Chapter 1

A Brief Survey of Data Compression

"Data compression is the art or science of representing information in a compact form" (Introduction to Data 1). Data can take the form of numbers, text, recorded sound, images, and movies. Even the notes that students take for a test or the charts and PowerPoint slides business people use to give presentations are common forms of compressing data. In any case, these "compact forms" are created by identifying unique repetition and other various patterns and structures particular to each medium of data. Though informal compression, like taking notes, has been around for a long time, mass storage of data and the need and ability to compress data did not arrive until the advent of the computer. One reason for this is that "Data compression is of interest in business data processing, both because of the cost savings it offers and because of the large volume of data manipulated in many business applications" (Hirschberg). That is not to say that no compression systems existed before computers, but computers make data compression highly practical.

Two classic early forms of compression that most are familiar with are Morse Code and Braille. Both were developed in the mid-nineteenth century and owe their compression to the statistical structure of the English language. When Samuel Morse developed his system of dashes and dots to send over telegraph wires, he noticed that several of the letters being sent occurred more than others. In order to save time, Morse assigned the more frequent characters such as 'e' and 'a' shorter codes, and less frequent characters such as 'q' and 'j' longer sequences. This concept is the basis behind Huffman encoding, which will be discussed more in depth in the next chapter. The other major compression technique of the time, Braille, not only exploited the frequency of characters, but also took advantage of the frequencies of certain words. Braille coding uses 2 X 3 arrays of dots, two of which are raised (the others are left flat). This results in 2^6 or 64 possible combinations. Twenty-six letters are used in grade one Braille, leaving 38 combinations. In grade two Braille, the remaining combinations frequently represent common words such as “and” and “for.” One combination is used to signal that the next set of dots is a word and not a character, which allows for a larger number of words. “These modifications, along with contractions of some of the words, result in an average reduction in space, or compression, of about 20%” (qtd. in Introduction to Data 1). While frequency characteristics play a major role in many modern text compression algorithms, limitations of human perception are also often exploited in compression of sound and graphics.

Humans experience reality through the five senses with their respective interpretations determined by the human brain. This ability to sense and interpret information, however, is not without limitation. For example, high frequency sounds that dogs can hear are completely imperceptible by the human hear. Thus, any frequencies in sound files or audio transmissions that cannot be heard by the human ear can be omitted with little, if any, perceived loss of quality. A similar case can be made for image information. The eye can distinguish a wide variety of hues, or shades of color, but some colors are so similar that the eye simply cannot perceive the difference. Thus one color could be used in place of two, which would be useful if the compression routine used relied on repletion of pixels.

One may think with the advent of new technologies, such as fiber optics and DVDs, that allow for increased transmission speeds and storage capacity, the need for compression may not be as important as it once was. This assumption could not be further from the truth, for “It seems that the need for mass storage and transmission increases at least twice as fast as storage and transmission capacities improve” (Introduction to Data 3). Many of the technologies that we take for granted, such as the fax machine and modem, would be so slow that they would be impractical in many cases without the use of compression. Another relatively new technology that owes a great deal to compression is High Definition Television (HDTV). Without compression, transmitters would need to transmit 884 Mbits per second, requiring a bandwidth of 220 MHz, but with compression, transmitters only need to transmit less than 20 Mbits per second, requiring only six MHz of bandwidth, the amount allocated for analogue television in the United States (Introduction to Data 2). Modern technology requires compression to work efficiently, which has given rise to several types and variations of data compression.

There are two major types of data compression, lossless and lossy. A, "compression algorithm that takes an input χ and generates a representation χ_c that requires fewer bits, and there is a reconstruction algorithm that operates on the compressed representation χ_c to generate the reconstruction Υ" (Introduction to Data 3). There is no difference between χ and Υ in the case of lossless compression. Υ is similar to χ in the case of lossy compression and deviates from χ in varying degrees depending upon the desired quality. Further discussion of lossless compression will be reserved for later, but it is important to note that both types of compression are often used together to achieve the highest compression ratios, and are sometimes "combined with error correcting codes to provide both compression and data integrity…" (Hirschberg).

"Lossy compression, in contrast [to lossless compression], works on the assumption that the data doesn't have to be stored perfectly" (Goebel), nor restored exactly back to its original state. Distortion is the term used to describe how similar the reconstructed data is to the original. Why would someone want to leave out some information? One reason is that leaving information out means there is less information to be stored. Another reason is simply that not all the information is needed. Consider, again the example of frequencies that humans cannot hear being taken out of sources that contain sound information. Many frequencies and other bits of information can be taken out, and when the information is reconstructed, the sound produced is still intelligible to the human ear. The amount of distortion allowed is generally determined by how much loss of quality can be tolerated. "If the quality of the reconstructed speech is to be similar to that heard on the telephone, a significant loss of information can be tolerated. However, if the reconstructed speech needs to be of the quality heard on a compact disc, the amount of information loss that can be tolerated is relatively low" (Introduction to Data 5). The same holds true for images as well; minor loss of quality for pictures and video often are barely noticeable, so lossy compression is often used when compressing such data. However, as will be seen in the discussion of lossless compression, many situations, including those involving sound and video, cannot tolerate any distortion.

As previously stated, lossless compression involves no loss of data and is generally used on discrete data. While the focus of modern compression was once on lossless compression, "…a significant amount of discrete data in the form of text, graphics, images, video, and audio that needs to be stored or transmitted, and display devices are of such quality that very little distortion can be tolerated" (Lossless Compression). In the case of the text, small discrepancies in the reconstructed text would at the very least be misleading, if not completely unintelligible. If a battle commander were sent a compressed message that said, "Do not go to battle today," but when the message was reconstructed for soldiers in the field, the message said, "Do now go to battle today," heavy casualties could be suffered by the soldiers because a lossy compression threw out information on a single letter. There are cases; however, where lossy compression would yield a respectable replica of the original, but when the data is to be processed or enhanced, the small, seemingly unnoticeable discrepancy becomes much larger. In a compressed radiological image, for example, the radiologist may want a certain area of the image enhanced in order to better diagnose a problem. If the enhancement focused on the one of the previously undetectable differences, the enhanced image would contain serious flaws and could seriously mislead the radiologist, and put someone's life in great jeopardy. This example illustrates the importance of understanding the limitations of a compression algorithm.

In order to be able to appreciate an algorithm's abilities, the abilities must first be measured. There are several ways to measure the performance of a compression algorithm: "the relative complexity of the algorithm, the memory required to implement the algorithm, how fast the algorithm performs on a given machine, the amount of compression, and how closely the reconstruction resembles the original"(Introduction to Data 5). One of these measurements, distortion, has been mentioned previously. Other common terms for distortion include fidelity and quality, and if the fidelity and quality are high, then the reconstructed version is very close to the original. While many of these measurements are beyond the scope of this thesis, the amount of compression is a measure that will be used extensively. One way to measure the amount of compression is to compute the ratio of the number of bits in the original data to the number of bits in the compressed data. "Lossless compression ratios are generally in the range of 2:1 to 8:1" (Hirschberg). For example, suppose that a file requires 95,934 bytes of storage, and after compression, that file occupies only 15,989 bytes of storage. Then the ratio would be 6:1. Another way to measure the amount of compression is the compression rate or, "the average number of bits required to represent a singe sample" (Introduction to Data 5). Continuing with the previous example, let one byte be a single sample, and let there be eight bits per byte. Since the average number of bits per byte of the original is six then the correct terminology would be that, "the rate is six bits per byte."

"Compressing data to be stored or transmitted reduces storage and/or communication costs" (Hirschberg). With its appealing reduction of cost and all around utility, compression has helped create and in some ways made possible the highly technological world people enjoy today. As a catalyst for the storage and transmission of data, compression is and will be an important tool of the information age.

Chapter 2

Huffman Coding

Huffman Coding "…was developed by David Huffman as part of a class assignment; the class was the first ever in the area of information theory and was taught by Robert Fano at MIT" (qtd. in Introduction to Data 27). Before Huffman's new system, the majority of algorithms relied on the fact that some data contained certain distributions or patterns of data that could be exploited, such as the Golomb coding which assumes a geometric distribution (Lossless Compression 27). In order to compress information one would have to use a permutation to achieve the proper distribution necessary for the given algorithm. Since the distributions that are produced from the permutations are unlikely to fit exactly, a certain level of inefficiency is introduced. On top of that, the information of the permutation must also be stored. Huffman presented a huge leap in compression and "was the first to give an exact, optimal algorithm to code symbols from an arbitrary distribution" (qt. in Sayood Handbook 79). Proof of why Huffman is the optimal algorithm for arbitrary distributed data requires several layers of proof that are beyond the scope of this Thesis; however, a thorough explanation of how Huffman coding weaves its compressing ways over arbitrarily distributed code will be included. First, however, a few key definitions and concepts must be understood.

Huffman coding is a particular way of assigning, "binary sequences to elements of an alphabet. The set of binary sequences is called a code and the individual members of the set are called codewords. An alphabet is a collection of symbols called letters"(Introduction to Data 25). As previously mentioned compression ratios and rates are good ways to see how well an algorithm compresses. A good indicator of how much compression will occur is the average length of the code. In the chart and the equation below let a_1, a_2, a_3, a_4 be the letters of a four letter alphabet with the probabilities $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, and $P(a_3) = P(a_4) = \frac{1}{8}$.

Letters	Code 1	Code 2	Code 3	Code 4
a_1	0	0	0	0
a_2	0	1	10	01
a_3	1	00	110	011
a_4	10	11	111	0111
Average length	1.125	1.25	1.75	1.875

(Table 2.1)

"The average length, *l*, for each code is given by

$$l = \sum_{i=1}^{4} P(a_i) \cdot n(a_i) \qquad \text{(Equation 2.1)}$$

where $n(a_i)$ is the number of bits in the codeword for letter a_i and the average length is in bits/symbol." (Introduction to Data 26). Multiplying the number of charaters in a particular message will yield the approximate number of bits after compression.

Consider the four codes in the table above, and examine the properties of each code. The first code yields the lowest average length, but proves not to be useful for coding because a_1 and a_2 have the same codeword, 0. When the reconstruction program goes to decode 0, it will have no way to determine whether the letters a_1 or a_2 were intended. Unlike code 1, code 2 has unique codewords for each of the letters, but it too has problems with ambiguity when it is immersed among other codewords. For example, if the binary string 100 were found in compressed text, the reconstructor could decode it as a_2 a_1 a_1 or a_2 a_3. In other words, it doesn't have unique decodability and isn't distinct. "A distinct code is uniquely decodable if every codeword is identifiable when immersed in a sequence of codewords" (Hirschburg). If tested Codes 3 and 4 would prove to be uniquely decodable. Code 3 has an additional property called the prefix property. “A uniquely decodable code is a prefix code (or prefix-free code) if it has the prefix property, which requires that no codeword is a proper prefix of any other codeword. All uniquely decodable block-block and variable-block codes are prefix codes” (Hirschburg). Now that it is clear what codewords are desirable, lets look at this problem from the letter’s point of view.

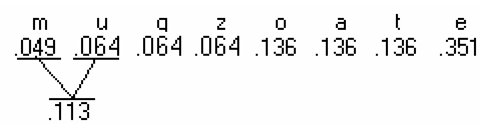
Samples, blocks, letters, and symbols are different ways of describing sections of the original data that is to be compressed. Samples are of no particular length but, at least in this context, cannot be smaller than the smallest unit. In general, the smallest unit is called a letter. Symbols or letters have the potential to be of variable length or of fixed length number of bits depending upon the nature of the data. All the following examples and references to letters will refer to those of fixed length. For example in standard ASCII code seven bits of information are required to represent a character. The character A is coded as 1000001, and the comma is 0011010 just to name a few. The smallest unit of addressable memory is the byte, which is eight bits, so an extra parity bit is often added to the end of the code for purposes of data integrity. Well, if it all comes down to bits anyway; why not pick a letter of size four or five for purposes of compression instead of the letter length (in this case eight)? It is possible to deal with four or five bits at a time, but since every eight bits corresponds to an English letter, punctuation, or symbol, patterns in the English language can be used to effectively compress the letters. While it is unlikely that anything smaller than a letter could present its own unique pattern, it is possible to combine letters together to achieve a better distribution of letters. “Codes that bunch (combine) source element symbols are called block codes. Diagrams and trigrams are examples of block codes. Shannon’s theorem allows for block codes to achieve the lowest possible cost. In most situations, block codes are required to achieve a desired cost" (Sacco 10). The term cost refers to the average block (or word) length. Now that codewords and letters have been defined, it is now time to figure how to match the two together.

Huffman coding uses an ingenious yet simple way to find the codeword for each letter using binary trees. The best way to explain the technique is to illustrate it with an example, so assume that a file has the letters below and that they occur according to the probabilities as indicated. Arrange the probabilities in ascending order.

m	u	q	z	o	a	t	e
.049	.064	.064	.064	.136	.136	.136	.351

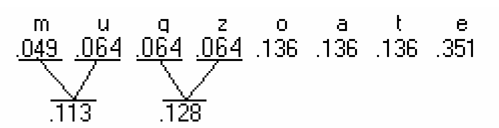
(figure 2.1)

Now combine the lowest two probabilities and give the result its own "node" connected to the to the parent nodes "m" and "u." The new node represents the probability that 'm' or 'u' will be occur if a letter is selected of random from the file.



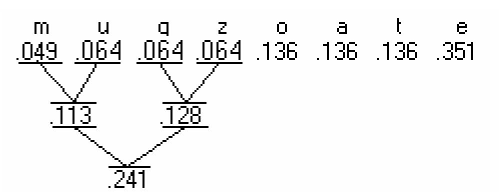
(figure 2.2)

Then do the same with the next two smallest probabilities.



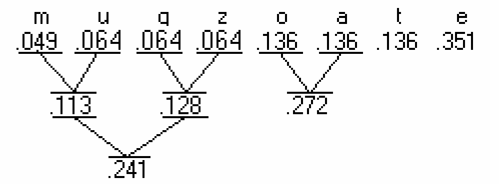
(figure 2.3)

Now note in the next step that it does not matter whether one of the original or one of the new combined probabilities is chosen, so long as it is one of the smallest two probabilities.



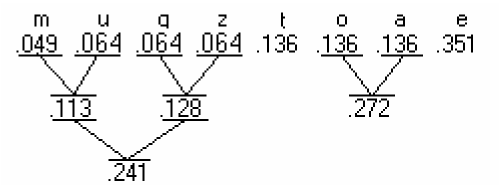
(figure 2.4)

Continue the same process for the next two probabilities.



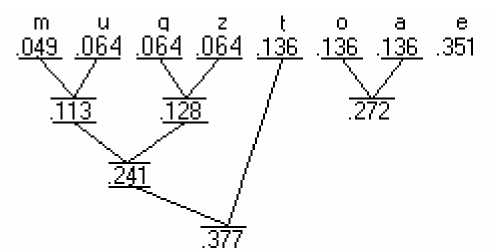
(figure 2.4)

Now we have run into a slight problem. In the previous steps the two lowest probabilities have been next to each other. Now for a computer this would be no problem, but conceptually and in terms of drawing pictures it is simpler just to reorder the probabilities.

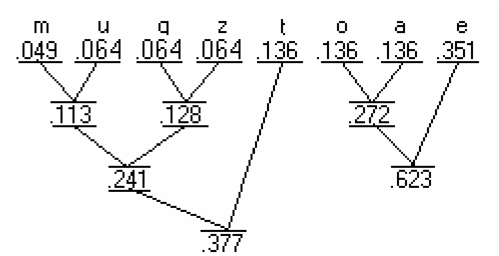


(figure 2.5)

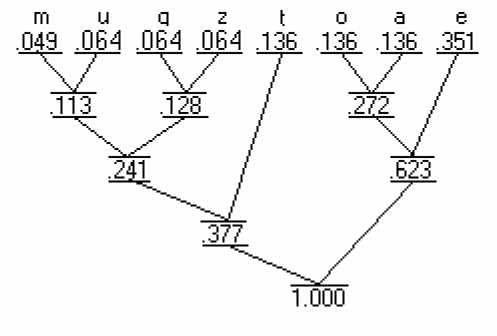
The final three steps are illustrated below.



(figure 2.6)



(figure 2.7)



(figure 2.8)

At the conclusion of this process, what remains is a tree with a trunk, or final node of 1.000, whose upper most leaves are the letters from the file. Note that since the letters were rearranged before step five there are no crossing lines. Note that starting from the trunk, the path to each leaf is unique. Let 'L' denote a left branch and 'R' denote a right branch. The paths to each leaf are listed in the table below.

m	u	q	z	o	a	t	e
L	L	L	L	R	R	L	R
L	L	L	L	L	L	R	R
L	L	R	R	L	R		
L	R	L	R				

(Table 2.2)

Now reading from top to bottom and putting a '1' in place of an 'L' and '0' for 'R.' the following table can be constructed.

m	1111	.049
u	1110	.064
q	1101	.064
z	1100	.064
o	011	.136
a	010	.136
t	10	.136
e	00	.351

(Table 2.3)

Note that the codes formed from this process are prefix codes, and the codes' lengths are directly related to the letter's probability. Since the process started with the lesser probabilities first the lesser probable letters got longer codes, which is part of the key to Huffman coding. Now with these codes let's run through an example of how Huffman would code a file.

Conceptually speaking, the nuts and bolts of how to do Huffman coding are fairly simple. Once the codewords have been created it is a simple matter of replacing each letter with its codeword. For example, if the word "quote" were found in a file containing the letters above in their relative frequencies then it would become "110111100111000." To see the compression the letters are changed to the ASCII binary equivalents and put on top of each other (in the first one spaces where put in to more easily see the letters).

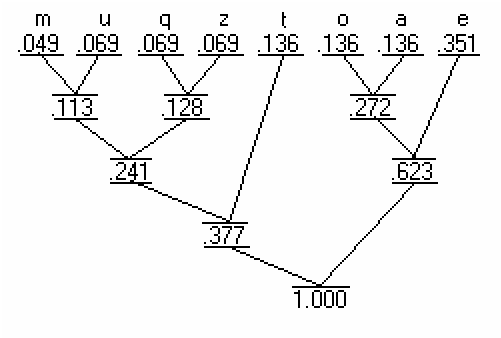
Letter	q	u	o	t	e
ASCII	01110001	01110101	01101111	01110100	01100101
Huffman	1101	1110	011	10	00

(Table 2.4)

Letter	quote
ASCII	0111000101110101011011110111010001100101
Huffman	110111100111000

(Table 2.5)

In the original word there are forty bits and there are only fifteen in the Huffman compressed file, a compression ratio of 8:3 at a rate of 3 bits per byte for this sample. The rest of the file would follow the same pattern. A slight overhead exists because the conversion table must also be stored so that the reconstruction program knows which letters go to which codes. Speaking of reconstruction, the tree that was previously built to determine the codewords can be use to quickly look up the letters.



(figure 2.8)

First start at the 1.000 and read the first bit of the compressed file. The bit is ‘1’, so take a left to .377. The next bit is ‘1’, so take a left to .241. The next two bits ‘0’ and ‘1’ will go to .128 and finally to the letter ’q’. This saves valuable search time over searching a list in which one would have to look at each element until they found the right one. The form of Huffman just described was the original method developed by David Huffman, but since that time other variations of Huffman coding have also been developed.

The category of data compression that the aforementioned coding comes from is called a static method. “A static method is one in which the mapping from the set of messages to the set of codewords is fixed before transmission begins, so that a given message is represented by the same codeword every time it appears in the message ensemble” (Hierschberg). In addition to the classic Huffman method other static variations include Modified Huffman codes, Huffman prefixed Codes, extended Huffman codes, and Length-Constrained Huffman Codes. Each attempts to resolve different types of limitations of classic static Huffman. For example length-constrained Huffman attempts to solve the problem of when the “situation arises when a compression application is severely constrained in time, for example, in multimedia or telecommunication applications, where timing is crucial”(Lossless Compression). As the name implies this version limits the size of the codes that puts an upper bound on the number of steps needed to decode a symbol and also make better use of computer memory. The end result is greater speed of execution. Several of the static methods mentioned also have modified versions, which are dynamic.

In many cases one cannot study the entire set of data, or even significant samples, to make an optimized code set, so the code must be made dynamically. "A code is dynamic if the mapping from the set of messages to the set of codewords changes over time. For example, dynamic Huffman coding involves computing an approximation to the probabilities of occurrence 'on the fly', as the ensemble is being transmitted" (Hierschberg). A dynamic code usually starts off like a static code with a set of codewords, but as new information comes into be coded, new frequencies for letters emerge and the code set is updated accordingly. Depending on the source of data, different levels of "look-ahead" will exist which allow for varying levels of optimization. The brute force adaptive Huffman coding updates the lookup tree every time a new letter is encountered. While adaptive Huffman may achieve the best compression, the time required to update the tree after every character can be prohibitive. Rather than updating the tree consistently it could be done after every k characters. This divides the update cost by k, but will reduce compression efficiency. Another possible technique would be to update the tree only when the relative frequencies in the tree become severely out of balance, but again a balance must be struck with execution time and compression.

Chapter 3

Alternative to Huffman: Prefix Codes That Are Multiples of Three Backwards

Huffman code revolutionized data compression with its optimized prefix code. Huffman coding uses trees to determine the codewords, but there are several other ways that a prefix code can be created. The objective of this chapter is to investigate prefix codes that are multiples of three backwards (with binary), and to discuss various observations and obstacles encountered in the implementation of a Huffman like algorithm that uses these codewords.

One of the first logical steps in dealing with these codewords is first finding a way to generate them. Unfortunately, deriving the codewords directly from binary trees is not very practical as they were with Huffman codes. The easiest way to generate this new code is to try each multiple of three backwards, and if any of the multiples that have already been checked are prefixes of the current codeword being looked at leave the codeword out. Finding the first eight codewords provides one with a good framework to find subsequent codewords. (A copy of the first 29 codewords can be found in Appendix A.) The first multiple of three is three, or 11 in binary. 11 backwards is 11, and since the list is empty 11 is added to the list.

Codewords
11

(Figure 3.1)

The next multiple is 110 (6), which is 011 backwards. The first two digits of 011 is 01 and not 11, so 011 is added to the list.

Codewords
11
011

(Figure 3.2)

The next multiple is 1001 (9) and is the same backwards and forwards. The first three digits of 1001 are not 011 and the first two are not 11, so 1001 is put into the list.

Codewords
11
011
1001

(Figure 3.3)

1100 (12) is the next multiple and is 0011 backwards. The first four digits of 0011 are clearly not 1001. The first three are not 011 and the first two are not 11 so 0011 is the list.

Codewords
11
011
1001
0011

(Figure 3.4)

The next multiple 1111 (15) will not be included in the list. The first two digits are 11, which are the same as the first codeword in the list 11, which violates the prefix-free rule. The next three multiples, 10010, 10101, and 11000, pass the test and are in the list.

Codewords
11
011
1001
0011
01001
10101
00011

(Figure 3.5)

The next multiple 11011 (30) is left out for the same reason 1111 was; the first two digits are 11, which is the same as the first codeword. The eighth item in the list is 100001 (33) and has no prefix of anything in the list.

Codewords
11
011
1001
0011
01001
10101
00011
100001

(Figure 3.6)

This process effectively makes a prefix code, but one may be wondering why the choice was made to reverse the multiples of three.

The choice to reverse the numbers and read them backwards came after some trial and error. The results of this experimentation showed that the size of the numbers of the prefix codes of multiples of three forwards increased faster than the multiples of three backwards. This can be seen in the charts in appendix A. The chart A.1 shows a side-by-side comparison of the codewords and their decimal equivalents, the second is a graph of the decimal equivalents, and the third is a graph of the sizes of the code words. From this chart one can see that not only do the multiples of three backwards have more prefix codewords under a given multiple of three, but the prefix codes for the multiples of three forwards appear to be a proper subset of the prefix codes of the multiples of three backwards (a proof would be required to say that this is true for all cases). An example of the first part of the previous statement would be the number 21, which has five valid prefix codes before it for the backwards multiples, and only two for the forwards multiples. Consequences of this are that more time is required to find codewords by using the forward multiples of three, and, as mentioned before, forward codewords get larger faster than the backwards codewords. The second part of the aforementioned statement is shown by the bold-face numbers in the chart. The first ten codewords of the forwards multiples are clearly elements of the set of codewords of the backwards multiples. The figure A.1 clearly shows that the decimal equivalents of the forward codewards increase at a much faster rate than the backwards codewords. The magnitude of the decimal number has a direct relationship to the number of digits in its binary equivalent, so it is not surprising that in the figure A.2 the length of the codewords of the forwards multiples increase at a greater rate than the backwards multiples. The fact that the backwards multiples produce smaller codewords and thereby achieve better compression was the reason backwards multiples were chosen over the forwards multiples. In addition to this, the first twenty-six codewords of the backwards multiples are all eight bits or under, so there will always be some compression for text containing letters only from a twenty-six letter alphabet. The forwards multiples have more than eight digits after the thirteenth codeword, so for text containing only characters from a twenty six letter alphabet it would be possible to "compress" the text and get a larger file than the original depending on the relative frequency of the letters. Now that the method of finding the codewords and the reasons of why it was chosen has been revealed, it is now time to discuss how this method was implemented.

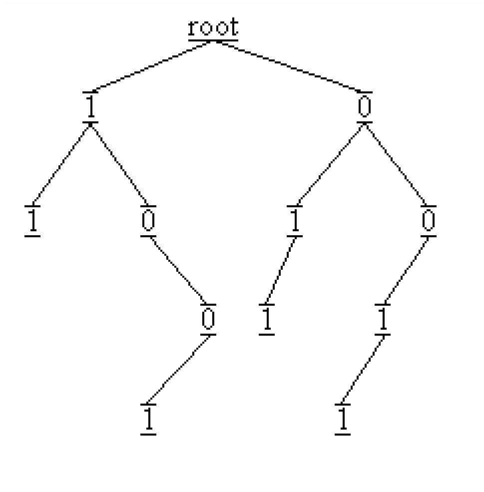
Almost any given algorithm has several different ways to be implemented. The program written in C++ in appendix B reflects one such implementation. An equivalent Huffman version of this program would be the same except the generation of the codewords section, so discussion of the implementation shall be limited to the parts that deal with the codeword generation. The heart of key generation of the codewords lies in the function "generatTempList" in the KeyList class (lines 451 to 482). At first this function can be very confusing because of the way the function (and the entire program) treats the numbers backwards from the way they were described earlier in this chapter. The reasons for treating the codewords in this fashion were to aid in the simplicity of the code. The multiple of three is stored in the variable "key" in its forward form. For example, six would be stored as "110." Rather than reversing the number in memory and storing it as “011” and checking to see if the “11” in the list is the same as the “01,” the function checks to see if the “10” in “110” are the same as “11.” In a sense the computer has generated a postfix code, but treats it backwards it becomes a prefix code. To see this more clearly, the first three steps of the pervious example have been repeated along side a representation of what the computer is actually doing.

Original Example	Multiple Backwards	Multiple Forwards	Computer Example
Codewords 11	011	110	Codewords 11
Codewords 11 011	1001	1001	Codewords 11 011
Codewords 11 011 1001	0011	1100	Codewords 11 011 1001

(Chart 3.1)

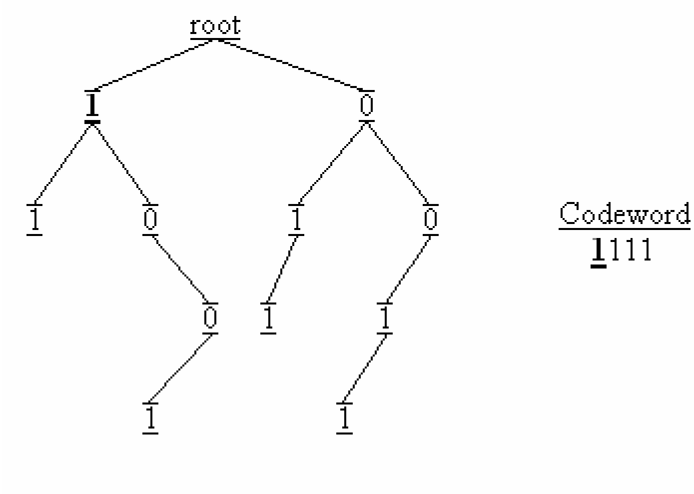
This method can be inefficient when a large number of codewords are to be generated because each item in a list, in this case the list is an array named tempKeyList (line 454), must be checked to see if it is a prefix of the possible codeword. Another possible implementation could use the concept of trees.

Due to the nature of prefix codes, they lend themselves well to trees. As can be seen in table 2.2 each leaf node has a unique path from the root, which can be used to determined it one code is a prefix of another or not. The function “generateLetterSearchTree” (lines 580-624) uses this fact to generate a search tree. In fact the search tree and the codeword generator could have been made at the same time, but due to the order in which the program was developed it was easier to leave it separate. Below is picture of a tree with the first four codewords already added.

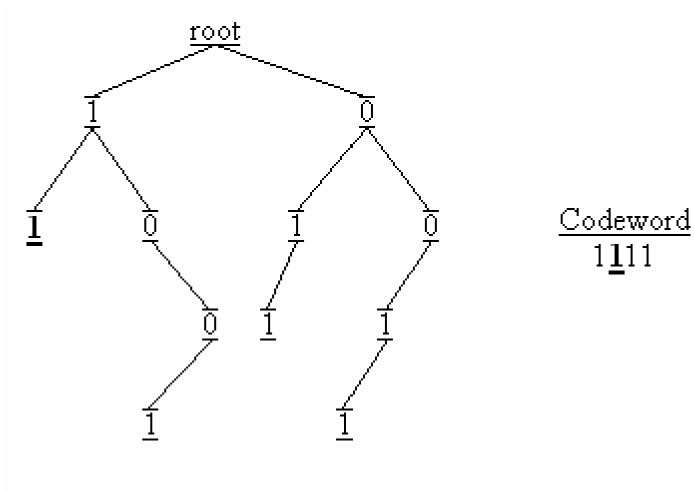


(Figure 3.7)

When the next backward multiple, “1111” (15), is checked to see if it is a valid codeword it would only need to check two nodes to see that it was invalid rather than checking to see if the four elements in the list were prefixes. Even though in this case the first element disqualified the codeword, a considerable amount of work is involved in checking to see if one codeword is a prefix of another. The function “isAaPostFixOfB” (lines 522-529) is responsible for checking to see if one codeword is a prefix of another codeword. On the surface this function seems short and simple, but on line 524 “isAaPostFixOfB” calls two functions, “getNumBits” (532-542) and “getSubBitStringFrom1To” (544-552), that have about twenty lines of code between them. “isAaPostFixOfB” is called once for each element in the list until a prefix is encountered or until the end of the list is encountered. As more codewords are put into the list, this process can become very inefficient. Returning to the example, the figures below show that only two nodes need to be checked to see that “1111” is an invalid codeword.

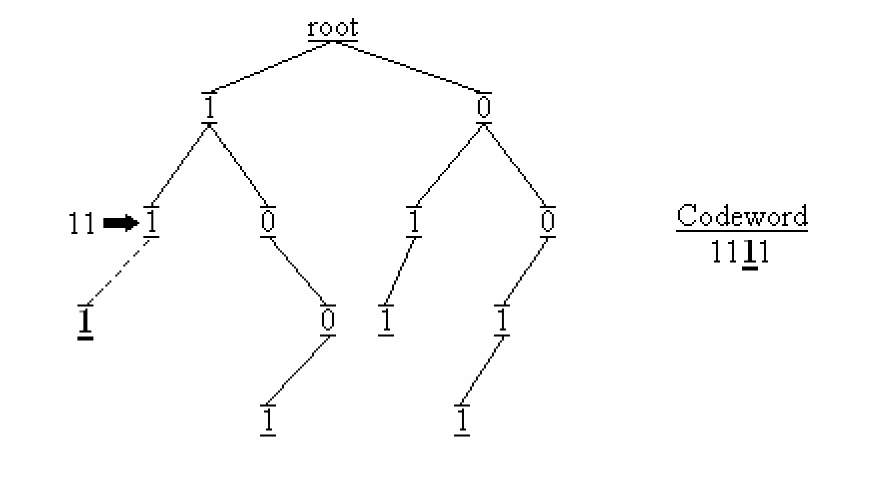


(Figure 3.8)



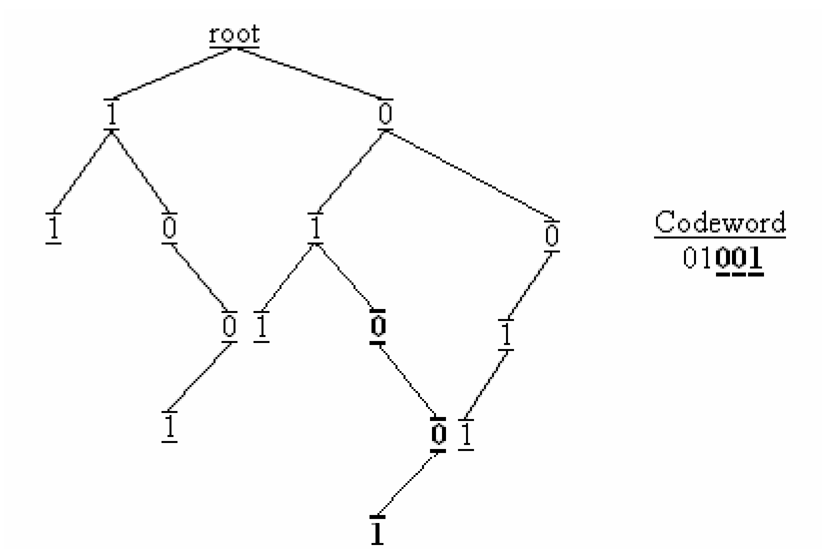
(Figure 3.9)

If another node were added (see figure 3.10) then the codeword “1111” would have the same initial path as “11,” so “1111” cannot be a codeword because the computer would not know whether or not to stop after the second “1.”



(Figure 3.10)

The next codeword, however, clearly has a unique path.



(Figure 3.11)

Basically, the rule is nodes can be added to other nodes to accommodate any new codeword unless the node is a leaf-node. If the node being added to is a leaf-node, leave out the codeword and try to find the next one.

Chapter 4

Huffman vs. Multiples of Three Backwards

Comparing compression algorithms requires knowledge of several aspects, each requiring its own test. These tests where mentioned in chapter one, and all are important in choosing the correct algorithm for a particular type of data, but only two major ones will be compared. The average codeword length and compression ratio are generally held to be good indicators of the amount of compression, so those will be used to see how the multiples of three backwards fare against Huffman code.

The average codeword length holds importance because it is often a good indication of how good the compression ratio is. This is often more useful in adaptive compression methods that deal with streams of data where it may be impractical to calculate the actual compression ratio on the entire set of data. Instead of indicating the compression ratio of the data as a whole, it gives a good indication of the amount of compression occurring at that section of data. With the static methods that are being compared here, the average codeword lengths are excellent indicators when the entire set of data can be analyzed, such as in a file on a computer. The average codeword length provides a good indication of the relative effectiveness of different codeword sets because it possesses an inverse relationship with the compression ratio; the lower the average code length, the higher and better the compression ratio. Using a formula similar to equation 2.1 in chapter two, chart B.2 in appendix B was created. The third coulomb in each table represents the multiplication of the percentage frequency of each text character and the codeword length for that particular text character. Then the third columns where added up and then divided by 100 since percentage frequencies need to be converted to relative frequencies. Huffman Codes, Multiples of three forwards, and Multiples of three backwards had average code word lengths of 4.20502, 4.92234, and 6.32809 respectively. Clearly given any set of data the Huffman codes would yield the best compression on a general set of text. The Huffman Code for any given letter in a text it is more likely to result in a shorter codeword the resulting codeword when multiples of three backwards or backwards is used. Over the course of compressing the whole text the savings of each letter builds upon each other and yields an overall better compression. As mentioned in chapter two the average codeword length can be used to approximate the length of the compressed text. Simply multiplying average codeword length by the number of letters in the text, yields the approximate length of the compressed text. The reason why it is an approximate solution is because the relative frequencies of the letters are often approximations or may only reflect a section of the data rather than the data as a whole. For example the percentage frequencies in chart B.1 do not add up to exactly one hundred percent since percentage frequencies were rounded to three decimal places. It is possible to have the computer keep track of all the decimal places (as much as memory will allow), but this may be prohibitive because it could have serious effects upon performance with regards to speed, and memory usage. However, if the actual percentage frequencies can be calculated exactly without round-off error, then the average codeword length and compressed files size differ by a factor equal to the number of characters in the original file. This observation can be seen by comparing charts B.2 and B.3. The average codeword lengths in B.2 and the compressed file sizes in B.3 differ by a factor of 100,000. The actual file size is 99999, but this discrepancy is due to round-off error.

The compression ratio is one of the simplest aspects of data compression and is one of the easiest to calculate. Chart B.3 shows the compression ratios of the three sets of codewords on a theoretical set of data. The theoretical data contains 99999 letters, and since each letter corresponds to 8 bits the data contains 799992 bits. By multiplying the frequency of occurrences of particular letter, by the number of bits that letter uses in the compressed file results in the total numbers of bits that perticular letter takes up in the compressed file. For example the letter ‘e’ occurs 12702 times in the uncompressed file, and each ‘e’ is represented by 3 bits with the Huffman code. By multiplying the number of e’s by the number of bits it takes up one finds that ‘e’ takes up 38106 bits of the compressed file when the Huffman code is used. Doing this for every letter in the alphabet and adding the respective number of bits will produce the total number of bits in the compressed file. In the case of the theoretical file, Huffman codes compress the file down to 420502 bits, backwards codewords, 492234 bits, and forwards codewords, 632809 bits. Taking these numbers and dividing them by the size of the original file yield the compression ratios 1.90246, 1.62522, and 1.26419 respectively. Huffman’s compression ratio is almost two, so that means that the original file is almost twice the size of the compressed file, or the compressed file is about half the size of the original. The original file size is about one and a half times the file compressed by multiples of three backwards codewords and about one and a quarter times the file compressed by the multiples of three forwards codewords.

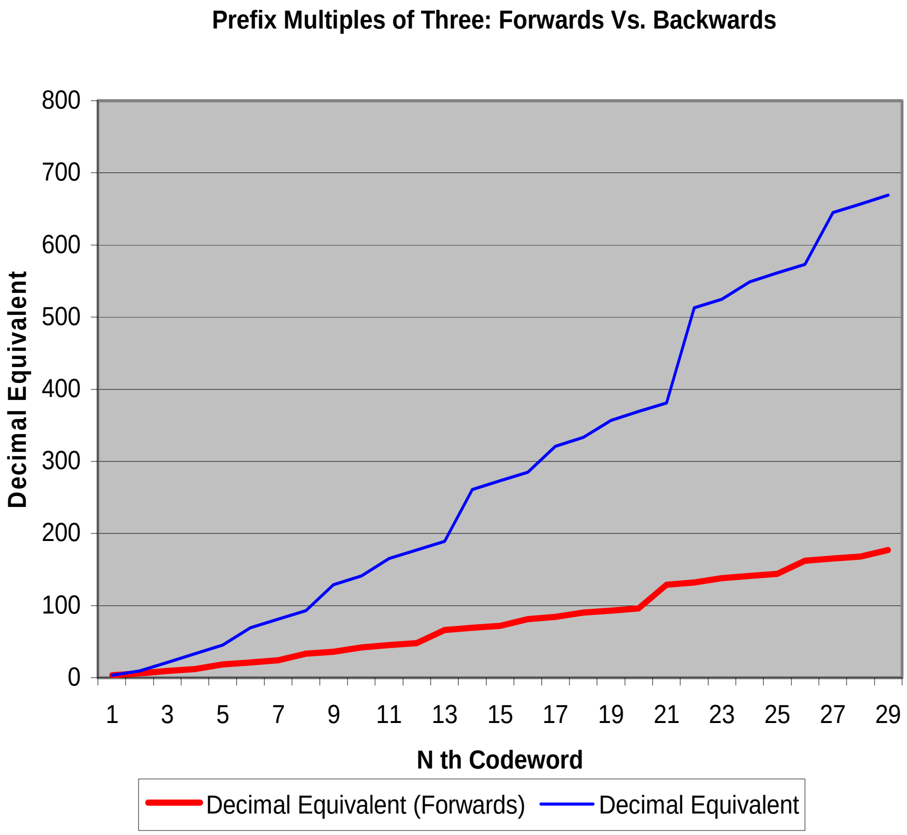
Clearly Huffman has proven, in general, to be the best of the three prefix codes. The others may prove to have purposes that suit them better than Huffman, but for now Huffman still remains the best for general data. The multiples of three backwards code proved to be a worthy endeavor and helped me to develop a better understanding of Huffman and data compression in general. The study of data compression, including the methods studied here, will continue converting long elements of the past into smaller elements of the future.

Appendix A

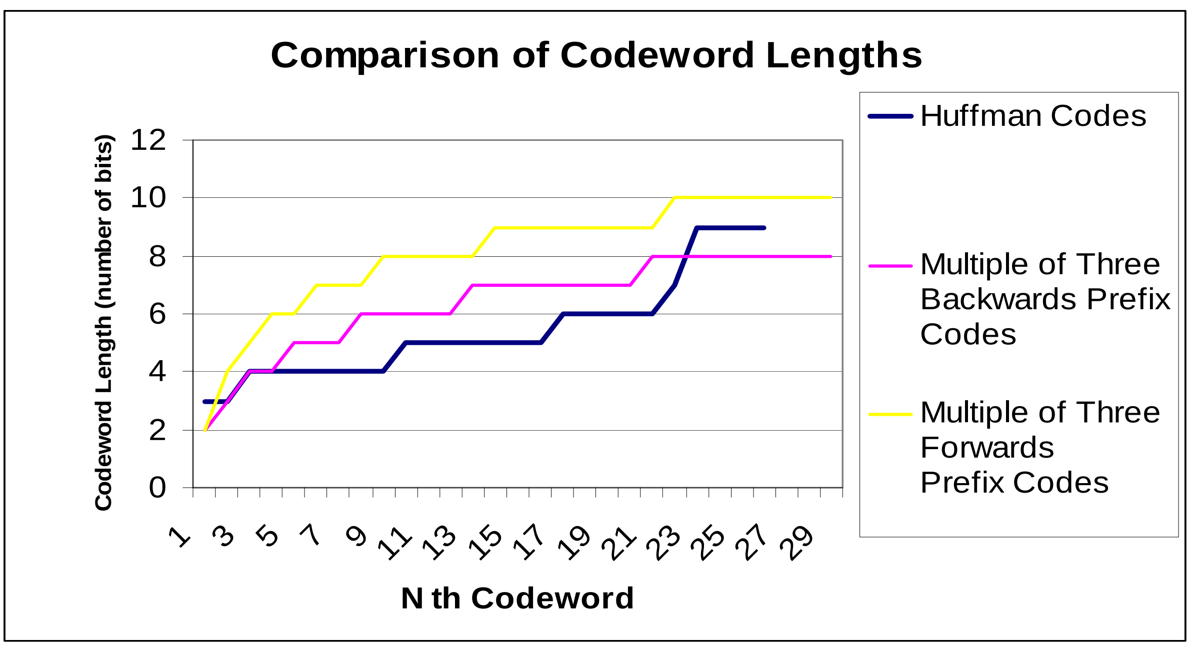
Huffman Codes*	Multiple of Three Backwards Prefix Codes	Decimal Equivalent (Forwards)	Multiple of Three Forwards Prefix Codes	Decimal Equivalent
000	**11**	3	**11**	3
110	011	6	**1001**	9
0100	**1001**	9	**10101**	21
0110	**0011**	12	**100001**	33
0010	01001	18	**101101**	45
0011	10101	21	**1000101**	69
1000	00011	24	**1010001**	81
1001	**100001**	33	**1011101**	93
1010	001001	36	**10000001**	129
01010	010101	42	**10001101**	141
01011	**101101**	45	10100101	165
10110	000011	48	10110001	177
10111	0100001	66	10111101	189
11100	1010001	69	100000101	261
11101	0001001	72	100010001	273
11110	**1000101**	81	100011101	285
011100	0010101	84	101000001	321
011101	0101101	90	101001101	333
011110	**1011101**	93	101100101	357
011111	0000011	96	101110001	369
111110	**10000001**	129	101111101	381
1111110	00100001	132	1000000001	513
111111100	01010001	138	1000001101	525
111111101	10110001	141	1000100101	549
111111110	00001001	144	1000110001	561
111111111	01000101	162	1000111101	573
	10100101	165	1010000101	645
	00010101	168	1010010001	657
	10001101	177	1010011101	669

*Huffman Codes Come From Appendix B

(Chart A.1)



(Figure A.1)



(Figure A.2)

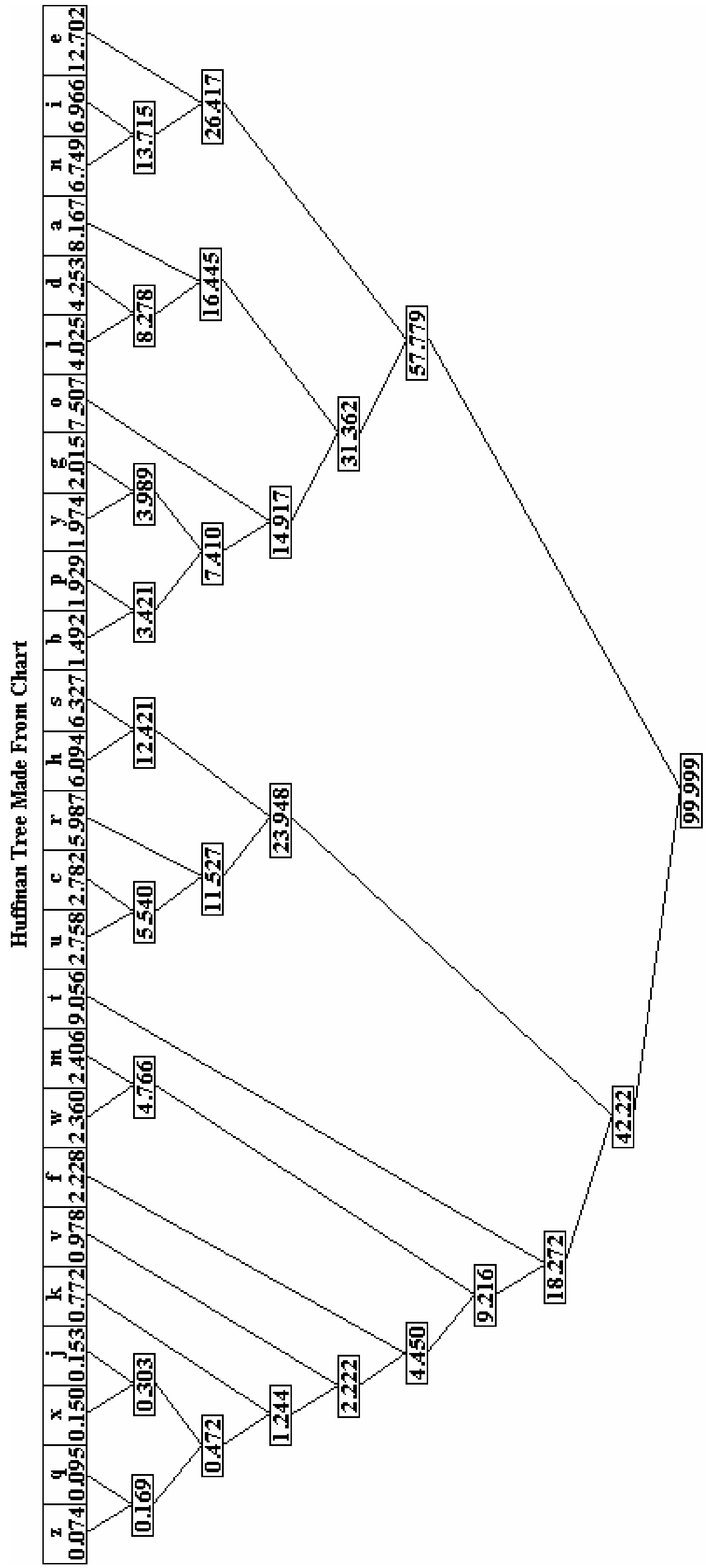
Appendix B

Relative Frequencies of the Letters of the English Language

Letter	Relative Frequency (%)*	Huffman Codes		Letter	Relative Frequency (%)*	Huffman Codes
a	8.167	0100		e	12.702	000
b	1.492	011111		t	9.056	110
c	2.782	10110		a	8.167	0100
d	4.253	01010		o	7.507	0110
e	12.702	000		i	6.966	0010
f	2.228	11110		n	6.749	0011
g	2.015	011100		s	6.327	1000
h	6.094	1001		h	6.094	1001
i	6.966	0010		r	5.987	1010
j	0.153	111111100		d	4.253	01010
k	0.772	1111110		l	4.025	01011
l	4.025	01011		c	2.782	10110
m	2.406	11100		u	2.758	10111
n	6.749	0011		m	2.406	11100
o	7.507	0110		w	2.36	11101
p	1.929	011110		f	2.228	11110
q	0.095	111111110		g	2.015	011100
r	5.987	1010		y	1.974	011101
s	6.327	1000		p	1.929	011110
t	9.056	110		b	1.492	011111
u	2.758	10111		v	0.978	111110
v	0.978	111110		k	0.772	1111110
w	2.360	11101		j	0.153	111111100
x	0.150	111111101		x	0.15	111111101
y	1.974	011101		q	0.095	111111110
z	0.074	111111111		z	0.074	111111111

*0.001% round-off error

(Chart B.1)


Huffman Tree Made From Chart
z 0.074
q 0.095
x 0.150
j 0.153
k 0.772
v 0.978
f 2.228
w 2.360
m 2.406
t 9.056
u 2.758
c 2.782
r 5.987
h 6.094
s 6.327
b 1.492
p 1.929
y 1.974
g 2.015
o 7.507
l 4.025
d 4.253
a 8.167
n 6.749
i 6.966
e 12.702
0.169
0.303
0.472
1.244
2.222
4.450
4.766
9.216
18.272
42.22
5.540
11.527
12.421
23.948
3.421
3.989
7.410
14.917
8.278
16.445
31.362
13.715
26.417
57.779
99.999

(Figure B.1)

Average Codeword Lengths

Huffman Codes		
Relative Freq.	Codeword Len. (bits)	Freq.* Length
12.702	3	38.106
9.056	3	27.168
8.167	4	32.668
7.507	4	30.028
6.966	4	27.864
6.749	4	26.996
6.327	4	25.308
6.094	4	24.376
5.987	4	23.948
4.253	5	21.265
4.025	5	20.125
2.782	5	13.91
2.758	5	13.79
2.406	5	12.03
2.36	5	11.8
2.228	5	11.14
2.015	6	12.09
1.974	6	11.844
1.929	6	11.574
1.492	6	8.952
0.978	6	5.868
0.772	7	5.404
0.153	9	1.377
0.15	9	1.35
0.095	9	0.855
0.074	9	0.666
Average Code Length		4.20502

Multiples of Three Backwards		
Relative Freq.	Codeword Len. (bits)	Freq.* Length
12.702	2	25.404
9.056	3	27.168
8.167	4	32.668
7.507	4	30.028
6.966	5	34.83
6.749	5	33.745
6.327	5	31.635
6.094	6	36.564
5.987	6	35.922
4.253	6	25.518
4.025	6	24.15
2.782	6	16.692
2.758	7	19.306
2.406	7	16.842
2.36	7	16.52
2.228	7	15.596
2.015	7	14.105
1.974	7	13.818
1.929	7	13.503
1.492	7	10.444
0.978	8	7.824
0.772	8	6.176
0.153	8	1.224
0.15	8	1.2
0.095	8	0.76
0.074	8	0.592
Average Code Length		4.92234

Multiples of Three Forwards		
Relative Freq.	Codeword Len. (bits)	Freq.* Length
12.702	2	25.404
9.056	4	36.224
8.167	5	40.835
7.507	6	45.042
6.966	6	41.796
6.749	7	47.243
6.327	7	44.289
6.094	7	42.658
5.987	8	47.896
4.253	8	34.024
4.025	8	32.2
2.782	8	22.256
2.758	8	22.064
2.406	9	21.654
2.36	9	21.24
2.228	9	20.052
2.015	9	18.135
1.974	9	17.766
1.929	9	17.361
1.492	9	13.428
0.978	9	8.802
0.772	10	7.72
0.153	10	1.53
0.15	10	1.5
0.095	10	0.95
0.074	10	0.74
Average Code Length		6.32809

(Chart B.2)

Expected Compression Size
(Of A File With 99,999 Characters)

Huffman Codes		
Number of Characters	Compress Len.(bits)	Number * Length
12702	3	38106
9056	3	27168
8167	4	32668
7507	4	30028
6966	4	27864
6749	4	26996
6327	4	25308
6094	4	24376
5987	4	23948
4253	5	21265
4025	5	20125
2782	5	13910
2758	5	13790
2406	5	12030
2360	5	11800
2228	5	11140
2015	6	12090
1974	6	11844
1929	6	11574
1492	6	8952
978	6	5868
772	7	5404
153	9	1377
150	9	1350
95	9	855
74	9	666
Average Code Length		420502
Compression Ratio		1.90246

Multiples of Three Backwards		
Number of Characters	Compress Len.(bits)	Number * Length
12702	2	25404
9056	3	27168
8167	4	32668
7507	4	30028
6966	5	34830
6749	5	33745
6327	5	31635
6094	6	36564
5987	6	35922
4253	6	25518
4025	6	24150
2782	6	16692
2758	7	19306
2406	7	16842
2360	7	16520
2228	7	15596
2015	7	14105
1974	7	13818
1929	7	13503
1492	7	10444
978	8	7824
772	8	6176
153	8	1224
150	8	1200
95	8	760
74	8	592
Average Code Length		492234
Compression Ratio		1.62522

Multiples of Three Forwards		
Number of Characters	Compress Len.(bits)	Number * Length
12702	2	25404
9056	4	36224
8167	5	40835
7507	6	45042
6966	6	41796
6749	7	47243
6327	7	44289
6094	7	42658
5987	8	47896
4253	8	34024
4025	8	32200
2782	8	22256
2758	8	22064
2406	9	21654
2360	9	21240
2228	9	20052
2015	9	18135
1974	9	17766
1929	9	17361
1492	9	13428
978	9	8802
772	10	7720
153	10	1530
150	10	1500
95	10	950
74	10	740
Average Code Length		632809
Compression Ratio		1.26419

(Chart B.3)

Appendix C

An Implementation in C++ of a Huffman style algorithm with codewords of prefix-free multiples of three backwards

```
                              //Runner of Three.cpp
/*
Huffman style compresion
Author: Jeremy Brown
Language: C++
Compiler: Microsoft Visual C++ 6.0 Compiler Introductory Edition
*/

#include <iostream.h>
#include <conio.h>
#include "H3Compressor.h"

int main()
{
       H3Compressor fileCompressor;
       fileCompressor.compressFile();
       fileCompressor.displayStatistics();
       fileCompressor.decompressFile();
       return 0;
}
```

```
                                //H3Compressor.h
/*
Jeremy Brown
Thesis Poject
H3Compressor class
Last Updated: 2/15/04
*/

#ifndef h3compressor_h_
#define h3compressor_h_

#include <iostream.h>
#include <fstream.h>
#include "KeyList.h"

/*
H3Compressor compresses or decompress a file with a variation of Huffman which uses prefix codes of multiples of three backwards
*/


class H3Compressor
{
private:

	KeyList keyList;
	bool mode; //true for compresstion. false for decompretion

	//statistics for compresstion
	unsigned long int originalFileSizeBytes;
	unsigned long int compressedFileSizeBits;

	//File stuff
	char* sourceFileName;
	fstream sourceFile;
	fstream outputFile;
	void resetSourceFile();
	char* getFileFrequency(int &charaterListSize);
public:
	//general functions
	H3Compressor();
	char* getNameFromUser();
	void prepareFiles(char* fileName);
	void closeFiles();

	//compress functions
	void compressFile();
	int findLetter(char letter,char* charaterList);
	void displayStatistics();

	//decompress functions
	void decompressFile();
};

H3Compressor::H3Compressor()
{
	mode = true;
	originalFileSizeBytes = 0;
	compressedFileSizeBits = 0;
}

void H3Compressor::compressFile()
```

```
{
        mode = true;
        prepareFiles(getNameFromUser());
        char* frequencyList = 0;
        int frequListSize = 0;
        frequencyList=getFileFrequency(frequListSize);
        keyList.generateKeyList(frequListSize);
        unsigned char* tempString = new unsigned char[2];
        //output the size of the dictionary
        tempString[0] = frequListSize;
        outputFile.write(tempString,1);
        //output the dictionary
        for(int letterCount = 0;letterCount<frequListSize;letterCount++)
        {

                tempString[0] = frequencyList[letterCount];
                outputFile.write(tempString,1);
        }
        char* byte = new char[1];
        int shift = 0;
        char* key = 0;
        BitIndex keyLength;
        tempString[0] = 0;tempString[1]=0;
        sourceFile.read(byte,1);
        while(sourceFile.gcount() !=0)
        {       //find the codeword for the letter
                key = keyList.getKey(findLetter(byte[0],frequencyList)).key;
                keyLength = 
keyList.getKey(findLetter(byte[0],frequencyList)).keySize;
                compressedFileSizeBits += (keyLength.getIndex() * 8) + 
keyLength.getOffset();
                char temp = 138;
                //output all the complete bytes of the codeword
                for(unsigned int index = 0;index<keyLength.getIndex();index++)
                        {
                                tempString[0] = (tempString[0] | (key[index] << 
shift));
                                tempString[1] = ~(((~((unsigned char)0))<<shift) | 
(~(tempString[1] | 
                                        (key[index]>> (8 - shift)))));
                                outputFile.write(tempString,1);
                                tempString[0] = tempString[1];
                                tempString[1] = 0;
                        }
                //output any remaining bits
                if(keyLength.getOffset()!=0)
                {
                        tempString[0] = tempString[0] | (key[keyLength.getIndex()] 
<< shift);
                        tempString[1] = tempString[1] | (key[keyLength.getIndex()] 
>> (8 - shift));
                        shift += keyLength.getOffset();
                        if(shift > 7)
                        {
                                outputFile.write(tempString,1);
                                tempString[0] = tempString[1];
                                tempString[1] = 0;
                                shift = shift%8;
                        }
                }
                //read the next charater out of the file
                sourceFile.read(byte,1);
        }
```

```
	//output any remainding bits
	if(shift != 0)
	{
		outputFile.write(tempString,1);
	}
	delete tempString;
	delete frequencyList;
	closeFiles();
}

char* H3Compressor::getNameFromUser()
{
	char* fileName = new char[20];
	if(mode)
	{
		cout<<"Please input the file you wish to compress. ";
	}
	else
	{
		cout<<"Plaese input the file you wish to decompress. ";
	}
	cin.getline(fileName,20);
	return fileName;
}

void H3Compressor::prepareFiles(char* fileName)
{
	sourceFileName = fileName;
	if(mode)
	{//Open source in text mode if compressing
		sourceFile.open(fileName,ios::in);
		if(!sourceFile)
		{
			cout<<"Failed to open sourceFile"<<endl;
			exit(0);
		}
	}
	else
	{//Open source in binary mode if decompressing
		sourceFile.open(fileName,ios::in | ios::binary);
		if(!sourceFile)
		{
			cout<<"Failed to open sourceFile"<<endl;
			exit(0);
		}
	}
	if(mode)
	{//Open output in binary mode if compressing
		const char* CompressedFile = "Compressed File";
		outputFile.open(CompressedFile,ios::out | ios::trunc |
ios::binary);
		if(!outputFile)
		{
			cout<<"Failed to open outputFile"<<endl;
			exit(0);
		}
	}
	else
	{//Open output in text mode if decompressing
		const char* DecompressedFile = "Decompressed File.txt";
		outputFile.open(DecompressedFile,ios::out | ios::trunc );
		if(!outputFile)
		{
```

```
                        cout<<"Failed to open outputFile"<<endl;
                        exit(0);
                }
        }
}

void H3Compressor::closeFiles()
{
        sourceFile.close();
        outputFile.close();
}

char* H3Compressor::getFileFrequency(int &charaterListSize)
{//the function retrun a list of letter in order of the most frequent (index 0) to the least
        //frequent
        char* charaterList = 0;
        charaterListSize = 0;
        unsigned char* byte =  new unsigned char[1];
        unsigned long int tempFrequencyList[256];
        for(unsigned long int i = 0;i<256;i++)
        {
                tempFrequencyList[i] = 0;
        }
        sourceFile.read(byte,1);
        while(sourceFile.gcount() !=0)
        {
                if(tempFrequencyList[(int)byte[0]] == 0)
                {
                        charaterListSize++;
                }
                tempFrequencyList[(int)byte[0]]++;
                originalFileSizeBytes++;
                sourceFile.read(byte,1);
        }
        resetSourceFile();
        charaterList = new char[charaterListSize];
        for(int t = 0; t < charaterListSize; t++)
        {
                charaterList[0] = 0;
        }
        unsigned long int tempNum = 0;
        for(int k = 0; k < charaterListSize; k++)
        {
                tempNum = 0;
                for(int j = 0;j<256;j++)
                {
                        if(tempFrequencyList[j] > tempNum)
                        {
                                tempNum = tempFrequencyList[j];
                                charaterList[k] = j;
                        }
                }
                tempFrequencyList[(int)charaterList[k]] = 0;
        }
        return charaterList;
}

void H3Compressor::resetSourceFile()
{
        sourceFile.close();
        if(mode)
        {//Open source in text mode if compressing
```

```
		sourceFile.open(sourceFileName,ios::in);
		if(!sourceFile)
		{
			cout<<"Failed to open sourceFile"<<endl;
			exit(0);
		}
	}
	else
	{//Open source in binary mode if decompressing
		sourceFile.open(sourceFileName,ios::in | ios::binary);
		if(!sourceFile)
		{
			cout<<"Failed to open sourceFile"<<endl;
			exit(0);
		}
	}
}

int H3Compressor::findLetter(char letter,char* charaterList)
{
	int count = 0;
	while(1)
	{
		if(charaterList[count] == letter)
		{
			return count;
		}
		count++;
	}
	return 0;
}

void H3Compressor::decompressFile()
{
	mode = false;
	prepareFiles(getNameFromUser());
	//get the number of letters different letters of the original
	char* fileGetter = new char[1];
	char* outToFile = new char[1];
	sourceFile.read(fileGetter,1);
	int letterListSize = fileGetter[0];
	//get the letter list
	char* letterList = new char[letterListSize];
	sourceFile.read(letterList,letterListSize);
	//getnerate the serch tree
	KeyList searchTree;
	searchTree.generateLetterSearchTree(letterListSize,letterList);
	//translate charaters
	sourceFile.read(fileGetter,1);
	unsigned long int key = 0;
	unsigned long int keyMarker = 1;
	while(sourceFile.gcount() != 0)
	{
		int byteMaker = 1;
		char letter;
		for(int shift = 0;shift<8;shift++)
		{
			byteMaker = 1;
			byteMaker = byteMaker << shift;
			if((fileGetter[0] & byteMaker) != 0)
			{
				key = key | keyMarker;
			}
```

```
				keyMarker = keyMarker << 1;
				if(searchTree.findLetterInTree(key,letter))
				{
					outToFile[0] = letter;
					outputFile.write(outToFile,1);
					keyMarker = 1;
					key = 0;
				}
			}
			sourceFile.read(fileGetter,1);
		}
}

void H3Compressor::displayStatistics(){
		cout<<"The original file size is: "<<endl;
		cout<<originalFileSizeBytes<<" bytes or"<<endl;
		cout<<originalFileSizeBytes * 8<<" bits"<<endl<<endl;
		unsigned long int compressedFileSizeBytes = 0;
		compressedFileSizeBytes = compressedFileSizeBits / 8;
		if((compressedFileSizeBits%8) != 0)
		{
			compressedFileSizeBytes++;
		}
		cout<<"The compressed file size is: "<<endl;
		cout<<compressedFileSizeBytes<<" bytes"<<endl;
		cout<<compressedFileSizeBits<<" bits   "<<compressedFileSizeBytes * 8
			<<" actual bits"<<endl<<endl;
		cout<<"compresstion ratio: "<<(unsigned long double)((double)originalFileSizeBytes/
			(double)compressedFileSizeBytes);
		cout<<" (bytes)"<<endl;
		cout<<"compresstion ratio: "<<(unsigned long double)((unsigned long double)
			(originalFileSizeBytes * 8)/compressedFileSizeBits);
		cout<<" (bits)"<<endl<<endl;
		unsigned long double percentOfOriginalBytes = 0;
		unsigned long double percentOfOriginalBits = 0;
		percentOfOriginalBytes = (double)((double)compressedFileSizeBytes/
			(double)originalFileSizeBytes) * 100;
		cout<<"percent of original: "<<percentOfOriginalBytes<<"% (bytes)"<<endl;
		percentOfOriginalBits = (double)((double)compressedFileSizeBits /
			(double)(originalFileSizeBytes * 8) * 100);
		cout<<"percent of original: "<<percentOfOriginalBits<<"% (bits)"<<endl<<endl;
		cout<<"percent savings: "<<(double)(100 - percentOfOriginalBytes)<<"% (bytes)"<<endl;
		cout<<"percent savings: "<<(double)(100 - percentOfOriginalBits)<<"% (bits)"<<endl;

}

#endif
```

```
                                    //KeyList.h
/*
Jeremy Brown
Thesis Poject
KeyList class
Last Updated: 2/15/04
*/
#ifndef keyList_h_
#define keyList_h_

#include <math.h>
#include "BitIndex.h"

/*
KeyList generate prefix codes that are comprsed of multiples of three backwards.
It will aslso genetate a decoding tree for decompression
*/

struct KeyEntry
{
public:
	char* key;
	BitIndex keySize;
};

struct LetterTreeNode
{
	LetterTreeNode* one;
	char letter;
	LetterTreeNode* zerro;
};

class KeyList
{
private:
	KeyEntry* keysList;//list of prefix codes
	int keysListSize;

	LetterTreeNode* searchTree;

	int getNumBits(unsigned long int number);//finds the numbers of bits in a number
	unsigned long int getSubBitStringFrom1To(int nthBit,unsigned long int number);
	bool isAaPostfixOfB(unsigned long int a,unsigned long int b);
	unsigned long int* generateTempList(int& tempListSize,int &numKeys);

public:

	KeyList();
	~KeyList();
	KeyEntry getKey(int index);
	void generateKeyList(int numKeys);
	void displayListContents();
	void generateLetterSearchTree(int numKeys,char* letterList);
	bool findLetterInTree(unsigned long int key,char& letter);
};

KeyList::KeyList()
{
	keysListSize = 0;
```

```
	keysList = 0;
	searchTree = new LetterTreeNode;
	(*searchTree).one=0;
	(*searchTree).zerro=0;
}

KeyList::~KeyList()
{
	for(int index = 0;index<keysListSize;index++)
	{
		//delete keysList[index].key;
	}
	if(keysList != 0)
	{
		delete keysList;
	}
}


unsigned long int* KeyList::genrateTempList(int& tempListSize,int &numKeys)
{
	unsigned long int key = 0;
	unsigned long int* tempKeyList = new unsigned long int [numKeys];
	tempListSize = 0;
	bool keyNotInList = false;

	while(tempListSize != numKeys)
	{
		key +=3;
		keyNotInList = true;
		//see if any of the keys in the list are a postfix of the new key
		if(tempListSize != 0)
		{
			for(int i = 0;i<tempListSize;i++)
			{
				if(isAaPostfixOfB(tempKeyList[i],key))
				{
					keyNotInList = false;
				}
			}
		}
		//add the key to the list if it does not have a postfix in the list
		if(keyNotInList)
		{
			tempKeyList[tempListSize]=key;
			tempListSize++;
		}
	}
	return tempKeyList;

}

void KeyList::generateKeyList(int numKeys)
{//generates a key list of multiples of three that are not postfixes of each other
	//or prefixes backwards.
	keysList = new KeyEntry[numKeys];
	keysListSize = numKeys;
	int tempListSize = 0;
	unsigned long int* tempKeyList = generateTempList(tempListSize,numKeys);
	for(int k = 0;k < tempListSize;k++)
	{
```

```
            KeyEntry entry;
            int numBits = getNumBits(tempKeyList[k]);
            int numBytes = (int)ceil((double)(numBits/8));
            entry.keySize+=numBits;
            entry.key = new char[numBytes];
            for(int p =0;p<=numBytes;p++)
            {
                entry.key[p] = 0;
            }
            unsigned int long intMarker = 1;
            for(int index = 0;index <= numBytes;index++)
            {
                int charMarker = 1;
                for(int i = 0;i<8;i++)
                {
                    if(intMarker & tempKeyList[k])
                    {
                        entry.key[index] = entry.key[index] | charMarker;
                    }
                    charMarker = charMarker << 1;
                    intMarker = intMarker << 1;
                }
            }
            keysList[k]=entry;
        }
        delete tempKeyList;
}


bool KeyList::isAaPostfixOfB(unsigned long int a,unsigned long int b)
{//return true if a is a postfix of be otherswist it is false
        if(getSubBitStringFrom1To(getNumBits(a),b) == a)
        {
            return true;
        }
        return false;
}


int KeyList::getNumBits(unsigned long int number)
{
        unsigned long int marker = 2147483648;//10000000000000000000000000000000 in binary
        int numbits = 32;
        while( (!(marker & number)) && (numbits>0))
        {
            numbits --;
            marker = marker >> 1;
        }
        return numbits;
}

unsigned long int KeyList::getSubBitStringFrom1To(int nthBit,unsigned long int number)
{//this funtion will return the number formed from the substirng of number from the
 //first bit to the nth bit

        int shift = 32 - nthBit;
        number = number << shift;
        number = number >> shift;
        return number;
```

```
}

void KeyList::displayListContents()
{
	for(int i = 0;i<keysListSize;i++)
	{
		cout<<"key size: ";keysList[i].keySize.displayPosition();cout<<" Key:";
		for(unsigned int j = 0;j<=keysList[i].keySize.getIndex();j++)
		{
			cout<<(int)keysList[i].key[j]<<" ";
		}
		cout<<endl;
	}
}

KeyEntry KeyList::getKey(int index)
{
	if(index < keysListSize)
	{
		return keysList[index];
	}
	else
	{
		cout<<"Error in KeyList.getKey: invalid index"<<endl;
		exit(0);
	}
}

void KeyList::generateLetterSearchTree(int numKeys,char* letterList)
{
	LetterTreeNode* treeNode = searchTree;
	int tempListSize = 0;
	unsigned long int* tempKeyList = generateTempList(tempListSize,numKeys);
	int keySize = 0;
	unsigned long int key = 0;
	//add each key to the list
	for(int index = 0;index<tempListSize;index++)
	{
		treeNode = searchTree;
		key = tempKeyList[index];
		keySize = getNumBits(key);
		unsigned long int marker = 1;
		//add a letter to the list
		for(int bitNum = 0;bitNum<keySize;bitNum++)
		{
			marker = 1;
			marker = marker << bitNum;
			if(marker & key)
			{//if the bit is one add a node to the one's side
				if((*treeNode).one == 0)
				{
					(*treeNode).one = new LetterTreeNode;
					//set blank node
					(*(*treeNode).one).one = 0;
					(*(*treeNode).one).zerro = 0;
				}
				treeNode = (*treeNode).one;
			}
			else
			{//if the bit is zerro add a node to the zerro's side
				if((*treeNode).zerro == 0)
				{
```

```
                                    (*treeNode).zerro = new LetterTreeNode;
                                    //set blank node
                                    (*(*treeNode).zerro).one = 0;
                                    (*(*treeNode).zerro).zerro = 0;
                              }
                              treeNode = (*treeNode).zerro;
                        }
                  }
                  (*treeNode).letter = letterList[index];
            }
}

bool KeyList::findLetterInTree(unsigned long int key,char& letter)
{
      LetterTreeNode* treeNode = searchTree;
      int keySize = 0;
      keySize = getNumBits(key);
      if(keySize != 0)
      {
            unsigned long int marker = 1;
            //find key in list
            for(int bitNum = 0;bitNum<keySize;bitNum++)
            {
                  marker = 1;
                  marker = marker << bitNum;
                  if(marker & key)
                  {//if the bit is one go to the one node
                        if((*treeNode).one == 0)
                        {
                              return false;
                        }
                        treeNode = (*treeNode).one;
                  }
                  else
                  {//if the bit is zerro go to the zerro node
                        if((*treeNode).zerro == 0)
                        {
                              return false;
                        }
                        treeNode = (*treeNode).zerro;
                  }
            }
            if(((*treeNode).one == 0) && ((*treeNode).zerro == 0))
            {
                  letter = (*treeNode).letter;
                  return true;
            }
            else
            {
                  return false;
            }
      }
      else
      {
            return false;
      }
}

#endif
```

```
                                    //BitIndex.h
/*
Jeremy Brown
Thesis Poject
BitIndex class
Last Updated: 2/15/04
*/
#ifndef bitindex_h_
#define bitindex_h_

#include <stdlib.h>
/*
This class stores a spot of a particular bit in an array
*/

class BitIndex
{
private:
	unsigned long int index;//index in the array of bytes
	unsigned int offset;//offset (index of the bit in the byte)

public:
	BitIndex();
	BitIndex(unsigned long int newIndex,int newOffset);
	void setIndexAndOffset(unsigned long int newIndex,int newOffset);
	unsigned long int getIndex();
	void setIndex(unsigned long int newIndex);
	unsigned int getOffset();
	void setOffset(unsigned long int newOffset);
	void displayPosition();

	void operator=(unsigned long int numBits);
	void operator+=(unsigned long int numBits);
	BitIndex operator+(unsigned long int numBits);
	BitIndex operator-(unsigned long int num);
	bool operator<(BitIndex compareIndex);
	bool operator==(BitIndex compareIndex);
	bool operator<=(BitIndex compareIndex);
	unsigned int operator-(BitIndex compareIndex);
};

BitIndex::BitIndex()
{
	index = 0;
	offset = 0;
}


BitIndex::BitIndex(unsigned long int newIndex,int newOffset)
{
	index = newIndex;
	offset = newOffset;
}

void BitIndex::setIndexAndOffset(unsigned long int newIndex,int newOffset)
{
	index = newIndex;
	offset = newOffset;
}

unsigned long int BitIndex::getIndex()
{
```

```
	return index;
}

void BitIndex::setIndex(unsigned long int newIndex)
{
	index = newIndex;
}

unsigned int BitIndex::getOffset()
{
	return offset;
}

void BitIndex::setOffset(unsigned long int newOffset)
{
	offset = newOffset;
}

void BitIndex::displayPosition()
{
	cout<<"BitIndex:  Index: "<<index<<" Offset Index: "<<offset<<endl;

}

void BitIndex::operator=(unsigned long int numBits)
{
	offset = numBits%8;
	index = (numBits - offset)/8;
}

void BitIndex::operator+=(unsigned long int numBits)
{
	int leftover = numBits%8;
	numBits = numBits - leftover;
	if((leftover + offset) >= 8)
	{
		index++;
		offset = leftover + offset - 8;
	}
	else
	{
		offset = leftover + offset;
	}
	index = index + (numBits/8);
}

BitIndex BitIndex::operator+(unsigned long int numBits)
{
	int leftover = numBits%8;
	numBits = numBits - leftover;
	unsigned long int tempIndex = index;
	unsigned int tempOffset = offset;
	if((leftover + tempOffset) >= 8)
	{
		tempIndex++;
		tempOffset = leftover + tempOffset - 8;
	}
	else
	{
		tempOffset = leftover + tempOffset;
	}
	tempIndex = tempIndex + (numBits/8);
	BitIndex newIndex(tempIndex,tempOffset);
```

```
	return newIndex;
}

BitIndex BitIndex::operator-(unsigned long int num)
{
	unsigned long int tempIndex = index;
	unsigned int tempOffset = offset;

	if((tempIndex != 0) || (tempOffset != 0))
	{
		if(tempOffset != 0)
		{
			tempOffset--;
		}
		else
		{
			tempIndex--;
			tempOffset = 7;
		}

	}
	else
	{
		cout<<"Error in BitIndex.operator-: cannot have negative index"<<endl;
		exit(0);
	}
	BitIndex newIndex(tempIndex,tempOffset);
	return newIndex;
}



bool BitIndex::operator<(BitIndex compareIndex)
{
	if(index < compareIndex.getIndex())
	{
		return true;
	}
	if((index == compareIndex.getIndex()) && (offset < compareIndex.getOffset()))
	{
		return true;
	}
	return false;
}

bool BitIndex::operator==(BitIndex compareIndex)
{
	if((index == compareIndex.getIndex()) && (offset == compareIndex.getOffset()))
	{
		return true;
	}
	return false;
}

bool BitIndex::operator<=(BitIndex compareIndex)
{
	return ((*this) < compareIndex) || ((*this) == compareIndex);
}

unsigned int BitIndex::operator-(BitIndex compareIndex)
```

```
{
	unsigned int tempIndex = index;
	unsigned int tempOffset = offset;
	if(compareIndex<(*this))
	{

		tempIndex = tempIndex - compareIndex.getIndex();
		if(tempOffset>=compareIndex.getOffset())
		{
			tempOffset = tempOffset - compareIndex.getOffset();
		}
		else
		{
			tempIndex--;
			tempOffset = (tempOffset - compareIndex.getOffset())%8;
		}
	}
	else
	{
		cout<<"Error in BitIndex: cannot have a negative index"<<endl;
		exit(0);
	}
	return 8*tempIndex+tempOffset;
}

#endif
```

Appendix D

Bibliography

Works Cited

Goebel, Greg. "Introduction / Lossless Data Compression." 1 May 2003. 9 Nov. 2004 http://www.vectorsite.net/ttdcmp1.html .

Hirshberg, Daniel S., and Debra A. Lelewer. "Data Compression." 8 Nov. 2004 http//www.ics.uci.edu/~dan/pubs/DataCompression.html .

LewLand, Robert Edward. Cryptological Mathematics. The Mathematical Association of America, 2000.

Sacco, William, et al. Information Theory: Saving Bits. Providence: Janson Pub. Inc., 1988.

Sayhood, Khalid. Introduction to Data Compression. San Francisco: Morgan Kaufmann Publishers, Inc., 1996.

Sayood, Khalid, ed. Lossless Compression Handbook. San Francisco: Academic Press, 2003.

Works Consulted

Schildt, Herbert. C/C++ Programmer's Reference. 3rd ed. New York: McGraw – Hill/Osborne, 2003.