

## ASGN6 Design

What is a Huffman tree?	
nodes-> pq-> TEST-> codes-> I/O-> stacks	

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

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 with a priority queue.
  - a.

Example:

### Encoder:

Instead of trying to represent each letter as 8 bits. we can just use two bits

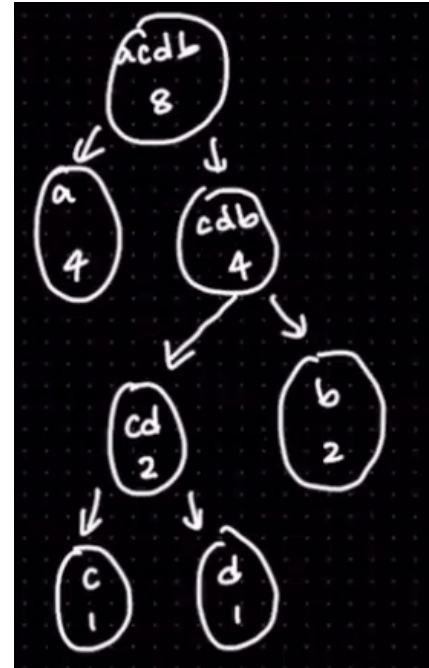
We enqueue all of these nodes into a priority queue

Priority Queue:

The lower the frequency, the higher the priority

Input aaaabbbcd 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

[ 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:

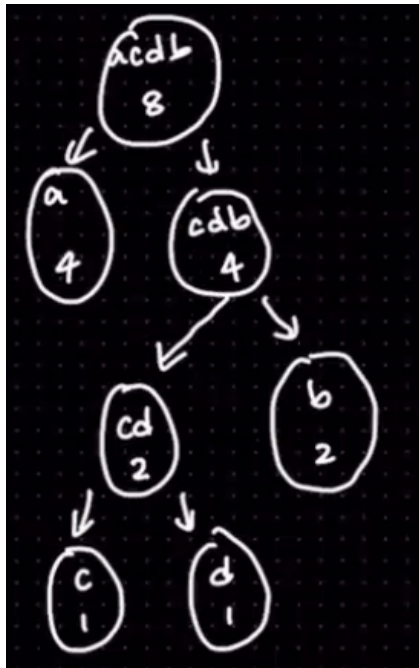
right (k,n) return (k + 1) % n	left(k,n) return (k + n-1 ) % n
-----------------------------------	------------------------------------

p[ slot -1 ] > 3

Now we do the actual encoding:

Codes a:0 b:11 c:100 d:101	aaaabbbcd  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 "aaaabbbcd"

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

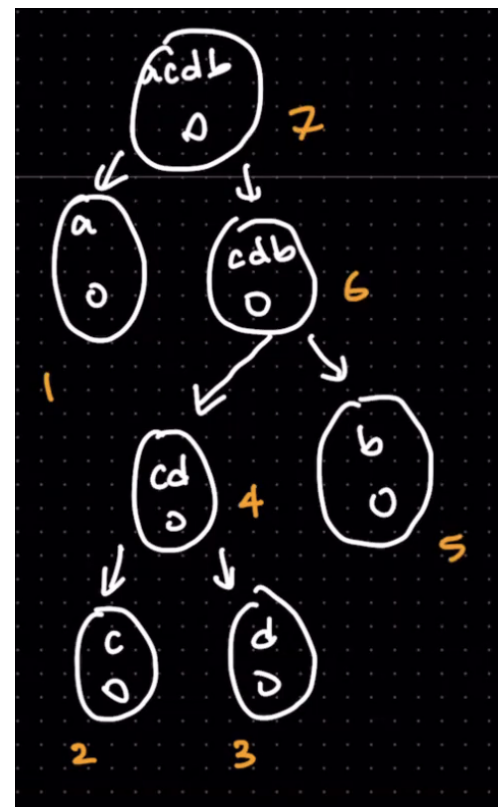
We need to recursively go through all of the nodes

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:  $100 \times (1 - (\text{compressed size}/\text{decompressed size}))$ .

Specifics:

Input file to decompress is named as infile

Output file is named outfile

### 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
-----	------	-----

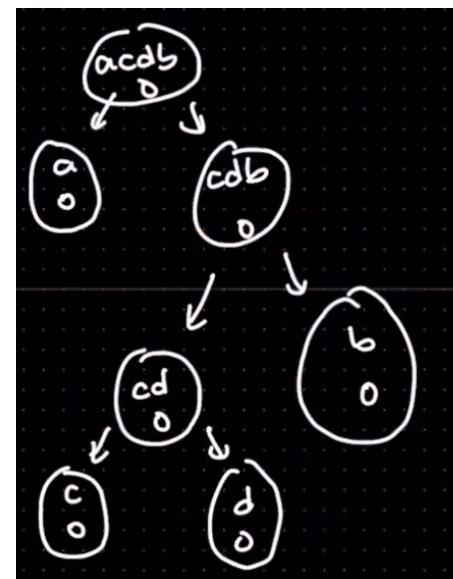
Once again we have an I so we pop twice.

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



## 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.

### 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

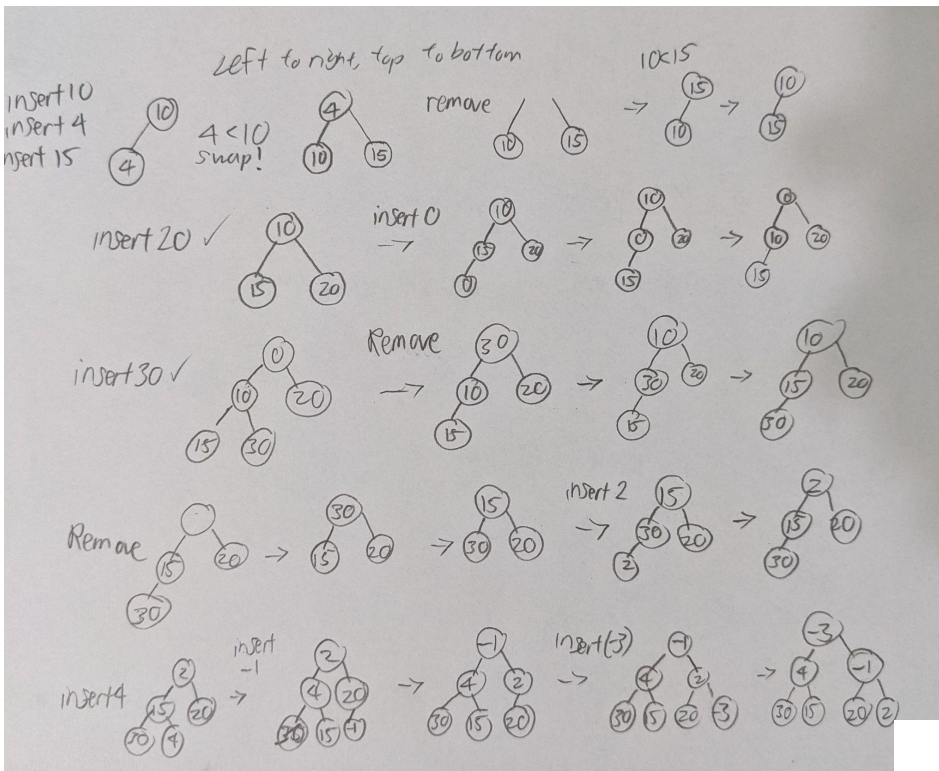
The encoder uses a priority queue of NODES. The difference between a regular queue and a priority queue is that, high priority are dequeued before elements of low priority.

The lower the frequency of a node, the higher its priority.

Priority Queue could be implemented through 2 methods:

Mimic insertion sort when enqueueing 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.

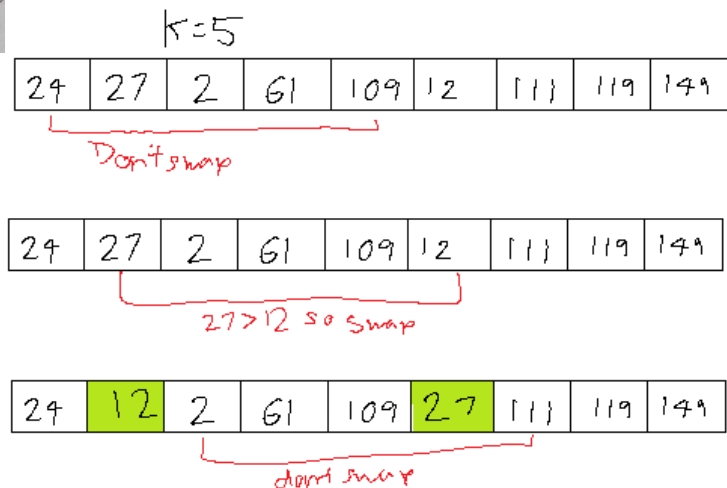
Min Heap Ex from <https://youtu.be/g9YK6sftDi0>



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

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.



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

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

## 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

read\_bytes (file, buf, nbytes) (Use this whenever we need to perform a read)

A loop to read() until we've read all the bytes or there are no more bytes (nbytes).

write\_bytes

Similar to read\_bytes. But instead we loop to write() until we have written all the bytes from the buffer or no bytes were written.

read\_bit

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.

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

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 outfile.

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.

## 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)

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.

rebuild tree