**Purpose of the program:**

I believe it's about encoding and decoding but the key idea is entropy, measuring the amount of information in a set of symbols.

**The following are probably the main .c files where you utilize all the other files**

**The Encoder:**

The encoder will read in an input file, find the Huffman encoding of its contents, and use the encoding to compress the file. The code must support any combination of the following command line options:

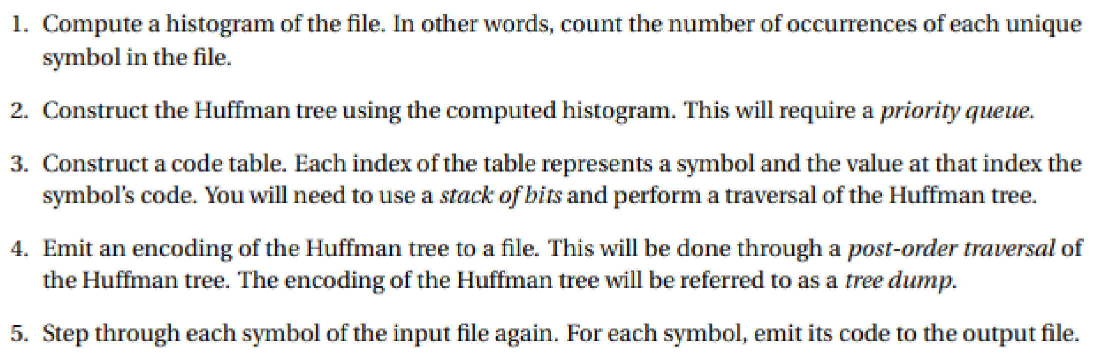
-h prints out help message

-i infile: specifies the input file to encode. The default input should be set as stdin

-o outfile: specified the output file to write the compressed input to. Default should be stdout

-v prints compression statistics to stderr. Stats are uncompressed file size, compressed file size, space saving (the formula is 100 * (1 - (compressed size/ uncompressed size))

Steps to the algorithm to encode a file



1. Compute a histogram of the file. In other words, count the number of occurrences of each unique symbol in the file.
2. Construct the Huffman tree using the computed histogram. This will require a *priority queue*.
3. Construct a code table. Each index of the table represents a symbol and the value at that index the symbol's code. You will need to use a *stack of bits* and perform a traversal of the Huffman tree.
4. Emit an encoding of the Huffman tree to a file. This 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*.
5. Step through each symbol of the input file again. For each symbol, emit its code to the output file.

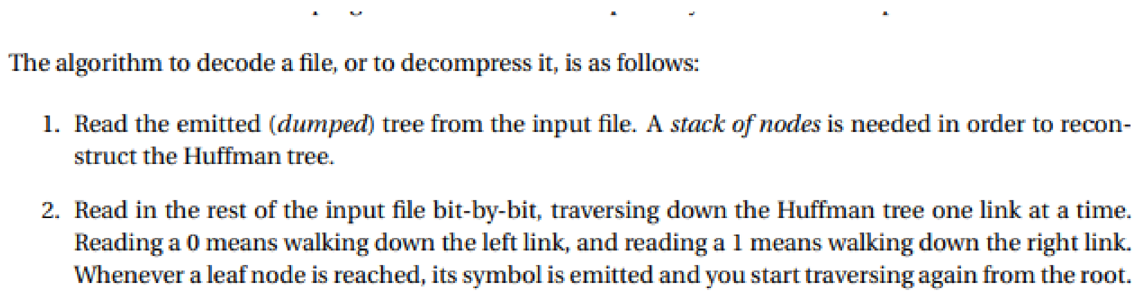
**The Decoder:**

The decoder will read in a compressed input file and decompress it, expanding it back to its original, uncompressed size.

Same command line options except for -v where space saving is

100* (1-(compressed size / decompressed size))

Steps to the algorithm to decode a file


The algorithm to decode a file, or to decompress it, is as follows:

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

**Nodes: ADT**

Huffman trees are composed of nodes, with each node containing a pointer to its left child, a pointer to its right child, a symbol, and the frequency of that symbol. The node's frequency **is only needed for the encoder**

Node create, node delete should be similar to the previous lab, so check that out

Node join is a little confusing, will return to this later

Node print - just a debug function

Priority Queues:

Encoder will make use of the *priority queue*. Functions like normal queue, but assigns each of its elements a priority, such that elements with a high priority are dequeued before elements with a low priority. Assuming that elements are enqueued at the tail and dequeued from the head, this implies that the enqueue() operation does not simply add the element at the tail. Dequeue() operation could *search* for the highest priority element each time but that is a bad idea.

How you implement priority queue is up to you but there are a couple of options

1) Mimicking an insertion sort when enqueueing a node, finding the correct position for the node and shifting everything back
2) Using min heap to serve as the priority queue. Why min heap? Because we dont want nodes with lower frequencies to be dequeued first. The lower the frequency of a node, the higher its priority. Your priority queue, no matter the implementation, must fulfill the interface that will be supplied to you in pq.h. Hint: check out heapsort implementation from asgn3

**Codes:**

After constructing a Huffman tree, you will need to maintain a stack of bits while traversing the tree in order to create a code for each symbol. This will be used in the future so nail this one

**Encoder pseudo:**

Openf to open a file and specify the type to read  or write etc
openf( user_file_input, “r”)

Fgets to get the contents of the first line of the file

Or use another function to read every line

(construct a histogram that should be a simple array of 256 (uint64_ts) im assuming this is the size limit)

Histogram[0] += 1
Histogram[255] += 1 (so now we atleast have 2 elements present)

Sscanf to convert type if necessary

buffer(Len - 1) for \n if necessary

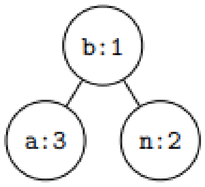
build_tree()

**Small Example of how the program works:**

coding the input `banana`. We will need to create a histogram of the input.

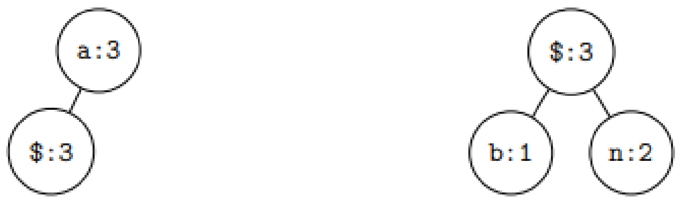
| Symbol | Frequency |
|---|---|
| `a` | 3 |
| `b` | 1 |
| `n` | 2 |

Nodes corresponding to each symbol are then placed into a priority queue. The lower the frequency of the symbol, the higher the priority of its corresponding node. The priority queue will be visually represented using a *min heap*. The priority queue is used to construct the Huffman tree, which will be shown alongside the priority queue. The Huffman tree is initially empty.



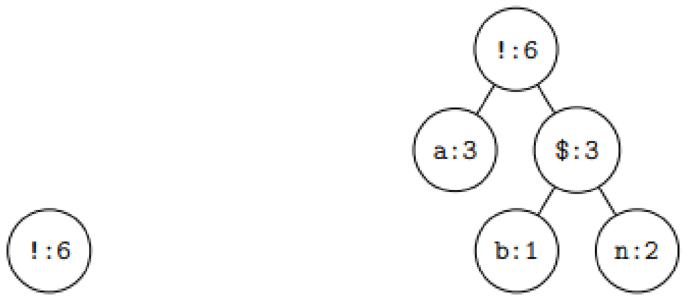
(a) Priority queue

(b) Huffman tree



(a) Priority queue (b) Huffman tree

We repeat the same step of dequeuing two nodes and joining them together. We will give the new
e the symbol !, just for distinction. The new node is then enqueued.



(a) Priority queue (b) Huffman tree

**Parsing-command-Line pseudocode:**

-h: prints out a help message
-v : Print compression statistics to stderr.
-i <input> : Specify input file to compress (stdin by default)
-o <output> : Specify output file to write the compressed input to(stdout by default)

```
Take in command line from the user
Get_h = false
Get_v= false
```

Get_i = false
Get_o = false

Case ‘h’: get_h = true
Case ‘v’: get_v = true
Case ‘i’: get_i = true
Case ‘o’: get_o = true

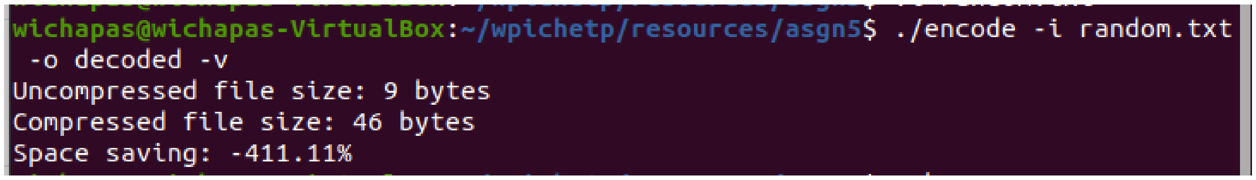
In the case of get_h being true…. Print a help message

In the case of get_v being true…. Print out the statistics collected from io.c bytes written and bytes read

In the case of get_i being true… take in the name of the file

In the case of get_o being true… outfile to the name of the file

Here is an example:



```
wichapas@wichapas-VirtualBox:~/wpichetp/resources/asgn5$ ./encode -i random.txt
 -o decoded -v
Uncompressed file size: 9 bytes
Compressed file size: 46 bytes
Space saving: -411.11%
```

For Stacks… Pretty similar to the previous assignment, just change a couple of function arguments

PRIORITY QUEUE UPDATE:

I ended up using a min heap for my priority queue. A couple things I changed are conditions for min_child and fix_heap. I had problems with those functions for a very long time because they were causing segfaults when I ran my encode and decode. The bug was caused by a single > sign in the fix_heap. Also, I was struggling with only dequeue because I believe I was having trouble with the index. So I drew that out on my Ipad and figured out that q->tail was counting wrong so I fixed that by moving the order of when I minus the size to after the operation of swapping the head and tail and cutting the tail off.