Project 2
Michael Watts

For the literal sizes of my files, the uncompressed file is smaller at approximately 406 kilobytes. The compressed files sits at 3100 kilobytes. This is due to the asci representations of the "bytes" for the compressed file taking up much more space than the literal characters they are meant to represent. This leads to a ratio of approximately 7.6 compressed file size to uncompressed. However, if the compressed files content is treated as actual bytes rather than asci characters, it sits at 300 kilobytes. This leads to a much smaller .7 ratio of compressed file size to uncompressed file.

In my file the top 5 letters were: ' ' with a frequency of 30,000 as it was present after every word, 'd' with a frequency of 12,687, 'i' with a frequency of 12,665, '.' with a frequency of 12,642, and 'h' with a frequency of 12,636. My 5 least frequent letters were: 'q' with a frequency of 12,291, 'w' with a frequency of 12,308, 'k' with a frequency of 12,329, 'e' with a frequency of 12,376, and 't' with a frequency of 12,433.

Rerunning my program with only the character pool: "abcdef" generates a new compressed and uncompressed file. The uncompressed file sits at about 406 kilobytes. The compressed file's literal size is 2200 kilobytes. This leads to a ratio of compressed to uncompressed file size of approximately 5.4. Taking the characters as literal bits leads to a compressed file size of approximately 195 kilobytes and a ratio of .5 of compressed file size to uncompressed.

Rerunning my program following the pattern of weighting each character based on its position in the alphabet gives me a literal compressed to uncompressed file size ratio of: 4.3 kilobytes and treating each character in the compressed file as a byte ratio of .52. This ratio is far better than the random string and approximately the same as the smaller character pool ratio.

The largest roadblock I encountered while developing this project was the min heap data structure. Adding a node to the data structure is not as simple as it is in most trees. In a binary search tree, one can start at the head and navigate logically down to the desired position for a data node based on comparisons. However, a min-heap is primarily based on a breadth addition, aka adding a node just to the very next spot open spot in the tree and then just bubbling it up. Because of this, one cannot start at the head and work down to the next open spot for a data node easily. To solve this problem, I used a data structure that already works by breadth addition as the back bone, an array. This meant I did not have to logically work out how to traverse the tree down, I could just push a new data node onto the array and bubble it up. With this data structure in place, the rest of the project just involved efficiently navigating the tree after forming and reforming it.