Samreen Azam (sa3tnc)
11/29/19
postlab10.pdf

To implement the Huffman coding tree, I first utilized a heap for encoding. I started by writing a class called HuffNode, which would create data nodes to store in the heap via a vector. These nodes contained information for their character value, how frequently the value should appear in the message, a prefix code, and pointers to both the left and right nodes. Using a vector to store them was optimal as their insertion and removal processes are more efficient than that of a linked list. Additionally, I needed to use a dynamic data structure for the heap; the amount of space needed to store all the nodes could not be determined beforehand, so an array would be insufficient. I also altered the binary heap code provided by Professor Bloomfield to suit a heap of HuffNode objects. Then, I used an unordered map structure to pair char values with a prefix code represented by strings. Maps require each value to have a unique key, which in turn allows for faster sorting and searching, so I felt that using this structure would be beneficial. I also included an iterator to read through the map and print relevant symbols. After that, I went on to build the actual Huffman tree using a while loop. Until the heap's size was equal to one, the two smallest values had their frequencies added to be stored as part of a new node and then were deleted. The new node was inserted into the heap after initializing the two minimums as its left and right pointers. I also made a recursive function in the Heap class called huffTree() to set the prefixes recursively by using a map. Following this, I created a program for decoding a message with a Huffman tree. I did not need the assistance of a heap for this, but I did use a map again to create the tree of HuffNode objects. Through loops and an iterator, I traversed through the map and checked if I should check a node's left child or right child based on whether its prefix was 0 or 1. Leaf nodes had their char values concatenated to a string.

Moreover, I was able to determine the time and space complexities for the encoding and decoding programs by testing different text files and checking the debugger. The runtime for storing the frequencies was always linear time because each character had to be read once. This also applies to the printing portion of the code as well as for decoding. When creating the Huffman tree, the worst case runtime was $\theta(\log n)$ since findMin() and deleteMin() were both called twice in the while loop. For the space complexity, the HuffNode class takes up at least 13 bytes: 1 byte for char val, 8 bytes for the pointers, and 4 bytes for int freq. The prefix string varies in size. The size of the vector varies by input size as well.