Tutorial - 1

Name:- Sairaj prakash rajput

Division:- TY-A

Roll no – 331049

PRN - 22111025

Aim - to research and put into practice the popular text compression algorithm known as Huffman coding. Additionally, the efficiency with which text data can be shrunk without sacrificing its original content.

Prerequisite Knowledge:

- The fundamentals of data structures like linked lists, dictionaries, and arrays.
- Knowledge of how to manipulate strings and match patterns.
- Knowledge of programming languages such as Java, C++, or Python.
- An understanding of binary representation and bit manipulation.

Theory :-

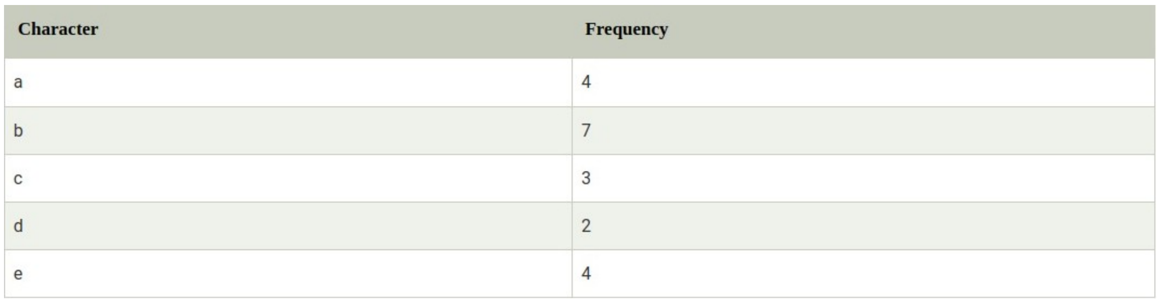
1. Known as Huffman coding after its creator, David A. Huffman, it is a commonly used lossless data compression technique. Numerous widely used compression algorithms, including JPEG (image compression) and ZIP (file compression), are based on it.
2. Prefix Codes: One kind of prefix coding is Huffman coding, in which no codeword is a prefix of any other codeword. This characteristic makes sure that the compressed data can be decoded without any uncertainty.
3. Frequency Analysis: The fundamental principle of Huffman coding is to give longer codes to symbols that occur less frequently and shorter codes to symbols that occur more frequently. Analyzing the input data's symbol frequency distribution allows for this to be achieved.
4. Optimal Prefix Codes: Huffman coding yields the most efficient representation of the input data in terms of the average length of the encoded symbols, which is known as an optimal prefix code. This guarantees that, for

a given input, Huffman coding achieves the highest possible compression ratio.

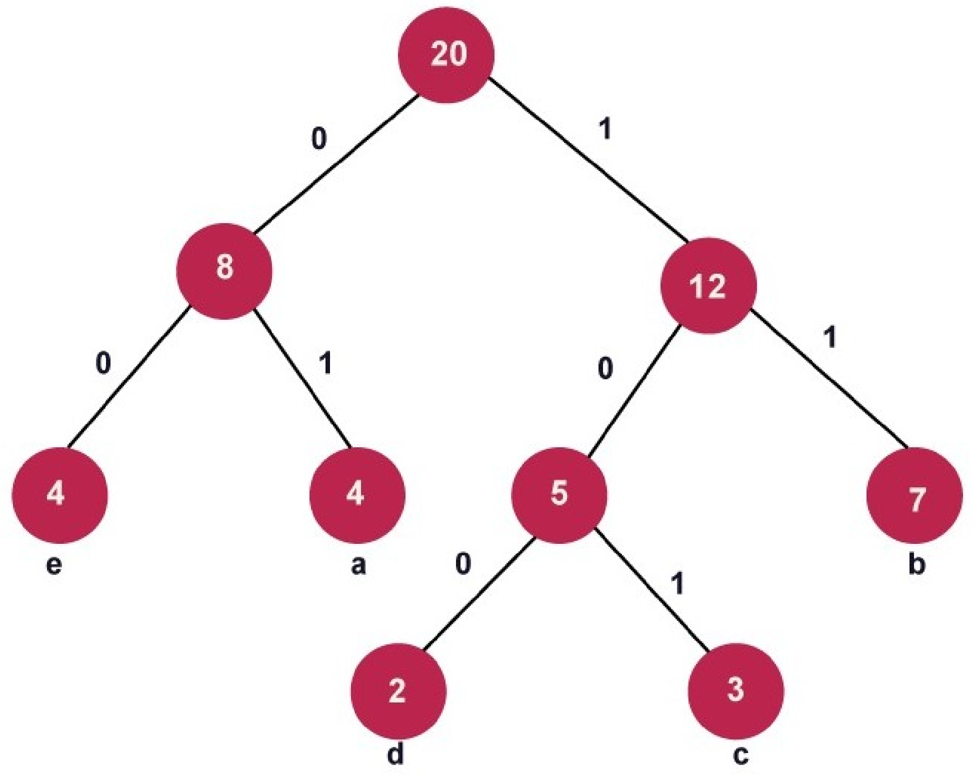
Algorithm:

- To find out how often each character appears in the input text, do a frequency analysis.
- Using the frequencies, build a Huffman tree.
- Assign each character a binary code by going through the Huffman tree.
- Using the generated Huffman codes, encode the input text.
- Use the Huffman tree to decode the encoded text back to the original.

Example:



Character	Frequency
a	4
b	7
c	3
d	2
e	4



e => 00

a => 01

b => 11

d => 100

c => 101

Conclusion:

By allocating longer codes to less frequent characters and shorter codes to more frequent characters, Huffman coding provides an effective method of data compression.

Code:

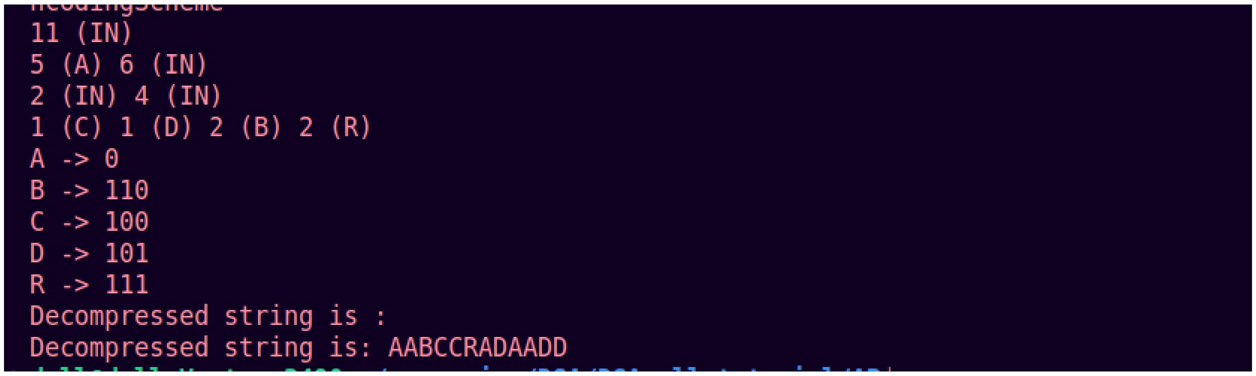
```
#include<bits/stdc++.h>

using namespace std;

class Node{
public:
string character;
int frequency;
Node* left;
Node* right;
Node(string character,int frequency){
this -> character = character;
this -> frequency = frequency;
this -> left = NULL;
this -> right = NULL;
}
};


class cmp{
public:
bool operator()(Node* a,Node* b){
return (a->frequency > b->frequency);
}
};


class HaufmanCoding{
public:
map<string,int> freq;
priority_queue<Node*,vector<Node*>,cmp> q;

void initialiseMap(string x){
for(char m : x){
// cout << string(1,m) << endl;
freq[string(1,m)] ++;
}
}

void initialiseMinHeap(){
for(auto it : freq){
Node* newNode = new Node(it.first,it.second);
q.push(newNode);
```

```
}
}

Node* constructHaufmanTree(string s){
initialiseMap(s);
initialiseMinHeap();
// now we got the filled priority queue
while(q.size() > 1){
Node* min1 = q.top();
q.pop();
Node* min2 = q.top();
q.pop();

Node* newNode = new Node("IN",(min1 -> frequency + min2 -> frequency));
newNode -> left = min1;
newNode -> right = min2;
q.push(newNode);
}
// cout << "run succesfully" << q.top()->frequency<< endl;
return q.top();
}

void getEncode(string temp,Node* ans,string answer,string &x){
if(ans->left == NULL and ans -> right == NULL and ans -> character != temp){
return;
}

if(ans -> left == NULL and ans -> right == NULL and ans -> character == temp){
x = answer;
return;
}

// answer = answer + "0";
getEncode(temp,ans->left,answer+"0",x);
// answer = answer + "1";
getEncode(temp,ans->right,answer+"1",x);
}

map<string,string> getAns(map<string,int> freq){
map<string,string> temp;
for(auto it : freq){
Node* topQ = q.top();
string ansS = "";
getEncode(it.first,topQ,"",ansS);
temp[it.first] = ansS;
```

```
}
return temp;
}

void decode(Node* root, string encoded, string &decoded, int &index) {
if (root == nullptr)
return;

// If leaf node is found
if (!root->left && !root->right) {
decoded += root->character;
return;
}

index++;

if (encoded[index] == '0')
decode(root->left, encoded, decoded, index);
else
decode(root->right, encoded, decoded, index);
}

string decompress(string encoded, Node* root) {
string decoded = "";
int index = -1;
while (index < (int)encoded.size() - 2) {
decode(root, encoded, decoded, index);
}
return decoded;
}

};

int main(){
// enter the project name
string name = "ABRACADABRA";
HaufmanCoding h;
Node* ans = h.constructHaufmanTree(name);
queue<Node*> pq;
pq.push(ans);
while(!pq.empty()){
int size = pq.size();
for(int i = 0; i < size; i ++){
Node* temp = pq.front();
pq.pop();
```

```
cout << temp -> frequency << " ("<<temp->character<<") ";
if(temp -> left != NULL){
pq.push(temp -> left);
}

if(temp -> right != NULL){
pq.push(temp -> right);
}
}
cout << endl;
}

map<string,string> answer = h.getAns(h.freq);
for(auto it : answer){
cout << it.first << " -> " << it.second << endl;
}

cout << "Decompressed string is : " << endl;
string encoded = "0011010010011101010010110 10";
string decoded = h.decompress(encoded, ans);
cout << "Decompressed string is: " << decoded << endl;

return 0;
}
```

Output/Input:



```
11 (IN)
5 (A) 6 (IN)
2 (IN) 4 (IN)
1 (C) 1 (D) 2 (B) 2 (R)
A -> 0
B -> 110
C -> 100
D -> 101
R -> 111
Decompressed string is :
Decompressed string is: AABCCRADAADD
```