```
class Node:
    def __init__(self,left=None,right=None,value=None,frequency=None):
        self.left = left
        self.right = right
        self.value = value
        self.frequency = frequency

    def children(self):
        return (self.left,self.right)



class Huffman_Encoding:
    def __init__(self,string):
        self.q = []
        self.string = string
        self.encoding = {}

    def char_frequency(self):
        count = {}
        for char in self.string:
            if char not in count:
                count[char] = 0
            count[char] += 1

        for char,value in count.items():
            node = Node(value=char,frequency=value)
            self.q.append(node)
        self.q.sort(key=lambda x: x.frequency)

    def build_tree(self):
        while len(self.q) > 1:
            n1 = self.q.pop(0)
            n2 = self.q.pop(0)
            node = Node(left=n1,right=n2,frequency=n1.frequency + n2.frequency)
            self.q.append(node)
            self.q.sort(key = lambda x:x.frequency)


    def helper(self,node:Node,binary_str="",):
        if type(node.value) is str:
            self.encoding[node.value] = binary_str
            return
        l,r = node.children()
        self.helper(node.left,binary_str + "0")
        self.helper(node.right,binary_str + "1")
        print(node.frequency)
        return


    def huffman_encoding(self):
        root = self.q[0]
        self.helper(root,"")
    def print_encoding(self):
        print(' Char | Huffman code ')
        for char,binary in self.encoding.items():
            print(" %-4r |%12s" % (char,binary))

    def encode(self):
        self.char_frequency()
        self.build_tree()
        self.huffman_encoding()
        self.print_encoding()

string = input("Enter string to be encoded: ")
# string = 'AAAAAAABBCCCCCCDDDEEEEEEEEE'
encode = Huffman_Encoding(string)
encode.encode()


# The time complexity for encoding each unique character based on its frequency
is O(nlog n).

# Extracting minimum frequency from the priority queue takes place 2*(n-1) times
and its complexity is O(log n). Thus the overall complexity is O(nlog n).
```