A Mini Project Report

on

DATA COMPRESSION USING HUFFMAN CODING

In Subject: Data Structures & Discrete Mathematics

By

Name	Roll No.	PRN no.
Rhishikesh Ravindra Sonawane	272053	22010585
Siddhesh Ramesh Songire	272054	22010862
Omsai Shankar Zadi	272065	2201132

Department of Artificial Intelligence and Data Science

VIIT

2020-2021

Contents

INTRODUCTION:

DATA COMPRESSION:

- Data compression is a process of encoding, restructuring, or otherwise modifying data in order to reduce its size. Fundamentally, it involves re-encoding information using fewer bits than the original representation.

- Compression is done by a program that uses functions or an algorithm to effectively discover how to reduce the sixe of the data.

- Text compression can usually succeed by removing all unnecessary characters, instead of inserting a single character a reference for a string of repeated characters then replacing a smaller bit string for a more common bit string.

ONE SUCH ALGORITHM TO CARRY OUT DATA COMPRESSION IS:

HUFFMAN CODING:

- Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

- The most frequent character gets the smallest code, and the least frequent character gets the largest code.

- The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character.

- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

HOW DOES HUFFMAN CODING WORK?

- Consider the following table containing the characters and their respective frequencies:

CHARACTERS	FREQUENCY
A	10
E	15
I	12
O	3
U	4
S	13
T	1
Total Characters:	58

- Each character occupies 8 bits. There are a total of 58 characters in the above string. Thus, a total of 8*58 = 464 bits is required to send the string.

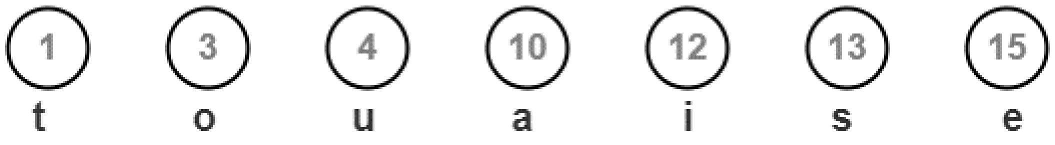
- Using the Huffman coding technique, we can compress the string to a smaller size.

- Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

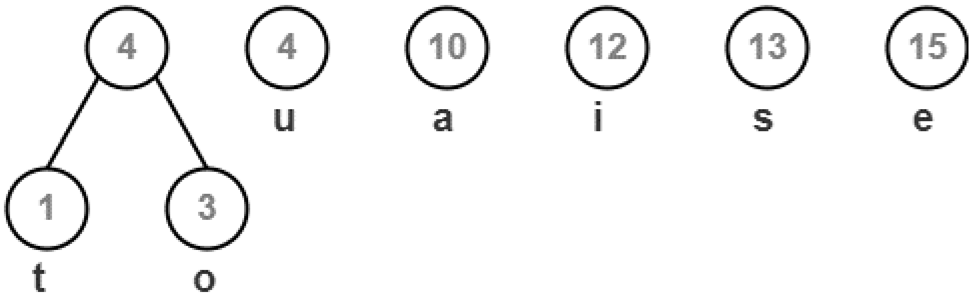
HOW IS HUFFMAN CODING DONE?

Consider the same table shown before:

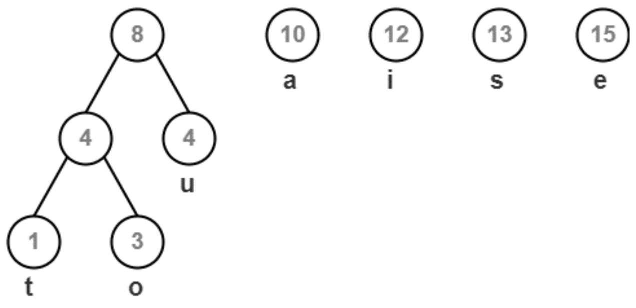
- STEP-1 Calculate the frequency.
- STEP-2 Sort the characters in increasing order of the frequency. These are stored in a priority queue Q as shown below:



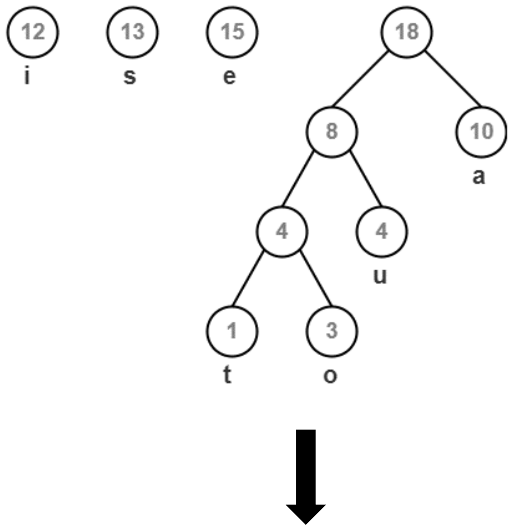
- STEP-3 Create a empty node and Assign the minimum frequency to the left child of the node and assign the second minimum frequency to the right child of the node. Set the value of the node as the sum of the above two minimum frequencies.
- STEP-4 Remove these two minimum frequencies from Queue and add the sum into the list of frequencies.



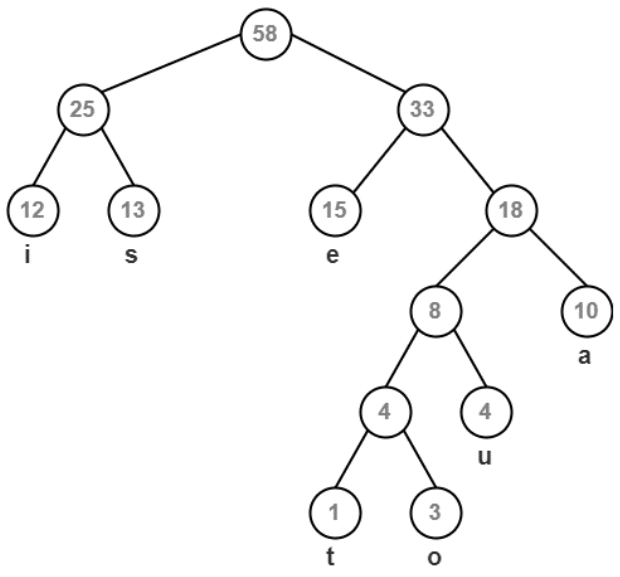
- STEP-5 Insert this node into the tree.



- STEP-6 Repeat steps 3 and 4 for all the characters.

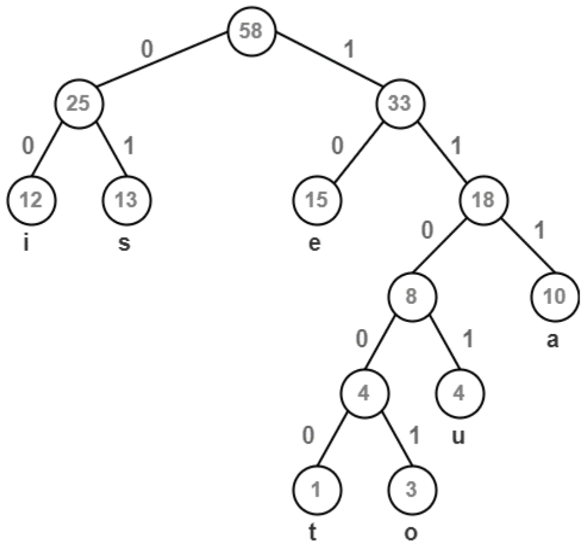


- The final tree will look like this.



- STEP-4 Finally for each non-leaf, assign 0 to the left edge and 1 to the right edge.

- After assigning weight to all the edges, the modified Huffman Tree is-



HUFFMAN CODE FOR CHARACTERS

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character. Following this rule, the Huffman Code for each character is-

A = 111

E = 10

I = 00

O = 11001

U = 1101

S = 01

T = 11000

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.

CHARACTERS	**FREQUENCY**	**CODE**	**SIZE**
A	**10**	**111**	**10*3=30**
E	**15**	**10**	**15*2=30**
I	**12**	**00**	**12*2=24**
O	**3**	**11001**	**3*5=15**
U	**4**	**1101**	**4*4=16**
S	**13**	**01**	**13*2=26**
T	**1**	**11000**	**1*5=5**

In the above table we can see that all the characters are having prefix-free code words.

COMPARISON:

Calculating the total size, Before Huffman Encoding,

- Each character 8 bits.
- Total 58 characters.
- Thus, a total of 8*58 = 464 bits

Calculating the total size, after Huffman Encoding

- 30 + 30 + 24 + 15 + 16 + 26 + 5 = 146 BITS
- Total size is reduced by 464 - 146= 318 BITS
- Size is reduced by 68.53%

ADVANTAGES OF HUFFMAN ENCODING-

- It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string
- The binary codes generated are prefix-free

DISADVANTAGES OF HUFFMAN ENCODING-

- Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images.
- Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.
- Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output.

REAL-LIFE APPLICATIONS OF HUFFMAN ENCODING-

- Huffman encoding is widely used in compression formats like GZIP, PKZIP (WinZip) and BZIP2.
- Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes)
- Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.

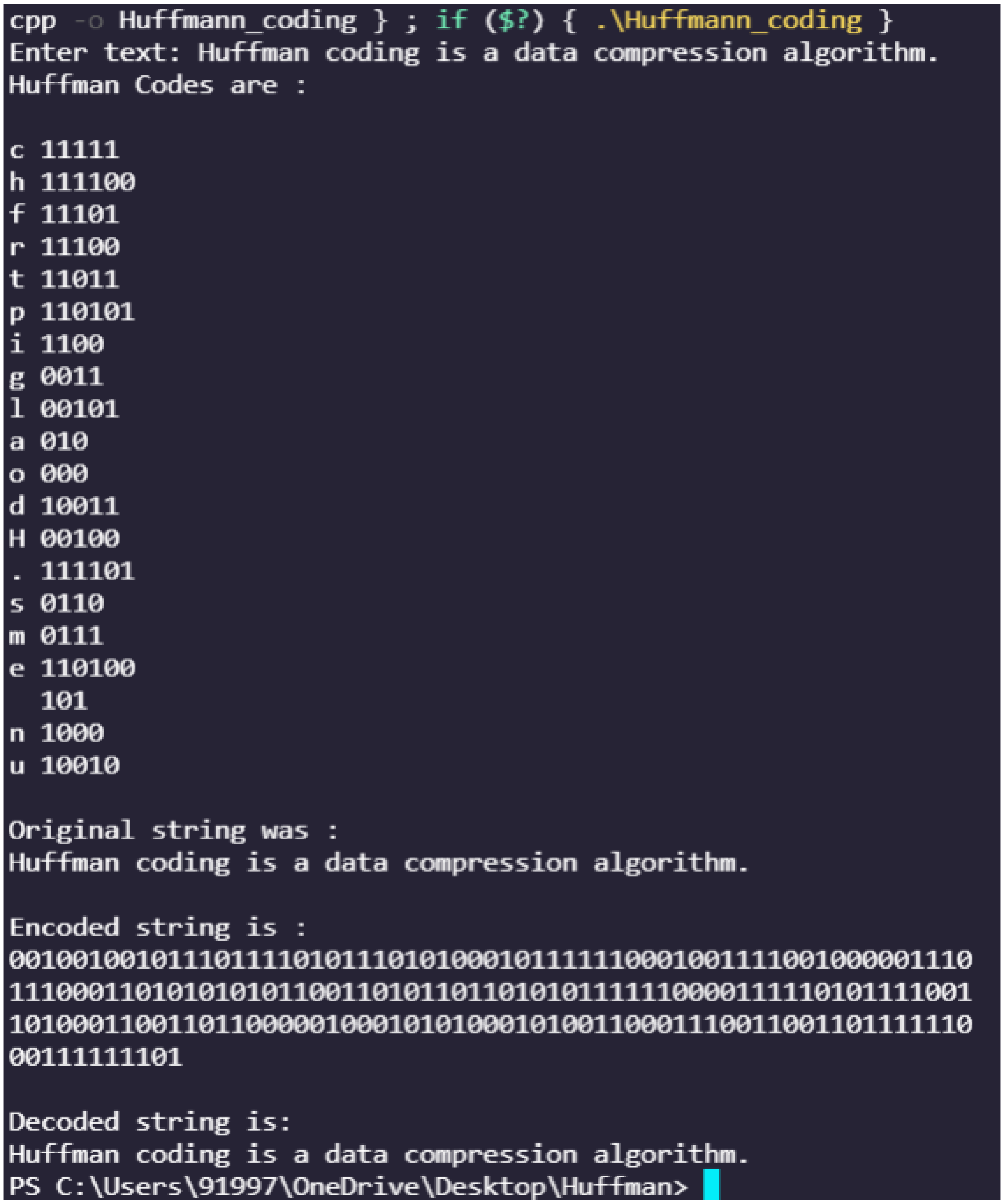
CODE

```
// * C++ code to implement Huffman coding

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

// A Tree node
struct Node
{
    char ch;
    int freq;
    Node *left, *right; // Left and right child of this node
};

// Function to allocate a new tree node
Node *getNode(char ch, int freq, Node *left, Node *right) // will return pointer
of node type
{
    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) // takes two pointer left and right
    {
        // highest priority item has lowest frequency
        return l->freq > r->freq; // returns the lowest frequency
    }
};

//*************** Encode Function ********************
```

```cpp
// traverse the Huffman Tree and store Huffman Codes
// in a map.
void encode(Node *root, string str,
            unordered_map<char, string> &huffmanCode)
{
    if (root == nullptr)
        return;

    // found a leaf node
    if (!root->left && !root->right)
    {
        huffmanCode[root->ch] = str;
    }

    encode(root->left, str + "0", huffmanCode);
    encode(root->right, str + "1", huffmanCode);
}

//********************** Decode function *************************
// 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);
}

//*******************************************************************************
*****************************
```

```
// Builds Huffman Tree and decode given input text
void buildHuffmanTree(string text)
{
    // count frequency of appearance of each character and store it in a map
    unordered_map<char, int> freq; // declaration
    for (char ch : text)
    {
        freq[ch]++;
    }

    // Create a priority queue to store leaf nodes of Huffman tree;
    priority_queue<Node *, vector<Node *>, comp> pq;
    // we are creating here min heap

    // 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));
    }

    //* Next step is to pop the two lowest frequency character and add them
together

    // do till there is more than one node in the queue
    while (pq.size() != 1)
    {
        // Remove the two nodes of (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. Add the new node
        // to the priority queue.
        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
    // in a map. Also prints them
    unordered_map<char, string> huffmanCode;
    encode(root, "", huffmanCode);

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

    cout << "\nOriginal string was :\n"
         << text << '\n';

    // print encoded string
    string str = "";
    for (char ch : text)
    {
        str += huffmanCode[ch];
    }

    cout << "\nEncoded string is :\n"
         << str << '\n';

    // traverse the Huffman Tree again and this time
    // decode the encoded string
    int index = -1;
    cout << "\nDecoded string is: \n";
    while (index < (int)str.size() - 2)
    {
        decode(root, index, str);
    }
}

// Huffman coding algorithm
int main()
{

    // string text = "Huffman coding is a data compression algorithm.";
    string text;
    cout << "Enter text: ";
    getline(cin, text);
    buildHuffmanTree(text);
    return 0;
}
```

OUTPUT:



```
cpp -o Huffmann_coding } ; if ($?) { .\Huffmann_coding }
Enter text: Huffman coding is a data compression algorithm.
Huffman Codes are :

c 11111
h 111100
f 11101
r 11100
t 11011
p 110101
i 1100
g 0011
l 00101
a 010
o 000
d 10011
H 00100
. 111101
s 0110
m 0111
e 110100
  101
n 1000
u 10010

Original string was :
Huffman coding is a data compression algorithm.

Encoded string is :
0010010010111011110101110101000101111110001001111001000001110
1110001101010101011001101011011010101111110000111110101111001
1010001100110110000010001010100010100110001110011001101111110
00111111101

Decoded string is:
Huffman coding is a data compression algorithm.
PS C:\Users\91997\OneDrive\Desktop\Huffman>
```

REFERENCES:

- https://www.cplusplus.com/reference/stl/
- https://www.geeksforgeeks.org/huffman-decoding/?ref=lbp
- https://www.programiz.com/dsa/huffman-coding
- https://cppsecrets.com/users/99991091111171101059711710350484848641031099710510846991111109/C00-Greedy-Approach-Huffman-Codes.php
- https://www.barracuda.com/glossary/data-compression