Matt Calayo mjc5gh 4/19/18 postlab10.pdf

Encoding

The first thing I did was create an array of size 128 to read in all of the possible character values. I read through the input file character by character and increased the count of the array based on the value of the character read. This worked by turning the character value to an int and using the int as the index for the array. This took linear time, based on n number of characters in the file as well as linear size. Insertion into the array was constant time because the array was the max number of psossible characters. Next I took each character with a frequency greater than 0 and created a node with the character value and frequency of the character. I inserted this node into a min-heap which has an underlying vector that resizes based on the number of nodes. Next I created a huffmanTree object to use my createHeap method. This method takes in one heap and combines the nodes to create a huffman encoding tree. The heap that this method returns just has one item, a huffman node that has children of the rest of the nodes in the original heap. After creating the second heap, I called my printPrefix method. This method traverses the tree and prints the prefixes associated with each character. This is done recursively and adds a 0 or 1 based on which side of the tree is being traversed. If the node that the method is called on is a leaf it outputs the prefix and the character value of the node to the console. Next I called a similar method to my print method on the heap to actually set the prefix values of the nodes. After this, I printed out the message with the encoded values. I did this by restarting the file reader to the beginning of the file and reading character by character. I made a vector based off the vector of the first heap and if the character read was equal to one of the characters in the vector it added the size of the compressed prefix as well as outputted the prefix for that character. This was done in linear time, based on the number of characters in the original input file. The last part was calculating compression. This was done by computing the size of the originial file, the number of characters multiplied by 8 bits divided by the size of the compressed bits as added before. Last was the cost, dividing the size of the compressed bits by the amount of characters in the original file.

Decoding

There was not as much to do for the decoding part. To decode first you must build a huffman tree based on the prefixes read in through the input file. To do this I created a method named buildTree. This method took in a node, a prefix, a character, an int count. The method reads the prefix string and if the character at the index of the string is a zero or one creates a new node at the left or right child of the node and recursively calls the function on that new node but with the next character in the prefix string. After the length of the prefix code is equal to the count int, it sets the nodes value to the character given. The space required for this is the size of the string, the size of the char, the size of the int, and two pointers to children for each node. This function is called until the end of the prefix section is over. After this all of the bits for the whole string are read in and added to a string through a stringstream (given as example code). The last part loops through the allbits string. This function takes each character of the string and tests to see if it is a 1 or 0 or is a leaf. If the node is a leaf then the loop adds the character value of the string onto the back of an empty string and sets the node being evaluated to the original root node.. If the value is a zero or one it sets the node being evaluated to the left or right child of the root node called. This loop would take linear time, the amount of characters in the allbits string. Finally the string is outputted to the console.