DESIGN

Purpose The purpose of dsgn.c / Huffman coding is to read an input file and encode it using Huffman methods i.e a Huffman tree, histogram nodes, priority queue, stack, ~~and~~ codes, and a custom input/output system. After encoding a file we ~~would also like~~ have successfully compressed it using less bits than the original file and can now decompress it w/ reverse logic. The decoder reconstructs a huffman tree given a tree dump and reads in bits to traverse the tree and output the proper symbols/characters to a given outfile.

Layout/Structure

Node.c File containg Node ADT that we will use for our priority queue

- Node {
  Frequency;
  Symbol;
  }
  - Nodes have two attributes i.e. Symbol and frequency
  - Symbol will be the numerical value of a character & frequency is how many times that character appears in ~~our~~ the input file

Node_print → to debug and see nodes

node_join → To connect nodes to one parent
- For organization sake, all parents have the symbol "$" and their frequency is the summation of both children's frequency.

Thus, we can access all children from one parent.

Code.C • File containg Code ADT

```
Code {
    top
    bits[]
}
```

• Two attributes i.e top & bits.

• Top is ~~where~~ index of where to place the next bit & where to pop bit
• Bits is an array of uint8_t that we use to house bits

• Code ADT is essentially a path to traverse our Huffman tree.

• A 0 will indicate to go left & a 1 will indicate to go right.

Huffman.C • File containg functions to encode and decode a Huffman tree

Void treeC PriorityQueue *pq)

• A helper function that helps build_tree.
• It takes a priority queue and dequeues two children to join them w/ a parent ... eventually we will only have 1 parent namely the "root" node
•

Node * build_treeC histogram)

• Function takes a histogram w/ all unique symbols and frequencies and appropriately creates a node for each one.
• It creates a priority queue and enqueues each node
• Calls "tree" function and returns the last root node.

build_codes (Node *root, table) • Function takes a root Node & a table ~~of~~ of which we will fill.
•Uses postorder helper function

PostOrder • traverses down the tree ~~Keep~~
• Initializes a code that tracks lefts and rights... IF we reach a ~~not~~ leaf node we add the current code
• When we go backwards up the tree we pop a bit accordingly.

rebuild_tree(n bytes, tree[])

- Function takes num. of bytes and a tree dump ~~that~~
- Iterates through all bytes in tree dump... if it ~~was~~ detects a 76 or "L" then we know the character after is a leaf node, so we add it to the stack
- If ~~it~~ we dtect a 73 or "I" we found an interior node & we pop two children and join them as a parent back onto the stack
- Eventually, we return one root node left.

IO.C

File containing functions to read input and give an output using a buffer system.

read_bytes(infile, *buf, n bytes)

- A helper function for read_bit
- Its purpose is to continuosly read through an infile and load bytes into a given buffer until it reaches a given amount of bytes.
- We accomplish this w/ a while loop that doesn't break until the total amount of bytes read is equal to the desired amount of bytes.

read_bit(infile, *bit)

- Since we want to preserve space, this function will return the bit at the current bitIndex of a given infile
- It uses read_bytes as a helper function to fill the buffer if it is empty or when it becomes full
- We use bitIndex and Shifting to retrive a particular bit in the buffer.
- Ex: If we want bit 9, then Index = $\frac{9}{8}$ = and the bit we wish to return would be bit = buffer[1] >> (9%8)&1)

write_bytes(outfile, *buf, nbytes)

- A helper func. for write_code; it continuously writes to an outfile until the amount of bytes written are equal to the desired amount.

While (total bytes written != bytes desired)

    write(outfile, buffer, bytes desired)

write_code(outfile, code *c)

- uses write_bytes to write only when our buffer is full of codes.
- Otherwise, we iterate through all bits in our code and add them to our buffer via shifting =>

Again, if we see a "1" in our code then we set a bit using

buffer[bitIndex/8] |= (1 << (bitIndex % 8));

- In the end we flush codes to ensure ~~that~~ we always write necessary codes

flush_codes(outfile)

- A helper function for write_code to ensure that all codes are written
- We ensure we have codes in our buffer by checking if bitIndex > 0... then we write_bytes()
  and initialize everything in the buffer back to zero.
- Set bitIndex to zero.

stack.c • File containing the stack ADT...

- Used to rebuild a huffman tree
- Attributes needed... - top (to track pushes/pops)
  - capacity
  - items (to hold nodes)

stack_push(Stack *s, Node *n)

- We wish to push a node where top is pointing
- We cannot push if stack is full

stack_pop(Stack *s, Node **n)

- We cannot pop if stack is empty
- We wish to decrement top (bc it is pointing to nothing) & use it to return the node at which it is pointing to in n.

encode.c A file that joins all components to encode a file of characters using a histogram, priority queue, huffman tree, and codes.

- We begin by iterating through the file... We read and fill a buffer again and again until we have read all bytes.

- ~~We count~~ Each "fill" of the buffer, we will ~~co~~ iterate through the buffer and count all ~~them~~ occurences of a character.

- Now we build a huffman tree via "build_tree()" and store result as a root node

- Next, we construct a header that contains file size, tree size, magic number (to ensure we are decoding a decodeable file), and permissions to transfer.

- To construct a dump, we traverse the tree w/ a post order method. When we have nowhere else to go we are either on a leaf or interior node. If it is a leaf node we add a "L" to an array and the symbol next to it. If interior node we add a "I", ~~and~~ Lastly we write the tree dump to the outfile.

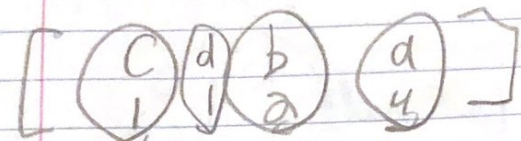
- ~~To c~~ Similarly, we construct codes for each unique symbol by traverse the tree and tracking a "path" w/ a code. 0 is left, 1 is right. When we have nowhere else to go, we pop a bit.

We write codes to outfile.

Encode Walkthrough

Input: aaaabbcd

Symbol	Frequency
a	4
b	2
c	1
d	1

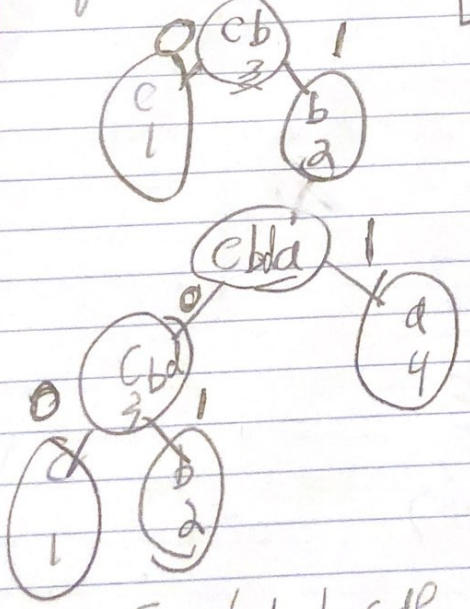


want we want

head

tail

dequeue & join

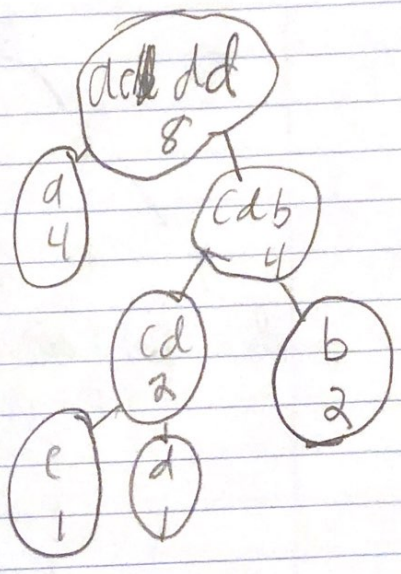


Total

c=000

b=001

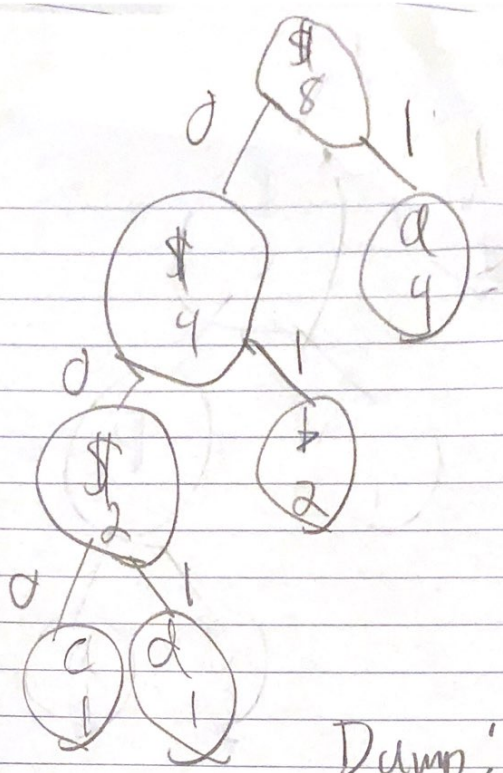
a=1



Symbol	Code
a	0
b	11
c	100
d	101

0000 1111 100 101

a a a a b b c d



Dump; LcLdIbIaI

Codes	Symbols
0	c
0 0 1	d
0 1	b
1	a

Decode.c

- We begin by reading the header to determine file_size & whether it is decodable (Magic number), file permissions, and tree size

- Next, we read the tree dump and place it into an array such that we can use to ~~ref~~ call "rebuild_tree()"

- We rebuild using a stack and store the result in a root node

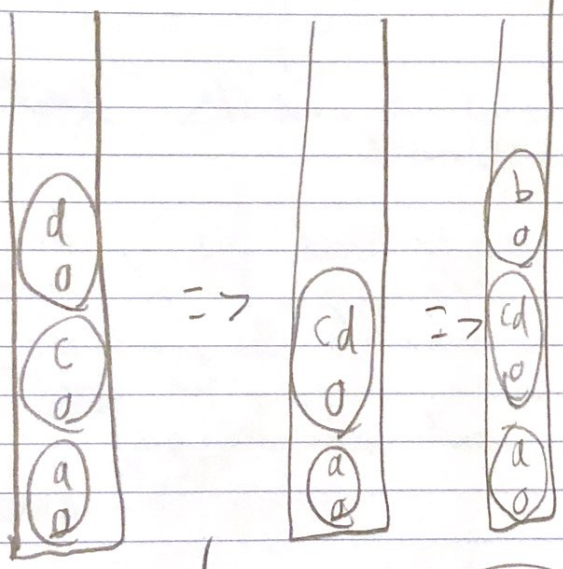
- Next, we continuously read bit by bit and traverse down our tree accordingly. If we read a 0, we go left. If we read a 1, we go right. Once we have nowhere to go, we have reached our desired symbol and can output it to outfile.

Decoder Walkthrough

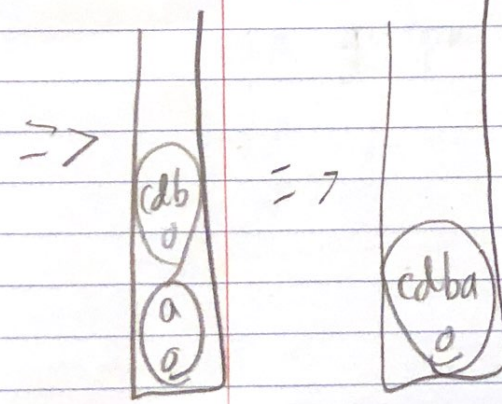
- We first ensure that the magic number is valid and that the file is decodeable (this is done by reading in the header file).
- Next, we read the tree dump & reconstruct

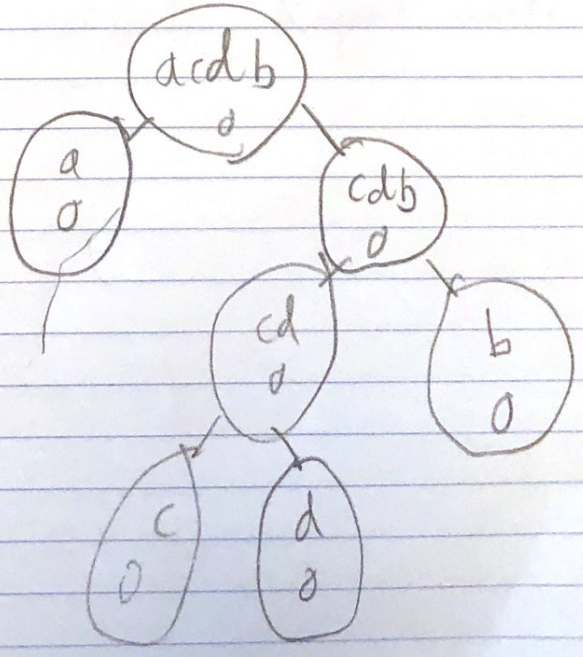
Use a stack

LaLcLd I Lb I I



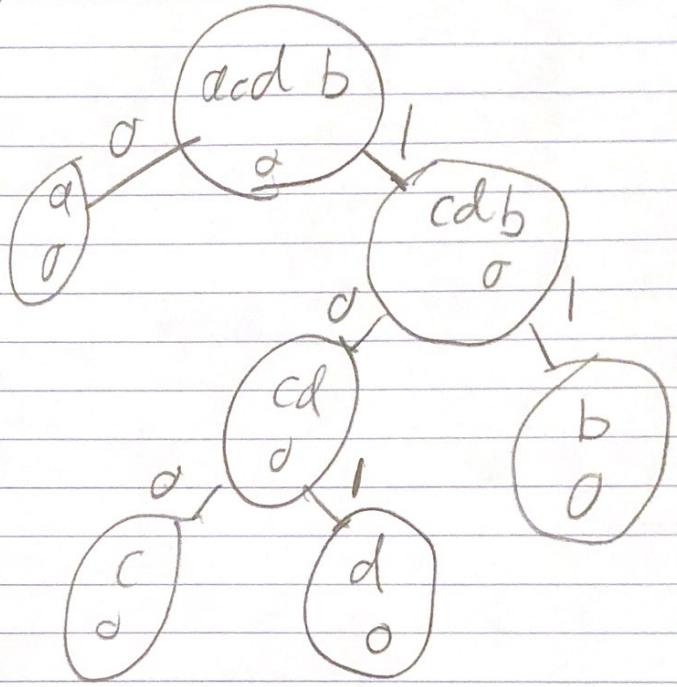
- We add node when "L" but pop two when "I"





Now we read bit by bit & traverse down our reconstructed tree.!

Ex: "101"



we reach Symbol "c"

and output to outfile. Repeat until no codes left...