Design Document

Description

The purpose of the first program is to implement a Huffman encoder. Given an input file, this encoder will construct a histogram of the file, construct a Huffman tree from the histogram using a priority queue, construct a code table that maps each symbol in the histogram to a value, and finally encode the input file and tree dump.

The priority queue will be implemented as a minheap such that the lowest frequency symbol will always be at the top of the priority queue. The Huffman tree will be implemented by recursively combining the two nodes with the highest priority in the priority queue. This will create one node in the Huffman tree that points to those two nodes. Repeat this until the priority queue has one node left. The encoding of the input file will be done by the code table, and the tree dump will be done by traversing the Huffman tree.

The purpose of the second program is to implement a Huffman decoder. Given a tree dump, we can easily recreate the Huffman tree. Whenever we encounter a new symbol, push it onto the stack. Whenever we encounter an interior node symbol, we combine the two nodes onto the stack into one node. Repeat this process for the whole tree dump.

To decode the input, iterate over the encoded input. Start from the root of the Huffman tree. Go down the left child if the bit is 0 and right child if the bit is 1. When you reach a leaf node, add it to the decoded input. Repeat this process for the encoded input until every character has been decoded.

Structure

This section contains the pseudocode for the Huffman Encoder and Decoder.

Huffman Encoder

```
// Create the histogram
int histogram[256] = {0,0,0...}
read infile
for c in infile:
      histogram[c]++
histogram[0]++
histogram[255]++

// Create a priority queue
priority_queue pq
for i in range (0,256):
      if histogram[i] > 0:
```

```
            enqueue (histogram[i], i) in pq

// Create a Huffman tree
while size of pq > 1:
      node1, node2 = pop 2 elements of pq
      node3 = (node1[0] + node2[0], new symbol)
      node3.children = node1 and node2
      enqueue node3 in pq

root = pq.pop
root.children = 2 orphan nodes in Huffman tree

// Assign binary codes to each character
// Credit to Darrell Long for this pseudocode
def build_table(node, table, code):
      if node has no children:
            table[node] = code
            return

      build_table(node->left, table, code + '0')
      build_table(node->right, table, code + '0')

build_table(root, empty table, " ")

// Output tree dump
// Credit to Darrell Long for this pseudocode
def tree_dump(node):
      if node has no children:
            print('L' + node.symbol)
            return
      if node.left exists:
            tree_dump(node.left)
      if node.right exists:
            tree_dump(node.right)

print('I')

// Encode input
read infile
for c in infile:
      c = table[c]
```

Huffman Decoder

```
// Reconstruct Huffman tree from tree dump
for c in tree_dump:
      if c is 'L':
            push next char onto stack
      else if c is 'I':
            node1, node2 = stack.pop(2)
            node3.children = node1 and node 2
            stack.push(node3)
      else:
            stderr: "Invalid tree dump"
root = stack.pop
root.children = 2 orphan nodes in Huffman tree

// Decoding encoded input
node N = root
for i in code:
      if i = 0:
            N = N.left
      else:
            N = N.right
      if N is at leaf node:
            print N.symbol
            N = root
```

Inputs and Outputs

- Inputs will be read from a file if a file is provided by the user. If not, the input will be from stdin.

```
FILE *infile = stdin;
if(file provided) {
      *infile = fopen("example.txt", "r");
}
```

- Outputs work similarly. If a file is provided, then the output will go there. If not, then the output will be sent to stdout.

```
FILE *outfile = stdout;
if(file provided) {
      *outfile = fopen("output.txt", "w");
}
```

- The input file will be read using read_bytes().

```
uint8_t buf[1024], *input;
FILE *infile;
int nbytes = 1024;
while read_bytes(infile, buf, nbytes) > 0:
      add buf to input
```

- The output will be written using write_bytes().

```
uint8_t buf[1024];
FILE *outfile;
int nbytes = 1024;
while nbytes > 0:
      int written = write_bytes(outfile, buf, nbytes);
      if written = 0:
            break;
      nbytes -= written;
```

This chart shows the expected inputs and outputs of this program.