Derick Pan
Dpan7

ASGN6 Design

Who is Huffman?	David A. Huffman is a graduate student from MIT, and at one point joined the faculty of UCSC until his passing. His works are used throughout the world for his works in data compression.
What is a Huffman tree?	Huffman coding is a type of lossless data compression. The tree follows the frequency of a character's occurrence and builds a tree out of it.

The Encoder

compresses a file
Following is the command-line options for the Huffman encoder:

- -h : Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards.
- -i infile : Specifies the input file to encode using Huffman coding. The default input should be set as stdin .
- -o outfile : Specifies the output file to write the compressed input to. The default output should be set as stdout .
- -v : Prints compression statistics to stderr . These statistics include the uncompressed file size, the compressed file size, and space saving. The formula for calculating space saving is: 100 × (1 − (compressed size/uncompressed size)).

Specifics:
Input file to compress is named as infile
Output file is named outfile

Pseudo:

1. Read infile to construct a 256 (ALPHABET) array.
2. Increment the count of element 0 and element 255 by one in the histogram.
3. Create a Huffman tree using a priority queue.
    a. Create a priority queue, and create a corresponding Node and insert into the priority queue
    b. While 2+ nodes in queue, dequeue it. The two dequeued will be joined using node_join, and enqueue the parent node.
    c. Last node in the queue is the root node.
4. Construct a code table by traversing Huffman tree (using build_codes())
    a. Create a new code Starting at root. Then start post-order traversal
    b. If node is a leaf then it will contribute to the node symbol.
    c. Else it's an interior node. Push 0 and recurse down the left link.
    d. After you return from left pink, pop, push, recurse.
5. Construct the header from header.h
    a. magic is the macro MAGIC. This is used to identify a file as the one being compressed from encoder.

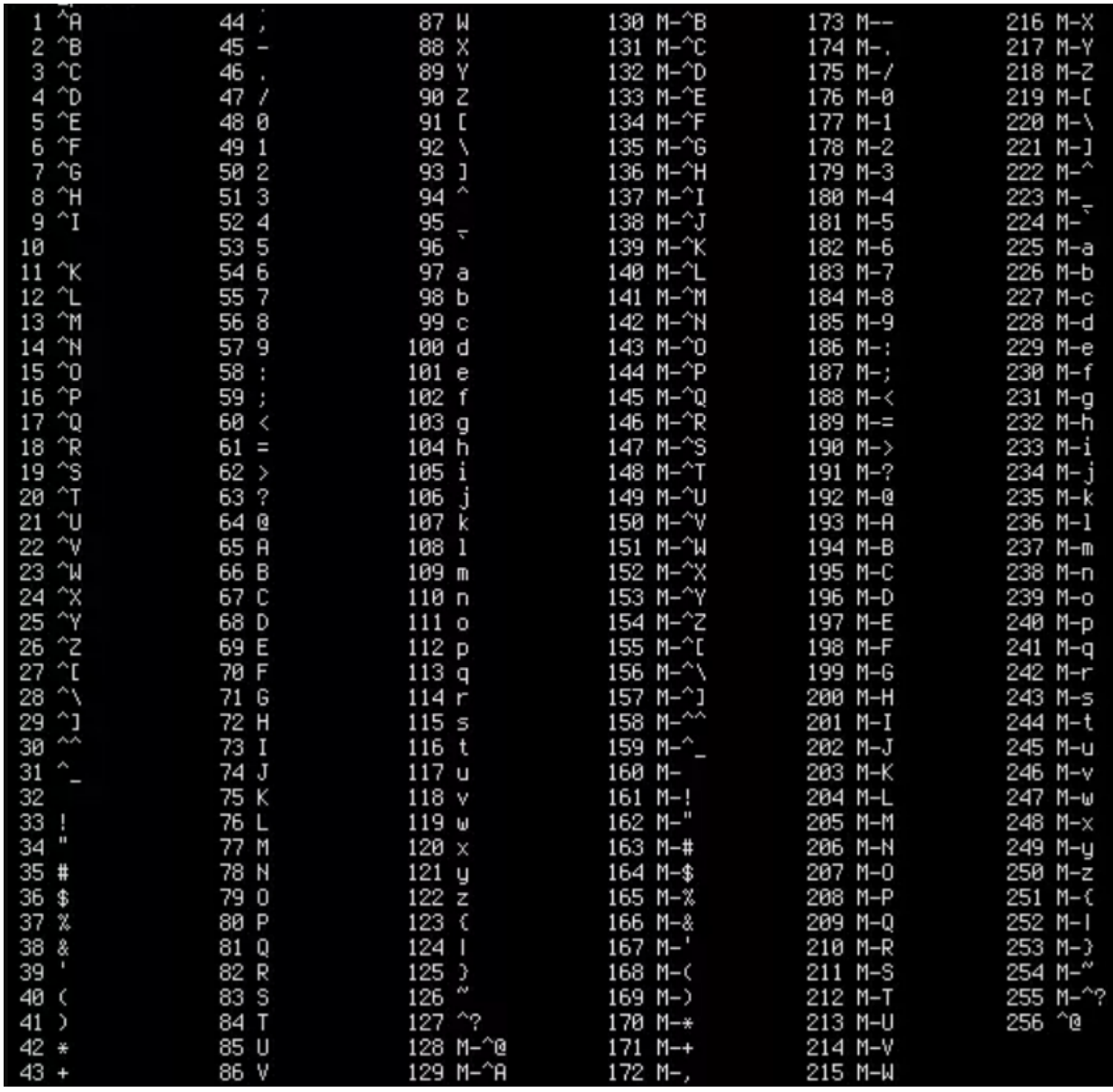
b. permissions stores the perms bits of infile. Set outfile to be the same.
c. Tree_size is the num of bytes that make up huffman tree
d. file size is the size in bytes of the file to compress.

6. Write header for outfile
7. Do post-order traversal on the huffman tree to write to outfile. In the style of tree-dump
8. Write the code for each symbol to outfile with write_code. Then flush cod

The 256 Alphabet array follows this format

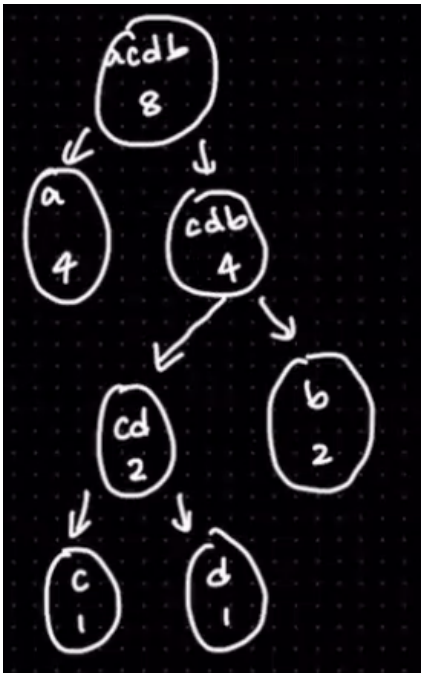
So if you want to add A to the array then you put it in index 96.

Letter q is in index 112.



```
  1 ^A          44 ,          87 W          130 M-^B          173 M--          216 M-X
  2 ^B          45 -          88 X          131 M-^C          174 M-.          217 M-Y
  3 ^C          46 .          89 Y          132 M-^D          175 M-/          218 M-Z
  4 ^D          47 /          90 Z          133 M-^E          176 M-0          219 M-[
  5 ^E          48 0          91 [          134 M-^F          177 M-1          220 M-\
  6 ^F          49 1          92 \          135 M-^G          178 M-2          221 M-]
  7 ^G          50 2          93 ]          136 M-^H          179 M-3          222 M-^
  8 ^H          51 3          94 ^          137 M-^I          180 M-4          223 M-_
  9 ^I          52 4          95 _          138 M-^J          181 M-5          224 M-`
 10             53 5          96 `          139 M-^K          182 M-6          225 M-a
 11 ^K          54 6          97 a          140 M-^L          183 M-7          226 M-b
 12 ^L          55 7          98 b          141 M-^M          184 M-8          227 M-c
 13 ^M          56 8          99 c          142 M-^N          185 M-9          228 M-d
 14 ^N          57 9         100 d          143 M-^O          186 M-:          229 M-e
 15 ^O          58 :         101 e          144 M-^P          187 M-;          230 M-f
 16 ^P          59 ;         102 f          145 M-^Q          188 M-<          231 M-g
 17 ^Q          60 <         103 g          146 M-^R          189 M-=          232 M-h
 18 ^R          61 =         104 h          147 M-^S          190 M->          233 M-i
 19 ^S          62 >         105 i          148 M-^T          191 M-?          234 M-j
 20 ^T          63 ?         106 j          149 M-^U          192 M-@          235 M-k
 21 ^U          64 @         107 k          150 M-^V          193 M-A          236 M-l
 22 ^V          65 A         108 l          151 M-^W          194 M-B          237 M-m
 23 ^W          66 B         109 m          152 M-^X          195 M-C          238 M-n
 24 ^X          67 C         110 n          153 M-^Y          196 M-D          239 M-o
 25 ^Y          68 D         111 o          154 M-^Z          197 M-E          240 M-p
 26 ^Z          69 E         112 p          155 M-^[          198 M-F          241 M-q
 27 ^[          70 F         113 q          156 M-^\          199 M-G          242 M-r
 28 ^\          71 G         114 r          157 M-^]          200 M-H          243 M-s
 29 ^]          72 H         115 s          158 M-^^          201 M-I          244 M-t
 30 ^^          73 I         116 t          159 M-^_          202 M-J          245 M-u
 31 ^_          74 J         117 u          160 M-            203 M-K          246 M-v
 32             75 K         118 v          161 M-!           204 M-L          247 M-w
 33 !           76 L         119 w          162 M-"           205 M-M          248 M-x
 34 "           77 M         120 x          163 M-#           206 M-N          249 M-y
 35 #           78 N         121 y          164 M-$           207 M-O          250 M-z
 36 $           79 O         122 z          165 M-%           208 M-P          251 M-{
 37 %           80 P         123 {          166 M-&           209 M-Q          252 M-|
 38 &           81 Q         124 |          167 M-'           210 M-R          253 M-}
 39 '           82 R         125 }          168 M-(           211 M-S          254 M-~
 40 (           83 S         126 ~          169 M-)           212 M-T          255 M-^?
 41 )           84 T         127 ^?         170 M-*           213 M-U          256 ^@
 42 *           85 U         128 M-^@       171 M-+           214 M-V
 43 +           86 V         129 M-^A       172 M-,           215 M-W
```

Example:



Encoder:

Instead of trying to represent each letter as 8 bits. we can just use two bits
We enqueue all of these nodes into a priority queue
Priority Queue:
The lower the frequency, the higher the priority

Input aaaabbcd gives us this huffman tree:

This huffman tree is after working through the priority queue and looking through everything

Once we constructed the huffman tree, we construct a Code table
We enqueued all of the nodes. Now we dequeue twice to get the children
We join the nodes
The priority Queue will now
If only one item in queue, that must be the root of huffman tree
Now we have to assign a code to each of the nodes
a:0
b:11
c:100
d:101
Priority Queue (5 elements) (Different example but still valid
5,3,2,1,6

slot

[5, __, __, __, __] -> [3 , 5, __, __, __] -> [3, 5,2, __, __] -> [2,3,5, __, __] -> eventually to [2,3,4,5,6]

Is slot position -1 greater than slot?

To dequeue we take out [2] then [3] then [4]
To get the right and left position:

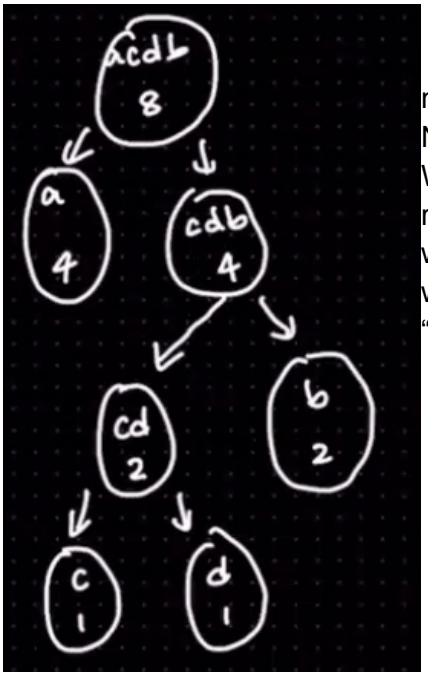
right (k,n) return (k + 1) % n	left(k,n) return (k + n-1) % n

p[slot -1] > 3

Now we do the actual encoding:

Codes a:0 b:11 c:100 d:101	aaaabbcd 00001111100101

To read binary and work down the binary tree, we follow the binary. If it's a 0 we move left, and if it's 1 we move right



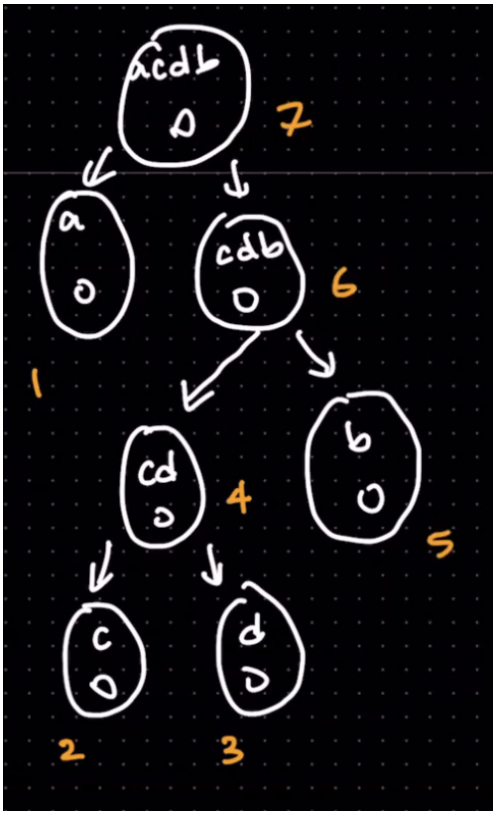
For this the first binary is 0, so we move left, and that is an "a"

next 3 is 0's so we moved left and it was "aaa".
Now we have "aaaa"
We see that the next binary value is 11. So we move down and right twice, which brings us "b". Then again, and we see 100, we move right left left, and get c. So on and so forth we keep working down this and get back to the input "aaaabbcd"

Now how do we tell the decoder what huffman tree we're using?

```
def postorder
        if n != null:
                postorder(n->left) // call post order on left child
                postorder(n->right) // call post on right child
                node_print
```

We need to recursively go through all of the nodes
Similar to dfs with hamiltonian path. We look at left child, right child. If there is no left child then go to the parent.
Print children of node before parent node.
Post order Traversal will print in this fashion:



The Decoder

Decompresses a file

Following is the command-line options for the Huffman decoder:

- -h : Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.
- -i infile : Specifies the input file to decode using Huffman coding. The default input should be set as stdin .
- -o outfile : Specifies the output file to write the decompressed input to. The default output should be set as stdout .
- -v : Prints decompression statistics to stderr . These statistics include the compressed file size, the decompressed file size, and space saving. The formula for calculating space saving is: 100 × (1 − (compressed size/decompressed size)).

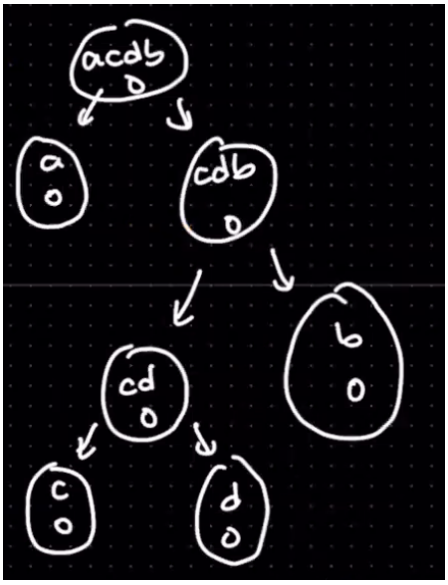
Specifics:

Input file to decompress is named as infile

Output file is named outfile

Pseudo:

1. Read in header from infile and make sure the magic number is valid. Then set the perms
2. Read the tree dump into an tree_size bytes long array. Then rebuild the tree
    a. Iterate over the tree dump from 0 to nbytes
    b. Create the tree with respect to the Interior node's and the Leaf nodes. This is shown in the example below.
    c. The last node in the stack is the root for the huffman tree.
3. Read infile using read_bit and store those into a buffer. Then give out the bits using the buffer.
    a. Starting at the root of huffman tree. Go down the tree, if the bit is 0, go left. If the bit is 1, go right.
    b. At leaf nodes, write the symbol to outfile. Then reset your current node back to the root.
    c. Keep doing this until we match original file size.



Decoder:
L = leaf, I = internal node ($)
This is the dump: La Lc Ld I Lb I I

In order for **decoder** to decode this, we need to use a stack

a 0	c 0	d 0

If the next element of the array is an “I” then we pop the stack twice to get the right and left child.

a 0	cd 0	b 0

Once again we have an I so we pop twice.

a 0	cbd 0

Again an Internal Node

acdb 0

There is only one node left so that is our Root node and our tree looks like that from encoder
We’ve now recreated the Huffman tree and we construct a code table with build_codes. Doing so will bring us the codes
a:0
b:11
c:100
d:101
We now traverse down the huffman tree until the number of decoded symbols == og file size.

Nodes

What is a node? In this lab, nodes are a pointer that holds a pointer to its left and right child, and symbol, and frequency.

What's the point of the nodes? Well Nodes is how we're creating the huffman tree and how we will be able to traverse through the tree and know the frequency of each node.

Node-create

Create The node, set the node symbol to "$" and frequency as the argument frequency
Set the left and right children to NULL, as this may or may not be used later on.

Node-Delete

Delete the node, free the memory and set pointer to NULL

Node-Join(left, right)

Create a PARENT node, and the children will be left and right.
Parent Node's symbol = "$" , Frequency is left frequency + right frequency

~Concerns~

- Make sure to only node_delete from bottom up. Or else some parent will just be NULL and this may cause issues
-

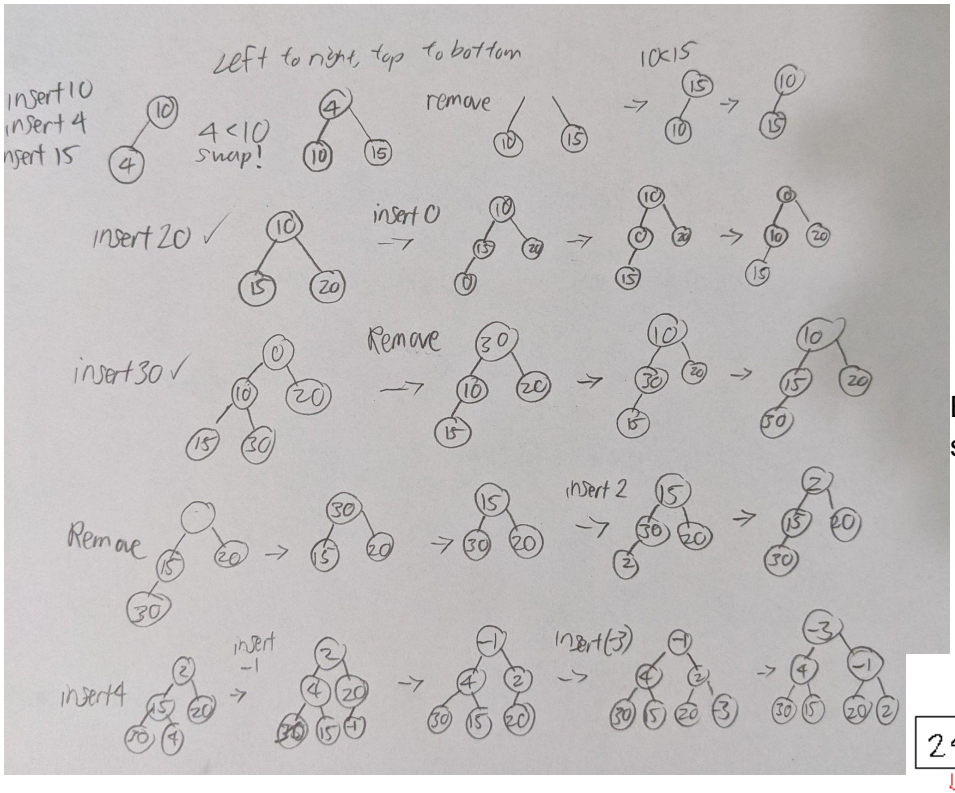
Priority Queues

The encoder uses a priority queue of NODES. The difference between a regular queue and a priority queue is that, high priority are dequeued before elements of low priority.
The lower the frequency of a node, the higher its priority.
Priority Queue could be implemented through 2 methods:

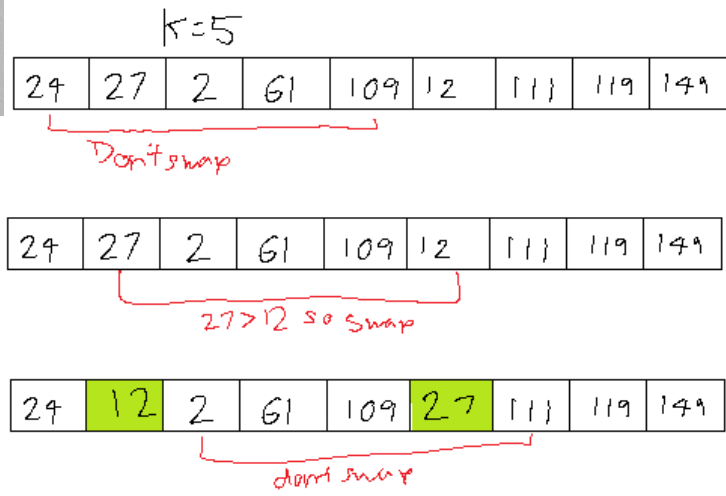
Mimic insertion sort when enqueuing an node. - Find the correct position -> shift everything back.	Using a min heap
What is Insertion sort? Insertion sort is similar to what I did back in Asgn3 with shell sort. Insertion sort is just the general name for shell sort.	What is a min heap? We can follow the williams method. We get the First element, get the next element, and add it.

Min Heap Ex from https://youtu.be/g9YK6sftDi0

Insertion sort has a worse time complexity than min heap does but insertion sort will be easier to debug than the other. So i will be using Insertion sort



Below is an idea of what Insertion sort will look like.



I will keep this min heap design here nonetheless because there are a multitude of options to implement this priority queue

We have to be able to Create a queue, Delete a Queue,
Priority Queue will be use a multitude of times especially during post traversal

In the Queue, the lower frequencies should stay towards the head, (Higher priority).
Enqueues a node into the priority queue
dequeue (DO NOT search for high priority element each time)
Dequeue the HIGHEST priority.
The dequeued node will have the HIGHEST priority over the rest of the nodes

The Queue WILL NOT hold an array of nodes. Queue will hold an array of node pointers. Doing this allows us to call the node functions easier and saves more memory.

Codes

The main point of this is to pass a struct by value. Inside of the Huffman tree we will need to remember and hold the stack of bits. This ADT represents a stack of bits

No Dynamic memory allocation is needed. All that will be done is create a new Code on the stack, set the top to 0, and zero out the array of bits. We initialize a struct, copy it
We need to know if it's empty or full
We need to push a bit
We need to pop a bit
And print the code
How do we Write_code?
With the code: 00001111100101,

we need to buffer bits
When the buffer is full, write out the entire buffer

The reason why we need to buffer the code is because it's just far more efficient than writing out one bit at a time.

```
for loop (i=0 to c->top)
        if get_bit(c,i) ==1:
                //If the current bit is a one then we need to set a bit in the buffer index
                set_bit(buffer, buffer index)
        else
                clr_bit(buffer, buffer index)
        buffer index +=1 // Increase the buffer by one
        If buffer index == block *8: // If buffer is full then write it
                write_bytes(outfile,buffer, BLOCK)
                buffer index =0 // Reset the buffer index
```

if the buffer isn't full but there's still bits inside of it then we need to flush_codes

```
Flush_codes(outfile):
        if buff index is greater than 0:
                write_bytes // Convert num of bits into the least number of bytes possible and write it
```

(This points at the LSB)
buffer[0] = MSB 0000 0000 LSB
If code d = [101]
We call code with (&d) // So we call code with a pointer.
then buffer[0] = MSB 0000 0001 LSB
then buffer[0] = MSB 0000 0101 LSB

If code c = [100]
We call code with (&c) // So we call code with a pointer.
then buffer[0] = MSB 0000 0101 LSB
then buffer[0] = MSB 0000 1101 LSB

I/O

I/O will be used by both encoder and decoder. I will be using low-level system calls (syscalls) like those used in MIPS. Such as read(), write(), open(), and close().
I will need to make use of static variables to keep tracks of bits
Do not do File *in fopen() because we require to use low-level system calls

I could either make this a circular read or make it linear. For the time being I will make this as a linear style because it's easier to work with.

read_bytes (file, buf, nbytes)

(Use this whenever we need to perform a read)
Pseudo code inspired from tutor Eric

Create Variables for Bytes read & another for total bytes read
A loop to read() until we've read all the bytes or there are no more bytes (nbytes).
if bytes == -1 then there's an error
Increase total by bytes
buffer += bytes //Increase the position of the buffer
if total ==nbytes then break
return

write_bytes

Pseudo code inspired from tutor Eric

Similar to read_bytes. But instead
Create Variable for Bytes written and for total bytes written
we loop to write() until we have written all the bytes from the buffer or no bytes were written.
If bytes == -1 then there's an error
Increase total by bytes
buffer += bytes
if total ==nbytes then break

write_code

Similar to read_bit. We use a static buffer and index.
Each bit in the code will be buffered into the buffer.
Once the BLOCk is full, we write it to the outfile.

Pseudocode:

Same as before we create the variables.
While loop for when we are still able to write something
Do a check to see if it's valid
Increase my total by bytes
Buffer += bytes
If statement to break out of the while loop

read_bit

Notes*
buffer a block of bits 4096 at a time we only read if a buffer pointer is back of 0 if it's end of the block then we would do a read Notice how we're supposed to read a BIT and not a Byte. We can't read just a single bit. So what we have to do is read a byte into a buffer and give it out one at a time. When that's done, we fill the buffer back up with bytes. We need to hold a STATIC buffer so we can remember the pointer. GLOBAL int index = 0 We need to maintain a static buffer of bytes and index into the buffer that tracks which bit to return

General Pseudocode from Euguene

If we want to read in a file and we have a buffer from buffer[0] to buffer[4095]
buffer[0] = MSB 1101 1101 LSB
when buffer is empty
    fill it
pass back bit at buff index
buff index +=1
    //Uses functionality of read_bytes

Pseudo code

Set a variable to the invalid return from read
Create an if statement that checks if there's less than a BLOCK to read
If yes then there is nothing equivalent to a block. I could only read less than a block.

*bit = buffer[index/8] >> (index %8) //What this does is Right shift and get that bit.
& with a 1

flush_codes

As there are 8 bits in a byte. We might not always have the full 8 bits in the write_code buffer. So we turn the extra bits into zeros.

Stacks

Stack of Node pointers
As we need a Huffman tree, we use a stack in our decoder to create one. We store the nodes in the stack. The struct of stack will hold the top number, capacity, and Node **items.
There isn't much to be said about stacks just do the standard: Create, delete, empty? full? size? push, pop, print. Something to note is that This stack holds node pointers, not actual nodes.

Huffman Coding Module

Let's build the huffman tree. We will be given a computed histogram and build the tree based off of that.

Build Tree (Histogram as argument) && **Delete Tree**

ALPHABET indices, one index for each symbol.
Return root node.

build codes

Fill up a code table, build the code for each symbol in huffman tree. These codes are copied to code table that have alphabet indices. Each index is a possible symbol.

rebuild tree

We Reconstruct the huffman tree from the post-order tree dump. The length is tree_dump. Then we return the root node.