Alex Yang (ahy3nz)

Postlab10

For reading in the input files, I created a hashtable. The keys would be the hashvalues of each character, and the values associated with the keys would be the frequencies. I chose to use a hashtable because it was a fast data structure in terms of storing and accessing data with a theta value of constant time.

After reading in the data structures, I created huffman nodes that contained fields for the character and the frequency. Because I implemented a hashtable, accessing this data was fast. I stored all the huffman nodes in a vector of huffman node pointers as a way to collect all the pointers and to facilitate creating the heap.

For the heap, I used a priority queue that implemented methods such as percolate up and percolate down. Choosing a min heap would facilitate making the huffman tree because the nodes in the heap would already be in order. This ordering would facilitate making a huffman tree, which will be described later. Constructing the heap was done by passing in the vector of huffman nodes. Making the heap involved percolating nodes up and down. These swaps had theta running time because a node that was just added may have to be percolated all the way up to the root node, so it would take almost n swaps.

Constructing the huffman tree had a running time of n. Upon each iteration, we are removing two elements from the priority queue and adding the combination into the priority queue. Furthermore, combining huffman nodes entails percolating half of the nodes downward. However, since the priority queue already had the nodes sorted, percolating had no effect on the running time. This simplifies to a linear operation of putting the first two elements of the priority queue together.

Generating codes for the huffman tree took n*log(n) running time. To reach one node would take, on average, log(n) time due to the number of comparisons to traverse tree.To find n nodes, it would take n*log(n) time.

Decoding the huffman tree can be broken down into a few steps - reading the input file, creating the tree, and decoding the text. Reading an input file is a matter of reading in each character, which is big O of n.

Creating the tree involves iterating through each character in a prefix code and constructing the tree accordingly. Each character corresponds to traversing through the tree once. With n characters, we are iterating through the tree n times, so the big O running time is N Decoding the text means reading a string bit by bit. Each bit corresponds to a particular traversal (go left or right), so decoding is also O of n. Overall, decompression is linear time.

I created the hash table to contain 500 spaces, including padding. Since each element in the hash table is an integer for the frequency, the amount of space occupied by the hash table is 500*4 bytes. However, ASCII has only 128 characters, so only 128 spaces in the hash table were actually necessary.

Each huffman node has a character and an associated frequency. The character is a char that is 1 byte, and the frequency is a 4-bit integer. The priority queue contains n huffman nodes, where n is the number of unique characters, so the space it occupies is 5*n bytes (each huffman node occupies 5 bytes of memory). The priority queue will contain a node for each unique character - duplicate characters are neglected since the huffman node will record the frequency. In a huffman tree, we also have internal nodes that don’t actually contain a character or a frequency. However, only the leaves will have real characters. In a balanced, complete huffman tree, there are n leaves for each character. Furthermore, there will n-1 internal nodes. In total, there will be 2n-1 nodes in the huffman tree. Since each node occupies 5 bytes (frequency integer and character char), the huffman tree will occupy 5(2n-1) bytes of memory, or 10n-5 bytes. Again, only unique characters are stored in the huffman tree.