Huffman

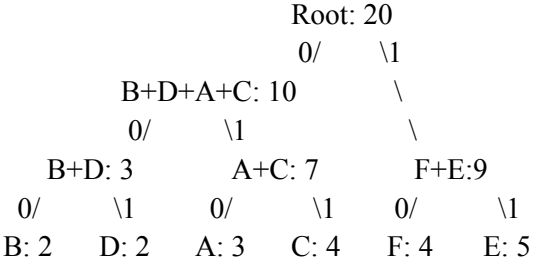
Huffman is meant to reduce sizes of files. This program works with specifically with text files and should work at any inputted file length.

Encrypt: Finds a more efficient code for each character and outputs an encode with a smaller number of bytes

Decrypt: Decrypts an inputted file and expands the bytes into chosen output.

Message: AAABBCCCCDDEEFFFFEEE

A: 3
B: 2
C: 4
D: 2
E: 5
F: 4



Code
A: 010
B: 000
C: 011
D: 001
E: 11
F: 10

The most frequent appearing symbols get the shortest codes. F,R with 2 bits, and 3 for the others.
AAABBCCCCDDEEFFFFEEE => Encoded
Encoded: 01001001000000001101101101100100111111010111111
Much shorter than having 160 bits, (8bits (20 Symbols))

Pseudo Code Encoding:
Use priority queue, with the lowest frequency being placed at the top. Use min-heap sort to order.

While queue > 1

- Dequeue 2 nodes and make a parent with children of 2 taken 2 nodes. Place the parent in its proper place in the queue.
- Eventually, there will be only 1 left in the queue which will be the root node

Open File:
While != EOF
- Take 1 character. In output file fprinf(New code)

Pseudo Code Decoding:
Open encode_output file
While != EOF:
- While(Parent node has childen):
    - dec_tmp: stores the bits of the bits read
    - Read 1 bit. 0 => getLeftChild, 1 => getRightChild
- Dec_tmp = Null

The Encoder:
1. Read an input file, find the encoding of its contents, use the encoding to compress the file
2. Options -[hi:o:v]

Encode File Algorithm?:
- Create Histogram of the File: Count the number of occurrences of each unique symbol
- Compute *Huffman Tree*? using computed Histogram, requires *Priority Queue*?
- Construct a code table. Each index of the table represents a symbol and the value at the index of the symbol's code. You will need to use a *stack of bits*? And perform a traversal of the Huffman Tree.
- Emit an encoding of the Huffman Tree to a fle. Thai will be done through a *post-order traversal*? Of the Huffman Tree. The encoding of the Huffman tree will be referred to as a Tree Dump?
- Step through each symbol of the input file again. For each symbol emit its code the output file.

The Decoder:
1. Read in a compressed input file and decompress it.
2. Options -[hi:o:v]

Decode File Algorithm:
- Read the dumped tree from the input file. A *stack of nodes*? Is needed in order to reconstruct the Huffman tree
- Read the rest of the input file bit-by-bit, traversing down the Huffman tree one link at a time. Reading a 0 means walk down the left link, and reading a 1 means walking donw the right link. Whenever a leaf node is reached, its symbol is emitted and you start traversing again from the root.

Nodes:

- Huffman trees are composed of nodes, each node contains a pointer to its left child, a pointer to its right child, a symbol and the frequency of that symbol.
  struct Node [L_child , R_child, Symbol, Frequency]
- Use uint8_t, instead of char to interpret as raw bytes

Priority Queues:

- Like a normal queue, but elements are assigned priority, such that high priority elements are dequeued faster than low priority ones.
  - The lower the frequency the higher the priority

2 Pass:

1. Create Histogram
2. Emit the code:

Io.c

int read_bytes(int infile, uint8_t *buf, int nbytes)

- Use the write()

int write_bytes(int outfile, uint8_t *buf, int nbytes)

-

bool read_bit(int infile, uint8_t *bit)

void write_code(int outfile, Code *c)

void flush_codes(int outfile)

Build codes (Node *root, Code Table [ALPHABET])

message = bananas

a : 3
b : 1
n : 2
s : 1

→ insert sort

a : 3
n : 1
b : 1
s : 1

Code c : //size 0

Root = $:7

↑ Not a Leaf

Pop:0

Left

Right

Code : 0

Node = a:3

↳ LEAF

↳ a : 0

Right

Code:1

Pop:0

Left

Code : 10

Node : $:2

↑ Not a Leaf

Pop:0

Right

Code:11

↳LEAF

↳ n:11

Left

code : 100

Node : s:1

↳LEAF

↳ s : 100

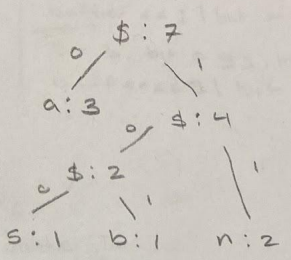
Right

Code : 101

Node : b:1

↳ LEAF

↳ b : 101



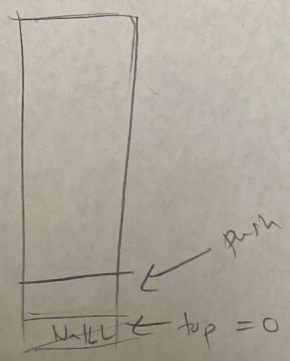
codes

a	0
b	100
n	11
s	101

Stack

Stack_push(stack s, Node an)

top is 1 above the current
so top == size



top+1

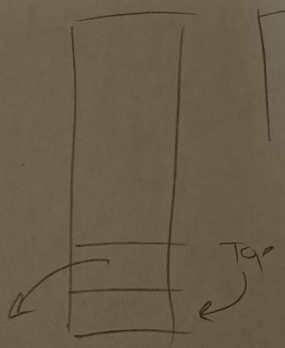
stack[top] = Node n

top++ increment

Stack_pop(Stack s, Node an)

top -- move down

an = stack[top]



Write_Code (int outfile, code c)

size_buff = 1
Buffer [1] = ----------

code: 1011 size 4

for (code_size (code); b ++)
  if (ind_b << sizeBuff * 8)
    bit = c [b]
    buffer << 1 | bit
    ind_b ++
  else
    Flush

leading 0-s

b = 0, bit = 0 1, ind_b = 1
buffer | bit = 00000001

b = 1, bit = 0 0, ind_b = 2
buffer << 1 | bit = 00000010

b = 2, bit = 0 1, ind_b = 3
buffer << 1 | bit = 00000101

b = 3, bit = 0 1, ind_b = 4
buffer << 1 | bit = 00001011

code 11001 size 5

buffer = 00001011 ← call it buff

b = 0, bit = 1, ind_b = 5
buff << 1 | bit = 00010111

b = 1, bit = 1, ind_b = 6
buff << 1 | bit = 00101111

b = 2, bit = 0 ind_b = 7
buff << 1 | bit = 01011110

b = 3, bit = 0, ind_b = 8 ← Flush after this Loop
buff << 1 | bit = 10111100

Flush
buff = 10111100 → outfile ind_b = 0, ind = 0

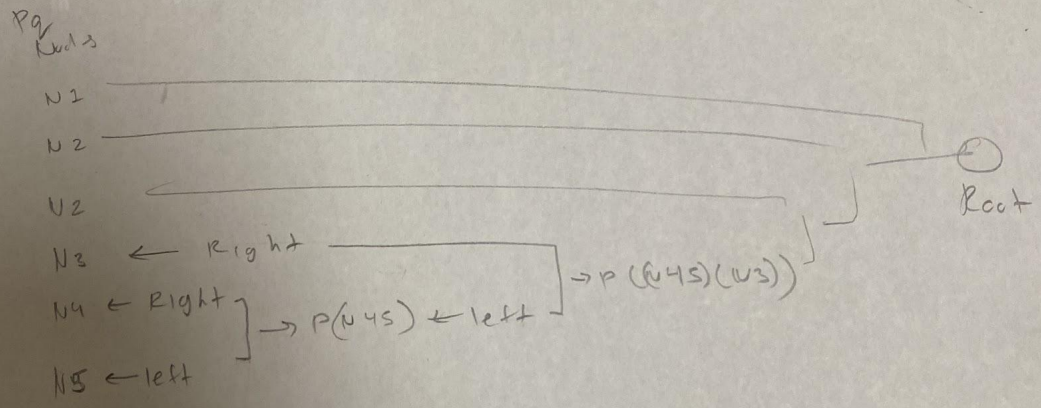
b = 0, bit = 1, ind_b = 1
buff << 1 | bit = 00000001

Build_Tree

```
pq-create(ALPHABET)
for(ALPHABET; a++)
    if(hist[a] > 0)//If there is more than 0 or a letter
        node_create(i, hist[i]); // node(char, frequency
        enqueue

while(pqsize > 1)
    left = dequeue
    Right = dequeue
    parent = join(Left,Right)
    equeue(parent)

dequeue(root)
```



IO...

read_bytes(infile, buf, nbytes)
read(infile, buf, nbytes)

write bytes
write(outfile, buf, nbytes)

---

write_code(int outfile, Code *c)

for(length of code)
bit = c -> bits[i]
buffer[ind/8] = buff[ind/8] << 1 | bit
ind++
i++
when(buffer full) FLUSH.

---

flush_codes
int bytes = (ind + 8) / 8
write_bytes(outfile, buffer, bytes)
ind = 0;