```
#Huffma ass 2
class Node:
    def __init__(self, probability, symbol, left=None, right=None):
        self.probability = probability
        self.symbol = symbol
        self.left = left
        self.right = right
        self.code = ''
def calculate_probability(data):
    symbols = {}
    for item in data:
        if symbols.get(item) is None:
            symbols[item] = 1
        else: symbols[item] += 1
    return symbols
def calculate_codes(node, value='', codes=None):
    if codes is None:
        codes = {}
    new_value = value + str(node.code)
    if node.left:
        calculate_codes(node.left, new_value, codes)
    if node.right:
        calculate_codes(node.right, new_value, codes)
    if not node.left and not node.right:
        codes[node.symbol] = new_value
    return codes
def total_gain(data, coding):
    before_compression = len(data) * 8
    after_compression = 0
    symbols = coding.keys()
    for symbol in symbols:
        the_count = data.count(symbol)
        after_compression += the_count * len(coding[symbol])
    print("Space usage before compression (in bits):", before_compression)
    print("Space usage after compression (in bits):", after_compression)
def huffman_encoding(data):
    symbol_with_probs = calculate_probability(data)
    the_symbols = symbol_with_probs.keys()
    the_probabilities = symbol_with_probs.values()
    print("symbols: ", the_symbols)
    print("probabilities: ", the_probabilities)
    the_nodes = []
    for symbol in the_symbols:
        the_nodes.append(Node(symbol_with_probs.get(symbol), symbol))
    while len(the_nodes) > 1:
        the_nodes = sorted(the_nodes, key=lambda x: x.probability)
        right = the_nodes[0]
        left = the_nodes[1]
        left.code = '0'
        right.code = '1'
        new_node = Node(left.probability + right.probability,
        left.symbol + right.symbol, left, right)
        the_nodes.remove(left)
        the_nodes.remove(right)
        the_nodes.append(new_node)
    huffman_codes = calculate_codes(the_nodes[0])
    print("symbols with codes", huffman_codes)
    total_gain(data, huffman_codes)
    encoded_output = output_encoded(data, huffman_codes)
    return encoded_output, huffman_codes
def output_encoded(data, coding):
    encoding_output = []
    for element in data:
```

```
        encoding_output.append(coding[element])
    the_string = ''.join([str(item) for item in encoding_output])
    return the_string
def huffman_decoding(encoded_data, huffman_codes):
    decoded_output = []
    current_code = ""
    for bit in encoded_data:
        current_code += bit
        for symbol, code in huffman_codes.items():
            if current_code == code:
                decoded_output.append(symbol)
                current_code = ""
                break
    string = ''.join(decoded_output)
    return string
the_data = "AAAAAAABCCCCCCDDEEEEE"
print(the_data)
encoded_output, huffman_codes = huffman_encoding(the_data)
print("Encoded output", encoded_output)
```

Huffman Encoding

```
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()
```