**HUFFMAN CODING**

Message Compression reduces the size of the string significantly. It is useful in situation like Communication over low bandwidth channels.

Standard Encoding schemes such as, ASCII or UNICODE uses fixed length binary strings to encode character (8 bits for ASCII & 16 bits for UNICODE). Whereas Huffman Coding uses variable length encoding optimise for the particular string “X”. The Optimisation is based on the character frequencies. The frequency represents the number of a character appears in the string.

Message: **BCCABBDDAECCBBAEDDCC**

We will be using this message for the rest of this presentation.

Length of the Message = **20**

If we store this message or send this message using ASCII code, then;

|  |  |  |
| --- | --- | --- |
| Character | Number | 8-bit Representation |
| A | 65 | 01000001 |
| B | 66 | 01000010 |
| C | 67 | 01000011 |
| D | 68 | 01000100 |
| E | 69 | 01000101 |

Therefore for the above message we need 8\*20 bits = 160 bits.

Now we need 8 bits of code to represent 5 alphabets.

We can use our own code to do so. i.e.’ we may use 3-bit representation of the character & share it. This is known as Fixed-Size codes or Fixed-Length codes.

Message: **BCCABBDDAECCBBAEDDCC**

|  |  |  |
| --- | --- | --- |
| Character | Frequency/Count | Code |
| A | 3/20 | 000 |
| B | 5/20 | 001 |
| C | 6/20 | 010 |
| D | 4/20 | 011 |
| E | 2/20 | 100 |

Now, total cost for the message (with 20 characters) we need 20\*3 bits = 60 bits.

But, if we send this message to someone then we also have to send the lookup table which will tell the receiver which code represents what.

Now, we represent the character we use ASCII codes.

Therefore, total cost = 5 character \* 8 bits = 40 bits.

& the code will take = 5 character \* 3 bits = 15 bits.

Therefore, the table will take = (40 + 15) bits = 55 bits.

Thus if we send the message using the fixed-length code then we need –

= (1) Message encoded + (2) lookup table

= (60 + 55) bits

**= 115 bits.**

This is still less than the original message used in the ASCII.

So, using fixed-length code we are able to reduce 35 – 40 per cent cost.

But we can reduce it further using Huffman Coding which uses variable-length code.

HUFFMAN CODING:

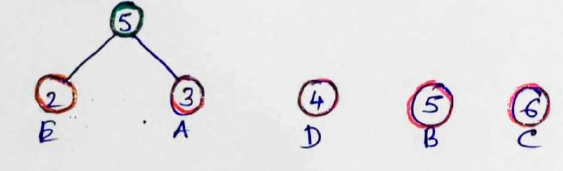
This is to use small size code for the frequently appearing characters.

Message: **BCCABBDDAECCBBAEDDCC**

Char: **A B C D E**

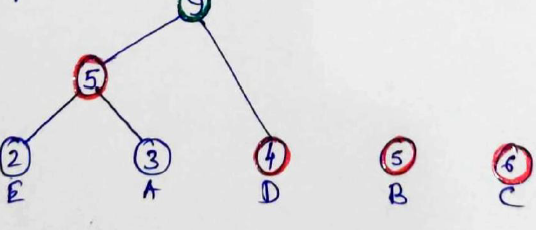
Count: **3 5 6 4 2**

1. Arrange the nodes in the increasing order of count and select the smallest pair to create a new node.

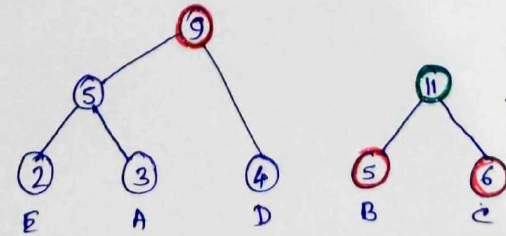


(Selected smallest two nodes among circled nodes and add them to create a new node marked as green)

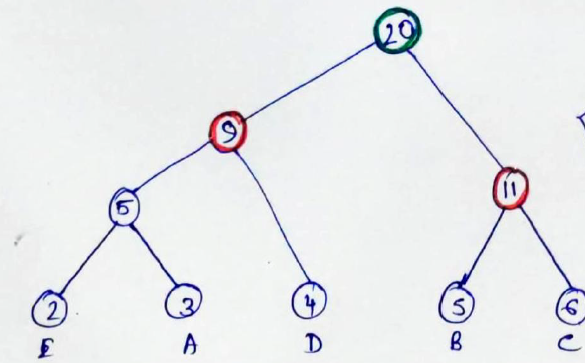
1. Repeat the process until we are forming a binary tree.



(Selected smallest two nodes among circled nodes to create a new node marked as green)

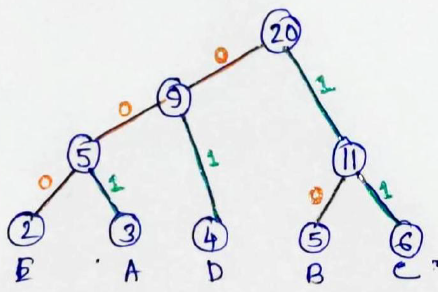


(Selected smallest two nodes among circled nodes to create a new node marked as green)



(Selected smallest two nodes among circled nodes to create a new node marked as green)

1. Tree formation is complete. Now mark the left hand side edges ‘0’ & the right hand side edges as ‘1’.



|  |  |  |  |
| --- | --- | --- | --- |
| Character | Count | Code | Size |
| A | 3 | 001 |  |
| B | 5 | 10 |  |
| C | 6 | 11 |  |
| D | 4 | 01 |  |
| E | 2 | 000 |  |
|  |  |  | Total = 45 bits |

Therefore, Size of the message = 45 bits.

But to decode we also have to send the lookup table to represent the characters = 5 characters \* 8 bits = 40 bits.

& the code needs = (3 + 5 + 6 + 4 + 2) bits = 12 bits.

Therefore, the lookup table will need = (40 + 12) bits = 52 bits.

Therefore, the message needs

= (1) size of the encoded message + (2) lookup table

= (45 + 52) bits = 97 bits.

**HUFFMAN ALGORITHM**: Huffman (X):

Input: String X of length n and width d distinct characters

Output: Coding tree for X

Compute the frequency f(c) of each character c of X

Initialize a priority queue Q

For each character c in X do

{

Create a single node binary tree T storing C

Insert T into Q with key f(c)

}

While Q.Size ()>1 do

{

f1 <- Q.minkey()

t1 <- Q.removemin()

f2 <- Q.minkey()

t2 <- Q.removemin()

Create a new binary tree T with left subtree T1 & right subtree T2

Insert T into Q with key f1 + f2

}

return tree Q.removemin()

**Huffman Coding in C++:**

#include<iostream>

#include<algorithm>

#include<vector>

#include<queue>

#include<string>

class HuffmanCodes

{

struct Node

{

char data;

size\_tfreq;

Node\*left;

Node\*right;

Node(char data, size\_tfreq) : data(data),

freq(freq),

left(nullptr),

right(nullptr),

{}

Node()

{

delete left;

delete right;

}

};

struct compare

{

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

{

return(l->freq > r->freq);

}

};

Node\*top;

void printCode(Node\*root,std::string str)

{

if(root == nullptr)

return;

if(root->data == ‘$’)

{

printCode(root->left, str + “0”);

printCode(root->right, str + “1”);

}

if(root->data! = ‘$’)

{

std::cout <<root->data << “ : ” << str <<”\n”;

printCode(root->left, str + “0”);

printCode(root->right, str + “1”);

}

}

public:

HuffmanCodes() {};

HuffmanCodes()

{

delete top;

}

void GenerateCode(std::vector<char>&data, std::vector<size\_t>&freq, size\_t size)

{

Node\*left;

Node\*right;

std::priority\_queue<Node\*, std::vector<Node\*>,compare > minHeap;

for(size\_t i = 0; i < size; ++i)

{

minHeap.push(newNode(data[i], freq[i]));

}

while(minHeap.size() ! = 1)

{

left = minHeap.top();

minHeap.pop();

top = new Node(‘$’, left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

printCode(minHeap.top(), “”);

}

};

int main()

{

HuffmanCodes set1;

std::vector<char>data({‘d’, ‘e’, ‘b’, ‘c’, ‘a’, ‘f’});

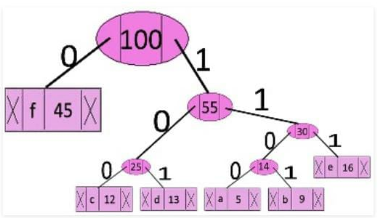
std::vector<size\_t>freq({16, 9, 13, 12, 45, 5});

size\_t size = data.size();

set1.GenerateCode(data, freq, size);

return 0;

}



The codes are as follows:

|  |  |
| --- | --- |
| Character | Code-word |
| f | 0 |
| c | 100 |
| e | 101 |
| a | 1100 |
| b | 1101 |
| e | 111 |

Time Complexity: O(logn)

Where,

n = Number of unique characters

If there are n nodes, extractMin() is called 2\*(n-1) times.extractMin() takes O(logn) time as it calls minHeapify(). So, overall complexity is O(nlogn).