Problem Statement:

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

Huffman Encoding Overview:

Huffman Encoding is a lossless data compression algorithm. It uses a greedy strategy to assign variable-length codes to characters based on their frequencies. Characters that occur more frequently are assigned shorter codes, while less frequent characters are assigned longer codes.

Steps for Huffman Encoding:

1. **Calculate Character Frequencies**: Count the frequency of each character in the given input string.
2. **Build a Priority Queue (Min-Heap)**: Insert each character and its frequency into a priority queue (min-heap). The heap is used to ensure that we can easily retrieve the two least frequent characters.
3. **Build the Huffman Tree**:
    - While there are more than one node in the heap:
        - Remove the two nodes with the smallest frequency from the heap.
        - Create a new internal node with a frequency equal to the sum of the two nodes' frequencies.
        - Insert this new node back into the heap.
    - The remaining node is the root of the Huffman Tree.
4. **Assign Codes**: Traverse the Huffman Tree and assign binary codes to characters:
    - Assign '0' for left branches and '1' for right branches of the tree.
5. **Encode the Input String**: Use the generated codes to encode the input string.
6. **Decode the Encoded String**: Traverse the Huffman Tree to decode the encoded string back to its original form.

Procedure (Huffman Encoding):

1. Build Frequency Table:

- Traverse the input string and count the occurrences of each character. This results in a frequency table.

2. Build a Min-Heap (Priority Queue):

- Create a min-heap where each element is a node containing a character and its frequency. Each node is considered as a leaf in the Huffman Tree.

3. Build the Huffman Tree:

- While there is more than one node in the heap:
    - Extract the two nodes with the smallest frequencies.

    - Create a new internal node whose frequency is the sum of the two nodes.
    - Insert this internal node back into the heap.
- The final node left in the heap will be the root of the Huffman Tree.

4. Assign Binary Codes to Characters:

- Perform a tree traversal (preorder or inorder) to assign a binary code to each character based on the path from the root to the leaf.

```
import heapq
# Node structure for Huffman Tree
class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    # Defining less than operator for priority queue comparison
    def __lt__(self, other):
        return self.freq < other.freq

# Function to generate Huffman codes
def generate_codes(root, current_code, codes):
    if root is None:
        return
    if root.char is not None:
        codes[root.char] = current_code
    generate_codes(root.left, current_code + "0", codes)
    generate_codes(root.right, current_code + "1", codes)

# Function to build Huffman Tree
def build_huffman_tree(frequency):
    heap = []
    # Insert all characters with their frequencies into the heap
    for char, freq in frequency.items():
        heapq.heappush(heap, HuffmanNode(char, freq))

    # Merge nodes until we have one tree
    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)

        # Create a new internal node with the combined frequency
        merged = HuffmanNode(None, node1.freq + node2.freq)
        merged.left = node1
        merged.right = node2
        heapq.heappush(heap, merged)

    # The root of the Huffman Tree
    return heapq.heappop(heap)

# Function to calculate frequency of characters
def calculate_frequency(data):
    frequency = {}
```

```
    for char in data:
        if char not in frequency:
            frequency[char] = 0
        frequency[char] += 1
    return frequency

# Huffman Encoding process
def huffman_encoding(data):
    frequency = calculate_frequency(data)
    huffman_tree_root = build_huffman_tree(frequency)

    codes = {}
    generate_codes(huffman_tree_root, "", codes)

    # Encode the input data
    encoded_data = "".join([codes[char] for char in data])

    return encoded_data, huffman_tree_root

# Huffman Decoding process
def huffman_decoding(encoded_data, huffman_tree_root):
    decoded_data = ""
    current_node = huffman_tree_root
    for bit in encoded_data:
        if bit == '0':
            current_node = current_node.left
        else:
            current_node = current_node.right

        if current_node.left is None and current_node.right is None:
            decoded_data += current_node.char
            current_node = huffman_tree_root

    return decoded_data

# Driver code
if __name__ == "__main__":
    data = "huffman encoding example"

    encoded_data, huffman_tree_root = huffman_encoding(data)
    print(f"Encoded Data: {encoded_data}")

    decoded_data = huffman_decoding(encoded_data, huffman_tree_root)
    print(f"Decoded Data: {decoded_data}")
```

Example:

Given an input string `data = "huffman encoding example"`, the steps are:

1. **Frequency Calculation**:
    - e: 3, f: 2, a: 2, etc.
2. **Build Min-Heap**: Insert the frequencies of characters into the heap.
3. **Build Huffman Tree**: Merge nodes based on frequency to form the Huffman Tree.
4. **Generate Codes**: Assign binary codes by traversing the Huffman Tree:

- `h`: 1101, `u`: 1100, `f`: 1110, etc.

5. **Encode**: Convert the input string into its binary encoded form using the generated codes.
6. **Decode**: Convert the binary encoded string back into the original string using the Huffman Tree.

Time and Space Complexity Analysis:

Time Complexity:

- **Building Frequency Table**: O(n), where `n` is the length of the input string.
- **Building Min-Heap**: O(d log d), where `d` is the number of distinct characters.
- **Building Huffman Tree**: O(d log d).
- **Encoding the Input String**: O(n).

Overall Time Complexity: O(n + d log d), where `n` is the length of the string, and `d` is the number of distinct characters.

Space Complexity:

- Storing the frequency table and the Huffman Tree: O(d).
- The encoded string may take more space than the original string, but typically, Huffman encoding results in space savings for large data.

Space Complexity: O(n + d), where `n` is the input size, and `d` is the number of distinct characters.

Java Codes

```
import java.util.PriorityQueue;
class HuffmanNode {
    int data;
    char c;
    HuffmanNode left;
    HuffmanNode right;
}
class MyComparator implements java.util.Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y) {
        return x.data - y.data;
    }
}
public class Huffman {
    public static void printCode(HuffmanNode root, String s) {
        if (root.left == null && root.right == null &&
Character.isLetter(root.c)) {
            System.out.println(root.c + ":" + s);
            return;
        }
        printCode(root.left, s + "0");
        printCode(root.right, s + "1");
    }
```

```
    public static void main(String[] args) {
        int n = 4;
        char[] charArray = { 'a', 'b', 'c', 'd' };
        int[] charfreq = { 5, 9, 12, 13 };

        PriorityQueue<HuffmanNode> q = new PriorityQueue<HuffmanNode>(n, new
MyComparator());

        for (int i = 0; i < n; i++) {
            HuffmanNode hn = new HuffmanNode();
            hn.c = charArray[i];
            hn.data = charfreq[i];
            hn.left = null;
            hn.right = null;
            q.add(hn);
        }
        HuffmanNode root = null;

        while (q.size() > 1) {
            HuffmanNode x = q.peek();
            q.poll();
            HuffmanNode y = q.peek();
            q.poll();
            HuffmanNode f = new HuffmanNode();
            f.data = x.data + y.data;
            f.c = '-';
            f.left = x;
            f.right = y;
            root = f;
            q.add(f);
        }
        printCode(root, "");
    }
}
```

Sample Output

```
c:00
a:01
b:10
d:11
```

Conclusion:

Huffman encoding uses a greedy strategy to compress data by assigning shorter codes to frequent characters and longer codes to less frequent ones. It is an efficient algorithm for lossless data compression, minimizing the total number of bits needed to represent the input.