ECE368
Project 2 Milestone 1
Chul Woo Park

In this project, I am going to implement the Huffman algorithm, including both compressing and decompressing. For the Huffman compressing, I'm assuming that the input file is a pure ASCII text file. For example, the input file would contain “go go gophers”, which is a normal text that we use everyday in a document. Firstly, during the process of creating the Huffman tree, I'm going to use two node structures, which are ListNode and TreeNode. I won't be directly reading from the input file and convert into a tree, but I would store the data in the input file into an array. This array will be used for a temporary container. Index number in this array will be the ASCII value and the value stored in such index will be the frequency. In that way, I would be able to access the information that which letters occur how many times in an input file. With that information contained in an array, it will actually go into a process of creating the Huffman tree. First of all, in order to implement the Huffman tree creation, I'm going to use three major method functions, which are addNode, buildTree, and addTree. In the beginning of the process, it will first convert the temporary container array into a linked list. This will be done with addNode function. It will go through that array and if the frequency other than 0 is found, it will call addNode function. In addition, addNode function will have a sorting algorithm so that the nodes in the linked list will be added and sorted in an order of ascending frequency at the same time. Once the linked list including the letters and frequency of each of them is successfully created, it will go into a process converting that linked list into a Huffman tree. In this process, buildTree function will be called recursively. What buildTree function does is getting the first two nodes in the linked list, which will be the two least frequent nodes, and creating a tree with them. Tree will have a head node and left node will be the first node in the linked list and right node will be the second node in the linked list. The reason is that the first node will definitely have a less frequency or less ascii value than the second node. Then, addTree function is going to be called because creating a tree with first two nodes in the linked list is also kind of creating a new node because it will have a frequency of sum of those two nodes. Therefore, the head node of such tree should go into the right place in the linked list. Right after that, buildTree function is going to be called at the end, which is a recursive function call. In this way, it is going to be executed until only one tree head node is left in the linked list. At this point, the Huffman tree will be succefully built. Then, I'll traverse this tree in order to print out the compressed output file. In addition, for decompression, I'm going to add a header at the begging of the output file. Header information will include the tree information. In order to do that, I'm going to do the pre-order traversal and print the result in the beginning of the output file. Therefore, in my decompression method, I will first read the header part of the input file and build a Huffman tree so that it can traverse the tree according to the input file. Decompressing will be basically reading the input file and finding the corresponding letter in the Huffman tree, which was given as a header information.