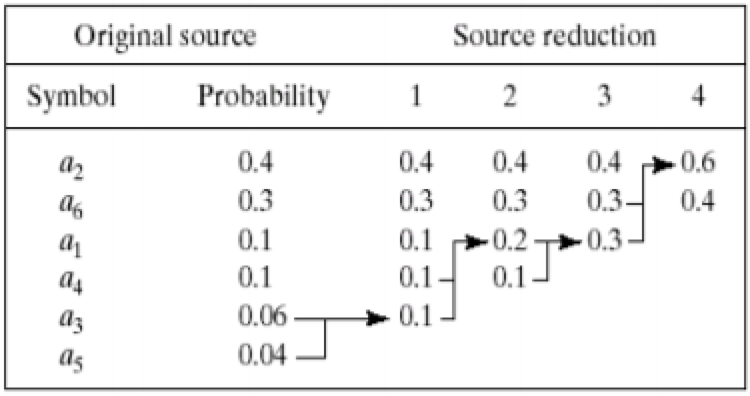
Huffman Coding and Decoding

Aniketh Sukhtankar (UF ID: 7819 9584) Dept. of CISE Email: asukhtankar@ufl.edu

1. Introduction

Huffman coding is one of the most popular technique for removing coding redundancy. It has been used in various compression applications, including image compression. It is a simple, yet elegant, compression technique that can supplement other compression algorithms. It utilizes the statistical property of symbols in the source stream, and then produces respective codes for these symbols. These codes are of variable length using integral number of bits. In Huffman coding, you assign shorter codes to symbols that occur more frequently and longer codes to those that occur less frequently [1].



Original source		Source reduction			
Symbol	Probability	1	2	3	4
a_2	0.4	0.4	0.4	0.4	0.6
a_6	0.3	0.3	0.3	0.3	0.4
a_1	0.1	0.1	0.2	0.3	
a_4	0.1	0.1	0.1		
a_3	0.06	0.1			
a_5	0.04				

Fig. 1 Huffman Source Reductions

1.1 Building a Huffman Tree

The compression process is based on building a binary tree that holds all symbols in the source at its leaf nodes. The code word for each symbol is obtained traversing the binary tree from its root to the leaf corresponding to the symbol. Symbols with the highest frequencies end up at the top of the tree, and result in the shortest codes [4]. The tree is built by going through the following steps:

1. Each of the symbols is laid out as leaf node which is going to be connected. The symbols are ranked according to their frequency, which represents the probabilities of their occurrences in the source.

2. Two nodes with the lowest frequencies are combined to form a new node, which is a parent node of these two nodes. This parent node is then considered as a representative of the two nodes with a frequency equal to the sum of the frequencies of two nodes. Moreover, one of the children is assigned a "0" and the other is assigned a “1”.

3. Nodes are then successively combined as above until a binary tree containing all of nodes is created.

4. The code representing a given symbol can be determined by going from the root of the tree to the leaf node representing the symbol. The accumulation of symbol "0" and "1" is the code of that symbol.

By using this procedure, the symbols are naturally assigned codes that reflect probability distribution. Highly probable symbols will be given short codes, and improbable symbols will have long codes. Therefore, the average code length will be reduced. If the statistic of symbols is very biased to some symbols, the reduction will be very significant [3]. For example, imagine we have a text file that uses only five characters (A, B, C, D, E). Before we can assign bit patterns to each character, we assign each character a weight based on its frequency of use. The algorithm is optimal in the sense that the average number of bits required to represent the source symbols is a minimum provided the prefix condition is met [8].

Table 1 Frequency of Characters

Character	A	B	C	D	E
Frequency	17	12	12	27	32

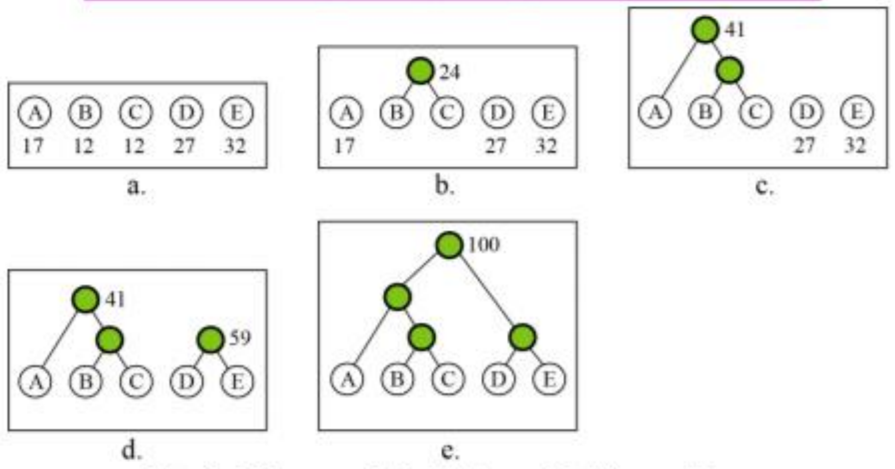


Fig. 1.1 Stages of Building a Huffman Tree

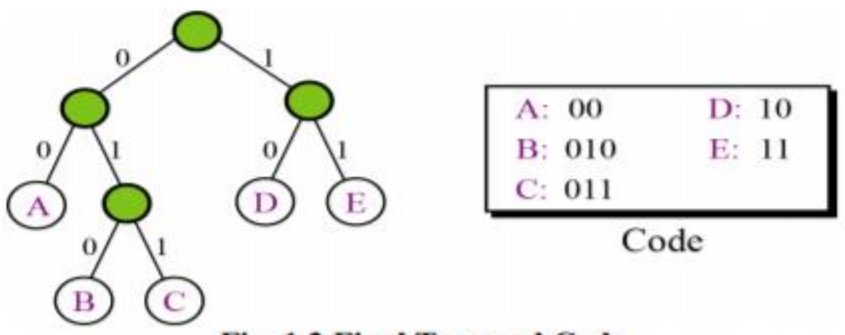


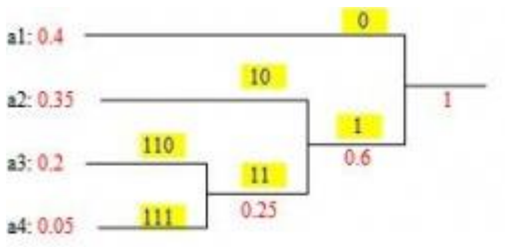
Fig. 1.2 Final Tree and Code

2 Encoder

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, **n**. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to **n** leaf nodes and **n-1** internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes has children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

A Huffman encoding can be computed by first creating a tree of nodes:



1. Create a leaf node for each symbol and add it to the priority queue.

2. While there is more than one node in the queue:

    a. Remove the node of highest priority (lowest probability) twice to get two nodes.

    b. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.

    c. Add the new node to the queue.

3. The remaining node is the root node and the tree is complete.

4. Traverse the constructed binary tree from root to leaves assigning and accumulating a '0' for one branch and a '1' for the other at each node. The accumulated zeros and ones at each leaf constitute a Huffman encoding for those symbols and weights:

Since Four Way Heap data structures require O (log *n*) time per insertion, and a tree with *n* leaves has 2*n*−1 nodes, this algorithm operates in O(*n* log *n*) time, where *n* is the number of symbols. In many cases, time complexity is not very important in the choice of algorithm here, since n here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when n grows to be very large.

3 Decoder

The process of decoding is simply a matter of converting the stream of prefix codes to individual byte values, usually by moving through the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that byte value). Before this can take place, however, the Huffman tree must be reconstructed from the data received in the form of code table.

1. Create a binary Huffman tree. Construct the Huffman tree from the code table by reading in the code for each symbol and traversing the tree while adding nodes. A left move is made for every "0" that is encountered and a right move for every "1" encountered until the code runs out. The symbol corresponding to this Huffman code is stored in the leaf terminating the path.

2. Read in the encoded binary file into a byte array 4096 elements at a time. While there are more elements to be processed:

    a. Convert the corresponding byte array to String and process one character at a time.

    b. Starting with the first bit in the stream, we then use successive bits from the stream to determine whether to go left or right in the decoding tree.

    c. When we reach a leaf of the tree, we've decoded a character, so we place that character onto the (uncompressed) output stream. The next bit in the input stream is the first bit of the next character.

3. Once the entire string in encoded binary is processed, the process is complete and the output will be generated.

Complexity of Decoder considering Reconstruction of Huffman tree (Size of code table * Log n) is part of Pre-Processing: No. of symbols in the decoded output (k) * Log n (Height of the Generated Huffman Tree)

4 Performance Evaluation

Huffman Tree Generation Times for Various Priority Queue Structures:

4-Way Cache Optimized Heap - 448 Millis

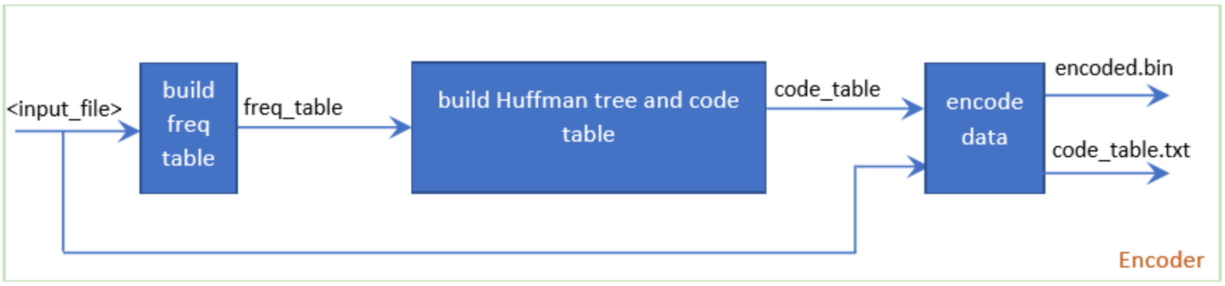
Binary Heap - 574 Millis

Pairing Heap - 659 Millis

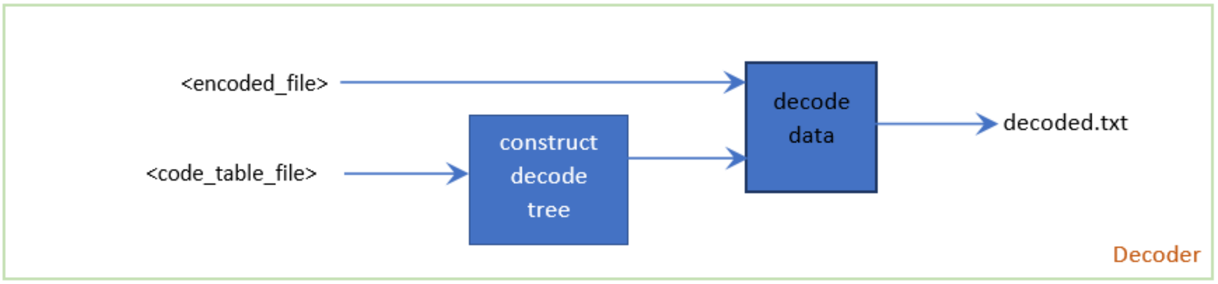
5 Program Structure and Function Prototypes

The program consists of 2 main classes.

1) encoder: Used to generate the code table and the encoded file from the input received.



2) decoder: Used to generate the decoded file from the encoded file and the code table.



The basic workflow of the program is the encoder takes as input a file and reads it simultaneously creating a frequency table which is used as input to generate the Huffman tree using the 4 way heap structure as the underlying Priority queue for the Tree generation. Once the Huffman tree has been successfully generated the codes corresponding to each symbol are written out in the code table. The codes are also stored in a Hash Map which is later used to decode the input file received from the user.

The decoder program takes as input 2 files: code table and encoded binary file. It first uses the code table to recreate the Huffman tree by traversing the code for each symbol and moving left for every ‘0’ and right for every ‘1’ that is encountered. The symbol corresponding to the traversed code is stored at the leaf node terminating the path. Once the Huffman tree is generated, the decoded file is reconstructed by using the encoded binary file as input, traversing the tree for every bit encountered and writing the symbol in leaf that terminates the path to the decoded output file.

Encoder

Variables	Type	Description
1. filename/ filepath	String	The input filename take from user.
2. hmap	HashMap<Integer,String>	The HashMap where Symbol Code Pairs are stored
3. hufftree	Bin4WayHeap	The instance of the 4 Way heap.
4. filename	String	Output file name for code table.
5. s	String	Stores the encoded input file.
6. CodeTree	Integer	The underlying Huffman tree structure.
7. leftchild	CodeTree	Stores the left subtree rooted at the node.
8. rightchild	CodeTree	Stores the right subtree rooted at the node.
9. frequency	Integer	Stores the frequency of the recurring symbol as part of the Huffman tree structure.
10. value	Integer	Key value to be stored in leaf node.
11. charFreqs	Integer Array	Frequency Table holding the frequency data for each recurring symbol in the

		input file

Function Name	Description
Void Bin4WayHeap.insert (New Huffman Tree)	Inserts an element into the Four Way Heap.
CodeTree Bin4WayHeap.removeMin (No Params)	Removes and returns the element with minimum frequency.
Integer Bin4WayHeap.size()	Returns the current Size of the Four Way Heap.
CodeTree	Implements the Huffman Tree Class
TreeLeaf extends CodeTree	Represents a Leaf Node containing the symbol in the Huffman tree. Default constructor initializes a leaf node with a set frequency and the symbol corresponding to this frequency.
InternalNode extends CodeTree	Represents an Internal Node in the Huffman tree. Default constructor takes 2 Huffman Tree Nodes and generates a Huffman tree with resulting frequency equal to the sum of frequency of these two nodes.
CodeTree buildTree (Freq Table)	Builds and returns Huffman Tree from the Frequency table.
Void codePrinter(CodeTree, codeBuffer, Writer)	Generates the codes for the inputs in the input file and writes them out to encoded.bin file.
public static void main (String args[])	The main function counts the frequency of the symbols in the input file and stores it in a frequency table charFreqs which is then passed to the buildTree function which generates the Huffman tree corresponding to this frequency table and returns the root. The codeprinter function is then called to traverse this Huffman tree and write the Symbol-Code pairs to the code table file. Finally, the input file is read and converted to the encoded binary file by traversing the generated Huffman tree. The encoded string does not contain ASCII characters, but binary values. For this purpose we make use of OutputStream to write one parsed byte at a time to the output file.

Decoder

Class Variable	Type	Description
1. codetable / encoded	String	Input Filenames taken from the user.
2. decoded	String	Filename for the decoded output.
3. tree	HuffmanTree	Binary Tree used to reconstruct the Huffman tree from the codes retrieved from the code table.

Function Name	Description
1. HuffmanNode `createHuffmanTree` (HuffmanNode node, String codePair, int currentPos, int codeLength)	The function recreates the Huffman Tree from the codes in the code table. The Symbol corresponding to a Huffman Code is stored at the leaf terminating the path traversed by the code. The root of the Huffman Tree is returned.
2. Void inOrder(root) / preOrder(root) / postOrder(root)	Gives the Inorder, Preorder and Postorder traversal of the Huffman Tree generated by taking as its input the root.
3. public static void main (String[] args)	This function takes as input the code table and the decoded binary fie. It passes the Symbol-Code values from the code table to the createHuffmanTree function to generate the corresponding Huffman tree. Once the Huffman tree is generated it is used to decode the encoded input. This is done by traversing the tree from the root and making a left move if a ‘0’ is encountered and a right move otherwise. The symbol pertaining to this code is retrieved from the leaf node terminating this path and written out to the decoded output file.

5 Summary

The objective of this assignment has been met. The program successfully creates an implementation of Four Way Heap and uses it to encode given input using Huffman encoding, while correctly performing the delete min and insert operations on a cache optimized four-way Heap. It further correctly decodes the encoded message by reconstructing the Huffman tree from the code table. Although Huffman's original algorithm is optimal for a symbol-by-symbol coding (i.e., a stream of unrelated symbols) with a known input probability distribution, it is not optimal when the symbol-by-symbol restriction is dropped, or when the probability mass functions are unknown. Huffman coding is optimal when each input symbol is a known independent and identically distributed random variable having a probability that is the inverse of a power of two.

7 References

[1] http://www.prepressure.com/library/compressionalgorithms/huffman

[2] David Salomon, Data Compression, The Complete Reference, 2nd Edition Springer-Verlag 1998.

[3] http://www.cprogramming.com/tutorial/computersciencetheory/huffman.html

[4] http://www.webopedia.com/TERM/H/Huffman_compression.html

[5] http://gradworks.umi.com/14/39/1439199.html

[6] www.hpl.hp.com/research/papers/seroussiIEEE.pdf

[7] http://ee.lamar.edu/gleb/dip/index.html

[8] http://www.prepressure.com/library/compression_algorithms/huffman

[9] http://www.webopedia.com/TERM/H/Huffman_compression.html

[10] Introduction to Data Compression, K. Sayood, Morgan Kauffman, Second Edition, 2000. (Primary)

[11] http://www.prepressure.com/library/file-formats

[12] www.FileFormt.info

[13] D. Clunie, Lossless Compression of Grayscale Medical Images, “Effectiveness of Traditional and State of the Art Approaches," in Proc. SPIE (Medical Imaging), vol.3980, Feb. 2000.

[14] Shannon CE, Weaver W, The mathematical theory of communication. University of Illinois Press, Urbana IL, 1949.