

```
import heapq
```

```
class HuffmanNode:
```

```
    def __init__(self, cha, f):
        self.cha = cha
        self.Frequency = Freq
        self.left = None
        self.right = None
```

```
    def __lt__(self, other):
        if self.Frequency == other.Frequency:
            if self.cha is None:
                return False
            if other.cha is None:
                return True
            return self.cha < other.cha
        return self.Frequency < other.Frequency
```

```
# Using MinHeap as the primary data structure
class MinHeap:
```

```
    def __init__(self):
        self.heaps = []

    def push(self, node):
        heapq.heappush(self.heaps, node)

    def pop(self):
        return heapq.heappop(self.heaps)

    def size(self):
        return len(self.heaps)
```

```
class HuffmanCodingTree:
```

```
    def __init__(self):
        self.root = None
        self.codes = {}
        self.mini_heap = MinHeap
```

```

        self.mini_heap = min_heap

def build_tree(self, char_freq):
    # Now, push all characters and their frequencies into the mini-heap
    for char, frequency in char_freq.items():
        self.mini_heap.push((frequency, char))

    # Building a tree using the mini-heap
    while self.mini_heap.size() > 1:
        # Pop the two smallest elements
        left = self.mini_heap.pop()
        right = self.mini_heap.pop()

        # Create an internal node
        internal = HuffmanNode()
        internal.left = left
        internal.right = right

        # Push the internal node back into the mini-heap
        self.mini_heap.push((left.freq + right.freq, internal))

    self.root = self.mini_heap.pop()

def generate_codes(self):
    def dfs(node, current_code):
        if node is None:
            return

        # If it's a leaf node, store the code
        if node.char is not None:
            self.codes[node.char] = current_code
            return

        # Recursively generate codes for left and right children
        dfs(node.left, current_code + '0')
        dfs(node.right, current_code + '1')

    dfs(self.root, "")

def print_tree(self, node):
    if node is None:
        node = self.root

    if node.char is not None:
        print(f"Character: {node.char}, Code: {self.codes[node.char]}")
    else:
        print(f"Internal Node: Left={node.left}, Right={node.right}")

```

```

        print(f"{prefix}{
    else:
        print(f"{prefix}{
        self.print_tree(r
        self.print_tree(r

```

```

def get_user_input():
    cha_Frequency = {}
    print("Enter character-f
    while True:
        cha = input(" Enter (
        if not cha:
            break
        try:
            Frequency = int(
            if Frequency <= 0
                print("Freque
                continue
            cha_Frequency[cha
        except ValueError:
            print("Invalid fi
    return cha_Frequency

```

```

def main():
    cha_Frequency = get_user_

    if not cha_Frequency:
        print("Input was not
        return

    huffman_tree = HuffmanCoc
    huffman_tree.build_tree(c
    huffman_tree.generate_coc

    print("\nHuffman Tree:")
    huffman_tree.print_tree()

    print("\nHuffman Codes:")

```

```
for cha, code in sorted(f
    print(f"{cha}: {code]
```

```
if __name__ == "__ma
    main()
```



Enter character–frequency pairs (press Enter without input to finish):

Enter Character or press enter if completed:

Input was not provided.So,Program is exited.

Double-click (or enter) to edit