UNITED INTERNATIONAL UNIVERSITY

HUFFMAN CODING ALGORITHM

Submitted By

Name	ID
Sadia Islam	011193156
Md. Ahasan Khan	011201122
MD.Shahidul Islam Sujon	011193107
Ovie Rahaman Sheikh	011201306

GROUP E1

Data Structure & Algorithms II

Section – E **Dept. -** CSE

Table of Contents

Introductions of Huffman coding

Huffman coding is a type of prefix coded algorithm that is used to encode or decode data. One of the most common applications of this algorithm is lossless data compression.

Inventor
The process of finding or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes, which he published in 1952.

Application

1. Cryptography
2. Compressing files to PKZIP, MP3, JPEG etc.

Principle
The principle behind Huffman coding is to assign variable-length codes to input characters and the lengths of the codes are dependent on the frequency of the associated characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.
There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Travers the Huffman Tree and assign codes to characters.

Huffman Coding

Quarks of Huffman Coding

We've learned about Huffman coding and what its used for, now we will learn how the algorithm works. But before that we need to know about its special quarks. Huffman coding has 2 special quarks,

- Prefix Code
- The more the upper

I know, the names are confusing specially the second one, after all I am the one who came up with that name. But we will discuss about all of these now.

Prefix Code

In short prefix code is a type of coding where *one codeword cannot become the prefix of another codeword.* Let's see this with an example, Let’s say we have only two characters **A** and **B**. Now, we want to represent these with binary numbers.

If we say,

A = 01

Then,

B = 010

B = 011

are not possible. Because both of them starts with 01 which is the codeword of A. However, other than codewords that starts with 01 all are valid for B. like, 110, 00, 111, 001 etc.

	✗	✓	✓	✓	✓
A	B	B	B	B	B
01	010	110	00	001	111

The interesting part is that these quark being applied automatically if we follow the algorithm through a binary tree which we will see soon.

The More The Upper

This quark mainly referring that, characters with more frequency will be upper in the tree. The more frequency one character has the more upper it will be placed on. **Now, the question is why is this a big deal?**

The thing is, whenever we encrypt or decrypt a text with the Huffman tree we have to traverse the tree for each character to find it’s codewords. Now, the upper a character will be on the less time it will take to find it from tree which reduces the encrypt and decrypt time. This is not all, *the main benefit* we will get from this quark is that the upper a character stay the less bit it’s codeword will use. For, example if a character directly connected with the root its codeword will be either 0 or 1 in other words 1 bit. If this same character had 3 ancestors, then it’s codeword side would’ve been 3 bit which means 3 times more. Doesn’t look like a lot, right?

Now, let’s say this character repeated or its frequency is 30k. In this case it will take 90k bits. How about now?

This quark also being generated automatically. How and why, we will see that now with how the algorithm works.

How the Algorithm Works

Let’s say we have 5 characters from A to E and they have some frequency. Now, first thing we need is a minimum priority queue. Priority queue is special type of queue which enqueue or dequeue an element from the queue based on priority. It, can be maximum priority queue or minimum priority queue which we will use here. Frequency of a character is the priority value.

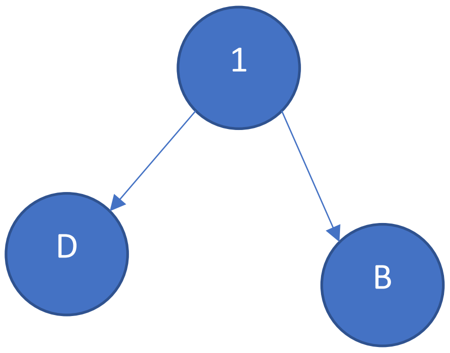
So, first this is push everything in a priority queue.

Character	A	B	C	D	E
Frequency	10	5	30	4	13

After that, extract the 2 minimum frequency element from the queue,

Character	A	~~B~~	C	~~D~~	E
Frequency	10	~~5~~	30	~~4~~	13

and combine them as a tree with two leaf and name it something, in our case 1.

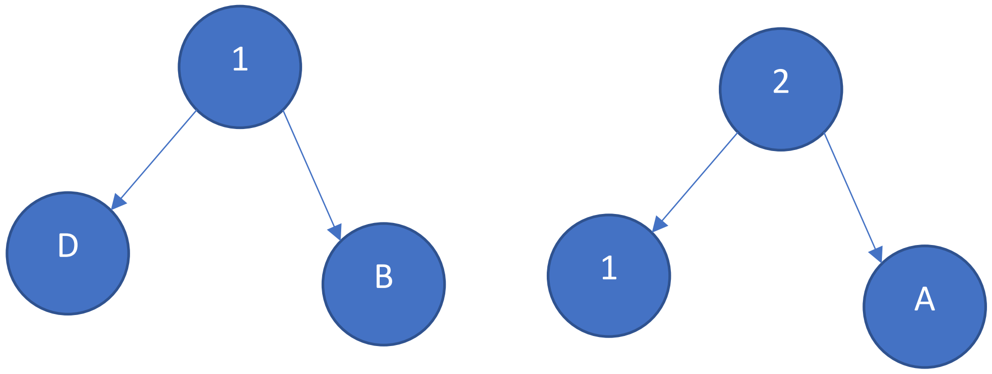


Now, push this new node into the queue and the frequency of this new node will be the summation of the leaf's frequency which is 9 in this case. So, the queue will look like this,

Character	A	C	E	1
Frequency	10	30	13	9

Now, again repeat the same thing, extract 2 min and combine and push them,

Character	~~A~~	C	E	~~1~~
Frequency	~~10~~	30	13	~~9~~

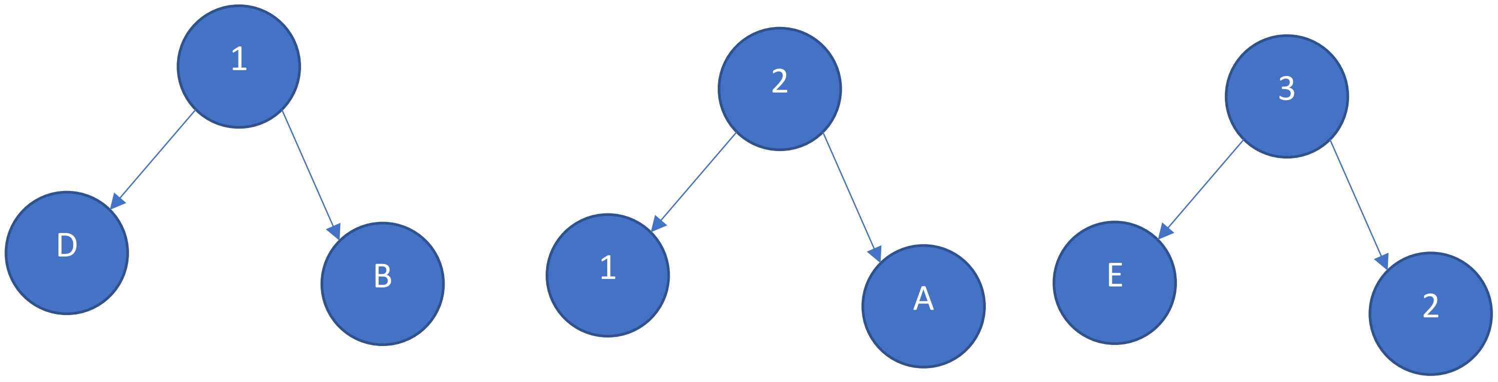


Character	C	E	2
Frequency	30	13	19

So, how many times we continue this process?
We will continue until our queue has 1 element left.

Since, queue has more than 1 element lets repeat again,

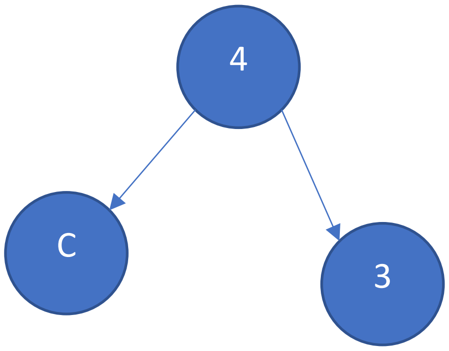
Character	C	E	2
Frequency	30	13	19

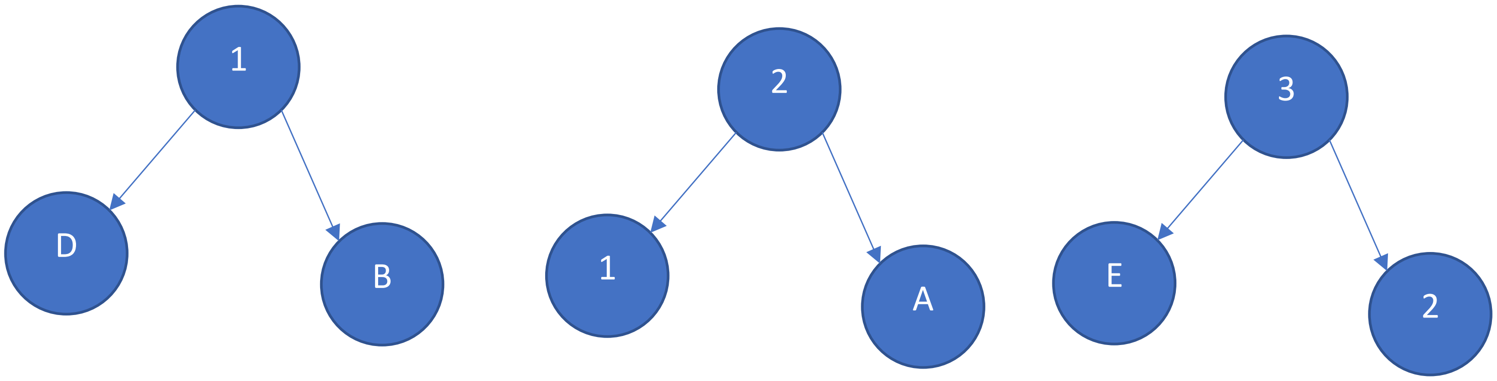


Character	C	3
Frequency	30	32

and again repeat

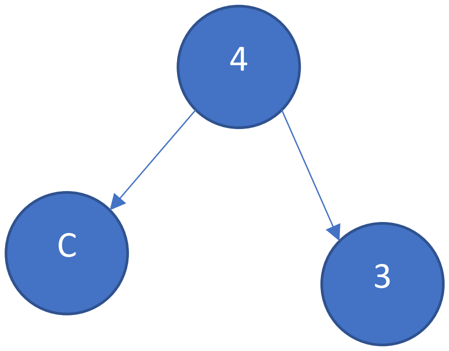
Character	C	3
Frequency	30	32



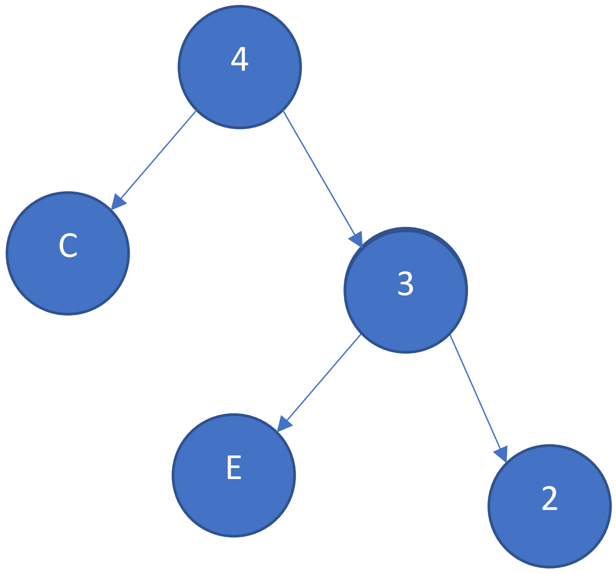


Character	4
Frequency	62

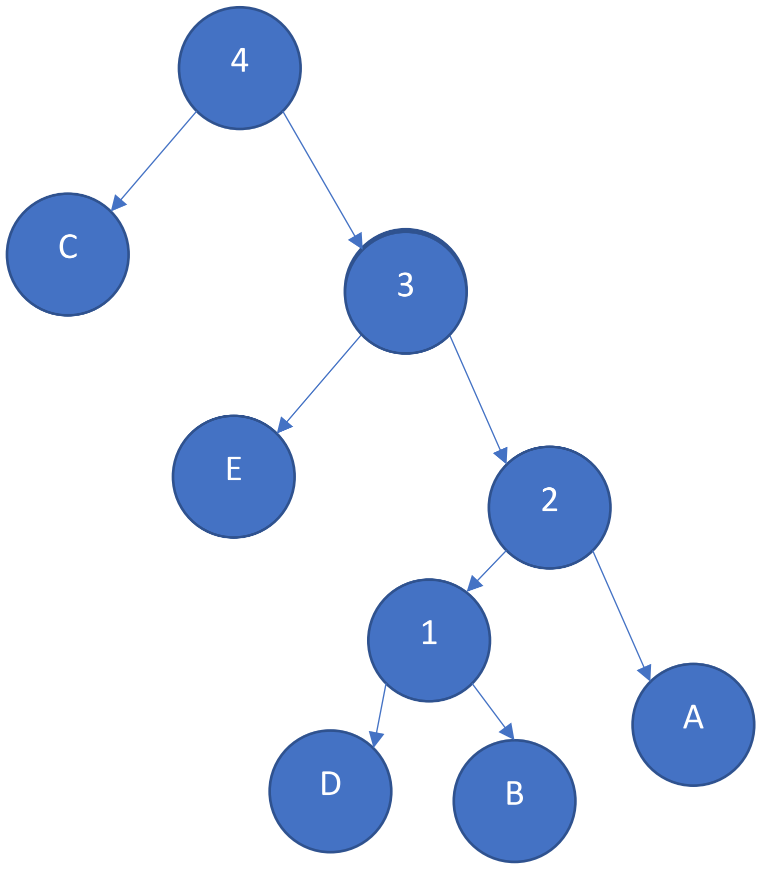
Now, in the queue we've only one element. So, this repeated process will end here. And with this we already got the tree. The remaining element we have is actually the tree. If we start with the tree that we created last,



We can see in this node named **4** has two leaf **C** and **3. C** is a character but **3** is another node. Now if we add these two together,

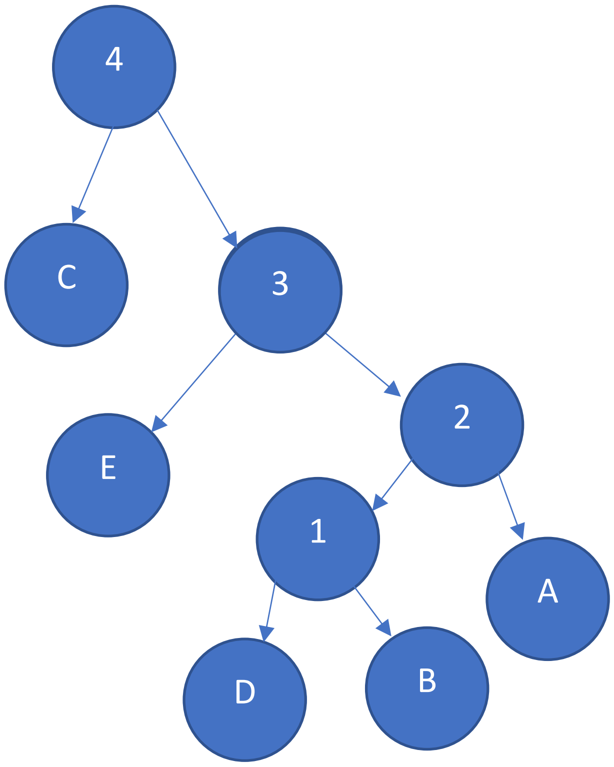


This is what we will get. And again here **2** is another node. So, after combining all nodes, the tree will look like,



Now, if we look closely at the tree and compare with the initial list,

Character	A	B	C	D	E
Frequency	10	5	30	4	13



Here, the character with the most frequency is **C** and it’s at the very top of the tree. The second highest frequency has **E** which is after the **C** in the tree and so on.

Why did this happen?

Remember we’ve always picked the minimum 2 and combined them?

So, in other words we were extracting the highest frequency character at the very last. And when we were constructing the tree we considered the last generated element as root. That means, The highest frequency element being added upper portion of the tree. This is how the quarks of the Huffman tree being generated. We will see how to get the prefix code from this tree in the simulation part.

Basic Pseudocode

The basic and simple pseudocode looks like this. Even though its simple it’s the actual process the whole algorithm works. It’s exactly the simulation we saw earlier.

```
huff(char[], freq[])
    Priority_Queue q <- (freq, char)
    While q.size > 1:
        a = q.pop()
        b = q.pop()
        q.push(a+b)
    return q.pop()
```

Why Huffman code is better

Naive approach

Suppose our string is: a b a b d c e c a b a c d a d e c a b c

This string contains 20 characters.

We know if we can compress data our total cost will be decrease. So, we need to reduce our text size for transmit it from sender to receiver.

Character	Frequency
a	6
b	4
c	5
d	3
e	2
	Total frequency = 20

In nave approach we can see our size of Total massage = 20*8=160 bit

Because of ASCII value is 8 bit per letter so we need to compress the size of our massage we can compress by two methods,

We can apply these 2 methods:

1. Fixed length code
2. Variable length code

Fixed length code

If we use 2 bits binary we can cover $=2^2 = 4$ letters

a=00

b=01

c=10

d=11

E=?

Hare is 5 separate character So, we need to use 3-bit code and we can assign 2^3 = 8 letters

Character	Frequency	Code
a	6	000
b	4	001
c	5	010
d	3	011
e	2	100
Total letters * ascii bit = 5*8=40		Total bit = 5*3=15

Total massage bit= 20*3=60

So total bit is: 40+15=55

So total massage size Is = 60+55=115 bit.

For fixed length code if we take 3-bit binary code we can apply 8 letter but if we have greeter then 4 and less than 8 letter we also use 3 bit so this algorithm is not efficient for compress message so we need a better one

Simulation

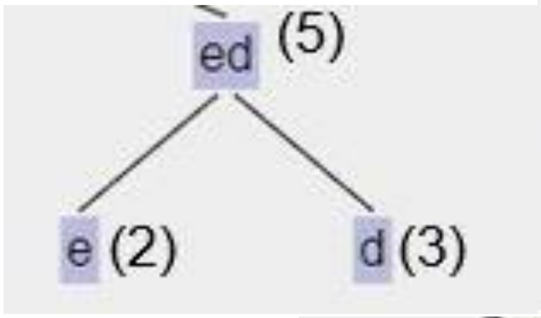
we can reduce more size by using Huffman code algorithm. And it is more efficient than fixed length code.

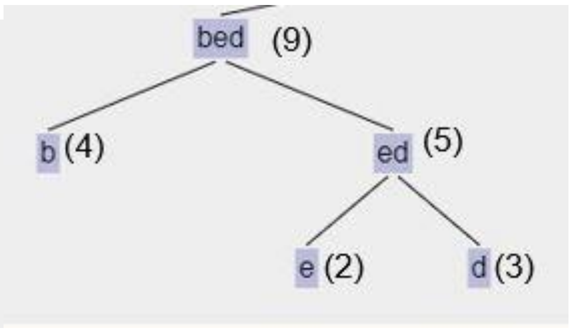
Huffman coding benefit is which letter occur maximum time those letter codes is lower bit and minimum time occurred letter size is larger bit. So, this algorithm is more efficient then fixed length code.

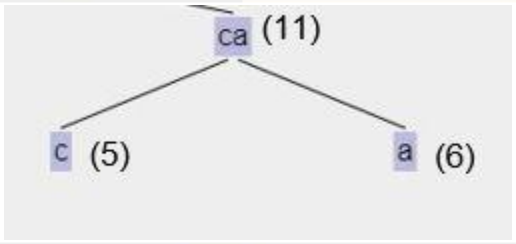
Character	Frequency
a	6
b	4
c	5
d	3
e	2
	Total frequency = 20

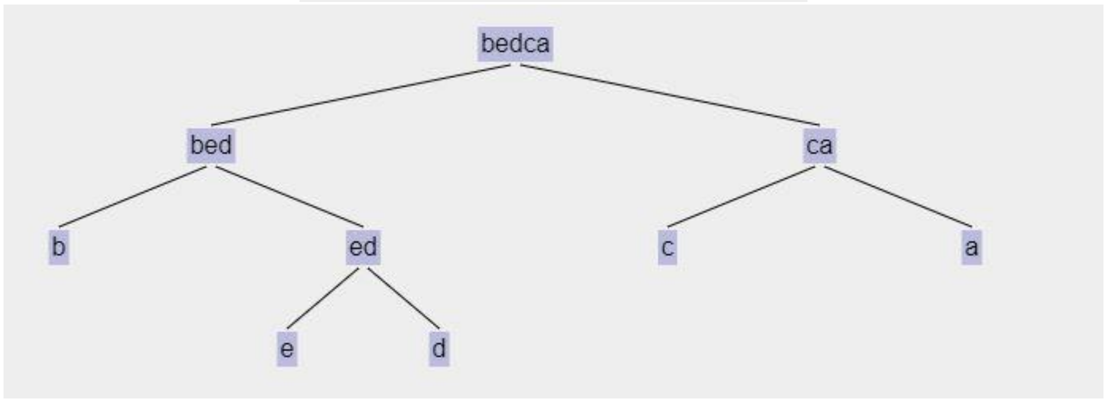
For Huffman coding we need to sort character non-decreasing order based on frequency

Character	Frequency
e	2
d	3
b	4
c	5
a	6

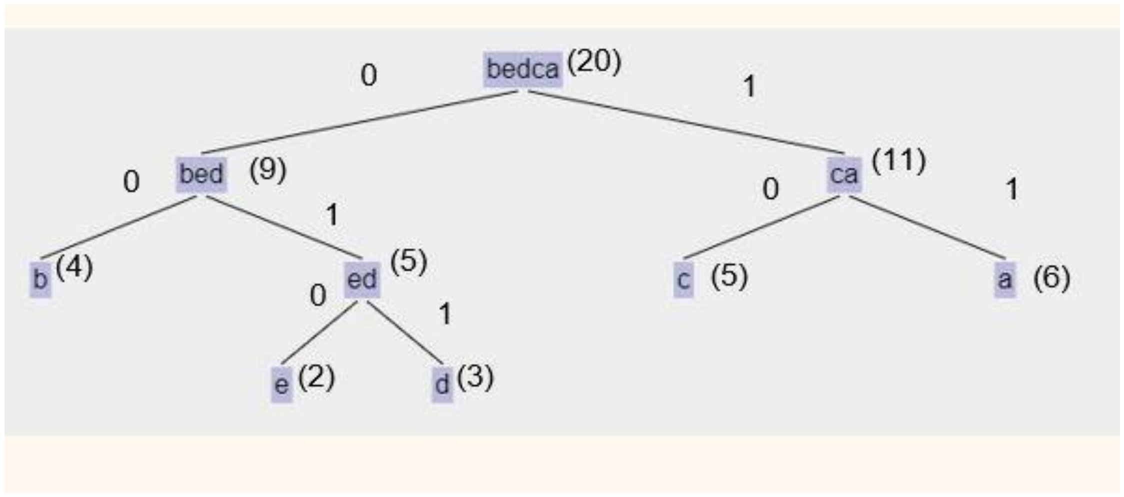








Assign left side edge 0 and right-side edge 1



Character	Frequency	Code	Frequency * code
a	6	11	6*2=12
b	4	00	4*2=8
c	5	10	5*2=10
d	3	011	3*3=9
e	2	010	2*3=6
Total letters * ascii bit = 5*8=40	+	Total bits = 12	Sum=45

Total bits of size is (40+12+45) =97 bits

So hare we can see a occur 6 time and a code is 2 bit which is 00 and e occur only 2 time and its code is 011 which is 3 bit. And our 3 length code has not contain 2 length code on prefix.

Time Complexity Analysis

```
1. huff(char[], freq[])
2.      Priority_Queue q <- (freq, char) ------------ O(n)
3.      While q.size > 1: -------------------------------- O(n)
4.              a = q.pop() ------------------------------ O(n) * O(lgn)
5.              b = q.pop() ------------------------------ O(n) * O(lgn)
6.              q.push(a+b) ---------------------------- O(n) * O(lgn)
7.      return q.pop() ----------------------------------- O(lgn)
```

=> 2n + 3 n log n + log n

=> O(n log n)

Code Implementation

The basic code implementation is given below,

```
struct node
{
    char ch;
    int freq;
    node *left, *right;
    node(char ch, int freq)
    {
        this->ch = ch;
        this->freq = freq;
        left = right = NULL;
    }
} typedef node;

typedef pair<int, node *> CH;
```

```
node *huffman_tree(vector<pair<char, int>> &freq)
{
    priority_queue<int, node, greater<int>> pq;
    for (auto f : freq)
        pq.push({f.second, new node(f.first, f.second)});
    while (pq.size() > 1)
    {
        auto left = pq.top();
        pq.pop();
        auto right = pq.top();
        pq.pop();
        node *top = new node('$', left.first + right.first);
        top->left = left.second;
        top->right = right.second;
        pq.push({top->freq, top});
    }
    return pq.top().second;
}
```

Full code implementation (with examples) can be found here:
https://github.com/Ahsan40/C-Programming-Practice/blob/master/DSA-Lab-2/Presentation/huffman.cpp

------------------------------------ END ------------------------------