//Name:Shahroz Imtiaz
//Email ID:si6rf
//File Name: postlab10.pdf
//Date:11/29/2018

I used several different data structures in my Huffman implementation. To store each input and its frequency, I created a huffmanNode class. The nodes in this class contained the character inputted, their frequency, their prefix, and left/right pointers. I created this class because it was an easy way to keep the character, it's frequency, and its prefix in one place (the node). Then in order to create the heap that would be used in the program, I modified the binary heap that was provided to us in lecture by Professor Bloomfield. The first modification I made was changing the vector in the heap, so it was able to handle my Huffman nodes. Using a vector in my heap implementation made sense because in the program, we are constantly adding and looking up Huffman nodes. Because the runtime for looking up an item in a vector is O(1), it was definitely the best choice to use in order to ensure a faster program. I used a vector instead of an array, because the program has to be able to handle any size input and considering vectors grow by themselves and I couldn't possibly guess what the input size would be, a vector was the logical choice over an array. To break the process of insertion down for my heap, as Huffman nodes were inserted into my heap, my heap used the provided percolateUp and percolateDown methods in order to shift parent nodes to ensure that the findMin() method was always giving the runtime of O(1). I didn't create a separate class to make my Huffman tree. I would be manipulating my heap to create the Huffman tree, and I felt it was better and faster to just program a createHuffmanTree method in my heap class. The createHuffmanTree method took in a heap that was set up by my program beforehand and manipulated it by continuously combining the two lowest frequency nodes until only one node remained, which would be used as the root node for my Huffman tree. In my huffmanenc.cpp file, in order to keep track of each character and its frequency, I created an array of length 128. The reason why I chose 128 for the length is the ASCII value of the chars inputted would be never be greater than 127 as that in is the highest ASCII value for a char. Every time a char was inputted, I used its ASCII value to increment the frequency at its ASCII value index in the array. An array allowed me to do this in the runtime of 0(1) for each individual character and that's why I chose it, and since I knew the range of the ASCII values, an array made more sense to use than a vector.

When looking at the worst-case running time of the compression stage of my implementation, reading the source file and determining the frequencies of the characters in the file took O(n) as I had to look through each character and put it in its appropriate index in the array, increasing the frequency at the index by 1 each time. Storing the character frequencies in my heap took O(log(n)) as nodes in my heap had to be percolated up and at most only one swap took place on each level of the heap on the path from the inserted node to the min. Since a Huffman tree is a full binary tree, building the tree of prefix codes that determines the unique bit codes for each character always takes worst-case O(log(n)). Lastly, writing the prefix codes to the output file/output and setting them to their respective characters took O(n) as each character and its prefix has to be outputted and set.

When looking at the worst-case running time of the decompression stage of my implementation, reading in the prefix code structure from the compressed file took O(n) as each character and its prefix have to be read in. Re-creating the Huffman tree also took O((n)) as each character and its prefix have to be read. Lastly, reading in one bit at a time from the compressed file and moving through the prefix code tree until a leaf node is reached and outputting the character stored at the leaf node took O(n) as each character has to be accounted for.

When analyzing the worst-case space complexity of implementation for the compression stage, each Huffman node took at least 21 bytes of space: left/right pointer (8 bytes each so 16 total), int frequency (4 bytes), char character (1 byte), and string prefix (whose space depended upon the length of the prefix code). The heap I used was a vector whose size and space complexity grew and was determined by how many Huffman nodes were created and inserted. There's no way to provide an exact estimate for the space complexity as it all depended on how many unique characters were read in so all that can be said is that the heap took at least 0 bytes. Lastly, the array I used to keep track of character frequency had a size of 128 and stored ints (4 bytes), so it took 512 bytes.

When analyzing the worst-case space complexity of implementation for the decompression stage, each Huffman node took at least 21 bytes of space: left/right pointer (8 bytes each so 16 total), int frequency (4 bytes), char character (1 byte), and string prefix (whose space depended on the length of the prefix) and a string of all the bits comibned was created (whose space depended on the length of all the prefixes put together). Thus, all that can be said is at least 21 bytes of space were used (assuming that at least one character was inputted).