To work with the huff and unhuff algorithms, I created a header file containing the data structures to be used in the solutions. This includes a tree node, heap, list node, and list header. Each tree node contains a character, integer, and two tree node pointers. This allows each tree node to contain a specific character, its frequency, and links throughout the tree. The heap structure contains an array of tree nodes and an integer to store the size of the array. In the huff and unhuff algorithms huff utilizes the tree node, heap, list node, and list structures, while unhuff only uses the linked list and list header.

In huff, the first function that is called reads through the input file once and returns the text as a string. This string is then passed into a function that fills two arrays with the characters found and their frequencies. First, a temporary integer array of length 256 is created and filled with zeros. This is the maximum range of ASCII characters that are possible in the input file. Each character is read from the string of data and adds one to the corresponding index in the temporary array. Since the values of the ASCII characters range from -128 to 127, the index is found by adding 128 to the character value. Once the string has been read through, the frequency and character arrays are filled with the character values with frequencies above zero. This is to minimize the size of the huff file so that only characters that actually appear are given codes. The arrays are filled so that the same index matches a character with its frequency.

These arrays are used in another function to create a heap with a tree node array containing one node for each character in the array. Nodes are initialized with their character and frequency. The left and right pointers are initialized to null. The heap array is then sorted by frequency of each tree node. After the heap has been sorted, it is used to build a Huffman tree. Since the heap is sorted, nodes are added one at a time to the tree and after being added, a node is removed from the heap.

After building the tree, the codes can be assigned for each of the characters. This is done recursively from the returned tree root. From the root down, the function is called again for the left and right pointers until the pointer values are null, meaning the node is a leaf. Digits are stored in a string on each node. For each movement left, a ‘1’ is added and for each movement right, a ‘0’ is added. Once a leaf node is reached, the code is

completed. The codes are then stored in a linked list that can be easily traversed while reading through the string of file data.

The header written to the .huff file begins with the total number of characters to be written to the file. Then each character is written, followed by its code, and then a ‘:’ character to signal the end of the code. After the header is completed, the text file data that was read earlier is translated character by character into corresponding codes that are written into the new .huff file.

When the .huff file is unhuffed, the number of nodes is read first. A linked list header is created with the size initialized to the number of nodes. The first character is read until the ‘:’ character is found, and a list node is added to the list header with the code and character. This continues until the number of nodes has reached the value determined earlier.

After reading the header, the linked list is complete. The rest of the file is read by bits until one of the codes is matched. Then the corresponding character is written into a new .unhuff file. This continues until the entire .huff file has been read.

The same function was used to find the new filename in both the huff and unhuff algorithms. This function accepts the old file name and a string to append, then adds them together.

To gauge performance, I measured the run times of huff and unhuff for different file sizes. For the 100k.txt file provided, huff ran in 20 milliseconds and unhuff ran in 75 milliseconds. For 1m.txt, huff ran in 216 milliseconds and unhuff ran in 768 milliseconds. In general, the huff algorithm runs just over three times faster than unhuff.

The compression ratio of the .huff file compared to the .txt file was not ideal. For small files, the compression ratio is close to 1, but for large files the ratio is worse. For a 98kb input file, the .huff file is 489kb, for a compression ratio of almost 5. For any file larger than this, the compression ratio is around the same.