**Exercise - 5**

**Huffman Coding**

**Aim:** To write a python code to Huffman Coding algorithm

**Algorithm:**

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

**Source Code:**

from heapq import heappush, heappop, heapify

from collections import defaultdict

class HuffmanNode:

    def \_\_init\_\_(self, char, frequency):

        self.char = char

        self.frequency = frequency

        self.left = None

        self.right = None

    def \_\_lt\_\_(self, other):

        return self.frequency < other.frequency

def build\_frequency\_table(data):

    frequency\_table = defaultdict(int)

    for char in data:

        frequency\_table[char] += 1

    return frequency\_table

def build\_huffman\_tree(frequency\_table):

    priority\_queue = []

    for char, frequency in frequency\_table.items():

        node = HuffmanNode(char, frequency)

        heappush(priority\_queue, node)

    while len(priority\_queue) > 1:

        left\_node = heappop(priority\_queue)

        right\_node = heappop(priority\_queue)

        sum\_frequency = left\_node.frequency + right\_node.frequency

        parent\_node = HuffmanNode(None, sum\_frequency)

        parent\_node.left = left\_node

        parent\_node.right = right\_node

        heappush(priority\_queue, parent\_node)

    return priority\_queue[0]

def build\_huffman\_codes(node, current\_code, huffman\_codes):

    if node is None:

        return

    if node.char:

        huffman\_codes[node.char] = current\_code

    build\_huffman\_codes(node.left, current\_code + '0', huffman\_codes)

    build\_huffman\_codes(node.right, current\_code + '1', huffman\_codes)

def huffman\_encoding(data):

    frequency\_table = build\_frequency\_table(data)

    huffman\_tree = build\_huffman\_tree(frequency\_table)

    huffman\_codes = {}

    build\_huffman\_codes(huffman\_tree, '', huffman\_codes)

    print(huffman\_codes)

    encoded\_data = ''.join(huffman\_codes[char] for char in data)

    return encoded\_data, huffman\_tree

def huffman\_decoding(encoded\_data, huffman\_tree):

    decoded\_data = ''

    current\_node = huffman\_tree

    for bit in encoded\_data:

        if bit == '0':

            current\_node = current\_node.left

        else:

            current\_node = current\_node.right

        if current\_node.char:

            decoded\_data += current\_node.char

            current\_node = huffman\_tree

    return decoded\_data

data = input("Enter the data: ")

encoded\_data, huffman\_tree = huffman\_encoding(data)

print("Encoded data:", encoded\_data)

decoded\_data = huffman\_decoding(encoded\_data, huffman\_tree)

print("Decoded data:", decoded\_data

**Sample Input and Output:**

A screenshot of a computer

Description automatically generated with low confidence

**Result:**

Thus, Huffman Coding algorithm has been successfully implemented using Python code and the output is verified.