Reg.No: 24BCE0554
Name   : Partha Pratim Gogoi
Topic : Huffman Coding

Algorithm:

```
1 text = "message"                                            [T.C: O(1)]
2 freq = []                                                   [T.C: O(1)]
3 struct Node                                                 [T.C: O(1)]
  3.1  char data
  3.2  unsigned freq
  3.3  Node *left, *right
  3.4  Node(data, freq)
     3.4.1  this→data = data
     3.4.2  this→freq = freq
     3.4.3  left = right = NULL
4 calcFrequencies(string text)                                [T.C: O(n)]
  4.1  for char in text                                             O(n)
  4.2  freq[char]++
5 buildHuffmanTree(unordered_map(char, int) freq)  [T.C: O(n+mlogm)]
  5.1  priority_queue(Node) minHeap = []                            O(1)
  5.2  for char,freqValue in freq                                   O(m)
  5.3      minHeap.push(new Node(char, freqValue))             O(log(m))
  5.4  while minHeap.size > 1                                     O(m-1)
  5.5      Node left = minHeap.top()
  5.6      minHeap.pop()                                       O(log(m))
  5.7      Node right = minHeap.top()
  5.8      minHeap.pop()                                       O(log(m))
  5.9      Node internal = new Node('$',left.freq + right.freq)
  5.10     internal.left = left
  5.11     internal.right = right
  5.12     minHeap.push(internal)                              O(log(m))
  5.13 return minHeap.top()
6 storeCodes(
       Node rootNode,
       string code,
       unordered_map(char, int) codes
  )                                                         [T.C: O(m*L)]
  6.1  if rootNode == NULL
  6.2      return
  6.3  if rootNode.left == NULL and rootNode.right == NULL
  6.4      codes[rootNode.data] = code
  6.5  storeCodes(rootNode.left, code+"0", codes)                   O(L)
  6.6  storeCodes(rootNode.right, code+"1", codes)                  O(L)
```

Time Complexity:

IMPORTANT CLARIFICATION

- A Balanced tree can have ---→ m!=n OR m=n
  - 'abcdefgh' → balanced tree w/ m=n (special case, all unique)
  - 'aaaabbbbccc' → balanced tree with m!=n
- A Skewed tree can have -----→ m!=n
  - 'bbbbbcda' --→ generates skewed tree with m!=n
- It is not possible to get a Skewed tree with m=n

1. buildHuffmanTree()

A temporary data structure 'minHeap' used to create the Huffman Tree.

Time Complexity:

O(mlog(m)) + O((m-1)*3log(m)) = O(mlog(m))

where m = no. of unique characters

- Repeated chars (m!=n) = O(mlog(m))
  - Same for skewed and balanced tree
- When all characters unique (m=n) = O(nlog(n))

2. storeCodes()

- Total nodes visited : 2m-1 — 'm'original+'m-1'internal
- At each node : O(L) — String concatenation
- m!=n Case → O((2m-1)*L) : O(m*L) — L = Huffman Tree Height
  - Balanced Tree → O(mlog(m))
  - Skewed Tree → O(m^2)
- m=n Case → O((2n-1)*L) : O(n*L)

Where,

Huffman Tree Height (L) is determined by no. of unique characters (m)

NOTE: calcFrequency() is O(n) and not dependent on 'm' or 'L'

So, including calcFrequency() the total time complexity will be:

Total Time Complexity = O(n) + O(mlog(m)) + O(m*L)

- Balanced Tree (Typical/Expected Structure):

  O(n) + O(mlog(m)) + O(mlog(m))

  Total Time Complexity

  Repeated chars (m!=n) = O(n + mlogm)

  All chars unique (m=n) = O(nlogn) (special case)

- Skewed Tree (Pathological Structure):

  O(n) + O(mlog(m)) + O(m^2)

  Only one case

  Repeated chars = O(m^2)

Source Code:

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


struct Node {
    char data;
    unsigned freq;
    Node *left, *right;

    Node(char data, unsigned freq) {
        this->data = data;
        this->freq = freq;
        left = right = nullptr;
    }
};

struct Compare {
    bool operator()(Node* l, Node* r) {
        return l->freq > r->freq;
    }
};

// Non-Recursive Function
Node* buildHuffmanTree(unordered_map<char, int>& freq) {
    // Define minHeap
    priority_queue<Node*, vector<Node*>, Compare> minHeap;

    // Populate minHeap
    for (auto pair: freq) {
        minHeap.push(new Node(pair.first, pair.second));
    }
    // Build Huffman Tree
    while (minHeap.size() > 1){
        // Select the two elements from the minHeap
        // having least frequencies
        Node* left = minHeap.top();
        minHeap.pop();
        Node* right = minHeap.top();
        minHeap.pop();

        // Generate non-leaf node
        // Generated node chracter -> '$'
        Node* internal = new Node('$', left->freq + right->freq);
        internal->left = left;
        internal->right = right;

        // Push generated node to tree
        minHeap.push(internal);
    }
    return minHeap.top();
}

// Recursive Function
```

```cpp
// Tree is traversed top-to-bottom
void storeCodes(Node* root, string code, unordered_map<char, string>& codes) {
    if (!root) return;
    if (!root->left && !root->right) {
        codes[root->data] = code;
        return;
    } // If at leaf node, character code is complete
    storeCodes(root->left, code+"0", codes);        // If left edge, append 0
    storeCodes(root->right, code+"1", codes);       // If right edge, append 1
}

int main() {
    // 1. Get text input
    string text;
    cout << "Enter text to encode: ";
    getline(cin, text);

    // 2. Calculate frequencies
    cout << endl << "Calculating frequencies..." << endl;
    unordered_map<char, int> freq;
    for (char c: text) {
        freq[c]++;
    }
    // 3. Build the tree
    cout << "Building Huffman tree..." << endl;
    Node* root = buildHuffmanTree(freq);

    // 4. Generate the huffman codes
    cout << "Generating Huffman encoding table..." << endl;
    unordered_map<char, string> codes;
    storeCodes(root, "", codes);
    cout << endl;

    // 4. Print out encoding table
    cout << "Huffman Encoding Table" << endl;
    for (auto pair: codes) {
        cout << pair.first << ": " << pair.second << endl;
    }
    cout << endl;

    // 5. Print out compression results
    int originalBits = 8*text.length();
    int compressedBits = 0;
    string compressedMessage = "";

    for (char c: text) {
        compressedBits += codes[c].length();
        compressedMessage += codes[c];
    } // Calculating new length of compressed message

    cout << "Compressed Message: " << compressedMessage << endl << endl;
    cout << "No. of bits required to store original message: " << originalBits << endl;
    cout << "No. of bits required to store compressed version: " << compressedBits <<
endl;
    cout << "Size reduction: " << (originalBits-compressedBits)*100/originalBits << "%"
<< endl;

    return 0;
}
```

Sample Output:

```
rug@Oxide [Algorithm-Code]>> ./a.out
Enter text to encode: Hello World

Calculating frequencies...
Building Huffman tree...
Generating Huffman encoding table...

Huffman Encoding Table
l: 10
 : 011
W: 1111
d: 010
o: 110
H: 001
e: 1110
r: 000

Compressed Message: 00111101010110011111111000010010

No. of bits required to store original message: 88
No. of bits required to store compressed version: 32
Size reduction: 63%
rug@Oxide [Algorithm-Code]>> ./a.out
Enter text to encode: She sells sea shells by the sea shore

Calculating frequencies...
Building Huffman tree...
Generating Huffman encoding table...

Huffman Encoding Table
t: 10110
 : 111
a: 1010
e: 110
r: 01111
l: 100
y: 01110
b: 10111
o: 01101
S: 01100
h: 010
s: 00

Compressed Message: 0110001011011100110100100001110011010101110001011010010000111101110111011110110010110111001101010111000100110101111110

No. of bits required to store original message: 296
No. of bits required to store compressed version: 118
Size reduction: 60%
```