. Otherwise I will add the node to the array of the PQ using the size. Afterwards, I will upheap to fix the order, update the size, and return true.

Dequeue:

I will start by creating the function using bool dequeue(PriorityQueue *q, Node **n). Then inside I will check if the q pointer is null. I will also check if the queue is not empty. Then I will make index 0 the size-1 index of the array and then pop the size-1 index and set it to the node passed in. Then I will update the size of the array and send the new array into the down heap. Finally I will return true. If any test fails I will return false.

Print:

I will start by creating the function using void pq_print(PriorityQueue *q). Then inside I will make sure the q is not null. Then I will loop through the array and send each node to node print.

Code -

Struct:

Inside I will create the vars for the code object. These vars will be top and bits.

Init:

I will start by creating the function using Code code_init(void). Then inside I will make an object and loop through the array setting each bit to zero. I will also set the top object var to 0. This whole thing will then get returned.

Size:

I will start by creating the function using uint32_t code_size(Code *c). Then inside I will return the value of top.

Empty:

I will start by creating the function using bool code_empty(Code *c). Then inside I will return true if the top var is 0 and false if not.

Full:

I will start by creating the function using bool code_full(Code *c). Then inside I will return true if top equals 256 and false otherwise.

Set:

I will start by creating the function using bool code_set_bit(Code *c, uint32_t i). Then inside I will check if the index is out of range and if so I will return false. Otherwise I will find the location using i divided by 8 and the position using i modulo 8. Then I will “bit shift or” the bit using the location and position. Finally return true.

Clear:

I will start by creating the function using bool code_clr_bit(Code *c, uint32_t i). Then inside I will check if the value passed is out of range and if so return false. Otherwise I will find the location using i divided by 8 and the position using i modulo 8. Then I will “bit shift and” the bit using the location and position. Finally return true.

Get:

I will start by creating the function using bool code_get_bit(Code *c, uint32_t i). Then inside I will check if the value is out of range and if so I will return false. Otherwise I will find the location using i divided by 8 and the position using i modulo 8. Then I will make a temporary copy of the bit using the location. Then I will “bit shift or” the temp bit. Afterwards, I will see if the actual bit equals the shifted bit. If it is equal then I will return true, otherwise false.

Push:

I will start by creating the function using bool code_push_bit(Code *c, uint8_t bit). Then inside I will check if the code is full using a call to code full and if so return false. Then I will check the bit value using get bit. Afterwards, if the bit is 1 I will call set bit using the index of top otherwise I will call clear bit using top. Finally return true.

Pop:

I will start by creating the function using bool code_pop_bit(Code *c, uint8_t *bit). Then inside I will make sure the code isn't empty using a call to empty. If it is, return false. Then I will get the bit using get bit of top-1 and set the pointer to it. Then I will call clear bit of top-1 and subtract one from top. Finally return true.

Print:

I will start by creating the function using void code_print(Code *c). Then inside I will loop through the array and get each bit. Print 1 if it returns true and 0 if false.

Stack -

Struct:

Inside I will create the object vars: top, capacity, and items.

Create:

I will first create the function using Stack *stack_create(uint32_t capacity). Then inside I will allocate memory using malloc for the object. I will then check to make sure it worked and if so I will use calloc and capacity to allocate memory for the items. I will check to make sure the memory was allocated and if so I will make top equal to 0 and capacity set to the passed in var. Finally return the object.

Delete:

I will start by creating the function using void stack_delete(Stack **s). Then inside I will check that the stack pointer is not null. If so I will free the object and its item and set them to null.

Empty:

I will start by creating the function using bool stack_empty(Stack *s). Then inside I will check if the stack is null and if not I will return true if the top is 0 and false otherwise.

Full:

I will start by creating the function using bool stack_full(Stack *s). Then inside I will check if the stack is null and if not I will check if the top is equal to the capacity and if it is I will return true. Otherwise False will be returned.

Size:

I will start by creating the function using uint32_t stack_size(Stack *s). Then inside I will check if the stack is not null and if so I will return top.

Push:

I will start by creating the function using bool stack_push(Stack *s, Node *n). Then inside I will check that the stack is not null if so I will set the node to the top, add one to top, and return true. Otherwise I will return false.

Pop:

I will start by creating the function using bool stack_pop(Stack *s, Node **n). Then inside I will check if the stack is null and return false if so. After I will check if the stack is empty by calling to empty and return false if so. Then set the node pointer to the top-1 index. After I will set the index to null and subtract one from top. Finally return true.

Print:

I will start by creating the function using void stack_print(Stack *s). Then inside I will check if the stack is null and if not I will loop through the range of top. Each time use print node on the index.

I/O -

Read Bytes:

I will start by creating the function using int read_bytes(int infile, uint8_t *buf, int nbytes). Then inside I will create a temp var to hold the new bytes read amount. Then I loop while the total temp bytes read is less than nbytes. Inside the loop I will add the bytes returned from read(file, buf, nbytes-temp) to the bytes read extern var and the temp var. Once the loop breaks, I will return the temp var value. Or, if the bytes stop reading I will return.

Write Bytes:

I will start by creating the function using int write_bytes(int outfile, uint8_t *buf, int nbytes). I will then loop while the temp var is less than the nbytes. Inside I will add the bytes returns from write(file, buf, extern - temp) to the bytes writen extern and the temp var. Once the loop breaks, I will return the temp var value. Or, if the bytes stop writting I will return.

Read bit:

I will start by creating the function using bool read_bit(int infile, uint8_t *bit). Then inside I will check if the index extern is at the same size as the block size and if so I will call for a new block from read bytes and set the index to 0. I will also check the size of the block and if there is nothing left, I will return false and set all the externs I make to 0. Otherwise, I will get the bit from the index (get bit style indexing) and set the bit to the bit pointer. Then I will increase the index of the vector by one and return true.

Write code:

I will start by creating the function using void write_code(int outfile, Code *c). Then inside I will loop until the block size is full. I will also check if the stack is empty using the code field in order to break. Inside the loop, I will pop each end bit using the pop bit function in the code stack and add it to the buffer using or shifts and the index / 8. Once the buffer is full (size is block size) I will send it to the write byte and start again with a cleared buffer and default var values. If you get a new block that is not full then flush the rest.

Flush:

I will start by creating the function using void flush_codes(int outfile). Then inside I will call write bytes using the outfile, the leftover buffer, and the current size of the block. I will make sure that the bytes are an even amount and fix them in bad conditions. Then clear the buffer and reset the variables to default.

Huffman -

Build tree:

I will start by creating the function using Node *build_tree(uint64_t hist[static ALPHABET]). Then inside I will create a q with the using alphabet. Then I will loop through alphabet. Inside the loop, I will look at each value in each index and seeing if the value is non zero. If so I will create a node using node create and the index. Then take that node and enqueue it. Now outside the loop in a new loop, I will loop while the size of the queue is greater than one. Inside I will set a temp left to the dequeued node of the PQ and the right temp to the next dequeued node of PQ. Then I will join the left and right and set the parent temp as the return from that. Next, I will enqueue using the PQ and parent. Now outside of the loop I will set the root temp var to the the dequeued node return of PQ. Return this value.

Build code:

I will start by creating the function using void build_codes(Node *root, Code table[static ALPHABET]). Then inside I will start by making a temp Code. Then I will check if the root passed in is null or not and if it isn't I will continue. Then I will see if the node passed has a right or left child and if they are both empty then I will set the tables index at the ascii value of the symbol as the temp code. Otherwise, I will push a code using code push with the code and 0. Then I will call build again with the left node and the

table. Then I will pop the bit of code and push the code with code and 1. Afterwards, build code again with the right and the table. Finally pop the bit of code.

Dump:

I will start by creating the function using void dump_tree(int outfile, Node *root). Then inside I will check if the root is not null. If so I will recall dump using outfile and left child as well as outfile and right child. Then I will check if the left child and right child are null. Then if they are I will use write byte with outfile, an array containing L, and 1 byte. Then I will do the same thing but with an array containing the symbol. Otherwise if the condition is not met, I will write using write byte with outfile, an array containing I, and 1 byte.

Rebuild:

I will start by creating the function using Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes]). Then inside I will make a stack with the create function. After that I will loop while the index is less than the tree dump array . For each index, I check if the index value and if it is I will go to index plus one and make a node for it. Then you take the node and push it onto the temp stack. The index will increase by two to go to the next symbol if the symbol was an L. If the index value was not L then check if the value is I in this case you will join to top two stack nodes using pop on them and passing them into join. Set that nodes symbol to the $ and the frequency to the sum of the joined nodes. Finally after everything return the root node.

Delete:

I will start by creating the function using void delete_tree(Node **root). Then inside I will check if the node has either children. If so, I will call delete tree again with the node. After in the other case, call node delete and free the root and finish with setting it to null.

Encoder -

First I will define the options H I O V and then use the get opt frame to parse through the options. In the H option, I will print the help message. In the I option, I will set the var to hold the file passed through. Then for O I will set the var to hold the out file name. Finally, for V I will flag to print the statistics. Then under the get opt I will make a buffer and loop while the read byte(infile, buffer,1) function does not return a 0. Inside I will go to the index that the int returned is and add one to the value. Then I will clear the buffer to 0. Then after the loop I will check the index value of 0 in the histogram and if it equals 0 I will send it to one. I will do the same thing for the 1 index value. Then I will call build tree using the histogram. Afterwards, I will make a code array and call build codes using the node returned from build tree and the array. Then I will make a new header object. For the object, I will set the magic var in the object to the defined magic number. Then I will use fstat() to get the permissions of the infile. I will use fchmod() to then set those permissions for the outfile. I will then loop through histogram and count the non zero indexes. I will set the tree size var to 3 times the count minus one. Then I will use fstats() to get the bytes from the infile and set the file size var to that number. Then write to the outfile using the header info and write bytes. After write the tree to the file using dump tree. Then I will loop through the code table from earlier and pass each code through to write code along with the outfile. Flush the remaining code and close the files. Then I check for stats and print if needed. Finally close files and do deletes.

Decoder -

First I will define the options H I O V and then use the get opt frame to parse through the options. In the H option, I will print the help message. In the I option, I will set the var to hold the file passed through. Then for O I will set the var to hold the out file name. Finally, for V I will flag to print the statistics. Then I will read in the header using the same formatting as the other file setup in order to know where I am in the contents of the file. This will be done using the byte amounts of the header fields. I will then compare the magic numbers and in the case that they differ, I will return an error code and quit. Then I will use the permissions var and fchmod to set the permissions for the file currently being worked on. After I will call rebuild tree using the dumped info and the bytes from the var. Then I will loop through the file until it is out of bits. Each time I will call read bit and check which direction to go in the tree. I will traverse the node and if a node ever does not have a child node, I will read the symbol and store it and reset the position to the root. Then I will check for stats and print them if needed. Finally I will close all of the files.

Readme -

In the readme I will follow the same format as always. I will start by describing the purpose of the lab and important files. Then under, I will give the instructions for compiling and running the program. All of the files and their purpose will follow this and under that will be the command line options and defaults.