Study on Huffman Coding

Submitted By

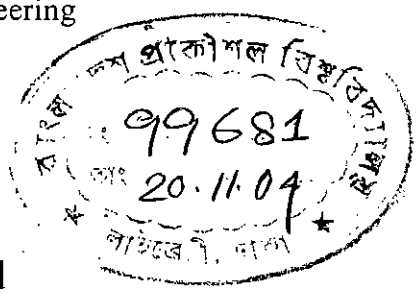
Mohammad Nurul Huda

M. Sc. Engineering Student

Department of Computer Science and Engineering

Student Id: 100105006P

A thesis submitted to the Department of Computer Science and Engineering in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Science and Engineering



Supervised by

Dr. M. Kaykobad

Professor, Department of CSE, BUET

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

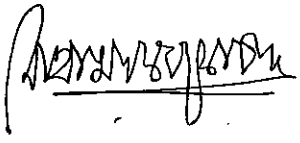
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DHAKA, BANGLADESH

OCTOBER 2004

Certificate

This is to certify that I have done the thesis entitled "Study on Huffman Coding", under the supervision of Dr. M. Kaykobad and it has not been submitted elsewhere for the award of any degree or diploma.



(Dr. M. Kaykobad)
Professor
Department of CSE, BUET
Dhaka-1000, Bangladesh

(Mohammad Nurul Huda)
M. Sc. Engineering Student
Student Id: 100105006P
Department of CSE, BUET
Dhaka-1000, Bangladesh

STUDY ON HUFFMAN CODING

A thesis submitted by

Mohammad Nurul Huda
Student Id: 100105006P
for the partial fulfillment of the degree of
M. Sc. Engineering (Computer Science and Engineering).
Examination held on October 9, 2004.

Approved as to style and contents by:

Dr. M. Kaykobad
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh

Chairman and
Supervisor

Dr. Md. Shamsul Alam
Professor and Head
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh

9/10/04

Ex-officio
(Member)

Dr. Mostafa Akbar
Assistant Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh

Member

Dr. Muhammed Zafar Iqbal
Professor and Head
Department of Computer Science and Engineering
Shahjalal University of Science and Technology
Sylhet, Bangladesh

External
(Member)

ABSTRACT

Since the discovery of the Huffman encoding scheme in 1952, Huffman codes have been widely used in the efficient storing of data, image and video. Huffman coding has been subjected to numerous investigations in the past 60 years. Many techniques have been proposed since then. But still this is an important field as it significantly reduces storage requirement and communication cost. In this thesis, "**Study on Huffman Coding**", we have directed our works mainly to Repeated Huffman Coding that is a lossless compression technique.

The application of Huffman coding technique again and again on a file, then it is called Repeated Huffman coding. While it is expected that encoded message length will be smaller in every pass of Repeated Huffman coding, nevertheless encoding the tree itself will be an overhead in each pass. So repetition count will depend upon how efficiently we can represent a Huffman tree.

A memory efficient representation of a Huffman tree is suggested in this thesis. This representation is very important in the context of Repeated Huffman coding where ultimate compression ratio depends upon the efficiency of encoding the Huffman tree. It reduces overhead which incurs in every pass of Repeated Huffman coding. Because of reduction of overhead, the Huffman Coding scheme can be applied effectively on a file for a larger number of iterations.

The proposed algorithms require less than $\lceil 5n/4 \rceil$ memory spaces in the worst case to represent a Huffman tree for n distinct symbols, which is an improvement over the existing technique that requires at most $\lceil 3n/2 \rceil$ memory spaces.

Such a saving in the representation of Huffman trees will improve the performance of the Repeated Huffman coding as well as Block Huffman coding where multiple tree headers must be stored for a file of large size.

The tree clustering algorithm is also analyzed to get an optimal clustering of a Huffman tree so that along with reduction of memory requirement search process for a symbol in a Huffman tree is speeded up.

ACKNOWLEDGEMENTS

The author likes to express his heartiest and more sincere gratitude to the patrons of this work, without whom it was impossible for him to accomplish this onerous task.

This work is mostly indebted to Dr. M. Kaykobad, Professor, Department of Computer Science and Engineering, BUET, who has supervised the whole work. It is a great pleasure for the author to acknowledge his profound gratitude to his supervisor for constant advice, constructive criticisms and valuable guidance. His encouragement helped in every stage of accomplishment of this work. He also provided the author with valuable research materials of this vast field of study. His immense help guided the author to complete the work and to prepare this thesis. Without his patience, concern, and efforts this thesis would have not been attainable.

The author likes to thank Dr. Md. Shamsul Alam, Professor and Head, Department of Computer Science and Engineering, BUET, for his valuable advice and encouragement.

The author likes to thank Dr. Muhammed Zafar Iqbal, Professor and Head, Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Sylthet, for agreeing to be an external examiner.

The author also likes to thank Dr. Mostafa Akbar, Assistant Professor, Department of Computer Science and Engineering, BUET, for his prompt help and valuable advice.

The author's ultimate tribute goes to the Almighty Allah for bringing this work on light.

(Mohammad Nurul Huda)

Contents

ABSTRACT iv

ACKNOWLEDGEMENT v

LIST OF TABLES viii

LIST OF FIGURES x

CHAPTER 1 INTRODUCTION

1.1 Introduction to Data Compression Techniques 1
1.2 Literature Review 2
1.3 Scopes and Objectives of the Thesis 3
1.4 Organization of the Thesis 4

CHAPTER 2 CODING TECHNIQUES

2.1 Introduction to Coding Techniques 5
2.2 Different Coding Techniques 5
2.3 Different Variable Length Coding Techniques 6
2.3.1 Shanon-Fano Coding Technique 6
2.3.2 Huffman Coding Technique 8
2.4 Arithmetic Coding Technique 11
2.5 Variations of Huffman Coding 13
2.5.1 Static Huffman Coding 13
2.5.2 Dynamic Huffman Coding 14
2.5.3 Block Huffman Coding 18
2.5.4 Repeated Huffman Coding 19
2.6 Conclusion 22

CHAPTER 3 REPRESENTATION OF HUFFMAN TREES

3.1 Introduction to Huffman Tree Representation 23
3.2 Existing Representation Techniques 23
3.2.1 Circular Leaf node Technique 23
3.2.2 Code for Every External node 25
3.3 Proposed Methods of a Huffman Tree Representation 29
3.3.1 Level Order Technique 29
3.3.2 Modified Level Order Technique 32
3.3.3 Preorder Technique 36
3.3.4 Modified Preorder Technique 39
3.3.5 Single Side Growing Tree Technique 42
3.3.6 Balanced Binary Tree Technique 45
3.4 Conclusion 50

CHAPTER 4 HUFFMAN TREE CLUSTERING

4.1 Introduction to Tree Clustering 51
4.2 Problems of Huffman Codes 51
4.3 Desirable Features of Huffman Codes 51
4.4 Solutions of Huffman Code Related Problems 52
4.5 Problems of the Tree Clustering Algorithm 52
4.6 Example of Huffman Tree Clustering 53
4.7 Algorithms Related to Tree Clustering 59
4.8 Decision Regarding Top Cluster 60
4.9 Conclusion 60

CHAPTER 5 BLOCK HUFFMAN CODING

5.1 Introduction 61
5.2 Proposed Algorithms for Block Huffman Coding 61
5.3 Conclusion 63

CHAPTER 6 DESIGN OF EXPERIMENTS AND RESULTS

6.1 Introduction 64
6.2 A Huffman Tree Storage Format in the Compressed File 65
6.3 Experimental Results 67
6.3.1The Existing Technique 67
6.3.2 The Proposed Techniques 69
6.3.3 Experimental data for Benchmark file 82
6.3.4 Block Huffman Coding 89
6.4 Comparison among Experimental Results 90
6.5 Analysis of Experimental Results 108
6.6 Conclusion 109

CHAPTER 6 CONCLUSION AND RECOMMENDATIONS

7.1 Conclusion 110
7.1.1 Huffman Tree Representation Techniques 110
7.1.2 Tree Clustering Technique 112
7.1.3 Block Huffman Coding 112
7.2 Recommendations 112

REFERRENCES 114

List of Tables

CHAPTER 2

2.1	Fixed Length Codes	5
2.2	Frequency Count for Shannon-Fano Coding	7
2.3	First Frequency Division for Shannon-Fano Coding	7
2.4	Second Frequency Division for Shannon-Fano Coding	7
2.5	Third Frequency Division for Shannon-Fano Coding	8
2.6	Fourth Frequency Division for Shannon-Fano Coding	8
2.7	Frequency Count for Huffman Coding	9
2.8	Huffman Code for Every Character	11
2.9	Probability Distribution for Arithmetic Coding	12
2.10	Arithmetic Encoding	12
2.11	Arithmetic Decoding	13

CHAPTER 3

3.1	Huffman Code Length for Every Symbol	44
3.2	Huffman Code for Every Symbol	44

CHAPTER 4

4.1	Memory Efficiency at Different Levels of Huffman tree	53
4.2	Frequency Count for Huffman Coding	53
4.3	Reduction Process in Huffman Coding	54
4.4	Huffman Code Lengths for Every Symbol	54
4.5	Similar Huffman Code Length for Different Symbols	55
4.6	Huffman Codeword for Every Symbol	55
4.7	Memory Efficiency at Different Levels of Huffman tree	56
4.8	Memory Space	57
4.9	Super Table for Clustered trees	57
4.10	Look-up Table for γ Cluster	57
4.11	Look-up Table for α Cluster	57
4.12	Look-up Table for β Cluster	58
4.13	Look-up Table for δ Cluster	58
4.14	Memory Mapping in Tree Clustering	58
4.15	Decoding in Tree Clustering	59

CHAPTER 6

6.1	Compression Ratio for Circular Leaf node Technique	67
6.2	Huffman tree Size for Circular Leaf node Technique	67
6.3	Standard Deviation for Circular Leaf node Technique	68
6.4	Compressed File Size for Circular Leaf node Technique	68
6.5	Average Code Length for Circular Leaf node Technique	69
6.6	Compression Ratio for Level order Technique	69
6.7	Huffman tree Size for Level order Technique	70
6.8	Standard Deviation for Level order Technique	70
6.9	Compressed File Size for Level order Technique	71
6.10	Average Code Length for Level order Technique	71
6.11	Compression Ratio for Modified Level order Technique	72
6.12	Huffman tree Size for Modified Level order Technique	72
6.13	Standard Deviation for Modified Level order Technique	73

List of Tables

6.14	Compressed File Size for Modified Level order Technique	73
6.15	Average Code Length for Modified Level order Technique	74
6.16	Compression Ratio for Preorder Technique	74
6.17	Huffman tree Size for Preorder Technique	75
6.18	Standard Deviation for Preorder Technique	75
6.19	Compressed File Size for Preorder Technique	76
6.20	Average Code Length for Preorder Technique	76
6.21	Compression Ratio for Modified Preorder Technique	77
6.22	Huffman tree Size for Modified Preorder Technique	77
6.23	Standard Deviation for Modified Preorder Technique	78
6.24	Compressed File Size for Modified Preorder Technique	78
6.25	Average Code Length for Modified Preorder Technique	79
6.26	Compression Ratio for Balanced Binary Tree Technique	79
6.27	Huffman tree Size for Balanced Binary Tree Technique	80
6.28	Standard Deviation for Balanced Binary Tree Technique	80
6.29	Compressed File Size for Balanced Binary Tree Technique	81
6.30	Average Code Length for Balanced Binary Tree Technique	81
6.31	Data for WORLD95.TXT Benchmark File	82
6.32	Data for OHS.DOC Benchmark File	83
6.33	Data for FP.LOG Benchmark File	84
6.34	Data for FLASHMX.PDF Benchmark File	85
6.35	Data for ENGLISH.DIC Benchmark File	86
6.36	Data for A10.JPG Benchmark File	87
6.37	Data for RAFALE.BMP Benchmark File	88
6.38	Compression Ratio and Tree size for Block Huffman(Level order)	89
6.39	Compression Ratio and Tree size for Block Huffman(M. Level order)	89
6.40	Compression Ratio Comparison	90
6.41	Compressed File Size Comparison	90
6.42	Comparison between Pure and Repeated Huffman(Circular Leaf node)	91
6.43	Comparison between Pure and Repeated Huffman(Level order)	91
6.44	Comparison between Pure and Repeated Huffman(Modified Level order)	92
6.45	Comparison between Pure and Repeated Huffman(Preorder)	92
6.46	Comparison between Pure and Repeated Huffman(Modified Preorder)	93
6.47	Comparison between Pure and Repeated Huffman(Balanced Binary Tree)	93
6.48	Comparison between Pure and Block Huffman(Level order)	94
6.49	Comparison between Pure and Block Huffman(M. level order)	94
6.50	Huffman Tree Size in Existing and Level order Technique	95
6.51	Huffman Tree Size in Existing and Modified Level order Technique	95
6.52	Huffman Tree Size in Existing and Preorder Technique	96
6.53	Huffman Tree Size in Existing and Modified Preorder Technique	96
6.54	Huffman Tree Size in Existing and Balanced Binary tree Technique	97
6.55	Repetition Count Comparison	97
6.56	Standard Deviation vs. Compression Ratio(Existing and Level)	98
6.57	Standard Deviation vs. Compression Ratio(Existing and M. Level)	98
6.58	Standard Deviation vs. Compression Ratio(Existing and Preorder)	99
6.59	Standard Deviation vs. Compression Ratio(Existing and M. Preorder)	99
6.60	Standard Deviation vs. Compression Ratio(Existing and Balanced Binary Tree)	100

List of Figures

CHAPTER 1

1.1 Data encoding process 1

CHAPTER 2

2.1 Shannon-Fano tree after First Frequency Division 7
2.2 Shannon-Fano tree after Second Frequency Division 7
2.3 Shannon-Fano tree after Third Frequency Division 8
2.4 Shannon-Fano tree after Fourth Frequency Division 8
2.5 Huffman Tree Construction Process 9-10
2.6 Dynamic Huffman Tree-I 14
2.7 Dynamic Huffman Tree-II 15
2.8 Dynamic Huffman Tree-III 15
2.9 Dynamic Huffman Tree-IV 15
2.10 Huffman Tree Update Operation-I 17
2.11 Huffman Tree Update Operation-II 17
2.12 Huffman Tree Update Operation-III 18
2.13 Structure of Compressed and Uncompressed File in Pure Huffman Coding 19
2.14 Structure of Compressed and Uncompressed File in Block Huffman Coding 19
2.15 Structure of Compressed and Uncompressed File 20
2.16 A Typical Huffman Tree-I from Given Message 20
2.17 A Typical Huffman Tree-II from Given Message 21

CHAPTER 3

3.1 Circular Leaf nodes in Huffman Tree 24
3.2 Structure of Huffman Tree Header for Circular Leaf node Technique 25
3.3 A Typical Huffman Tree 26
3.4 Structure of Huffman Tree Header in Code for Every External node Technique 26
3.5 Received Tree Header for Decoder 26
3.6 Reconstruction Process of Huffman Tree 27-28
3.7 Huffman Tree for Level order Technique 29
3.8 Huffman Tree for Modified Level order Technique 32
3.9 A Balanced Huffman Tree 35
3.10 A Right Side Growing Huffman Tree 36
3.11 Huffman Tree for Preorder Technique 37
3.12 Huffman Tree for Modified Preorder Technique 39
3.13 Left Side Growing Huffman Tree 41
3.14 A Huffman Tree for Best case of Preorder 42
3.15 A Huffman Tree 43
3.16 A Right Side Growing Huffman Tree 44
3.17 A Huffman Tree for 11 Symbols 46
3.18 A Balanced Huffman Tree 47
3.19 A Right Side Growing Huffman Tree 47
3.20 A Huffman Tree for 5 Symbols 48
3.21 A Huffman Tree for 6 Symbols 48
3.22 A Huffman Tree for 7 Symbols 49
3.23 A Huffman Tree for 8 Symbols 49

List of Figures

CHAPTER 4

4.1	Huffman Tree Showing Levels	52
4.2	A Huffman Tree for 7 Symbols	54
4.3	A Single Side Growing Huffman Tree	55
4.4	A Huffman Tree for Clustering	56
4.5	Top Cluster	56
4.6	Cluster for α, β, δ	56
4.7	Super Tree	57

CHAPTER 5

5.1	Uncompressed and Compressed File for Pure Huffman Coding	62
5.2	Block Huffman Coding Compressed File Structure	63

CHAPTER 6

6.1	Tree Header for Level order Technique	65
6.2	Tree Header for Modified Level order Technique	65
6.3	Tree Header for Preorder Technique	65
6.4	Tree Header for Modified Preorder Technique	66
6.5	Tree Header for Circular Leaf node Technique	66
6.6	Tree Header for Balanced Binary Tree Technique	66
6.7	Chart for Compression Ratio in Pure and Repeated Huffman (.TXT File)	100
6.8	Chart for Compression Ratio in Pure and Repeated Huffman (.DOC File)	101
6.9	Chart for Compression Ratio in Pure and Repeated Huffman (.C File)	101
6.10	Chart for Compression Ratio in Pure and Repeated Huffman (.RTF File)	102
6.11	Chart for Compression Ratio in Pure and Repeated Huffman (.HTM File)	102
6.12	Repetition Count (Circular vs. Level order, Modified Level order)	103
6.13	Repetition Count (Circular vs. Preorder, Modified Preorder)	103
6.14	Repetition Count (Circular vs. Balanced Binary Tree Technique)	104
6.15	Standard Deviation vs. Compression Ratio(.TXT, .DOC, .C, .RTF)	104
6.16	Standard Deviation vs. Compression Ratio(.BMP, .EXE, .HTM, .PDF, .BACK)	105
6.17	Compression Ratio vs. Average Code Length(.TXT, .DOC, .C, .RTF)	105
6.18	Huffman Tree Size in Circular vs. Level order(.TXT File)	106
6.19	Huffman Tree Size in Circular vs. Modified Level order(.TXT File)	106
6.20	Huffman Tree Size in Circular vs. Preorder(.TXT File)	107
6.21	Huffman Tree Size in Circular vs. Modified Preorder(.TXT File)	107
6.22	Huffman Tree Size in Circular vs. Balanced Binary Tree(.TXT File)	108

CHAPTER 1

INTRODUCTION

1.1 Introduction to Data Compression Techniques

The main objective of data compression techniques is to transform raw data into a form which is in reduced size but from which original data can easily be extracted. Mainly a compression technique focuses on reduction of storage requirements and communication cost over the network.

Data compression techniques can be divided into two major categories.

i) Lossy data compression
ii) Lossless data compression

Lossy data compression concedes a certain loss of accuracy in exchange for greatly increased compression. Lossy compression proves effective when it is applied to graphical images and digitized voice. Most lossy compression techniques can be adjusted to different quality levels, gaining accuracy in exchange for less effective compression.

Lossless compression technique guarantees to generate an exact duplicate of the input data stream after an encoding or a decoding cycle. This type of compression is used when storing database records, spreadsheets, or word processing files. In these applications, the loss of even a single bit could be catastrophic. All of the techniques discussed in this thesis are lossless.

Data Compression = Modeling + Coding

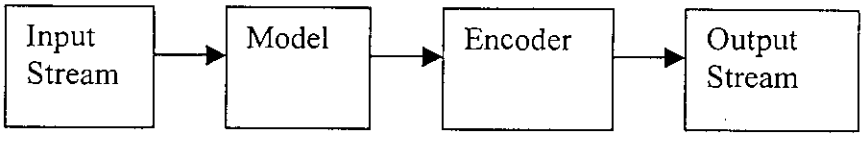


Figure 1.1: Data Encoding Process

A model is simply a collection of data and rules used to process input symbols and determine which code(s) to output. A program uses the model to accurately define the probability for each symbol and the coder to produce an appropriate code based on those probabilities.

A statistical model with Huffman coding counts occurrence of each symbol in the message, then constructs a Huffman tree based upon symbol probabilities and generates a code for each input symbol. Using these codes, Huffman tree and input symbols the encoder builds a compressed output stream.

Data compression enters into the field of Information Theory because of its concern with redundancy. Redundant information in a message takes extra bits to encode, and if we can get rid of that extra information, we will have reduced the size of the message.

Information Theory uses the term entropy as a measure of how much information is encoded in a message. The higher the entropy of a message, the more information it contains. The entropy of a symbol is defined as the negative logarithm of its probability (p).

Information content of a symbol = $-\log_2 p$

The entropy of an entire message is simply the sum of the entropy of all individual symbols.

1.2 Literature Review

Data can be compressed using variable length compression techniques whenever some data symbols are more likely to appear than others. Shannon [28-29] showed that for the best possible compression code in the sense of minimum average code length, output length contains a contribution of $-\log_2 p$ bits from the encoding of each symbol whose probability of occurrence is p. The term redundancy has been defined by Shannon as a property of the code. D. A. Huffman [16] first introduced a minimum redundancy method of source coding called a "Huffman code". Connel [6] derived a Huffman-Shannon-Fano (HSF) code by adopting a notion of Shannon-Fano [9] code and combining it with Huffman code, wherein the code symbols appear lexicographically.

Huffman algorithm requires two passes over the text. This causes delay when used for network communication. In file compression applications the extra disk accesses can slow down the operation. Faller [8] and Gallager [10] independently proposed a one-pass scheme, which has been improved substantially by Knuth [19] for dynamic Huffman codes, usually known as FGK codes. J. S Vitter [33]

has analyzed the dynamic Huffman codes and proposed an optimal algorithm. He also derived a tight upper bound for the Dynamic Huffman codes.

There are some practical problems in using Traditional Huffman coding. One of the most prominent problems is that the whole stream must be read prior to coding. This is a major problem when file size is too large. Mannan and Kaykobad [23] solved the problem through introducing the Block Huffman coding.

In every pass of Repeated Huffman coding, a Huffman tree must be stored in output stream. So an efficient representation of Huffman tree is required. Hashemian [11] presented a tree clustering algorithm to avoid sparsity of Huffman trees. Finding the optimal solution of this clustering problem is still open. Chung [5] gave a data structure requiring memory size of only 2n-3 to represent the Huffman tree, when n is the number of symbols. Chen et al. [3] improved the data structure to reduce the memory requirements to $\lceil 3n/2 \rceil + \lceil (n/2)\log n \rceil + 1$. Chowdhury et al. [4] further improved the data structure to reduce memory requirement to $\lceil 3n/2 \rceil$ by considering circular leaf nodes (node with two external nodes).

Mandal [22] introduced a universal, lossless data compression technique named as Running Difference Method. R. Schack [27] proposed a bound regarding the length of a typical Huffman codeword. Abu-Mostafa and McEliece [1] proposed a maximum codeword length in Huffman codes. Vitter [13] gave a technique of parallel lossless image compression using Huffman and Arithmetic coding. Hu and Chang [15] proposed a new lossless compression scheme based on Huffman coding for image compression. Turpin [31] introduced an efficient finite-state machine implementation of Huffman decoders. Elabdalla and Irshid [7] proposed an efficient bitwise Huffman coding technique based on source mapping.

Langdon [20] introduces Arithmetic coding technique, which was analyzed by Vitter [12] and subsequently implemented by Apiki [2].

1.3 Scopes and Objectives of the Thesis

While it is expected that encoded message length will be smaller in every pass of Repeated Huffman coding, nevertheless encoding the tree itself will be an overhead in each pass. Because of this overhead in every pass, compression ratio cannot be indefinitely improved. An efficient representation of a Huffman tree is required so that the scheme can be economically applied for a larger number of iterations.

The objectives of the thesis are:

- Efficient representation of Huffman tree through less than $\lceil 3n/2 \rceil$ memory spaces, which is very important in the context of Repeated Huffman coding.
- Determination of effective number of iterations for Repeated Huffman coding.
- Improvement of performance of Huffman coding using Block Huffman coding.
- Selecting L (Maximum level for each cluster) to do effective clustering of a Huffman tree to speed up search process for a symbol in a Huffman tree and reduce memory size.

1.4 **Organization of the Thesis**

Chapter One introduces the area of current research work, and states the importance and objective of the work. A discussion on works related to the current one has also been presented. This chapter also focuses on entropy, lossless and lossy compression techniques.

Different Coding Techniques such as Shannon-Fano, Huffman Coding and Arithmetic Coding are given in Chapter Two. This Chapter also concentrates on variations of Huffman coding.

Chapter Three illustrates mainly the proposed methods of representation of a Huffman tree. It also discusses existing techniques of this tree representation.

Chapter Four focuses on Huffman tree clustering algorithm. Reduction of sparsity of Huffman tree, memory mapping, decoding by a clustered tree and optimal cluster length.

Chapter Five illustrates Block Huffman coding. It also focuses on algorithms for source and continuous data, and block size selection.

Chapter Six has been devoted to design of the experiments for the proposed methods of Huffman Tree representation and Block Huffman Coding. It also presents results of experiments, comparisons among different proposed methods over existing methods.

Chapter Seven concentrates on conclusion of the thesis and recommendation for future work

CHAPTER 2

CODING TECHNIQUES

2.1 Introduction to Coding Techniques

Once Information Theory had advanced to where the number of bits of information in a symbol could be determined, the next step was to develop new methods for encoding information. To compress data it is required to encode symbols with exactly the number of bits of information the symbol contains. If the character 'e' only gives us four bits of information, then it should be coded with exactly four bits. If 'x' contains twelve bits, it should be coded with twelve bits.

2.2 Different Coding Techniques

Every character can be coded with same number of bits or with the bits that are required by the information content. Depending on this, coding can be done with the two techniques i) Fixed length Coding ii) Variable length Coding.

Fixed Length Coding

In fixed length coding a character is coded with the same number of bits. By encoding character using EBCDIC or ASCII, we clearly are not going to be very close to an optimum method.

Table 2.1 shows fixed length codes for each of the symbol of the message "CAB"

Table 2.1: Fixed Length Codes

Character	Fixed Length Code
C	01000011
A	01000001
B	01000010

Variable Length Coding

The codes in which the source symbols are encoded with different lengths of code symbols are considered as variable length codes. These codes are becoming

increasingly important as the costs of communication in distributed systems and external storage are beginning to dominate the costs for internal memory and computation. Variable length codes are more efficient in the sense that fewer bits for representation of same piece of information are required on an average than fixed length codes. This can yield tremendous savings in communication system and file compression.

2.3 **Different Variable Length Coding Techniques**

Solving the problem of fixed length coding practitioners of Information Theory use Shanon-Fano and Huffman coding techniques-two different ways of generating variable length codes when given a probability table for a given set of symbols.

The problem with Huffman or Shannon-Fano coding is that they use an integral number of bits in each code. If the entropy of a given character is 2.5 bits, the Huffman code for that character must be either 2 or 3 bits, not 2.5. Because of this Huffman codes cannot be considered as an optimal coding method.

Though variable length coding does not necessarily achieve theoretical lower bound of compression due to using an integral number of bits per code, it is relatively easy to implement.

2.3.1 **Shannon-Fano Coding Technique**

Shannon-Fano coding technique produces instantaneous decodable code words. This technique builds a decoding/encoding tree known as Shannon-Fano tree and it can be built by the algorithm 2.1 from the list of source symbols with a corresponding list of probabilities or frequency counts.

Step 1: Build a node with the total frequency count; this node is the root of the tree.

Step 2: Divide the list of symbols into two parts, with the total frequency counts of the upper half being as close to the total of the bottom half as possible.

Step 3: The weight of upper half of the list is assigned to the right child of the previous node, and the lower half to the left.

Step 4: Recursively apply the same procedure to each half, subdividing groups and adding to preceding nodes until each symbol has become a leaf on the tree.

Algorithm 2.1: Shannon-Fano Tree Construction

The following Tables and Figure 2.1-2.4 describe the construction process of Shannon-Fano tree

Assume source text is "ABCCDDEEE"

Step 1: **Count Frequency for Every Distinct Character**

Table 2.2: Frequency Count

Symbol	Count
A	1
B	1
C	2
D	2
E	3

Step 2: **Shannon-Fano Tree Construction**

Table 2.3: First Frequency Division

Symbol	Count	Code
A	1	0
B	1	
C	2	
Division 1		
D	2	1
E	3	

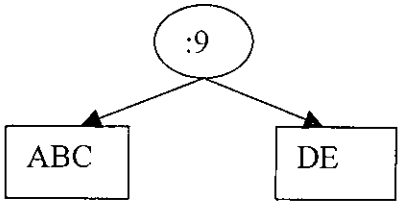


Figure 2.1: Tree after First Division

Table 2.4: Second Frequency Division

Symbol	Count	Code
A	1	0
B	1	
C	2	
Division 1		
D	2	10
Division 2		
E	3	11

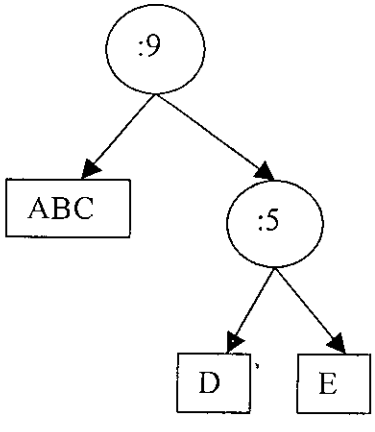


Figure 2.2: Tree after Second Division

Table 2.5: Third Frequency Division

Symbol	Count	Code
A	1	00
B	1	
Division 3		
C	2	01
Division 1		
D	2	10
Division 2		
E	3	11

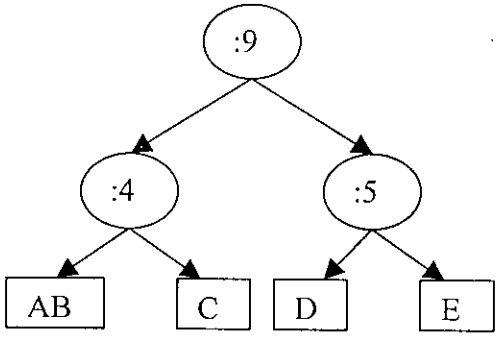


Figure 2.3: Tree after Third Division

Table 2.6: Fourth Frequency Division

Symbol	Count	Code
A	1	000
Division 4		
B	1	001
Division 3		
C	2	01
Division 1		
D	2	10
Division 2		
E	3	11

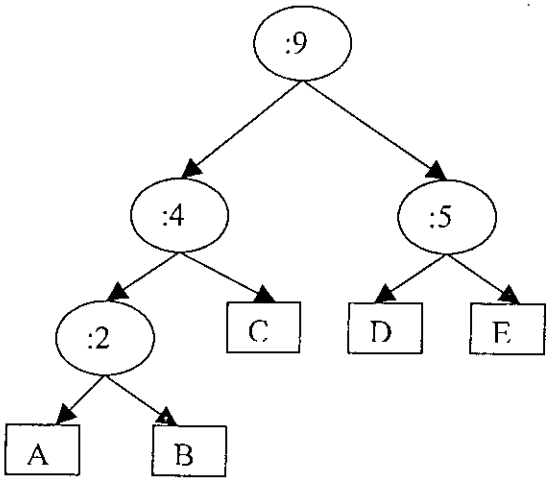


Figure 2.4: Tree after Fourth Division

2.3.2 Huffman Coding Technique

Huffman coding achieves the minimum amount of redundancy possible in a fixed set of variable-length codes. This does not mean that Huffman coding is an optimal coding method. It means that it provides the best approximation for coding symbols when using fixed length coded.

It is a statistical data-compression technique. It produces variable length codes for the source symbols. In this technique, both encoding and decoding are done by a decision tree which is popularly known as Huffman tree, devised by D. A. Huffman. Its application reduces the average code length used to represent the symbols of the source alphabet. Algorithm 2.2 describes how to construct a Huffman tree from a list of symbols with their occurrences in the file.

```
Procedure Huffman(C)
        N=|C|
        Insert(Q, C)
        For i= 1 to N-1 do
                X=MINIMUM(Q)
                Y=MINIMUM(Q)
                Z=CREATE_NODE()
                Z↑.LEFT=X
                Z↑.RIGHT=Y
                Z↑.FREQ= X↑.FREQ+ Y↑.FREQ
                Insert(Q, Z)
        End for
End Procedure
```

Algorithm 2.2: Huffman Tree Construction

The Figure 2.5 and Tables 2.7-2.8 describe the construction process of Huffman tree, generate code for each character of source text and then produce encoded bit stream.

Assume source text is "ENGINEERING"

Step 1: **Count Frequency for Every Distinct Character**

Table 2.7: Frequency Count

Symbol	Count
R	1
I	2
G	2
N	3
E	3

Step 2: **Huffman Tree Construction**

Q: R:1 I:2 G:2 E:3 N:3

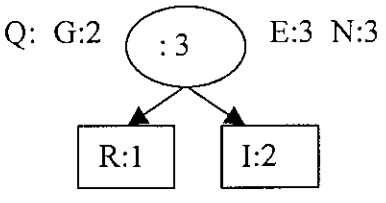


Figure 2.5: Huffman Tree Construction Process

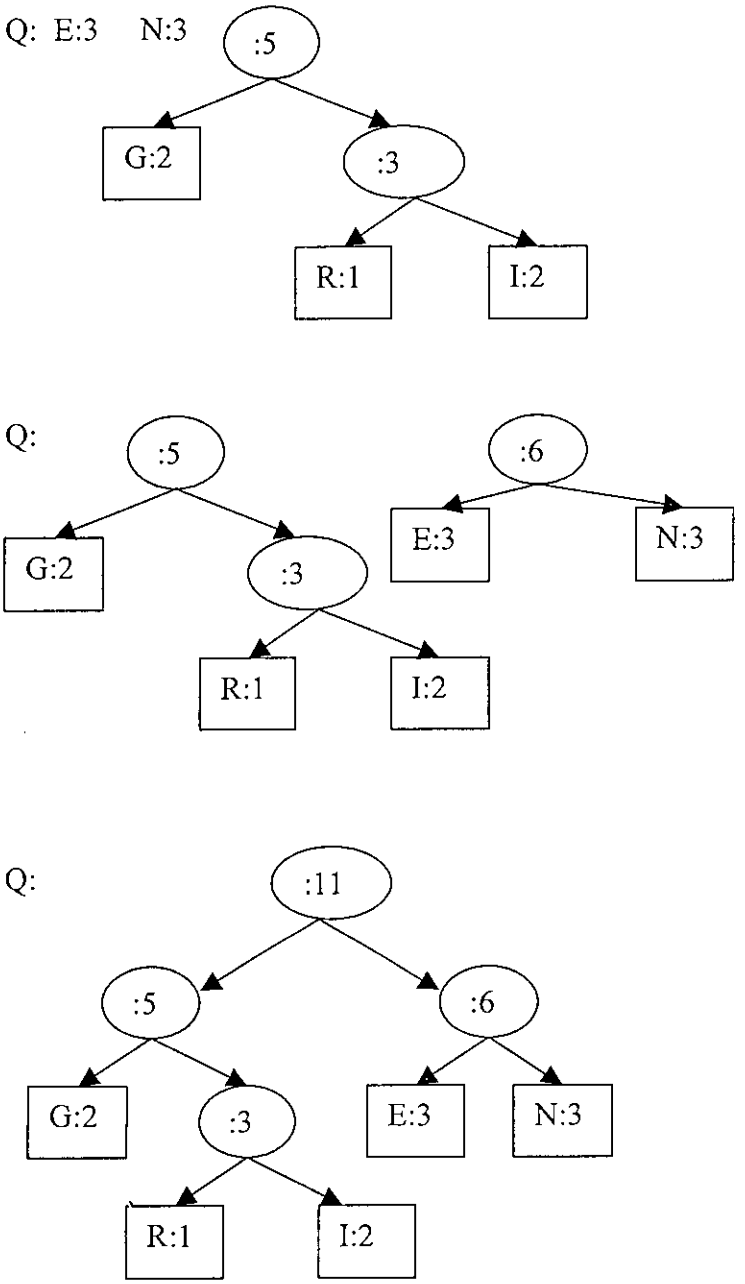


Figure 2.5: Huffman Tree Construction Process (continued)

Step 3: **Code Generation**

Table 2.8: Code for Each Character

Character	Code
E	10
N	11
G	00
I	011
R	010

Step 4: **Encoded Bit Stream**
101100011111010010011110 0

2.4 Arithmetic Coding Technique

Arithmetic coding is a viable successor to Huffman coding. It does not produce a single code for each symbol. Instead, it produces a code for an entire message by incrementally modifying the output code. This is an improvement because the net effect of each input symbol on the output code can be a fractional number of bits instead of an integral number. Algorithm 2.3 shows encoding process in Arithmetic coding technique.

Step 1: A current interval [L, H) initialized to [0, 1).

Step 2: For each symbol of the file, it performs two steps

Step 2.1: Subdivide the current interval into subintervals, one for each possible alphabet symbol. The size of a symbol's subinterval is proportional to the estimated probability that the symbol will be the next symbol in the file, according to the model of the input.

Step 2.2: Select the sub-interval corresponding to the symbol that actually occurs next in the file, and make it the new current interval.

Step 3: Finally output enough bits to distinguish the final current interval from all other possible final intervals.

Algorithm 2.3: Arithmetic Coding

Example

The Tables 2.9-2.11 describe the step by step procedure to get compressed code from a source text.

Assume source text is "CSE DEPTT."

Step 1: **Probability Count and Distribution**

Table 2.9: Probability Distribution

Character	Probability	Range
Space	0.1	[0.00, 0.10)
C	0.1	[0.10,0.20)
D	0.1	[0.20,0.30)
E	0.2	[0.30,0.50)
P	0.1	[0.50,0.60)
S	0.1	[0.60, 0.70)
T	0.2	[0.70, 0.90)
.	0.1	[0.90, 1.00)

Step 2: **Result of the Arithmetic Coding of the Message "CSE DEPTT."**

Table 2.10: Low and High Values

New Character	Low Value	High Value
	0.0000000000	0.1000000000
C	0.1000000000	0.2000000000
S	0.1600000000	0.1700000000
E	0.1630000000	0.1650000000
Space	0.1630000000	0.1632000000
D	0.1630400000	0.1630600000
E	0.1630460000	0.1630500000
P	0.1630480000	0.1630484000
T	0.1630482800	0.1630483600
T	0.1630483360	0.1630483520
.	0.1630483504	0.1630483520

Step 3: **Result of Decoding Process of the Encoded Message**

Table 2.11: Arithmetic Decoding

Encoded Number	Output Character	Range
0.1630483504	C	[0.10,0.20)
0.6304835040	S	[0.60, 0.70)
0.3048350400	E	[0.30,0.50)
0.0241752000	space	[0.00, 0.10)
0.2417520000	D	[0.20,0.30)
0.4175200000	E	[0.30,0.50)
0.5876000000	P	[0.50,0.60)
0.8760000000	T	[0.70, 0.90)
0.8800000000	T	[0.70, 0.90)
0.9000000000	.	[0.90, 1.00)
0.0000000000		

Difficulties of Arithmetic Coding

- Arithmetic coding requires more CPU power than was available until recently. Even now it will generally suffer from a significant speed disadvantage when compared to other coding methods.
- The shrinking of the current interval requires the use of high precision arithmetic.

2.5 Variations of Huffman Coding

Traditional Huffman coding suffers from many problems such as slowness of disk access and network communication. Besides it cannot be continued for an effective number of iterations. As a result it degrades compression ratio. Different variations of Huffman coding are available to improve the performance of Traditional Huffman coding.

2.5.1 Static Huffman Coding

It is a traditional Huffman scheme that collects frequency of occurrence of different symbols to be coded in the first pass, constructs a tree based upon which symbols receive codes. Then in the second pass symbols are coded and sent to the receiver. However, along with codes corresponding to symbols the Huffman tree is also sent.

Problems of Static Huffman Coding

- This technique requires two passes.
- Since it needs two passes, this causes delay when used for network communication.
- In file compression applications the extra disk accesses can slow down the algorithm.

Solutions of Problems of Static Huffman Coding

- Faller (an adaptive system for data compression) and Gallager (Variations on a theme by Huffman) independently proposed a one-pass scheme, later improved substantially by Knuth (Dynamic Huffman coding), for constructing Dynamic Huffman codes. Dynamic Huffman coding solves problems of Static Huffman coding.
- Scott J. Vitter gave an optimum solution of updating a Huffman tree dynamically.

2.5.2 Dynamic Huffman Coding

In Dynamic Huffman coding both the sender and receiver start with the same initial tree and thereafter stay synchronized; they use the same algorithm to modify the tree after each letter is processed. Thus there is never need for the sender to transmit the tree to the receiver, unlike the case of Static Huffman coding.

Example

Dynamic Huffman coding algorithm operates on the following message

"abcd..."

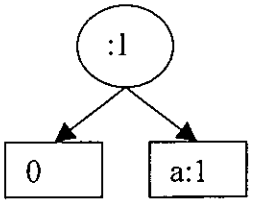


Figure 2.6: Huffman Tree after Receiving 'a'

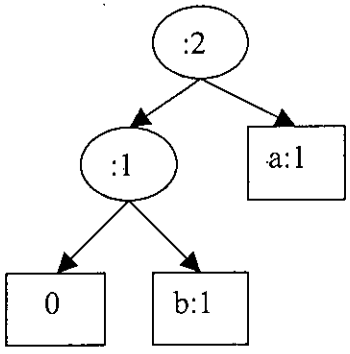


Figure 2.7: Huffman Tree after Receiving 'b'

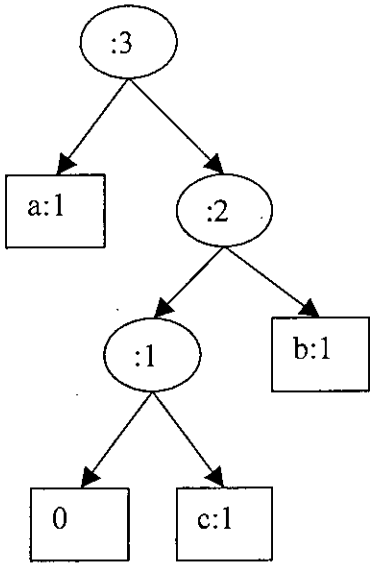


Figure 2.8: Huffman Tree after Receiving 'c'

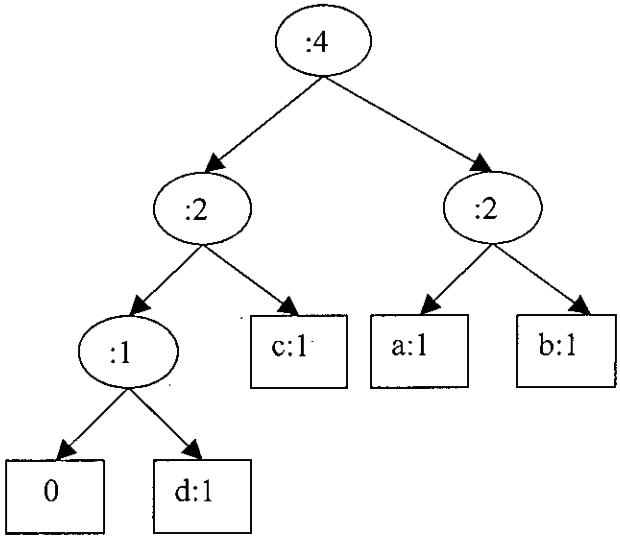


Figure 2.9: Huffman Tree after Receiving 'd'

Sibling Properties

When a symbol comes, tree must be updated to maintain the following properties

- The n leaves have nonnegative weights $W=(w_i \mid i=1..\ n)$, and the weight of each internal node is the sum of the weights of its children.
- The nodes can be numbered in non-decreasing order by weight, so that nodes 2j-1 and 2j is sibling, for $1 \leq j \leq n-1$, and their common parent node is higher in numbering.

Algorithm 2.4 shows how to update a dynamic Huffman tree for each symbol of the message.

```
Procedure Update

    Begin
        q=leaf node corresponding to a_{t+1}
        IF q is the 0-node and k<n-1 then
            Begin
                Replace q by a parent 0-node with two leaf 0-node
                children, numbered in the order left child, right
                child, and parent
                q=right child just created
            End

        IF q is the sibling of a 0-node then
            Begin
                Interchange q with the highest numbered leaf of the
                same weight
                Increment q's weight by 1
                q=parent of q
            End

        WHILE q is not the root of the Huffman tree
            Begin
                Interchange q with the highest numbered node of
                the same weight
                Increment q's weight by 1
                q=parent of q
            End
    End
End Procedure
```

Algorithm 2.4: Dynamic Huffman Tree Construction

Tree Update Operation

The Figures 2.10-2.12 shows the update operation of a Huffman tree.

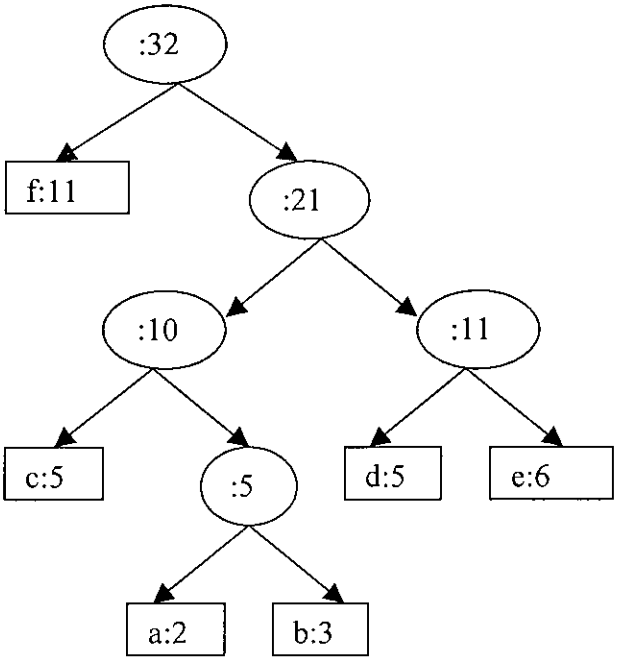


Figure 2.10: Huffman Tree before Updating

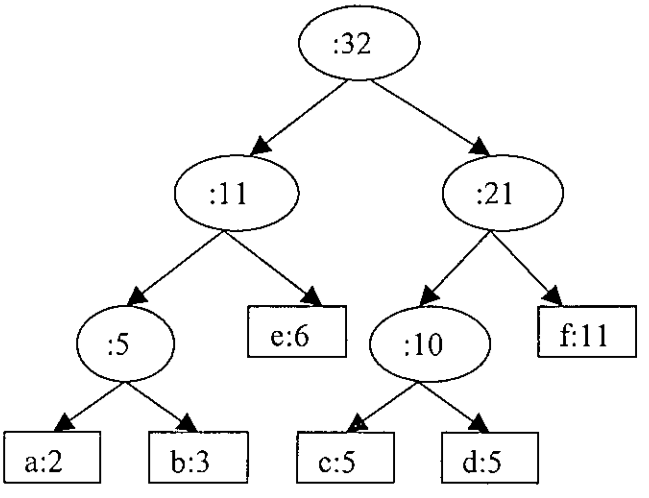


Figure 2.11: Tree Processing to Update after Receiving another 'b'

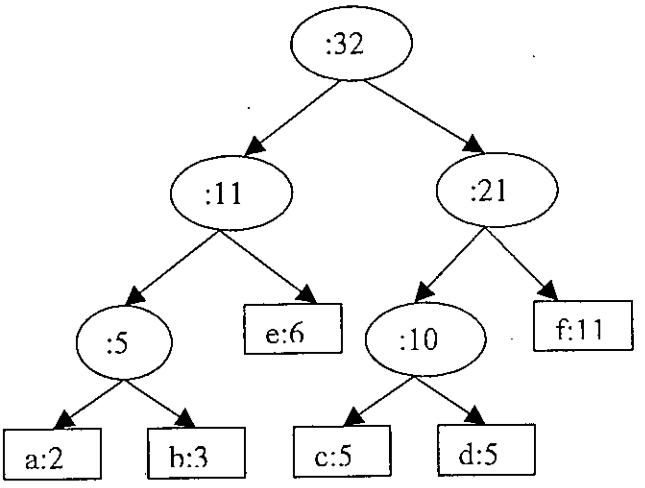


Figure 2.12: Updated Huffman Tree by 'b'

2.5.3 Block Huffman Coding

Although Huffman coding is not the best among existing coding methods, it is the most-cited coding method till present time. Huffman published his paper on coding in 1952, and it instantly became the most imperative work in Information Theory. Huffman's original work spawned many variations. And it dominated the world of coding till the early 1980's.

But there are some practical problems in using Original Huffman coding. One of the most important problems is that we have to read the whole stream prior to coding. This is a major problem when

- File size is too large
- Source stream is continuous

When file size is large, it will take much longer time to build the Huffman header for compression. This happens because it is required to read the whole file twice from the source stream that is in most cases from hard disk. In the first read pass it is required to build the Huffman tree to get a code for each individual character. In the second read pass we do the real work of compression. If file size, which is small, can be stored in main memory, reading in second pass can be done from main memory instead of hard disk. This will reduce overhead time of reading from a slow speed device twice. But when file size is large, it cannot be stored in main memory. In this case it is unavoidable to read second time from hard disk drive. Though currently hard disk drives with data transfer rate 3 to 5 KB/second are available, still it will take significant time. With original method of Huffman coding, there is no way through which we can avoid it. Algorithm 2.5 illustrates how Block Huffman coding works.

Step 1: Read a block from the File into main memory.
Step 2: Build the Huffman tree and code for this Block.
Step 3: Compress this block by reading it from main memory.
Step 4: Put the header and compressed data to output file.
Step 5: If there is more data in input file, goto step1. Otherwise coding is ended

Algorithm 2.5: Block Huffman Coding

How It Solves Our Problems

This method can handle both the drawbacks mentioned above. In the first case, the file is read from hard disk only once; compression speed increases significantly because second pass reading is done from the main memory that is much faster than the hard disk. Now the file size may be as large as possible without suffering double penalty for reading two times from hard disk drive. So first problem mentioned above is solved.

Multiple Header Storage

There is a potential problem of increase in size of compressed data due to the storage space of multiple headers for a single file. The problem may be shown pictorially in the following way

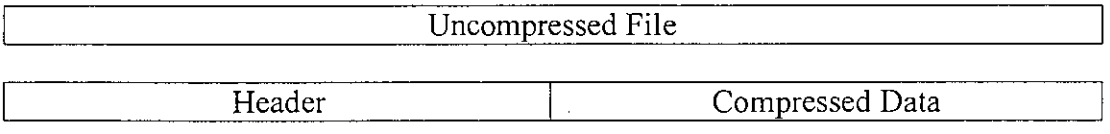


Figure 2.13: Uncompressed and Compressed File in Huffman Coding

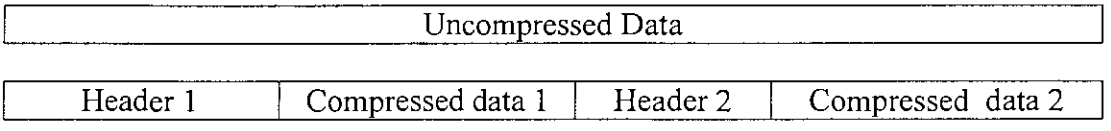


Figure 2.14: Block Huffman Coding

In this method as the number of blocks increases, the overhead for storing multiple headers becomes significant. This causes penalty in compression ratio. That's why a redesign of the storage method of multiple headers is needed.

2.5.4 Repeated Huffman Coding

If Huffman coding technique can be applied effectively on a file again and again,

then it is called Repeated Huffman coding. While it is expected that encoded message length will be smaller in every pass of Repeated Huffman coding, nevertheless encoding the tree itself will be an overhead in each pass. So repetition count will depend upon how efficiently we can represent a Huffman tree. If a Huffman tree can be represented efficiently in memory, Repeated Huffman coding technique can be applied in an effective number of times. If we can do so, compression ratio will be increased.

Before compressed

Uncompressed File

After Compressed

Tree Header	Compressed data

Figure 2.15: Structure of Compressed and Uncompressed File

Example

Assume a file contains total 102 times A, 100 times B and 1 time C.

File content is given below

BB..BCAA..A

Required bits to represent the file = (102+100+1) ×8
=1624

First Pass of Compression:

Huffman tree is just like as follow

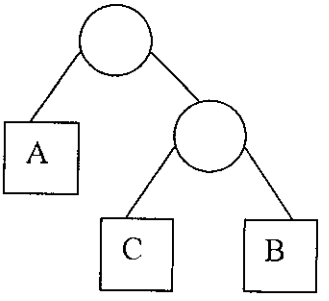


Figure 2.16 Huffman tree for 100 B, 1 C and 102 A

Code for A, C and B are 0, 10 and 11 respectively.

Bits to code the Huffman tree=3(Preorder sequence and then discard right most 0's)

Required bits for compressed code= (100×2+2+102) +Tree Header
= 304+ Code for Tree + Distinct Symbols
= 304+3+3×8
= 331

Format of compressed file in first pass of compression

Code for 100 B's	Code for 1 C's	Code for 102 A's	Tree Code	Distinct Symbols
111..1	10	000..0	101	ABC

s_1, s_2 and s_3 symbols can be defined by taking 8 bits for each symbol where s_1=11111111, s_2=10000000 and s_3=00000000. Symbol s_4 is 101.

Second Pass of Compression:

There are 25 s_1, 1 s_2, 12 s_3, 1 s_4, 1 A, 1 B and 1 C in the source file of second pass.

Huffman tree is just like as follow

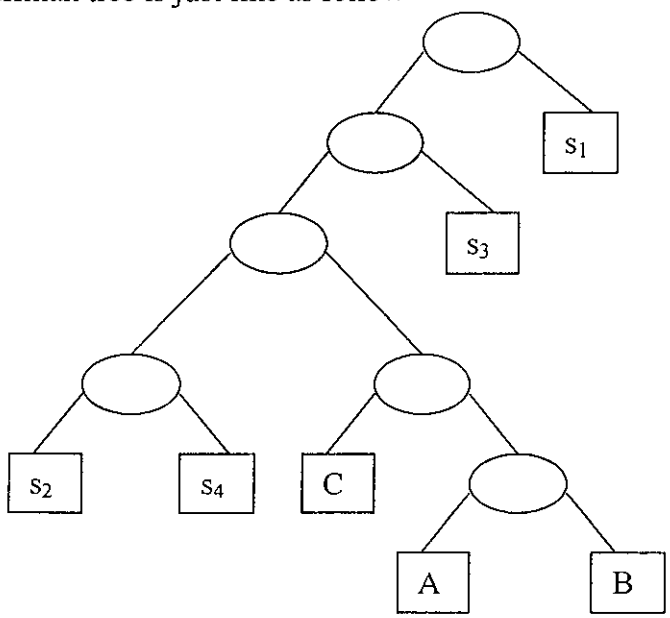


Figure 2.17 Huffman tree for 25 s_1, 12 s_3, and 1 s_2, 1 s_4, 1 A, 1 B, 1 C

Code of s_1, s_2, s_3, s_4, A, B and C are 1, 0000, 01, 0001, 00110, 00111, and 0010 respectively.

Bits to code the Huffman tree=9 (Preorder Sequence and then discard right most 0's)
Required bits for Compressed code= (1×25+4×1+2×12+4+5+5+4) +Tree Header
= 71+ Code for Tree + Distinct Symbols
= 71+9+7×8
= 136

Third Pass of Compression:
This pass cannot be continued because of degeneration.

Compression ratio for Traditional and Repeated Huffman coding is 79.62% and 91.63% respectively. For mentioned text Repeated version compresses 12% more than Traditional Huffman coding.

Problems of Repeated Huffman Coding

An efficient representation of a Huffman tree is needed because the performance of the Repeated Huffman Coding depends on it.

Solution of these Problems

- Chung [5] gave a data structure requiring memory size only 2n-3 to represent the Huffman tree, where n is the number of distinct symbols.

- Chen et al. [3] improved the data structure further to reduce memory requirement to $\lceil 3n/2 \rceil + \lceil n/2\lg n \rceil + 1$.

- Chowdhury et al. [4] improved memory requirements by considering circular leaf nodes (nodes with two adjacent external nodes). Since a Huffman tree has at most $\lceil n/2 \rceil$ circular leaf nodes, their memory requirement is $\lceil 3n/2 \rceil$ at worst case

2.6 Conclusion

Traditional Huffman coding has many problems that are solved by Dynamic Huffman coding, Block Huffman coding and Repeated Huffman coding. Arithmetic coding generates a single code for the whole source message. But this coding technique needs high computational power as well as arithmetic precision.

CHAPTER 3

REPRESENTATION OF HUFFMAN TREES

3.1 Introduction to Huffman Tree Representation

Coding of a Huffman tree is needed to get the tree from decoder side; otherwise extraction of data is not possible. Here coding of a Huffman tree means representation of the tree so that it can be reconstructed by decoder with minimal effort. There are many techniques to represent a Huffman tree. Next sections are devoted to discuss to the existing and proposed techniques of a Huffman tree representation.

3.2 Existing Representation Techniques

Two existing techniques of Huffman tree representation are discussed.

3.2.1 Circular Leaf node Technique

In this technique along with information symbols it is required to send information of all circular leaf nodes (nodes with two adjacent external nodes) that uniquely determines a Huffman tree.

If $\dot{T}$ is a Huffman tree of height h, then $\check{T}$, the truncated Huffman tree, is obtained by removing all leaves (square nodes) and will have height h-1. $\check{T}$ uniquely determines the Huffman tree T since symbols correspond to left and right son nodes of the circular leaf nodes and the other sons of other single-son circular nodes. Then Huffman codes, representing circular leaf nodes of $\check{T}$ uniquely determine T. If there is m circular leaf nodes, m codewords corresponding to

them, and n symbols in order of appearance in the tree will suffice to represent the Huffman tree.

Consider the example of the following Figure. Let the leaf circular nodes of Figure are denoted by C1, C2 and C3 respectively in order. Then corresponding Huffman codes are 00, 0100 and 101. However, each of these codewords can be represented uniquely by integers whose binary representations are these codewords with 1 as prefix. The corresponding integers for C1, C2 and C3 are respectively 4 for 100, 20 for 10100 and 13 for 1101. Along with A to J symbols 4, 20, and 13 are also sent as codes for circular leaf nodes. The receiver can obtain Huffman codes 00, 0100 and 101 by deleting the attached prefix. These bit patterns will enable the receiver to construct all paths to the circular leaf nodes, and thus reconstruct the Huffman tree. So to reconstruct Huffman tree T, the following theorem is used.

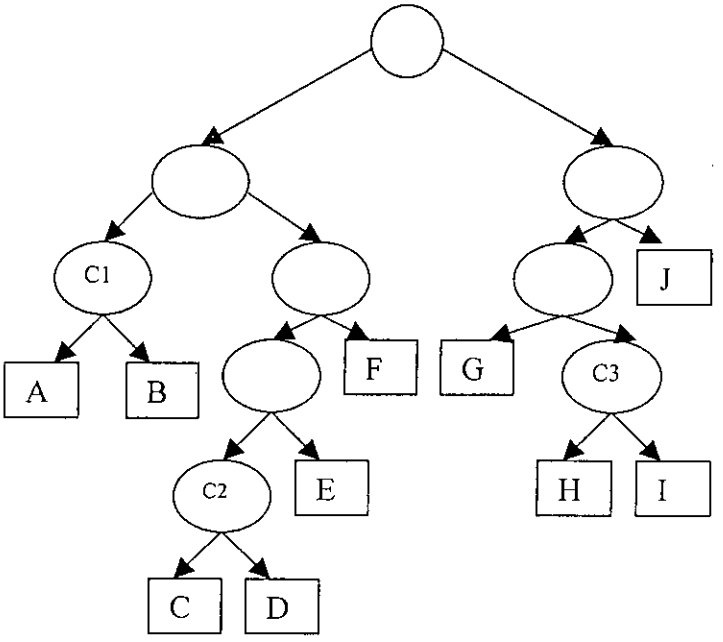


Figure 3.1: Circular Leaf nodes c1, c2 and c3

Theorem 3.1 *Let S be the array of symbols in order of appearance in the Huffman tree T and D be the set of integer codes, corresponding to circular leaf nodes, sent to the receiver. Then the receiver can uniquely reconstruct the Huffman tree T.*

Proof. Let d_i be the integer corresponding to a circular leaf node i. By deleting the appropriate prefix from the binary representation of d_i, one can reconstruct the whole path leading to that node uniquely since bit 0 will take to the left and 1 to the right, and there is no ambiguity. Since each internal node of the Huffman tree

will appear on the paths of some of these circular leaf nodes, each internal node of the Huffman tree will appear in at least one of these paths. Now external nodes will appear either as left or right sons of the circular leaf nodes, or as one of the sons of circular nodes on the path to the circular leaf nodes. This will result in the construction of a unique Huffman tree T since paths constructed for each d_i are unique. Furthermore, the set of symbols S will label the external nodes in an orderly fashion with appropriate symbols. ■

Receiver will receive the following information with compressed data.

Tree Header	Compressed data

Tree header contains the following information

Information Symbols	Codes of Circular leaf node

For the Huffman tree of Figure 3.1, tree header contains the following information

Information Symbols	Codes of Circular leaf nodes		
ABCDEFGHIJ	100	10100	1101

Figure 3.2: Structure of Tree Header

Space Complexity Analysis

Total number of distinct symbols=n
Maximum number of circular leaf node in a Huffman tree=$\lceil n/2 \rceil$
Space for circular leaf node representation=$\lceil n/2 \rceil$
Space for Huffman tree representation = distinct character+ circular leaf node
$= n+\lceil n/2 \rceil$
$= \lceil 3n/2 \rceil$

3.2.2 Code for Every External node

In this method the sender sends a codeword of 0 and 1 for every distinct symbol. The sender also sends all information symbols with codeword. After receiving the tree header, the receiver reconstructs the Huffman tree from information symbols and codeword. Then it starts decoding using constructed Huffman tree.

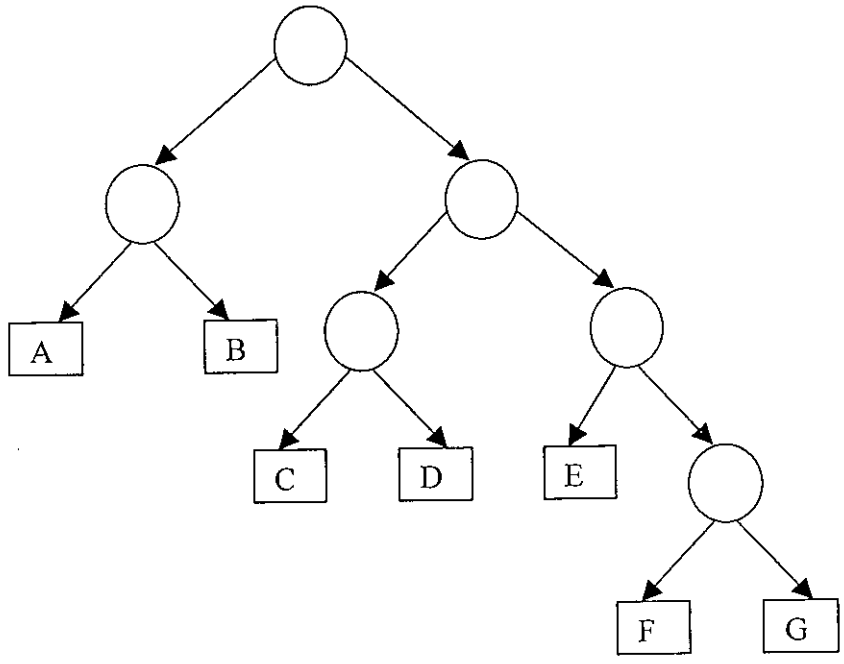


Figure 3.3: A Huffman Tree

The sender sends the following tree header with compressed data to the receiver.

Symbols	A	B	C	D	E	F	G
Code	00	01	100	101	110	1110	1111

Figure 3.4: Structure of Tree Header

A Huffman Tree Extraction Process

When the receiver receives a tree header from the sender, it follows the following steps to reconstruct a Huffman tree

Example

Symbols	A	B	C	D	E	F	G
Code	00	01	100	101	110	1110	1111

Figure 3.5: Received Tree Header

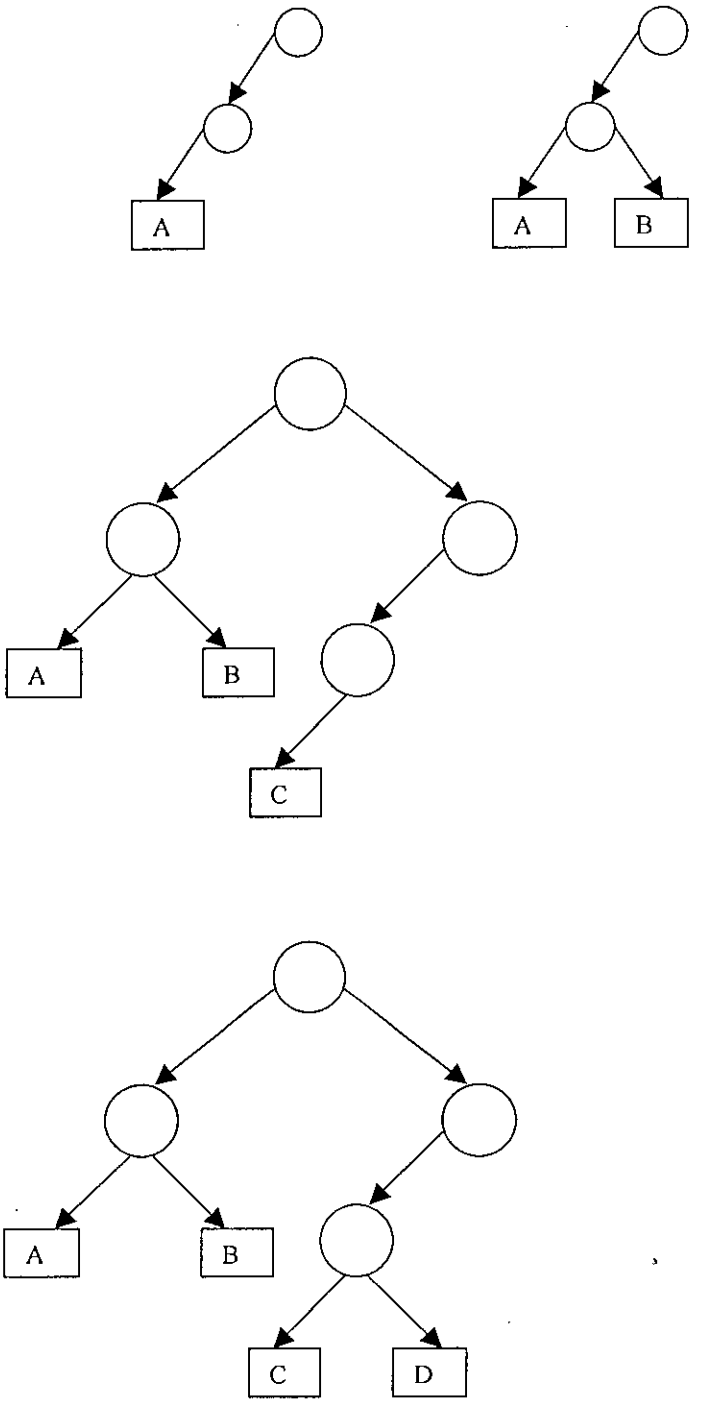


Figure 3.6: Reconstructing Huffman Tree

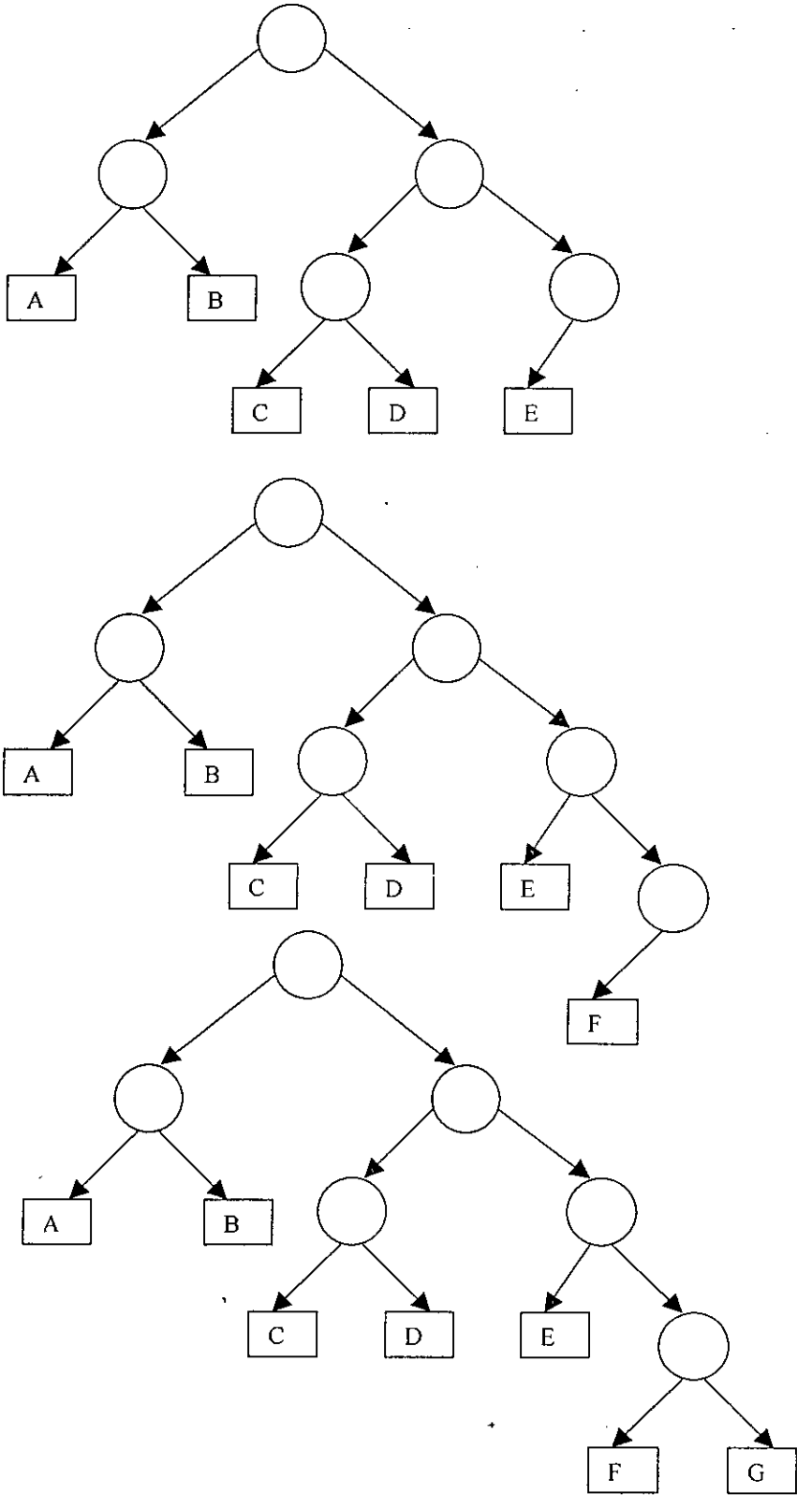


Figure 3.6: Reconstructing Huffman Tree (continued)

Space Complexity Analysis

Total number of distinct symbols=n
Space for all codes=n
Total space for Huffman tree representation = Symbols + codes
= n+n
= 2n

3.3 Proposed Methods of a Huffman Tree Representation

This thesis has proposed a series of methods to represent a Huffman tree efficiently in memory. Each of the methods takes space that is less than $\lceil 3n/2 \rceil$ at worst case. Every proposed method improves the existing methods.

3.3.1 Level Order Technique

In this method each internal and external node is represented by '1' and '0' respectively. After traversing the Huffman tree by level order, a bit stream of 1's and 0's, and a character sequence (external node only) of n distinct symbols are obtained.

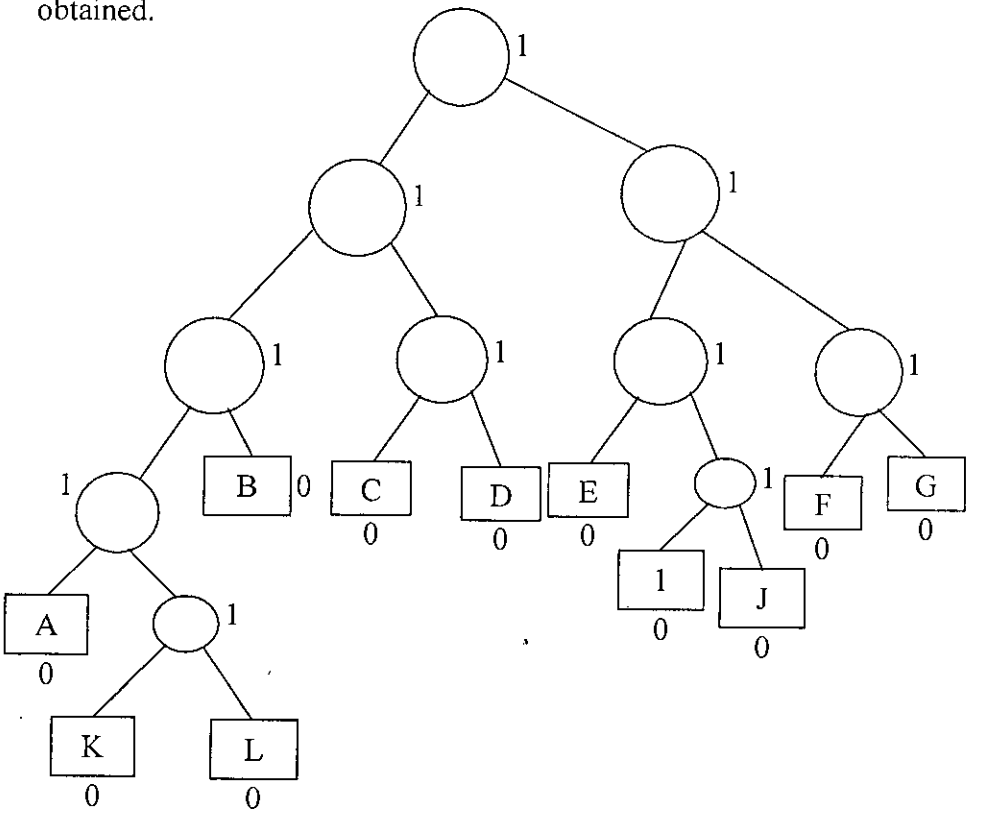


Figure 3.7: Huffman Tree for Level Order Technique

Level Order Bit Stream: 11111111000010 0010000

Level Order Character Sequence: BCDE FGAIJKL

Algorithm 3.1 shows how to represent a Huffman tree using level order technique.

Step 1: Represent each internal node by '1'

Step 2: Represent each external node by '0'

Step 3: Traverse Huffman tree by level order to get a bit stream of 1's (internal node) and 0's (external node), and a character sequence of n distinct symbols (external node only).

Algorithm 3.1: A Huffman Tree Representation in Level Order Technique

Theorem 3.2 *Let S and D be the array of symbols and set of characters (obtained from bit stream of 1's & 0's) respectively, which have been obtained by traversing the Huffman Tree T level order, sent to the receiver. Then the receiver can uniquely reconstruct the Huffman tree T.*

Proof.

Here S = Set of n distinct symbols
D = Set of characters that is obtained from bit stream
B = Bit stream that is generated from D

Scan the bit stream B from left to right. For each bit of B, either an internal or an external node will be created. If bit is '1', it will be an internal node; otherwise it will be an external node. If it is external node, corresponding symbol will be extracted from S. Both pointer of B and S is forwarded. Later internal node will be expanded for current bit of B as prior. Since the receiver is scanning both arrays from left to right always, there is no possibility to get more than one Huffman tree for the same bit stream and array of symbols. It concludes that T is a unique Huffman tree. ■

Algorithm 3.2 describes Huffman tree reconstruction procedure.

```
i =0
r=CREATE_NODE()
Q⇐r
While Q≠empty do
        p=DELETE(Q)
        If (t[++i] ='1') then
                newnode= CREATE_NODE()
                p↑.left=newnode
                Q⇐newnode
        Else
                newnode= CREATE_NODE()
                newnode↑.info=character
                p↑.left=newnode
                newnode↑.left=NIL
                newnode↑.right=NIL
        End if
        If (t[++i] ='1') then
                newnode= CREATE_NODE()
                p↑.right=newnode
                Q⇐newnode
        Else
                newnode= CREATE_NODE()
                newnode↑.info=character
                p↑.right=newnode
                newnode↑.left=NIL
                newnode↑.right=NIL
        End if
End While
```

Algorithm 3.2: Huffman Tree Reconstruction

Space Complexity Analysis

Number of distinct characters =n
So, number of internal nodes in a Huffman tree=n-1
And number of external nodes in a Huffman tree=n
n external nodes and n-1 internal nodes representation are required
∴ Total required bits for internal and external nodes $= n + (n-1)$
$= 2n - 1$

Total space required for a Huffman tree $= n + (2n-1)/8$
$= n + n/4 - 1/8$
$= 5n/4 - 1/8$

So, required memory in this method is less than existing method which takes at most $\lceil 3n/2 \rceil$ memory spaces for a Huffman tree of n distinct symbols.

3.3.2 **Modified Level Order Technique**

In this method each internal and external node is represented by '1' and '0' respectively. After traversing the Huffman tree by level order, a bit stream of 1's and 0's, and a character sequence (external nodes only) of n distinct symbols are got. The right most 0's (zeros) of the bit stream is discarded to obtain the final bit stream.

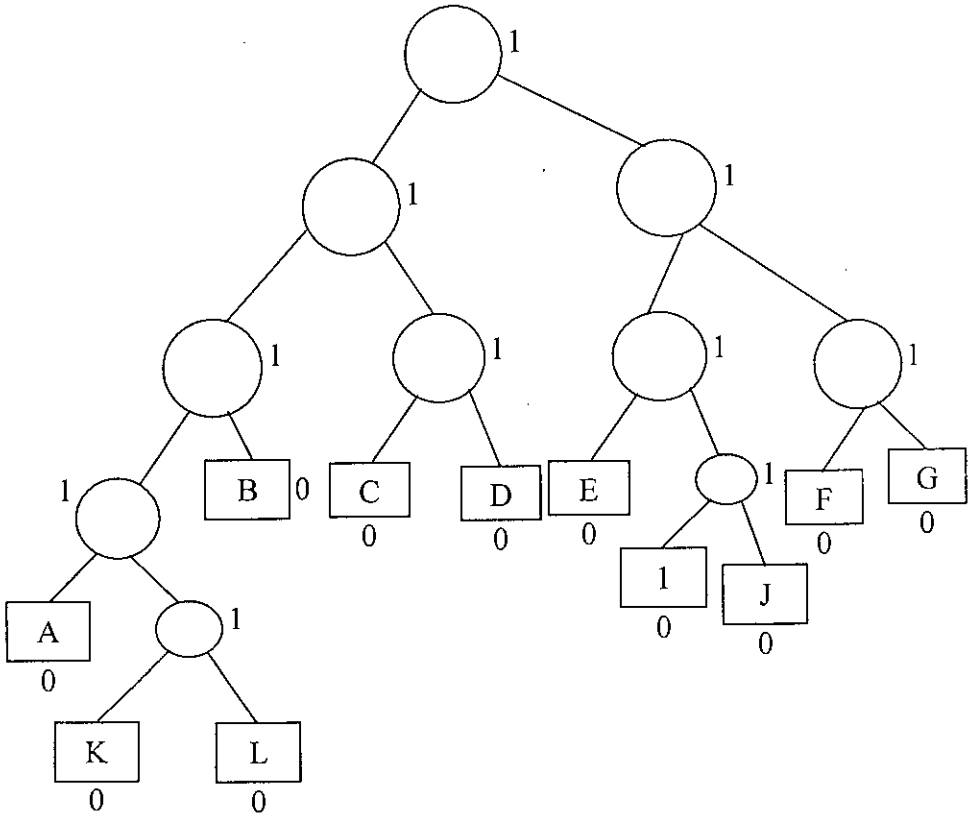


Figure 3.8: Huffman Tree for Modified Level Order Technique

Level Order Bit Stream: 11111111000010 0010000

Level Order Character Sequence: BCDE FGAIJKL

After Discarding Right Most Zeros Sequence: 11111111000010 001

Algorithm 3.3 illustrates how to represent a Huffman Tree in Modified Level order technique.

Step 1: Represent each internal node by '1'

Step 2: Represent each external node by '0'

Step 3: Traverse Huffman tree by level order to get a bit stream of 1's (internal node) and 0's (external node), and a character sequence of n distinct symbols (external node only).

Step 4: Discard the right most zeros from the bit stream.

Algorithm 3.3: A Huffman Tree Representation in Modified Level Order Technique

Theorem 3.3 *Let S and D be the array of symbols and set of characters (obtained from bit stream of 1's & 0's after discarding right most zeros) respectively, which have been obtained by traversing the Huffman Tree T level order, sent to the receiver. Then the receiver can uniquely reconstruct the Huffman tree T.*

Proof.

Here S = Set of n distinct symbols
D = Set of characters that is obtained from bit stream
B = Bit stream that is generated from D

Scan the bit stream B from left to right. For each bit of B, either an internal or an external node will be created. If bit is '1', it will be an internal node; otherwise it will be an external node. If it is external node, corresponding symbol will be extracted from S. Both pointer of B and S is forwarded. Later internal node will be expanded for current bit of B as prior. When bit stream has been finished, add two external nodes for each of the internal node that is not expanded. Extract symbol for each external node from S array. Since the receiver is scanning both arrays from left to right always so, there is no possibility to get more than one Huffman tree for the same bit stream and array of symbols. It concludes that T is a unique Huffman tree.■

Algorithm 3.4 shows how to reconstruct a Huffman tree

```
i =0
r=CREATE_NODE()
Q⇐r
While Q≠empty do
        p=DELETE(Q)
        If tree bit stream is finished
                Add an external node of corresponding character to
                the left of p.
        Else If (t[++i] ='1') then
                newnode= CREATE_NODE()
                p↑.left=newnode
                Q⇐newnode
        Else
                newnode= CREATE_NODE()
                newnode↑.info=character
                p↑.left=newnode
                newnode↑.left=NIL
                newnode↑.right=NIL
        End if
        If tree bit stream is finished
                Add an external node of corresponding character to
                the right of p.
        Else If (t[++i] ='1') then
                newnode= CREATE_NODE()
                p↑.right=newnode
                Q⇐newnode
        Else
                newnode= CREATE_NODE()
                newnode↑.info=character
                p↑.right=newnode
                newnode↑.left=NIL
                newnode↑.right=NIL
        End if
End While
```

Algorithm 3.4: Huffman Tree Reconstruction

Space Complexity Analysis

In this method best and worst cases are analyzed for the space complexity.

Best Case Analysis

Best case occurs when the Huffman tree is fully balanced binary.

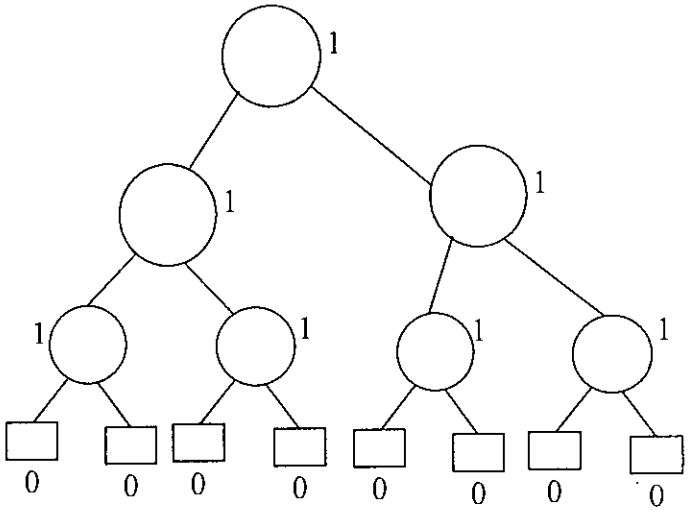


Figure 3.9: A Balanced Huffman Tree

Level Order Bit Stream: 111 11 110000 0000

After Discarding Right Most Zeros Sequence: 111 1111

No. of distinct characters=n
Number of internal nodes in a Huffman tree = n-1
Number of external nodes in a Huffman tree = n

There is no need to represent the external nodes

Total space for the Huffman tree =Symbols +internal nodes
= n + (n-1)/8
= n + n/8 – 1/8
= 9n/8 – 1/8

Worst Case Analysis

Worst case occurs when a Huffman tree is like the Figure 3.10

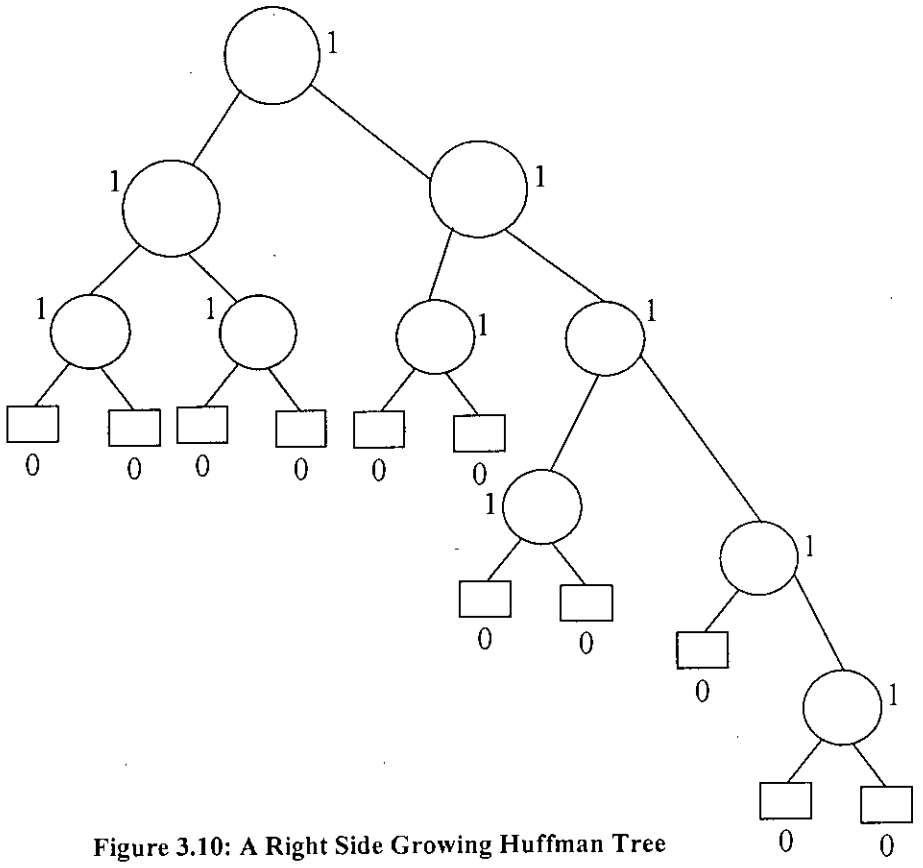


Figure 3.10: A Right Side Growing Huffman Tree

Level Order Bit Stream: 1111111000000000100

After Discarding Right Most Zeros Sequence: 111 11110000000001

Number of internal nodes in a Huffman tree = n-1
Number of external nodes in a Huffman tree = n
Only (n-1) internal nodes and (n-2) external nodes representation are required
Total spaces for the Huffman tree =Symbols + node representation
= n + (2n-1-2)/8
= n + (2n-3)/8
= n + n/4-3/8
= 5n/4-3/8

So, required memory in this method is less than existing method which takes at most $\lceil 3n/2 \rceil$ memory spaces for a Huffman tree of n distinct symbols

3.3.3 Preorder Technique

In this method each internal and external node is represented by '1' and '0' respectively. After traversing the Huffman tree by preorder, a bit stream of 1's and 0's, and a character sequence (external node only) of n distinct symbols are obtained.

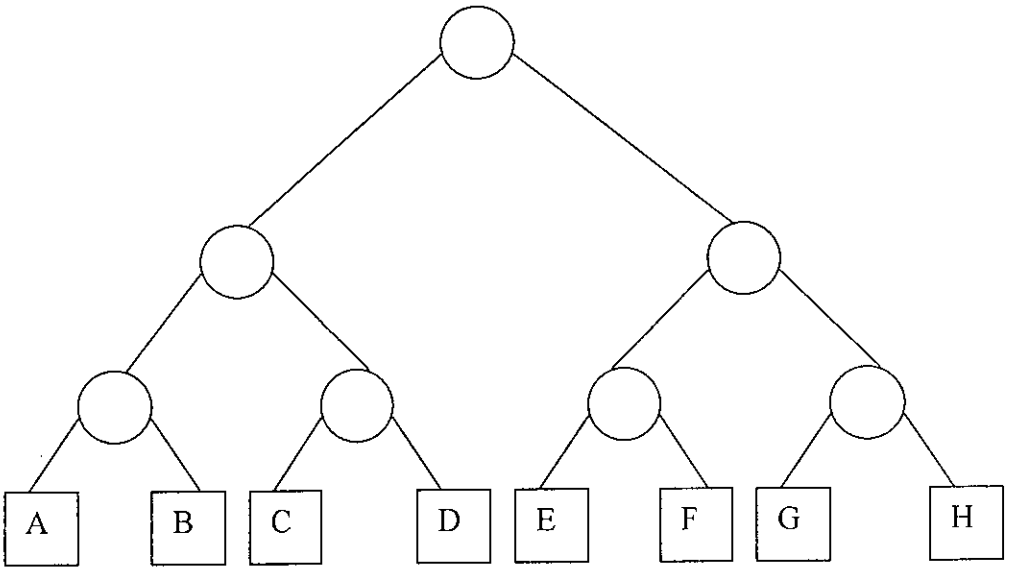


Figure 3.11: Huffman Tree for Preorder Technique

Preorder Bit Stream: 11100 100 1100100

Preorder Character Sequence: ABCDEFGH

Algorithm 3.5 shows how to represent a Huffman tree in Preorder technique.

Step 1: Represent each internal node by '1'

Step 2: Represent each external node by '0'

Step 3: Traverse Huffman tree by preorder to get a bit stream of 1's (internal node) and 0's (external node), and a character sequence of n distinct symbols (external node only).

Algorithm 3.5: Huffman Tree Representation in Preorder Technique

Theorem 3.4 *Let S and D be the array of symbols and set of characters (obtained from bit stream of 1's & 0's) respectively, which have been obtained by traversing the Huffman Tree T preorder, sent to the receiver. Then the receiver can uniquely reconstruct the Huffman tree T.*

Proof.

Here S = Set of n distinct symbols
D = Set of characters that is obtained from bit stream
B = Bit stream that is generated from D

Scan the bit stream B from left to right. For each bit of B, either an internal or an external node will be created. If bit is '1', it will be an internal node; otherwise it will be an external node. If it is an internal node, this node is expanded recursively until a 0's in B. Add an external node for this '0' and extract a character from S. Pointer of B is forwarded for each 1's of B. Pointer of S is forwarded only for a 0's in B. Since the receiver is scanning both arrays from left to right always so, there is no possibility to get more than one Huffman tree for the same bit stream and array of symbols. It concludes that T is a unique Huffman tree. ■

Algorithm 3.6 shows how to reconstruct a Huffman Tree.

```
Procedure Preorder(r, t, Cset)
        Begin
                i=0
                if (t[++i]='1')then
                        Add an internal node "temp" to the left of r
                        Preorder(temp, t, Cset)
                Else
                        Add an external node "temp" to the left of r
                        temp↑.info=character
                End if

                if (t[++i]='1')then
                        Add an internal node "temp" to the right of r
                        Preorder(temp, t, Cset)
                Else
                        Add an external node"temp" to the right of r
                        temp↑.info=character
                End if
        End
```

Algorithm 3.6: Huffman Tree Reconstruction

Space Complexity Analysis

Number of distinct characters =n
So, number of internal nodes in a Huffman tree=n-1
And number of external nodes in a Huffman tree=n
n external nodes and n-1 internal nodes representation are required
∴ Total required bits for internal and external nodes $= n + (n-1)$
$= 2n - 1$

Total space required for a Huffman tree $= n + (2n-1)/8$
$= n + n/4 - 1/8$
$= 5n/4 - 1/8$

So, required memory in this method is less than existing method which takes at most $\lceil 3n/2 \rceil$ memory spaces for a Huffman tree of n distinct symbols.

3.3.4 **Modified Preorder Technique**

In this method each internal and external node is represented by '1' and '0' respectively. After traversing the Huffman tree by preorder, a bit stream of 1's and 0's, and a character sequence (external nodes only) of n distinct symbols are got. The right most 0's (zeros) of the bit stream is discarded to get the final bit stream.

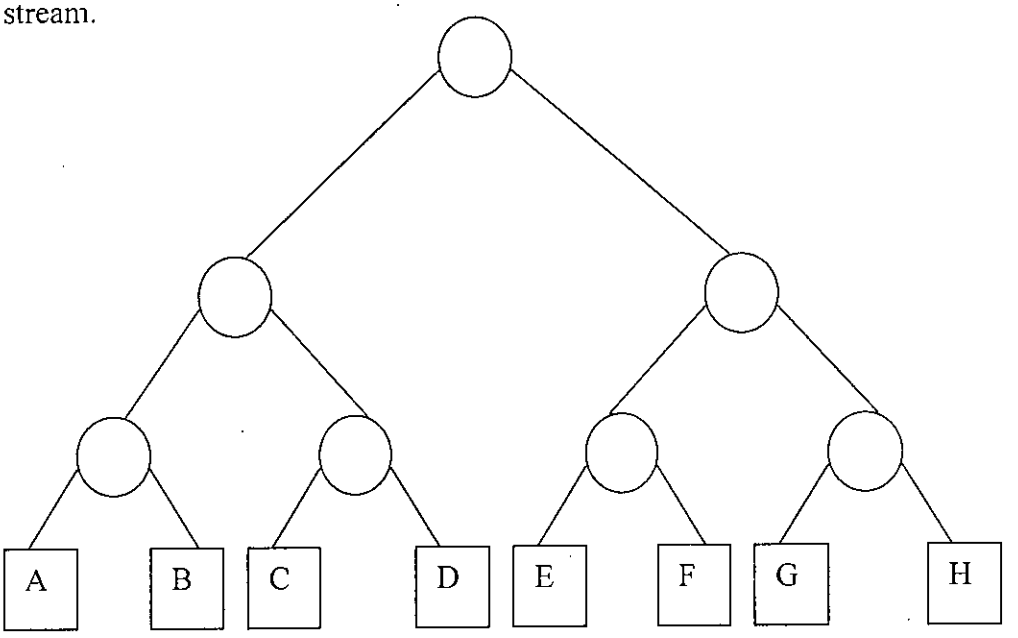


Figure 3.12: Huffman Tree for Modified Preorder Technique

Preorder Bit stream: 11100 100 1100100

Preorder Character Sequence: ABCDEFGH

After Discarding Right Most Zeros Sequence: 11100 100 11001

Algorithm 3.7 describes how to represent a Huffman Tree in Modified Preorder technique.

Step 1: Represent each internal node by '1'
Step 2: Represent each external node by '0'
Step 3: Traverse Huffman tree by Preorder to get a bit stream of 1's (internal node) and 0's (external node), and a character sequence of n distinct symbols (external node only).
Step 4: Discard the right most zeros from the bit stream.

Algorithm 3.7: Huffman Tree Representation in Modified Preorder Technique

Theorem 3.5 *Let S and D be the array of symbols and set of characters (obtained from bit stream of 1's & 0's after discarding right most zeros) respectively, which have been obtained by traversing the Huffman Tree T preorder, sent to the receiver. Then the receiver can uniquely reconstruct the Huffman tree T.*

Proof.

Here S = Set of n distinct symbols
D = Set of characters that is obtained from bit stream
B = Bit stream that is generated from D

Scan the bit stream B from left to right. For each bit of B, either an internal or an external node will be created. If bit is '1', it will be an internal node; otherwise it will be an external node. If it is an internal node, this node is expanded recursively until a 0's in B. Add an external node for this '0' and extract a character from S. Pointer of B is forwarded for each 1's of B. Pointer of S is forwarded only for a 0's in B. When bit stream has been finished, add two external nodes for each of the internal node that is not expanded. Extract symbol for each external node from S array. Since the receiver is scanning both arrays from left to right always so, there is no possibility to get more than one Huffman tree for the same bit stream and array of symbols. It concludes that T is a unique Huffman tree. ■

Algorithm 3.8 illustrates a Huffman Tree reconstruction procedure

```
Preorder(r, t, Cset)
    i = 0
    if tree bit stream is finished
        Add an external node of corresponding character to the left of r
    Else if (t[++i]=1) then
        Add an internal node "temp" to the left of r
        Preorder(temp, t, Cset)
    Else
        Add an external node "temp" to the left of r
        temp↑.info=character
    Endif
    if tree bit stream is finished
        Add an external node of corresponding character to the right of r
    Else if (t[++i]=1) then
        Add an internal node "temp" to the right of r
        Preorder(temp, t, Cset)
    Else
        Add an external node "temp" to the right of r
        temp↑.info=character
    End if
End Procedure
```

Algorithm 3.8: Huffman Tree Reconstruction

Space Complexity Analysis

In this method best and worst cases are analyzed for the space complexity.

Best Case Analysis

This case occurs when the Huffman tree is left eccentric

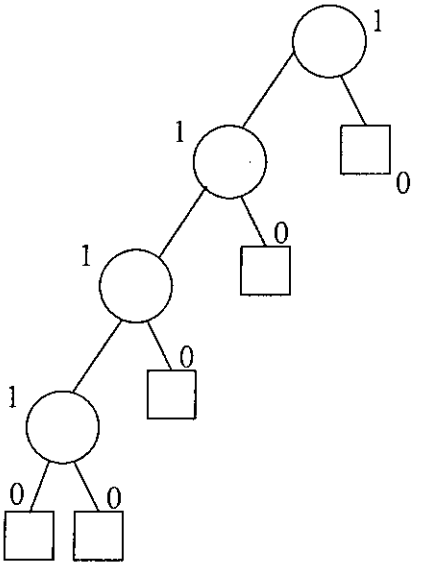


Figure 3.13: A Left Side Growing Huffman Tree

Preorder Bit Stream: 1111 00000

After Discarding Right Most Zeros Sequence: 1111

n = no. of distinct characters
Number of internal nodes in a Huffman tree = n-1
Number of external nodes in a Huffman tree = n
There is no need to represent the external nodes

Total space for the Huffman tree =Symbols +internal nodes
= n + (n-1)/8
= n + n/8 – 1/8
= 9n/8 – 1/8

Worst Case Analysis

Worst case occurs when a Huffman tree is like the Figure 3.14

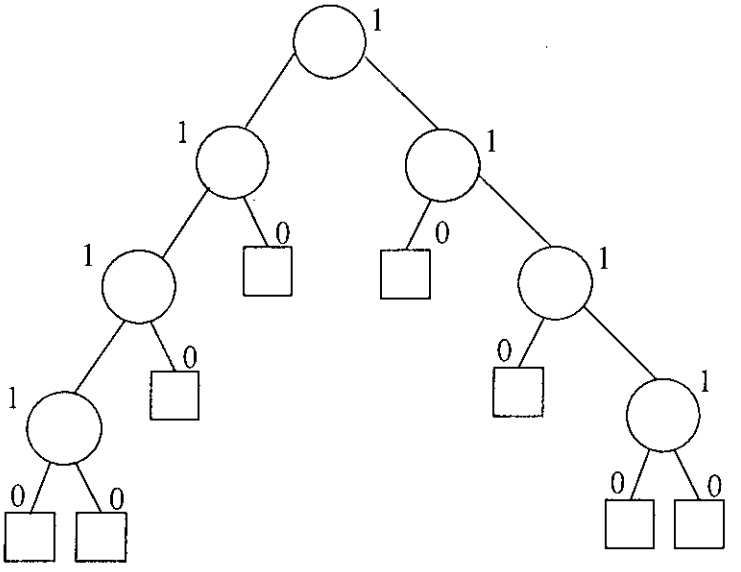


Figure 3.14: A Huffman Tree

Preorder Bit Stream: 111100001010100

After Discarding Right Most Zeros Sequence: 1111 000010101

Number of internal nodes in a Huffman tree = n-1
Number of external nodes in a Huffman tree = n
Only (n-1) internal nodes and (n-2) external nodes representation are required

Total spaces for the Huffman tree =Symbols + node representation
= n + (2n-1-2)/8
= n + (2n-3)/8
= n + n/4-3/8
= 5n/4-3/8

So, required memory in this method is less than existing method which takes at most $\lceil 3n/2 \rceil$ memory spaces for a Huffman tree of n distinct symbols

3.3.5 **Single Side Growing Tree Technique**

In this method a Huffman is made as eccentric as possible to reduce the number of circular leaf nodes (internal node with two external nodes). Then apply the technique of Chowdhury et al. [4]. Since their technique requires at most $\lceil 3n/2 \rceil$ memory spaces by using the circular leaf node concept, it is required to reduce the number of circular leaf nodes.

Algorithm 3.9 describes how to build an Eccentric Huffman Tree

Step 1: Count the frequency of every distinct character.

Step 2: Follow the Huffman tree construction process to reduce into a single root.

Step 3: Find all the code lengths of symbols occurring in each reduction step. Construct a table.

Step 4: From the table of step3 find a code length of every distinct character occurring in given text. Construct a table.

Step 5: Construct a code table using the following procedure

Step 5.1: Assign zero codeword to S_1 i.e. C_1 =0..0 Where S_1 is first symbol of the table.

Step 5.2: When we changes rows, we have to expand the last codeword, after being incremented, by placing extra zeros to right, until the codeword length matches the level.

Step 6: Construct an eccentric Huffman tree from the code table

Algorithm 3.9: Single Side Growing Huffman Tree Construction

Example

The Figure 3.15 shows a Huffman tree

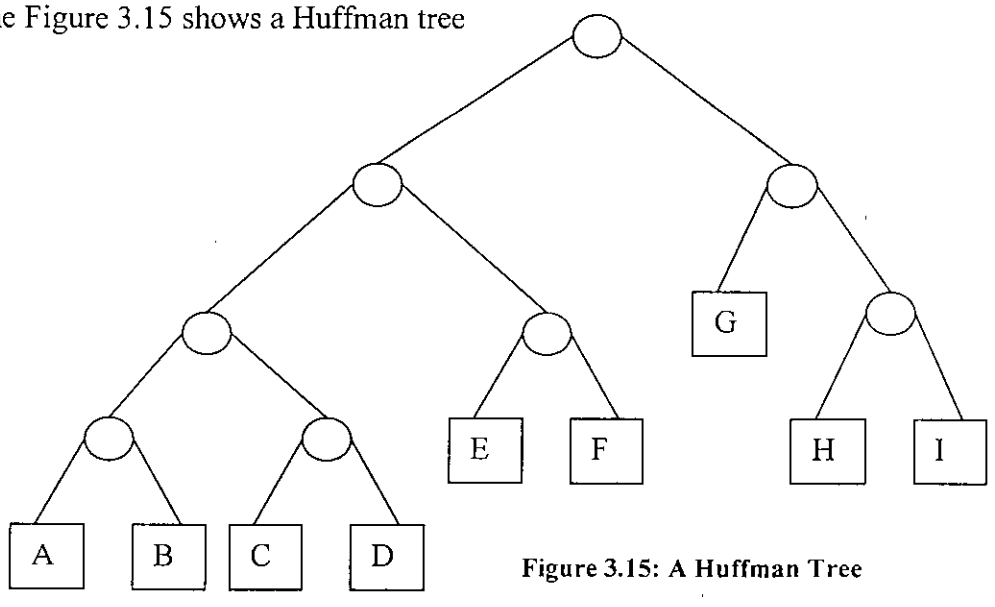


Figure 3.15: A Huffman Tree

Table 3.1: Code Length for Every Symbol

ROW	CL	Symbol
1	2	G
2	3	E, F, H, I
3	4	A, B,C, D

Table 3.2: Code for Every Symbol

CL	Codeword	Symbol
2	00	G
3	010	E
	011	F
	100	H
	101	I
4	1100	A
	1101	B
	1110	C
	1111	D

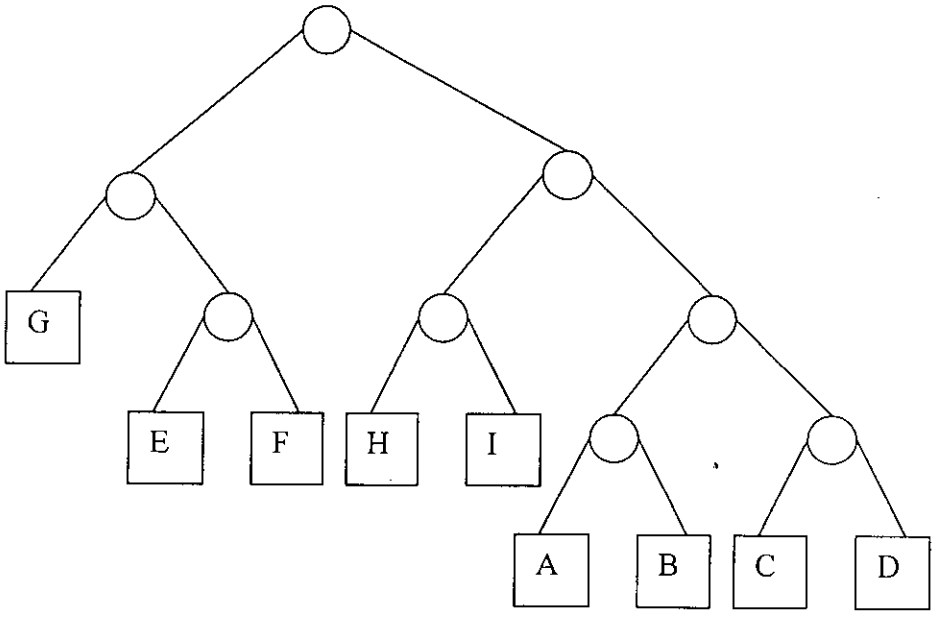


Figure 3.16: Huffman Tree for Code of Table 3.2

Space Complexity Analysis

Best case space requirement to represent a Huffman tree=n+1
Worst case space requirement to represent a Huffman tree=n+$\lceil n/2 \rceil$
So, average case space complexity = $(2n+\lceil n/2 \rceil +1)/2$
$\leq \lfloor 3n/2 \rfloor$

In this case at most memory requirement is $\lfloor 3n/2 \rfloor$

3.3.6. Balanced Binary Tree Technique

In our method we are always looking for fully balanced part of Huffman tree with respect to its external nodes. For the balanced part of height greater than 1(one) requires only two spaces (one for height of the balanced part and another for code of the root of balanced part) to store it. If the whole Huffman tree is fully balanced, only one space (height) is required to store it. Balanced part of height 1(one) needs one space (code of root of the balanced part).

Algorithm 3.10 describes how to represent a Huffman Tree in Balanced Binary tree technique.

Step 1: If the tree is fully balanced then
    Store height of the Huffman tree
    GOTO step 5
Endif

Step 2: Identify a balanced part of the tree with respect to its external nodes

Step 3: If the balanced part is height 1(one) then
    Store a code for the root of the balanced part
Else
    Allocate two spaces one for code of root of this balanced part and another for height of this part
End if

Step 4: If more balanced part exist then
    GOTO step 2
End if

Step 5: Stop algorithm

Algorithm 3.10: Huffman Tree Representation in Balanced Binary Tree Technique

Space Complexity Analysis

Space for a Huffman tree presentation=S(T)

$$S(T) = \begin{cases} 1 & \text{if } T = \text{balance at first time} \\ & \text{otherwise} \\ 1 & \text{if } T = \text{balanced and } h = 1 \\ 2 & \text{if } T = \text{Balanced and } h > 1 \\ 0 & \text{if } T = \text{external node} \\ S(T_r) + S(T_l) & \text{otherwise} \end{cases}$$

Total space for tree = n + S(T)

Examples

n is the number of distinct symbols of the message.

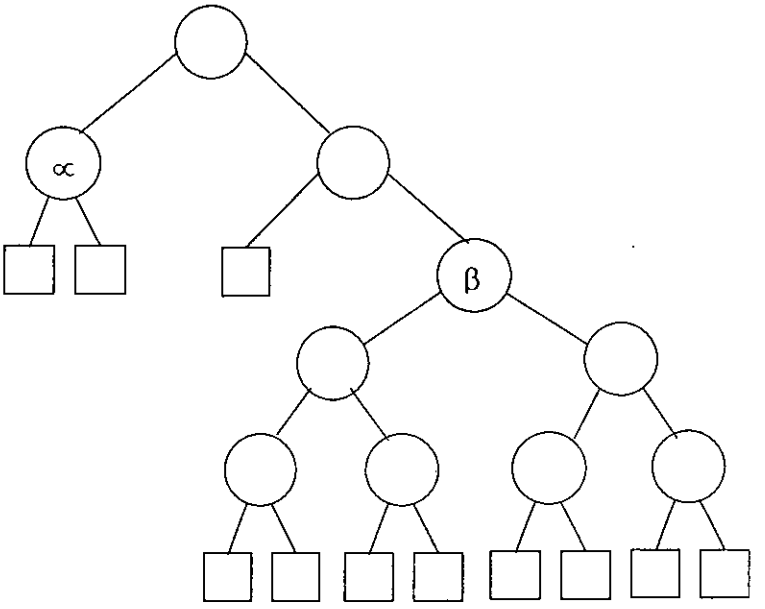


Figure 3.17: A Huffman Tree for 11 Symbols

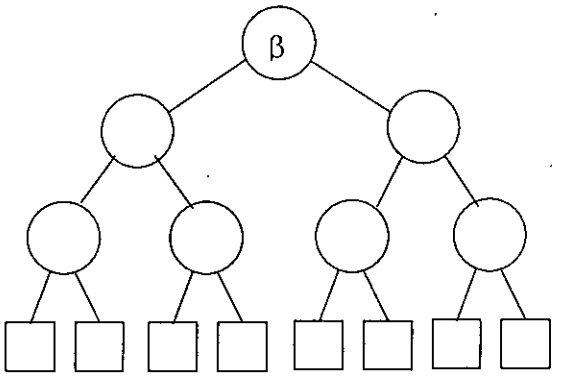
$$\begin{aligned} S(T) &= S(T_l) + S(T_r) \\ &= 1+S(T_l)+S(T_r) \\ &= 1+0+2 \\ &= 3 \end{aligned}$$

$$\begin{aligned} \text{Total space for a Huffman tree} &= n+ S(T) \\ &= n+3 \\ &= 11+3 \\ &= 14 \end{aligned}$$

Best case Space Complexity Analysis

Best case occurs when a Huffman tree is like the Figure 3.18-3.19

a)

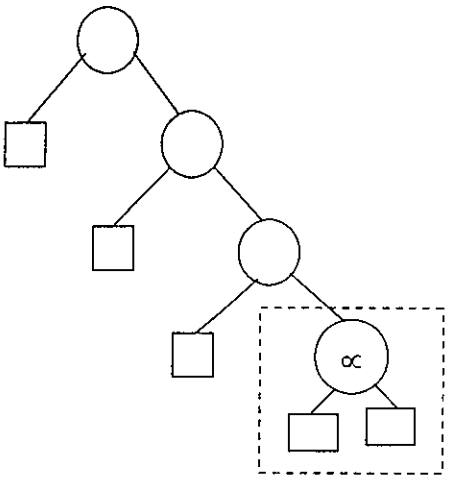


Space = height
= 1

Figure 3.18: A Balanced Huffman Tree

Only one space for height is required.
To represent the tree total memory space = n+1

b)



Space = code
= 1

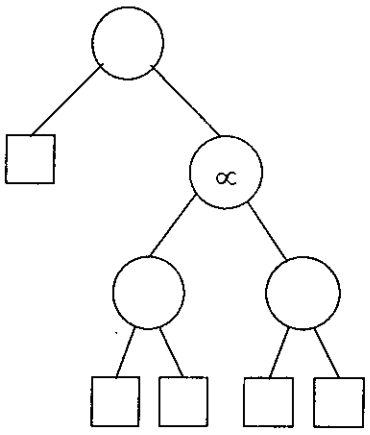
Figure 3.19: A Right Side Growing Huffman Tree

Only one space for code of ∝ is required.
Total memory space to represent a Huffman tree = n+1

Worst case Space Complexity Analysis

Worst case occurs when a Huffman tree is like the Figure 3.20

a)



Space for ∝ = height + code
= 1+1
= 2

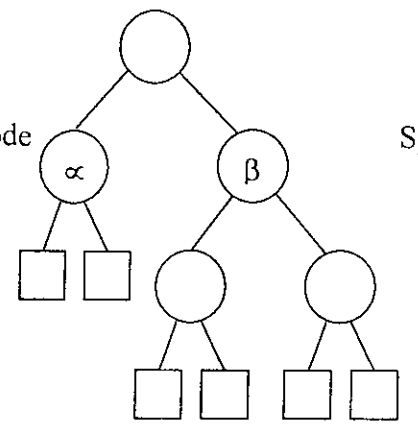
Figure 3.20: A Huffman Tree for 5 Symbols

Space = 2* height2 balanced subtree + 0
= 2*1+0 = 2

Total tree space = 5+2 = 7 = ⌊3n/2⌋

b)

Space for ∝ = code
= 1



Space for β = height + code
= 1+1
= 2

Figure 3.21: A Huffman Tree for 6 Symbols

Space = 2* height2 balanced subtree +1
= 2*1+1
= 3

Total tree space = n+3
= n+3
= 6+3
= 9 = ⌊3n/2⌋

c)

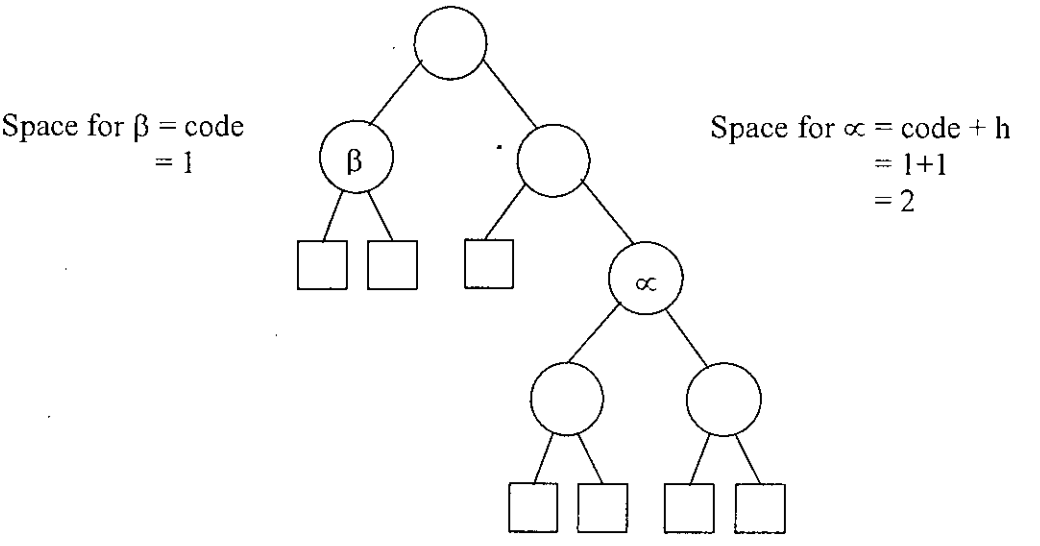


Figure 3.22: A Huffman Tree for 7 Symbols

Space = 2* height2 balanced subtree + 1
= 2*1+1
= 3

Total Tree space = n+3
= 7+3
= 10
= ⌊3n/2⌋

d)

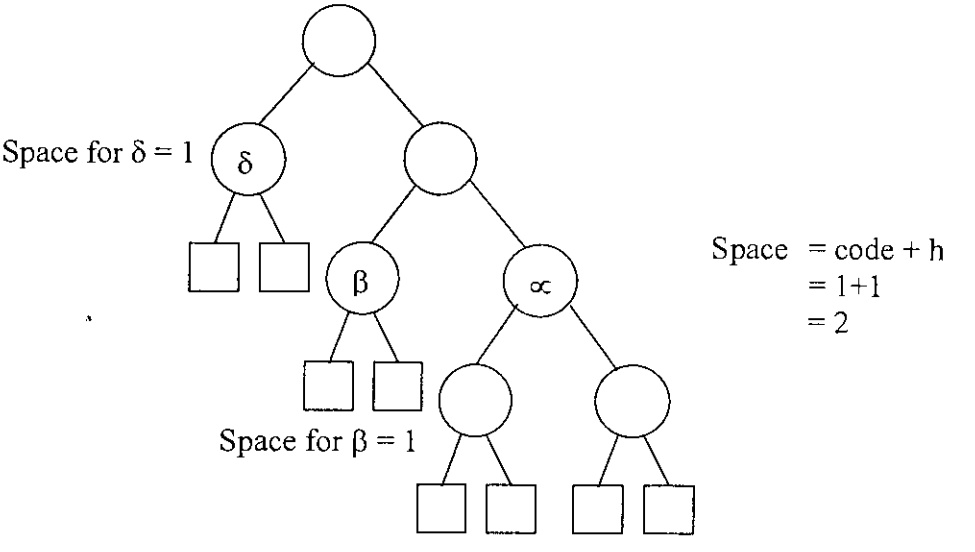


Figure 3.23: A Huffman Tree for 8 Symbols

Space for $\propto$ = 2*height2 balanced subtree +2
= 2*1+2
= 4
Total space for the tree = n +4
= 8+4
= 12
$= \lfloor 3n/2 \rfloor$

Worst case space complexity of this method $= \lfloor 3n/2 \rfloor$

So, required memory in this method is less than existing method which takes at most $\lceil 3n/2 \rceil$ memory spaces for a Huffman tree of n distinct symbols

Algorithm 3.11 illustrates reconstruction of a Huffman tree

```
Procedure RECONSTRUCTION(C)
    Tree=empty
    Repeat
        Read a codeword from C
        Identify the height and code of root of the subtree
        Draw a subtree T using this information
        Tree= Tree U T
    Until all codewords for tree are not read from C
    Return Tree
End Procedure
```

Algorithm 3.11: Huffman Tree Reconstruction

3.4. **Conclusion**

A memory efficient representation of a Huffman tree reduces the overhead of every pass of Repeated Huffman Coding as well as compression ratio is increased.

CHAPTER 4

HUFFMAN TREE CLUSTERING

4.1 Introduction to Tree Clustering

A Huffman tree clustering means the partitioning of the Huffman tree effectively to reduce the sparsity of the tree. Main objective of the clustering is to represent the Huffman tree efficiently to reduce the wastage of memory and to make search process of most frequent symbol faster.

4.2 Problems of Huffman Codes

i) Due to variable-length coding, the Huffman tree gets progressively sparse as it grows from the root.

- Wastage of memory
- A lengthy search procedure for locating a symbol.

ii) If k-bit is the longest Huffman code assigned to a set of symbols, the memory size for the symbols may easily reach 2^k words in size

4.3 Desirable Features of Huffman Codes

It is desirable to reduce the memory size from the typical value of 2^k, to a size proportional to the number of the actual symbols. The following two things are desired from the Huffman code.

- Reduce memory size
- Quicker access

4.4 Solutions of Huffman Code Related Problems

Hashemian [11] presented an algorithm to speed up the search process for a symbol in a Huffman tree and reduce the memory size. He proposed a tree clustering algorithm to avoid the sparsity of a Huffman tree.

- The search time for the more frequent symbols (shorter codes) is substantially reduced compare to less frequent symbols, resulting in an overall faster response.

- For long codewords the search for the symbol is also speed up. This is achieved through a specific partitioning technique that groups the code bits in a codeword, and the search for a symbol is conducted by jumping over the groups of bits rather than going through the bit individually.

4.5 Problems of the Tree Clustering Algorithm

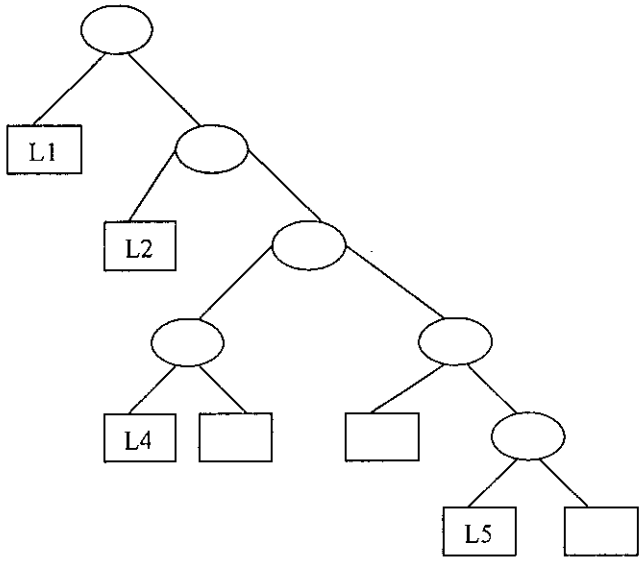


Figure 4.1: Huffman Tree Showing Levels

The memory efficiency B_k for a k-level binary Huffman tree is given below

$$B_k = \frac{N}{2^k} \times 100$$

Where N is number of effective leaf nodes at level k.

Table 4.1: Memory Efficiency

Level Number (k)	Effective Leaf Nodes	Memory efficiency (B_k)
1	2	100%
2	3	75%
3	4	50%
4	6	37%
5	7	22%

Higher memory efficiency for the top levels (with smaller C_L) is a clear indication that partitioning the tree into smaller and less sparse clusters will reduce the memory size. In addition, clustering also helps to reduce the search time for a symbol.

Search time $\propto C_L/L$

Where C_L is code length of a symbol and L is the maximum level selected for each cluster

Increasing L decreases search time and hence speeds up decoding but increases memory space requirement. Decision regarding L is needed. Finding an optimal solution of this clustering is still open.

4.6 Example of Huffman Tree Clustering

Table 4.2: Frequency Count

Character	Frequency
A	7
B	5
C	4
D	4
E	3
F	2
G	1

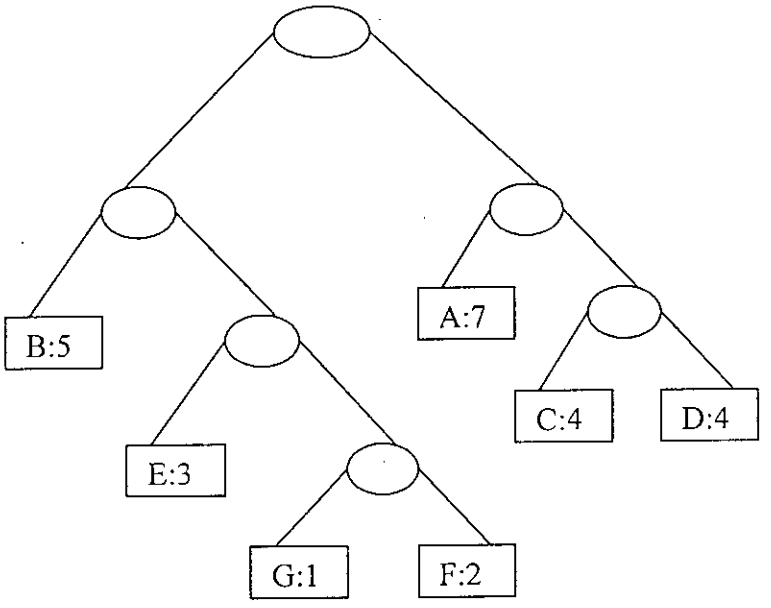


Figure 4.2: A Huffman Tree for 7 Symbols

Table 4.3: Reduction Process

A	7	A	7	A	7	a_3	8	a_4	11	a_5	15	a_6	26
B	5	B	5	a_2	6	A	7	a_3	8	a_4	11		
C	4	C	4	B	5	a_2	6	A	1				
D	4	D	4	C	4	B	3						
E	3	E	3	D	4								
F	2	a_1	3										
G	1												

Table 4.4: Code Length (CL)

S_i	S_{i-1}	CL
G	F	4
a_1	E	3
C	D	3
B	a_2	2
A	a_3	2
a_4	a_5	1

Table 4.5: Same Code of Length for Different Symbols

CL	Symbol
1	-
2	A, B
3	C,D,E
4	G,F

Table 4.6: Codeword for Every Symbol

CL	Code word	Symbol
1	0	-
2	00	A
	01	B
3	100	C
	101	D
	110	E
4	1110	G
	1111	F

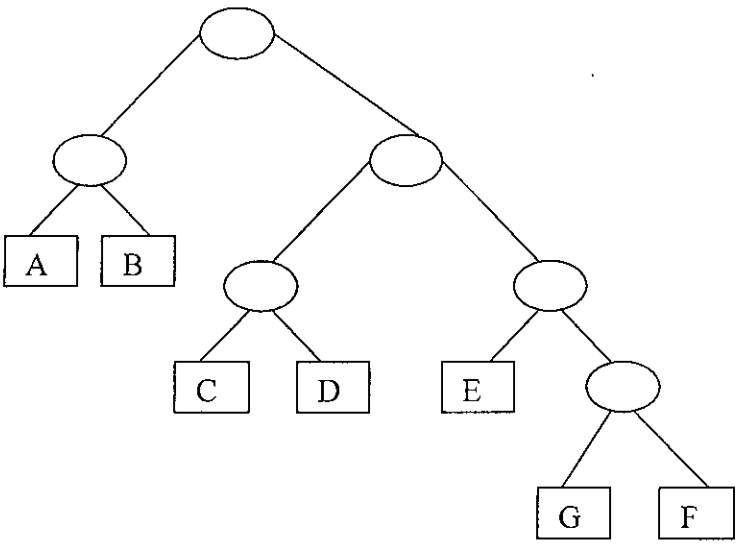


Figure 4.3: Single side Growing Huffman Tree

Table 4.7: Memory Efficiency

Level Number (k)	Effective leaf nodes	Memory efficiency (B_k)
1	2	100%
2	4	100%
3	6	75%
4	7	44%

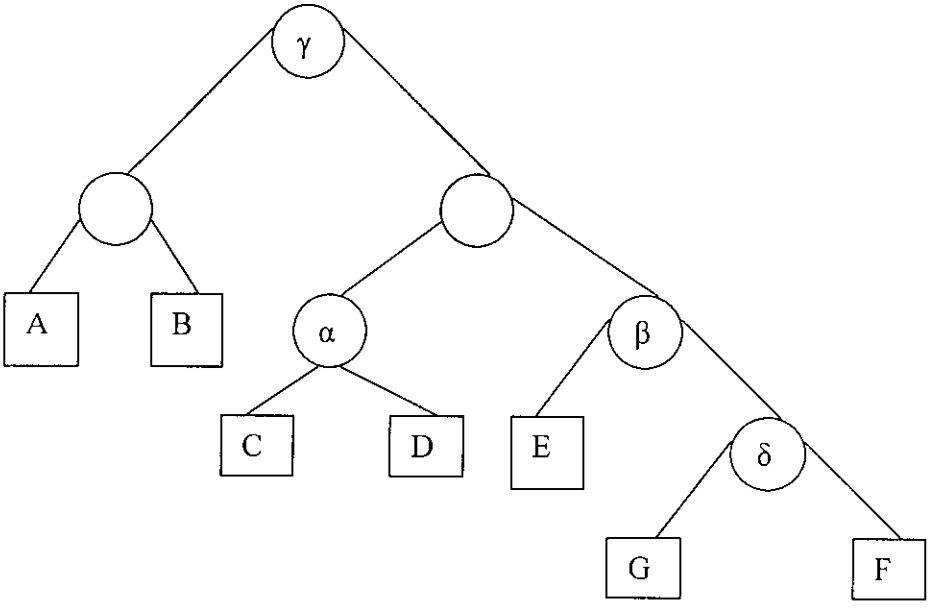


Figure 4.4: A Huffman Tree for Clustering

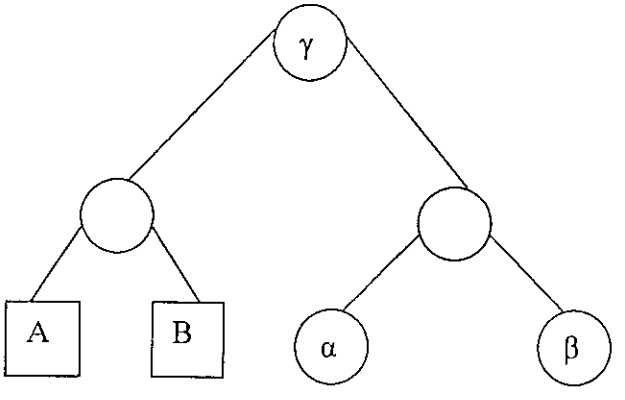


Figure 4.5: Top Cluster

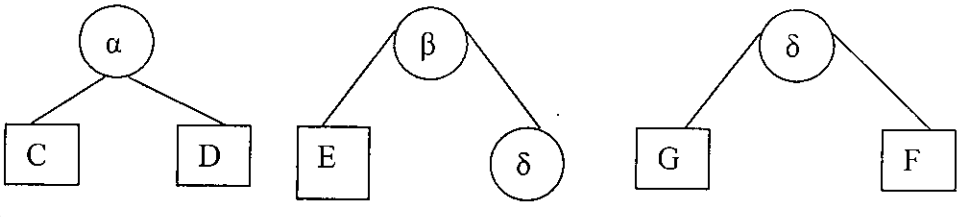


Figure 4.6: Cluster for α, β and δ

Table 4.8: Memory Space

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	A	B			C	D	E		G	F						

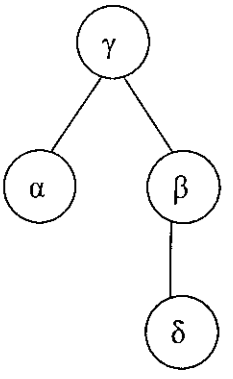


Figure 4.7: Super-tree

Table 4.9: Super Table

α	β	δ	Description
04	06	08	Base Address of each Cluster
00	00	00	Cluster length-1
1	2	3	Index Number

Table 4.10: Look up Table for γ Cluster

For γ cluster				
4	5	1	2	4=$(1\underline{00})_2$, Address of A=$(00)_2$, cluster length=2 5==$(1\underline{01})_2$, Address of B=$(01)_2$, cluster length=2 1=Index of α in super table 2=Index of β in super table
0	0	1	1	'0' External node '1' internal node
0	1	2	3	Index Number

Table 4.11: Look up Table for α Cluster

For α cluster		
2	3	2=$(01\underline{0})_2$, Address of C=$(0)_2$, cluster length=1 3=$(01\underline{1})_2$, Address of D=$(1)_2$, cluster length=1
0	0	'0' External node
0	1	Index Number

Table 4.12: Look up Table for β Cluster

For β cluster		
2	3	2=(01$\underline{0}$)$_2$, Address of E=(0)$_2$, cluster length=1 3=index of δ in super table
0	1	'0' External node '1' internal node
0	1	Index Number

Table 4.13: Look up Table for δ Cluster

For δ cluster		
2	3	2=(01$\underline{0}$)$_2$, Address of G=(0)$_2$, cluster length=1 3=(01$\underline{1}$)$_2$, Address of F=(1)$_2$, cluster length=1
0	0	'0' External node
0	1	Index Number

Table 4.14: Memory Mapping

Entry	Cluster length	Mem [base+offset]	Character
4=(1$\underline{00}$)$_2$	2	Mem[00+00]	'A'
5=(1$\underline{01}$)$_2$	2	Mem[00+01]	'B'
2=(01$\underline{0}$)$_2$	1	Mem[04+0]	'C'
3=(01$\underline{1}$)$_2$	1	Mem[04+1]	'D'
2=(01$\underline{0}$)$_2$	1	Mem[06+0]	'E'
2=(01$\underline{0}$)$_2$	1	Mem[08+0]	'G'
3=(01$\underline{1}$)$_2$	1	Mem[08+1]	'F'

Table 4.15: Decoding Process

▪ Received Code word=1000011101111 ▪ Top cluster length, L=2						
Selected Bit stream	Cluster Search	Table Entry	If Found	Next cluster search	From super table Cluster length	base
"10"→2	γ	1/1	×	α	00+1=1	04
"0" →0	α	0/2	00000010 CL=1 Mem[04+0]='C'	×	×	×
"00" →0	γ	0/4	00000100 CL=2 Mem[00+00]='A'	×	×	×
"11" →3	γ	1/2	×	β	00+1=1	06
"1" →1	β	1/3	×	δ	00+1=1	08
"0" →0	δ	0/2	00000010 CL=1 Mem[08+0]='G'	×	×	×
"11" →3	γ	1/2	×	β	00+1	06
"1" →1	β	1/3	×	δ	00+1	08
"1"→1	δ	0/3	00000011 CL=1 Mem[08+1]='F'	×	×	×
▪ **Decoded word="CAGF"**						

4.7 Algorithms Related to Tree Clustering

We have discussed how to construct a single side growing Huffman tree from a table of code length and then decoding process.

Algorithm 4.1 illustrates how to construct a single side growing Huffman tree.

Step1: Start from the first row of the table and assign an all zero codeword C_1=00...0 to the symbol.

Step 2: Increment this codeword and assign the new value to the next symbol of the same row of table code length.

Step 3: When we change rows, we have to expand the last codeword, after being incremented, by placing extra zeros to the right, until the codeword length matches the level (C_L).

Algorithm 4.1: Single-side Growing Huffman Tree Construction

Algorithm 4.2 describes decoding process using clustered Huffman tree.

Step 0: base=0, i=1

Step 1: If Encoded bit stream is finished
GOTO step4
Else
Take L bit code (L=maximum length of top cluster) C_j

Step2: Use C_j as the address to the look-up table of corresponding cluster.

Step3: If look-up table entry is zero/VAL
Corresponding character is found in the following memory location:
Dec=8 bit stream of VAL
Offset=Value after discarding MSB '1' from Dec
DecodedTxt[i]=Content[Mem[base+Offset]]
i=i+1
GOTO step1
Else If look-up table entry is **1/VAL**
Search **SuperTable[VAL]**
RetrievedEntry=**c/base**
C_j=Next (c+1) bits from encoded bit stream
GOTO step2
End if
End if

Step 4: Stop Algorithm

Algorithm 4.2: Decoding Process

4.8 Decision Regarding Top Cluster

Average code length, AVG = $\sum_{i=1}^{n} p_i l_i$

Where p_i is the probability of i^{th} symbol
And l_i is the code-length of i^{th} symbol

Maximum length of top cluster, L=⌊AVG⌋

This selection criterion of L will reduce wastage of memory as well as search time.

4.9 Conclusion

Clustering of a Huffman tree will speed up search process as well as reducing memory requirement. Optimal selection of top cluster length is required for this purpose. Tree clustering reduces sparsity of Huffman tree.

CHAPTER 5

BLOCK HUFFMAN CODING

5.1 Introduction

Traditional Huffman coding has some practical problems. One of the most important problems is that we have to read the whole stream prior to coding. This is a major problem when

- File size is too large
- Source stream is continuous

When file size is large, it will take much longer time to build the Huffman header for compression. This happens because it is required to read the whole file twice from the source stream that is in most cases from hard disk. In the first read pass it is required to build the Huffman tree to get a code for each individual character. In the second read pass we do the real work of compression. If file size, which is small, can be stored in main memory, reading in second pass can be done from main memory instead of hard disk. This will reduce overhead time of reading from a slow speed device twice. But when file size is large, it cannot be stored in main memory. In this case it is unavoidable to read second time from hard disk drive. Though currently hard disk drives with data transfer rate 3 to 5 KB/second are available, still it will take significant time. With original method of Huffman coding, there is no way through which we can avoid it. Block Huffman coding can solve the problems of Traditional Huffman coding that are mentioned.

5.2 Proposed Algorithms for Block Huffman Coding

Proposed method is pretty simple. The main idea is to break the input stream into blocks of particular size and compressing each block separately. A block size must be chosen in such a way that it can be stored in main memory. Algorithm

5.1-5.2 illustrates how the Block Huffman coding works for static and continuous data.

Step 1: Read a block from the File into main memory.
Step 2: Build the Huffman tree and code for this Block.
Step 3: Compress this block by reading it from main memory.
Step 4: Put the header and compressed data to output file.
Step 5: If there is more data in input file, go to step1. Otherwise coding is ended

Algorithm 5.1: Block Huffman Coding for Static Data

Step 1: Read a block from the stream into main memory.
Step 2: If the block is not completed, then goto step1.
Step 3: Build the Huffman tree and code for this Block.
Step 4: Compress this block by reading it from main memory.
Step 5: Put the header and compressed data to output stream.
Step 6: If there is more data in input file, go to step1. Otherwise coding is ended.

Algorithm 5.2: Block Huffman Coding for Continuous Stream

How It Solves Our Problems

This method can handle both the drawbacks mentioned above. In the first case, as we are reading the file from hard disk only once, compression speed increases significantly because second pass reading is done from the main memory that is much faster than the hard disk. Now the file size may be as large as we can imagine without suffering double penalty for reading two times from hard disk drive. So first problem mentioned above is solved.

Multiple Header Storage

There is a potential problem of increasing size of compressed data due to the storage space of multiple headers for a single file. The problem may be shown pictorially in the following way

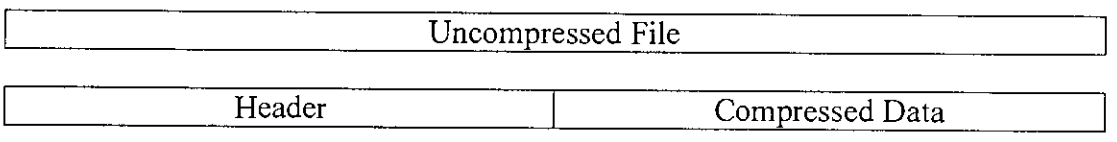


Figure 5.1: Uncompressed and Compressed File in Huffman Coding

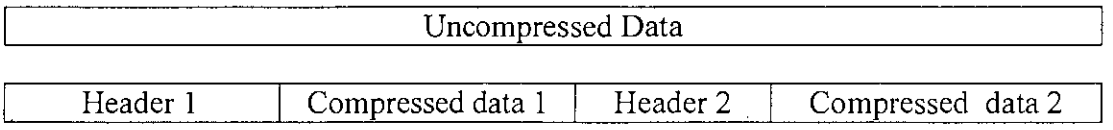


Figure 5.2: Block Huffman Coding

In this method as the number of blocks increases, the overhead for storing multiple headers becomes significant. This causes penalty in compression ratio. That's why a redesign of the storage method of multiple headers is needed.

Block Size for Block Huffman Coding

The most important factor for Block Huffman coding is the selection of block size. The main limiting factor here is the size of the usable main memory. If we take block size to be as small as 1Kbyte there would be large number of blocks. However this will incur very less memory overhead. But the storage overhead is significant because of storing a tree header for each block. The storage overhead decreases with the increasing size of block. But a large size block must be accommodated in the main memory. So a moderate size of block is required for Block Huffman coding.

Advantages of Block Huffman Coding

- In some cases Block Huffman coding provides better compression ratio than Traditional Huffman coding because of the locality characteristics of the new method.
- In terms of reading time from the secondary devices Block Huffman coding is more advantageous than traditional one.

5.3 Conclusion

Moderate block size provides better efficiency for the Block Huffman coding. Block size does not depend on file types. So we can use this method as a general method of coding.

CHAPTER 6

DESIGN OF EXPERIMENTS AND RESULTS

6.1 **Introduction**

This chapter wants to establish effectiveness of Repeated and Block Huffman coding, and impact of efficient representation of Huffman tree on repetition count. The following files are used for the experiments.

- .C File
- .PDF File
- .TXT File
- .DOC File
- .RTF File
- .EXE File
- .HTM File
- .BAK File
- .GIF File
- .BMP File
- .JPG File
- .LOG File
- .DIC File

For assessing effectiveness of methods compression ratio is used.
Compression ratio is defined as

$$\text{Compression ratio} = \frac{Original - Compressed}{Original} \times 100$$

File containing Huffman tree has the format that is discussed in section 6.2. Repeated Huffman coding was first used with normal coding of the tree and then memory efficient coding was used to see whether repetition count increases.

A Huffman tree representation is also related to average code length for a symbol in a message. Average code length can be defined as

$$\text{Average code length, AL} = \sum_{i=1}^{n} p_i l_i$$

Where p_i is the probability of i^{th} symbol
And l_i is the code-length of i^{th} symbol

The efficiency of Huffman coding in terms of size reduction is the result of difference of frequencies of characters. When this difference is less, the probability of degeneration of Huffman coding is more. Consequently compression ratio decreases. So compression ratio depends on standard deviation.

Standard deviation for a symbol from its average code length can be defined by the following formula

$$\text{Standard deviation, SD} = \sqrt{\frac{\sum_{i=1}^{n} (AL - l_i)^2}{n}}$$

Repetition count is also very important in the context of Repeated Huffman coding. Repetition count is the number of times Repeated Huffman coding can be continued on a file up to the time when it is no longer compressed significantly.

6.2 A Huffman Tree Storage Format in the Compressed File

Total number of tree information	Tree representation information	Number of bits for the last representation symbol of tree	Total number of distinct symbols in a Huffman tree	Character symbols

Figure 6.1: Tree Header for Level order Technique

Total number of tree information	Tree representation information	Number of bits for the last representation symbol of tree	Total number of distinct symbols in a Huffman tree	Character symbols

Figure 6.2: Tree Header for Modified Level order Technique

Total number of tree information	Tree representation information	Number of bits for the last representation symbol of tree	Total number of distinct symbols in a Huffman tree	Character symbols

Figure 6.3: Tree Header for Preorder Technique

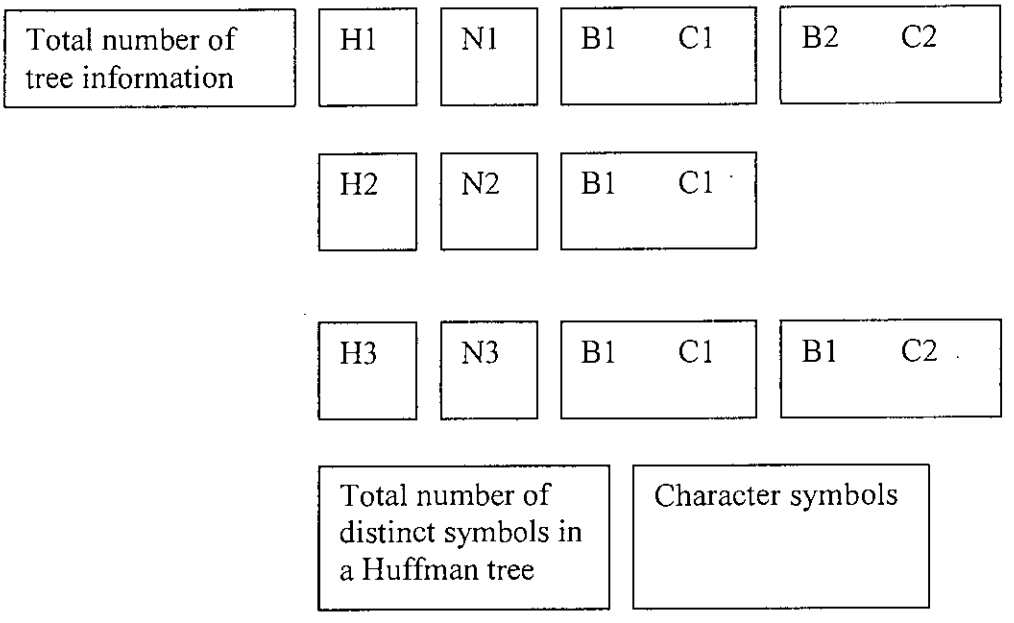
Total number of tree information	Tree representation information	Number of bits for the last representation symbol of tree	Total number of distinct symbols in a Huffman tree	Character symbols

Figure 6.4: Tree Header for Modified Preorder Technique

Total number of tree information	B_1 C_1	...	B_k C_k	Total number of distinct symbols in a Huffman tree	Character symbols

B_i=Total number of bits for the code of the root of a circular leaf node
C_i= The code of the root of a circular leaf node

Figure 6.5: Tree Header for Circular Leaf node Technique



H1= i_1 Height Subtrees, N1=Number of subtrees of H1 height

H2= i_2 Height Subtrees, N2=Number of subtrees of H2 height

H3= i_3 Height Subtrees, N3=Number of subtrees of H3 height

B_i=Total number of bits for the code of the root of that subtree

C_i= The code of the root of that subtree

Figure 6.6: Tree Header for Balanced Binary Tree Technique

6.3 Experimental Results

This section shows the experimental data such as compression ratio, tree size and standard deviation of existing and proposed techniques of Huffman tree representation. It also focuses on compression ratio and tree overhead for Block Huffman coding.

6.3.1 The Existing Technique

Table 6.1-6.5 shows the experimental data of the existing technique.

TABLE 6.1: Compression Ratio for Circular Leaf node Technique

File Name	Original Size (Bytes)	Compression Ratio (%)					Repeated Huffman Compression Ratio(%)
		Pass1	Pass2	Pass3	Pass4	Pass5	
Test4.txt	01209956	65.174	11.053	1.514	N/A	N/A	69.492
Test3.doc	03997184	46.184	0.616	0.004	N/A	N/A	46.517
Test2.c	01000679	40.072	6.710	0.377	0.004	N/A	44.310
Test2.bak	01000679	40.072	6.710	0.377	0.004	N/A	44.310
Test5.rtf	07706903	38.670	1.939	0.0155	N/A	N/A	39.870
Game.exe	04387088	16.561	0.349	N/A	N/A	N/A	16.853
Test7.htm	06196553	33.815	1.785	0.013	N/A	N/A	35.000
Test8.pdf	13300157	24.790	1.373	0.008	N/A	N/A	25.830
K1.bmp	01678134	80.222	56.885	21.959	4.558	0.215	93.660
Ast.gif	03159678	0.479	N/A	N/A	N/A	N/A	0.479
Astha.gif	00496564	0.495	N/A	N/A	N/A	N/A	0.495

TABLE 6.2: A Huffman Tree Size for Circular Leaf node Technique

File Name	Original Size (Bytes)	A Huffman Tree Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	143	195	500	N/A	N/A
Test3.doc	03997184	458	262	258	N/A	N/A
Test2.c	01000679	138	324	310	325	N/A
Test2.bak	01000679	138	324	310	325	N/A
Test5.rtf	07706903	133	413	263	N/A	N/A
Game.exe	04387088	258	315	N/A	N/A	N/A
Test7.htm	06196553	173	264	505	N/A	N/A
Test8.pdf	13300157	571	388	497	N/A	N/A
K1.bmp	01678134	030	247	235	336	318
Ast.gif	03159678	444	N/A	N/A	N/A	N/A
Astha.gif	00496564	461	N/A	N/A	N/A	N/A

TABLE 6.3: Standard Deviation for Circular Leaf node Technique

File Name	Original Size (Bytes)	Standard Deviation				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	8.228	1.818	0.473	N/A	N/A
Test3.doc	03997184	6.110	0.492	0.153	N/A	N/A
Test2.c	01000679	4.506	1.738	0.445	0.217	N/A
Test2.bak	01000679	4.506	1.738	0.445	0.217	N/A
Test5.rtf	07706903	5.989	0.983	0.153	N/A	N/A
Game.exe	04387088	2.705	0.347	N/A	N/A	N/A
Test7.htm	06196553	6.862	0.942	0.188	N/A	N/A
Test8.pdf	13300157	3.657	0.766	0.188	N/A	N/A
K1.bmp	01678134	6.288	10.983	5.358	1.410	0.497
Ast.gif	03159678	0.504	N/A	N/A	N/A	N/A
Astha.gif	00496564	0.495	N/A	N/A	N/A	N/A

TABLE 6.4: Compressed File Size for Circular Leaf node Technique

File Name	Original Size (Bytes)	Compressed File Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	00421384	0374808	0369132	N/A	N/A
Test3.doc	03997184	02151142	2137895	2137803	N/A	N/A
Test2.c	01000679	00599687	0559446	0557339	557318	N/A
Test2.bak	01000679	00599687	0559446	0557339	557318	N/A
Test5.rtf	07706903	04726620	4634984	4634266	N/A	N/A
Game.exe	04387088	03660526	3647734	N/A	N/A	N/A
Test7.htm	06196553	04101214	4028013	4027486	N/A	N/A
Test8.pdf	13300157	10002995	9865591	9864775	N/A	N/A
K1.bmp	01678134	00331907	0143101	0111678	106588	106359
Ast.gif	03159678	03144562	N/A	N/A	N/A	N/A
Astha.gif	00496564	00494106	N/A	N/A	N/A	N/A

TABLE 6.5: Average Code Length for Circular Leaf node Technique

File Name	Original Size (Bytes)	Average Code Length				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	2.785	7.112	7.868	N/A	N/A
Test3.doc	03997184	4.304	7.949	7.999	N/A	N/A
Test2.c	01000679	4.793	7.459	7.966	7.995	N/A
Test2.bak	01000679	4.793	7.459	7.966	7.995	N/A
Test5.rtf	07706903	4.906	7.844	7.998	N/A	N/A
Game.exe	04387088	6.675	7.971	N/A	N/A	N/A
Test7.htm	06196553	5.295	7.857	7.998	N/A	N/A
Test8.pdf	13300157	6.017	7.890	7..999	N/A	N/A
K1.bmp	01678134	1.582	3.443	6.230	7.611	7.959
Ast.gif	03159678	7.961	N/A	N/A	N/A	N/A
Astha.gif	00496564	7.953	N/A	N/A	N/A	N/A

6.3.2 The Proposed Techniques

Table 6.6-6.30 shows the experimental data of the proposed techniques.

TABLE 6.6: Compression Ratio for Level Order Technique

File Name	Original Size (Bytes)	Compression Ratio (%)					Repeated Huffman Compression Ratio (%)
		Pass1	Pass2	Pass3	Pass4	Pass5	
Test4.txt	01209956	65.176	11.024	1.560	N/A	N/A	69.500
Test3.doc	03997184	46.187	00.613	N/A	N/A	N/A	46.517
Test2.c	01000679	40.075	06.712	0.383	N/A	N/A	44.311
Test2.bak	01000679	40.075	06.712	0.383	N/A	N/A	44.311
Test5.rtf	07706903	38.671	01.941	0.013	N/A	N/A	39.870
Game.exe	04387088	16.560	00.349	N/A	N/A	N/A	16.851
Test7.htm	06196553	33.817	01.777	N/A	N/A	N/A	34.990
Test8.pdf	13300157	24.792	01.375	0.0107	N/A	N/A	25.834
K1.bmp	01678134	80.222	57.090	22.089	4.289	0.381	93.690
Ast.gif	03159678	0.482	N/A	N/A	N/A	N/A	0.482
Astha.gif	00496564	0.523	N/A	N/A	N/A	N/A	0.523

TABLE 6.7: A Huffman Tree Size for Level Order Technique

File Name	Original Size (Bytes)	A Huffman Tree Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	115	323	322	N/A	N/A
Test3.doc	03997184	323	323	N/A	N/A	N/A
Test2.c	01000679	108	323	323	N/A	N/A
Test2.bak	01000679	108	323	323	N/A	N/A
Test5.rtf	07706903	103	322	323	N/A	N/A
Game.exe	04387088	323	322	N/A	N/A	N/A
Test7.htm	06196553	054	322	N/A	N/A	N/A
Test8.pdf	13300157	322	322	322	N/A	N/A
K1.bmp	01678134	025	156	300	323	322
Ast.gif	03159678	323	N/A	N/A	N/A	N/A
Astha.gif	00496564	323	N/A	N/A	N/A	N/A

TABLE 6.8: Standard Deviation for Level Order Technique

File Name	Original Size (Bytes)	Standard Deviation				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	8.228	1.818	0.473	N/A	N/A
Test3.doc	03997184	6.110	0.492	N/A	N/A	N/A
Test2.c	01000679	4.506	1.815	0.471	N/A	N/A
Test2.bak	01000679	4.506	1.815	0.471	N/A	N/A
Test5.rtf	07706903	5.989	0.983	0.108	N/A	N/A
Game.exe	04387088	2.705	0.347	N/A	N/A	N/A
Test7.htm	06196553	6.862	0.942	N/A	N/A	N/A
Test8.pdf	13300157	3.657	0.766	0.188	N/A	N/A
K1.bmp	01678134	6.288	10.951	5.063	1.307	0.512
Ast.gif	03159678	0.504	N/A	N/A	N/A	N/A
Astha.gif	00496564	0.518	N/A	N/A	N/A	N/A

TABLE 6.9: Compressed File Size for Level Order Technique

File Name	Original Size (Bytes)	Compressed File Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	00421356	0374907	0369058	N/A	N/A
Test3.doc	03997184	02151007	2137818	N/A	N/A	N/A
Test2.c	01000679	00599657	0559407	0557265	N/A	N/A
Test2.bak	01000679	00599657	0559407	0557265	N/A	N/A
Test5.rtf	07706903	04726590	4634861	4634271	N/A	N/A
Game.exe	04387088	03660591	3647806	N/A	N/A	N/A
Test7.htm	06196553	04101056	4028193	N/A	N/A	N/A
Test8.pdf	13300157	10002746	9865231	9864172	N/A	N/A
K1.bmp	01678134	00331902	0142419	0110959	106199	105795
Ast.gif	03159678	03144441	N/A	N/A	N/A	N/A
Astha.gif	00496564	00493968	N/A	N/A	N/A	N/A

TABLE 6.10: Average Code Length for Level Order Technique

File Name	Original Size (Bytes)	Average Code Length				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	2.785	7.112	7.868	N/A	N/A
Test3.doc	03997184	4.304	7.949	N/A	N/A	N/A
Test2.c	01000679	4.793	7.459	7.965	N/A	N/A
Test2.bak	01000679	4.793	7.459	7.965	N/A	N/A
Test5.rtf	07706903	4.906	7.844	7.999	N/A	N/A
Game.exe	04387088	6.675	7.972	N/A	N/A	N/A
Test7.htm	06196553	5.295	7.857	7.997	N/A	N/A
Test8.pdf	13300157	6.017	7.889	7.998	N/A	N/A
K1.bmp	01678134	1.582	3.443	6.216	7.634	7.945
Ast.gif	03159678	7.961	N/A	N/A	N/A	N/A
Astha.gif	00496564	7.953	N/A	N/A	N/A	N/A

TABLE 6.11: Compression Ratio for Modified Level Order Technique

File Name	Original Size (Bytes)	Compression Ratio (%)					Repeated Huffman Compression Ratio (%)
		Pass1	Pass2	Pass3	Pass4	Pass5	
Test4.txt	01209956	65.176	11.024	1.553	N/A	N/A	69.500
Test3.doc	03997184	46.187	00.613	N/A	N/A	N/A	46.517
Test2.c	01000679	40.075	06.712	0.383	N/A	N/A	44.312
Test2.bak	01000679	40.075	06.712	0.383	N/A	N/A	44.312
Test5.rtf	07706903	38.671	01.941	0.014	N/A	N/A	39.870
Game.exe	04387088	16.560	00.349	N/A	N/A	N/A	16.851
Test7.htm	06196553	33.817	01.784	0.022	N/A	N/A	35.010
Test8.pdf	13300157	24.792	01.375	0.011	N/A	N/A	25.834
K1.bmp	01678134	80.222	56.916	21.895	3.837	N/A	93.600
Ast.gif	03159678	0.482	N/A	N/A	N/A	N/A	0.482
Astha.gif	00496564	0.523	N/A	N/A	N/A	N/A	0.523

TABLE 6.12: A Huffman Tree Size for Modified Level Order Technique

File Name	Original Size (Bytes)	A Huffman Tree Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	114	323	322	N/A	N/A
Test3.doc	03997184	313	318	N/A	N/A	N/A
Test2.c	01000679	108	323	324	N/A	N/A
Test2.bak	01000679	108	323	324	N/A	N/A
Test5.rtf	07706903	103	322	322	N/A	N/A
Game.exe	04387088	324	322	N/A	N/A	N/A
Test7.htm	06196553	054	322	322	N/A	N/A
Test8.pdf	13300157	322	322	322	N/A	N/A
K1.bmp	01678134	025	156	310	324	N/A
Ast.gif	03159678	324	N/A	N/A	N/A	N/A
Astha.gif	00496564	323	N/A	N/A	N/A	N/A

TABLE 6.13: Standard Deviation for Modified Level Order Technique

File Name	Original Size (Bytes)	Standard Deviation				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	8.228	1.818	0.473	N/A	N/A
Test3.doc	03997184	6.110	0.492	N/A	N/A	N/A
Test2.c	01000679	4.506	1.815	0.471	N/A	N/A
Test2.bak	01000679	4.506	1.815	0.471	N/A	N/A
Test5.rtf	07706903	5.989	0.983	0.108	N/A	N/A
Game.exe	04387088	2.705	0.347	N/A	N/A	N/A
Test7.htm	06196553	6.862	0.942	0.188	N/A	N/A
Test8.pdf	13300157	3.657	0.766	0.188	N/A	N/A
K1.bmp	01678134	6.288	10.983	5.284	1.270	N/A
Ast.gif	03159678	0.504	N/A	N/A	N/A	N/A
Astha.gif	00496564	0.518	N/A	N/A	N/A	N/A

TABLE 6.14: Compressed File Size for Modified Level Order Technique

File Name	Original Size (Bytes)	Compressed File Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	00421355	0374906	0369082	N/A	N/A
Test3.doc	03997184	02150997	2137803	N/A	N/A	N/A
Test2.c	01000679	00599657	0559407	0557265	N/A	N/A
Test2.bak	01000679	00599657	0559407	0557265	N/A	N/A
Test5.rtf	07706903	04726590	4634860	4634198	N/A	N/A
Game.exe	04387088	03660592	3647807	N/A	N/A	N/A
Test7.htm	06196553	04101095	4027946	4027080	N/A	N/A
Test8.pdf	13300157	10002746	9865231	9864172	N/A	N/A
K1.bmp	01678134	00331902	0142997	0111688	107402	N/A
Ast.gif	03159678	03144442	N/A	N/A	N/A	N/A
Astha.gif	00496564	00493968	N/A	N/A	N/A	N/A

TABLE 6.15: Average Code Length for Modified Level Order Technique

File Name	Original Size (Bytes)	Average Code Length				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	2.785	7.112	7.869	N/A	N/A
Test3.doc	03997184	4.304	7.949	N/A	N/A	N/A
Test2.c	01000679	4.793	7.459	7.965	N/A	N/A
Test2.bak	01000679	4.793	7.459	7.965	N/A	N/A
Test5.rtf	07706903	4.906	7.844	7.998	N/A	N/A
Game.exe	04387088	6.675	7.971	N/A	N/A	N/A
Test7.htm	06196553	5.295	7.857	7.998	N/A	N/A
Test8.pdf	13300157	6.017	7.889	7.999	N/A	N/A
K1.bmp	01678134	1.582	3.443	6.231	7.669	N/A
Ast.gif	03159678	7.961	N/A	N/A	N/A	N/A
Astha.gif	00496564	7.953	N/A	N/A	N/A	N/A

TABLE 6.16: Compression Ratio for Preorder Technique

File Name	Original Size (Bytes)	Compression Ratio (%)					Repeated Huffman Compression Ratio (%)
		Pass1	Pass2	Pass3	Pass4	Pass5	
Test4.txt	01209956	65.176	11.084	1.562	0.014	N/A	69.524
Test3.doc	03997184	46.187	00.624	0.013	N/A	N/A	46.530
Test2.c	01000679	40.075	06.734	0.419	0.017	N/A	44.354
Test2.bak	01000679	40.075	06.734	0.419	0.017	N/A	44.354
Test5.rtf	07706903	38.671	01.942	0.020	N/A	N/A	39.874
Game.exe	04387088	16.561	00.349	N/A	N/A	N/A	16.853
Test7.htm	06196553	33.817	01.788	0.018	N/A	N/A	35.010
Test8.pdf	13300157	24.792	01.376	0.012	N/A	N/A	25.840
K1.bmp	01678134	80.222	56.945	22.117	3.329	0.339	93.610
Ast.gif	03159678	0.483	N/A	N/A	N/A	N/A	0.483
Astha.gif	00496564	0.526	N/A	N/A	N/A	N/A	0.526

TABLE 6.17: A Huffman Tree Size for Preorder Technique

File Name	Original Size (Bytes)	A Huffman Tree Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	115	68	315	96	N/A
Test3.doc	03997184	322	89	71	N/A	N/A
Test2.c	01000679	108	197	122	142	N/A
Test2.bak	01000679	108	197	122	142	N/A
Test5.rtf	07706903	103	279	75	N/A	N/A
Game.exe	04387088	258	315	N/A	N/A	N/A
Test7.htm	06196553	21	136	319	N/A	N/A
Test8.pdf	13300157	317	241	312	N/A	N/A
K1.bmp	01678134	23	59	130	211	278
Ast.gif	03159678	324	N/A	N/A	N/A	N/A
Astha.gif	00496564	306	N/A	N/A	N/A	N/A

TABLE 6.18: Standard Deviation for Preorder Technique

File Name	Original Size (Bytes)	Standard Deviation				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	8.228	1.818	0.473	0.188	N/A
Test3.doc	03997184	6.110	0.492	0.153	N/A	N/A
Test2.c	01000679	4.506	1.815	0.458	0.153	N/A
Test2.bak	01000679	4.506	1.815	0.458	0.153	N/A
Test5.rtf	07706903	5.989	0.983	0.153	N/A	N/A
Game.exe	04387088	2.705	0.347	N/A	N/A	N/A
Test7.htm	06196553	6.862	0.942	0.153	N/A	N/A
Test8.pdf	13300157	3.657	0.766	0.188	N/A	N/A
K1.bmp	01678134	6.288	11.027	5.206	1.104	0.534
Ast.gif	03159678	0.504	N/A	N/A	N/A	N/A
Astha.gif	00496564	0.518	N/A	N/A	N/A	N/A

TABLE 6.19: Compressed File Size for Preorder Technique

File Name	Original Size (Bytes)	Compressed File Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	00421356	0374652	0368801	368750	N/A
Test3.doc	03997184	02151006	2137585	2137309	N/A	N/A
Test2.c	01000679	00599657	0559279	0556933	556839	N/A
Test2.bak	01000679	00599657	0559279	0556933	556839	N/A
Test5.rtf	07706903	04726590	4634818	4633887	N/A	N/A
Game.exe	04387088	03660526	3647734	N/A	N/A	N/A
Test7.htm	06196553	04101062	4027726	4026983	N/A	N/A
Test8.pdf	13300157	10002741	9865147	9863988	N/A	N/A
K1.bmp	01678134	00331900	0142899	0111294	107589	107224
Ast.gif	03159678	03144409	N/A	N/A	N/A	N/A
Astha.gif	00496564	00493951	N/A	N/A	N/A	N/A

TABLE 6.20: Average Code Length for Preorder Technique

File Name	Original Size (Bytes)	Average Code Length				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	2.785	7.112	7.868	7.997	N/A
Test3.doc	03997184	4.304	7.949	7.999	N/A	N/A
Test2.c	01000679	4.793	7.459	7.965	7.997	N/A
Test2.bak	01000679	4.793	7.459	7.965	7.997	N/A
Test5.rtf	07706903	4.906	7.844	7.998	N/A	N/A
Game.exe	04387088	6.675	7.971	N/A	N/A	N/A
Test7.htm	06196553	5.295	7.857	7.997	N/A	N/A
Test8.pdf	13300157	6.017	7.889	7.999	N/A	N/A
K1.bmp	01678134	1.582	3.443	6.223	7.718	7.952
Ast.gif	03159678	7.961	N/A	N/A	N/A	N/A
Astha.gif	00496564	7.953	N/A	N/A	N/A	N/A

TABLE 6.21: Compression Ratio for Modified Preorder Technique

File Name	Original Size (Bytes)	Compression Ratio (%)					Repeated Huffman Compression Ratio (%)
		Pass1	Pass2	Pass3	Pass4	Pass5	
Test4.txt	01209956	65.176	11.084	1.615	0.0317	N/A	69.546
Test3.doc	03997184	46.187	0.624	0.012	N/A	N/A	46.529
Test2.c	01000679	40.075	6.734	0.416	0.005	N/A	44.346
Test2.bak	01000679	40.075	6.734	0.416	0.005	N/A	44.346
Test5.rtf	07706903	38.671	1.942	0.019	N/A	N/A	39.873
Game.exe	04387088	16.561	0.349	N/A	N/A	N/A	16.853
Test7.htm	06196553	33.817	1.788	0.018	N/A	N/A	35.010
Test8.pdf	13300157	24.792	1.376	0.0116	N/A	N/A	25.840
K1.bmp	01678134	80.222	56.933	22.081	3.644	0.443	93.633
Ast.gif	03159678	0.483	N/A	N/A	N/A	N/A	0.483
Astha.gif	00496564	0.526	N/A	N/A	N/A	N/A	0.526

TABLE 6.22: A Huffman Tree Size for Modified Preorder Technique

File Name	Original Size (Bytes)	A Huffman Tree Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	113	068	067	072	N/A
Test3.doc	03997184	322	089	071	N/A	N/A
Test2.c	01000679	108	197	146	320	N/A
Test2.bak	01000679	108	197	146	320	N/A
Test5.rtf	07706903	103	279	077	N/A	N/A
Game.exe	04387088	258	315	N/A	N/A	N/A
Test7.htm	06196553	012	136	319	N/A	N/A
Test8.pdf	13300157	317	241	312	N/A	N/A
K1.bmp	01678134	022	103	121	157	284
Ast.gif	03159678	291	N/A	N/A	N/A	N/A
Astha.gif	00496564	306	N/A	N/A	N/A	N/A

TABLE 6.23: Standard Deviation for Modified Preorder Technique

File Name	Original Size (Bytes)	Standard Deviation				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	8.228	1.818	0.334	0.217	N/A
Test3.doc	03997184	6.110	0.492	0.153	N/A	N/A
Test2.c	01000679	4.506	1.815	0.440	0.109	N/A
Test2.bak	01000679	4.506	1.815	0.440	0.109	N/A
Test5.rtf	07706903	5.989	0.983	0.108	N/A	N/A
Game.exe	04387088	2.707	0.347	N/A	N/A	N/A
Test7.htm	06196553	6.862	0.942	0.187	N/A	N/A
Test8.pdf	13300157	3.657	0.766	0.188	N/A	N/A
K1.bmp	01678134	6.288	10.972	5.197	1.176	0.543
Ast.gif	03159678	0.504	N/A	N/A	N/A	N/A
Astha.gif	00496564	0.518	N/A	N/A	N/A	N/A

TABLE 6.24: Compressed File Size for Modified Preorder Technique

File Name	Original Size (Bytes)	Compressed File Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	00421354	374650	0368601	368484	N/A
Test3.doc	03997184	02151006	2137585	2137331	N/A	N/A
Test2.c	01000679	00599657	0559279	0556951	556922	N/A
Test2.bak	01000679	00599657	0559279	0556951	556922	N/A
Test5.rtf	07706903	04726590	4634818	4633919	N/A	N/A
Game.exe	04387088	03660526	3647734	N/A	N/A	N/A
Test7.htm	06196553	04101053	4027717	4027001	N/A	N/A
Test8.pdf	13300157	10002741	9865147	9864006	N/A	N/A
K1.bmp	01678134	00331899	00142939	0111377	107319	106844
Ast.gif	03159678	03144409	N/A	N/A	N/A	N/A
Astha.gif	00496564	00 493951	N/A	N/A	N/A	N/A

TABLE 6.25: Average Code Length for Modified Preorder Technique

File Name	Original Size (Bytes)	Average Code Length				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	2.785	7.112	7.869	7.996	N/A
Test3.doc	03997184	4.304	7.949	7.999	N/A	N/A
Test2.c	01000679	4.793	7.459	7.964	7.995	N/A
Test2.bak	01000679	4.793	7.459	7.964	7.995	N/A
Test5.rtf	07706903	4.906	7.844	7.998	N/A	N/A
Game.exe	04387088	6.674	7.971	N/A	N/A	N/A
Test7.htm	06196553	5.295	7.856	7.997	N/A	N/A
Test8.pdf	13300157	6.017	7.889	7.998	N/A	N/A
K1.bmp	01678134	1.582	3.443	6.227	7.697	7.943
Ast.gif	03159678	7.961	N/A	N/A	N/A	N/A
Astha.gif	00496564	7.953	N/A	N/A	N/A	N/A

TABLE 6.26: Compression Ratio for Balanced Binary Tree Technique

File Name	Original Size (Bytes)	Compression Ratio (%)					Repeated Huffman Compression Ratio (%)
		Pass1	Pass2	Pass3	Pass4	Pass5	
Test4.txt	01209956	65.173	11.057	1.569	N/A	N/A	69.511
Test3.doc	03997184	46.184	0.622	0.015	N/A	N/A	46.527
Test2.c	01000679	40.072	6.712	0.397	N/A	N/A	44.316
Test2.bak	01000679	40.072	6.712	0.397	N/A	N/A	44.316
Test5.rtf	07706903	38.670	1.939	0.020	N/A	N/A	39.872
Game.exe	04387088	16.560	0.348	N/A	N/A	N/A	16.850
Test7.htm	06196553	33.815	1.785	0.019	N/A	N/A	35.010
Test8.pdf	13300157	24.792	1.374	0.012	N/A	N/A	25.840
K1.bmp	01678134	80.222	56.922	21.981	3.661	0.008	93.600
Ast.gif	03159678	0.481	N/A	N/A	N/A	N/A	0.481
Astha.gif	00496564	0.509	N/A	N/A	N/A	N/A	0.509

TABLE 6.27: A Huffman Tree Size for Balanced Binary Tree Technique

File Name	Original Size (Bytes)	A Huffman Tree Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	145	177	308	N/A	N/A
Test3.doc	03997184	441	124	53	N/A	N/A
Test2.c	01000679	140	314	173	N/A	N/A
Test2.bak	01000679	140	314	173	N/A	N/A
Test5.rtf	07706903	135	395	59	N/A	N/A
Game.exe	04387088	380	348	N/A	N/A	N/A
Test7.htm	06196553	175	240	303	N/A	N/A
Test8.pdf	13300157	340	351	288	N/A	N/A
K1.bmp	01678134	030	123	225	199	259
Ast.gif	03159678	378	N/A	N/A	N/A	N/A
Astha.gif	00496564	391	N/A	N/A	N/A	N/A

TABLE 6.28: Standard Deviation for Balanced Binary Tree Technique

File Name	Original Size (Bytes)	Standard Deviation				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	8.228	1.818	0.473	N/A	N/A
Test3.doc	03997184	6.110	0.492	0.188	N/A	N/A
Test2.c	01000679	4.506	1.738	0.458	N/A	N/A
Test2.bak	01000679	4.506	1.738	0.458	N/A	N/A
Test5.rtf	07706903	5.989	0.983	0.153	N/A	N/A
Game.exe	04387088	2.707	0.347	N/A	N/A	N/A
Test7.htm	06196553	6.862	0.942	0.188	N/A	N/A
Test8.pdf	13300157	3.657	0.766	0.188	N/A	N/A
K1.bmp	01678134	6.288	10.929	5.223	1.083	0.463
Ast.gif	03159678	0.504	N/A	N/A	N/A	N/A
Astha.gif	00496564	0.518	N/A	N/A	N/A	N/A

TABLE 6.29: Compressed File Size for Balanced Binary Tree Technique

File Name	Original Size (Bytes)	Compressed File Size(Bytes)				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	00421386	0374792	0368910	N/A	N/A
Test3.doc	03997184	02151125	2137740	2137415	N/A	N/A
Test2.c	01000679	00599689	0559438	0557217	N/A	N/A
Test2.bak	01000679	00599689	0559438	0557217	N/A	N/A
Test5.rtf	07706903	04726622	4634968	4634046	N/A	N/A
Game.exe	04387088	03660648	3647891	N/A	N/A	N/A
Test7.htm	06196553	04101216	4027992	4027210	N/A	N/A
Test8.pdf	13300157	10002764	9865284	9864123	N/A	N/A
K1.bmp	01678134	00331907	0142978	0111550	107466	107457
Ast.gif	03159678	03144496	N/A	N/A	N/A	N/A
Astha.gif	00496564	00494036	N/A	N/A	N/A	N/A

TABLE 6.30: Average Code Length for Balanced Binary Tree Technique

File Name	Original Size (Bytes)	Average Code Length				
		Pass1	Pass2	Pass3	Pass4	Pass5
Test4.txt	01209956	2.785	7.112	7.868	N/A	N/A
Test3.doc	03997184	4.304	7.949	7.999	N/A	N/A
Test2.c	01000679	4.793	7.459	7.966	N/A	N/A
Test2.bak	01000679	4.793	7.459	7.966	N/A	N/A
Test5.rtf	07706903	4.906	7.844	7.998	N/A	N/A
Game.exe	04387088	6.675	7.971	N/A	N/A	N/A
Test7.htm	06196553	5.295	7.857	7.998	N/A	N/A
Test8.pdf	13300157	6.017	7.889	7.999	N/A	N/A
K1.bmp	01678134	1.582	3.443	6.229	7.693	7.979
Ast.gif	03159678	7.961	N/A	N/A	N/A	N/A
Astha.gif	00496564	7.953	N/A	N/A	N/A	N/A

6.3.3 Experimental Data for Benchmark File

TABLE 6.31: Data for WORLD95.TXT File

File Name: WORLD95.TXT File Size: 2988578 Bytes						
Technique of Huffman Tree Representation		**Pass1**	**Pass2**	**Pass3**	**Pure Huffman Compression Ratio(%)**	**Repeated Huffman Compression Ratio (%)**
Balanced Binary Tree Technique	**SD**	6.726	0.652	N/A	35.756	36.300
	TSIZE	166	325	N/A		
	CFSIZE	1919984	1903815	N/A		
	CRATIO	35.756	0.842	N/A		
	ACL	5.139	7.931	N/A		
Level order Technique	**SD**	6.726	0.652	N/A	35.761	36.300
	TSIZE	4	322	N/A		
	CFSIZE	1919822	1903644	N/A		
	CRATIO	35.762	0.843	N/A		
	ACL	5.139	7.931	N/A		
Modified Level order Technique	**SD**	6.726	0.652	N/A	35.762	36.300
	TSIZE	4	322	N/A		
	CFSIZE	1919822	1903644	N/A		
	CRATIO	35.762	0.843	N/A		
	ACL	5.139	7.931	N/A		
Preorder Technique	**SD**	6.726	0.652	N/A	35.762	36.310
	TSIZE	3	224	N/A		
	CFSIZE	1919821	1903545	N/A		
	CRATIO	35.762	0.848	N/A		
	ACL	5.139	7.931	N/A		
Modified Preorder Technique	**SD**	6.726	0.652	N/A	35.762	36.310
	TSIZE	3	224	N/A		
	CFSIZE	1919821	1903545	N/A		
	CRATIO	35.762	0.848	N/A		
	ACL	2988578	7.931	N/A		
Circular Leaf node Technique	**SD**	6.726	0.652	N/A	35.741	36.280
	TSIZE	274	338	N/A		
	CFSIZE	1920423	1904254	N/A		
	CRATIO	35.741	0.842	N/A		
	ACL	5.139	7.931	N/A		

ACL=Average Code Length, SD=Standard Deviation, TSIZE= Huffman tree size in Bytes, CFSIZE= Compressed File Size in Bytes, CRATIO= Compression ratio

TABLE 6.32: Data for OHS.DOC File

File Name: OHS.DOC File Size: 4168192 Bytes						
Technique of Huffman Tree Representation		**Pass1**	**Pass2**	**Pass3**	**Pure Huffman Compression Ratio (%)**	**Repeated Huffman Compression Ratio (%)**
Balanced Binary Tree Technique	**SD**	2.878	0.710	0.188	15.063	15.954
	TSIZE	370	395	71		
	CFSIZE	3540362	3504612	3503204		
	CRATIO	15.063	1.009	0.040		
	ACL	6.794	7.918	7.997		
Level order Technique	**SD**	2.878	0.710	0.217	15.064	15.950
	TSIZE	323	322	323		
	CFSIZE	3540315	3504482	3503437		
	CRATIO	15.064	1.012	0.030		
	ACL	6.794	7.918	7.997		
Modified Level order Technique	**SD**	2.878	0.710	0.188	15.064	15.952
	TSIZE	317	322	322		
	CFSIZE	3540309	3504474	3503277		
	CRATIO	15.064	1.012	0.034		
	ACL	6.794	7.918	7.997		
Preorder Technique	**SD**	2.878	0.710	0.188	15.065	15.954
	TSIZE	265	295	84		
	CFSIZE	3540257	3504392	3503188		
	CRATIO	15.065	1.013	0.035		
	ACL	6.794	7.918	7.997		
Modified Preorder Technique	**SD**	2.878	0.710	0.188	15.065	15.954
	TSIZE	265	295	84		
	CFSIZE	3540257	3504392	3503195		
	CRATIO	15.065	1.013	0.034		
	ACL	6.794	7.918	7.997		
Circular Leaf node Technique	**SD**	2.878	0.710	0.217	15.060	15.947
	TSIZE	429	442	266		
	CFSIZE	3540421	3504728	3503473		
	CRATIO	15.060	1.008	0.036		
	ACL	6.794	7.918	7.997		

TABLE 6.33: Data for FP.LOG File

File Name: FP.LOG File Size: 20617071 Bytes						
Technique of Huffman Tree Representation		**Pass1**	**Pass2**	**Pass3**	**Pure Huffman Compression Ratio (%)**	**Repeated Huffman Compression Ratio (%)**
Balanced Binary Tree Technique	**SD**	8.580	1.095	0.217	32.062	33.424
	TSIZE	3	259	272		
	CFSIZE	14006944	13730227	13726141		
	CRATIO	32.062	1.976	0.029		
	ACL	5.435	7.842	7.997		
Level order Technique	**SD**	8.580	1.095	0.217	32.062	33.423
	TSIZE	4	323	322		
	CFSIZE	14006945	13730292	13726256		
	CRATIO	32.062	1.975	0.029		
	ACL	5.435	7.842	7.998		
Modified Level order Technique	**SD**	8.580	1.095	0.217	32.062	33.423
	TSIZE	4	323	322		
	CFSIZE	14006945	13730292	13726256		
	CRATIO	32.062	1.975	0.029		
	ACL	5.435	7.842	7.997		
Preorder Technique	**SD**	8.580	1.096	0.217	32.062	33.424
	TSIZE	3	146	285		
	CFSIZE	14006944	13730114	13726073		
	CRATIO	32.062	1.977	0.029		
	ACL	5.435	7.842	7.998		
Modified Preorder Technique	**SD**	8.580	1.096	0.217	32.062	33.424
	TSIZE	3	146	285		
	CFSIZE	14006944	13730114	13726073		
	CRATIO	32.062	1.977	0.029		
	ACL	5.435	7.842	7.998		
Circular Leaf node Technique	**SD**	8.580	1.096	0.217	32.0614	33.423
	TSIZE	5	317	353		
	CFSIZE	14006944	13730276	13726231		
	CRATIO	32.0614	1.975	0.029		
	ACL	5.435	7.842	7.998		

TABLE 6.34: Data for FLASHMX.PDF File

File Name: FLASHMX.PDF File Size: 4526946 Bytes						
Technique of Huffman Tree Representation		**Pass1**	**Pass2**	**Pass3**	**Pure Huffman Compression Ratio (%)**	**Repeated Huffman Compression Ratio (%)**
Balanced Binary Tree Technique	**SD**	0.735	0.188	N/A	1.928	1.932
	TSIZE	236	290	N/A		
	CFSIZE	4439692	4439475	N/A		
	CRATIO	1.928	0.005	N/A		
	ACL	7.845	7.999	N/A		
Level order Technique	**SD**	0.735	0.188	N/A	1.926	1.930
	TSIZE	323	322	N/A		
	CFSIZE	4439779	4439594	N/A		
	CRATIO	1.926	0.004	N/A		
	ACL	7.845	7.999	N/A		
Modified Level order Technique	**SD**	0.735	0.188	N/A	1.926	1.930
	TSIZE	313	322	N/A		
	CFSIZE	4439769	4439583	N/A		
	CRATIO	1.926	0.004	N/A		
	ACL	7.845	7.999	N/A		
Preorder Technique	**SD**	0.735	0.188	N/A	1.928	1.932
	TSIZE	213	313	N/A		
	CFSIZE	4439669	4439474	N/A		
	CRATIO	1.928	0,004	N/A		
	ACL	7.845	7.999	N/A		
Modified Preorder Technique	**SD**	0.735	0.188	N/A	1.928	1.932
	TSIZE	213	313	N/A		
	CFSIZE	4439669	4439474	N/A		
	CRATIO	1.928	0.004	N/A		
	ACL	7.845	7.999	N/A		
Circular Leaf node Technique	**SD**	0.735	0.188	N/A	1.924	1.924
	TSIZE	386	498	N/A		
	CFSIZE	4439842	4439832	N/A		
	CRATIO	1.924	0.0002	N/A		
	ACL	7.845	7.999	N/A		

TABLE 6.35: Data for ENGLISH.DIC File

File Name: ENGLISH.DIC File Size: 4067439 Bytes						
Technique of Huffman Tree Representation		**Pass1**	**Pass2**	**Pass3**	**Pure Huffman Compression Ratio (%)**	**Repeated Huffman Compression Ratio (%)**
Balanced Binary Tree Technique	**SD**	6.561	0.864	N/A	45.602	46.700
	TSIZE	70	355	N/A		
	CFSIZE	2212619	2167792	N/A		
	CRATIO	45.602	2.026	N/A		
	ACL	4.352	7.837	N/A		
Level order Technique	**SD**	6.561	0.864	N/A	45.602	46.710
	TSIZE	55	322	N/A		
	CFSIZE	2212604	2167744	N/A		
	CRATIO	45.602	2.028	N/A		
	ACL	4.352	7.999	N/A		
Modified Level order Technique	**SD**	6.561	0.864	N/A	45.602	46.710
	TSIZE	54	322	N/A		
	CFSIZE	2212603	2167743	N/A		
	CRATIO	45.602	2.028	N/A		
	ACL	4.352	7.837	N/A		
Preorder Technique	**SD**	6.561	0.864	N/A	45.602	46.710
	TSIZE	55	253	N/A		
	CFSIZE	2212604	2167675	N/A		
	CRATIO	45.602	2.031	N/A		
	ACL	4.352	7.837	N/A		
Modified Preorder Technique	**SD**	6.561	0.864	N/A	45.602	46.710
	TSIZE	54	253	N/A		
	CFSIZE	2212603	2167674	N/A		
	CRATIO	45.602	2.031	N/A		
	ACL	4.352	7.837	N/A		
Circular Leaf node Technique	**SD**	6.561	0.864	N/A	45.602	46.700
	TSIZE	69	391	N/A		
	CFSIZE	2212618	2167827	N/A		
	CRATIO	45.602	2.025	N/A		
	ACL	4.352	7.837	N/A		

TABLE 6.36: Data for A10.JPG File

File Name: A10.JPG File Size: 713569 Bytes						
Technique of Huffman Tree Representation		**Pass1**	**Pass2**	**Pass3**	**Pure Huffman Compression Ratio (%)**	**Repeated Huffman Compression Ratio (%)**
Balanced Binary Tree Technique	**SD**	0.408	N/A	N/A	0.193	0.193
	TSIZE	182	N/A	N/A		
	CFSIZE	712190	N/A	N/A		
	CRATIO	0.193	N/A	N/A		
	ACL	7.983	N/A	N/A		
Level order Technique	**SD**	0.408	N/A	N/A	0.174	0.174
	TSIZE	323	N/A	N/A		
	CFSIZE	712331	N/A	N/A		
	CRATIO	0.174	N/A	N/A		
	ACL	7.983	N/A	N/A		
Modified Level order Technique	**SD**	0.408	N/A	N/A	0.173	0.173
	TSIZE	324	N/A	N/A		
	CFSIZE	712332	N/A	N/A		
	CRATIO	0.173	N/A	N/A		
	ACL	7.983	N/A	N/A		
Preorder Technique	**SD**	0.408	N/A	N/A	0.203	0.203
	TSIZE	115	N/A	N/A		
	CFSIZE	712123	N/A	N/A		
	CRATIO	0.203	N/A	N/A		
	ACL	7.983	N/A	N/A		
Modified Preorder Technique	**SD**	0.408	N/A	N/A	0.203	0.203
	TSIZE	115	N/A	N/A		
	CFSIZE	712123	N/A	N/A		
	CRATIO	0.203	N/A	N/A		
	ACL	7.983	N/A	N/A		
Circular Leaf node Technique	**SD**	0.408	N/A	N/A	0.179	0.179
	TSIZE	282	N/A	N/A		
	CFSIZE	712290	N/A	N/A		
	CRATIO	0.179	N/A	N/A		
	ACL	7.983	N/A	N/A		

TABLE 6.37: Data for RAFALE.BMP File

File Name: RAFALE.BMP File Size: 4149414 Bytes						
Technique of Huffman Tree Representation		**Pass1**	**Pass2**	**Pass3**	**Pure Huffman Compression Ratio (%)**	**Repeated Huffman Compression Ratio (%)**
Balanced Binary Tree Technique	**SD**	4.387	0.582	N/A	31.979	32.434
	TSIZE	86	323	N/A		
	CFSIZE	2822479	2803598	N/A		
	CRATIO	31.979	0.669	N/A		
	ACL	5.442	7.946	N/A		
Level order Technique	**SD**	4.387	0.582	N/A	31.979	32.434
	TSIZE	96	323	N/A		
	CFSIZE	2822489	2803606	N/A		
	CRATIO	31.979	0.669	N/A		
	ACL	5.442	7.946	N/A		
Modified Level order Technique	**SD**	4.387	0.582	N/A	31.979	32.434
	TSIZE	95	319	N/A		
	CFSIZE	2822488	2803601	N/A		
	CRATIO	31.979	0.669	N/A		
	ACL	5.442	7.946	N/A		
Preorder Technique	**SD**	4.387	0.582	N/A	31.979	32.437
	TSIZE	61	231	N/A		
	CFSIZE	2822454	2803479	N/A		
	CRATIO	31.979	0.672	N/A		
	ACL	5.442	7.946	N/A		
Modified Preorder Technique	**SD**	4.387	0.582	N/A	31.979	32.437
	TSIZE	61	231	N/A		
	CFSIZE	2822454	2803479	N/A		
	CRATIO	31.979	0.672	N/A		
	ACL	5.442	7.946	N/A		
Circular Leaf node Technique	**SD**	4.387	0.582	N/A	31.979	32.433
	TSIZE	87	383	N/A		
	CFSIZE	2822480	2803658	N/A		
	CRATIO	31.979	0.667	N/A		
	ACL	5.442	7.946	N/A		

6.3.4 Block Huffman Coding

This section provides experimental data such as compression ratio and tree size for Block Huffman coding when tree is represented by level order and its modified version.

TABLE 6.38: Compression Ratio and Tree size for Block Huffman Coding (Level Order)

Level Order Tree Representation					
File Name	Block Size (Kbytes)	Source File (Bytes)	Compressed File (Bytes)	Compression Ratio(%)	Tree size (Bytes)
Test4.txt	10	01209956	0432446	64.259361	12562
Test3.doc	10	03997184	2170875	45.689891	25634
Test2.c	10	01000679	0609746	39.066774	10582
Test2.bak	10	01000679	0609746	39.066774	10582
Game.exe	10	04387088	3686404	15.971505	28741
Test7.htm	10	06196553	4184803	32.465630	84695

TABLE 6.39: Compression Ratio and Tree size for Block Huffman Coding (M. Level Order)

Modified Level Order Tree Representation					
File Name	Block Size (Kbytes)	Source File (Bytes)	Compressed File (Bytes)	Compression Ratio(%)	Tree size (Bytes)
Test4.txt	10	01209956	0432400	64.263163	12516
Test3.doc	10	03997184	2168747	45.743128	23506
Test2.c	10	01000679	0609714	39.069971	10550
Test2.bak	10	01000679	0609714	39.069971	10582
Game.exe	10	04387088	3685784	15.985638	28121
Test7.htm	10	06196553	4184583	32.469181	84475

6.4 Comparisons among Experimental Results

This section shows different tables and figures to compare data between existing and proposed techniques of a Huffman tree representation. It also compares Block Huffman coding and Traditional Huffman coding.

TABLE 6.40: Compression Ratio Comparison among All Representation Methods

File name	Original Size(Bytes)	Compression Ratio (%) Repeated Huffman Coding Technique					
		Level order	Modified Level order	Preorder	Modified Preorder	Balanced Binary tree	Circular node Tech.
Test4.txt	01209956	69.500	69.500	69.524	69.546	69.511	69.492
Test3.doc	03997184	46.517	46.517	46.530	46.529	46.527	46.517
Test2.c	01000679	44.311	44.312	44.354	44.346	44.316	44.310
Test2.bak	01000679	44.311	44.312	44.354	44.346	44.316	44.310
Test5.rtf	07706903	39.870	39.870	39.874	39.873	39.872	39.870
Game.exe	04387088	16.851	16.851	16.853	16.853	16.850	16.853
Test7.htm	06196553	34.990	35.010	35.010	35.010	35.010	35.000
Test8.pdf	13300157	25.834	25.834	25.840	25.840	25.840	25.830

TABLE 6.41: Compressed File Size Comparison among All Representation Methods

File name	Original Size(Bytes)	Compressed File Size(Bytes) Repeated Huffman Coding Technique					
		Level order	Modified Level order	Preorder	Modified Preorder	Balanced Binary tree	Circular node Tech.
Test4.txt	01209956	0369058	0369082	0368750	0368484	0368910	0369132
Test3.doc	03997184	2137818	2137803	2137309	2137331	2137415	2137803
Test2.c	01000679	0557265	0557265	0556839	0556922	0557217	0557318
Test2.bak	01000679	0557265	0557265	0556839	0556922	0557217	0557318
Test5.rtf	07706903	4634271	4634198	4633887	4633919	4634046	4634266
Game.exe	04387088	3647806	3647807	3647734	3647734	3647891	3647734
Test7.htm	06196553	4028193	4027080	4026983	4027001	4027210	4027486
Test8.pdf	13300157	9864172	9864172	9863988	9864006	9864123	9864775

TABLE 6.42: Comparison between Pure and Repeated Huffman (Circular node Tech.)

Circular node Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Repeated Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	0421384	65.174	0369132	69.492
Test3.doc	03997184	2151142	46.184	2137803	46.517
Test2.c	01000679	0599687	40.072	0557318	44.310
Test2.bak	01000679	0599687	40.072	0557318	44.310
Test5.rtf	07706903	4726620	38.670	4634266	39.870
Game.exe	04387088	3660526	16.561	3647734	16.853
Test7.htm	06196553	4101214	33.815	4027486	35.000
Test8.pdf	13300157	10002995	24.790	9864775	25.830

TABLE 6.43: Comparison between Pure and Repeated Huffman (Level Order)

Level Order Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Repeated Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	0421356	65.176	0369058	69.500
Test3.doc	03997184	2151007	46.187	2137818	46.517
Test2.c	01000679	0599657	40.075	0557265	44.311
Test2.bak	01000679	0599657	40.075	0557265	44.311
Test5.rtf	07706903	4726590	38.671	4634271	39.870
Game.exe	04387088	3660591	16.560	3647806	16.851
Test7.htm	06196553	4101056	33.817	4028193	34.990
Test8.pdf	13300157	10002746	24.792	9864172	25.834

TABLE 6.44: Comparison between Pure and Repeated Huffman (Modified Level Order)

Modified Level Order Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Repeated Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	0421355	65.176	0369082	69.500
Test3.doc	03997184	2150997	46.187	2137803	46.517
Test2.c	01000679	0599657	40.075	0557265	44.312
Test2.bak	01000679	0599657	40.075	0557265	44.312
Test5.rtf	07706903	4726590	38.671	4634198	39.870
Game.exe	04387088	3660592	16.560	3647807	16.851
Test7.htm	06196553	4101095	33.817	4027080	35.010
Test8.pdf	13300157	10002746	24.792	9864172	25.834

TABLE 6.45: Comparison between Pure and Repeated Huffman (Preorder)

Preorder Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Repeated Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	00421356	65.176	0368750	69.524
Test3.doc	03997184	02151006	46.187	2137309	46.530
Test2.c	01000679	00599657	40.075	0556839	44.354
Test2.bak	01000679	00599657	40.075	0556839	44.354
Test5.rtf	07706903	04726590	38.671	4633887	39.874
Game.exe	04387088	03660526	16.561	3647734	16.853
Test7.htm	06196553	04101062	33.817	4026983	35.010
Test8.pdf	13300157	10002741	24.792	9863988	25.840

TABLE 6.46: Comparison between Pure and Repeated Huffman (Modified Preorder)

Modified Preorder Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Repeated Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	0421355	65.176	0368484	69.546
Test3.doc	03997184	2150997	46.187	2137331	46.529
Test2.c	01000679	0599657	40.075	0556922	44.346
Test2.bak	01000679	0599657	40.075	0556922	44.346
Test5.rtf	07706903	4726590	38.671	4633919	39.873
Game.exe	04387088	3660592	16.561	3647734	16.853
Test7.htm	06196553	4101095	33.817	4027001	35.010
Test8.pdf	13300157	10002746	24.792	9864006	25.840

TABLE 6.47: Comparison between Pure and Repeated Huffman (Balanced Binary Tree)

Balanced Binary Tree Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Repeated Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	00421386	65.173	0368910	69.511
Test3.doc	03997184	02151125	46.184	2137415	46.527
Test2.c	01000679	00599689	40.072	0557217	44.316
Test2.bak	01000679	00599689	40.072	0557217	44.316
Test5.rtf	07706903	04726622	38.670	4634046	39.872
Game.exe	04387088	03660648	16.560	3647891	16.850
Test7.htm	06196553	04101216	33.815	4027210	35.010
Test8.pdf	13300157	10002764	24.792	9864123	25.840

TABLE 6.48: Comparison between Pure and Block Huffman (Level Order)

Level Order Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Block Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	0421356	65.176	0432446	64.259361
Test3.doc	03997184	2151007	46.187	2170875	45.689891
Test2.c	01000679	0599657	40.075	0609746	39.066774
Test2.bak	01000679	0599657	40.075	0609746	39.066774
Game.exe	04387088	3660591	16.560	3686404	15.971505
Test7.htm	06196553	4101056	33.817	4184803	32.465630

TABLE 6.49: Comparison between Pure and Block Huffman (Modified Level Order)

Modified Level Order Technique					
File name	Original Size(Bytes)	Pure Huffman Coding		Block Huffman Coding	
		Compressed File Size (Bytes)	Compression Ratio (%)	Compressed File Size (Bytes)	Compression Ratio (%)
Test4.txt	01209956	0421355	65.176	0432400	64.263163
Test3.doc	03997184	2150997	46.187	2168747	45.743128
Test2.c	01000679	0599657	40.075	0609714	39.069971
Test2.bak	01000679	0599657	40.075	0609714	39.069971
Game.exe	04387088	3660592	16.560	3685784	15.985638
Test7.htm	06196553	4101095	33.817	4184583	32.469181

TABLE 6.50: A Huffman Tree Comparison between Existing and Proposed Method (contd.)

A Huffman Tree Size(Bytes)									
File Name	Original Size (Byte)	Existing Method (Circular Leaf node Tech.)				Proposed Method (Level Order Technique)			
		Pass One	Pass Two	Pass Three	Pass Four	Pass One	Pass Two	Pass Three	Pass Four
Test4.txt	01209956	143	195	500	N/A	115	323	322	N/A
Test3.doc	03997184	458	262	258	N/A	323	323	N/A	N/A
Test2.c	01000679	138	324	310	325	108	323	323	N/A
Test2.bak	01000679	138	324	310	325	108	323	323	N/A
Test5.rtf	07706903	133	413	263	N/A	103	322	323	N/A
Game.exe	04387088	258	315	N/A	N/A	323	322	N/A	N/A
Test7.htm	06196553	173	264	505	N/A	54	322	N/A	N/A
Test8.pdf	13300157	571	388	497	N/A	322	322	322	N/A

TABLE 6.51: A Huffman Tree Comparison between Existing and Proposed Method (contd.)

A Huffman Tree Size(Bytes)									
File Name	Original Size (Byte)	Existing Method (Circular Leaf node Tech.)				Proposed Method (Modified Level Order)			
		Pass One	Pass Two	Pass Three	Pass Four	Pass One	Pass Two	Pass Three	Pass Four
Test4.txt	01209956	143	195	500	N/A	114	323	322	N/A
Test3.doc	03997184	458	262	258	N/A	313	318	N/A	N/A
Test2.c	01000679	138	324	310	325	108	323	324	N/A
Test2.bak	01000679	138	324	310	325	108	323	324	N/A
Test5.rtf	07706903	133	413	263	N/A	103	322	322	N/A
Game.exe	04387088	258	315	N/A	N/A	324	322	N/A	N/A
Test7.htm	06196553	173	264	505	N/A	54	322	322	N/A
Test8.pdf	13300157	571	388	497	N/A	322	322	322	N/A

TABLE 6.52: A Huffman Tree Comparison between Existing and Proposed Method (contd.)

A Huffman Tree Size(Bytes)									
File Name	Original Size (Byte)	Existing Method (Circular Leaf node Tech.)				Proposed Method (Preorder Technique)			
		Pass One	Pass Two	Pass Three	Pass Four	Pass One	Pass Two	Pass Three	Pass Four
Test4.txt	01209956	143	195	500	N/A	115	68	315	96
Test3.doc	03997184	458	262	258	N/A	322	89	71	N/A
Test2.c	01000679	138	324	310	325	108	197	122	142
Test2.bak	01000679	138	324	310	325	108	197	122	142
Test5.rtf	07706903	133	413	263	N/A	103	279	75	N/A
Game.exe	04387088	258	315	N/A	N/A	258	315	N/A	N/A
Test7.htm	06196553	173	264	505	N/A	21	136	319	N/A
Test8.pdf	13300157	571	388	497	N/A	317	241	312	N/A

TABLE 6.53: A Huffman Tree Comparison between Existing and Proposed Method (contd.)

A Huffman Tree Size(Bytes)									
File Name	Original Size (Byte)	Existing Method (Circular Leaf node Tech.)				Proposed Method (Modified Preorder)			
		Pass One	Pass Two	Pass Three	Pass Four	Pass One	Pass Two	Pass Three	Pass Four
Test4.txt	01209956	143	195	500	N/A	113	68	67	72
Test3.doc	03997184	458	262	258	N/A	322	89	71	N/A
Test2.c	01000679	138	324	310	325	108	197	146	320
Test2.bak	01000679	138	324	310	325	108	197	146	320
Test5.rtf	07706903	133	413	263	N/A	103	279	77	N/A
Game.exe	04387088	258	315	N/A	N/A	258	315	N/A	N/A
Test7.htm	06196553	173	264	505	N/A	12	136	319	N/A
Test8.pdf	13300157	571	388	497	N/A	317	241	312	N/A

TABLE 6.54: A Huffman Tree Comparison between Existing and Proposed Method

A Huffman Tree Size(Bytes)									
File Name	Original Size (Byte)	Existing Method (Circular Leaf node Tech.)				Proposed Method (Balanced Binary Tree)			
		Pass One	Pass Two	Pass Three	Pass Four	Pass One	Pass Two	Pass Three	Pass Four
Test4.txt	01209956	143	195	500	N/A	145	177	308	N/A
Test3.doc	03997184	458	262	258	N/A	441	124	53	N/A
Test2.c	01000679	138	324	310	325	140	314	173	N/A
Test2.bak	01000679	138	324	310	325	140	314	173	N/A
Test5.rtf	07706903	133	413	263	N/A	135	395	59	N/A
Game.exe	04387088	258	315	N/A	N/A	380	348	N/A	N/A
Test7.htm	06196553	173	264	505	N/A	175	240	303	N/A
Test8.pdf	13300157	571	388	497	N/A	340	351	288	N/A

TABLE 6.55: Repetition Count Comparison among All Representation Methods

File name	Original Size(Bytes)	Repetition Count Repeated Huffman Coding Technique					
		Level order	Modified Level order	Preorder	Modified Preorder	Balanced Binary tree	Circular node Tech.
Test4.txt	01209956	3	3	4	4	3	3
Test3.doc	03997184	2	2	3	3	3	3
Test2.c	01000679	3	3	4	4	3	4
Test2.bak	01000679	3	3	4	4	3	4
Test5.rtf	07706903	3	3	3	3	3	3
Game.exe	04387088	2	2	2	2	2	2
Test7.htm	06196553	2	3	3	3	3	3
Test8.pdf	13300157	3	3	3	3	3	3

TABLE 6.56: Standard Deviation vs. Compression Ratio (Contd.)

Method	File Name	Original Size(Bytes)	Standard Deviation vs. Compression Ratio				
			Pass1	Pass2	Pass3	Pass4	Repeated Huffman Compression Ratio (%)
(Existing Method) Circular node Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.492
	Test3.doc	03997184	6.110	0.492	0.153	N/A	46.517
	Test2.c	01000679	4.506	1.738	0.445	0.217	44.310
	Test2.bak	01000679	4.506	1.738	0.445	0.217	44.310
	Test5.rtf	07706903	5.989	0.983	0.153	N/A	39.870
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.853
	Test7.htm	06196553	6.862	0.942	0.188	N/A	35.000
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.830
(Proposed Method) Level Order Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.500
	Test3.doc	03997184	6.110	0.492	N/A	N/A	46.517
	Test2.c	01000679	4.506	1.815	0.471	N/A	44.311
	Test2.bak	01000679	4.506	1.815	0.471	N/A	44.311
	Test5.rtf	07706903	5.989	0.983	0.108	N/A	39.870
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.851
	Test7.htm	06196553	6.862	0.942	N/A	N/A	34.990
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.834

TABLE 6.57: Standard Deviation vs. Compression Ratio(Contd.)

Method	File Name	Original Size(Bytes)	Standard Deviation vs. Compression Ratio				
			Pass1	Pass2	Pass3	Pass4	Repeated Huffman Compression Ratio (%)
(Existing Method) Circular node Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.492
	Test3.doc	03997184	6.110	0.492	0.153	N/A	46.517
	Test2.c	01000679	4.506	1.738	0.445	0.217	44.310
	Test2.bak	01000679	4.506	1.738	0.445	0.217	44.310
	Test5.rtf	07706903	5.989	0.983	0.153	N/A	39.870
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.853
	Test7.htm	06196553	6.862	0.942	0.188	N/A	35.000
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.830
(Proposed Method) Modified Level Order Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.500
	Test3.doc	03997184	6.110	0.492	N/A	N/A	46.517
	Test2.c	01000679	4.506	1.815	0.471	N/A	44.312
	Test2.bak	01000679	4.506	1.815	0.471	N/A	44.312
	Test5.rtf	07706903	5.989	0.983	0.108	N/A	39.870
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.851
	Test7.htm	06196553	6.862	0.942	0.188	N/A	35.010
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.834

TABLE 6.58: Standard Deviation vs. Compression Ratio (Contd.)

Method	File Name	Original Size(Bytes)	Standard Deviation vs. Compression Ratio				
			Pass1	Pass2	Pass3	Pass4	Repeated Huffman Compression Ratio (%)
(Existing Method) Circular node Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.492
	Test3.doc	03997184	6.110	0.492	0.153	N/A	46.517
	Test2.c	01000679	4.506	1.738	0.445	0.217	44.310
	Test2.bak	01000679	4.506	1.738	0.445	0.217	44.310
	Test5.rtf	07706903	5.989	0.983	0.153	N/A	39.870
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.853
	Test7.htm	06196553	6.862	0.942	0.188	N/A	35.000
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.830
(Proposed Method) Preorder Tech.	Test4.txt	01209956	8.228	1.818	0.473	0.188	69.524
	Test3.doc	03997184	6.110	0.492	0.153	N/A	46.530
	Test2.c	01000679	4.506	1.815	0.458	0.153	44.354
	Test2.bak	01000679	4.506	1.815	0.458	0.153	44.354
	Test5.rtf	07706903	5.989	0.983	0.153	N/A	39.874
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.853
	Test7.htm	06196553	6.862	0.942	0.153	N/A	35.010
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.840

TABLE 6.59: Standard Deviation vs. Compression Ratio (Contd.)

Method	File Name	Original Size(Bytes)	Standard Deviation vs. Compression Ratio				
			Pass1	Pass2	Pass3	Pass4	Repeated Huffman Compression Ratio (%)
(Existing Method) Circular node Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.492
	Test3.doc	03997184	6.110	0.492	0.153	N/A	46.517
	Test2.c	01000679	4.506	1.738	0.445	0.217	44.310
	Test2.bak	01000679	4.506	1.738	0.445	0.217	44.310
	Test5.rtf	07706903	5.989	0.983	0.153	N/A	39.870
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.853
	Test7.htm	06196553	6.862	0.942	0.188	N/A	35.000
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.830
(Proposed Method) Modified Preorder Tech.	Test4.txt	01209956	8.228	1.818	0.334	0.217	69.546
	Test3.doc	03997184	6.110	0.492	0.153	N/A	46.529
	Test2.c	01000679	4.506	1.815	0.440	0.109	44.346
	Test2.bak	01000679	4.506	1.815	0.440	0.109	44.346
	Test5.rtf	07706903	5.989	0.983	0.108	N/A	39.873
	Game.exe	04387088	2.707	0.347	N/A	N/A	16.853
	Test7.htm	06196553	6.862	0.942	0.187	N/A	35.010
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.840

TABLE 6.60: Standard Deviation vs. Compression Ratio

Method	File Name	Original Size(Bytes)	Standard Deviation vs. Compression Ratio				
			Pass1	Pass2	Pass3	Pass4	Repeated Huffman Compression Ratio (%)
(Existing Method) Circular node Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.492
	Test3.doc	03997184	6.110	0.492	0.153	N/A	46.517
	Test2.c	01000679	4.506	1.738	0.445	0.217	44.310
	Test2.bak	01000679	4.506	1.738	0.445	0.217	44.310
	Test5.rtf	07706903	5.989	0.983	0.153	N/A	39.870
	Game.exe	04387088	2.705	0.347	N/A	N/A	16.853
	Test7.htm	06196553	6.862	0.942	0.188	N/A	35.000
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.830
(Proposed Method) Balanced Binary Tree Tech.	Test4.txt	01209956	8.228	1.818	0.473	N/A	69.511
	Test3.doc	03997184	6.110	0.492	0.188	N/A	46.527
	Test2.c	01000679	4.506	1.738	0.458	N/A	44.316
	Test2.bak	01000679	4.506	1.738	0.458	N/A	44.316
	Test5.rtf	07706903	5.989	0.983	0.153	N/A	39.872
	Game.exe	04387088	2.707	0.347	N/A	N/A	16.850
	Test7.htm	06196553	6.862	0.942	0.188	N/A	35.010
	Test8.pdf	13300157	3.657	0.766	0.188	N/A	25.840

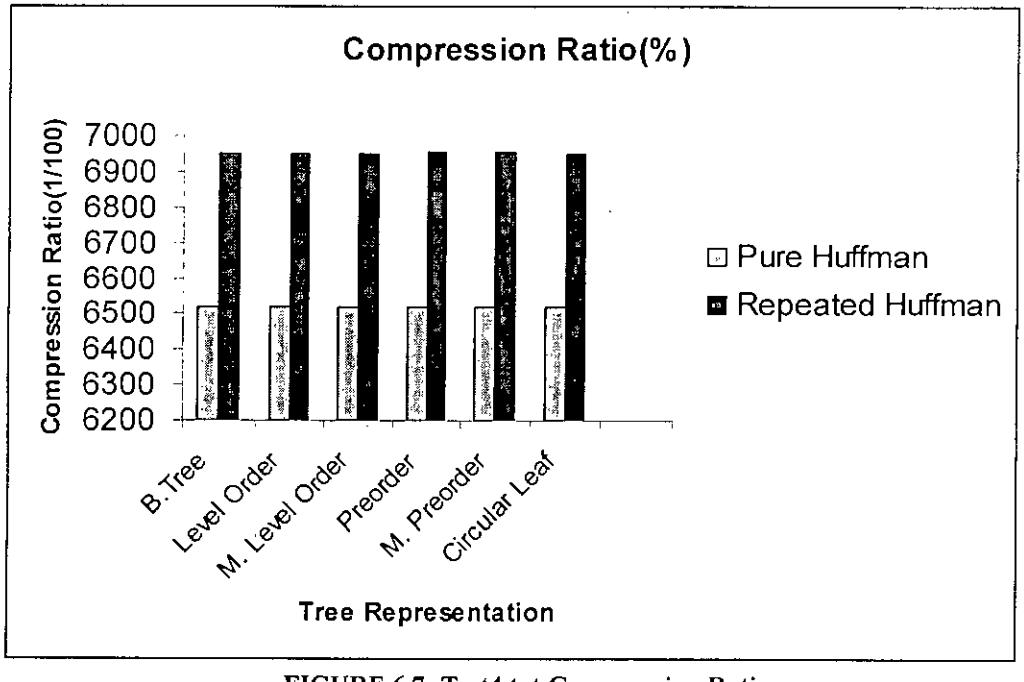


FIGURE 6.7: Test4.txt Compression Ratios

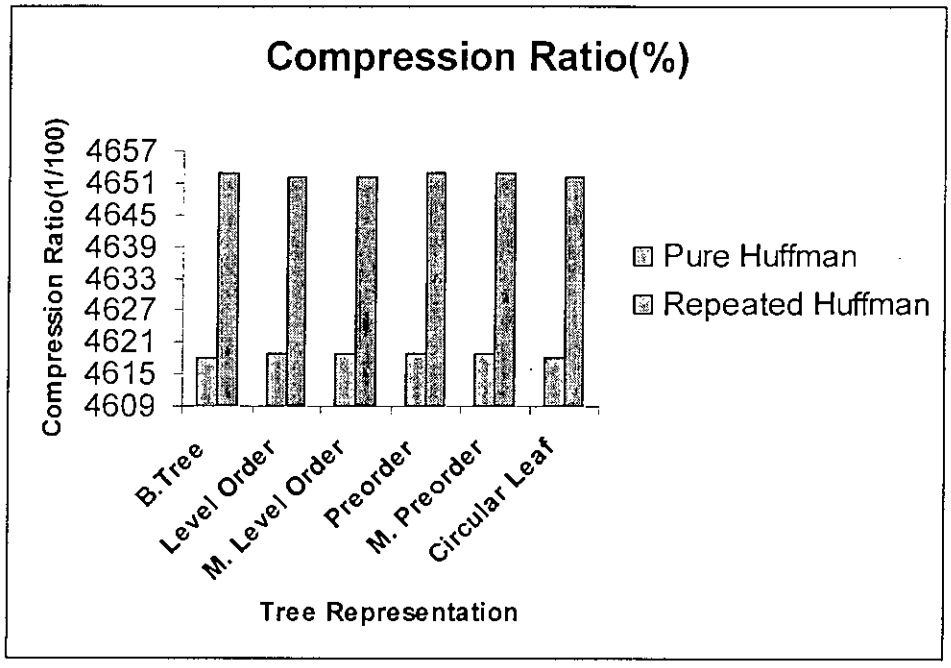


FIGURE 6.8: Test3.doc Compression Ratios

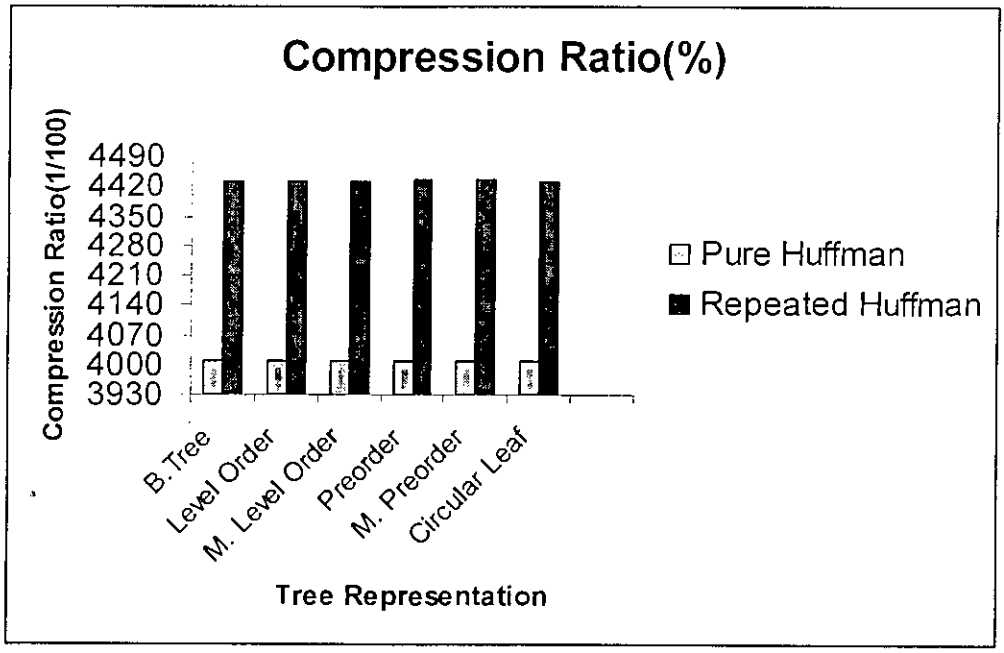


FIGURE 6.9: Test2.c Compression Ratios

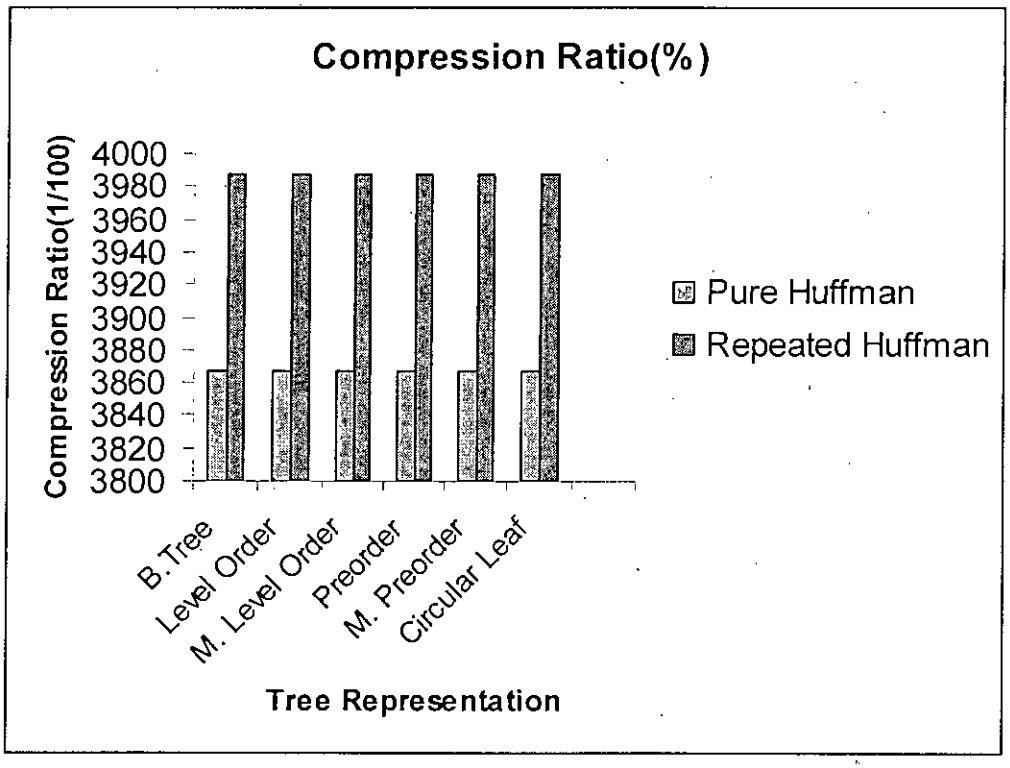


FIGURE 6.10: Test5.rtf Compression Ratios

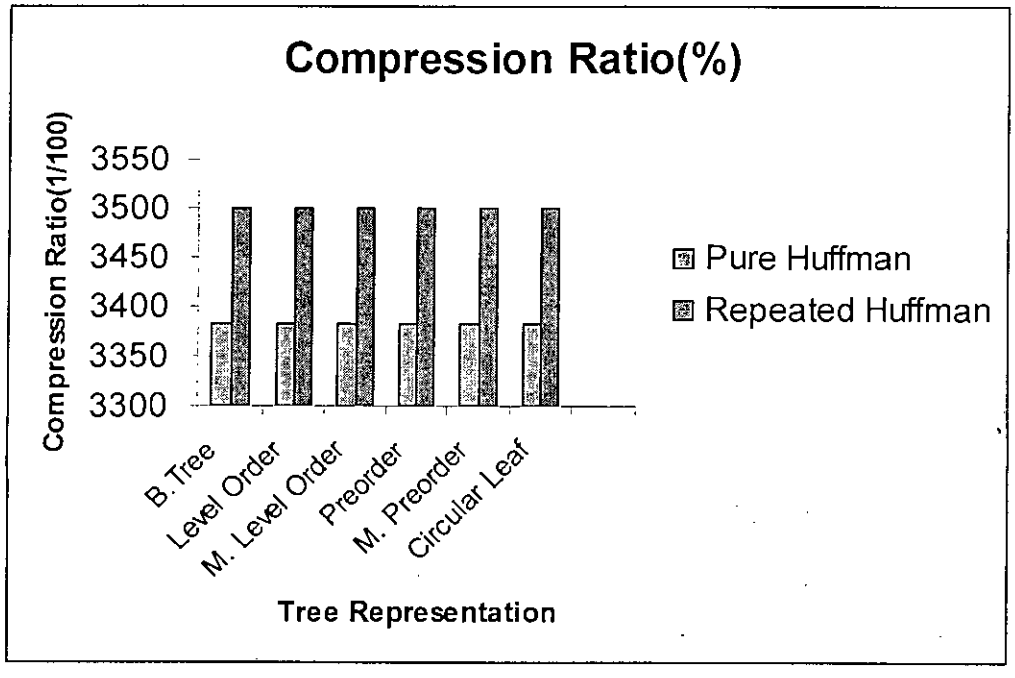


FIGURE 6.11: Test7.htm Compression Ratios

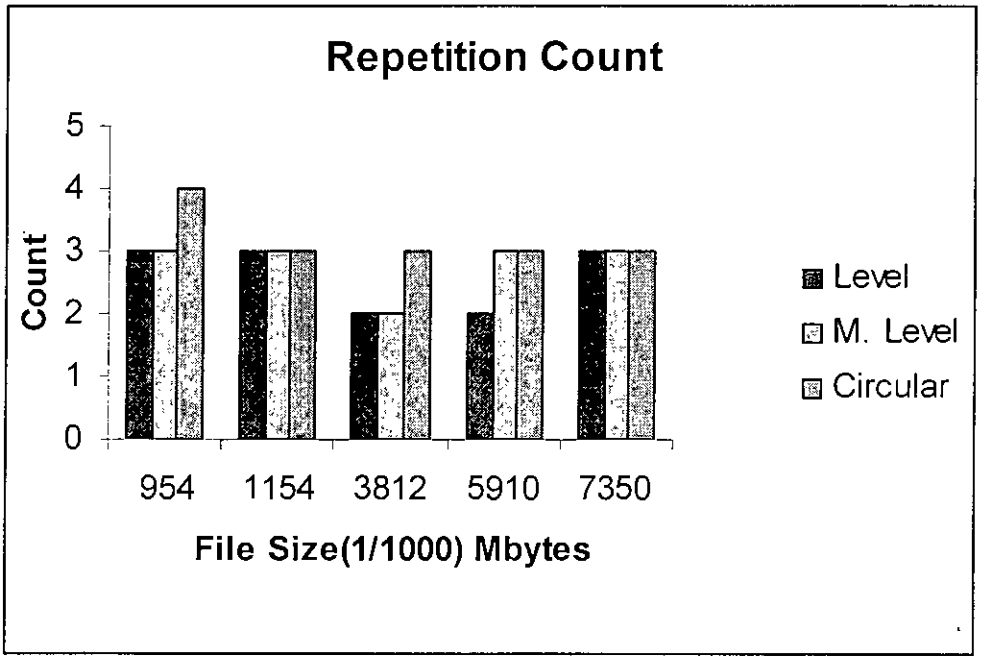


FIGURE 6.12: Repetition Count for C, txt, doc, htm and rtf files (Contd.)

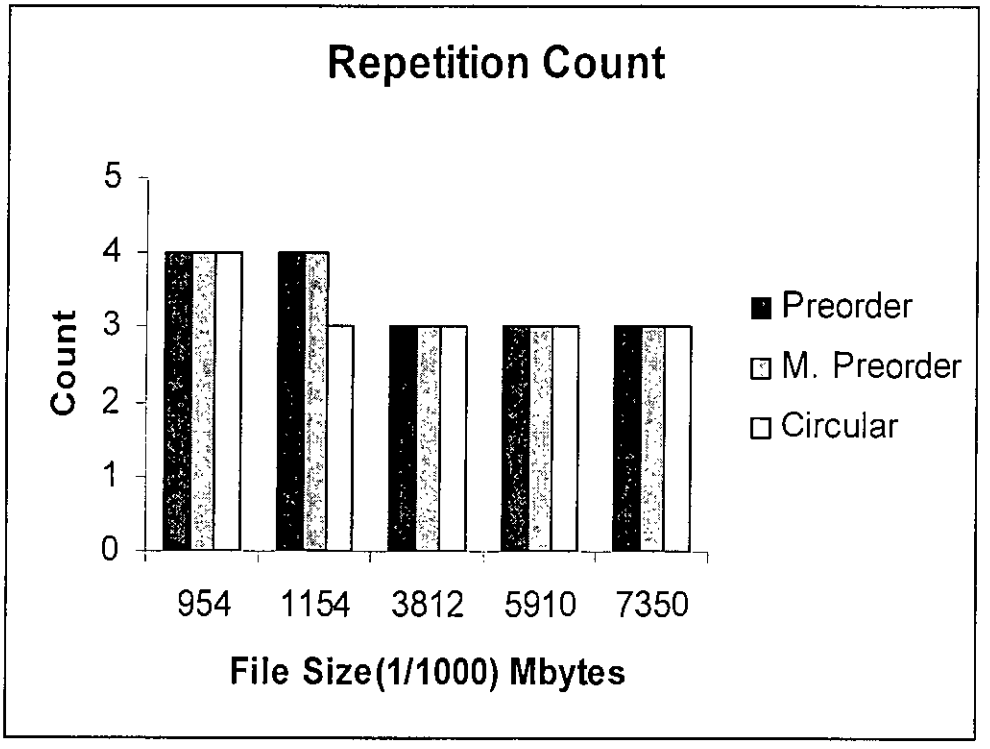


FIGURE 6.13: Repetition Count for C, txt, doc, htm and rtf files (Contd.)

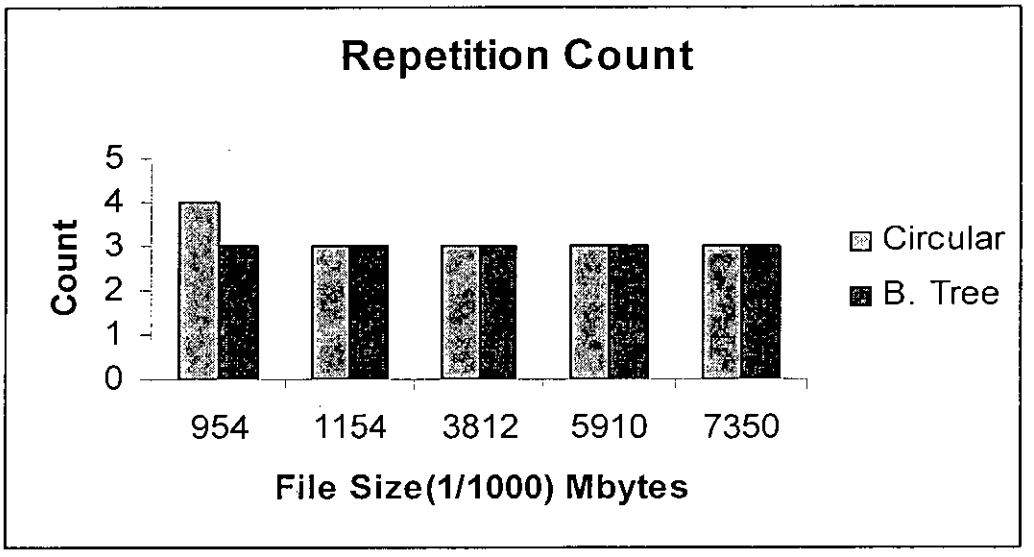


FIGURE 6.14: Repetition Count for C, txt, doc, htm and rtf files

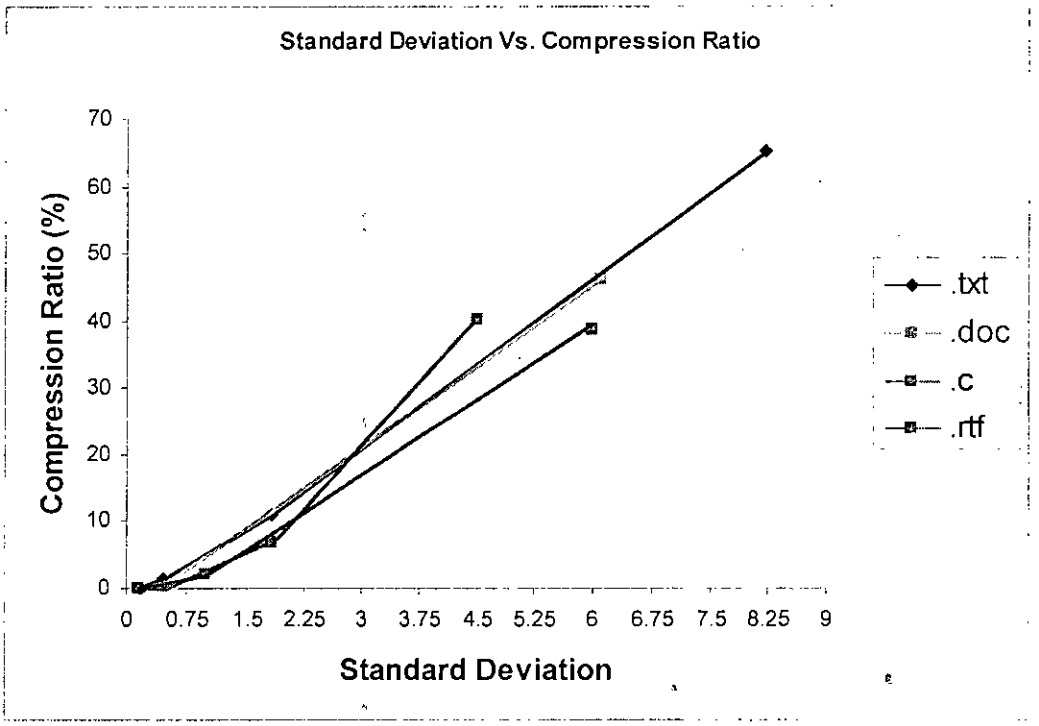


FIGURE 6.15: Change of Compression Ratio (.txt, .doc, .C, .rtf) in preorder technique

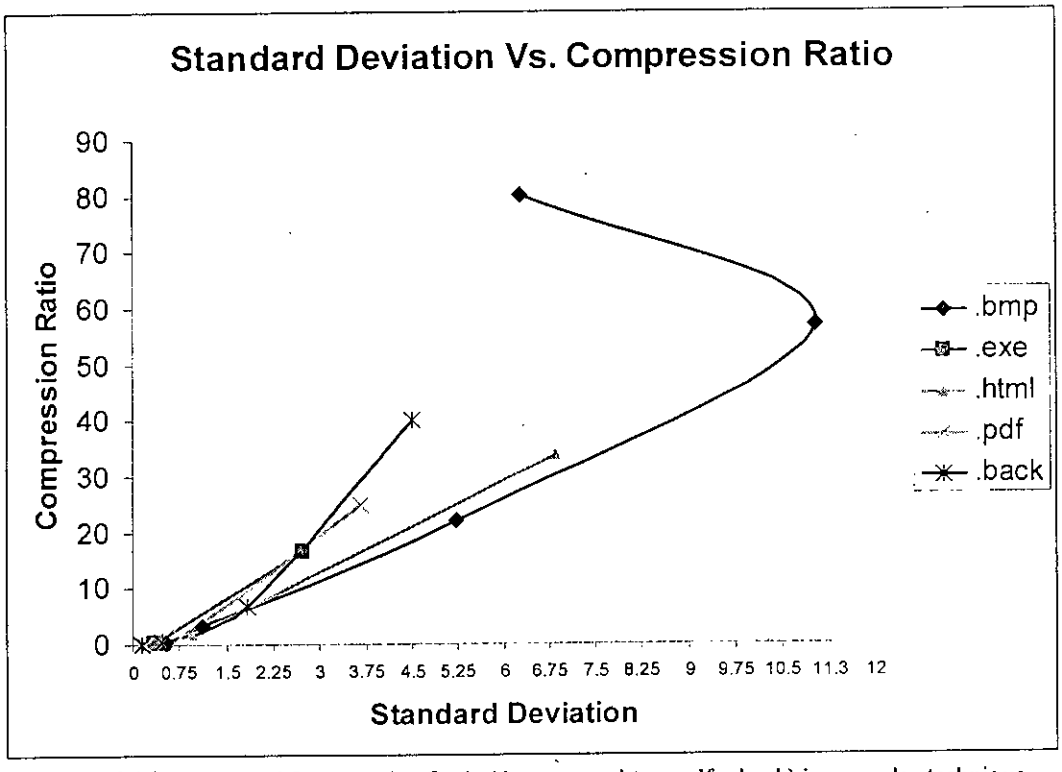


FIGURE 6.16: Change of Compression Ratio (.bmp, .exe, .htm, .pdf,, .back) in preorder technique

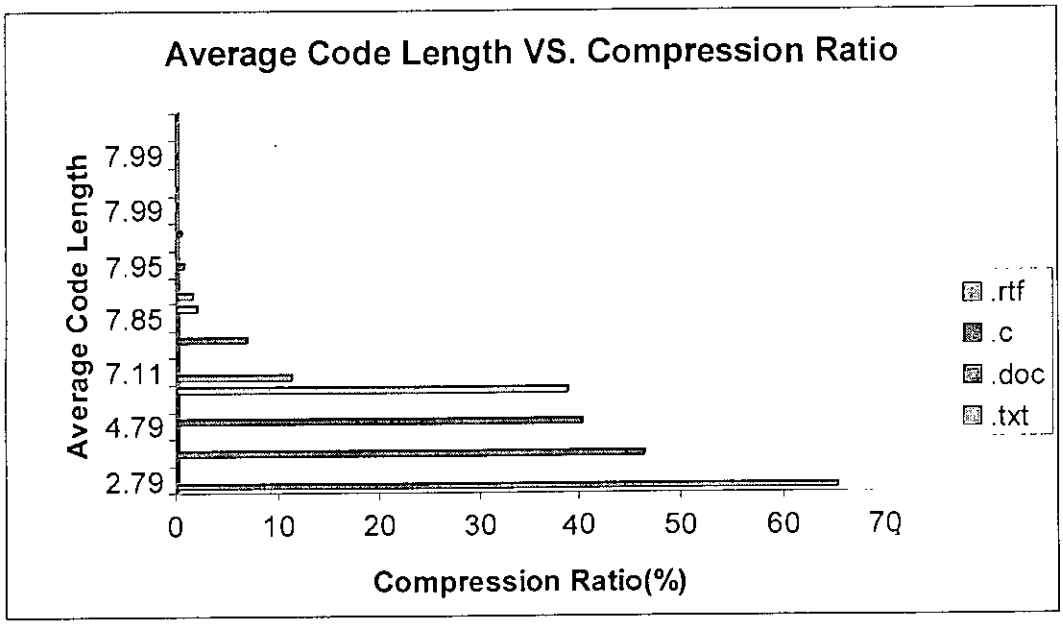


FIGURE 6.17: Change of Compression Ratio (.txt, .doc, .C, .rtf) in preorder technique

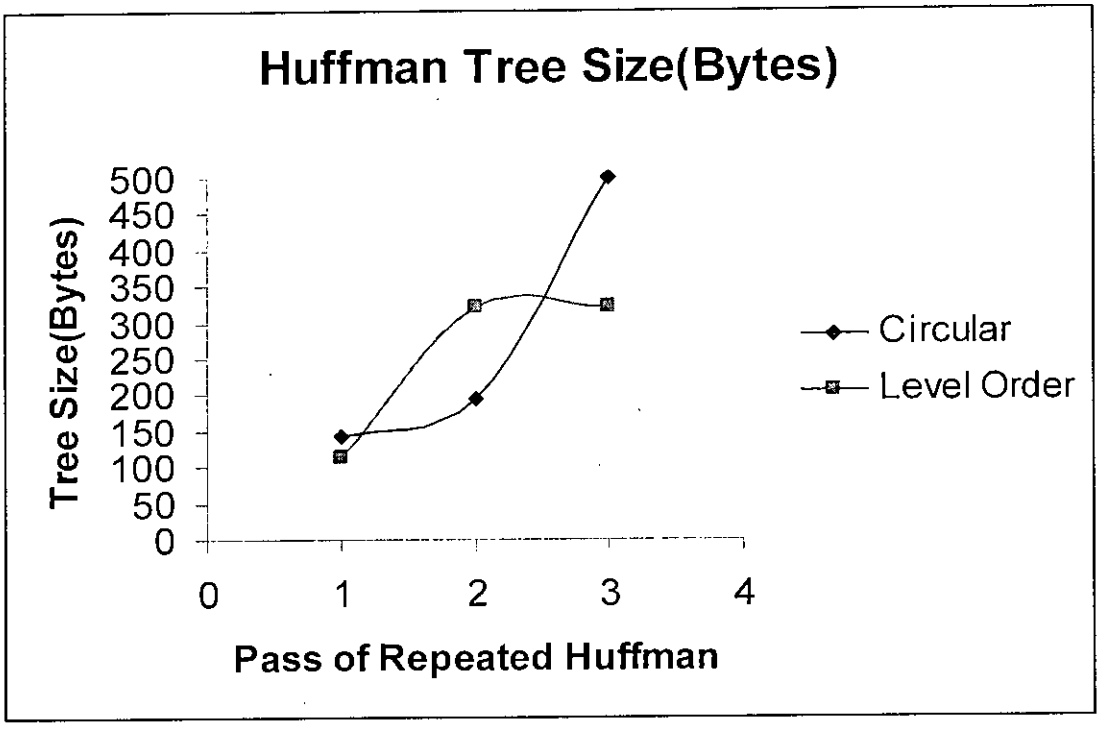


FIGURE 6.18: Huffman tree Size in Circular Leaf node and Level order (Test4.txt)

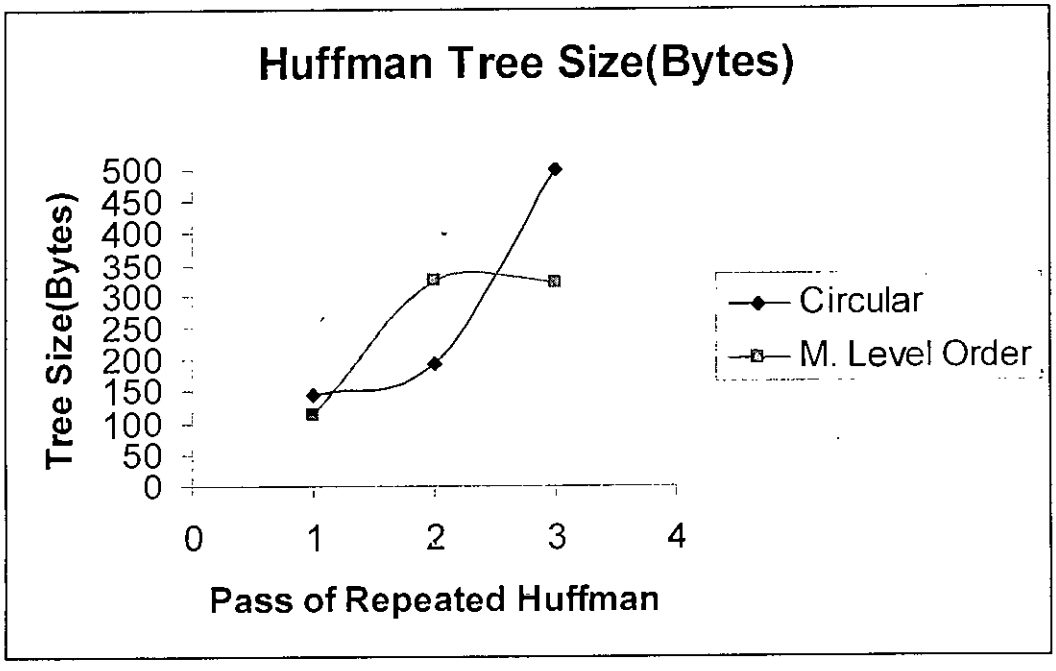


FIGURE 6.19: Huffman tree Size in Circular Leaf node and Modified Level order(Test4.txt)

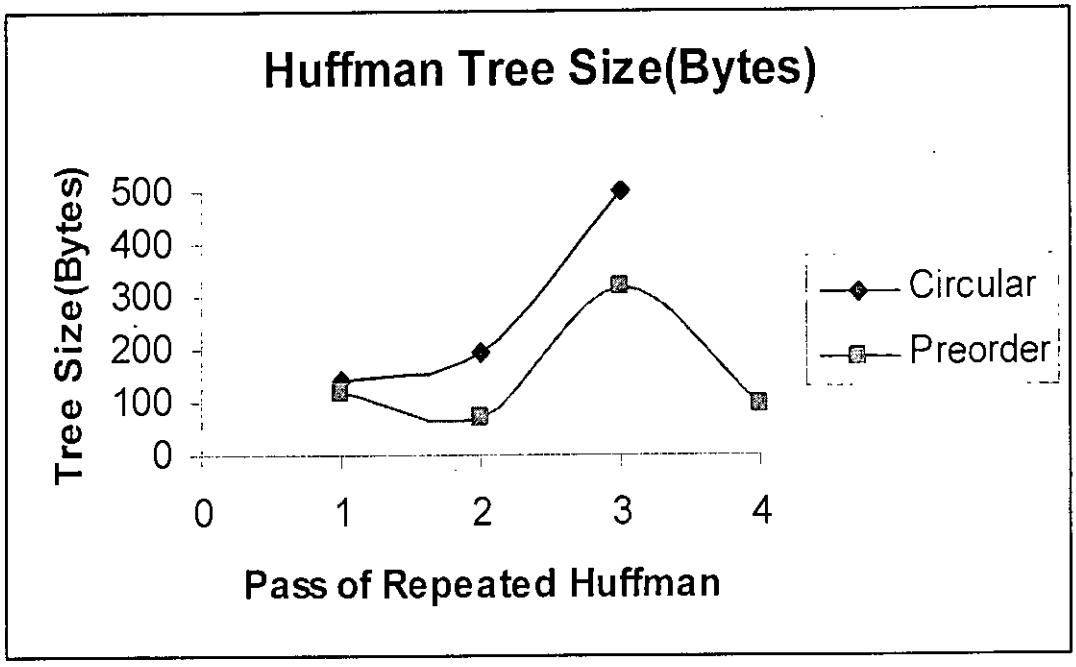


FIGURE 6.20: Huffman tree Size in Circular Leaf node and Preorder(Test4.txt)

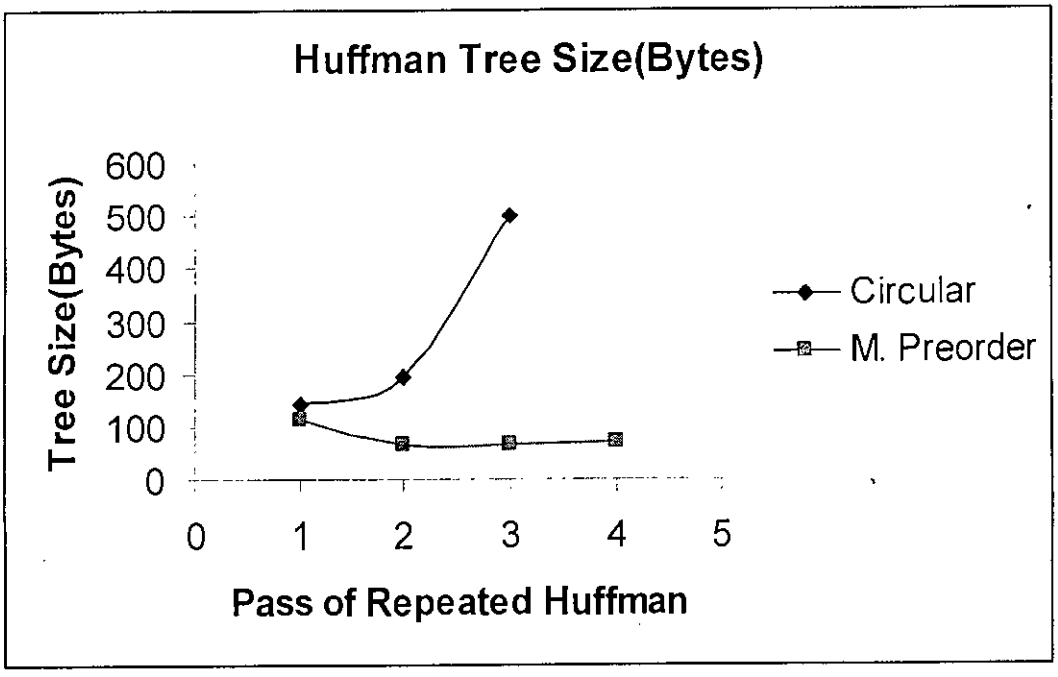


FIGURE 6.21: Huffman tree Size in Circular Leaf node and Modified Preorder (Test4.txt)

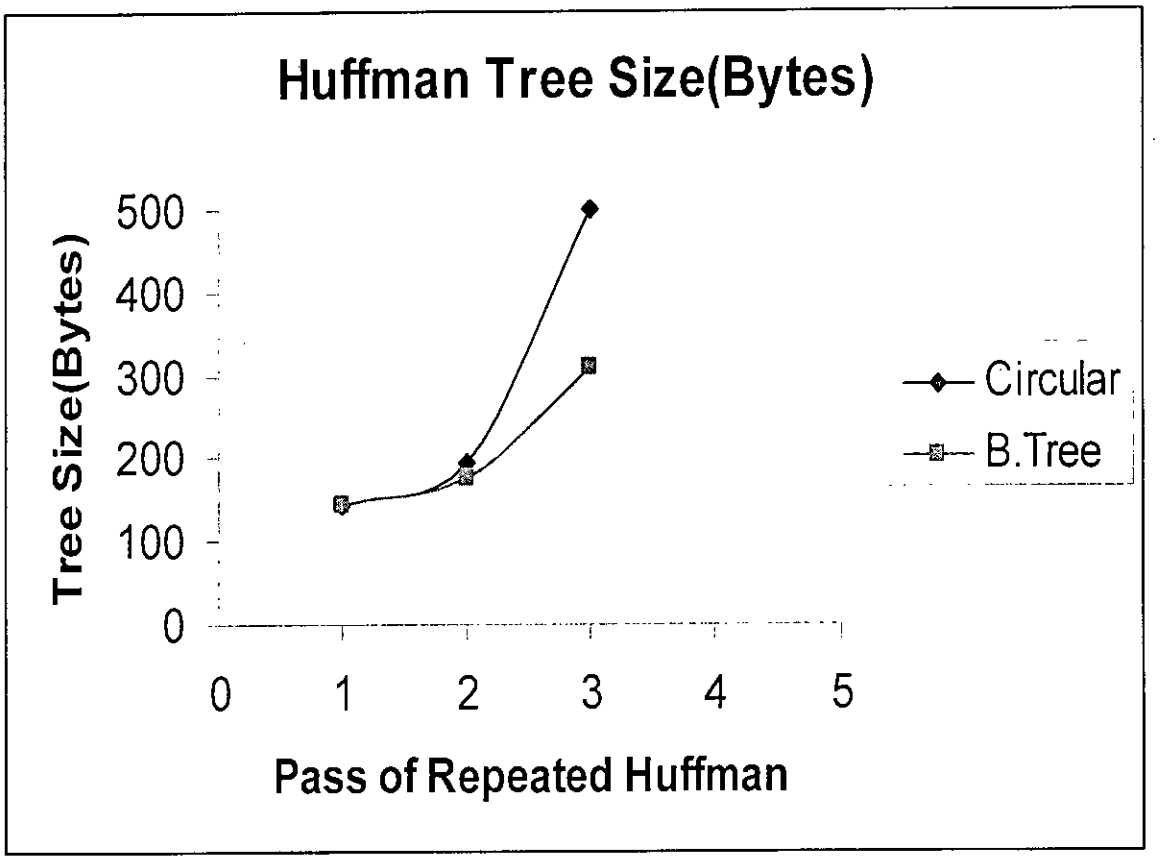


FIGURE 6.22: Huffman tree Size in Circular Leaf node and Balanced B. Tree (Test4.txt)

6.5 Analysis of Experimental Results

Table 6.40 presents compression ratios in existing (circular leaf node) and proposed methods of Huffman tree representation. Compression ratios for preorder and its modified version are higher than all other remaining methods. Table 6.42-6.47 compares Pure and Repeated Huffman with respect to compression ratios. Repeated Huffman shows better performance. Table 6.50-6.54 shows reduced form of Huffman tree compared to existing methods. From Table 6.55 it may be mentioned that Preorder and its modified version also increase repetition count because of memory efficient representation of a Huffman tree that is necessary for Repeated Huffman coding. Table 6.56-6.60 shows change of compression ratio with standard deviation. If change of standard deviation is slowed down, repetition count is increased. Abrupt fall of standard deviation makes Repeated Huffman coding to be degenerated.

Figure 6.7-6.11 compares compression ratios using chart. These figures consider TXT, DOC, C, RTF and HTM files for existing and proposed methods of Huffman tree representation. Figure 6.12-6.14 visualizes the repetition count for each of the proposed techniques compared to existing technique. Change of compression ratio with standard deviation is shown in Figure 6.15-6.16 for different files using preorder technique. Figure 6.17 shows change of compression ratio with respect to average code length. A Huffman tree size in every pass of Repeated Huffman in existing and proposed methods is also compared in figure 6.18-6.22. In almost all of the cases proposed methods show memory efficient tree representation.

6.6 **Conclusion**

Experimental results show that the proposed methods provide a memory-efficient representation of a Huffman tree over existing methods. Because of this efficient representation overall compression ratio is increased.

CHAPTER 7

CONCLUSION AND RECOMMENDATIONS

7.1 Conclusion

In this thesis different techniques of Huffman tree representation have been studied and implemented. Clustering of a Huffman tree is also discussed. It also concentrates on Block Huffman coding. This thesis has mainly focused on how efficiency of Huffman coding technique can be improved through the Repeated Huffman coding. Optimal cluster length regarding tree clustering is also studied. Block Huffman coding improves the performance of Pure Huffman coding with respect to speed of compression.

7.1.1 Huffman Tree Representation Techniques

Level order technique

Theoretically level order technique requires less than $\lceil 5n/4 \rceil$ memory spaces at any cases to represent a Huffman tree which is less than existing method (Circular node) that requires at most $\lceil 3n/2 \rceil$ memory spaces for the Huffman tree of n distinct symbols. Experimental results show compression ratio and tree size for the level order technique. Especially for the single side growing Huffman tree, circular node technique shows better performance than level order technique. For the fully balanced Huffman tree the performance of level order technique shows excellent performance.

Modified Level Order Technique

Modified level order technique requires less than $\lceil 9n/8 \rceil$ memory spaces at best case. It needs less than $\lceil 5n/4 \rceil$ memory spaces at worst case to represent a Huffman tree, which is less than existing method (Circular node) that requires at most $\lceil 3n/2 \rceil$ memory spaces for the Huffman tree of n distinct symbols. Experimental results show compression ratio and tree size for the Modified level order technique. Especially for the single side growing Huffman tree, circular node technique shows better performance than Modified level order technique. For the fully balanced Huffman tree the performance of Modified level order technique shows excellent performance because only internal node representation is required.

Preorder Technique

Theoretically Preorder technique requires less than $\lceil 5n/4 \rceil$ memory spaces at any cases to represent a Huffman tree which is less than existing method (Circular node) that requires at most $\lceil 3n/2 \rceil$ memory spaces for the Huffman tree of n distinct symbols. Experimental results show compression ratio and tree size for the Preorder technique. Especially for the single side growing Huffman tree, circular node technique requires n+1 memory spaces. For the fully balanced Huffman tree the performance of Preorder technique shows excellent performance than existing method.

Modified Preorder Technique

Modified Preorder technique requires less than $\lceil 9n/8 \rceil$ memory spaces at best case. It needs less than $\lceil 5n/4 \rceil$ memory spaces at worst case to represent a Huffman tree, which is less than existing method (Circular node) that requires at most $\lceil 3n/2 \rceil$ memory spaces for the Huffman tree of n distinct symbols. Experimental results show compression ratio and tree size for the Modified Preorder technique. Especially for the left eccentric Huffman tree, Circular Leaf node and Modified Preorder techniques show same performance. For the fully balanced Huffman tree the performance of Modified Preorder technique shows excellent performance than existing technique.

Balanced Binary Tree Technique

Balanced binary tree technique requires n+1 memory spaces at best case. It needs $\lfloor 3n/2 \rfloor$ memory spaces at worst case to represent a Huffman tree which is less than existing method (Circular Leaf node) that requires at most $\lceil 3n/2 \rceil$ memory spaces for the Huffman tree of n distinct symbols. Experimental results show

compression ratio and tree size for the Modified Preorder technique. Especially for the eccentric or the fully Balanced Huffman tree, Balanced Binary tree technique shows better performance. For the fully Balanced Binary tree existing technique (Circular Leaf node) provides worse performance.

7.1.2 **Tree Clustering Technique**

Clustering of a Huffman tree is required to reduce the sparsity of a Huffman tree. It also reduces the wastage of memory and makes search process faster for a symbol. Since search time is proportional to $1+C_L/L$, so increasing value of L (top cluster length) minimizes search time but memory wastage is increased. Therefore a decision regarding L is needed. Length of top cluster can be selected as close to average code length so that in first chance most of the symbols can be found in look-up table of top cluster and wastage of memory can be kept minimal.

7.1.3 **Block Huffman Coding**

Block Huffman coding is used to speed up Pure Huffman coding in the case of large file. Block Huffman coding cannot improve the compression ratio significantly. Because of locality characteristics this method sometimes provides better compression than traditional one. Block size must be selected such a way that it does not hamper the compression ratio significantly.

7.2 **Recommendations**

In this thesis work lossless, variable length and repeated version of static Huffman coding is studied, implemented and tested for each method of Huffman tree representation. Future works on Huffman tree representation and tree clustering algorithm can be carried on

- An optimal representation technique for a Huffman tree of n distinct symbols to get more compression ratio for the Block Huffman coding using Repeated Huffman coding technique.

- An efficient decoding technique for the Repeated Huffman Coding. This decoding technique will extract the original information quickly. Less complex data structure for this technique is required.

- An effective clustering for the Huffman tree is also required. This clustering technique makes memory mapping faster, speed up search process and reduces memory wastage.

- Repeated Huffman Coding Scheme for the Bangla text Compression-Decompression to reduce the redundancy of text. Coding and decoding time can be studied.

- Repeated Huffman Coding Scheme for image, video and audio data so that compression ratio will be higher. Embedded the Repeated Huffman Coding with Arithmetic coding to get better compression ratio.

- Huffman code optimization by biasing a Huffman tree. Reduce the external path length for each symbol. Optimized code for each symbol will provide better compression ratio.

REFERRENCES

[1] Abu-Mostafa, Y.S. and McEliece, R. J., "Maximal Codeword Lengths in Huffman Codes", Computers and Mathematics with Applications 39(2000)129-134.

[2] Apiki, S., "Lossless Data Compression", BYTE (March 1991), 309-314, 386-387

[3] Chen, H. –C., Wang, Y. –L. and Lan, Y. –F., "A memory -efficient and fast Huffman decoding algorithm", Inform. Process. Lett. 69(1999)119-122.

[4] Chowdhury, Rezaul Alam, Kaykobad, M. and King, Irwin, "An efficient decoding technique for Huffman codes", Inform. Process. Lett. 81(2002)305-308.

[5] Chung, K. L., "Efficient Huffman decoding", Inform. Process. Lett. 61(1997)9799.

[6] Connel, J. B., "A Huffman-Shannon-Fano Code", Proc. IEEE 61 (Jul. 1973), 1046-1047.

[7] Elabdalla, Abel-Rahman and I. Irshid, Mansour, "An efficient bitwise Huffman Coding technique based on source mapping", Computer and Electrical Engineering 27(2001)265-272.

[8] Faller, N., "An adaptive system for data compression", in record of the 7th Asilomar Conference on Circuits, Systems, and Computers. 1973, pp. 593-597.

[9] Fano, R. M., "Transmission of Information", Cambridge, MA; MIT Press and New York; Yiley, 1961.

[10] Gallager, R. G., "Variations on a theme by Huffman", IEEE Trans. Inf. Theory IT-24, 6(Nov. 1978), 668-674.

[11] Hashemian, R., "Memory efficient and high-speed search Huffman coding", IEEE Trans. Comm. 43(10)(1995)2576-2581.

[12] Howard, P.G. and Vitter, J.S., "Analysis of Arithmetic Coding for Data Compression", in Proc. Data Compression Conference, J. A. Storer and J. H. Reif, eds., Snowbird, Utah, Apr. 8-11, 1991, 3-12, invited paper, also to appear as an in the special issue of Information Processing and management, also appears as Brown University Technical Report No. CS-91-03.

[13] Howard, P.G. and Vitter, J. S., "Parallel lossless image compression using Huffman and arithmetic coding", Inform. Process. Lett. 59(1996)65-73.

[14] Howard, P.G. and Vitter, J. S., "Design and analysis of fast text compression based on quasi-arithmetic coding, Inform. Process. Management 30(6)(1994)777-794.

[15] Hu, Yu-Chen and Chang, Chin-Chen, "A new lossless compression scheme based on Huffman coding scheme for image compression", Signal Processing: Image Communication 16(2000)367-372.

[16] Huffman, D. A, "A method for construction of minimum redundancy codes, Proc. IRE 40(1952) 1098-1101.

[17] Humayun, S. M., Rahman, S. H. and Kaykobad M., "Static Huffman code for Bangla Text", 15th Annual Conference of BAAS, Section III, AERE, Savar, March 5-8, 1990

[18] Katona, G. O. H. and Nemetz, T. O. H., "Huffman codes and self-information", IEEE Trans. Inform. Theory IT-22, 337-340 (May 1976)

[19] Knuth, D. E., "Dynamic Huffman Coding", Journal of Algorithms 6, 163-180(1985).

[20] Langdon, G.G., "An Introduction to Arithmetic Coding", IBM J. Res. Develop. 28, 2(Mar. 1984), 135-149.

[21] Longo, G. and Galasso, G., "An Application of Informational Divergence to Huffman Codes", IEEE Trans. Inform. Theory, IT-28(1) (Jan. 1982), 36-43.

[22] Mandal, J. N., "An approach towards development of efficient data compression algorithms and correction techniques", Ph. D. Thesis, Jadavpur University, India, 2000.

[23] Mannan, Mohammad Abdul and Kaykobad, M., "Block Huffman Coding", Computers and Mathematics with Applications 0(2003)1-0.

[24] Moffat, A., "Word-Based Text Compression", Software-Practice and Experience, 19 (Feb. 1989), 185-198.

[25] Parker, D. S., "Conditions for Optimality of the Huffman Algorithm", SIAMJ. Computer 9(3), (Aug. 1980), 470-489.

[26] Rissanen, J. J. and Mohiuddin, K. M., "A multiplication-free multialphabet arithmetic code, IEEE Trans. Comm. 37(2)(1989)93-98

[27] Schack, R., "The length of a typical Huffman codeword", IEEE Trans. Inform. Theory 40(4)(1994)1246-1247.

[28] Shannon, C. E., "A Mathematical Theory of Communication", Bell Syst. Tech. J. 27 (July 1948), 398-403.

[29] Shannon, C. E., "Prediction of Entropy of Printed English", Bell Syst. Tech. J. 30 (1951), 50-64.

[30] Tanaka, H., "Data Structure of Huffman Codes and Its Application to Efficient Encoding and Decoding", IEEE Trans. Inform. Theory IT-33 (Jan. 1987), 154-156.

[31] Turpin, Andrew and Moffat, Alistair, Comment on "Efficient Huffman decoding" and "An efficient finite-state machine implementation of Huffman decoders", Inform. Process. Lett. 68(1998)1-2.

[32] Voorhis, D.C. Van, "Constructing codes with bounded codeword lengths, IEEE Trans. Inform. Theory. 20(2) (1974)288-290.

[33] Vitter, J. S., "Design and analysis of Dynamic Huffman Codes", J. ACM 34(4)(1987)825-845.

[34] Welch, T., "A Technique for High Performance Data Compression", IEEE Computer, 17(6), (1984), 8-19.

[35] Wells, M., "File Compression Using Variable Length Encoding", Computer Journal, 15, 4(1973), 308-813.

[36] Yannakoudakis, E. J., Goyal, P. and Huggill, J. A. , "The Generation and Use of Text Fragments for Data Compression", Information Processing and Management, 18, 1(1982), 15-21.

[37] Yeung, R. W., "Local Redundancy and Progressive Bounds on the Redundancy of a Huffman Code", IEEE Trans. Inform. Theory, IT-37, 3(May 1991), 687-690.

[38] Zimmermann, S., "An Optimal Search Procedure", Amer. Math. Monthly, 66, (1959), 690-693.

[39] Ziv, J. and Lempel, A., "Compression of Individual Sequences via Variable-Rate Coding", IEEE Trans. Inform. Theory. IT-24, 5(Sep 1978), 530-536.

[40] Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression", IEEE Trans. Inform. Theory. IT-23, 3(May 1977), 337-343.