Huffman Coding Algorithm

Every information in computer science is **encoded** as strings of **1s and 0s**. The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous. This tutorial discusses about fixed-length and variable-length encoding along with Huffman Encoding which is the basis for all data encoding schemes

Encoding, in computers, can be defined as the process of transmitting or storing sequence of characters efficiently. Fixed-length and variable lengthare two types of encoding schemes, explained as follows-

Fixed-Length encoding - Every character is assigned a binary code using same number of bits. Thus, a string like "aabacdad" can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

Variable- Length encoding - As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text. Thus, for a given string like "aabacdad", frequency of characters 'a', 'b', 'c' and 'd' is 4,1,1 and 2 respectively. Since 'a' occurs more frequently than 'b', 'c' and 'd', it uses least number of bits, followed by 'd', 'b' and 'c'. Suppose we randomly assign binary codes to each character as follows-

a 0 b 011 c 111 d 11

Thus, the string "aabacdad" gets encoded to **00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),** using fewer number of bits compared to fixed-length encoding scheme.

But the real problem lies with the decoding phase. If we try and decode the string 00011011111011, it will be quite ambiguous since, it can be decoded to the multiple strings, few of which are-

aaadacdad (0 | 0 | 0 | 11 | 0 | 111 | 11 | 0 | 11) aaadbcad (0 | 0 | 0 | 11 | 011 | 111 | 0 | 11) aabbcb (0 | 0 | 011 | 011 | 111 | 011)

... and so on

To prevent such ambiguities during decoding, the encoding phase should satisfy the **"prefix rule"** which states that no binary code should be a prefix of another code. This will produce uniquely **decodable codes**. The above codes for 'a', 'b', 'c' and 'd' do not follow prefix rule since the binary code for a, i.e. 0, is a prefix of binary code for b i.e 011, resulting in ambiguous **decodable codes.**

Lets reconsider assigning the binary codes to characters 'a', 'b', 'c' and 'd'.

a 0 b 11 c 101 d 100

Using the above codes, string **"aabacdad"** gets encoded to 0011010011000100 (0 | 0 | 11 | 0 | 101 | 100 | 0 | 100). Now, we can decode it back to string **"aabacdad"**.

Problem Statement-

Input: Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

Output: Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

Huffman Encoding-
Huffman Encoding can be used for finding solution to the given problem statement.

- Developed by **David Huffman** in 1951, this technique is the basis for all data compression and encoding schemes
- It is a famous algorithm used for lossless data encoding
- It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

The major steps involved in Huffman coding are-

Step I - Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

- A Huffman tree, similar to a binary tree data structure, needs to be created having **n** leaf nodes and **n-1** internal nodes
- Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
- Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
- Internal nodes, on the other hand, contain weight and links to two child nodes

Step II - Assigning the binary codes to each symbol by traversing Huffman tree

- Generally, bit '0' represents the left child and bit '1' represents the right child

Algorithm for creating the Huffman Tree-

Step 1- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step 2- Repeat Steps 3 to 5 while heap has more than one node

Step 3- Extract two nodes, say x and y, with minimum frequency from the heap

Step 4- Create a new internal node z with x as its left child and y as its right child.
Also $\text{frequency}(z) = \text{frequency}(x) + \text{frequency}(y)$

Step 5- Add z to min heap

Step 6- Last node in the heap is the root of Huffman tree

Let's try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

Characters	Frequencies
a	10
e	15

i	12
o	3
u	4
s	13
t	1

Step A- Create leaf nodes for all the characters and add them to the min heap.

- **Step 1**- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

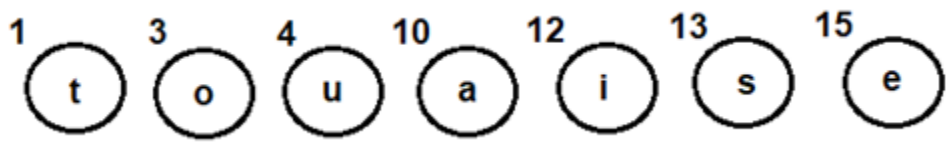

Fig 1: Leaf nodes for each character

Step B- Repeat the following steps till heap has more than one nodes

- **Step 3**- Extract two nodes, say x and y, with minimum frequency from the heap
- **Step 4**- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)
- **Step 5**- Add z to min heap

i. Extract and Combinenode u with an internal node having 4 as the frequency

ii. Add the new internal node to priority queue-

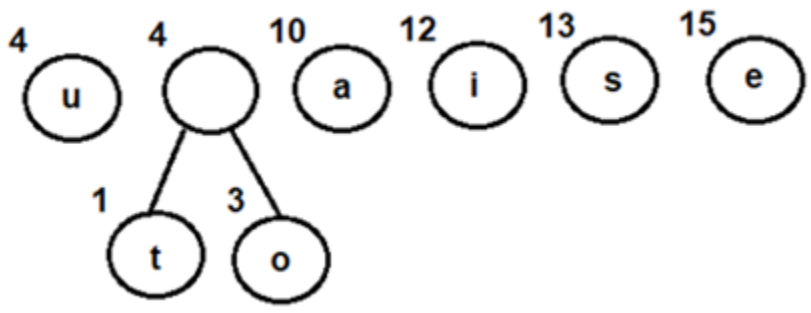

Fig 2: Combining nodes o and t

i. Extract and Combine node awith an internal node having 8 as the frequency

ii. Add the new internal node to priority queue-

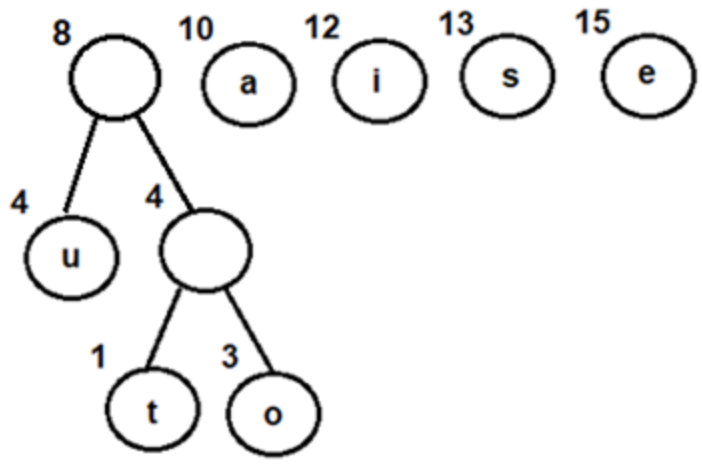

Fig 3: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s

ii. Add the new internal node to priority queue-

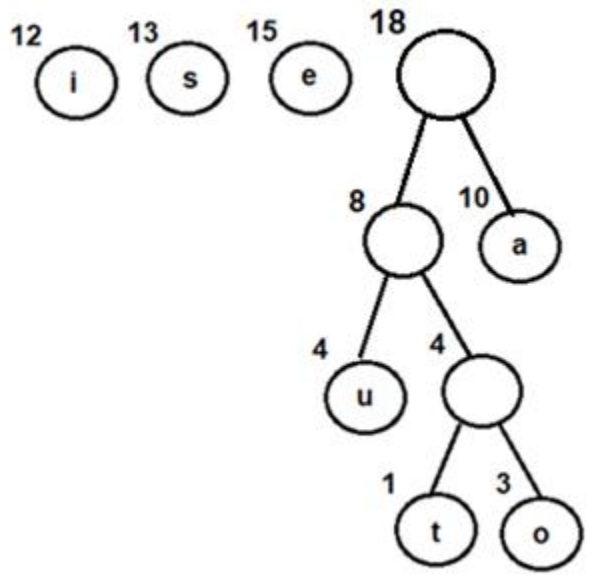

Fig 4: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s

ii. Add the new internal node to priority queue-

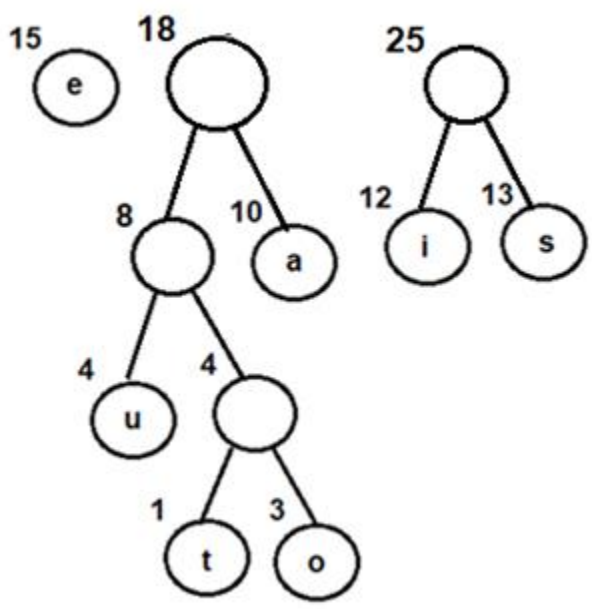

Fig 5: Combining nodes i and s

i. Extract and Combine node ewith an internal node having 18 as the frequency

ii. Add the new internal node to priority queue-

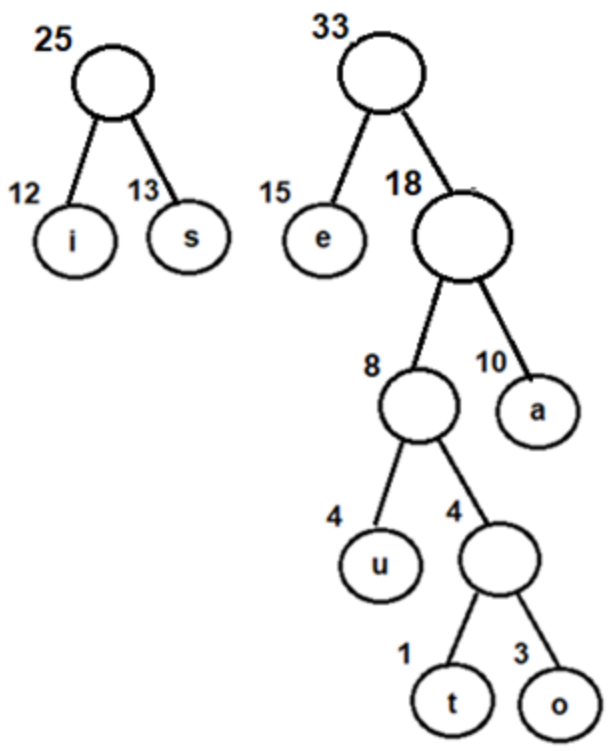

Fig 6: Combining node e with an internal node having 18 as frequency

i. Finally, Extract and Combine internal nodes having 25 and 33 as the frequency

ii. Add the new internal node to priority queue-

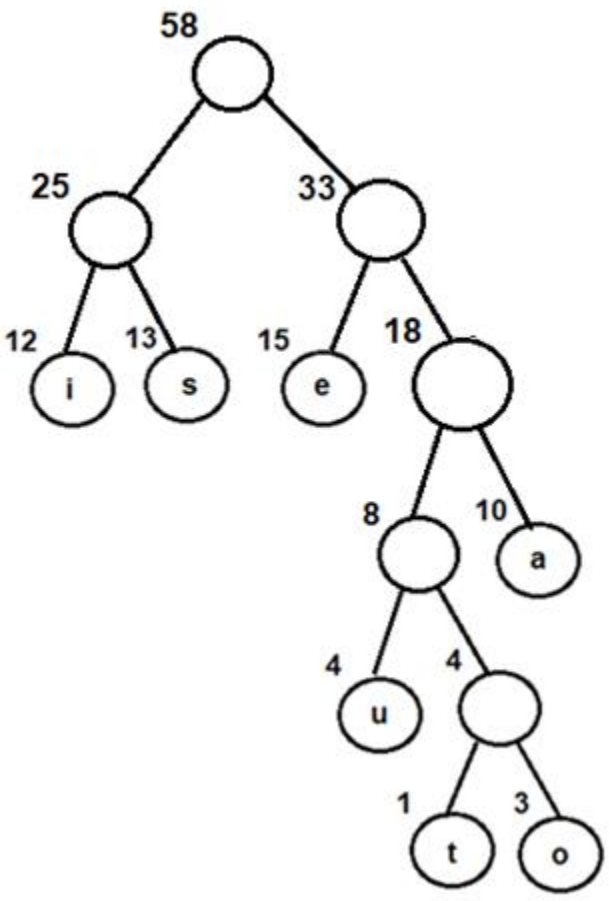

Fig 7: Final Huffman tree obtained by combining internal nodes having 25 and 33 as frequency

Now, since we have only one node in the queue, the control will exit out of the loop

Step C- Since internal node with frequency 58 is the only node in the queue, it becomes the root of **Huffman tree**.

Step 6- Last node in the heap is the root of Huffman tree

Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to arraywhile traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length-

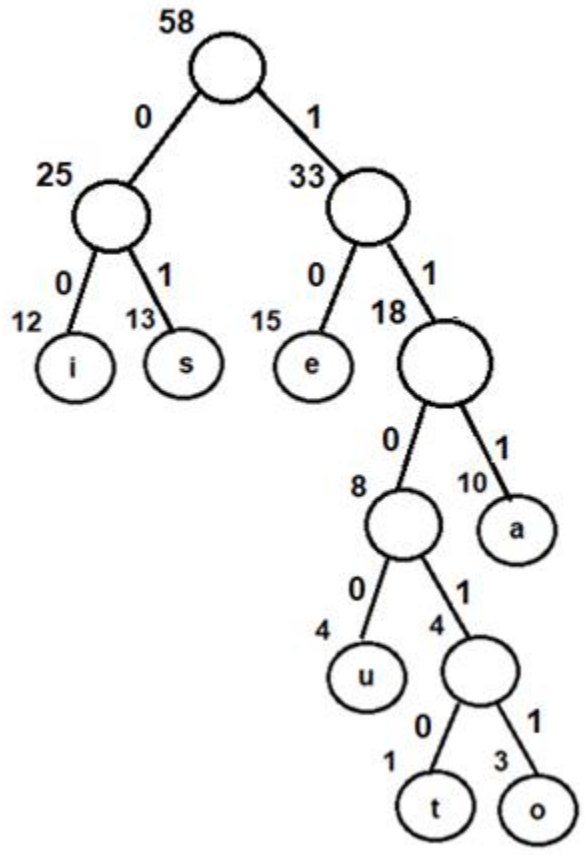

Fig 8: Assigning binary codes to Huffman tree

Characters	Binary Codes
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

Using the above binary codes-

Suppose the string “staeiout” needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to **“01110101111000110111110011010” (01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010)** at the sender side.

Once received at the receiver’s side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the

string. A ‘1’ or ‘0’ in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

Thus for the above bit stream

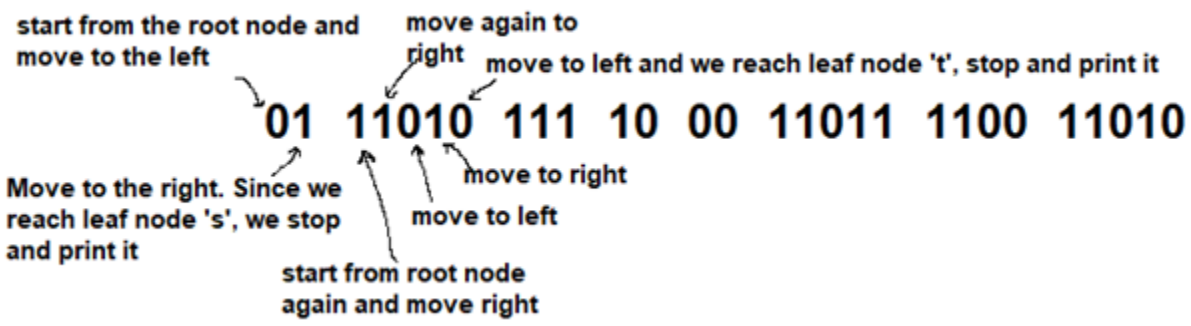

Fig 9: Decoding the bit stream

On similar lines-

- 111 gets decoded to ‘a’
- 10 gets decoded to ‘e’
- 00 gets decoded to ‘i’
- 11011 gets decoded to ‘o’
- 1100 gets decoded to ‘u’
- And finally, 11010 gets decoded to ‘t’, thus returning the string “staeiout” back

Implementation-

Following is the C++ implementation of Huffman coding. The algorithmcan be mapped to any programming language as per the requirement.

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>

using namespace std;

class Huffman_Codes
{
 struct New_Node
 {
  char data;
  size_t freq;
  New_Node* left;
```

```
  New_Node* right;

  New_Node(char data, size_t freq) : data(data),

                                     freq(freq),

left(NULL),

right(NULL)

                                     {}

 ~New_Node()

 {

   delete left;

   delete right;

 }

 };


 struct compare

 {

  bool operator()(New_Node* l, New_Node* r)

  {

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

  }

};


New_Node* top;


void print_Code(New_Node* root, string str)

{

if(root == NULL)

   return;


 if(root->data == '$')

 {
```

```
 print_Code(root->left, str + "0");

 print_Code(root->right, str + "1");

}



if(root->data != '$')

{

  cout << root->data <<" : " << str << "\n";

  print_Code(root->left, str + "0");

  print_Code(root->right, str + "1");

}

 }

}



public:

  Huffman_Codes() {};

  ~Huffman_Codes()

  {

    delete top;

  }

  void Generate_Huffman_tree(vector<char>& data, vector<size_t>& freq,
size_t size)

  {

   New_Node* left;

   New_Node* right;

   priority_queue<New_Node*, vector<New_Node*>, compare > minHeap;



for(size_t i = 0; i < size; ++i)

   {

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

   }



while(minHeap.size() != 1)
```

```
    {
      left = minHeap.top();
minHeap.pop();

      right = minHeap.top();
minHeap.pop();

      top = new New_Node('$', left->freq + right->freq);
      top->left  = left;
      top->right = right;
      minHeap.push(top);
     }
    print_Code(minHeap.top(), "");
 }
};

 int main()
 {
  int n, f;
  char ch;
  Huffman_Codes set1;
  vector<char> data;
  vector<size_t> freq;
  cout<<"Enter the number of elements \n";
  cin>>n;
  cout<<"Enter the characters \n";
  for (int i=0;i<n;i++)
  {
      cin>>ch;
data.insert(data.end(), ch);
```

```
  }
  cout<<"Enter the frequencies \n";
  for (int i=0;i<n;i++)
  {
      cin>>f;
freq.insert(freq.end(), f);
  }

  size_t size = data.size();
  set1.Generate_Huffman_tree(data, freq, size);

  return 0;
}
```

Copy

The program is executed using same inputs as that of the example explained above. This will help in verifying the resultant solution set with actual output.

```
Enter the number of elements
7
Enter the characters
a e i o u s t
Enter the frequencies
10 15 12 3 4 13 1
i : 00
s : 01
e : 10
u : 1100
t : 11010
o : 11011
a : 111
```

Fig 10: Output

Time Complexity Analysis-

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

- Building a min heap takes $O(nlogn)$ time (Moving an element from root to leaf node requires $O(logn)$ comparisons and this is done for n/2 elements, in the worst case).
- Building a min heap takes $O(nlogn)$ time (Moving an element from root to leaf node requires $O(logn)$ comparisons and this is done for n/2 elements, in the worst case).

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

Advantages of Huffman Encoding-

- This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length
- 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.