Assignment 5: Huffman Coding

Brandon Chuang

10/26/2021

A general overview of how the Huffman encoder/decoder (for Assignment 5) functions.

1 Introduction

In this assignment we will be making a huffman encoder/decoder. There will be 2 executable binaries, "encoder" and "decoder", both with their own inputs and outputs. "encoder" will take a file with raw text as input and will output compressed data with the huffman tree (explained later), while "decoder" will take a file with a huffman tree and compressed text and will output the uncompressed text.

2 Huffman Encoding

The idea of Huffman encoding is that there is a way to compress text by making characters vary by the amount of bits that represents them and making the most frequently used characters be represented by the smallest amount of bits. Contrast this with uncompressed text, where each character takes up 8 bits of data.
The biggest problem with doing something like this is that, since computers can only read 1's and 0's, how do we know where to stop reading bits and represents a character? This will be explained later on with huffman trees.
Although some characters may take up more than 8 bits in the compressed data, since they are less frequently used it ends up being more efficient than raw text because of the frequently used characters taking up less bits. When calculating the amount of data we managed to save through compression it is important to also include the size of the huffman tree, which will be required to have to decompress the data.

3 Data Structures

There will be several custom (and common) data structures required to be made for this program that are not provided in C.

3.1 Nodes

Nodes will be used to make the huffman tree. Because of the nature of this assignment, we will need to base the stacks and priority queue around. of our data structures around nodes rather than other data types (such as integers, characters, or booleans). All the node consists of is:

- a pointer to its child node to the left (if it has one).
- a pointer to its child node to the right (if it has one).
- the ASCII character it represents.
- an integer representing the amount of times the ASCII character showed up in the text.

The reason why we know nodes will only have two children is because huffman trees are binary trees.
Nodes will have associated functions to make, delete, and join them. The process of joining two nodes is just setting the pointer of the parent node to whichever direction you make the child node take (left if zero and right if one).

3.2 Stacks

Stacks are just a list of things (in this case, nodes) that are structured in a way so that you can only observe values that were put in last in the stack, and you can only remove the values that were last put in the stack. Think of it like a stack of books, where you can only read the cover of the book at the very top of the stack. To get to the books/values that were put first into the stack, you need to remove the values at the top (which were put in last). This will be used to reconstruct the huffman tree in "decoder".
This data structure is comprised of:

- a value that tells you what the next empty space of the stack is, which will help you decide what positions you have to remove values from and how many values are in the stack.
- a value that tells you how many items are allowed in the stack so you don't go over the amount of slots available.
- an array of pointers to nodes.

Stacks will have several functions associated with it to peek, push, and pop values, as well as return the size of the stack, make, and delete stacks.

3.3 I/O (Bit Manipulation)

This will be used to both read and write bits. It is important to do so since our text is not going to be uniformly sized, meaning conventional was of navigating text are not going to be applicable here. The data structure is basically a stack of 1's and 0's, comprising of an array of bits and an integer that represents the top of the stack of bits.

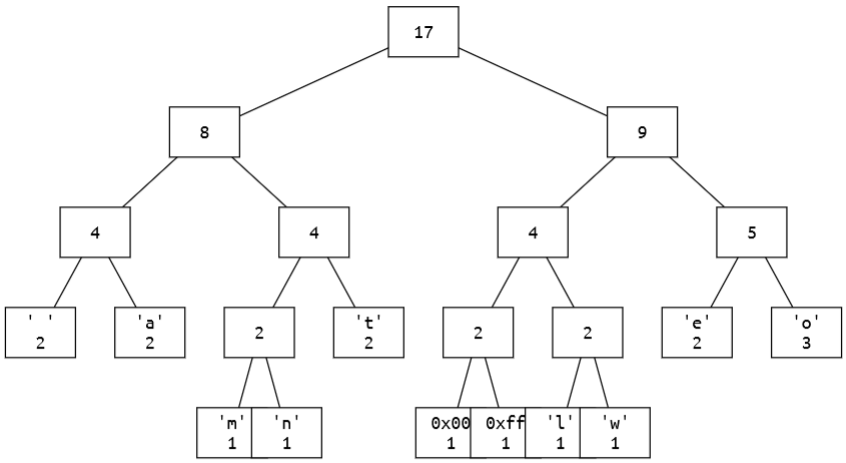
3.4 Codes

This is basically just a stack of bits. The reason why this is required (rather than making a special kind of stack for it) is because when encoding something with huffman's encoder, each character takes up a different amount of bits, so reading 9 bits in a row will not give us characters like the uncompressed text. Some special functions are required to fully utilize this "data structure":

3.5 Huffman Tree

The huffman tree is comprised of the node data structures. The huffman tree is a binary tree, meaning each node can only have two children. This is used to both encode and decode the raw and compressed text, and is just a tree with only the roots (nodes with no children) having a character associated with it. Constructing it will be explained later in 4.2: Encoding. In the figure below, you can see that the nodes with children do not have any letter associated with them. This is intentional, and a core part of its functionality (0xff and 0x00 are start and end string characters respectively). The text used in all the example huffman trees will be "one two oatmeal"

Figure 1: Credit: Ben Grant



4 encoder

encode will take the text from a file and encode it with huffman coding, leaving the output file with a huffman tree and the encoded text.

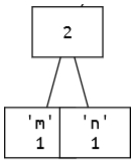
4.1 Input

- -h: prints out the help statement.
- -i: specifies the infile, the default infile is stdin.
- -o: specifies the outfile, the default outfile is stdout.
- -v: if called, decoder will print out the compression statistics.

4.2 Encoding

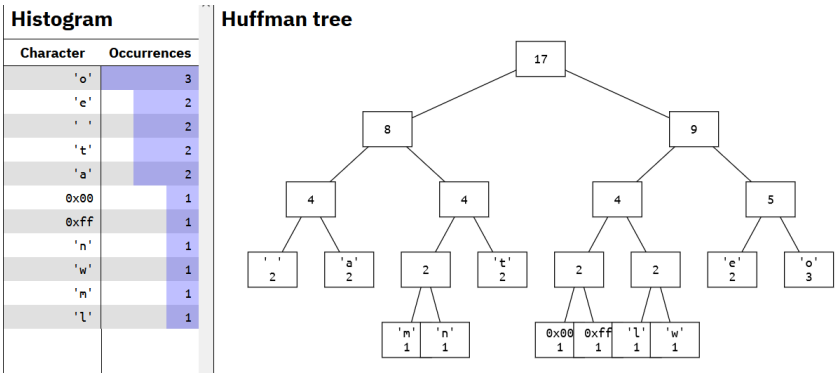
This will be the the biggest part of the assignment, and requires the most complexity. The first step would be to create a 256 long integer array. Each element in the array will represent the frequency of an ASCII character in the inputted text, which there are 256 of. We then create a node for each character that is used at least once.
The next step is to use a modified min heap sort to give us a priority queue, where the characters with the smallest frequencies have the highest priority. our heap sort should push the two nodes with the smallest frequencies and join them together as children to a parent create a node that does not represent a character and has a "frequency" of both the children nodes. This newly created node is also put back into the priority queue as a pointer, so that we can give it a sibling in the tree.

Figure 2: Credit: Ben Grant



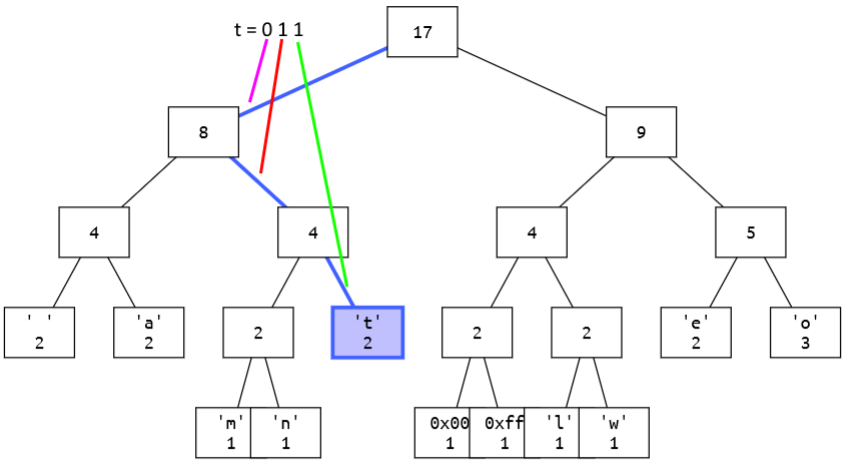
This is continued until we only have one node left in the priority queue, which shouldn't represent any character and should have a "frequency" of the length in character of the entire file's contents, and goes at the very top of the tree, connecting all the nodes together at the top. For example, this is a huffman tree for the text "one two oatmeal".

Figure 3: Credit: Ben Grant



Once we have this tree, we will loop through every character in the input file and use depth first search to find its location on the tree. Once we find its location, we record a bit of 0 whenever we travelled through the left node, and a bit of 1 whenever we travelled the right node. Once the node with the corresponding character is found, you move on to the next character and start back up at the tree.

Figure 4: Credit: Ben Grant



4.3 Output

The output is dumped into the specified file (or stdout if not specified). The first part of the output will be the tree.

5 decoder

Decoder will simply take the dumped huffman tree and the encoded text, rebuild the huffman tree, and decode the text.

5.1 Input

The inputs are the same a encoder.

5.2 Decoding

This will do the reverse of encoder: it will take the tree and compressed data and decompress it into raw text. The method of decompressing is actually quite easy, but require us to rebuild the tree first. As stated before, every time we go down the We will take a look at the "one two oatmeal" example from before. As explained in encoding, the letter "t" would be encoded a 011. Since each 0 represents a movement left on the tree and each 1 represents a move right on the tree, it is not hard to follow the tree from top to bottom to locate it. This is where the I/O and Codes data structures will help a lot with reading through each bit. It stops reading once it gets 0x00 as a value (indicating the end of the string/file).
The beauty of this encoding method is how, despite all the different storage sizes of each character, there will never be any problems with going over or under the amount of bits to read characters, since each character in the tree also happens to be a dead end with no children.
So basically, the strategy would be to essentially loop through each bit in the file (that's part of the code dump). This will be done with I/O's read bit function.

5.3 Output

The output is just the decoded text written in the correct file. If -v is input, it will also print out the compression statistics. The way it is calculated is:

$$100(1 - (x/y))$$

where x is the compressed size and y is the decompressed size.

Figure 5: Credit: Ben Grant