the huffman algorithm!

Data compression.
encoder and decoder to save memory!
Will use a priority queue in order to build a huffman tree. The huffman tree will serve as a "traslation" table to encode and decode.
need to build nodes and trees!
need to build priority queue (use a heap!)
need an ADT to track binary codes!
need to build functions to read/write bits/bytes
need a stack to decode!

encoding

keep track of symbol frequencies when parsing a file!
256 indicies. (each char is 1 byte = 256 possible vals)

construct the huffman tree! use priorty queue!
it should be a min heap (go fast!) so you can constantly pull off smallest frequency!
fill priority queue; after pop off pq and start building nodes from popped vals. then sum left and right node and throw the new parrent back into pq
do this until pq size > 1, the last element is root node!
deque last node after loop.

encoding

traverse the tree to build codes! when going left, add 0 to code when going right, add a 1
when you find a leaf node, assign the curr code to that symbol. traverse the whole tree. symbol/code table has 256 indicies

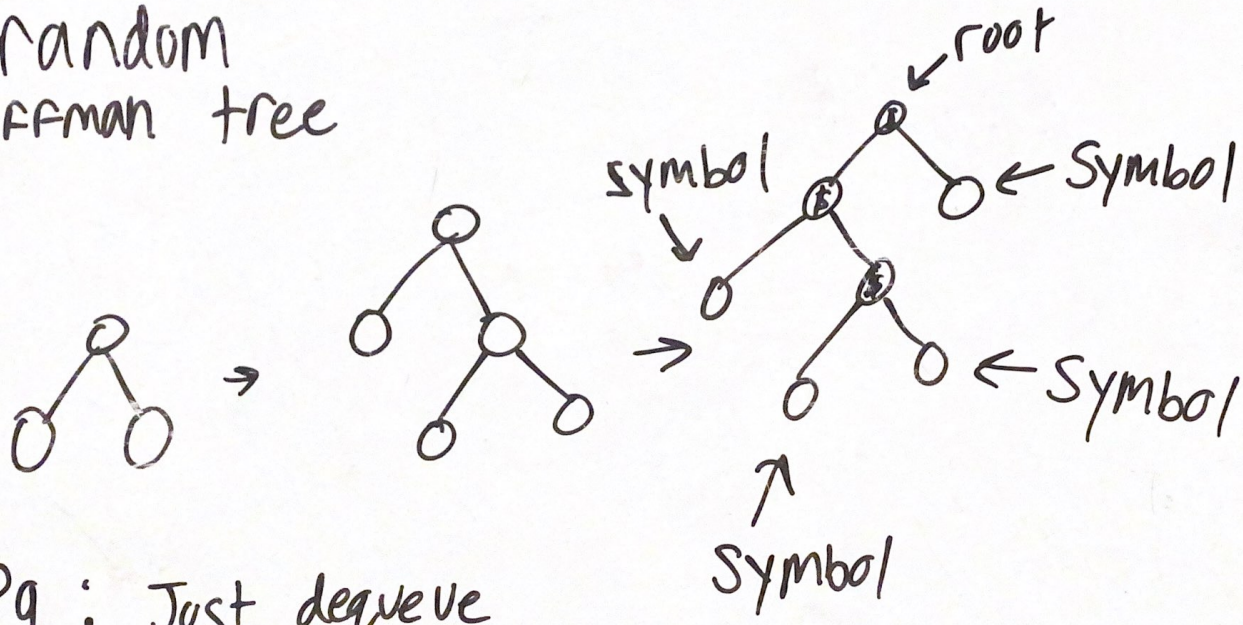
dump the tree by traversing the tree using post order traversal. at a leaf node, write L and symbol when not on a leaf node write I and keep traversing until whole tree is traversed.

go through output and write out each symbol's code

Decoder

takes encoded file as input. reconstruct huffman tree using the tree dump and a stack
create a node for each symbol and push to stack when decoder reads 'L' when decoder reads an 'I'
right = pop() left = pop(), join these nodes with a parent, and push parent onto stack. keep repeating until tree dump is finished. tree is reconstructed.
traverse the codes made in encode. go left if 0 right if 1
when you hit a leaf, write symbol, go back to top of tree. repeat until end of code!

random
huffman tree



Pq: Just dequeue
smallest element
to build huffman tree

codes includes U32 top and array of bits @ $\frac{256}{8}$ = length

codes will be allocated ON the Stack

set code.top = 0 set all bits in array to 0

code size = top

code empty = true if top = 0

Set bit: or code with 1 shifted left i times!

clear bit: and code with the negation of 1 shifted i times

get bit: and code with 1 shifted left i times. then right shift i times.

Push bit: Shift code left and set LSB to i inc top

Pop bit decrement top. return popped val

Priority queue:

empty returns true if size=0

full returns true if size = capacity

size returns size

priority queue's list will be a min heap that dequeues the smallest element then fixes the heap.

enqueue throws an element in the list then fixes heap.

print prints the array in priority queue

Priority queue:

empty returns true if size=0

full returns true if size = capacity

size returns size

priority queue's list will be a min heap that dequeues the smallest element then fixes the heap.

enqueue throws an element in the list then fixes heap.

print prints the array in priority queue

nodes: have right pointer, left pointer, symbol and frequency.

joining nodes takes a left node and a right node and creates a node parent with the children left and right.

create: allocate memory for node

delete: free memory from node

print: print all attributes of node n

I/O using low level syscalls. static buffer for file

read and write will be loops calling read(), write() until a certain number of bytes are read, or EOF.

read bit will take in a buffer of bytes and use the "codes" functions to get one bit at a time.

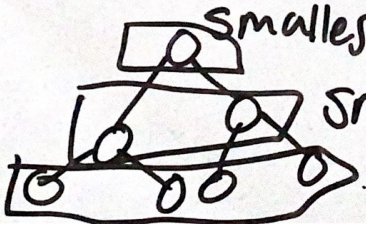
write code will also use the buffer to slowly take bits then, write those bits into outfile.

flush codes clears left over bits in the buffer.

Stack will be normal (same as before)
so S→top and S→capacity except the array
items will now hold Node pointers

Stack print will just print the nodes
in the Stack calling node print.

Min heap (for Priority queue)
just like max heap from heap sort, but the smallest element should be on top so look for min child. the heap elements get bigger down the tree. Pull the top element off using dequeue and then fix the heap for the next dequeue! (maybe call before dequeue so you dont have to call after every enqueue.)



node: left pointer, right pointer, symbol frequency.

Joining two nodes creates a node and then sets the two nodes as left and right

---

Priority queue
- Array is min_heap
- when enqueueing, make a heap
- when dequeueing fix the heap
- dequeue swaps first and last element! it then lowers tail. heap will fix before next dequeue

I/O

read_bytes: loop over read, filling into a buffer. increment total bytes read

write_bytes: loop over write, from buffer increment increment total written bytes.

read_bit: use bv techniques to go into a particular byte and bit. [index/8] for byte index %.8 for bit.

write_code: writes bit from a code into buffer, if, buffer is full, flush code to outfile!

flush: does ceil division on index to see how many bytes needed to fit that many bits.

File Size - bytes_read = bytes to read

once bytes to read < Block
then read(bytes to read)

if bytes to read > Block
read in a block

when looping over buffer,
loop over

if bytes_to_read < Block
loop over buffer using
file size - bytes read as
the max!

For encode!

header:

magic num: 0x BEEFD00D

File size: From Fstat!

tree size: loop through hist, if an index is >0 then add to count. after, (count *3) - 1 is the tree size

Permission: from Fstat. set the outfile to the same permissions.

do this in encode, write this header to outfile!

encoding!

bytes_read in io.c will say total bytes read.
you have size of file!

while (bytes_read < size of file) read whole file
 bytes_to_read = size_of_file - bytes read
 if bytes_to_read > Block:
    read in block
    Fill hist
 else:
    read in bytes_to_read
    fill in hist

while filling in hist keep track of unique symbols!

decoding!

read in header and use bit shifting to recover numbers from bits!

read in tree dump (you know the size)

use read_bits to read bits, if 0 go left, if 1 go right. (after using rebuild_tree)

when node.left and node.Right == NULL print symbol. loop this until bytes_written = old file size