**VIET NAM NATIONAL UNIVERSITY**

**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY**



PROJECT REPORT

**HUFFMAN CODING**

|  |  |
| --- | --- |
| **Student’s name** | **ID number** |
| Nguyễn Tùng Dương |  |
| Nguyễn Phạm Tiến Dũng | 2051061 |
| Phạm Quốc Điền | 1951199 |

Contents

[**I. Theory:** 1](#_Toc121935598)

[**I.1. Instruction:** 1](#_Toc121935599)

[**I.2. Objective** 2](#_Toc121935600)

[**II. Algorithm:** 6](#_Toc121935602)

[**II.1. Describe the algorithm:** 6](#_Toc121935603)

[**II.2. Flowchart** 7](#_Toc121935604)

**III**[**. Conclusion** 21](#_Toc121935610)

[**I. Theory:**](#_Toc121935598)

**I.1. Instruction:**

The Huffman coding is widely used in data compression techniques. It is used for both encoding and decoding. It uses a greedy approach to solve problems. The greedy approach allows us to solve a problem and find an optimal solution using the optimal substructure. An optimal substructure to a problem is achieved if an optimal solution can be created from the optimal solutions of its subproblems.

The Huffman coding uses a prefix rule that avoids ambiguity while decoding. The two steps involved in Huffman coding are:

* Construct a Huffman tree from the input string or text or characters.
* Assigning a Huffman code to each character by traversing the tree.

In this report, we will describe the Huffman coding algorithm, including its implementation and experimental results.

**I.2. Objective**

The main objective of Huffman coding is to minimize the total number of bits required to represent a message while maintaining the integrity of the information. By using variable-length codes, symbols that appear more frequently in a message are assigned shorter codes, which results in a significant reduction in the overall size of the message.

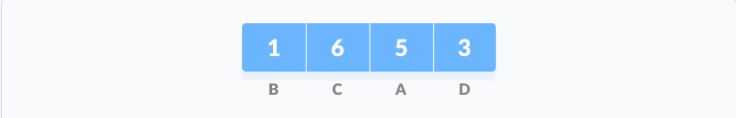
**II. Algorithm:**

**II.1. Describe the algorithm:**

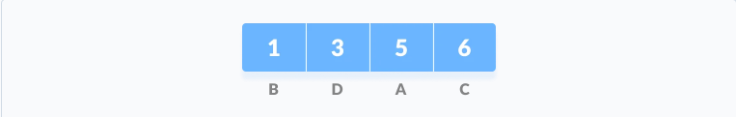
Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.

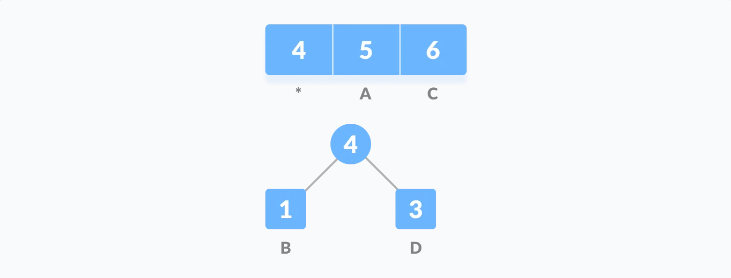
For example , we have a string:



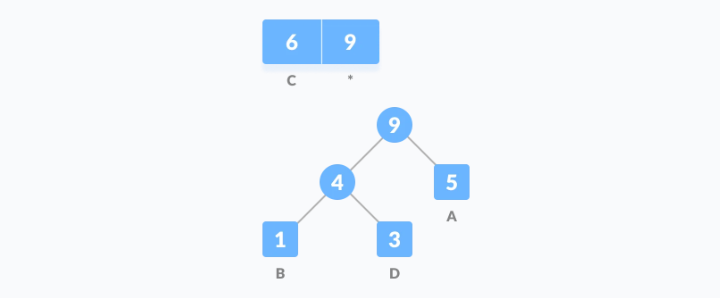
1. Sort the characters in increasing order of the frequency. These are stored in a priority queue

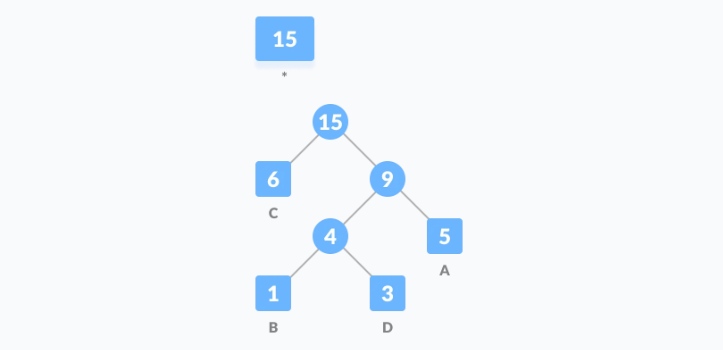


1. Make each unique character as a leaf node.
2. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

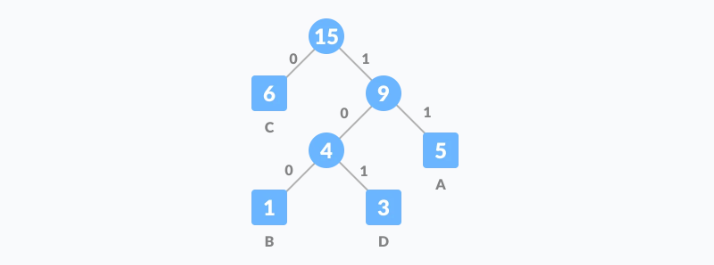


1. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (\* denote the internal nodes in the figure above).
2. Insert node z into the tree.
3. Repeat steps 3 to 5 for all the characters.

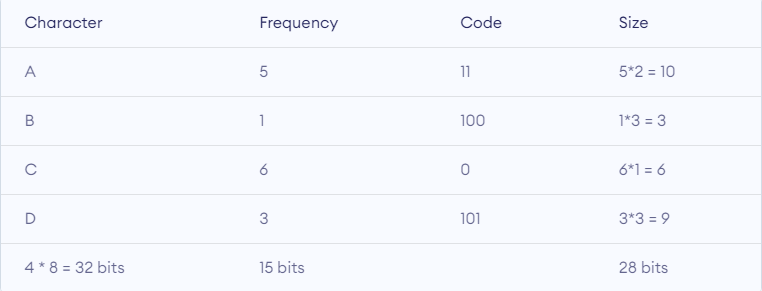




1. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.



Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28= 75

**Decoding the code**

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.

A screenshot of a computer

Description automatically generated with low confidence

**II.2. Flowchart**

NO

YES

NO

YES

**II.3 Explain the algorithm**

1. Count the frequency of each character in the input string.

The first step of the Huffman coding algorithm is to count the frequency of each character in the input string. This is done by iterating through the string and keeping track of how many times each character appears.

1. Create a priority queue to store live nodes of the Huffman tree.

Next, we create a priority queue to store the live nodes of the Huffman tree. The priority queue is implemented as a min-heap, where the node with the lowest frequency has the highest priority.

1. Create a leaf node for each character and add it to the priority queue.

For each character in the input string, we create a leaf node with the character and its frequency, and add it to the priority queue.

1. Remove the two nodes of highest priority (lowest frequency) from the queue.

We then repeatedly remove the two nodes with the highest priority (lowest frequency) from the priority queue.

1. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequencies.

For each pair of nodes that we remove from the priority queue, we create a new internal node with these two nodes as children. The frequency of the new internal node is equal to the sum of the frequencies of the two child nodes.

1. Repeat steps 4-5 until there is only one node left in the queue, which is the root of the Huffman tree.

We repeat steps 4-5 until there is only one node left in the priority queue, which is the root of the Huffman tree.

1. Traverse the Huffman Tree and store Huffman Codes.

Once we have constructed the Huffman tree, we traverse it to assign unique binary codes to each character based on their frequency. We start at the root of the tree and recursively traverse down the left and right branches, assigning a '0' to each left branch and a '1' to each right branch. The resulting binary code for each character is the sequence of '0's and '1's from the root to the leaf node representing that character.

1. Encode the input string using the Huffman codes.

We then encode the input string using the Huffman codes we just generated. We replace each character in the input string with its corresponding Huffman code, resulting in a compressed binary string.

1. Print various statistics about the Huffman codes and frequencies.

We print various statistics about the Huffman codes and frequencies, such as the total number of bits used to encode the input string and the compression ratio achieved.

1. Decode the encoded string back to the original input string using the Huffman codes.

Finally, we decode the compressed binary string back to the original input string using the Huffman codes. We start at the root of the Huffman tree and traverse down the tree, following the '0's and '1's in the encoded string until we reach a leaf node representing a character. We then output that character and continue decoding the rest of the string until we have reconstructed the original input string.

**IV. Implementation result:**

**IV.1. Experimental result:**

**A screenshot of a computer

Description automatically generated with medium confidence**

**A screenshot of a computer

Description automatically generated**

**IV.2. Source code:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The code is implementing a Huffman coding algorithm, which is a lossless data compression technique.

The program takes an input string and compresses it using Huffman coding, which assigns shorter codes to more frequently occurring characters in the input string.

The program first counts the frequency of each character in the input string and creates a binary tree where each leaf node represents a character and its frequency.

It then traverses the tree to assign unique binary codes to each character based on their frequency.

The program then encodes the input string using the Huffman codes and prints the encoded string to the console.

It also prints various statistics about the Huffman codes and frequencies, such as the total number of bits used to encode the input string

and the compression ratio achieved.

Finally, the program decodes the encoded string back to the original input string using the Huffman codes and prints it to the console.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <queue>

#include <unordered\_map>

#include <fstream>

#include <vector>

#include <algorithm>

#include <string>

#include <iomanip>

using namespace std;

const int MAX\_CONSOLE\_COLUMNS = 200;

const int ERROR\_FILE\_OPEN = 1;

// A Tree node

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;

}

// Comparison object to be used to order the heap

struct comp {

    bool operator()(Node\* l, Node\* r)

    {

        return l->freq > r->freq;

    }

};

// Traverse the Huffman Tree and store Huffman Codes

void encode(Node\* root, string str,

            unordered\_map<char, string>& huffmanCode,

            vector<pair<char, string>>& order)

{

    if (root == nullptr)

        return;

    if (!root->left && !root->right) {

        huffmanCode[root->ch] = str;

        order.push\_back(make\_pair(root->ch, str));

    }

    encode(root->left, str + "0", huffmanCode, order);

    encode(root->right, str + "1", huffmanCode, order);

}

// Traverse the Huffman Tree and decode the encoded string

void decode(Node\* root, int& index, string str)

{

    if (root == nullptr) {

        return;

    }

    // found a leaf node

    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);

}

int countTotalCharacters(string text) {

    int count = 0;

    for (char ch : text) {

        count++;

    }

    cout << "\nTotal number of characters: " << count << endl;

    return count;

}

void printError(string message) {

    cerr << "Error: " << message << endl;

}

// Builds Huffman Tree and decodes the input text

void buildHuffmanTree(string text)

{

    // count frequency of appearance of each character

    unordered\_map<char, int> freq;

    for (char ch : text) {

        if (!isascii(ch)) {

            printError("Input text contains non-ASCII characters.");

            return;

        }

        freq[ch]++;

    }

    countTotalCharacters(text);

    // Create a priority queue to store live nodes of the Huffman tree

    priority\_queue<Node\*, vector<Node\*>, comp> pq;

    // Create a leaf node for each character and add it to the priority queue.

    for (auto pair : freq) {

        pq.push(getNode(pair.first, pair.second, nullptr, nullptr));

    }

    // do till there is more than one node in the queue

    while (pq.size() != 1)

    {

        // Remove the two nodes of highest priority (lowest frequency) from the queue

        Node\* left = pq.top(); pq.pop();

        Node\* right = pq.top();    pq.pop();

        // Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequencies.

        int sum = left->freq + right->freq;

        pq.push(getNode('\0', sum, left, right));

    }

    // root stores pointer to root of Huffman Tree

    Node\* root = pq.top();

    // Traverse the Huffman Tree and store Huffman Codes

    unordered\_map<char, string> huffmanCode;

    vector<pair<char, string>> order;

    unordered\_map<int, int> bitCountMap;

    unordered\_map<int, int> freqBitCountMap;

    encode(root, "", huffmanCode, order);

    // Sort the order vector by frequency

    sort(order.begin(), order.end(), [&](pair<char, string>& a, pair<char, string>& b) {

        return freq[a.first] < freq[b.first];

    });

    // Calculate the bit count and frequency of each Huffman code

    for (auto pair : huffmanCode) {

        bitCountMap[pair.second.length()]++;

        freqBitCountMap[pair.second.length()] += freq[pair.first];

    }

    // Find the maximum bit count

    int maxBitCount = 0;

    for (auto pair : freqBitCountMap) {

        if (pair.first > maxBitCount) {

            maxBitCount = pair.first;

        }

    }

    // Print out the characters with their Huffman codes and frequencies in order of frequency

    cout << "\nHuffman Codes and Frequencies are (in order of frequency):\n" << endl;

    for (auto pair : order) {

        cout << setw(1) << pair.first << "  ";

        cout << setw(maxBitCount) << pair.second << "  ";

        cout <<"("<< freq[pair.first]<< ")" << endl;

    }

    // Print out the bit count, number of characters, and frequency for each group of characters with the same bit count

    cout << "\nBits, Number, and Frequency of characters with the same bit count: " << endl;

    for (auto pair : freqBitCountMap) {

        int bitCount = pair.first;

        int frequency = pair.second;

        int numChars = bitCountMap[bitCount];

        cout << "Bits: " << bitCount << "  Numbers: " << numChars << "  Frequency: " << frequency << endl;

    }

    cout << "\nThe original string is :\n" << text << endl;

    // print encoded string

    string str = "";

    for (char ch : text) {

        str += huffmanCode[ch];

    }

    cout << "\nThe encoded string is :\n" << str << endl;

    // decode the encoded string

    int index = -1;

    cout << "\nThe decoded string is :\n";

    while (index < (int)str.size() - 2) {

        decode(root, index, str);

    }

    //

    string encodedText = "";

    for (char c : text) {

        encodedText += huffmanCode[c];

    }

    // Calculate compression ratio

    double compressionRatio = (double)encodedText.length() / (text.length() \* 8) \* 100;

    // Print the bits before and after encoding, as well as the compression ratio

    cout << "\n\nBits before encoding: " << text.length() \* 8 << endl;

    cout << "Bits after encoding: " << encodedText.length() << endl;

    cout << "Compression ratio: " << compressionRatio << "%" << endl;

}

// Driver program to test above functions

int main()

{

    ifstream file("input.txt");

    if (!file) {

        printError("Could not open input file.");

        return ERROR\_FILE\_OPEN;

    }

    string text;

    getline(file, text);

    buildHuffmanTree(text);

    cout << "\nPress Enter to exit...";

    cin.get(); // Wait for user to press Enter

    return 0;

}

**V.Conclusion**

We have thoroughly discussed the Huffman coding approach (Greedy Design Technique) and its algorithm. Now let us see the applications of the Huffman coding mechanism, which are as follows:

* It is applied where a series of frequently occurring characters are used.
* Used for transmitting the data in the form of text or fax etc.
* Used by conventional compression formats like GZIP, PKZIP, etc.
* Multimedia formats like JPEG, PNG, and MP3 use Huffman encoding.

About complexity

The algorithm works by creating a binary tree where each leaf node represents a character and the path from the root to the leaf node represents the code for that character. The code for a character is determined by traversing the tree from the root to the leaf node corresponding to that character.

The complexity of the Huffman coding algorithm is O(nlogn), where n is the number of characters in the text. This is because the algorithm involves building a binary tree that has n nodes, and each node requires logn operations to be processed.

Despite the relatively high complexity, Huffman coding is widely used in data compression because it can achieve significant compression ratios while maintaining good performance.

**VI.References**

Huffman, D. A. (1952). *"A method for the construction of minimum-redundancy codes".*

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. (2009). *Introduction to algorithms.*