Justin Choi (jc8mc)
CS 2150 -106
April 21st 2017

postlab10.pdf

In Lab 10: Huffman Coding, we both compressed and decompressed text files. In Huffman Coding, it uses the frequencies of symbols in a string to build a prefix code, assuming that no code in our encoding is a prefix of another code. The more frequent a character, the fewer bits are used to represent it.

In order to implement the Huffman encoding, I used the given heap data structure that uses a vector under the hood and a tree data structure. With the given heap data structure, I modified it so that each node can hold a Huffman node that I created so that a Huffman node that holds the character, code, and frequency with left and right pointers so that it can holds a child, if needed. As data is read from the file or modified, the heap is able to add (insert) or delete the minimum (deleteMin) value when necessary. As each node holds the specific values, the makeHuffmanTree method in my Huffman Tree implementation creates a Huffman tree based on the two nodes that has the lowest frequencies repeatedly, remove these two nodes, and make them children of a new node. In the end, our goal for the Huffman tree is to create a tree that holds nodes with the highest frequency towards the top so that the most used characters are taken less time to encode or decode.

I chose these data structures since the vector provides an overall order and mapping of where the elements are located. And the heap can be shown in a vector with the 0th element not used. As vectors allow us to access the most recently used element in order to properly encode its bit value. Later in constructing a tree, I used the overall general tree, but in this case a specific Huffman tree, since we want to have the character with the highest frequency to be at the top so it holds this order and structure property. Very similarly, the decoding used a huffmanNode and used a tree to take those nodes in order to decode the file from the information converted from the tree.

Justin Choi (jc8mc)
CS 2150 -106
April 21st 2017

For time and complexity analysis, there are two parts: encoding and decoding. For encoding, as we start with an array of ints with size 128, we move through the data structure in linear time, which yields a running time of Big Theta (n) since in order to find a value, it requires to search through each and every element, step by step, in the array to get to the targeted frequency rate for the character, worst case. As we read the message from the file, creating a tree requires operations such as insert() and removeMin() since each character has different frequency rates. In order to correctly make the Huffman tree, so that it holds nodes with the highest frequency towards the top so that the most used characters are taken less time to encode or decode, inserting each character and then percolating it based on its frequency has a time complexity of Big Theta (log n), which is the same for trees. As we also can call printPrefix, it needs to iterate through all the nodes that the tree has so it is Big-Theta (n), linear time no matter how balanced or how well the Huffman tree was created. Also, as we are using a vector as an underlying data structure in the heap, we could be causing a linear time.

For encoding, in order to find the worst-case space complexity, I used the sizeof() method. Node = 4 bytes (frequency, int) + 1 byte (char) + 8 bytes (left and right pointers) = 13 bytes. Frequency array of ints = 128 * 4 bytes = 512 bytes. Heap = 100 (as vector of size 100) * 13 (from node) = 1300 bytes. So in total, 1812 bytes were used.

For decoding, in order to calculate the time complexity it takes to read in a file and create a Huffman tree. Since we linearly read in a file causing linear time until we hit the end of the file causing Big-Theta (n). We also have a running time for the creation of the Huffman tree, causing Big-Theta (n), linear time. Also for printing the Huffman tree it has time complexity of Big-Theta (n), linear time. For decoding, in order to find the worst-case space complexity, left node = 13 bytes as well as right node = 13 bytes. The C-String for ASCII array of characters = 256 bytes. So, a total space of 282 bytes is required.