```
class Node:
    def __init__(self, prob, symbol, left=None, right=None):
        # probability of symbol
        self.prob = prob

        # symbol
        self.symbol = symbol

        # left node
        self.left = left

        # right node
        self.right = right

        # tree direction (0/1)
        self.code = ''
```

```
""" A helper function to calculate the probabilities of symbols in given data"""
def Calculate_Probability(data):
    symbols = dict()
    for element in data:
        if symbols.get(element) == None:
            symbols[element] = 1
        else:
            symbols[element] += 1
    return symbols
```

```
""" A helper function to obtain the encoded output"""
def Output_Encoded(data, coding):
    encoding_output = []
    for c in data:
        print(coding[c], end = '')
        encoding_output.append(coding[c])

    string = ''.join([str(item) for item in encoding_output])
    return string
```

```
""" A helper function to calculate the space difference between compressed and non co
def Total_Gain(data, coding):
    before_compression = len(data) * 8 # total bit space to stor the data before compr
    after_compression = 0
    symbols = coding.keys()
    for symbol in symbols:
        count = data.count(symbol)
        after_compression += count * len(coding[symbol]) #calculate how many bit is r
    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)
    symbols = symbol_with_probs.keys()
    probabilities = symbol_with_probs.values()
    print("symbols: ", symbols)
    print("probabilities: ", probabilities)

    nodes = []

    # converting symbols and probabilities into huffman tree nodes
    for symbol in symbols:
        nodes.append(Node(symbol_with_probs.get(symbol), symbol))

    while len(nodes) > 1:
        # sort all the nodes in ascending order based on their probability
        nodes = sorted(nodes, key=lambda x: x.prob)
        # for node in nodes:
        #       print(node.symbol, node.prob)

        # pick 2 smallest nodes
        right = nodes[0]
        left = nodes[1]

        left.code = 0
        right.code = 1

        # combine the 2 smallest nodes to create new node
        newNode = Node(left.prob+right.prob, left.symbol+right.symbol, left, right)

        nodes.remove(left)
        nodes.remove(right)
        nodes.append(newNode)

    huffman_encoding = Calculate_Codes(nodes[0])
    print(huffman_encoding)
    Total_Gain(data, huffman_encoding)
    encoded_output = Output_Encoded(data,huffman_encoding)
    print("Encoded output:", encoded_output)
    return encoded_output, nodes[0]
```

```
AAAAAAABCCCCCCDDEEEEE
symbols:  dict_keys(['A', 'B', 'C', 'D', 'E'])
probabilities:  dict_values([7, 1, 6, 2, 5])
symbols with codes {'A': '00', 'C': '01', 'E': '10', 'D': '110', 'B': '111'}
Space usage before compression (in bits): 168
Space usage after compression (in bits): 45
Encoded output 000000000000001110101010101011101101010101010
```