Ternary Tree & A New Huffman Decoding Technique

Dr. Pushpa R.Suri † and Madhu Goel ††,

Department of Computer Science & Applications, Kurukshetra University, Kurukshetra, India

Summary
In this paper, the focus is on the use of ternary tree over binary tree. First of all, we give the introduction of Huffman's coding. Then Huffman decoding is discussed. Here, a new one pass Algorithm for Decoding adaptive Huffman ternary tree codes is implemented. To reduce the memory size and speed up the process of searching for a symbol in a Huffman ternary tree, we purpose a memory efficient array data structure to decode the binary codeword. Here we develop two algorithms. In first algorithm we use Huffman ternary tree with height h with binary codeword which results out the corresponding symbol of the given codeword in very short time and requires less memory. In the second algorithm, we use array data structure to decode the binary codeword. Both algorithms show totally new formulas and require less effort.

Keywords:
Ternary tree, Huffman's Algorithm, Adaptive Huffman coding, Huffman decoding, prefix codes, compression ratio, error detecting & correcting

1. INTRODUCTION:

Ternary tree or 3-ary tree is a tree in which each node has either 0 or 3 children (labeled as LEFT child, MID child, RIGHT child).

Huffman coding is divided in to two categories:-

1. Static Huffman coding
2. Adaptive Huffman coding

Static Huffman coding suffers from the fact that the uncompressed need have some knowledge of the probabilities of the symbol in the compressed files. This can need more bits to encode the file. If this information is unavailable, compressing the file requires two passes. FIRST PASS finds the frequency of each symbol and constructs the Huffman tree. SECOND PASS is used to compress the file. We already use the concept of static Huffman coding [12] using ternary tree And we conclude that representation of Static Huffman Tree [12] using Ternary Tree is more beneficial than representation of Huffman Tree using Binary Tree in terms of number of internal nodes, Path length [8], height of the tree, in memory representation, in fast searching and in error detection & error correction. Static Huffman coding methods have several disadvantages.

Therefore we go for adaptive Huffman coding.

Adaptive Huffman coding calculates the frequencies dynamically based on recent actual frequencies in the source string. Adaptive Huffman coding which is also called dynamic Huffman coding is an adaptive coding technique based on Huffman coding building the code as the symbols are being transmitted that allows one-pass encoding and adaptation to changing conditions in data. The benefits of one-pass procedure is that the source can be encoded in real time, through it becomes more sensitive to transmission errors, since just a single loss ruins the whole code.

Implementations of adaptive Huffman coding: -

There are number of implementations of this method, the most notable are

1. FGK (Faller Gallager Knuth) Algorithm
2. Vitter Algorithm

We already use the concept of FGK Huffman coding [13] using ternary tree And we conclude that representation of FGK Huffman Tree using Ternary Tree is more beneficial than representation of Huffman Tree using Binary Tree in terms of number of internal nodes, Path length [12], height of the tree, in memory representation, in fast searching and in error detection & error correction.

We also already use the concept of Vitter Huffman coding [14] using ternary tree And we conclude that representation of algorithm V Huffman Tree using Ternary Tree is more beneficial than representation of Huffman Tree using Binary Tree in terms of number of internal nodes, Path length [8], height of the tree, in memory representation, in fast searching and in error detection & error correction.

Manuscript received March 5, 2010
Manuscript revised March 20, 2010

All of these methods are defined- word schemes that determine the mapping from source messages to code-words on the basis of a running estimate of the source message probabilities. The code is adaptive, changing so as to remain optimal for the current estimates. In this way, the adaptive Huffman codes responds to locality, in essence, the encoder is learning the characteristics of the source. The decoder must learn along with the encoder by continually updating the Huffman tree so as to stay in synchronization with the encoder. Here we are given the concept of error detection and error correction. And the main point is that, this thing is only beneficial in TERNARY TREE neither in binary tree nor in other possible trees.

Now here we try to use the concept of adaptive Huffman decoding algorithm using ternary tree.

In 1951, David Huffman [2] and his MIT information theory classmates gave the choice of a term paper or a final exam. Huffman hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient. In doing so, the student out did his professor, who had worked with information theory inventor Claude Shannon to develop a similar code. Huffman built the tree from the bottom up instead of from the top down.

Huffman codes are widely used in the area of data compression and telecommunications. Some applications include JPEG [3] picture compression and MPEG video and audio compression. Huffman codes are of variable word length, which means that the individual symbols used to compose a message are represented (encoded) each by a distinct bit sequence of distinct length. This characteristic of the codeword helps to decrease the amount of redundancy in message data, i.e., it makes data compression possible.

The use of Huffman codes [7] affords compression, because distinct symbols have distinct probabilities of incidence. This property is used to advantage by tailoring the code lengths corresponding to those symbols in accordance with their respective probabilities of occurrence. Symbols with higher probabilities of incidence are coded with shorter codeword, while symbols with lower probabilities are coded with longer codeword. However, longer codeword still show up, but tend to be less frequent and hence the overall code length of all codeword in a typical bit string tends to be smaller due to the Huffman coding.

A basic difficulty in decoding Huffman codes is that the decoder cannot know at first the length of an incoming codeword. As previously explained, Huffman codes are of variable length codes. Huffman codes can be detected extremely fast by dedicating enormous amounts of memory. For a set of Huffman code words with a maximum word length of N bits, 2^N memory locations are needed, because N incoming bits are used as an address into the lookup table to find the corresponding code words.

A technique requiring less memory is currently performed using bit-by-bit decoding, which proceeds as follows. One bit is taken and compared to all the possible codes with a word length of one. If a match is not found, another bit is shifted in to try to find the bit pair from among all the code words with word length of two. This is continued until a match is found. Although this approach is very memory-efficient, it is very slow, especially if the codeword being decoded is long.

Another technique is the binary tree search method. In this implementation technique, Huffman tables used should be converted in the form of binary trees. A binary tree is a finite set of elements that is either empty or partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are referred to as left and right sub trees of the original tree. Each element of a binary tree is called a node of the tree. A branch connects two nodes. Nodes without any branches are called leaves. Huffman decoding for a symbol search begins at the root of a binary tree and ends at any of the leaves; one bit for each node is extracted from bit-stream while traversing the binary tree [1]. This method is a compromise between memory requirement and the number of Huffman code searches as compared to the above two methods. In addition, the coding speed of this technique will be down by a factor related to maximum length of Huffman code.

Another technique currently used to decode Huffman codes is to use canonical Huffman codes. The canonical Huffman codes are of special interest since they make decoding easier. They are generally used in multimedia and telecommunications. They reduce memory and decoding complexity. However, most of these techniques use a special tree structure in the Huffman codeword tables for encoding and hence are suitable only for a special class of Huffman codes and are generally not suitable for decoding a generic class of Huffman codes.

As indicated in the above examples, a problem with using variable codeword lengths is the difficulty in achieving balance between speed and reasonable memory usage.

Huffman is a fairly standard compression algorithm, and it is still commonly used. In order to do this you need a very simple tree. The nodes need a char and a number of occurrences (I used an unsigned short in mine). The tree does not need any of the standard BST methods, but you will need to be able to create a tree by merging two existing trees. All data is stored in the leaf nodes, frequency information is stored in every node in the tree.

- The message "go eagles" requires 144 bits in Unicode but only 38 using Huffman coding
- A Huffman tree is a binary tree [10] used to store a code that facilitates file compression

There are basically two concepts in Huffman coding

- Huffman Encoding
- Huffman Decoding

2. HUFFMAN ENCODING:

This is a two pass problem. The first pass is to collect the letter frequencies. You need to use that information to create the Huffman tree. Note that char values range from -128 to 127, so you will need to cast them. I stored the data as unsigned chars to solve for this problem, and then the range is 0 to 255.

Open the output file and write the frequency table to it. Open the input file, read characters from it, gets the codes, and writes the encoding into the output file.

Once a Huffman code has been generated, data may be encoded simply by replacing each symbol with its code.

3. HUFFMAN DECODING:-

This can be done in one pass. Open the encoded file and read the frequency data out of it. Create the Huffman tree [14] base on that information (The total number of encoded bytes is the frequency at the root of the Huffman tree.). Read data out of the file and search the tree to find the correct char to decode (a 0 bit means go left, 1 go right for binary tree and 00 bit means go left, 01 bit means go mid, 10 bit means go right in case of ternary tree) This gets tricky since you read in 8 bit blocks, but the codes can be shorter or longer than that and there are no separators.

If you know the Huffman code for some encoded data, decoding may be accomplished by reading the encoded data one bit at a time. Once the bits read match a code for symbol, write out the symbol and start collecting bits again.

Huffman codes to binary data

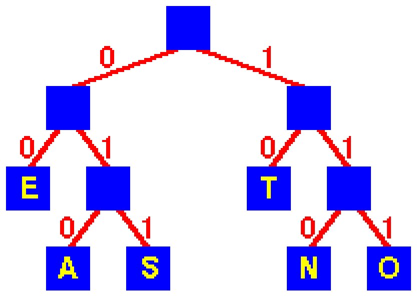
Since they are arbitrary in length, Huffman codes can be difficult to represent. The string data type has major advantages, the length [15] can be changed, and characters can be appended to them, or removed from them at either end. While you will probably use strings to represent the codes, you are not going to write a string of ones and zeros to the file. That would defeat the point of the program, which is file compression. You will need to convert from a string of length 8 to a char value which can written to the file, and do there reverse process as well. This problem is that of finding the minimum length bit string which can be used to encode a string of symbols.

One application is text compression:

What's the smallest number of bits (hence the minimum size of file) we can use to store an arbitrary piece of text?

Huffman's scheme uses a table of frequency of occurrence for each symbol (or character) in the input. This table may be derived from the input itself or from data which is representative of the input. For instance, the frequency of occurrence of letters in normal English might be derived from processing a large number of text documents and then used for encoding all text documents. We then need to assign a variable-length bit string to each character that unambiguously represents that character. This means that the encoding for each character must have a unique prefix. If the characters to be encoded are arranged in a binary tree:

Encoding tree for ETASNO



An encoding for each character is found by following the tree from the route to the character in the leaf: the encoding is the string of symbols on each branch followed.

For example:

String	Encoding
TEA	10 00 010
SEA	011 00 010
TEN	10 00 110

We already use the concept of Huffman encoding using ternary tree. Here I try to use the concept of Huffman decoding using ternary tree .we now implemented an algorithm which is used for Huffman decoding.
Huffman codes are widely used and very effective techniques for compressing data. Huffman's algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string (i.e. a codeword). The running time of Huffman algorithm on a set of n characters is o (log n).
Hasemian presented an algorithm [6] to speed up the search process for a symbol in a Huffman tree and to reduce the memory size. He used a tree clustering algorithm to avoid high sparsity of the Huffman tree. However, finding the optimal solution of the clustering problem is still open. Moreover, the codeword of a single side growing [16] Huffman tree is different from the codeword of the original Huffman tree. Later, Chung gave a memory efficient data structure, which needs the memory size 2n-3, to represent the Huffman tree. In this paper, we shall purpose a more efficient algorithm to save memory space.
The remaining part of this paper is organized as follows. In section 4, for easy understanding, we give the idea of the algorithm. After that, a memory efficient version of our algorithm is presented; section 5 contains our conclusion remarks.

4. IDEA OF OUR ALGORITHM:

In this section, we introduce algorithm A without saving any memory space in order to present our idea simply. Then Algorithm B, we shall describe how to implement our algorithm so that the memory requirement is extremely efficient.
Let T be a Huffman tree which contains n symbols. the symbols the leaves of T are labeled from left to right as $s_0, s_1, s_2, \ldots\ldots s_{n-1}$
The level of a node with respect to T is defined by saying that the root has level 0 and other nodes have a level that is one higher than they have with respect to the sub tree of the root which contains them. The largest level is the height of the Huffman tree. The weight of a symbol is defined to be 3^{2h-d} where h is the height of the Huffman tree and d is the depth of the symbol. Let w_i be the weight of the symbol s_i for i=0,1,…….n-1. Define the $count_0=w_0$ and $count_i=count_{i-1} + w_i$ for i=1,2,……n-1. For example see figure 1. The value of w_i, $count_i$ and s_i, i=0, 1, n-1 in the Huffman tree are shown in table 1. Notice that the height h of the Huffman tree is 4

Example based on Algorithm A and Algorithm B

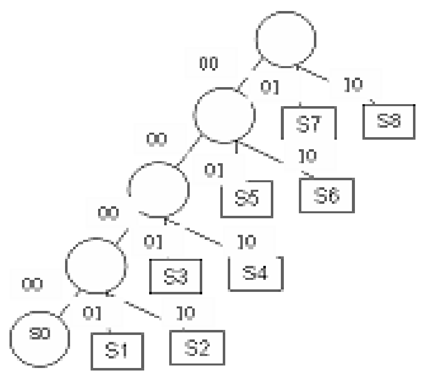


Figure 1

In above Huffman Tree, There are Eight Symbols

Symbols	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
Weight	1	1	1	3	3	9	9	27	27
Count	1	2	3	6	9	18	27	54	81

Table 1

$$weight_i = 3^{2h-d}$$

Where h is the height of the tree
d is the depth of the tree
Count = Cumulative Total

ALGORITHM A

Input:
The value of S_i, W_i and $count_i$, $i = 0, 1, 2 \ldots\ldots\ldots\ldots n-1$, of a Huffman ternary Tree T with height h and a binary codeword c.

Output:-
The corresponding symbol s_k of c.

METHOD:-
STEP 1: Compute

$$t = (c+1).3\frac{2h-d}{2},$$

Where d is the number of binary digits in C.

STEP 2:

Search t if from array count, if t is not in the array count, then c is not a codeword of T; otherwise assume that Count$_k$ = t.

STEP 3:

If $w_k \neq 3^{2h-d}$, then c is not a codeword of T; otherwise S_k is the corresponding symbol of codeword c.

End of Algorithm A

Here, weight of a symbol is defined to be 3^{2h-d} (For Single growing Huffman Tree).

If there are more than 3 nodes on a Huffman ternary Tree on a single level, then add 1 in every $(3+1)^{th}$ node, 5th and 6th node weight will be same.

ALGORITHM B

Input:

The arrays S, b and COUNT of a Huffman ternary Tree T with height h and a binary codeword c.

Output: The corresponding symbol S_k of C.

STEP 1:

Compute

$t = (c+1) \times 3 \frac{2h-d}{2}$, where d is the number of binary digits in C.

STEP 2:

Find count$_k$ such that $count_{k-1} < t \leq count_k$

STEP 3:

Compute $x = count_k - count_{k-1}$

STEP 4:

Decompose x in to x_1, x_2 and x_3 such that $x = x_1 + x_2 + x_3$, $x_i = 3^{ei}$, $i = 1, 2, ...$ for some non-negative integer e_i and assume, without loss of generality that $e_1 \leq e_2$.

STEP 5:

Use b_k, x_1, x_2 & x_3 to determine w_a, w_b and w_c which are the weights of S_{3k-1}, S_{3k} and S_{3k+1} respectively.

STEP 6:

If $t = count_k$ and $w_c = 3\frac{2h-d}{2}$ then S_{3k-1} is the corresponding symbol of C.Let $k = 3k-1$ and stop.

STEP 7:

If $t = count_k - w_c$ and $w_b = 3\frac{2h-d}{2}$ then S_{3k} is the corresponding symbol of C, Let $k = 3k$ and stop.

STEP 8:

If $t = count_k - w_b$ and $w_a = 3\frac{2h-d}{2}$ then S_{3h+1} is the corresponding symbol of C, Let $k = 3k+1$ and stop.

Otherwise C is not a codeword of binary code c.

Here:

$$w_i = w_{3k-1} + w_{3k} + w_{3k+1}$$

For $$i = 0,\ 1,\ 2,\ ....\ n-1$$

$$count_i = count_{i-1} + w_i$$

For $i = 0, 1, 2, ...n-1$

Now Apply Algorithm A

ALGORITHM A

Example 1.

Let $C = 000010$
Step 1: Compute

$$t = (C+1).3^{\frac{2h-d}{2}}$$

$$= (2+1).3^{\frac{2\times 4-6}{2}}$$

$$= (3).3^{\frac{8-6}{2}} = (3).3^{\frac{2}{2}} = 9$$

Step 2 :

$$count_4 = 9$$

Step 3:

$$w_4 = 3^{4-3} = 3$$

Therefore S_4 is the corresponding symbol of code word 000010.

Example – 2.

Let $C = 00001101$
Step 1:
Compute

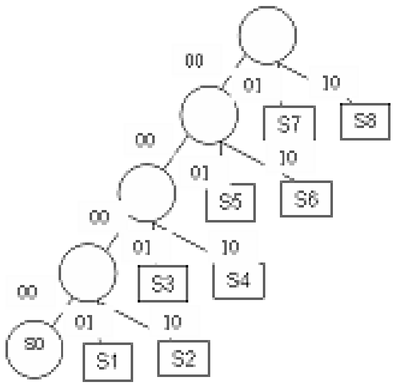
$$t = (C+1).3^{\frac{2h-d}{2}}$$

$$=(14).3^{\frac{8-8}{2}} = 14$$

$d =$ Number of binary digits in C.

Step 2:
We can not find 14 from array count and therefore 00001101 are not a codeword of T.

Algorithm B



There are eight symbols. We are using Ternary Trees. Therefore we divide symbols in to three parts i.e.

$$\frac{0+8}{3} = 2.66 = 3.$$

Therefore $i = 0, 1, 2, 3$

i	0	1	2	3
$count_i$	2	9	54	81
b_i	0	1	0	1

Example

$$w_i = w_{3_{k-1}} + w_{3_k} + w_{3_{k+1}}$$

For $i = 0, 1, 2, ... n-1$

$$count_i = count_{i-1} + w_i$$

For $i = 0, 1, 2, ... n-1$

For i = 0
$W_o = W_{-1} + W_0 + W_1$
$= 1 + 1$
$= 2$
$Count_o = Count_{-1} + W_0$
$= 2$
$Count_o = Count_{-1} + W_0$
$= 0 + 2$
$= 2$
For i= 1
$W_1 = W_2 + W_3 + W_4$
$= 7$
$Count_1 = Count_0 + W_1$
$= 2 + 7 = 9$
For i = 2
$W_2 = W_5 + W_6 + W_7$
$= 45$

$Count_2$ $Count_1 + W_2$
$= 9 + 45$
$= 54$
For i = 3
$W_3 = W_8 + W_9 + W_{10}$
$= 27 + 0 + 0$
$Count_3 = Count_2 + W_3$
$= 54 + 27$
$= 81$
Let C =0001
Step 1:-

$$t = (c+1).3\frac{2h-d}{2}$$

$$= (1+1).3\frac{2*4-4}{2}$$

$$= (2).\ 3\frac{4}{2}$$

$= 29 = 18$

Step 2:
$COUNT_1, < 18 < COUNT_2$
And K = 2

Step 3: X = 54 – 9
= 45

Step 4: Decompose X in to X_1, X_2, X_3
i.e. X_1=9, X_2=9, X_3= 27

Step: 5
W_a = 9(weight of $S_{3k-1} = S_{3x2-1} = S_5$) = 9

W_b = 9 (weight of $S_{3k} = S_6 = 9$)

W_c = 27 (weight of $S_{3k+1} = S_7 = 27$)

Respectively

Step 6: t = 18

$Count_2 - W_a$
$Count_2 - W_b$
$Count_2 - W_c$

Step: 7 if t = $Count_k$
$COUNT_2$ =

$$Wc = 3^{\frac{2h-d}{2}} = 3^{\frac{8-4}{2}} = 9$$

$= S_{3k-1} = S_5$ is corresponding

Symbol.

4.1 Benefits of Ternary decoding algorithm Over Binary decoding algorithm:

If we will represent the same data item with same weights in Binary Tree as well as in Ternary Tree then we can easily point out the comparison between two representation as follows: -

IN TERNARY TREE: -

- Level of the tree = 4
- Height of the tree = 4
- x is decomposed into three parts x_1, x_2 and x_3
- Weight= 3^{h-l}
- t = (c+1). $3^{(2h-d)/2}$
- i=(w_0+…..w_n)/3
- $w_i = w_{3k-1} + w_{3k} + w_{3k+1}$
- $count_i = count_{i-1} + w_i$
- Number of Internal nodes= 4
- Total Number of Nodes (Internal + External) = 13
- Searching on Node is fast

WHILE IN BINARY TREE: -

- Level of the tree = 8
- Height of the tree = 8
- Weight= 2^{h-l}
- x is decomposed into two parts x_1 and x_2
- t = (c+1) 2^{h-d}
- i= (w_0+w_n)/2
- $w_i = w_{2k} + w_{2k+1}$
- $count_i = count_{i-1} + w_i$
- Number of Internal Nodes = 8
- Total Number of Nodes (Internal + External) = 17
- Searching on Node is slow

5. CONCLUSION:

We conclude that our algorithm can be done in less than o(log n) time and needs memory space less than n + [n/2] + [n/2 log n] + 1 which is required in the case of Huffman binary tree. Moreover our algorithm can also be parallelized easily. We can conclude that decoding the binary codeword generated by Huffman ternary tree by using above two algorithms requires less time and it is more efficient than the algorithm which is given by R. Hasemian for Huffman binary tree. We already showed that representation of Huffman Tree using Ternary tree is more beneficial than representation of Huffman Tree using Binary tree in terms of number of internal nodes, Path length, height of the tree, in memory representation, in fast searching and in error detection & error correction.

6. ACKNOWLEDGEMENT:

The author Madhu Goel would like to thank Kurukshetra University Kurukshetra for providing me University Research Scholarship & support of Kurukshetra Institute of Technology & Management.

REFERENCES:

[1] BENTLEY, J. L., SLEATOR, D. D., TARJAN, R. E., AND WEI, V. K. A locally adaptive data compression scheme. Commun. ACM 29,4 (Apr. 1986), 320-330.
[2] DAVID A. HUFFMAN, Sept. 1991, profile Background story: Scientific American, pp. 54-58
[3] ELIAS, P. Interval and regency-rank source coding: Two online adaptive variable-length schemes. IEEE Trans. InJ Theory. To be published.
[4] FALLER, N. An adaptive system for data compression. In Record of the 7th Asilomar Conference on Circuits, Systems, and Computers. 1913, pp. 593-591.
[5] GALLAGER, R. G. Variations on a theme by Huffman. IEEE Trans. Inj Theory IT-24, 6 (Nov.1978), 668-674.
[6] Hashemain, "memory efficient and high-speed search Huffman Coding" IEEE Trans. Communication 43(1995) pp. 2576-2581.
[7] Hu, Y.C. and Chang, C.C., "A new losseless compression scheme based on Huffman coding scheme for image compression",
[8] KNUTH, D. E, 1997. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd edition. Reading, MA: Addison-Wesley, pp. 402-406
[9] KNUTH, D. E. Dynamic Huffman coding. J. Algorithms 6 (1985), 163-180.
[10] MacKay, D.J.C., Information Theory, Inference, and Learning Algorithms, Cambridge University Press, 2003.
[11] MCMASTER, C. L. Documentation of the compact command. In UNIX User's Manual, 4.2 Berkeley Software Distribution, Virtual VAX- I Version, Univ. of California, Berkeley, Berkeley,Calif., Mar. 1984. ,
[12] PUSHPA R. SURI & MADHU GOEL, Ternary Tree & A Coding Technique, IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.9, September 2008
[13] PUSHPA R. SURI & MADHU GOEL, Ternary Tree & FGK Huffman Coding Technique, IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.1, January 2009
[14] PUSHPA R. SURI & MADHU GOEL, A NEW APPROACH TO HUFFMAN CODING, Journal of Computer Science. VOL.4 ISSUE 4 Feb. 2010 .
[15] ROLF KLEIN, DERICK WOOD, 1987, on the path length of Binary Trees, Albert-Lapwings University at Freeburg.
[16] ROLF KLEIN, DERICK WOOD, 1988, On the Maximum Path Length of AVL Trees, Proceedings of the 13th Colloquium on the Trees in Algebra and Programming, p. 16-27, March 21-24.
[17] SCHWARTZ, E. S. An Optimum Encoding with Minimum Longest Code and Total Number of Digits. If: Control 7, 1 (Mar. 1964), and 37-44.
[18] TATA MCGRAW HILL, 2002 theory and problems of data structures, Seymour lipshutz, tata McGraw hill edition, pp 249-255
[19] THOMAS H. CORMEN, 2001 Charles e. leiserson, Ronald l. rivest, and clifford stein.
[20] Thomas H.Cormen Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Section 16.3, pp. 385–392.

Dr. Pushpa Suri is a reader in the department of computer science and applications at Kurukshetra University Haryana India. She has supervised a number of Ph.D. students. She has published a number of research papers in national and international journals and conference proceedings.

Mrs. Madhu Goel has Master's degree (University Topper) in Computer Science. At present, she is pursuing her Ph.D. and working as Lecturer in Kurukshetra Institute of Technology & Management (KITM), Kurukshetra University Kurukshetra. Her area of research is Algorithms and Data Structure where she is working on Ternary tree structures. . She has published a number of research papers in national and international journals.