Akarsh Kumar
EID: ak39969
EE-360 Algorithms Assignment 2
Dr. Santacruz

Part 1.
Provide and Explain the complexity of the constructHuffmanTree method.

This method runs in O(k*log k) time and O(k) space complexity, where k is the number of characters in the character list.

This method initially creates a min heap (PriorityQueue) and adds all leaf nodes to that heap, taking O(k) time.
It then continually removes two min nodes and combines them (O(1) operation) and adds that node back to the queue until only one node remains. This takes O(3*log k) time per combination and combines O(k) times, resulting in a total of O(k*log k) time complexity.
It then traverses the entire tree from the root until it gets to the leaf nodes and assigns the leaf nodes on a encode and decode map (O(1) operation). This process takes O(2*k) in worst case time to go through the whole tree.

This method holds two HashMaps from characters to encodings the backwards and one HashMap from nodes to characters. These all take O(k) space.

Overall, the total after adding these three processes is O(k*log k) in time and O(k) in space.

Part 2.
Provide and Explain the complexity of the encode method.

This method runs in O(n) time and O(n*log k) space complexity, where n is the number of characters in the message to encode and k is the number of characters in the character list.

The runtime is O(n) because it simply goes through n characters and searches for each of them in the HashMap from character to encoding and appends this encoding to the overall encoding (appending is O(1) with StringBuilder).

This space complexity is O(n*log k) because each character requires ~log(k) 0s and 1s to encode with using a Huffman tree of k characters. Doing this for n characters results in O(n*log k) space complexity.

Part 3.

Provide and Explain the complexity of the decode method.

This method runs in O(m) time and O(m/log k) space complexity, where m is the number of 0s and 1s in the message to decode and k is the number of characters in the character list.

The runtime is O(m) because it simply goes through m 0s and 1s until it has a built a StringBuilder that the decoding HashMap contains as a key. Once this happens, it appends that character to the decoding and keeps on going. Since (appending is O(1) with StringBuilder), this loop runs through only the m encoded 0s and 1s.

This space complexity is O(m/log k) because each character requires ~log(k) 0s and 1s to encode with using a Huffman tree of k characters. If the encoded version is m long, the decode string will come out to be around m/log k characters.

Part 4.
Prove that your encoding is prefix-free.

The entire Huffman algorithm ensures that no encodings are prefixes of each other.
After creating the k nodes that represent the characters to encode, the construction of the tree is done by taking the two minimum frequency nodes and combining them with a common parent and putting that common parent back into the queue. This ensures that ALL initial k nodes WILL be leaf nodes, because no nodes are assigned children except from brand new empty nodes (nothing from the initial set). It also ensures that every node will have 0 or 2 children, because the leaves have zero children and all new nodes will have 2 children, because they are combining two existing nodes.

This property of all character nodes being leaf nodes results in the following statement.
No character node is an ancestor of another character node (ancestor meaning some parent, grandparent, etc.). This is true, because if it weren't, that means that a character node would have children (hence not being a leaf node).

So, since no character nodes are ancestors of any other character node, and each edge traversal means a certain encoding character, no character encoding starts with another character encoding, being prefix-free.

QED.