EXPERIMENT NO: 02

```
import heapq
from collections import defaultdict
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    # Define less than for heapq to compare Nodes by frequency
    def __lt__(self, other):
        return self.freq < other.freq
def build_huffman_tree(text):
    # Count frequency of each character
    frequency = defaultdict(int)
    for char in text:
        frequency[char] += 1
    # Create a priority queue (min-heap)
    heap = [Node(char, freq) for char, freq in frequency.items()]
    heapq.heapify(heap)
    # Build the Huffman tree
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)  # Create a new internal node
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]  # Return the root of the tree
```

```
def generate_codes(node, current_code="", codes={}):
    if node is None:
        return
    # If it's a leaf node, add the character and its code to the dictionary
    if node.char is not None:
        codes[node.char] = current_code
    # Traverse left and right children
    generate_codes(node.left, current_code + "0", codes)
    generate_codes(node.right, current_code + "1", codes)
    return codes
def huffman_encoding(text):
    if not text:
        return "", None
    root = build_huffman_tree(text)
    huffman_codes = generate_codes(root)
    # Encode the input text
    encoded_text = "".join(huffman_codes[char] for char in text)
   return encoded_text, huffman_codes
def huffman_decoding(encoded_text, huffman_codes):
    if not encoded_text or not huffman_codes:
        return ""
    # Create a reverse mapping of codes to characters
    reverse_codes = {v: k for k, v in huffman_codes.items()}
    current_code = ""
    decoded_text = ""
    for bit in encoded_text:
        current_code += bit
        if current_code in reverse_codes:
            decoded_text += reverse_codes[current_code]
            current_code = ""  # Reset for next character
    return decoded_text
if __name__ == "__main__":
    text = "Huffman Encoding is a data compression algorithm."
    print("Original Text:", text)
    # Encode the text
    encoded_text, huffman_codes = huffman_encoding(text)
    print("Encoded Text:", encoded_text)
    print("Huffman Codes:", huffman_codes)
    # Decode the text
    decoded_text = huffman_decoding(encoded_text, huffman_codes)
    print("Decoded Text:", decoded_text)
    # Verify that decoding works correctly
    assert text == decoded_text, "Decoded text does not match original text!"
```

OUTPUT:-

```
Original Text: Huffman Encoding is a data compression algorithm.

Encoded Text:
10111011111101100000101001011100111001001100101101111010101011111001111
0001011111010101101010010111111011001110001110101000011111011110110001 1
1110111001000001000101111000010000000

Huffman Codes: {' ': '00', 'H': '010', 'u': '1001', 'f': '1101', 'm': '1100', 'a': '1011', 'n':
'1110', 'E': '1111', 'n': '0110', 'c': '0010', 'o': '0111', 'd': '0001', 'i': '0100', 'g': '0111', 's':
'0000', 't': '0011', 'l': '1000', '.': '1010'}

Decoded Text: Huffman Encoding is a data compression algorithm.
```