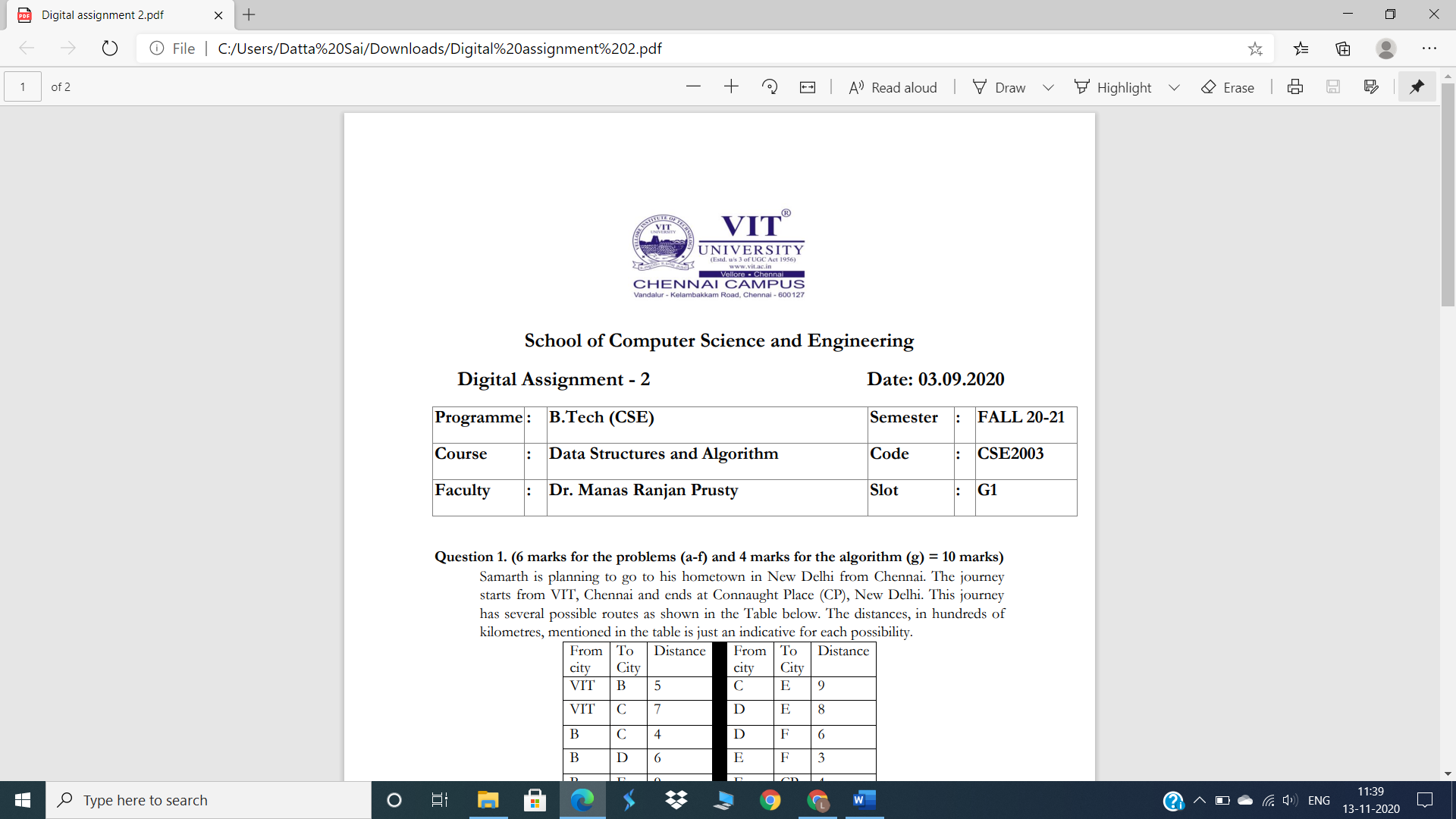
****

**Data Structures and Algorithms (CSE2003)**

**J-component**

**Faculty:** **Dr. Manas Ranjan Prusty**

**Slot: G1**

**Project Report**

**By:-**

**Lingam VVSN Datta Sai-19bce1373**

**Y Rajeev Krishna Chaitanya – 19bce1360**

**Topic: -**

**Data Compression Techniques**

**Contents: -**

**1. Abstract…………………………………………. 3**

**2. Introduction…………………………………….. 3**

**3. Literature survey / Related works…………….. 4**

**4. Existing work / system…………………………. 5**

**5. Proposed work / system………………………... 9**

**6. Working modules and Flow chart/ system design of proposed work……..................................... 9**

**7. Description of each modules with appropriate diagram and explanation…………………………...12**

**8. Complete Program…………………................. 18**

**9. Screen shots of output………………………… 34**

**10. Conclusion…………………………………...... 39**

**11. Future work…………………………………... 40**

**12. References……………………………………... 41**

**ABSTRACT: -**

**There are two categories of compression techniques, lossy and lossless. Which each uses different techniques to compress files, both have the same aim: To look for duplicate data in the graphic (GIF for LZW) and use a much more compact data representation. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. On the other hand, Lossy compression reduces bits by removing unnecessary or less important information.**

**So, we need Data Compression mainly because:**

**• Uncompressed data can take up a lot of space, which is not good for limited hard drive space and internet download speeds.**

**• While hardware gets better and cheaper, algorithms to reduce data size also helps technology evolve.**

**• Example: One minute of uncompressed HD video can be over 1 GB. How can we fit a two-hour film on a 25 GB Blu-ray disc?**

**The idea of the compression algorithm is the following: as the input data is being processed, a dictionary keeps a correspondence between the longest encountered words and a list of code values. The words are replaced by their corresponding codes and so the input file is compressed. Therefore, the efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.**

**Our aim is to study the data compression algorithms of**

**a. Huffman coding,**

**b.** **LZW compression technique.**

**And understand how exactly these algorithms compress the given data, the space required after encryption.**

**INTRODUCTION: -**

**Data compression is a reduction in the number of bits needed to represent data. Compressing data can save storage capacity, speed up file transfer, and decrease costs for storage hardware and network bandwidth.**

**How compression works?**

**Compression is performed by a program that uses a formula or algorithm to determine how to shrink the size of the data. For instance, an algorithm may represent a string of bits -- or 0s and 1s -- with a smaller string of 0s and 1s by using a dictionary for the conversion between them, or the formula may insert a reference or pointer to a string of 0s and 1s that the program has already seen.**

**As we are going to study two of the available data compression techniques let us see about them**

* **Huffman Coding:**

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

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

* **LZW compression technique:**

**Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations.[1] It is the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format.**

**The scenario described by Welch's 1984 paper encodes sequences of 8-bit data as fixed-length 12-bit codes. The codes from 0 to 255 represent 1-character sequences consisting of the corresponding 8-bit character, and the codes 256 through 4095 are created in a dictionary for sequences encountered in the data as it is encoded. At each stage in compression, input bytes are gathered into a sequence until the next character would make a sequence with no code yet in the dictionary. The code for the sequence (without that character) is added to the output, and a new code (for the sequence with that character) is added to the dictionary.**

**The idea was quickly adapted to other situations. In an image based on a colour table, for example, the natural character alphabet is the set of color table indexes, and in the 1980s, many images had small color tables (on the order of 16 colors).**

**For such a reduced alphabet, the full 12-bit codes yielded poor compression unless the image was large, so the idea of a variable-width code was introduced: codes typically start one bit wider than the symbols being encoded, and as each code size is used up, the code width increases by 1 bit, up to some prescribed maximum (typically 12 bits). When the maximum code value is reached, encoding proceeds using the existing table, but new codes are not generated for addition to the table.**

**Literature Survey/Related Works: -**

**1)International Journal of Computer Science Trends and Technology (IJCST) –** **A Study on Data Compression Using Huffman CodingAlgorithms**

**Authors: D.Jasmine Shoba(Research Scholar) , Dr.S.Sivakumar (Assistant Professor),Thanthai Hans Roever College, Perambalur**

**Tamil Nadu – India**

**Abstract:** Data reduction is one of the data preprocessing techniques which can be applied to obtain a reduced representation of the data set that is much smaller in volume, yet closely maintains the integrity of the original data. That is, mining on the reduced data set should be more efficient yet produce the same analytical results. Data compression is useful, where encoding mechanisms are used to reduce the data set size. In data compression, data encoding or transformations are applied so as to obtain a reduced or compressed representation of the original data. Huffman coding is a successful compression method used originally for text compression. Huffman's idea is, instead of using a fixed-length code such as 8 bit extended ASCII or DBCDIC for each symbol, to represent a frequently occurring character in a source with a shorter codeword and to represent a less frequently occurring one with a longer codeword.

2) American Journal of Engineering Research (AJER)

Author: Dheemanth H N, Dept of Computer Science, National Institute of Engineering, Karnataka, India

Abstract: Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. LZW compression is one of the Adaptive Dictionary techniques. The dictionary is created while the data are being encoded. So encoding can be done on the fly. The dictionary need not be transmitted. Dictionary can be built up at receiving end on the fly. If the dictionary overflows then we have to reinitialize the dictionary and add a bit to each one of the code words.

Choosing a large dictionary size avoids overflow, but spoils compressions. A codebook or dictionary containing the source symbols is constructed. For 8-bit monochrome images, the first 256 words of the dictionary are assigned to the gray levels 0-255. Remaining part of the dictionary is filled with sequences of the gray levels.LZW compression works best when applied on monochrome images and text files that contain repetitive text/patterns.

**Existing Works:**

**There are many existing data compression techniques**

**So, let us see what are they**

**1)lossless compression methods:**

**General purpose:**

* **bzip2 – Combines Burrows–Wheeler transform with RLE and Huffman coding**
* **Finite State Entropy – Entropy encoding, a tabled variant of ANS, used by LZFSE and Zstandard**
* **Huffman coding – Entropy encoding, pairs well with other algorithms, used by Unix's pack utility**
* **Lempel-Ziv compression (LZ77 and LZ78) – Dictionary-based algorithm that forms the basis for many other algorithms**
* **Lempel–Ziv–Markov chain algorithm (LZMA) – Very high compression ratio, used by 7zip and xz**
* **Lempel–Ziv–Oberhumer (LZO) – Designed for speed at the expense of compression ratios**
* **Lempel–Ziv–Storer–Szymanski (LZSS) – Used by WinRAR in tandem with Huffman coding**
* **DEFLATE – Combines LZSS compression with Huffman coding, used by ZIP, gzip, and PNG images**
* **Lempel–Ziv–Welch (LZW) – Used by GIF images and Unix's compress utility**
* **Lempel–Ziv Finite State Entropy (LZFSE) – Combines Lempel–Ziv and Finite State Entropy, used by iOS and macOS**
* **Zstandard – Combines LZ77, Finite State Entropy, and Huffman coding, used by the Linux kernel**
* **Prediction by partial matching (PPM) – Optimized for compressing plain text**
* **Run-length encoding (RLE) – Simple scheme that provides good compression of data containing many runs of the same value**

**Audio:**

* **Apple Lossless (ALAC – Apple Lossless Audio Codec)**
* **Adaptive Transform Acoustic Coding (ATRAC)**
* **Audio Lossless Coding (also known as MPEG-4 ALS)**
* **Direct Stream Transfer (DST)**
* **Dolby TrueHD**
* **DTS-HD Master Audio**
* **Free Lossless Audio Codec (FLAC)**
* **Meridian Lossless Packing (MLP)**
* **Monkey's Audio (Monkey's Audio APE)**
* **MPEG-4 SLS (also known as HD-AAC)**
* **OptimFROG**
* **Original Sound Quality (OSQ)**
* **RealPlayer (RealAudio Lossless)**
* **Shorten (SHN)**
* **TTA (True Audio Lossless)**
* **WavPack (WavPack lossless)**
* **WMA Lossless (Windows Media Lossless)**

**Raster graphics:**

* **HEIF – High Efficiency Image File Format (lossless or lossy compression, using HEVC)**
* **ILBM – (lossless RLE compression of Amiga IFF images)**
* **LDCT – Lossless Discrete Cosine Transform[2][3]**
* **JBIG2 – (lossless or lossy compression of B&W images)**
* **JPEG 2000 – (includes lossless compression method via LeGall-Tabatabai 5/3[4][5][6] reversible integer wavelet transform)**
* **JPEG XR – formerly WMPhoto and HD Photo, includes a lossless compression method**
* **JPEG-LS – (lossless/near-lossless compression standard)**
* **PCX – PiCture eXchange**
* **PDF – Portable Document Format (lossless or lossy compression)**
* **PNG – Portable Network Graphics**
* **TIFF – Tagged Image File Format (lossless or lossy compression)**
* **TGA – Truevision TGA**
* **WebP – (lossless or lossy compression of RGB and RGBA images)**
* **FLIF – Free Lossless Image Format**
* **AVIF – AOMedia Video 1 Image File Format**

**3D Graphics:**

* **OpenCTM – Lossless compression of 3D triangle meshes**

**2)Lossy compression methods:**

**Image:**

**Discrete cosine transform (DCT):**

* **JPEG[9]**
* **WebP (high-density lossless or lossy compression of RGB and RGBA images)**
* **High Efficiency Image Format (HEIF)**
* **Better Portable Graphics (BPG) (lossless or lossy compression)**
* **JPEG XR, a successor of JPEG with support for high dynamic range, wide gamut pixel formats (lossless or lossy compression)**

**Wavelet compression:**

* **JPEG 2000, JPEG's successor format that uses wavelets (lossless or lossy compression)**
* **DjVu**
* **ICER, used by the Mars Rovers, related to JPEG 2000 in its use of wavelets**
* **PGF, Progressive Graphics File (lossless or lossy compression)**
* **Cartesian Perceptual Compression, also known as CPC**
* **Fractal compression**
* **JBIG2 (lossless or lossy compression)**
* **S3TC texture compression for 3D computer graphics hardware**

**3D computer graphics:**

* **glTF**

**Video**

**Discrete cosine transform (DCT):**

* **H.261**
* **Motion JPEG**
* **MPEG-1 Part 2**
* **MPEG-2 Part 2**
* **MPEG-4 Part 2**
* **Advanced Video Coding (AVC / H.264 / MPEG-4 AVC) (may also be lossless, even in certain video sections)**
* **High Efficiency Video Coding**
* **Ogg Theora (noted for its lack of patent restrictions)**
* **VC-1**

**Wavelet compression**

* **Motion JPEG 2000**
* **Dirac**
* **Sorenson video codec**

**Audio**

**General**

* **Modified discrete cosine transform (MDCT)**
* **Dolby Digital (AC-3)**
* **Adaptive Transform Acoustic Coding (ATRAC)**
* **MPEG Layer III (MP3)**
* **Advanced Audio Coding (AAC / MP4 Audio)**
* **Vorbis**
* **Windows Media Audio (WMA) (Standard and Pro profiles are lossy. WMA Lossless is also available.)**
* **LDAC**
* **Opus (Notable for lack of patent restrictions, low delay, and high quality speech and general audio.)**
* **Adaptive differential pulse-code modulation (ADPCM)**
* **Master Quality Authenticated (MQA)**
* **MPEG-1 Audio Layer II (MP2)**
* **Musepack (based on Musicam)**
* **aptX/ aptX-HD**

**Speech**

* **Linear predictive coding (LPC)**
* **Adaptive predictive coding (APC)**
* **Code-excited linear prediction (CELP)**
* **Algebraic code-excited linear prediction (ACELP)**
* **Relaxed code-excited linear prediction (RCELP)**
* **Low-delay CELP (LD-CELP)**
* **Adaptive Multi-Rate (used in GSM and 3GPP)**
* **Codec2 (noted for its lack of patent restrictions)**
* **Speex (noted for its lack of patent restrictions)**
* **Modified discrete cosine transform (MDCT)**
* **AAC-LD**
* **Constrained Energy Lapped Transform (CELT)**
* **Opus (mostly for real-time applications)**

**Proposed work: -**

**Our proposed work is to study two of the compression techniques i.e Huffman coding and lzw compression.**

**And understand how they are working, how to use them and implement them by using the following methods:**

**1)Huffman compression:**

* In this implementation, we use an array-based data structure for representing binary trees, which takes advantage of the manner in which the coding tree is built from the bottom up.
* In Huffman coding, we choose the nodes with the lowest weights.

2)LZW compression:

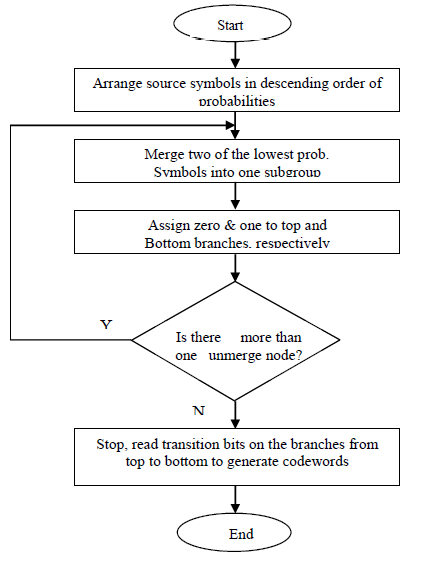
The idea of the compression algorithm is the following: as the input data is being processed, a dictionary keeps a correspondence between the longest encountered words and a list of code values. The words are replaced by their corresponding codes and so the input file is compressed. Therefore, the efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.

**Design And flow Chart:**

**1)Huffman coding:**

There are mainly two major parts in Huffman Coding.

* Build a Huffman Tree from input characters.
* Traverse the Huffman Tree and assign codes to characters.



**Figure1: flow chart of Huffman coding process.**

**2)Huffman Decoding:**

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

**3)Lzw compression:**

**LZW compression uses a code table, as illustrated in. A common choice is to provide 4096 entries in the table. In this case, the LZW encoded data consists entirely of 12 bit codes, each referring to one of the entries in the code table. Uncompression is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. For example, if only these first 256 codes were used, each byte in the original file would be converted into 12 bits in the LZW encoded file, resulting in a 50% larger file size. During uncompression, each 12 bit code would be translated via the code table back into the single bytes. Of course, this wouldn't be a useful situation.**

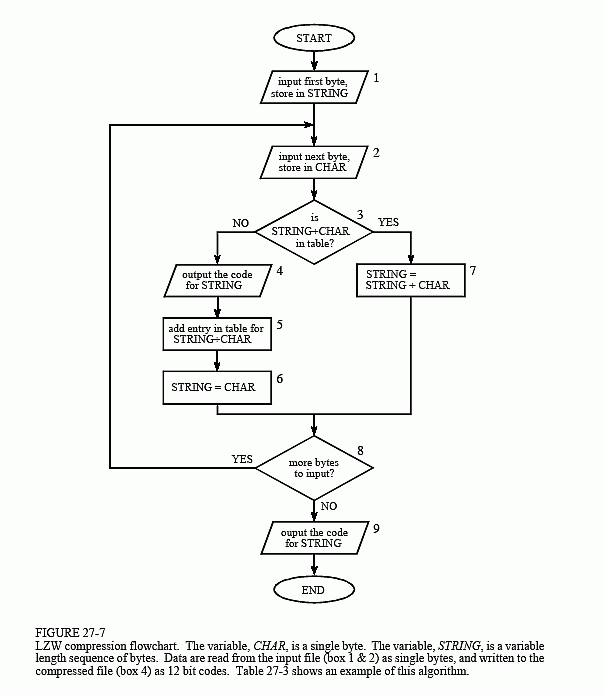
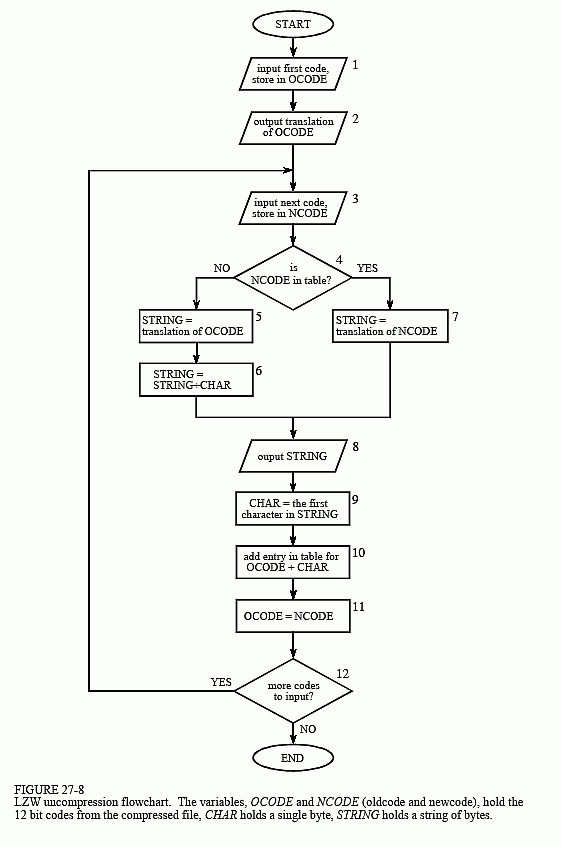


Figure2:Flow chart of LZW compression process.



**Figure 3:Flow chart of LZW decoding process.**

**Explanation:**

**Huffman coding:**

Suppose the string below

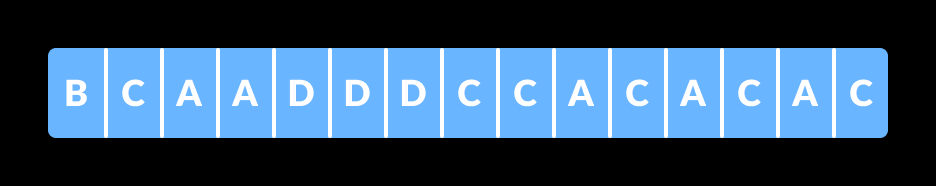


Figure 4:Example string.

**Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 \* 15 = 120 bits are required to send this string.** **Using the Huffman Coding technique, we can compress the string to a smaller size.**

**Huffman coding is done with the help of the following steps.**

**1)Calculate the frequency of each character in the string.**

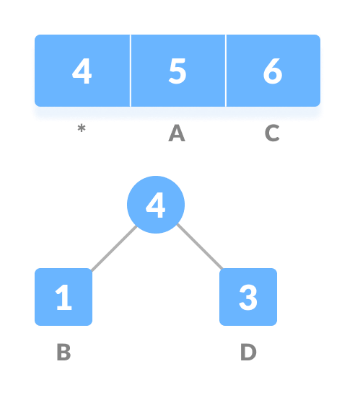
**B-1, C-6, A-5, D-3**

**2)** **Sort the characters in increasing order of the frequency. These are stored.**

**B-1, D-3, A-5, C-6**

**3)Make each unique character as a leaf node.**

**4)Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.**

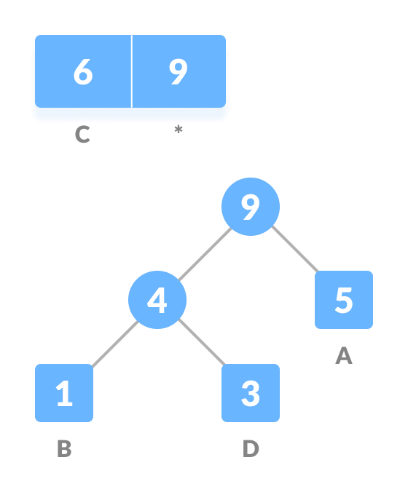


**Figure5:Basic Tree.**

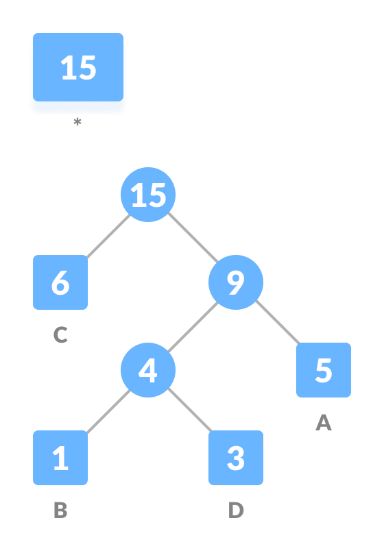
**5)Remove these two minimum frequencies from Q and add the sum into the list of frequencies (\* denote the internal nodes in the figure above).**

**6)Insert node z into the tree.**

**7)Repeat steps 3 to 5 for all the characters.**

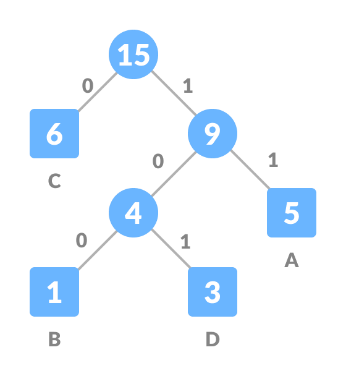


**Figure6:Example construction of build heap tree.**

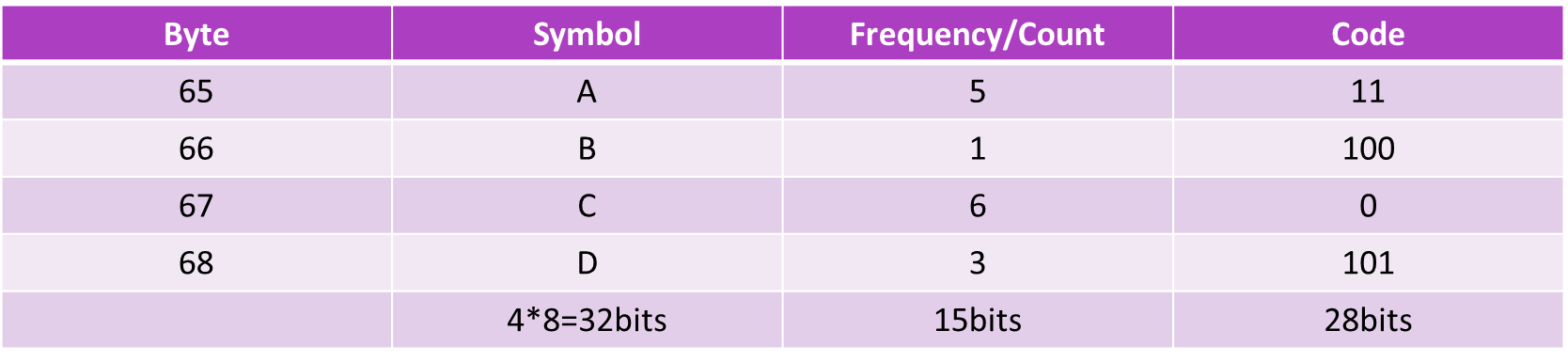


**Figure7:Building tree process continues.**

**8)** **For each non-leaf node, assign 0 to the left edge and 1 to the right edge.**

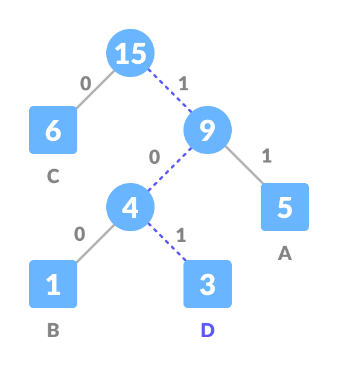


**Figure8:Final tree.**

****

* Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 \* 15 = 120 bits are required to send this string before encoding.
* Now after encoding the size is reduced to 32+15+28=75bits

**Decoding:**

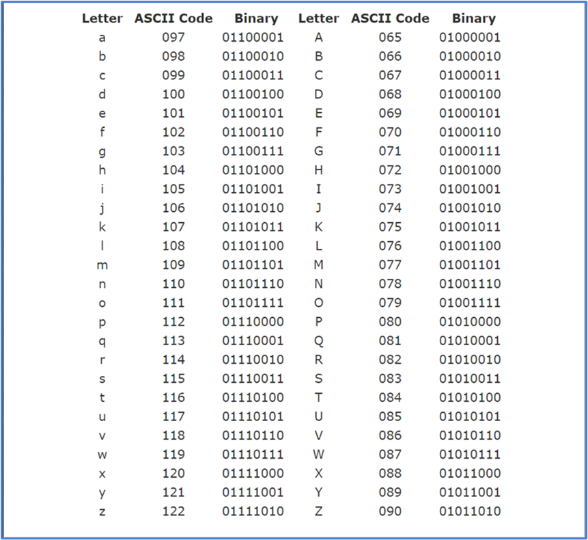


**Figure9:Decoding Tree.**

For decoding if we get one in the encoded file we move right and once the leaf nodes comes that will be symbol and continue the process , if we find zero as starting we move left until leaf node according to code and find the letter and prints it. This process gets continued.

For examples lets take 101 so first move right to 9 and again to 4 and arrive at 3.So the symbol is D.

**Ascii representation:**

****

**Figure 10:ASCII representation.**

**Lzw Compression:**

**LZW compression works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes. Because the codes take up less space than the strings they replace, we get compression.Characteristic features of LZW includes,**

**LZW compression uses a code table, with 4096 as a common choice for the number of table entries. Codes 0-255 in the code table are always assigned to represent single bytes from the input file.**

**When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks. Compression is achieved by using codes 256 through 4095 to represent sequences of bytes.**

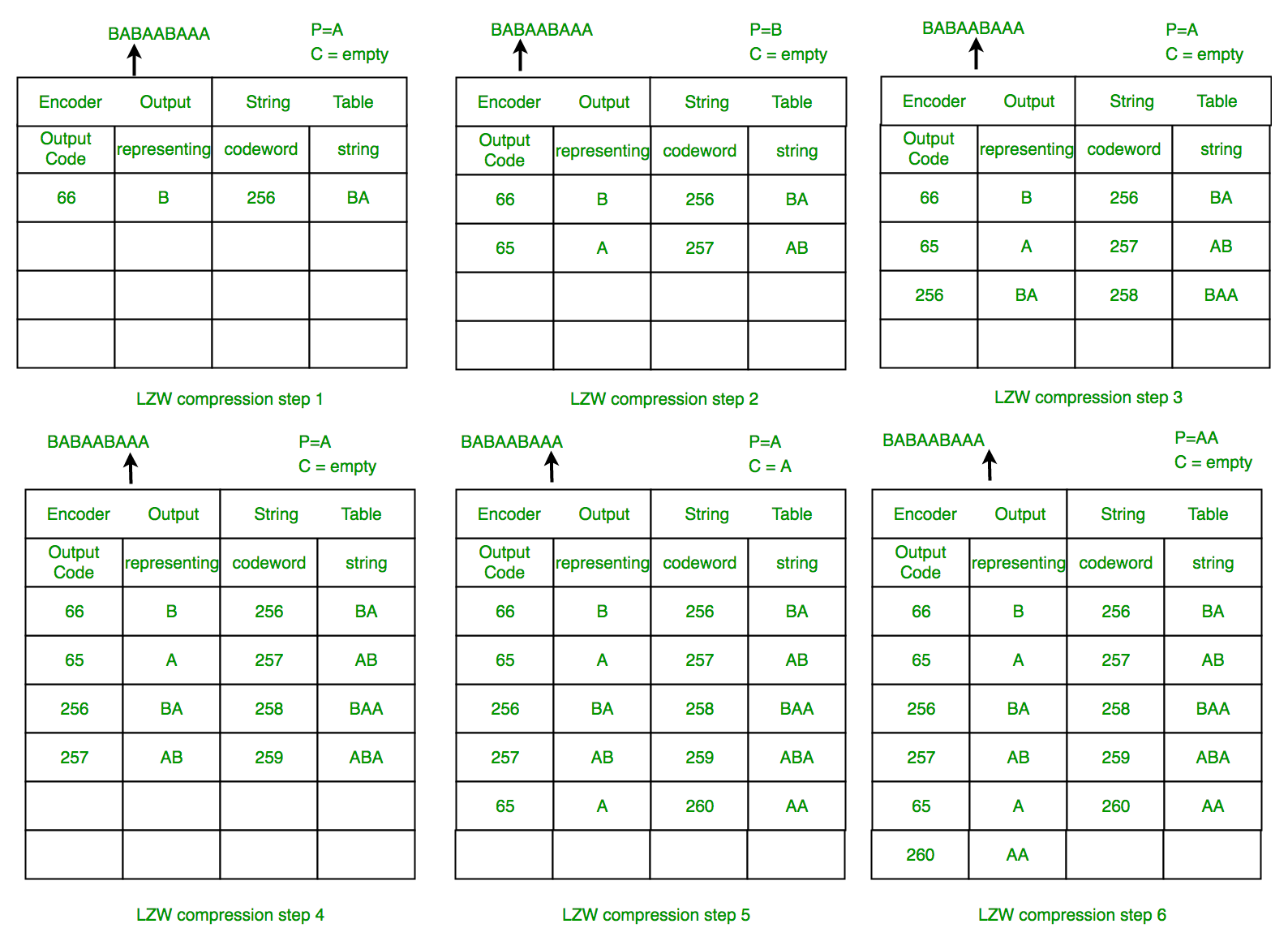
**As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table.**

**Decoding is achieved by taking each code from the compressed file and translating it through the code table to find what character or characters it represents.**

**Example: ASCII code. Typically, every character is stored with 8 binary bits, allowing up to 256 unique symbols for the data. This algorithm tries to extend the library to 9 to 12 bits per character.The new unique symbols are made up of combinations of symbols that occurred previously in the string. It does not always compress well, especially with short, diverse strings. But is good for compressing redundant data, and does not have to save the new dictionary with the data: this method can both compress and uncompress data.**

**Example : Use the LZW algorithm to compress the string: BABAABAAA**

**The steps involved are systematically shown in the diagram below.**



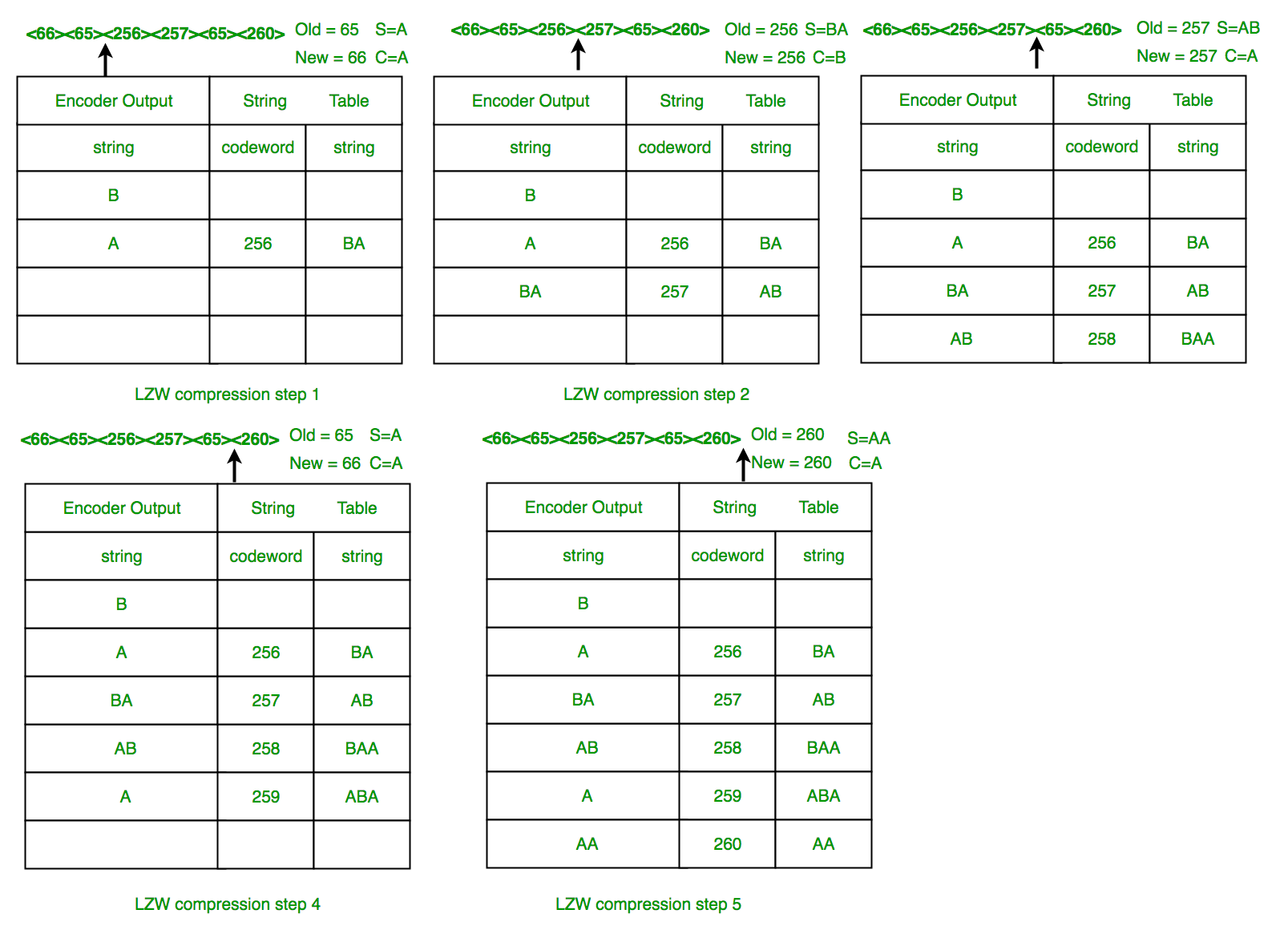
**Figure 11:LZW encoding process.**

**Lzw uncompression:**

**The LZW decompressor creates the same string table during decompression. It starts with the first 256 table entries initialized to single characters. The string table is updated for each character in the input stream, except the first one.Decoding achieved by reading codes and translating them through the code table being built.**

**Example : LZW Decompression: Use LZW to decompress the output sequence of : <66><65><256><257><65><260>**

**The steps involved are systematically shown in the diagram below.**

**Figure 12:LZW Decoding process.**

**In this example, 72 bits are represented with 72 bits of data. After a reasonable string table is built, compression improves dramatically.**

**LZW Summary: This algorithm compresses repetitive sequences of data very well. Since the codewords are 12 bits, any single encoded character will expand the data size rather than reduce it.**

**Algorithms:**

* **LZW compression:**

**Initialize table with single character strings**

**P = first input character**

**WHILE not end of input stream**

**C = next input character**

**IF P + C is in the string table**

**P = P + C**

**ELSE**

**output the code for P**

**add P + C to the string table**

**P = C**

**END WHILE**

**output code for P**

* **LZW decompression:**

**Initialize table with single character strings**

**OLD = first input code**

**output translation of OLD**

**WHILE not end of input stream**

**NEW = next input code**

**IF NEW is not in the string table**

**S = translation of OLD**

**S = S + C**

**ELSE**

**S = translation of NEW**

**output S**

**C = first character of S**

**OLD + C to the string table**

**OLD = NEW**

**END WHILE**

* **Huffman encoding:**

**Procedure Huffman(C): // C is the set of n characters and related information**

**n = C.size**

**Q = priority\_queue()**

**for i = 1 to n**

**n = node(C[i])**

**Q.push(n)**

**end for**

**while Q.size() is not equal to 1**

**Z = new node()**

**Z.left = x = Q.pop**

**Z.right = y = Q.pop**

**Z.frequency = x.frequency + y.frequency**

**Q.push(Z)**

**end while**

**Return Q**

* **Huffman decoding:**

**Procedure HuffmanDecompression(root, S): // root represents the root of Huffman Tree**

**n := S.length // S refers to bit-stream to be decompressed**

**for i := 1 to n**

**current = root**

**while current.left != NULL and current.right != NULL**

**if S[i] is equal to '0'**

**current := current.left**

**else**

**current := current.right**

**endif**

**i := i+1**

**endwhile**

**print current.symbol**

**endfor**

**COMPLETE PROGRAMS:**

**Huffman.c: -**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define MAX\_BUFFER\_SIZE 256**

**#define INVALID\_BIT\_READ -1**

**#define INVALID\_BIT\_WRITE -1**

**#define FAILURE 1**

**#define SUCCESS 0**

**#define FILE\_OPEN\_FAIL -1**

**#define END\_OF\_FILE -1**

**#define MEM\_ALLOC\_FAIL -1**

**int num\_alphabets = 256;**

**int num\_active = 0;**

**int \*frequency = NULL;**

**unsigned int original\_size = 0;**

**typedef struct {**

**int index;**

**unsigned int weight;**

**} node\_t;**

**node\_t \*nodes = NULL;**

**int num\_nodes = 0;**

**int \*leaf\_index = NULL;**

**int \*parent\_index = NULL;**

**int free\_index = 1;**

**int \*stack;**

**int stack\_top;**

**unsigned char buffer[MAX\_BUFFER\_SIZE];**

**int bits\_in\_buffer = 0;**

**int current\_bit = 0;**

**int eof\_input = 0;**

**int read\_header(FILE \*f);**

**int write\_header(FILE \*f);**

**int read\_bit(FILE \*f);**

**int write\_bit(FILE \*f, int bit);**

**int flush\_buffer(FILE \*f);**

**void decode\_bit\_stream(FILE \*fin, FILE \*fout);**

**int decode(const char\* ifile, const char \*ofile);**

**void encode\_alphabet(FILE \*fout, int character);**

**int encode(const char\* ifile, const char \*ofile);**

**void build\_tree();**

**void add\_leaves();**

**int add\_node(int index, int weight);**

**void finalise();**

**void init();**

**void determine\_frequency(FILE \*f) {**

**int c;**

**while ((c = fgetc(f)) != EOF) {**

**++frequency[c];**

**++original\_size;**

**}**

**for (c = 0; c < num\_alphabets; ++c)**

**if (frequency[c] > 0)**

**++num\_active;**

**}**

**void init() {**

**frequency = (int \*)**

**calloc(2 \* num\_alphabets, sizeof(int));**

**leaf\_index = frequency + num\_alphabets - 1;**

**}**

**void allocate\_tree() {**

**nodes = (node\_t \*)**

**calloc(2 \* num\_active, sizeof(node\_t));**

**parent\_index = (int \*)**

**calloc(num\_active, sizeof(int));**

**}void finalise() {**

**free(parent\_index);**

**free(frequency);**

**free(nodes);**

**}**

**int add\_node(int index, int weight) {**

**int i = num\_nodes++;**

**while (i > 0 && nodes[i].weight > weight) {**

**memcpy(&nodes[i + 1], &nodes[i], sizeof(node\_t));**

**if (nodes[i].index < 0)**

**++leaf\_index[-nodes[i].index];**

**else**

**++parent\_index[nodes[i].index];**

**--i;**

**}**

**++i;**

**nodes[i].index = index;**

**nodes[i].weight = weight;**

**if (index < 0)**

**leaf\_index[-index] = i;**

**else**

**parent\_index[index] = i;**

**return i;**

**}**

**void add\_leaves() {**

**int i, freq;**

**for (i = 0; i < num\_alphabets; ++i) {**

**freq = frequency[i];**

**if (freq > 0)**

**add\_node(-(i + 1), freq);**

**}**

**}**

**void build\_tree() {**

**int a, b, index;**

**while (free\_index < num\_nodes) {**

**a = free\_index++;**

**b = free\_index++;**

**index = add\_node(b/2,**

**nodes[a].weight + nodes[b].weight);**

**parent\_index[b/2] = index;**

**}**

**}**

**int encode(const char\* ifile, const char \*ofile) {**

**FILE \*fin, \*fout;**

**if ((fin = fopen(ifile, "rb")) == NULL) {**

**perror("Failed to open input file");**

**return FILE\_OPEN\_FAIL;**

**}**

**if ((fout = fopen(ofile, "wb")) == NULL) {**

**perror("Failed to open output file");**

**fclose(fin);**

**return FILE\_OPEN\_FAIL;**

**}**

**determine\_frequency(fin);**

**stack = (int \*) calloc(num\_active - 1, sizeof(int));**

**allocate\_tree();**

**add\_leaves();**

**write\_header(fout);**

**build\_tree();**

**fseek(fin, 0, SEEK\_SET);**

**int c;**

**while ((c = fgetc(fin)) != EOF)**

**encode\_alphabet(fout, c);**

**flush\_buffer(fout);**

**free(stack);**

**fclose(fin);**

**fclose(fout);**

**return 0;**

**}**

**void encode\_alphabet(FILE \*fout, int character) {**

**int node\_index;**

**stack\_top = 0;**

**node\_index = leaf\_index[character + 1];**

**while (node\_index < num\_nodes) {**

**stack[stack\_top++] = node\_index % 2;**

**node\_index = parent\_index[(node\_index + 1) / 2];**

**}**

**while (--stack\_top > -1)**

**write\_bit(fout, stack[stack\_top]);**

**}**

**int decode(const char\* ifile, const char \*ofile) {**

**FILE \*fin, \*fout;**

**if ((fin = fopen(ifile, "rb")) == NULL) {**

**perror("Failed to open input file");**

**return FILE\_OPEN\_FAIL;**

**}**

**if ((fout = fopen(ofile, "wb")) == NULL) {**

**perror("Failed to open output file");**

**fclose(fin);**

**return FILE\_OPEN\_FAIL;**

**}**

**if (read\_header(fin) == 0) {**

**build\_tree();**

**decode\_bit\_stream(fin, fout);**

**}**

**fclose(fin);**

**fclose(fout);**

**return 0;**

**}**

**void decode\_bit\_stream(FILE \*fin, FILE \*fout) {**

**int i = 0, bit, node\_index = nodes[num\_nodes].index;**

**while (1) {**

**bit = read\_bit(fin);**

**if (bit == -1)**

**break;**

**node\_index = nodes[node\_index \* 2 - bit].index;**

**if (node\_index < 0) {**

**char c = -node\_index - 1;**

**fwrite(&c, 1, 1, fout);**

**if (++i == original\_size)**

**break;**

**node\_index = nodes[num\_nodes].index;**

**}**

**}**

**}**

**int write\_bit(FILE \*f, int bit) {**

**if (bits\_in\_buffer == MAX\_BUFFER\_SIZE << 3) {**

**size\_t bytes\_written =**

**fwrite(buffer, 1, MAX\_BUFFER\_SIZE, f);**

**if (bytes\_written < MAX\_BUFFER\_SIZE && ferror(f))**

**return INVALID\_BIT\_WRITE;**

**bits\_in\_buffer = 0;**

**memset(buffer, 0, MAX\_BUFFER\_SIZE);**

**}**

**if (bit)**

**buffer[bits\_in\_buffer >> 3] |=**

**(0x1 << (7 - bits\_in\_buffer % 8));**

**++bits\_in\_buffer;**

**return SUCCESS;**

**}**

**int flush\_buffer(FILE \*f) {**

**if (bits\_in\_buffer) {**

**size\_t bytes\_written =**

**fwrite(buffer, 1,**

**(bits\_in\_buffer + 7) >> 3, f);**

**if (bytes\_written < MAX\_BUFFER\_SIZE && ferror(f))**

**return -1;**

**bits\_in\_buffer = 0;**

**}**

**return 0;**

**}**

**int read\_bit(FILE \*f) {**

**if (current\_bit == bits\_in\_buffer) {**

**if (eof\_input)**

**return END\_OF\_FILE;**

**else {**

**size\_t bytes\_read =**

**fread(buffer, 1, MAX\_BUFFER\_SIZE, f);**

**if (bytes\_read < MAX\_BUFFER\_SIZE) {**

**if (feof(f))**

**eof\_input = 1;**

**}**

**bits\_in\_buffer = bytes\_read << 3;**

**current\_bit = 0;**

**}**

**}**

**if (bits\_in\_buffer == 0)**

**return END\_OF\_FILE;**

**int bit = (buffer[current\_bit >> 3] >>**

**(7 - current\_bit % 8)) & 0x1;**

**++current\_bit;**

**return bit;**

**}**

**int write\_header(FILE \*f) {**

**int i, j, byte = 0,**

**size = sizeof(unsigned int) + 1 +**

**num\_active \* (1 + sizeof(int));**

**unsigned int weight;**

**char \*buffer = (char \*) calloc(size, 1);**

**if (buffer == NULL)**

**return MEM\_ALLOC\_FAIL;**

**j = sizeof(int);**

**while (j--)**

**buffer[byte++] =**

**(original\_size >> (j << 3)) & 0xff;**

**buffer[byte++] = (char) num\_active;**

**for (i = 1; i <= num\_active; ++i) {**

**weight = nodes[i].weight;**

**buffer[byte++] =**

**(char) (-nodes[i].index - 1);**

**j = sizeof(int);**

**while (j--)**

**buffer[byte++] =**

**(weight >> (j << 3)) & 0xff;**

**}**

**fwrite(buffer, 1, size, f);**

**free(buffer);**

**return 0;**

**}**

**int read\_header(FILE \*f) {**

**int i, j, byte = 0, size;**

**size\_t bytes\_read;**

**unsigned char buff[4];**

**bytes\_read = fread(&buff, 1, sizeof(int), f);**

**if (bytes\_read < 1)**

**return END\_OF\_FILE;**

**byte = 0;**

**original\_size = buff[byte++];**

**while (byte < sizeof(int))**

**original\_size =**

**(original\_size << (1 << 3)) | buff[byte++];**

**bytes\_read = fread(&num\_active, 1, 1, f);**

**if (bytes\_read < 1)**

**return END\_OF\_FILE;**

**allocate\_tree();**

**size = num\_active \* (1 + sizeof(int));**

**unsigned int weight;**

**char \*buffer = (char \*) calloc(size, 1);**

**if (buffer == NULL)**

**return MEM\_ALLOC\_FAIL;**

**fread(buffer, 1, size, f);**

**byte = 0;**

**for (i = 1; i <= num\_active; ++i) {**

**nodes[i].index = -(buffer[byte++] + 1);**

**j = 0;**

**weight = (unsigned char) buffer[byte++];**

**while (++j < sizeof(int)) {**

**weight = (weight << (1 << 3)) |**

**(unsigned char) buffer[byte++];**

**}**

**nodes[i].weight = weight;**

**}**

**num\_nodes = (int) num\_active;**

**free(buffer);**

**return 0;**

**}**

**void print\_help() {**

**fprintf(stderr,**

**"USAGE: ./huffman [encode | decode] "**

**"<input out> <output file>\n");**

**}**

**int main(int argc, char \*\*argv) {**

**if (argc != 4) {**

**print\_help();**

**return FAILURE;**

**}**

**init();**

**if (strcmp(argv[1], "encode") == 0)**

**encode(argv[2], argv[3]);**

**else if (strcmp(argv[1], "decode") == 0)**

**decode(argv[2], argv[3]);**

**else**

**print\_help();**

**finalise();**

**return SUCCESS;**

**}**

**Make file:**

**CC=gcc**

**CFLAGS=-Wall -g**

**huffman: huffman.c**

**${CC} ${CFLAGS} -o huffman huffman.c**

**clean:**

**rm Huffman**

**For LZW:**

**Algorithms.c:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include "dictionary.c"**

**#include "file.c"**

**#include "array.c"**

**enum {**

**dictionarySize = 4095,**

**codeLength = 12,**

**maxValue = dictionarySize - 1**

**};**

**void compress(FILE \*inputFile, FILE \*outputFile);**

**void decompress(FILE \*inputFile, FILE \*outputFile);**

**int decode(int code, FILE \* outputFile);**

**void compress(FILE \*inputFile, FILE \*outputFile) {**

**int prefix = getc(inputFile);**

**if (prefix == EOF) {**

**return;**

**}**

**int character;**

**int nextCode;**

**int index;**

**nextCode = 256;**

**dictionaryInit();**

**while ((character = getc(inputFile)) != (unsigned)EOF) { // ch = read a character;**

**if ((index = dictionaryLookup(prefix, character)) != -1) prefix = index;**

**else { /**

**writeBinary(outputFile, prefix);**

**if (nextCode < dictionarySize) dictionaryAdd(prefix, character, nextCode++);**

**prefix = character;**

**}**

**}**

**writeBinary(outputFile, prefix);**

**if (leftover > 0) fputc(leftoverBits << 4, outputFile);**

**dictionaryDestroy();**

**}**

**void decompress(FILE \* inputFile, FILE \* outputFile) {**

**int previousCode; int currentCode;**

**int nextCode = 256;**

**int firstChar;**

**previousCode = readBinary(inputFile);**

**if (previousCode == 0) {**

**return;**

**}**

**fputc(previousCode, outputFile);**

**while ((currentCode = readBinary(inputFile)) > 0) {**

**if (currentCode >= nextCode) {**

**fputc(firstChar = decode(previousCode, outputFile), outputFile);**

**} else firstChar = decode(currentCode, outputFile);**

**if (nextCode < dictionarySize) dictionaryArrayAdd(previousCode, firstChar, nextCode++);**

**previousCode = currentCode;**

**} }**

**int decode(int code, FILE \* outputFile) {**

**int character; int temp;**

**if (code > 255) {**

**character = dictionaryArrayCharacter(code);**

**temp = decode(dictionaryArrayPrefix(code), outputFile);**

**} else {**

**character = code;**

**temp = code;**

**}**

**fputc(character, outputFile);**

**return temp;**

**}**

**Array.c:**

**typedef struct{**

**int prefix;**

**int character;**

**} DictElement;**

**void dictionaryArrayAdd(int prefix, int character, int value);**

**int dictionaryArrayPrefix(int value);**

**int dictionaryArrayCharacter(int value);**

**DictElement dictionaryArray[4095];**

**void dictionaryArrayAdd(int prefix, int character, int value) {**

**dictionaryArray[value].prefix = prefix;**

**dictionaryArray[value].character = character;**

**}**

**int dictionaryArrayPrefix(int value) {**

**return dictionaryArray[value].prefix;**

**}**

**int dictionaryArrayCharacter(int value) {**

**return dictionaryArray[value].character;**

**}**

**Dictionary.c:**

**#include <stdio.h>**

**#include <stdlib.h>**

**enum {**

**emptyPrefix = -1**

**};**

**struct DictNode {**

**int value;**

**int prefix;**

**int character;**

**struct DictNode \*next;**

**};**

**void dictionaryInit();**

**void appendNode(struct DictNode \*node);**

**void dictionaryDestroy();**

**int dictionaryLookup(int prefix, int character);**

**int dictionaryPrefix(int value);**

**int dictionaryCharacter(int value);**

**void dictionaryAdd(int prefix, int character, int value);**

**struct DictNode \*dictionary, \*tail;**

**void dictionaryInit() {**

**int i;**

**struct DictNode \*node;**

**for (i = 0; i < 256; i++) {**

**node = (struct DictNode \*)malloc(sizeof(struct DictNode));**

**node->prefix = emptyPrefix;**

**node->character = i;**

**appendNode(node);**

**}**

**}**

**void appendNode(struct DictNode \*node) {**

**if (dictionary != NULL) tail->next = node;**

**else dictionary = node;**

**tail = node;**

**node->next = NULL;**

**}**

**void dictionaryDestroy() {**

**while (dictionary != NULL) {**

**dictionary = dictionary->next;**

**}**

**}**

**int dictionaryLookup(int prefix, int character) {**

**struct DictNode \*node;**

**for (node = dictionary; node != NULL; node = node->next) {**

**if (node->prefix == prefix && node->character == character) return node->value;**

**}**

**return emptyPrefix;**

**}**

**int dictionaryPrefix(int value) {**

**struct DictNode \*node;**

**for (node = dictionary; node != NULL; node = node->next) {**

**if (node->value == value) return node->prefix;**

**}**

**return -1;**

**}**

**int dictionaryCharacter(int value) {**

**struct DictNode \*node;**

**for (node = dictionary; node != NULL; node = node->next) {**

**if (node->value == value) {**

**return node->character;**

**}**

**}**

**return -1;**

**}**

**void dictionaryAdd(int prefix, int character, int value) {**

**struct DictNode \*node;**

**node = (struct DictNode \*)malloc(sizeof(struct DictNode));**

**node->value = value;**

**node->prefix = prefix;**

**node->character = character;**

**appendNode(node);**

**}**

**File.c:**

**#include <stdio.h>**

**void writeBinary(FILE \* output, int code);**

**int readBinary(FILE \* input);**

**int leftover = 0;**

**int leftoverBits;**

**void writeBinary(FILE \* output, int code) {**

**if (leftover > 0) {**

**int previousCode = (leftoverBits << 4) + (code >> 8);**

**fputc(previousCode, output);**

**fputc(code, output);**

**leftover = 0;**

**} else {**

**leftoverBits = code & 0xF;**

**leftover = 1;**

**fputc(code >> 4, output);**

**}**

**}**

**int readBinary(FILE \* input) {**

**int code = fgetc(input);**

**if (code == EOF) return 0;**

**if (leftover > 0) {**

**code = (leftoverBits << 8) + code;**

**leftover = 0;**

**} else {**

**int nextCode = fgetc(input);**

**leftoverBits = nextCode & 0xF;**

**leftover = 1;**

**code = (code << 4) + (nextCode >> 4);**

**}**

**return code;**

**}**

**Lzw.c:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include "algorithms.c"**

**FILE \*inputFile;**

**FILE \*outputFile;**

**int main(int argc, char\*\* argv) {**

**if (argc > 2) {**

**if (strcmp(argv[1], "c") == 0) {**

**inputFile = fopen(argv[2], "r");**

**outputFile = fopen(strcat(argv[2], ".lzw"), "w+b");**

**if (outputFile == NULL || inputFile == NULL) {**

**printf("Can't open files\n'"); return 0;**

**}**

**compress(inputFile, outputFile);**

**} else {**

**inputFile = fopen(argv[2], "rb");**

**char temp[20]; int length = strlen(argv[2])-4;**

**strncpy(temp, argv[2], length);**

**temp[length] = '\0';**

**outputFile = fopen(temp, "w");**

**if (outputFile == NULL || inputFile == NULL) {**

**printf("Can't open files\n'"); return 0;**

**}**

**decompress(inputFile, outputFile);**

**}**

**fclose(inputFile); fclose(outputFile);**

**} else {**

**printf("Usage: lzw <command> <input file>\n\n");**

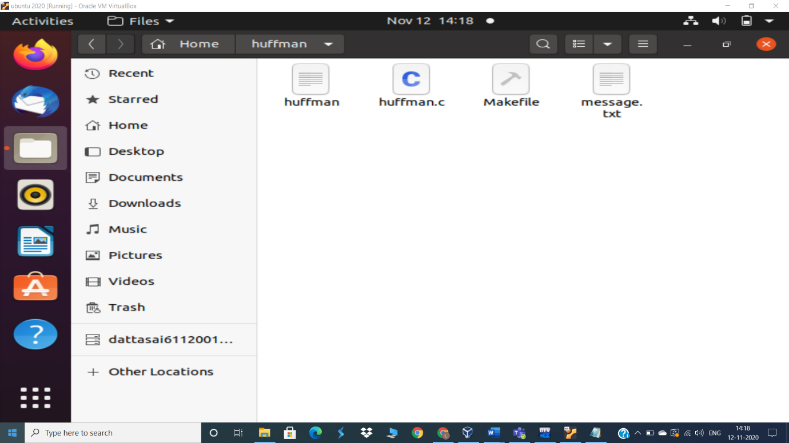
**printf("<Commands>\n c Compress\n d Decompress\n");**

**} return 0;**

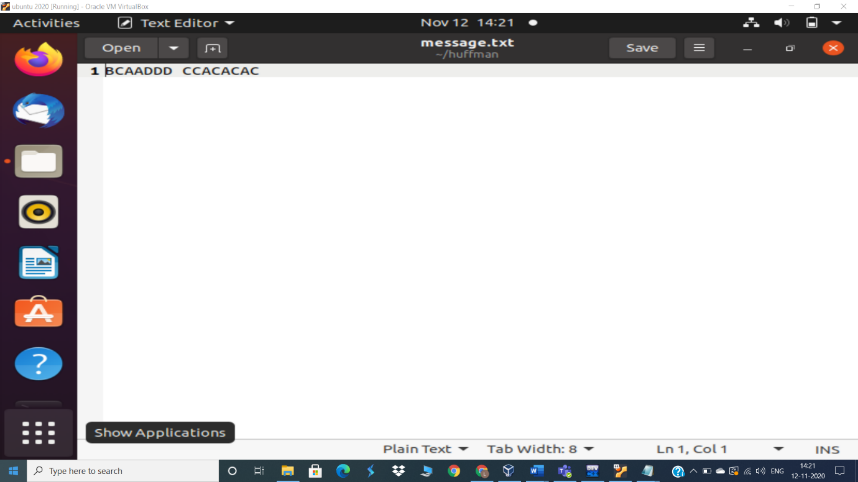
**}**

**Output Screenshots:**

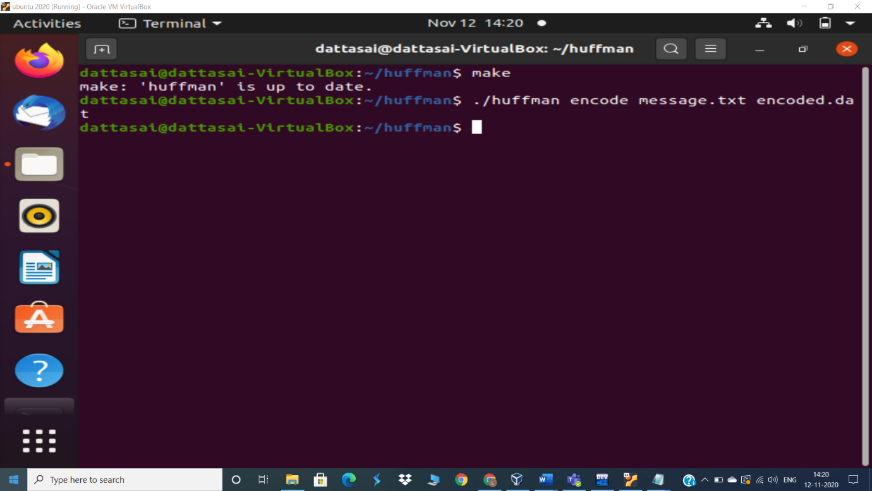
**Huffman:**



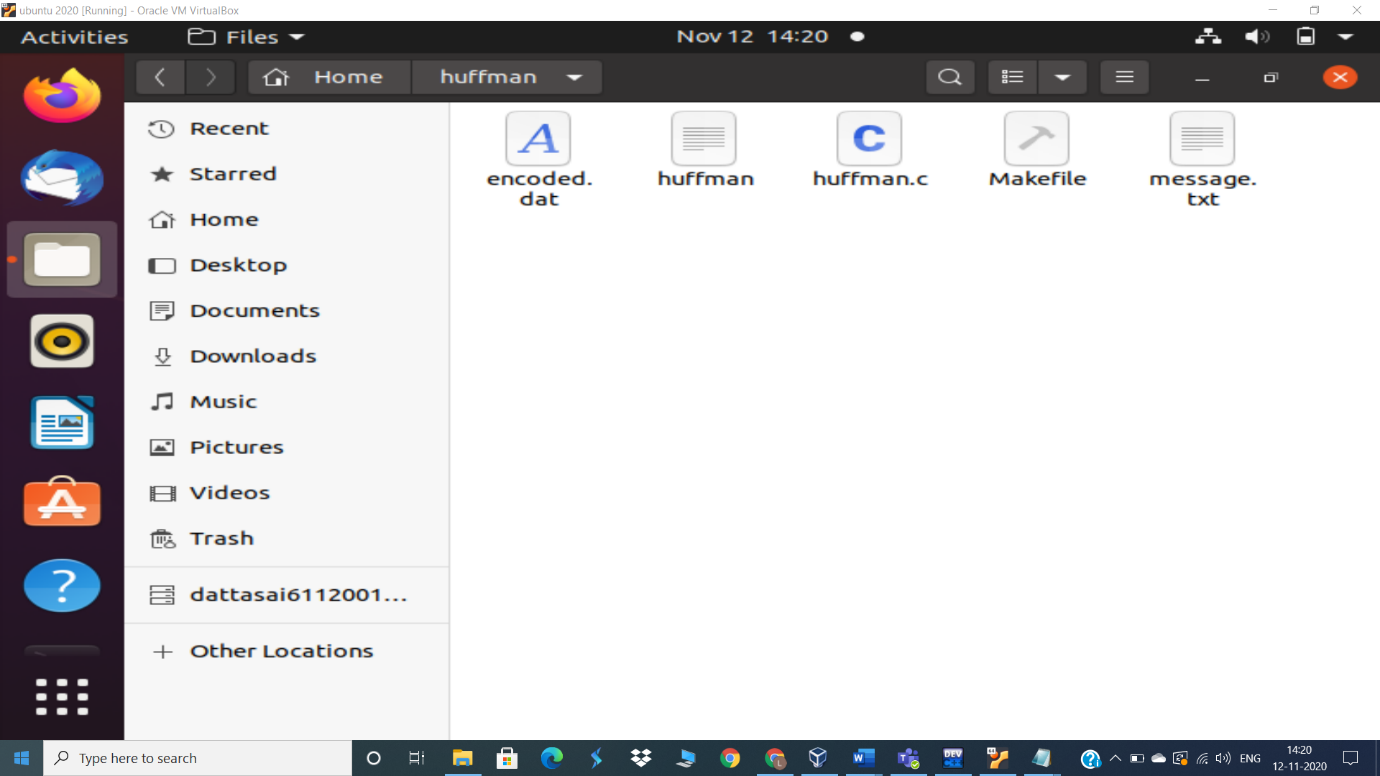
**Figure 13:Intially the files with message.txt**



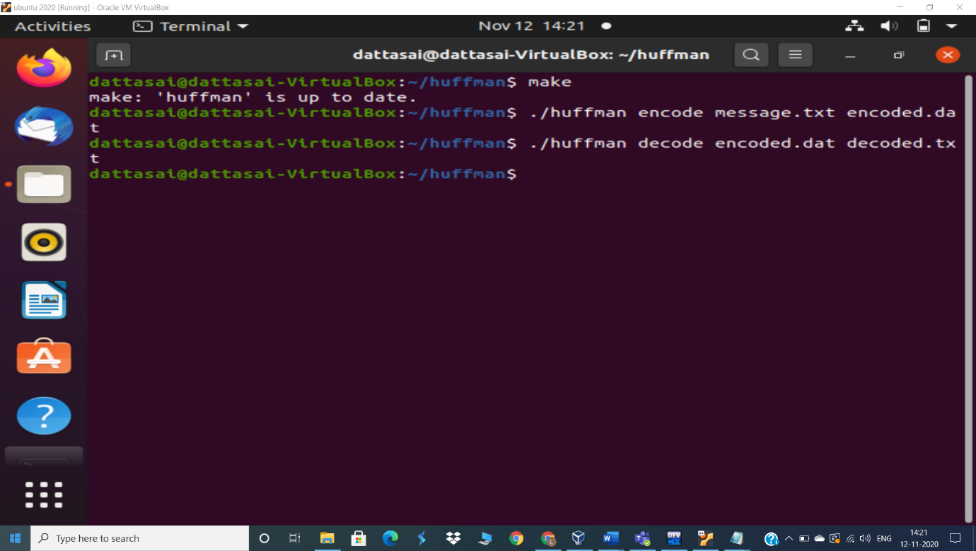
**Figure 14:Contents of message.txt file.**



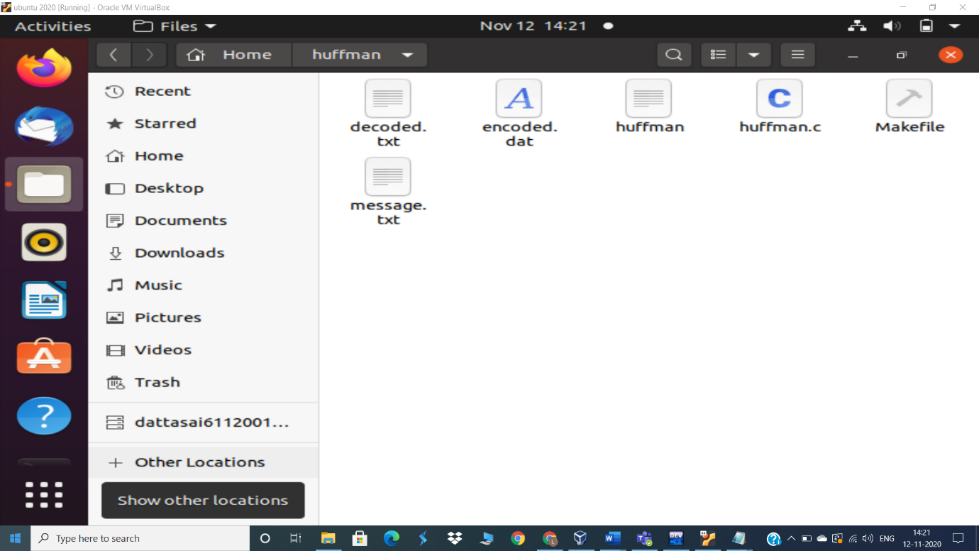
**Figure 15:Usage of encoding command.**



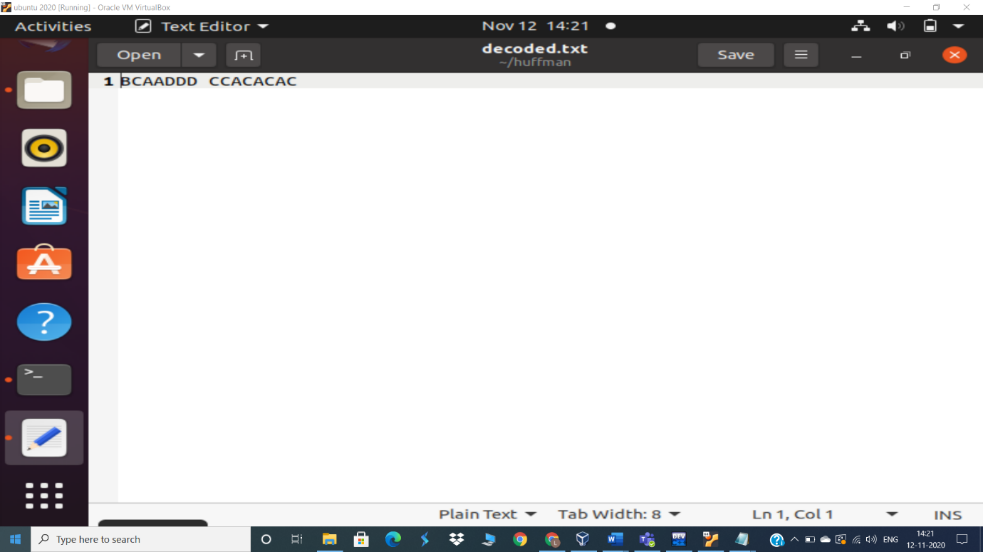
**Figure 16: We can find encoded file highlighted in the box.**



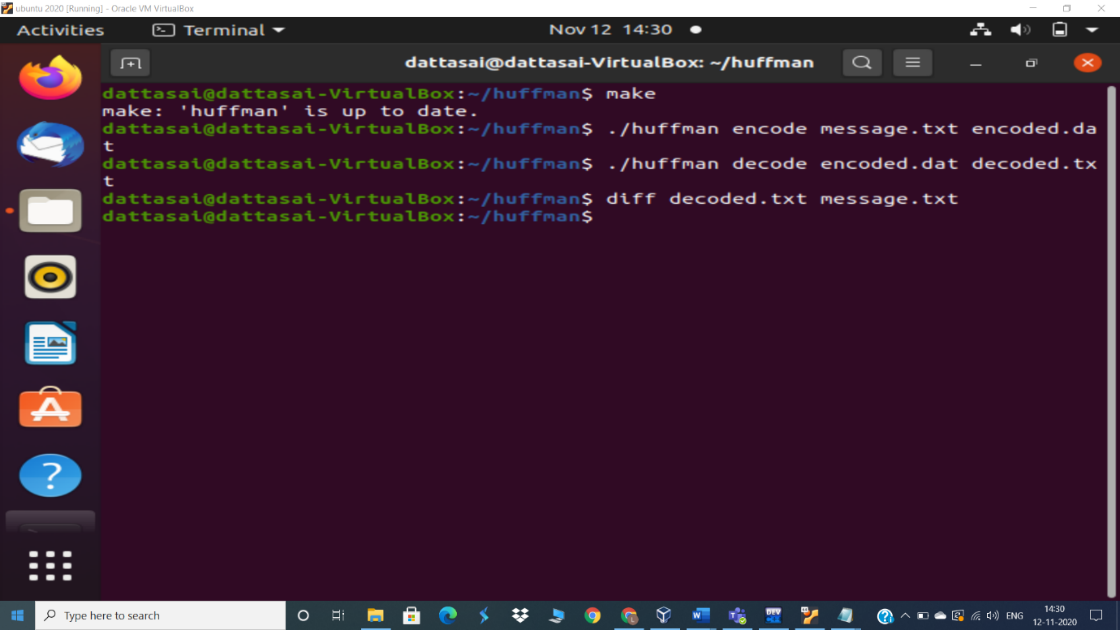
**Figure 17: Using Decompression command.**



**Figure 18:We can see the decoded file now.**

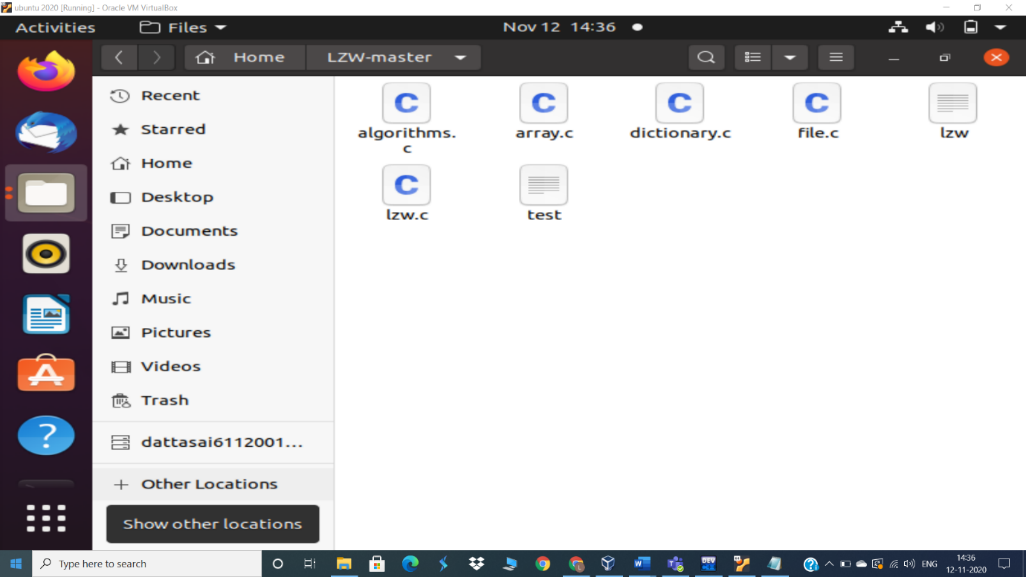


**Figure 19:Contents present in the decoded file.**

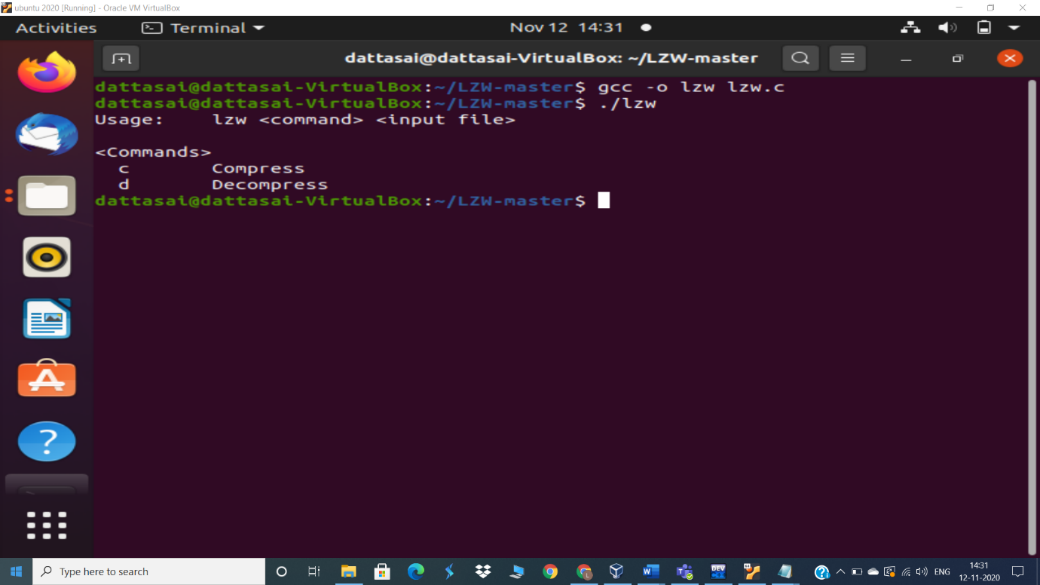


**Figure 20:We can see there is no difference between decoded an the orginal file.**

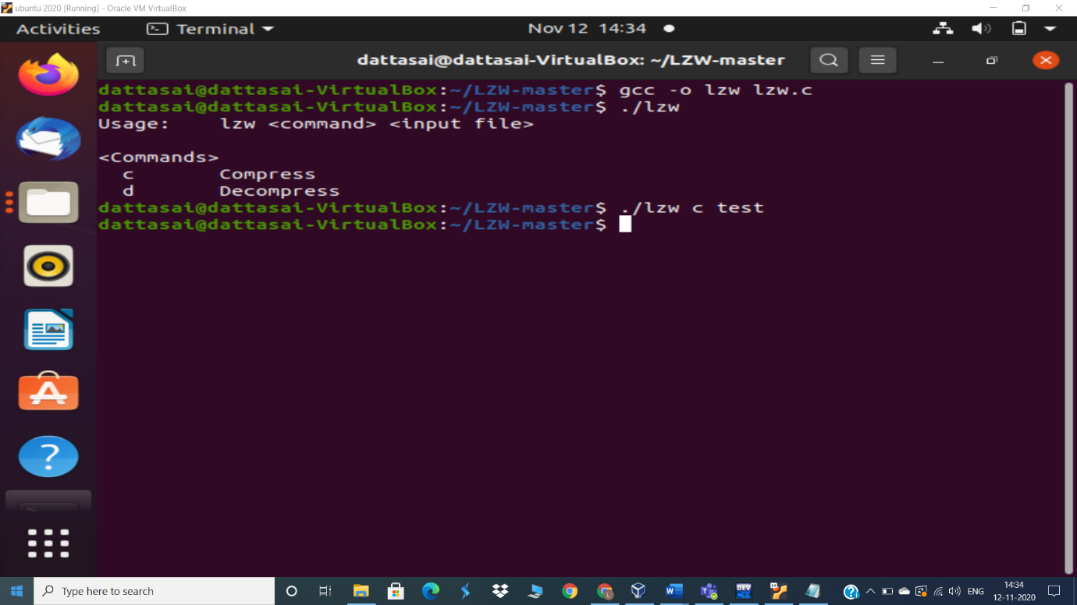
**Lzw :**



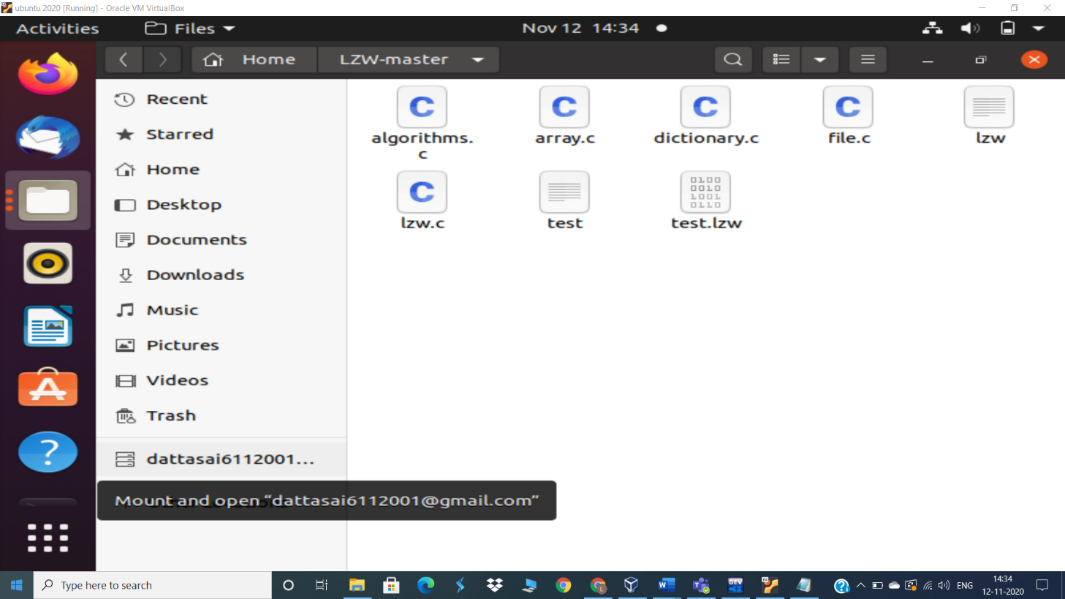
**Figure 21:Files present initially.**

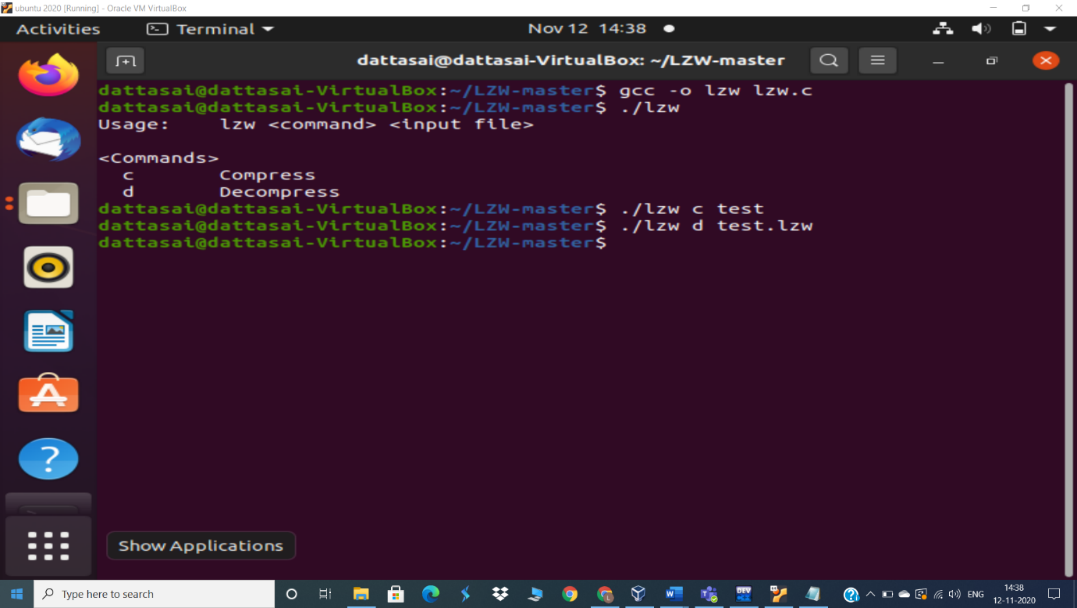


**Figure 22:Compiling and running the LZW file.**

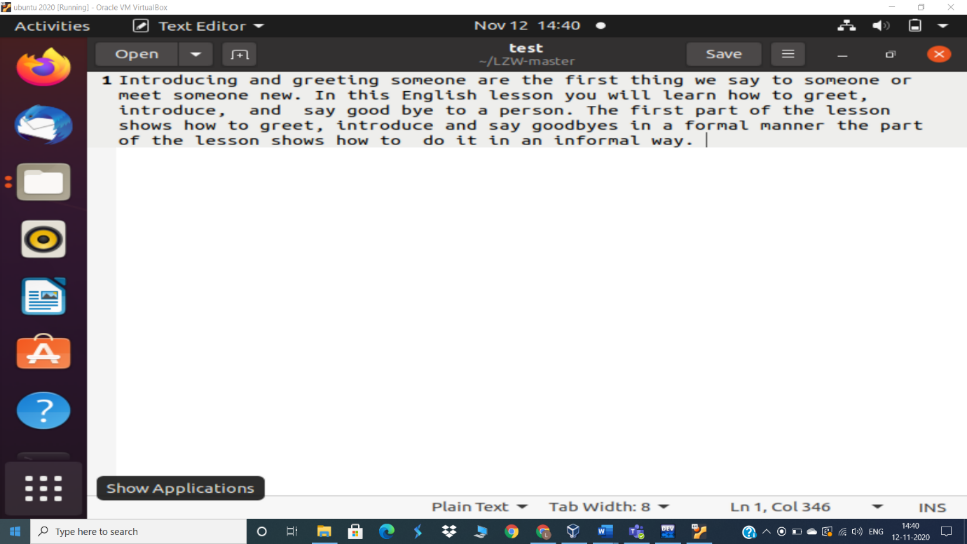


**Figure 23: Compressing the file using compress command.**

 **Figure 24: After compression we can see new test.lzw file.(binary)**



**Figure 25:Decompressing the compressed file.**



**Figure 26: Final contents of the decoded file and the orginal file contents are same as it is lossless compression.**

**Conclusion: -**

* **The original representation has 8 bytes(64 bits) and the new representation have only 9 bits, that is 86% smaller than the original. So the Huffman Coding turns to be a simple and efficient way to encode data into a short representations without loosing any piece of information.**
* I have studied various techniques for compression and compare them on the basis of their use in different applications and their advantages and disadvantages.
* I have concluded that arithmetic coding is very efficient for more frequently occurring sequences of pixels with fewer bits and reduces the file size dramatically.
* LZW algorithm is better to use for TIFF, GIF and Textual Files. It is easy to implement, fast and lossless algorithm whereas Huffman algorithm is used in JPEG compression.
* It produces optimal and compact code but relatively slow. Huffman algorithm is based on statistical model which adds to overhead.

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Huffman coding** | **LZW** |
| **ADVANTAGES** | **1.Easy to implement**  **2. Lossless technique**  **3. Produces optimal**  **and compact code** | **1.Easy to implement**  **2. Fast compression.**  **3. Lossless technique.**  **3. Dictionary based**  **technique.** |
| **DISADVANTAGES** | **1. Relatively slow.**  **2. Depends upon**  **statistical model of**  **data.**  **3. Decoding is**  **difficult due to**  **different code**  **lengths.**  **4. Overhead due to**  **Huffman tree.** | **1. Management of**  **string table is**  **difficult.**  **2. Amount of storage**  **needed is**  **indeterminate.**  **3. Royalties have to**  **be paid to use LZW**  **compression.** |
| **APPL ICATION** | **Used in JPEG.** | **Used in TIFF and**  **GIF files.** |

**Future Work: -**

With the advancements in compression technology, it is now very easy and efficient to compress video files. Various video compression techniques are available. The most common video compression standard is MPEG (Moving Picture Experts Group)[31]. It is a working group of ISO/IEC charged with the development of video and audio encoding standards. MPEG's official designation is ISO/IEC JTC1/SC29 WG11.Many advances are being made for improving the video quality Advancements in MPEG standard are MPEG-1(MP3)[33],MPEG-2,MPEG3,MPEG-4(MPEG-4 Part 2 or Advanced Simple Profile) and MPEG-4 Part 10 (or Advanced Video Coding or H.264). MPEG-7. A formal system for describing multimedia content. MPEG-21 describes this standard as a multimedia framework. MPEG standard is very efficient in the use of DVDs.Various H.261 standards could be used in future for advancement in video conferencing technology Other applied fields that are making use of wavelets in the coming future, include astronomy, acoustics, nuclear engineering, sub-band coding, signal and image processing, neurophysiology, music, magnetic resonance imaging, speech discrimination, optics, fractals, turbulence, earthquake-prediction, radar, human vision, and pure mathematics applications such as solving partial differential equations.

**References:-**

**[1] http://www .H\_261 - Wikipedia, the free encyclopedia.htm**

**[2] http://en.wikipedia.org/wiki/Entropy\_encoding**

**[3] http://www .JPEG - Wikipedia, the free encyclopedia.htm**

**[4] http://www.rz.go.dlr.de:8081/info/faqs/graphics/jpeg1.html**

**[5] http://www.amara.com/IEEEwave/IEEEwavelet.html**

**[6] http://en.wikipedia.org/wiki/Lossless\_JPEG**

**[7]** <http://en.wikipedia.org/wiki/Arithmetic_coding>

**[8]** <https://www2.cs.duke.edu/csed/curious/compression/lzw.html>

**[9]** <https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/#:~:text=It%20is%20the%20algorithm%20of,to%20its%20simplicity%20and%20versatility>**.**

**[10]** <https://www.programiz.com/dsa/huffman-coding>

**THANKYOU**