Brian Aguirre
ba5bx
postlab10.pdf
Post Lab

Implementation:

For the encoding portion of the lab, I built a heap class that would then create a vector of huffman trees. These data structures were stored each in their own class, meaning I started with the huffmanNode, which was then used to create a huffmanTree (as it is composed of nodes) and then these trees were then implemented under the heap class. The way I devised the information to be stored was so that each node held a value, this value was the letter of type char, a frequency, which was the number of times the letter appeared within the input text, and the left and right nodes. As it goes through the text and the nodes are identified, the heap inserts the min values in order to eventually achieve the standard heap data structure. Once the heap has been formed, I also included printing methods within the tree, which as long as there are nodes not yet visited (as it traverses the data structures), it prints out the values and frequencies of each of the unique characters.

Since I knew the max number of values that each character could be I used an int of size 128 to store the 128 possible characters. The position would then correlated to this value of 128, which originates from the ASCII letters. The iteration of the list (array) turns out being a linear one since despite the huffman tree having a triangular shape of a tree, when matching the letters, as it is going through the array, it will have to go one by one. The matching of the letters, which is the value of the nodes to that of the tree and setting the codes to the ones within the nodes ends up being theta(n), n being the size of the file (number of letters). The decoding portion is somewhat similar, meaning based on the implementation of the encoding portion of the lab, since a tree is formed containing the coded values for the letters, a matching tree which then goes through the nodes can be formed. In order words, a space holder tree can be formed and then filled with all of the information of the encoded.

Time and Space:

Through this lab, there are a couple of different running times encountered. When starting within the array, in order to place things in and run through it and in order to see the number of items in it, we see a Theta(n) running time. This is due to the fact that we have to

loop through it one by one, thus it's linear. Then as we insert things into the heap, which then form trees, we see a Theta(log(n)) time. The Huffman trees contains several methods, some of which get a Theta(log(n)) time as well, such as deleting, inserting, finding. Taking into account other methods within the heap data structure, we eventually come to see that the running time is Theta(n^4log(n)), since we have the linearity of the array, printing methods, setting the values, and then logarithmic running time of trees. (Running times looked up and retrieved from PDR slides within github repo of Aaron Bloomfield).

Though this running doesn't seem too bad, or too efficient, the actual average performance of heaps ends up being much better. This is probably due to the number of actual unique characters being much shorter and there only being so much need to go through the array.

In terms of space needed, the encoding portion logically takes much more data. The creation of the nodes for the letters takes a big portion of it since each node needs a frequency value (4 bytes), a letter (1 byte), and a left and right node (8 byte), so the least it requires is 13 bytes. This obviously does not include how much the value actually needs which is then read in order to decode it. The Array is 128 spots and each is 4 bytes, so in total that's 512 bytes at least. Then the heap is at least 1400 bytes, bringing the total to at least 1800 bytes. This is much less for the decoding since there is no need to form as many nodes nor keep the array, which took the most. There really is only need for the root node, which is as we said about 13 bytes, and a string which is used to read the values for the nodes.