HUFFMAN CODING

1. Node Structure

- Defines a tree node to store:
    - A character (ch).
    - The frequency of occurrence (freq).
    - Left and right child pointers (left, right).

```
struct Node
{
        char ch;
        int freq;
        Node *left, *right;
};
```

2. Creating a New Node

- Allocates a new Node and initializes its values.

```
Node* getNode(char ch, int freq, Node* left, Node* right)
{
        Node* node = new Node();
        node->ch = ch;
        node->freq = freq;
        node->left = left;
        node->right = right;
        return node;
}
```

3. Priority Queue Comparison Structure

- Defines a custom comparator for the priority queue.
- Ensures the node with the **smallest frequency** has the highest priority.

```
 struct comp
{
        bool operator()(Node* l, Node* r)
        {
                return l->freq > r->freq; // Min-Heap: lowest frequency at the top
        }
};
```

4. Encoding Function

- Recursively traverses the Huffman Tree.
- Assigns 0 for left and 1 for right.
- Stores the Huffman codes in unordered_map<char, string>.

```
  void encode(Node* root, string str, unordered_map<char, string>
&huffmanCode)
{
        if (root == nullptr)
                return;
        // Leaf node (character node)
        if (!root->left && !root->right) {
                huffmanCode[root->ch] = str;
        }
        encode(root->left, str + "0", huffmanCode);
        encode(root->right, str + "1", huffmanCode);
}
```

5. Decoding Function

- Traverses the Huffman Tree using the encoded string.
- Prints the decoded characters when reaching a leaf node.

```
void decode(Node* root, int &index, string str)
{
    if (root == nullptr) {
        return;
    }
    // Leaf node reached, print character
    if (!root->left && !root->right)
    {
        cout << root->ch;
        return;
    }
    index++;
    if (str[index] == '0')
        decode(root->left, index, str);
    else
        decode(root->right, index, str);
}
```

6. Building Huffman Tree

- **Step 1:** Count frequency of each character.
- **Step 2:** Push all characters as leaf nodes into a min-heap.
- **Step 3:** Construct Huffman Tree by merging nodes with the smallest frequency.
- **Step 4:** Generate Huffman codes using the encode function.
- **Step 5:** Encode and decode the input string.

```
   void buildHuffmanTree(string text)
{
        // Step 1: Count frequency of characters
        unordered_map<char, int> freq;
        for (char ch: text) {
                freq[ch]++;
        }
        // Step 2: Create priority queue (min-heap)
        priority_queue<Node*, vector<Node*>, comp> pq;
        for (auto pair: freq) {
                pq.push(getNode(pair.first, pair.second, nullptr, nullptr));
        }
        // Step 3: Construct Huffman Tree
        while (pq.size() != 1)
        {
                Node *left = pq.top(); pq.pop();
                Node *right = pq.top(); pq.pop();
                int sum = left->freq + right->freq;
                pq.push(getNode('\0', sum, left, right)); // Internal node with sum of
frequencies
        }
        // Step 4: Generate Huffman codes
        Node* root = pq.top();
        unordered_map<char, string> huffmanCode;
        encode(root, "", huffmanCode);

        // Step 5: Print Huffman codes
        cout << "Huffman Codes are :\n";
```

```
        for (auto pair: huffmanCode) {
                cout << pair.first << " " << pair.second << '\n';
        }
        // Encode input text
        cout << "\nOriginal string was:\n" << text << '\n';
        string str = "";
        for (char ch: text) {
                str += huffmanCode[ch];
        }
        cout << "\nEncoded string is:\n" << str << '\n';
        // Decode the encoded string
        int index = -1;
        cout << "\nDecoded string is: \n";
        while (index < (int)str.size() - 2) {
                decode(root, index, str);
        }
}
```

7. Main Function

- Calls buildHuffmanTree() with the input text.

```
int main()
{
        string text = "Huffman coding is a data compression algorithm.";
        buildHuffmanTree(text);
        return 0;
}
```

C++ implementation

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// A Tree node for Huffman coding
struct Node {
  char ch;
  int freq;
  Node *left, *right;
};

// Function to allocate a new tree node
Node* getNode(char ch, int freq, Node* left, Node* right) {
  Node* node = new Node();
  node->ch = ch;
  node->freq = freq;
  node->left = left;
  node->right = right;
  return node;
}

// Min-Heap implementation
struct MinHeap {
```

```
Node* heap[1000]; // Assuming max 1000 elements
int size;

MinHeap() { size = 0; }

void heapify(int i) {
  int smallest = i;
  int left = 2 * i;
  int right = 2 * i + 1;

  if (left <= size && heap[left]->freq < heap[smallest]->freq)
    smallest = left;
  if (right <= size && heap[right]->freq < heap[smallest]->freq)
    smallest = right;

  if (smallest != i) {
    swap(heap[i], heap[smallest]);
    heapify(smallest);
  }
}

void insert(Node* node) {
  size++;
  int i = size;
  heap[i] = node;
  while (i > 1 && heap[i]->freq < heap[i / 2]->freq) {
    swap(heap[i], heap[i / 2]);
    i = i / 2;
```

```
    }
  }

  Node* extractMin() {
    Node* minNode = heap[1];
    heap[1] = heap[size];
    size--;
    heapify(1);
    return minNode;
  }
};

// Function to generate Huffman codes recursively
void encode(Node* root, string str, unordered_map<char, string>&
huffmanCode) {
  if (!root) return;
  if (!root->left && !root->right)
    huffmanCode[root->ch] = str;
  encode(root->left, str + "0", huffmanCode);
  encode(root->right, str + "1", huffmanCode);
}

// Function to decode a Huffman encoded string
void decode(Node* root, int& index, string str) {
  if (!root) return;
  if (!root->left && !root->right) {
    cout << root->ch;
    return;
```

```
    }
    index++;
    if (str[index] == '0')
        decode(root->left, index, str);
    else
        decode(root->right, index, str);
}

// Function to build Huffman Tree and perform encoding & decoding
void buildHuffmanTree(string text) {
    unordered_map<char, int> freq;
    for (char ch : text) freq[ch]++;

    MinHeap minHeap;
    for (auto pair : freq)
        minHeap.insert(getNode(pair.first, pair.second, nullptr, nullptr));

    while (minHeap.size > 1) {
        Node* left = minHeap.extractMin();
        Node* right = minHeap.extractMin();
        Node* sumNode = getNode('\0', left->freq + right->freq, left, right);
        minHeap.insert(sumNode);
    }

    Node* root = minHeap.extractMin();
    unordered_map<char, string> huffmanCode;
    encode(root, "", huffmanCode);
```

```
    cout << "Huffman Codes:\n";
    for (auto pair : huffmanCode)
        cout << pair.first << " : " << pair.second << '\n';

    string encodedString = "";
    for (char ch : text) encodedString += huffmanCode[ch];
    cout << "\nEncoded String: " << encodedString << "\n";

    int index = -1;
    cout << "\nDecoded String: ";
    while (index < (int)encodedString.size() - 2)
        decode(root, index, encodedString);
    cout << "\n";
}

int main() {
    string text = "Huffman coding is a data compression algorithm.";
    buildHuffmanTree(text);
    return 0;
}
```