## horizontal line



Huffman Encoding

Design and Analysis of Algorithms

# Overview

In computer science and information theory, a **Huffman code** is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding and/or using such a code proceeds by means of **Huffman coding**, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

# Goals

Computers generally encode characters using the standard ASCII chart, which assigns an 8-bit code to each symbol. For an example, the letter ‘a’ has an ASCII value of 97, and is encoded as ‘01100001’. Characters that occur more frequently (such as ‘e’) are treated the same as rare characters, such as ‘¸’. A file that has 100 characters will require 800 bits – this value is fixed, whether the file contains 100 unique characters or if it has 100 occurrences of the same character. The advantages of the ASCII encoding scheme is that boundaries between characters are easily determined, and the pattern used for each character is fixed and universal.

However, in almost any given text file, some characters occur with more frequency than others. Wouldn’t it thus make more sense to assign shorter bit codes to more frequent characters than less frequent ones?

By doing so, we can save lots of storage. And that’s our goal when we implement this algorithm.

# Algorithm:-

The Algorithm can be divided into two Parts, Encoding and Decoding:-

Encoding:-

There are mainly two major parts in **Huffman Coding**

**1)** Build a Huffman Tree from input characters.

**2)** Traverse the Huffman Tree and assign codes to characters.

***Steps to build Huffman Tree***

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

**1.** Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

**2.** Extract two nodes with the minimum frequency from the min heap.

**3.** Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

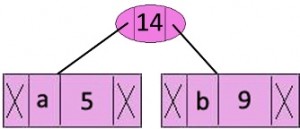
**4.** Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character Frequency  
 a 5  
 b 9  
 c 12  
 d 13  
 e 16  
 f 45

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

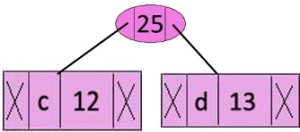
**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character Frequency  
 c 12  
 d 13  
 Internal Node 14  
 e 16  
 f 45

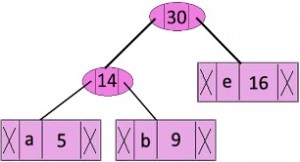
**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

character Frequency  
Internal Node 14  
 e 16  
Internal Node 25  
 f 45

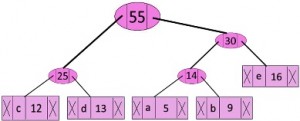
**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

character Frequency  
Internal Node 25  
Internal Node 30  
 f 45

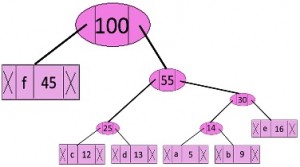
**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

character Frequency  
 f 45  
Internal Node 55

**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



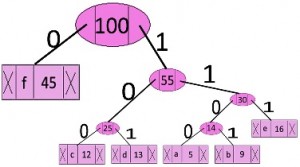
Now min heap contains only one node.

character Frequency  
Internal Node 100

Since the heap contains only one node, the algorithm stops here.

***Steps to print codes from Huffman Tree:***

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.





**Decoding:-**

To decode the encoded data we require the Huffman tree. We iterate through the binary encoded data. To find character corresponding to current bits, we use following simple steps.

1. We start from root and do following until a leaf is found.
2. If current bit is 0, we move to left node of the tree.
3. If the bit is 1, we move to right node of the tree.
4. If during traversal, we encounter a leaf node, we print character of that particular leaf node and then again continue the iteration of the encoded data starting from step 1.

# Code:-

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

*#define MAX\_TREE\_HT 256*

*using namespace std;*

*map<char, string> codes;*

*map<char, int> freq;*

*struct MinHeapNode*

*{*

*char data;*

*int freq;*

*MinHeapNode \*left, \*right;*

*MinHeapNode(char data, int freq)*

*{*

*left = right = NULL;*

*this->data = data;*

*this->freq = freq;*

*}*

*};*

*struct compare*

*{*

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

*{*

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

*}*

*};*

*void printCodes(struct MinHeapNode\* root, string str)*

*{*

*if (!root)*

*return;*

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

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

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

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

*}*

*void storeCodes(struct MinHeapNode\* root, string str)*

*{*

*if (root==NULL)*

*return;*

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

*codes[root->data]=str;*

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

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

*}*

*priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>, compare> minHeap;*

*void HuffmanCodes(int size)*

*{*

*struct MinHeapNode \*left, \*right, \*top;*

*for (map<char, int>::iterator v=freq.begin(); v!=freq.end(); v++)*

*minHeap.push(new MinHeapNode(v->first, v->second));*

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

*{*

*left = minHeap.top();*

*minHeap.pop();*

*right = minHeap.top();*

*minHeap.pop();*

*top = new MinHeapNode('$', left->freq + right->freq);*

*top->left = left;*

*top->right = right;*

*minHeap.push(top);*

*}*

*storeCodes(minHeap.top(), "");*

*}*

*void calcFreq(string str, int n)*

*{*

*for (int i=0; i<str.size(); i++)*

*freq[str[i]]++;*

*}*

*string decode\_file(struct MinHeapNode\* root, string s)*

*{*

*string ans = "";*

*struct MinHeapNode\* curr = root;*

*for (int i=0;i<s.size();i++)*

*{*

*if (s[i] == '0')*

*curr = curr->left;*

*else*

*curr = curr->right;*

*if (curr->left==NULL and curr->right==NULL)*

*{*

*ans += curr->data;*

*curr = root;*

*}*

*}*

*return ans+'\0';*

*}*

*int main()*

*{*

*string str;*

*std::ofstream out("output.txt");*

*std::ifstream t("input.txt");*

*std::ifstream dt("decode.txt");*

*std::ofstream dout("decoded.txt");*

*int operation;*

*std::stringstream buffer;*

*string encodedString, decodedString;*

*cout << "encode(1) or decode(2)?";*

*cin >> operation;*

*switch(operation){*

*case(1):*

*buffer << t.rdbuf();*

*str=buffer.str();*

*for ( auto it = str.begin() ; it < str.end(); it++ )*

*{*

*if(\*it==' ') \*it='\_';*

*else if (\*it=='\n') \*it='#';*

*}*

*calcFreq(str, str.length());*

*HuffmanCodes(str.length());*

*out << "Number of key elements:\n\n " << codes.size();*

*out << "Character With there Frequencies:\n";*

*for (auto v=codes.begin(); v!=codes.end(); v++) {*

*out << v->first <<' ' << v->second << endl;*

*}*

*for (auto i: str)*

*encodedString+=codes[i];*

*out << "\nEncoded Huffman data:\n" << encodedString << endl;*

*out.close();*

*cout << "\nData has been encoded in output.txt\n";*

*break;*

*case(2):*

*int keys;*

*char key;*

*string val;*

*string ans="";*

*cout << "Enter the number of keys:";*

*cin >> keys;*

*cout << "Enter the key value pair:";*

*for(int i=0;i<keys;i++){*

*cin >> key;*

*cin >> val;*

*codes.insert({ key, val });*

*}*

*buffer << dt.rdbuf();*

*str=buffer.str();*

*int i=0,j=0, length=str.length();*

*while(i+j<=length){*

*string tmp= str.substr(i,j);*

*for (auto it=codes.begin();it!=codes.end();it++){*

*if(tmp==it->second){*

*ans+=it->first;*

*i=i+j;*

*j=0;*

*}*

*}*

*j++;*

*}*

*for ( auto it = ans.begin() ; it < ans.end(); it++ )*

*{*

*if(\*it=='\_') \*it=' ';*

*else if (\*it=='#') \*it='\n';*

*}*

*cout << ans;*

*dout << ans;*

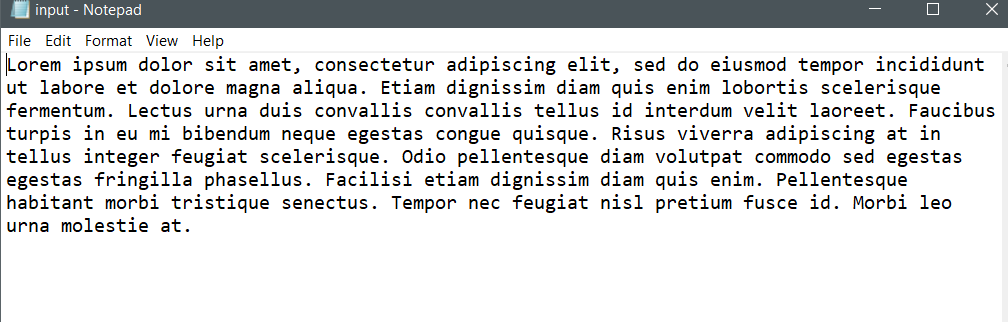
*}*

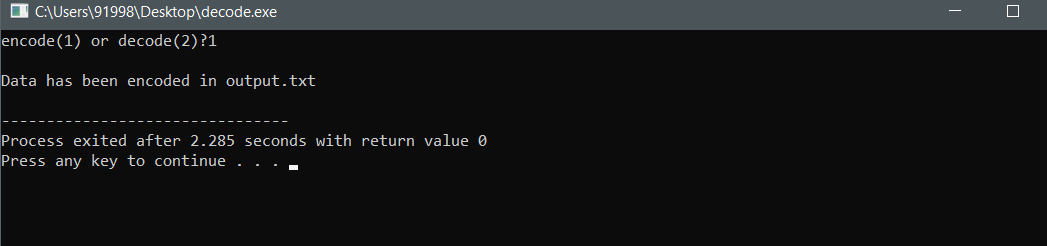
*return 0;*

*}*



# Result:-





# 

# 

# 

# 

# Conclusion:-

**Comparing Input file size and Output file size:**

Comparing the input file size and the Huffman encoded output file. We can calculate the size of the output data in a simple way. Let’s say our input is a string “geeksforgeeks” and is stored in a file input.txt.

**Input File Size:**

Input: "huffmancoding"  
Total number of character i.e. input length: 13  
Size: 13 character occurrences \* 8 bits = 104 bits or 13 bytes.

**Output File Size:**

Input: "huffmancoding"  
------------------------------------------------  
Character | Frequency | Binary Huffman Value |  
------------------------------------------------  
 a | 1 | 0110 |  
 c | 1 | 1010 |   
 d | 1 | 0111 |  
 f | 2 | 110 |  
 g | 1 | 1011 |  
 h | 1 | 1111 |  
 i | 1 | 000 |  
 m | 1 | 010 |  
 n | 2 | 100 |  
 o | 1 | 1110 |  
 u | 1 | 001 |  
------------------------------------------------  
  
So to calculate output size:  
a: 1 occurrences \* 4 bits = 4 bits  
c: 1 occurrence \* 4 bits = 4 bits  
d: 1 occurrences \* 4 bits = 4 bits  
f: 2 occurrences \* 3 bits = 6 bits  
g: 1 occurrence \* 4 bits = 4 bits  
h: 1 occurrence \* 4 bits = 4 bits  
i: 1 occurrences \* 3 bits = 3 bits  
m: 1 occurrences \* 3 bits = 3 bits  
n: 2 occurrences \* 3 bits = 6 bits  
o: 1 occurrences \* 4 bits = 4 bits  
u: 1 occurrences \* 3 bits = 6 bits

Total Sum: 39 bits approx 5 bytes

Hence, we could see that after encoding the data we have saved a large amount of data.

# References:-

* Huffman Coding | Greedy Algo:  
  <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>
* Huffman Decoding:  
  <https://www.geeksforgeeks.org/huffman-decoding/>
* Huffman coding wikipedia:  
  <https://en.wikipedia.org/wiki/Huffman_coding>