Code

```python
import heapq

# this class will represent the huffman tree node
class HuffmanTree:
    def __init__(self, priority, char, l_child=None, r_child=None):
        self.priority = priority
        self.char = char
        self.l_child = l_child
        self.r_child =r_child
        self.position = ''


    # heapq uses this to measure the priority
    def __lt__(self, other):
        return self.priority < other.priority

    def __str__(self):
        return f"{self.char}"


# function for creating huffman tree nodes
def make_huffman_tree(char_count) -> []: # [(char, freq), ...]
    min_heap = []

    # adds all the nodes into the priority queue
    for char, freq in char_count.items():
        node = HuffmanTree(priority=freq, char=char)
        heapq.heappush(min_heap, node)

    while len(min_heap) > 1:
        l_child = heapq.heappop(min_heap)
        r_child = heapq.heappop(min_heap)

        l_child.position = '0'
        r_child.position = '1'
        internal_node = HuffmanTree(
            priority=(l_child.priority + r_child.priority),
            char=str(l_child.char+r_child.char),
            l_child=l_child,
            r_child=r_child
        )

        heapq.heappush(min_heap, internal_node)

    return min_heap[0]

def get_huffman_codes(node: HuffmanTree, code='', code_dict=None) ->
dict:
    if code_dict is None:
        code_dict = {}

    if not node.l_child and not node.r_child:
        code_dict[node.char] = code

    if node.l_child:
        get_huffman_codes(node.l_child, code + '0', code_dict)

    if node.r_child:
        get_huffman_codes(node.r_child, code + '1', code_dict)

    return code_dict

def encode_message(msg, codes) -> str:
    encoded_message = ""

    for char in msg:
        encoded_message+=codes[char]

    return encoded_message

def decode_message(msg, codes) -> str:
    decoded_message=""

    rev_codes = {v: k for k,v in codes.items()}

    # parses the message until a huffman code is found for the bit
```

```
pattern
    while msg:
        for i in range(1, len(msg) + 1):
            code = msg[:i]

            if code in rev_codes:
                decoded_message += rev_codes[code]
                msg = msg[i:]
                break

    return decoded_message
def make_char_freqList(str):
    char_count = dict()

    for char in list(str):
        if char in char_count:
            char_count[char] += 1
        else:
            char_count[char] = 1

    return char_count


# --- Task 1: Testing with message
msg = input("Please enter a message: ")

char_count = make_char_freqList(msg)
huffman_tree = make_huffman_tree(char_count)
huffman_codes = get_huffman_codes(huffman_tree)
e_msg = encode_message(msg, huffman_codes)

print(f"Input: {msg}")
print(f"Encoded: {e_msg}\n")
print(f"Guide: ")

for k,v in huffman_codes.items():
    print(f"{k}:{v}")


# encoding the file
with open("user_message.txt", "r+") as file:
```

```python
    msg = ""
    for line in file.readlines():
        msg += line

    char_count = make_char_freqList(msg)
    huffman_tree = make_huffman_tree(char_count)
    huffman_codes = get_huffman_codes(huffman_tree)

    with open("encoded.txt", "w+") as encoded_file:
        e_msg = encode_message(msg, huffman_codes)
        print(f"\n\nencoded message: {e_msg}")
        encoded_file.write(e_msg)

# decoding the file
with open("encoded.txt", "r+") as decode_file:
    msg=decode_file.read()

    d_msg = decode_message(msg, huffman_codes)
    print(f"Decoded Message: {d_msg}")
```

Output

```
C:\Users\aaron\AppData\Local\Programs\Python\Python39\python.exe
C:\Users\aaron\Projects\classWork\CMPSC413\lab_5_v2\huffman_tree.py
Please enter a message:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBCCCDD
Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBCCCDD
Encoded:
1111111111111111111111111111111111111111111110101010101010100100100100
0000

Guide:
D:000
C:001
B:01
A:1
```

```
encoded message:
0110101001110011101010011110100011010011111111111101011011000000100001
0101001111111000000001100100110010111001011001111110001011101100010111 1
1111010001000011100110111010100111110101110110000101010110101110011101
0101011101111000000001100100000010100011000001
Decoded Message: HELLO, HUFFMAN CODING IS SUPER EFFECTIVE FOR
COMPRESSING TEXT

Process finished with exit code 0
```

Time complexity analysis

this program consists of a few separate parts that work in conjunction to make the algorithm function. First, a priority queue (min_heap) is used to order the nodes by their frequency from least to greatest.

the process of creating the Huffman tree involved appending the nodes of unique characters to a min heap and then popping and reinserting internal nodes that represented their summed priorities. For n-nodes "heapification" takes $O(logn)$ time and popping the nodes from the priority queue takes $O(2(n-1)) = O(n)$ time. In total, we have a combined time complexity of:

$$O(nlogn)$$

there was also the overhead of counting the frequency of each character in the string. This operation takes $O(m)$ time where m is the number of characters in the string. This algorithm is quite efficient since its almost always the case that the length of the string is greater than the number of unique characters. We wont consider this though since practically computers are able to read files quite quickly.