Specification

Assignment 6. Huffman codes

Files needed:

Makefile: a make file to easily run the program by prompting make into the Linux terminal on our Ubuntu virtual machine.

README.md: describes how the program works and runs in addition to how to use the makefile

Encode.c:this file contains the encoded huffman codes

Decode.c:this file contains the decoded huffman codes

Entropy.c: like last assignment entropy.c will be used to test our codes level of entropy

defines.h: This file will contain the macro definitions used throughout the assignment.

header.h: This will contain thestruct definition for a file header.

node.h: This file will contain the node ADT interface. This file will be provided.

node.c: This file will contain your implementation of the node ADT.

pq.h: This file will contain the priority queue ADT interface. This file will be provided.

pq.c: This file will contain your implementation of the priority queue ADT. You Must Define your priority queue struct in this file.

code.h: This file will contain the code ADT interface. This file will be provided.

code.c: This file will contain your implementation of the code ADT.

io.h: This file will contain the I/O module interface. This file will be provided.

io.c: This file will contain your implementation of the I/O module.

stack.h: This file will contain the stack ADT interface. This file will be provided.

stack.c: This file will contain your implementation of the stack ADT. You must define your stack struct in this file.

huffman.h: This file will contain the Huffman coding module interface. This file will be provided.

huffman.c : This file will contain your implementation of the Huffman coding module interface.

Program Description:

The Huffman code is a data compression algorithm that performs a lossless compression and encodes (compresses) and decodes (uncompress) messages. The way that the compression works is through the construction of what are called huffman trees and the huffman trees utilize many ADT, including nodes, the stack, a priority queue, and a code. So essentially what is happening is everything gets divided into how many characters there are and how many occurrences each character has. Each value will be their own child node and will be connected to a parent node based on the smallest to largest. Then the parent nodes can form more parent nodes which will construct the huffman tree. Do note that you can have separate trees that combine into a bigger parent node if the sums of a particular node do not add up with the parent nodes values. When multiplying a node's bit amount with its frequency, we know how much total bit each child node takes, and it usually so happens to be that this will compress the data. For example if the letter a occurred 15 times and was at the value 011 bit it would hold a total value of 45 bits vs the 120 bits it takes to hold 15 characters (each taking up 1 byte). So in a way this week's assignment is very much the opposite of last week's assignment where we try to compress data instead of giving it extra parity bits. Though it seems challenging, it also sounds very rewarding and fascinating to work on this assignment, I look forward to what's ahead.

Pseudocode:

Encode.c

```
Post traversal()
  If node = null
  Put "L" into array
  Put symbol into array
  Return

  Post traversal left
  Post traversal right
  Put "I" into array


main()
optarg
   Switch h
     Print helper message
   Switch i
     Reads to specified binary infile
   Switch o
     Writes to specified binary outfile
   Switch v
     Prints out statistics of compression to stderr

Build histogram{}
  While i in alphabet
    Hist[i] += 1 if symbol occurs

Build tree()

Build codes of the tree for each symbol()

Set file permissions
Chmod() & fstat()

Create header
H.magic = magic
H.permission = st.mode
H.tree size = 3 * leaf nodes - 1
H file size = st.size

Write out header

Post traversal of "L" and "I"
Write out post traversal message

Lseek aka reset reading file from the start
For i in range file size
```

```
  Write code()

Close infile
Close outfile
```

Just like last assignment we will need to encode a message of some sort but this time we will need to compress the file into smaller bits. Therefore it will generate gibberish once more but gibberish means that it is working. Per usual as well we have a few command line arguments that it can take, those being, -h, -i, -o, and -v. -h per usual is a helper message, -i is reading in some infile, -o is writing to some outfile, and -v will show how much the file was compressed by. We can assume each character is 8 bits long so we can compare how many characters there are with this compression statistic to see how much the file was compressed by. Lot goes down in the encode that you can track through the pseudocode. We need to make a histogram, build the tree and codes, build the header, and build another post traversal and write everything properly out.

Decode.c

```
main()
optarg
   Switch h
     Print helper message
   Switch i
     Reads to specified binary infile
   Switch o
     Writes to specified binary outfile
   Switch v
     Prints out statistics of compression to stderr

Read in header

Rebuild tree

While i < eof
  Read in bit by bit and walk the tree
  Decode message

Close in
Close out
```

Decode is very similar to encode except it does the opposite thing. It will take that compressed infile that we had in encode.c and decompress it by going through the huffman tree. The huffman tree is how we can utilize the post order traversal like search to decode the original message. It will read bit by bit until it hits a leaf node and once it hits a leaf node it will know that it needs to access that symbol. We have officially decoded or decompressed the message.

Stack.c

```
Struct stack {
  Uint32_t top;
  Uint32_t capacity;
  Int64_t **items;
};
Stack create{
*create stack
}
Stack delete{
Deletes stack
}
Stack full{
*stack full = top of stack = capacity
}
Stack empty{
*Stack empty = top is at 0
}
Stack size{
Returns stack size
}
Stack push{
*push element to stack. LIFO
}
Stack pop{
*take last element in first out
}
Stack print{
Print elements in a stack
}
```

The stack is once again used and similar to the ones that we used from assignments 3 and 4. The stack will be used to push the nodes to the stack in order to determine the post order tree. The post order tree will go to the very left as possible then make its way around. This function will only be used in the rebuild tree where it will need to push on values and pop off nodes and join them back together. What's amazing about this is that it only uses that post traversal that we made in the encode to rebuild the tree. Such a smart program :D.

Node.c

```
Struct Node {
  Node *left
  Node *right
  Uint8_t symbol
  Uint8_t frequency
}
```

```
Node_create{
*creates node
}
Node_delete
*deletes node
}
Node_join{
Joins a left and right node to make a parent node
}
Node print{
Prints node
}
```

The node data type will be imperative to helping construct the actual tree for this assignment. Each node will be representing a character and the node_join function will help construct parent nodes denoted by '$' to help construct the overall tree. Once there is only one node left we can declare that as a root node. It is imperative that the nodes work in order for this lab to work because they are like the fundamental building blocks to the binary tree.

Pq.c

```
Struct insertion sort{
Emulate insertion sort
}

Pq create{
*creates priority queue
}
Pq_delete{
Deletes pq
}

Pq empty{
If pq is empty return true
}

Pq_full{
Returns true if pq is full
}

Pq_size{
Returns the tail or head depending on how the pq works
}

Pq_enqueue{
Enqueues the pq
}
```

```
Pq_dequeues{
Dequeues the pq by priority
}

Pq_print{
Prints pq
}
```

The priority queue is another thing that will help us know how to construct the huffman tree. It will order the things in the queue so then when we begin constructing the tree, it will take 2 nodes at a time and join the correct ones together. The priority queue is meant to keep the tree construction organized and follow the rules of the huffman coding algorithm. The higher the frequency the less the priority so if it occurs more the code should be smaller. Example: a occurs once and has a code 1010 while b occurs 5 times and has a code 11. Though the pq doesn't actually manage the codes this relationship that they share will make more sense holistically.

Code.c

```
Code_init{
Constructor function for the code adt
}

Code_size{
Returns code size which is how many bits are pushed onto the code
}

code_empty {
Returns true if code is empty
}

Code_full{
Returns true if code is full
}

Code_push_bit{
Pushes bit so it can be properly decoded
}
Code_pop_bit{
Pops a bit off the code after being processed
}

Code_print{
Debugging print statement to make sure everything works
}
```

I mentioned codes earlier but this ADT is meant to construct the codes using the huffman tree. Because of these codes this is what makes it an actual compression algorithm. Ascii character a is usually denoted as 61 in hex which is denoted as 0110 0001. This ascii character contains 8 total bits but thanks to these codes we can compress it to something less than 8 bits.

I/O.c

```
read_bytes{
Reads bytes
}
Write_bytes{
Writes bytes
}
Read_bit{
Reads bits
}
Write_code{
Writes the code
}
Flush_code{
Flushes the code
}
```

In this assignment we have to construct an i/o which uses only open() close() and read() write(). This will read in a byte of data which will then read in bits and construct the code. It will also flush codes once done. These functions will be frequently called in our encode and decode modules as this is how we will be reading in from an infile and writing out to an outfile. If these do not work, there is a good chance no information or messages will be communicated between the encoder and decoder so this is just as important as everything else in the lab.

Huffman.c

```
Build tree{
Constructs the huffman tree
}
Build_code{
Builds the code for each character
}
Rebuild_tree{
Reconstructs tree in post order tree
}
Delete_tree{
Deconstructs tree function
}
```

And finally we have the tree constructor which builds the huffman tree. Everything I described above will allow us to construct the tree. Utilizing nodes, codes and more this will be the tree or the map to help

compress rather than decode the encoded messages. In the build tree it will simply use the priority queue and make the tree until there is only the root node left in which it will dequeue the final item and be returned as the root node. Then we have the build codes which will traverse the tree in a post order traversal and it will store codes once it hits a node. The codes table only stores codes for leaf nodes and leaf nodes only. Then we have rebuild codes which will be used in the decode module which will check whether it is a “L” or an “I” and determine when to make a new node and join them together. This will be imperative in the sense that this tree or map will be decoding the whole message based on its construction

Totality

Overall this lab sounds extremely cool if I get it to work by Sunday. I am doing my best but this has by far been one of the hardest assignments of my life. Back to work I go.