Huffman Coding

by

John Kennedy
Mathematics Department
Santa Monica College
1900 Pico Blvd.
Santa Monica, CA 90405

rkennedy@ix.netcom.com

Except for this comment explaining that it is blank for some deliberate reason, this page is intentionally blank!

Huffman Coding

There have been many codes used throughout history to represent letters and messages. Three of the most popular codes are Morse Code, ASCII code, and UNICODE. The next two tables below show some of the characters used in both Morse Code and in 8-bit ASCII codes.

Letter	Morse Code	8-bit ASCII	Decimal #
A	.-	0100 0001	65
B	-...	0100 0010	66
C	-.-.	0100 0011	67
D	-..	0100 0100	68
E	.	0100 0101	69
F	..-.	0100 0110	70
G	--.	0100 0111	71
H	....	0100 1000	72
I	..	0100 1001	73
J	.---	0100 1010	74
K	-.-	0100 1011	75
L	.-..	0100 1100	76
M	--	0100 1101	77
N	-.	0100 1110	78
O	---	0100 1111	79
P	.--.	0101 0000	80
Q	--.-	0101 0001	81
R	.-.	0101 0010	82
S	...	0101 0011	83
T	-	0101 0100	84
U	..-	0101 0101	85
V	...-	0101 0110	86
W	.--	0101 0111	87
X	-..-	0101 1000	88
Y	-.--	0101 1001	89
Z	--..	0101 1010	90

Letter	Morse Code	8-bit ASCII	Decimal #
0	-----	0011 0000	48
1	.----	0011 0001	49
2	..---	0011 0010	50
3	...--	0011 0011	51
4	....-	0011 0100	52
5	.....	0011 0101	53
6	-....	0011 0110	54
7	--...	0011 0111	55
8	---..	0011 1000	56
9	----.	0011 1001	57
?	..--..	0011 1111	63
.	.-.-.-	0010 1110	46
,	--..--	0010 1100	44
/	-..-.	0010 1111	47
$	...-..-	0010 0100	36
-	-....-	0010 1101	45
:	---...	0011 1010	58
;	-.-.-.	0011 1011	59

Samuel B. Morse is credited with having invented the telegraph. In 1844 Morse sent the following message over his telegraph between Washington, DC and Baltimore, Maryland: "WHAT HATH GOD WROUGHT". This message has a total of 21 letters including spaces. The letters and their frequencies are noted in the two tables below.

Letter →	W	H	A	T	G
Frequency →	2	4	2	3	2
ASCII bits →	0101 0111	0100 1000	0100 0001	0101 0100	0100 0111
Morse Code →	.--	....	.-	-	--.

Letter →	O	D	R	U	space
Frequency →	2	1	1	1	3
ASCII bits →	0100 1111	0100 0100	0101 0010	0101 0101	0010 0000
Morse Code →	---	-..	.-.	..-	none

If we use ASCII characters as binary integers to write this message, each character uses 8 bits which means the entire 21 letter message requires 168 bits. ASCII is an example of what is called a fixed-length encoding scheme since each character is comprised of exactly 8 bits.

Morse Code is an example of a variable-length encoding scheme. In general, if all letters had the same length, then there would be no advantage in using Morse Code over a type of ASCII code. But Morse knew that the letters E and T appear much more frequently in English text than letters like Q and Z appear. For that reason, Morse gave the letters E and T the single dit **.** and dah **-** codes and he gave the vowels A and I the 2-codes **.-** and **..**. He gave seldom-used letters like Q and Z the longer 4-codes like **--.-** and **--..**. Morse's original code was later modified and internationalized and now includes numbers and some punctuation marks. Similarly the ASCII code has been modified and extended and is now a subpart of a newer coding scheme called UNICODE.

Using Morse Code, we can treat each dit and dah mark as the equivalent of one bit each. Unfortunately, there is no space character in Morse Code but we will assume a space would require some pattern of 5 dits and dahs. In this case, coding the above message in Morse Code would only require the use of 62 bits. You might note that Morse used at most 4 dit and dah codes to create all 26 upper-case letters of the English alphabet. If we think about it, 26 different letters of equal bit-length should require at least 5 bits to encode. As will be explained below, Morse's variable 1-bit, 2-bit, 3-bit and 4-bit scheme suffers from a problem that must be overcome to make it practical to use to un-encode long strings of dits and dahs that are run together. This other problem is called ***the prefix problem***. Ignoring the prefix problem for the moment, Morse Code results in a tremendous savings of 106 bits out of 168 bits. As a percentage of space, we save approximately 63% by using Morse Code over ASCII code, at least for the given message.

Now we can ask the question: Is there an even more efficient scheme for coding this message? In fact, we can ask what is the most efficient scheme for coding this or any other message? Since our message has 21 letters, if we used a fixed length code we would require at least 5 bits per letter, since 4 bits could only be used to encode at most 16 distinct items. As we will see, we can do better using a variable-length code.

Variable length codes have the disadvantage that they are difficult to manipulate in computer memory that is made to work with fixed size objects. Some variable length codes may also fail to have the prefix property. You can appreciate the difficulties of not having the prefix property if we ask you to un-encode the following message in Morse Code where the dits and dahs of multiple letters have been run together. How would you decode the sequence of dits and dahs shown below?

.--.--.....---.--

The first question that immediately occurs is this. Does this message start with the letter A, or does it start with the letter W, or does it start with the letter P? Without knowing where to separate the dits and dahs of each character, the answers to these questions can all be Yes or they can all be No or they can be some combination of Yes's and No's. The problem is that strings of dits and dahs in Morse Code are ambiguous unless you have some means to isolate the individual characters. The above message could in fact begin with any of the letters A, or W or P.

When you look carefully at the Morse Code you find it fails to have the prefix property. To have the desirable prefix property, a code must be such that no letter is the prefix to another letter. In Morse Code the letter A is a prefix to both the letters W and P and the letter W is also a prefix to the letter P. Any code that fails to have the prefix property can be altered by inserting a new symbol that represents a separation character, but adding a separation character between each two transmitted characters just doubles the amount of data that needs to be transmitted and this defeats the whole point of encoding data in the first place. And there is an additional problem that is yet to be solved. Using Morse Code, how do you encode the separation character itself using dits and dahs?

Above, when we said Morse Code would require 62 bits for the sample message, we did not take into account the failure of the prefix property. In the old days, telegraph operators used timing between taps of the telegraph key to separate characters. Thus the receiving operator had some idea of when one character ended and when the next character began. But inside a computer's memory, there is no such timing information between bits. There are only large sections of memory that contain only 0's and 1's. So to obtain a useable coding scheme for computer use, we can't use true Morse Code, but we can borrow (or "steal") some useful ideas from Morse Code and improve upon it.

We are going to build a tree structure to represent the coding scheme for any message that uses the letters shown in the table below. This time we list the letters, sorted with the smallest frequencies written first. In this first table the individual letters are also considered as subtrees.

Table #1

Frequency	Letter (Subtree)
1	D
1	R
1	U
2	G
2	O
2	A
2	W
3	T
3	space
4	H

The Huffman tree building algorithm can be described by the following five steps:

1. Create an initial list of characters (subtrees) and frequencies. (This is our Table #1).
2. If the subtree list contains only one tree item then you are done!
3. Otherwise, remove from the list the two subtrees with the smallest frequencies and make them children of a new combined subtree whose frequency is the sum of the two child frequencies.
4. Add the new combined subtree to the list, keeping the entire list sorted by frequency.
5. Go back to step 2.

We will show the sequence of steps through this algorithm by constructing a sequence of tables. Our tables will always be sorted in the first column by the frequency number. Whenever we insert a new subtree in a table, we place it as high as it can go in the table while keeping the entire list sorted by frequency.

Table #2

Frequency	Subtree
1	U
2	(D,R)
2	G
2	O
2	A
2	W
3	T
3	space
4	H

Table #3

Frequency	Subtree
2	G
2	O
2	A
2	W
3	(U,(D,R))
3	T
3	space
4	H

Table #4

Frequency	Subtree
2	A
2	W
3	(U,(D,R))
3	T
3	space
4	(G,O)
4	H

Table #5

Frequency	Subtree
3	(U,(D,R))
3	T
3	space
4	(A,W)
4	(G,O)
4	H

Table #6

Frequency	Subtree
3	space
4	(A,W)
4	(G,O)
4	H
6	((U,(D,R)),T)

Table #7

Frequency	Subtree
4	(G,O)
4	H
6	((U,(D,R)),T)
7	(space,(A,W))

Table #8

Frequency	Subtree
6	((U,(D,R)),T)
7	(space,(A,W))
8	((G,O),H)

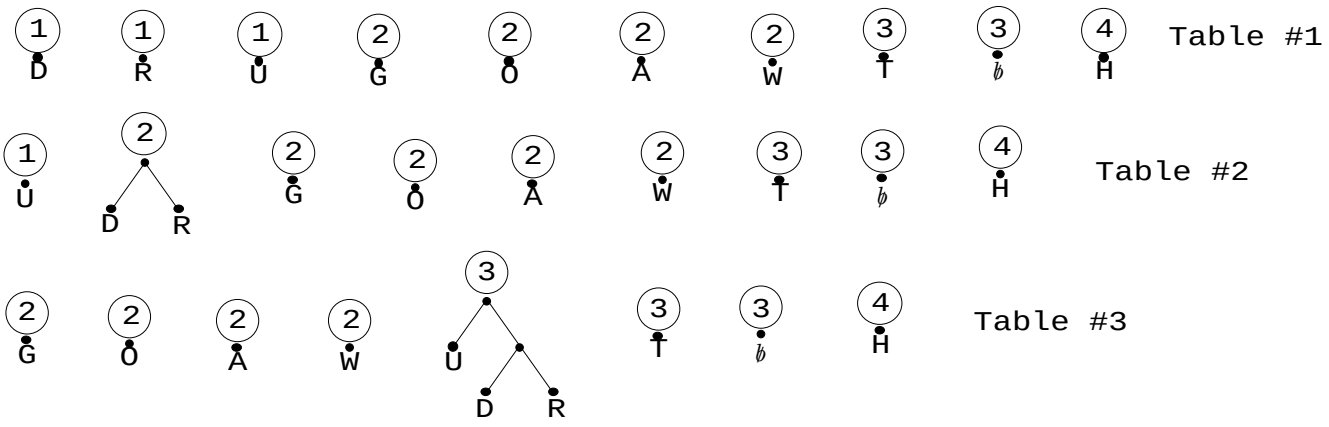
Table #9

Frequency	Subtree
8	((G,O),H)
13	(((U,(D,R)),T),(space,(A,W)))

Table #10

Frequency	Subtree
21	(((G,)),H),(((U,(D,R)),T),(space,(A,W))))

Next we show the constructions of these same tables in terms of trees. We use the character ␣ to denote the **space** character. The circled numbers are the frequency counts of the various subtree nodes.



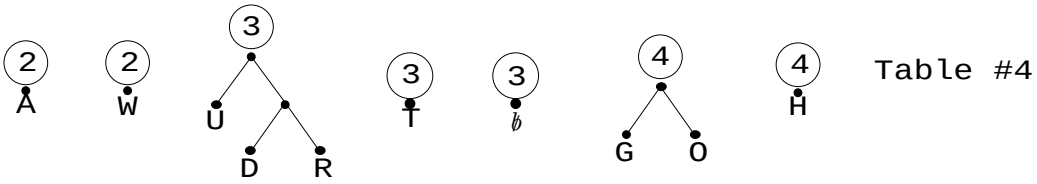


Table #4

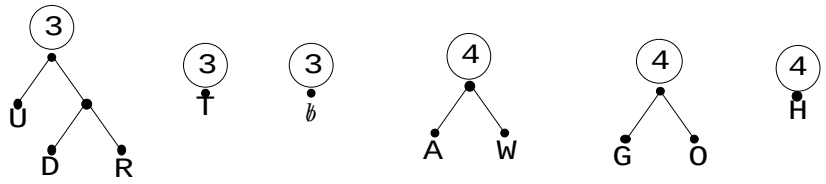


Table #5

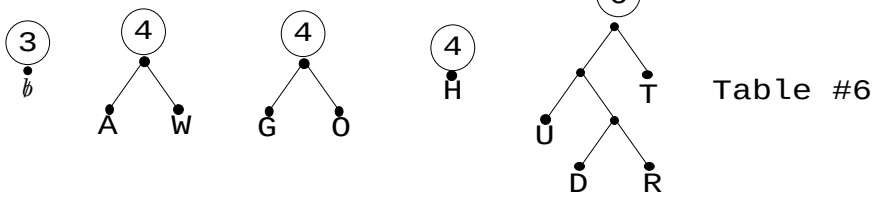


Table #6

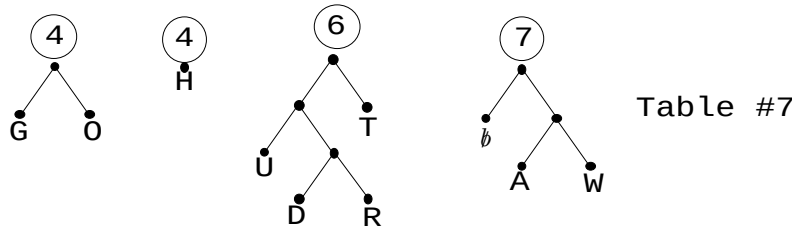


Table #7

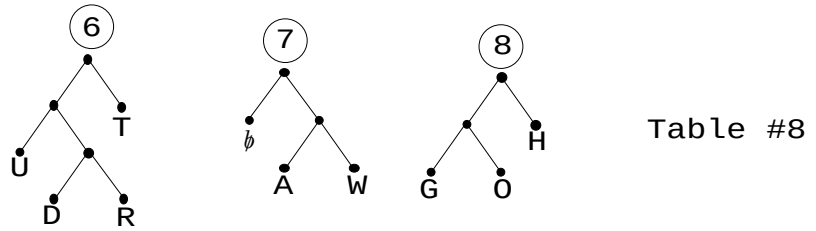


Table #8

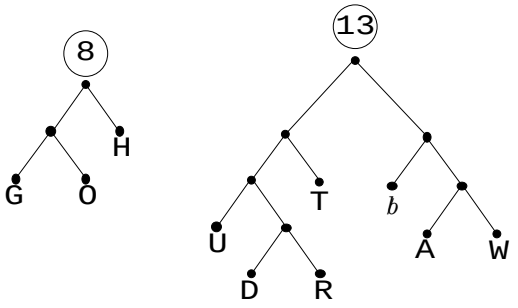
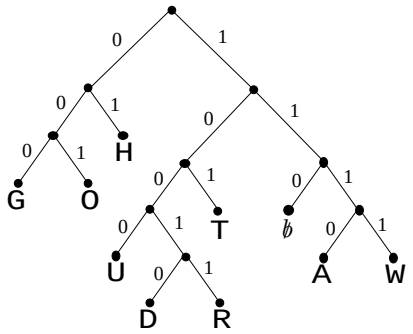


Table #9

When we get down to just two subtrees left, we combine these into the final tree that we call the Huffman Tree.

The Huffman Tree



In creating the final Huffman tree, we glue together the two final subtrees. After that, we dispense with writing a frequency number and we label all the connecting lines between nodes using the rule that left branches are always labeled with a 0 bit while right branches are always labeled with a 1 bit. Note that characters only appear as leaf nodes within the tree, and only leaf nodes within the tree represent characters.

By reading the final Huffman Tree we can make the following table of binary codes for the letters in our message. Start with the root node at the top of the tree and trace the unique path going down to any desired letter. The 0's and 1's along the path to the letter determine the Huffman binary code for the letter. For example, the binary code for the letter T is 101, obtained by moving along the path from the root node that first branches right and then branches left and finally branches right. Each left branch contributes a 0 bit and each right branch contributes a 1 bit.

The Huffman Final Table.

Letter	Huffman Binary Code
H	01
space = $\not b$	110
T	101
G	000
O	001
W	1111
A	1110
U	1000
R	10011
D	10010

Note that our Huffman binary codes have the property that no character code is a prefix to any other character code. When we count the letters and their frequencies we find the total encoded message consists of 68 bits. Because Morse Code has the prefix problem that we never fixed, the Huffman Coding scheme is actually more efficient than, and is superior to, the pseudo 62-bit Morse Code scheme.

The binary codes we just devised for the simple message "WHAT HATH GOD WROUGHT" are not necessarily the same codes we would use with another message. There are two schemes you might wish to consider. If you were going to use Huffman Coding to encode ASCII text files consisting of plain English text, then you could write a program that used a fixed coding scheme based on the frequencies of all letters taken from a representative sampling of a large number of common English texts. This is rarely done.

A second and more popular alternative is to perform what is called adaptive coding for each file you wish to encode. This second alternative has the slight disadvantage that you have to read the text file twice. The first time you read the text file you keep track of all distinct characters that appear in the file and you count and keep track of their frequencies. The second time you read the file you will have already built the Huffman Tree/Table and then you just look up each character's code and output the encoded character bits. However, you are assured of using the most efficient coding by using this second scheme for each different file.

The proof that a Huffman Code has the prefix property just depends on the fact that all the characters occur at only the leaf nodes in the tree. Otherwise, in order for a first character to be a prefix of a second character, the first character would have to appear earlier and along the same branch as the second character. This is impossible because once a character is reached, you must be at a leaf node. There are no characters further down from the leaf because there is no path coming out of the leaf.

There are several implementation details that you will find helpful if you try to write programs to both encode and decode using the Huffman scheme. The first implementation detail is that a node in your tree should probably have a structure with five fields such as:

```
ATreeNode
   Character (1 byte)
   Frequency (long integer)
   Parent (tree node pointer)
   LeftChild (tree node pointer)
   RightChild (tree node pointer)
```

The Parent pointer can be used to navigate upwards in your tree from a character leaf to the root node.

A second implementation detail is related to the fact that computer files don't have a granularity that goes beyond the byte level. Chances are 7 out of 8 that your encoded file will not have a bit count that is an exact multiple of 8. Thus the last byte in your encoding stream will contain a few unused bits, and these unused bits may confuse your decoder unless you plan for this. One scheme is to store a long integer at the beginning of your encoded file that gets read first. This long integer tells how many characters there were in the original file.

A third surprising but useful idea is that you can actually store the final Huffman Tree as additional header information at the beginning of the encoded file in an extremely efficient way. When you read the encoded file, after you read the original file byte count, the next thing you do is read the header information and build the Huffman Coding tree in computer memory. Then you start at the top or root node of the tree and you read the encoded data bits and make your way down through the tree. When you hit a leaf node you know you have finished with the current character and you can start back at the top of the tree for the next character until you determine (using the long integer noted above) that all characters have been decoded.

The algorithm to completely perform Huffman encoding and decoding has too many implementation details to describe here, but everything that is required is clearly explained in an article written by Jonathan Amsterdam that appeared in the May, 1986 issue of Byte Magazine on pages 99-108. The original technique by David A. Huffman was described in an article published by Huffman in 1952 in a professional journal for the Institute of Radio Engineers. "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the IRE 40 (1952) pp. 1098-1101. It is a long time between 1952 and 1986, and you will find the 1986 article in Byte Magazine is probably much more readable and enjoyable.

Applications of Huffman coding are pervasive throughout computer science. This coding scheme is not limited to encoding messages in English text. In fact, Huffman coding can be used to compress parts of both digital photographs and other files such as digital sound files (MP3) and ZIP files. If you have ever used a JPEG picture file then you have probably used Huffman coding without even knowing it. In the case of JPEG files the main compression scheme uses a discrete cosine transform, but Huffman coding is used as a minor player in the overall JPEG format. There are of course many other file compression techniques besides Huffman coding, but next to run-length encoding, Huffman coding is one of the simplest forms of file compression. If you want to understand other more sophisticated compression schemes such as arithmetic coding and Lempel-Ziv-Welch coding you will be better served by first mastering Huffman coding. Huffman coding can be used effectively anywhere there is a need for a compact code to represent a long series of a relatively small number of distinct bytes.

The following code fragments written in Pascal should give you a good idea of what is required just to write and to read back the Huffman Coding tree. In addition to these routines you need to write bit-oriented file I/O routines and you need to write the main Huffman encoding and Huffman decoding routines. In the code fragments below, comments are delimited by curly braces. You call the procedure **`WriteTheCodeTree`** only once using the root node of the entire tree as the input parameter. You call the function **`ReadTheCodeTree`** only once and you assign the returned value from that function to the variable that is to hold the pointer to the root node.

```
procedure WriteTheCodeTree(NextNode : NodePointerType);
begin
  { a procedure called WriteABit writes a single bit to the output stream }
  { a procedure called Write8Bits writes a byte to the output stream }
  if NextNode.LeftChild <> NIL then
    begin
      { If you get here, NextNode is NOT a leaf node. }
      { A 1-bit will later (on input) tell us it is not a leaf node. }
      WriteABit(1);
      WriteTheCodeTree(NextNode.LeftChild);
      WriteTheCodeTree(NextNode.RightChild)
    end
    else
      begin
        { If you get here, NextNode is a leaf node. }
        { A 0-bit will later (on input) tell us it is a leaf node }
        WriteABit(0);
        Write8Bits(NextNode.Character)
      end
end;  {procedure WriteTheCodeTree}


function ReadTheCodeTree : NodePointerType;
var  LeftChild  : NodePointerType;
     RightChild : NodePointerType;
     NewNode    : NodePointerType;
     Character  : byte;
begin
  { the function called ReadABit returns the next bit 0 or 1 from the input stream }
  { the function Read8Bits returns one ASCII character from the input stream }
  if ReadABit=1 then
    begin     { the bit just read was a 1 bit }
      LeftChild := ReadTheCodeTree;
      RightChild := ReadTheCodeTree;
      New(NewNode);  { allocate memory for a new non-leaf node }
      NewNode.LeftChild := LeftChild;
      NewNode.RightChild := RightChild;
      ReadTheCodeTree := NewNode
    end
    else
      begin   { the bit just read was a 0 bit }
        Character := Read8Bits;
        New(NewNode);  { allocate memory for a new leaf node }
        NewNode.Character := Character;
        NewNode.LeftChild := nil;
        NewNode.RightChild := nil;
        ReadTheCodeTree := NewNode
      end
end;  {function ReadTheCodeTree}
```