Name: Vaishnavi Pravin Kolse
Class: BE - A
Roll No.37

Practical No.2

Aim: Write a program to implement Huffman Encoding using a greedy strategy.

In [3]:
```python
import heapq

# Defining the Node class for Huffman Tree
class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq                    # Frequency of the character
        self.symbol = symbol                # The character itself
        self.left = left                    # Left child node
        self.right = right                  # Right child node
        self.huff = ""                      # This will store the Huffman code

    # Defining comparison operators to be able to use heapq
    def __lt__(self, other):
        return self.freq < other.freq

# Function to print the Huffman Codes and calculate their lengths
def printNodes(node, val=""):
    newval = val + node.huff

    # If the node is not a leaf node, continue traversing
    if node.left or node.right:
        if node.left:
            printNodes(node.left, newval)
        if node.right:
            printNodes(node.right, newval)
    else:
        # If it's a leaf node, print the symbol and its Huffman code
        print(f"{node.symbol} -> {newval}")
        encoded_lengths[node.symbol] = len(newval)

# Getting user input for characters and their frequencies
num_chars = int(input("Enter number of characters: "))
chars = []
freqs = []

for i in range(num_chars):
    char = input(f"Enter character {i + 1}: ")
    freq = int(input(f"Enter frequency of character {char}: "))
    chars.append(char)
    freqs.append(freq)

# Building the Huffman Tree
nodes = []

# Pushing all characters and their frequencies as nodes into a priority queue
for i in range(len(chars)):
    heapq.heappush(nodes, Node(freqs[i], chars[i]))

# Merging nodes to create the Huffman Tree
while len(nodes) > 1:
    left = heapq.heappop(nodes)
    right = heapq.heappop(nodes)

    # Assign Huffman codes: '0' to left, '1' to right
    left.huff = "0"
    right.huff = "1"
```

```
    # Create new internal node with combined frequency
    newnode = Node(left.freq + right.freq, left.symbol + right.symbol, left, r
    heapq.heappush(nodes, newnode)

# Calculating total size before encoding (each character occupies 8 bits)
total_size_before = sum(freqs) * 8

# Dictionary to store the length of Huffman codes for each character
encoded_lengths = {}

# Print the Huffman codes and calculate their lengths
print("\nHuffman Codes:")
printNodes(nodes[0])

# Calculating total size after encoding based on Huffman codes
total_size_after = sum(freqs[i] * encoded_lengths[chars[i]] for i in range(num

# Calculating Encoded Data Representation
characters = num_chars * 8  # Each character represented in 8 bits
frequency = sum(freqs)      # Sum of frequencies
encoded_data_representation = characters + frequency + total_size_after

# Displaying the results
print("\nTotal size before encoding:", total_size_before, "bits")
print("Total size after encoding:", total_size_after, "bits")
print("Encoded Data Representation:", encoded_data_representation, "bits")
```

```
Enter number of characters: 4
Enter character 1: B
Enter frequency of character B: 1
Enter character 2: C
Enter frequency of character C: 6
Enter character 3: A
Enter frequency of character A: 5
Enter character 4: D
Enter frequency of character D: 3

Huffman Codes:
C -> 0
B -> 100
D -> 101
A -> 11

Total size before encoding: 120 bits
Total size after encoding: 28 bits
Encoded Data Representation: 75 bits
```

In [ ]: