Joseph Elsisi jre3wjh
Lab 108
November 29
Compression:

The first step of the compression phase is to read in the file. I chose to use an unordered_map(hashtable) to store the frequencies of each character for two reasons. 1. Each key (char) would be distinct, I did not want any duplicates. 2. Fast insertion and find (theta(1)). With the linear reading of the file and the constant storage, the first step takes theta(n+1), where n is the number of characters in the file. This can be simplified to just theta(n). The unordered_map is initialized to 56 bytes. The standard lib uses a separate chaining implementation that adds 32 bytes for every new bucket created. In the worst case that every character is unique, the space complexity will be linear plus the constant 56 bytes, so theta(n).

With a full hashtable, I than create nodes from each key for my heap implementation. In the worst case that every character in the file was distinct, it would take theta(n) to iterate through the hash table, and theta(log(n)) (worst case) to insert each element, bringing the current program total to theta(nlog(n)). Each node is 24 bytes and is initialized upon a distinct character leading to linear space complexity.

Next the huffman tree is built, which consists of several constant time operations including deleteMin() for the heap, and initializing new huffman nodes/dereferencing pointers. In the worst case that every character is distinct, this will be a linear operation through each node in the tree. Finally, the new tree is inserted into the “forest” with a theta(log(n)) operation, making this operation theta(nlog(n)). In the worst case that the huffman tree is a perfect binary tree, and every character is distinct, then understanding that the tree will have (2^h) - 1 non-leaves, (n-1) new nodes will be created, making the space complexity theta(n).

I then have a helper function turn the huffman tree into the encoding needed for each character. This requires recursing through the binary tree and inserting the found prefix code into a hashtable. My helper function will step through each node, just like a print operation, and will require theta(n) time. In this case a constant space unordered_map is made, with a linear space node + pointer being initialized with every distinct character. This linear space complexity assumes the worst case that every element in the hash table will collide.

Finally, the text file is iterated through one more time, and a constant time find() from the unorderd_map is called on each character to replace it with it’s prefix code. This will take linear time.

Decompression:

The first step is to read in all the characters and their prefixes, each step of the way building the huffman tree. It will take linear time to get through all the characters, and log(n) time to insert each one into the tree. This step takes theta(nlog(n)) time. If n leaf nodes are initialized, then in the worst case that the huff tree is perfect binary, n - 1 dummy nodes are initialized. The space complexity is therefore theta(2n-1) which simplifies to theta(n).

Next, it takes linear time to read through all the encoded characters that are then stored into a string. This string will take up 1 byte for each character in the message, and one more byte for the ending terminator, so the space complexity is linear.

Next, the encoded string is read, with each bit representing a turn left or right in the huffman tree. With n being the number of characters in the message, each character will take log(n) time to be decoded. This is because each bit will specify a direction for finding a character, and since our implementation is a binary tree, half of the options will be cut off on each turn of direction. My implementation stores the characters into a string before outputting, which will take linear space for the number of characters in the message.