Project 2 – Alex Dunker – adunker@purdue.edu

My project is split up into four files, the huff.c and huff.h and the unhuff.c and unhuff.h. Starting in huff, the first thing we had to handle was allocating additional memory for concatenating the '.huff' string with the inputted file from Argv. A new location of memory is needed as the buffer for Argv is not large enough. Next, we scan the inputted file and get a frequency count and the total number of characters for the files. This works by looping through the file and getting each character, then incrementing the corresponding ASCII character. We created an array of integers for one of each ASCII characters so we can handle any kind of characters, numbers, letters, punctuation, and more. It then returns the total number of characters. Next we build a linked list of the characters we found. As we found characters which appear in the file we insert them into the linked list in ascending order. We call the List_insert_ascend function which will correctly place a node corresponding to a character and its frequencies based off its frequency. This function recursively moves through the linked list to find the correct position to move the new node into the correct position. Sorted from lowest frequency to highest frequency, excluding the zeroes. Next it builds a new linked list and nodes for a binary tree based of this previous linked list. It took every node and created binary tree node, still sorted based off the frequency. It next combines all of these nodes into a binary tree based off combining the two smallest frequency of and created a new parent node with a weight from the combined weight of the previous two, and then freeing the weighted list nodes. By the end, all of the weighted list nodes will be destroyed. Next we build a codebook as a two dimensional array. Using the leaf nodes it fills in the code book with the characters. From this it builds the tree header based off the binary tree. Going through the tree you write the header in using the bit representation of all of the files, and recursively calling itself through the tree. It then writes one more 0 to the end of the header as we use the 7 bit representation of ASCII characters, and writes the number of characters to the header and a new line character before jumping to compressing the full file. We use bit shifting to properly make this happen. To actually compress the file we loop through the input file and use the codebook to write the bit representation of the characters to the files. It uses the padZero function to finish off, and then the file is then successfully compressed.

On the unhuff side of things the first thing to handle is the inputted file, which we'll need to append an '.unhuff', so we have to allocate new memory to concatenate the strings. Then we call our decode function to begin the process. It starts with some error checking and then reads the header. It reads through the bits of the header and reverses the process that we did in the huff.c file. It creates the tree nodes for each character and then created the weighted node list. These functions are extremely similar to their corresponding functions in the huff.c files. Then using the reconstructed binary tree, it reads the bits and prints them out to the output file. The output file is opened as a normal 'w' open format to create a new fresh file. This should output a successfully unhuffed file. The new file has the ".unhuff" extension, and are a normal ASCII text file. Any associated helper functions go along with recursive calls where the original function is needed to establish, the proper recursion. During the process I did reference code in Lu's when I got stuck with the implementation of the code. The overall size of the project made it extremely difficult to debug, and especially when moving between the compression to decompression portion of the code.