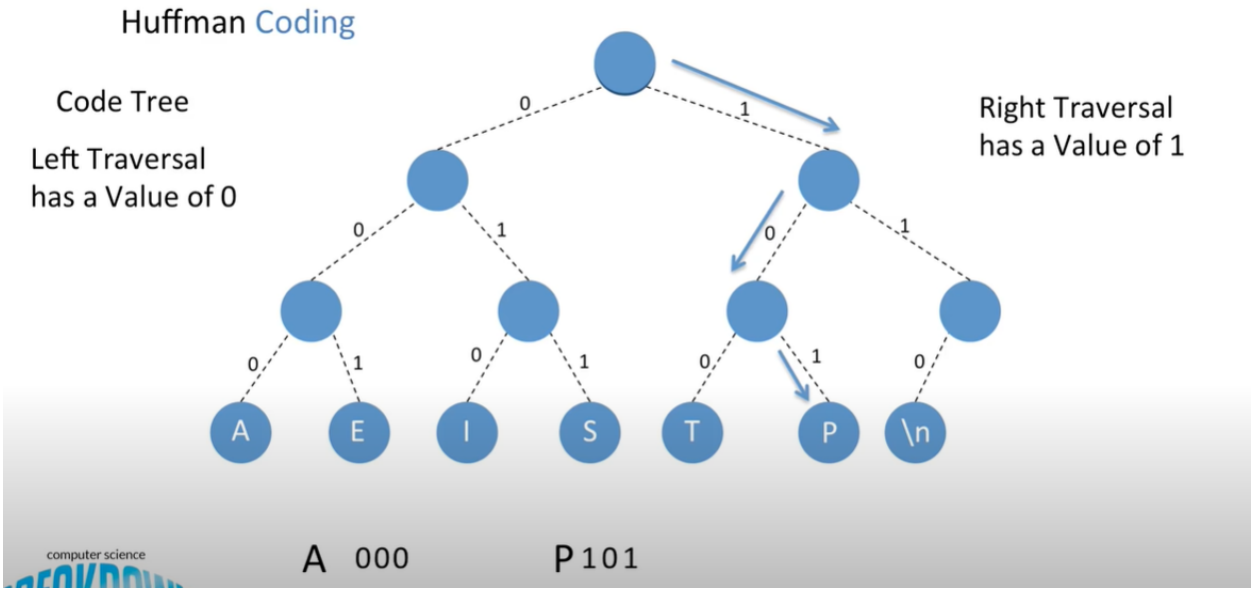
Purpose of the lab:
The program will implement Huffman encoder and decoder. Encoder will read in a text, use tree data structure to reassign bits for each character based on the text and compress the file size. Decoder will receive information about the tree data structure and encode output, and then decode it back to the original text characters.

Command line options:
Encode:
-h: print help message that describe purpose of the program, command line options,
-i: Specifies input file
-o: Specifies output file
-v: Print compression statics(include uncompressed file size, compressed file size, and space saving)

Decode:
-h: print help message that describe purpose of the program, command line options,
-i: Specifies input file
-o: Specifies output file
-v: Print compression statics(include uncompressed file size, compressed file size, and space saving)

Files:
Encode.c: Encoder's main function
Decode.c: Decoder's main function
Printtree: helper binary to debug(will graph the tree data structure based encode output(I for interio nodes(child/element), L for out parent node)
Entropy.c: Provided, check for entropy
Defines.h: macro definitions, should used throughout the program
Header.h: Include information about magic, permissions, tree size, and file size after encode
node.c/.h: Node is the element of the tree data structure(for more detail, check encode section below), node.c will implement node ADT interface(node create, node delete, node join)
pq.c/.h: we will enqueue and dequeue nodes based on character's frequency(how many times each character appeared in the text?)
code.c/.h: record the new code for each character based on tree data structure
stack.c/.h: Stack ADT interface that need to use in decode process
Huffman.c/.h: the functions to build tree data structure
io.c/h: Functions to read in file and output file

Tree data structure:



Encode:

1) Command options
2) Read in file, if magic number not equal 0xDEADBEEF (defines), then meaning invalid file.

3) Huffman.c, build tree(create pq, use pq to insert node), for detail build_tree function, read pdf specifics for encode part 3

   (a) Create a priority queue. For each symbol histogram where its frequency is greater than 0 (there should be at minimum two elements because of step 2), create a corresponding `Node` and insert it into the priority queue.

   (b) While there are two or more nodes in the priority queue, dequeue two nodes. The first dequeued node will be the left child node. The second dequeued node will be the right child node. Join these nodes together using `node_join()` and enqueue the joined parent node. The frequency of the parent node is the sum of its left child's frequency and its right child's frequency.

   (c) Eventually, there will only be one node left in the priority queue. This node is the *root* of the constructed Huffman tree.

4) Create code table with 25 bits to record new codes from tree data structure(code.c) Read pdf to build functions

(a) Create a new `Code c` using `code_init()`. Starting at the root of the Huffman tree, perform a *post-order* traversal.

(b) If the current node is a leaf, the current code `c` represents the path to the node, and thus is the code for the node's symbol. Save this code into code table.

(c) Else, the current node must be an interior node. Push a 0 to `c` and recurse down the left link.

(d) After you return from the left link, pop a bit from `c`, push a 1 to `c` and recurse down the right link. Remember to pop a bit from `c` when you return from the right link.

5) Construct header to out file(wrtie_bytes)
6) Write_tree, I for interior nodes, L for parent nodes
7) Write_code and flush_code
8) Close files

Decode:

1) Command line options
2) Read in files after encode, if not 0xDEADBEEF, then it's wrong file
3) Permission stuff (chmod)
4) Read header file informations(tree_size), then rebuild_tree(read pdf for how to build rebuild_tree)

`rebuild_tree()`:

(a) The array containing the dumped tree will be referred to as `tree_dump`. The length of this array will be `nbytes`. A stack of nodes will be needed to reconstruct the tree.

(b) Iterate over the contents `tree_dump` from 0 to `nbytes`.

(c) If the element of the array is an `'L'`, then the next element will be the symbol for the leaf node. Use that symbol to create a new node with `node_create()`. Push the created node onto the stack.

(d) If the element of the array is an `'I'`, then you have encountered an interior node. Pop the stack once to get the *right* child of the interior node, then pop again to get the *left* child of the interior node. Note: the pop order *is important*. Join the left and right nodes with `node_join()` and push the joined parent node on the stack.

(e) There will be one node left in the stack after you finish iterating over the contents `tree_dump`. This node is the root of the Huffman tree.

5) Read_bit, for more detailed check pdf specifics for the decoder for part 4

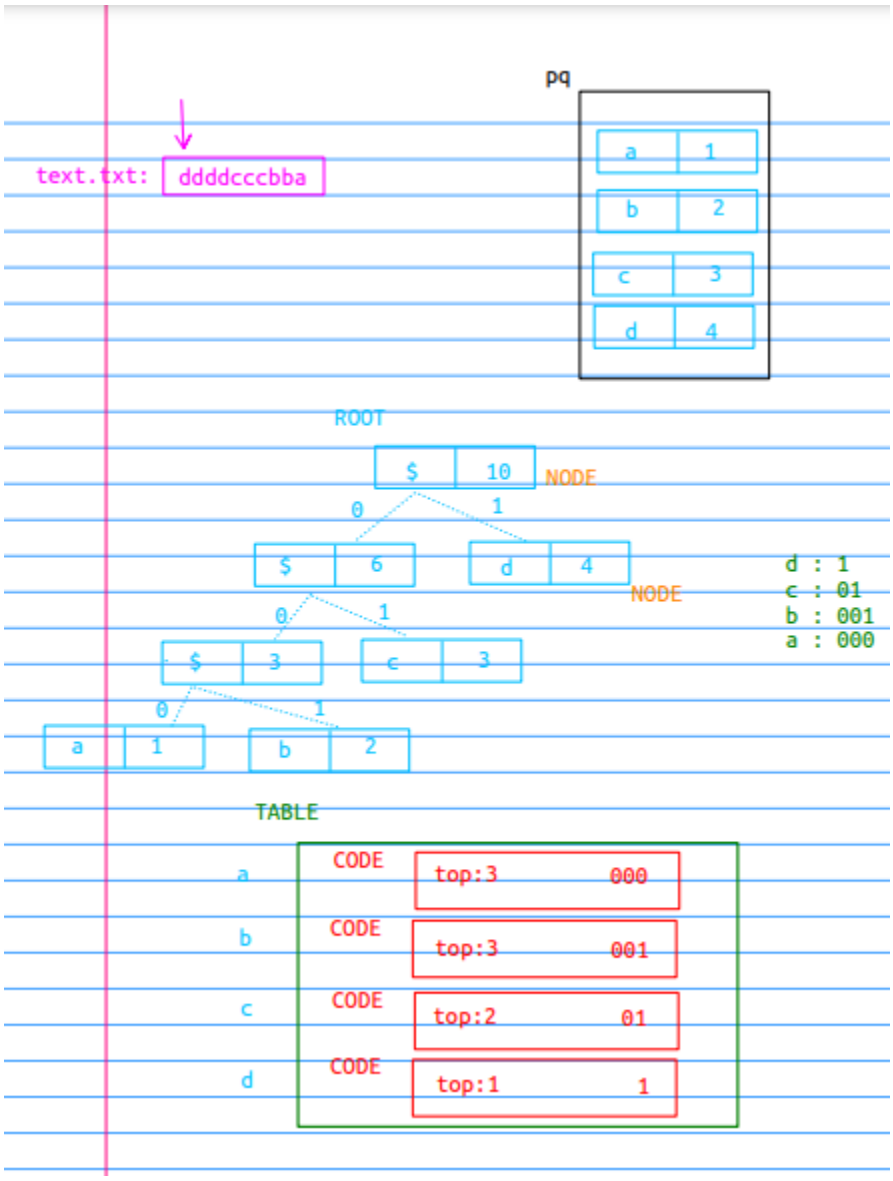
(a) Begin at the root of the Huffman tree. If a bit of value 0 is read, then walk down to the left child of the current node. Else, if a bit of value 1 is read, then walk down to the right child of the current node.

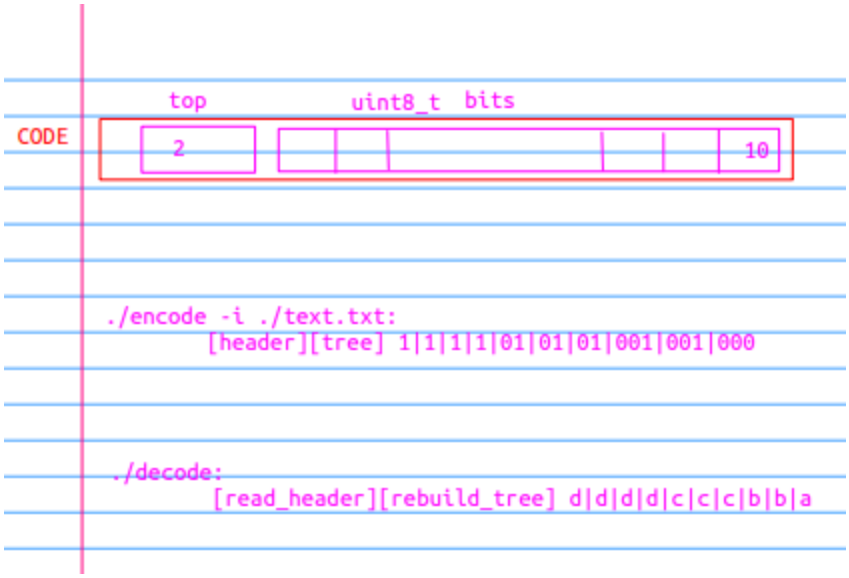
(b) If you find yourself at a leaf node, then write the leaf node's symbol to `outfile`. Note: you may alternatively buffer these symbols and write out the buffer whenever it is filled (this will be more efficient). After writing the symbol, reset the current node back to the root of the tree.

(c) Repeat until the number of decoded symbols matches the original file size, which is given by the `file_size` field in the header that was read from `infile`.

6) Close files

Visual concept:





Some extra stuffs:
Unique symbol = aaaabbbc = 3

Pseudocode:
Stack.c: Same as previous assignments
Code.c:
Code_init:
Code c
C.top = 0
For loop i<maxcodesize,
C.bits[i] = 0
Return c

Code_size: return c->top
Code empty: check c-> top ==0?
Code full: check if c->top == max codesize

Code push bit:
Bit <<= top %8
c->top /8 or bit;
Top++

Code pop bit:
Top --
location = c->top / 8
position = c->top % 8
bit = c->bits[location] & (1 << position)

bits[location] &= (~(1 << position))

Node.c:
Node create: initial everything
Node *n = malloc size of node
n->left = null n->right =null
n-> symbol and n-> frequency = symbol and frequency

Node delete:
Node delete left and right
 Free *n
*n = null

Node join:
Node *parent = node_create($, left frequency + right frequency)
Parent -> left = left
Parent -> right = right

Huffman.c:
build_tree(uint64_t hist[static ALPHABET]):
Pq create
For loop i<256, i++{
When histogram[buff]>0:
Node create
 Enqueue
}
Node * left and * right
Node *joined
 while(size>1)
        Dequeue left node
        Dequeue right node
        Node join(left and right)
        Enqueue join node
Dequeue the root
Delete pq
Return root

Build_codes: code init,
if left and right == null, table[symbol]=c
Else
        code_push(&c, 0)
        build_codes(left, table)

```
        Code_pop
        code_push(&c, 1)
        build_codes(right, table)
        code_pop()

Rebuild_tree:
Stack_create
for(uint16_t i = 0; i<nbytes; i++){
If tree[i] == 'L'

        i++
        stack_push(s,node_create(tree[i],0));
else if tree[i] == I
        Node *right;
        stack_pop(s,&right)
        Node *left
        stack_pop(s,&left)
        Node *parent = node_join(left,right);
        stack_push(s,parent)

Delete tree: just call node_delete
```

Pq.c:

```
Pq create:
Create pointer using malloc
Head tail size and capacity set to default
Items = calloc

Pq delete: free pointer and set it to null
Pq_empty: check pq size == 0?
Pq_full: size == capacity?
Pqsize: return size

Enqueue:
Find the correct position for the node first,
 And then shift everything from i to tail back
 Add node to queue at index i

Dequeue:
*n = q->items[q->head]
   q->head = (q->head + 1) % q->capacity
   size--
```

Io.c:
Read_bit:
Keep track of the end of the buffer
Bytes = read_bytes
If bytes!=block
    End of buffer = bytes *8
*bit = get_bit(buf, bit_index)
Bit_index = (bit_index +1) % BLock * 8

Write_code:
For i< code_size
Getbit
If bit is 1 , set bit
If bit is 0, clear bit
Bit_index += 1
Check if bit_index == 8* Block
    write_bytes(outfile, buf, block)
    Bit_index = 0

Encode.c:
Command Line options
Create hist[256]
Loop through the file, save frequency for each character to hist[], for example: aa, then hist[a(97 for ascii)]=2

Hist[0]++ and hist[255]++ based on pdf
Node *root = buildtree(hist)

H.magic, h.permissions, fstat, p.tree_size, h.file_size, fchmod stuffs

Write bytes
Write tree: if root left and right == null, meaning L, otherwise I
lseek(infile, 0, 0)
While loop through file and write cod
Flush code
Verbose
Close files
Decode.c:
Command line options
Read the magic number
read_bytes(infile, header, size of header)

Permission stuffs(fstat, fchmod)

```
Read header, and rebuild tree
tree[h.treesize]
read_bytes(infile, tree, treesize)
Rebuild tree

while(symbol number < h.filesize){
Read bit(infile, &bit);
if(bit){
N = right
}
Else : n = right
if (n->left == NULL and n->right == NULL)
        Symbol_number++
        text[text_idx++] = symbol
        n = root
        if (text_idx == BLOCK)
            write_bytes(outfile, text, BLOCK)
            text_idx = 0
}
Writebytes to read flush code

Verbose stuffs

Delete tree
Close infile and outfile
```