MOI UNIVERSITY

DATA COMPRESSION

Project documentation

Ouma molly Adhiambo and Ouma Sylvester Ouma
7-20-2022

Table of Contents

1.DATA COMPRESSION

Data compression, source coding, or bit-rate reduction is the process of encodingg information using fewer bits than the original representation thereby reducing the size of the computer file. Compression is done by programs that use functions or algorithms to effectively discover how to reduce size of the data. There are two types of compression: lossy and lossless. Lossy compression reduces bits by removing unnecessary or less important information. This format is suitable for file types in which lost details are hardly perceptible.

Types of data compression

a. Row compression

Row level compression will provide savings by not storing blank characters within fixed character strings like char data type. NULL and 0 values are not stored and so do not incur additional storage overhead.

b. Page compression

Page compression uses a complex algorithm to minimize the storage space of the data. This is also known as dictionary compression. SQL Server 2008 looks at all the data stored in a page and builds a dictionary based upon that data which can be referenced for repeated values. Only the dictionary ID and the changes of the dictionary value are stored. This provides great savings for similar patterned data. Page compression includes row compression as well, so we get the benefit of both the compressions.

Importance of data compression

Data compression has various applications in todays connected world. The following are some of its importance;

i. Save on storage space

   Applications continue to generate massive amounts of data. All these take huge amount of disk space. Compressing documents when not needed for use and decompressing them later when needed saves on data storage cost

ii. Fast data transmission over a network

   The main network, internet now involves exchange of files, documents, papers and metadata. When this data is compressed at the sending terminal and sent over a network, a huge amount of bandwidth is saved hence transfer is faster. This saves on the data transfer cost.

iii. Network efficiency

Passing compressed data over a network improves network performance by reducing the network bandwidth requirement. Improved network performance is a great achievement on reducing latency of IOT devices and user experience.

iv. Optimizes backup storage performance

Relevance of Data Compression

- Genetics

Genetics compression algorithms are the latest generation of lossless algorithms that compress data using both conventional compression algorithms and genetic algorithms adapted to the specific datatype.

- World wide web

Today compression is used the world wide web which is increasingly becoming more complex there is an evident need to have compression algorithms that are fast, lossless and efficient. Compression plays a key role when it comes to bandwidth saving and speeding up the sites.

gzip is a compression method that is used to make files smaller for storage and faster transmission over the network. gzip is one of the most popular, powerful, and effective ways of compressing data and it can reduce the file size by up to 70%.gzip is based on the deflate algorithm, which in turn is a combination of lz77 and Huffman coding. understanding how lz77 works is essential to understand how compression methods like deflate and gzip work.

Brotli compression a new compression method called Brotli was introduced not too long ago. the Brotli compression algorithm is optimized for the web and specifically for small text documents. we will discuss more this compression method and what it has to offer to the world wide web community in the second part of the article.

Big Data

Big data refers to the large, diverse sets of information that grow at ever-increasing rates. It encompasses the volume of information, the velocity or speed at which it is created and collected, and the variety or scope of the data points being covered (known as the "three v's" of big data). Big data often comes from data mining and arrives in multiple formats. In data communication, scientific research and medicine, an increasing number of today's applications require the capture and analysis of massive amounts of data in real time.

But "big data," as it's known, can present big problems, particularly in specialized fields in which the events being studied occur at rates that are too fast to be sampled and converted into digital data in real time. For example, in order to detect rare cancer cells in blood, researchers must screen millions of cells in a high-speed flow stream. The growth of big data has created a demand for ever-increasing processing power and efficient storage. Digital Globe's databases, for example, expand by roughly 100TBs a day and cost an estimated $500K a month to store. Compressing big data can help address these demands by reducing the amount of storage and bandwidth required for data sets. Compression can also remove irrelevant or redundant data, making analysis and processing easier and faster. This makes data compression relevant today since data grows faster than the available memory and therefore the speed of transmission and space occupied by big data on the devices and transparency is a concern.

2.LOSSLESS COMPRESSION

Lossless compression is a class of data compression that allows the original data to be perfectly reconstructed from the compressed data with no loss of information. It is possible because most real-world data exhibit statistical redundancy.

Lossless compression is used in cases where it is important that the original and the decompressed data be identical, or where deviations from the original data would be unfavourable. Typical examples are executable programs, text documents, and source code. Some image file formats, like PNG or GIF, use only lossless compression, while others like TIFF and MNG may use either lossless or lossy methods. Lossless audio formats are most often used for archiving or production purposes, while smaller lossy audio files are typically used on portable players and in other cases where storage space is limited or exact replication of the audio is unnecessary.

Lossless data compression is used in many applications. For example, it is used in the ZIP file format and in the GNU tool gzip. It is also often used as a component within lossy data compression technologies.

Why lossless compression is better than lossy compression

With lossless compression the quality of the text or file compressed is not degraded. Quality is maintained and reduced file size is also obtained.

Lossless compression is used in compressing many different types of files like text files, program codes and other critical data while lossy is only used in video, image and audio compression.

Lossless can reduce file size 50% without losing its quality while lossy can remove some data permanently.

Lossless compression algorithm guarantees security of the compressed information which lossy cannot offer or rather guarantee.

3.HUFFMAN CODING ALGORITHM AND DATA COMPRESSION

In the early 1980s, personal computers had hard disks that were no larger than 10MB; today, the puniest of disks are still measured in hundreds of gigabytes. Even though hard drives are getting bigger, the files we want to store (funny pictures of cats, videos, music and so on) seem to keep pace with that growth which makes even today's gargantuan disk seem too small to hold everything. One technique to use our storage more optimally is to compress the files. By taking advantage of redundancy or patterns, we may be able to "abbreviate" the contents in such a way to take up less space yet maintain the ability to reconstruct a full version of the original when needed. Such compression could be useful when trying to cram more things on a disk or to shorten the time needed to copy/send a file over a network.
There are compression algorithms that you may already have heard of. Some compression formats, such as JPEG, MPEG, or MP3, are specifically designed to handle a particular type of data file. They tend to take advantage of known features of that type of data (such as the propensity for pixels in an image to be same or similar colors to their neighbors) to compress it. Other tools such as compress, zip, or pack and programs like StuffIt or ZipIt can be used to compress any sort of file. These algorithms have no a priori expectations and usually rely on studying the particular data file contents to find redundancy and patterns that allow for compression.
Some of the compression algorithms (e.g. JPEG, MPEG) are lossy. That is decompressing the compressed result doesn't recreate a perfect copy of the original. Such an algorithm compresses by "summarizing" the data.
The summary retains the general structure while discarding the more minute details. For sound, video, and images, this imprecision may be acceptable because the bulk of the data is maintained and a few missed pixels or milliseconds of video delay is no big deal. For text data, though, a lossy algorithm usually isn't ap-
propriate. An example of a lossy algorithm for compressing text would be to remove all the vowels. Compressing the previous sentence by this scheme results in: n xmpl f lssy lgrthm fr cmprssng txt wld b t rmv ll th vwls.
This shrinks the original 87 characters down to just 61 and requires only 70% of the original space. To decompress, we could try matching the consonant patterns to English words with vowels inserted, but we cannot reliably reconstruct the original in this manner. Is the compressed word "fr" an abbreviation for the word "four" or the word "fir" or "far"? An intelligent reader can usually figure it out by context, but, alas, a brainless computer can't be sure and would not be able to faithfully reproduce the original. For files

containing text, we usually want a lossless scheme so that there is no ambiguity when re-creating the original meaning and intent.

An Overview

The standard ASCII character encoding uses the same amount of space (one byte or eight bits, where each bit is either a 0 or a 1) to store each character. Common characters don’t get any special treatment; they require the same 8 bits that are used for much rarer characters such as 'ü' or '¥'. A file of 1000 characters encoded using the ASCII scheme will take 1000 bytes (8000 bits); no more, no less, whether it be a file of 1000 spaces to a file containing 4 instances each of 250 different characters. A fixed-length encoding like ASCII is convenient because the boundaries between characters are easily determined and the pattern used
for each character is completely fixed (i.e. 'a' is always exactly 97).
In practice, it is not the case that all 256 characters in the ASCII set occur with equal frequency. In an English text document, it might be the case that only 90 or so distinct characters are used at all (meaning166 characters in the ASCII never even appear) and within those 90 there are likely to be significant differences in the character counts. The Huffman encoding scheme takes advantage of the disparity between frequencies and uses less storage for the frequently occurring characters at the expense of having to use more storage for each of the rarer characters. Huffman is an example of a variable-length encoding—some characters may only require 2 or 3 bits and other characters may require 7, 10, or 12 bits. The savings from not having to use a full 8 bits for the most common characters makes up for having to use more than 8 bits for the rare characters and the overall effect is that the file almost always requires less space.

ASCII Encoding

The example we’re going to use throughout this handout is encoding the particular string "happy hip hop". Using the standard ASCII encoding, this 13-character string requires 13*8 = 104 bits total. The table below shows the relevant subset of the standard ASCII table.

char	ASCII	bit pattern (binary)
h	104	01101000
a	97	01100001
p	112	01110000
y	121	01111001

i	105	01101001
o	111	01101111
space	32	00100000

The string "happy hip hop" would be encoded in ASCII as 104 97 112 112 121 32 104 105 112 32 104 111 112. Although not easily readable by humans, it would be written as the following
stream of bits (each byte is spaced to show the boundaries):
01101000 01100001 01110000 01110000 01111001 00100000 01101000 01101001 01110000 00100000 01101000 01101111 01110000
To decode such a string (i.e. translate the binary encoding back to the original characters), we merely need to break the encoded stream of bits up into 8-bit bytes, and then convert each byte using the fixed ASCII encoding. The first 8 bits are 01101000, which is the pattern for number 104, and position 104 in the
ASCII set is assigned to lowercase 'h'. A file encoded in ASCII does not require any additional information to be decoded since the mapping from binary to characters is the same for all files and computers.

A More Compact Encoding

The first thing you might notice about ASCII encoding is that using 8 bits per character can be excessively generous. Although it allows for the possibility of representing 256 different characters, we only have seven distinct characters in the phrase we’re trying to encode, and thus could distinguish among these patterns
with fewer bits. We could set up a special coding table just for this phrase using 3 bits for each character.
Creating such an encoding is trivial: we create a list of the unique characters, and then go through and assign each a distinct encoded number from 0 to N-1. For example, here is one possible 3-bit encoding (of the 7! possible permutations):

Char	number	bit pattern
h	0	000
a	1	001
p	2	010
y	3	011
i	4	100
o	5	101
space	6	110

Using this table, "happy hip hop" is encoded as 0 1 2 2 3 6 0 4 2 6 0 5 2, or in binary:
000 001 010 010 011 110 000 100 010 110 000 101 010
Using three bits per character, the encoded string requires 39 bits instead of the original 104 bits, compressing to 38% of its original size.
However, to decode this binary representation, one would need to know the special mapping used, since using 000 for 'h' is not standard practice and in fact, in this scheme, each compressed string uses its own special-purpose mapping that is not necessarily like any other. Some sort of header or auxiliary file would have to be attached or included with the encoded representation that provides the mapping information. That header would take up some additional space that would cut into our compression savings. For a large enough file, though, the savings from trimming down the per-character cost would likely outweigh the expense of the additional table storage.

A Variable-Length Encoding

What if we drop the requirement that all characters take up the same number of bits? By using fewer bits to encode characters like 'p', 'h', and space that occur frequently and more to encode characters like'y' and 'o' that occur less frequently, we may be able to compress even further. We'll later show how we generated the table below, but for now just take our word for it that is represents an optimal Huffman encoding for the string "happy hip hop":

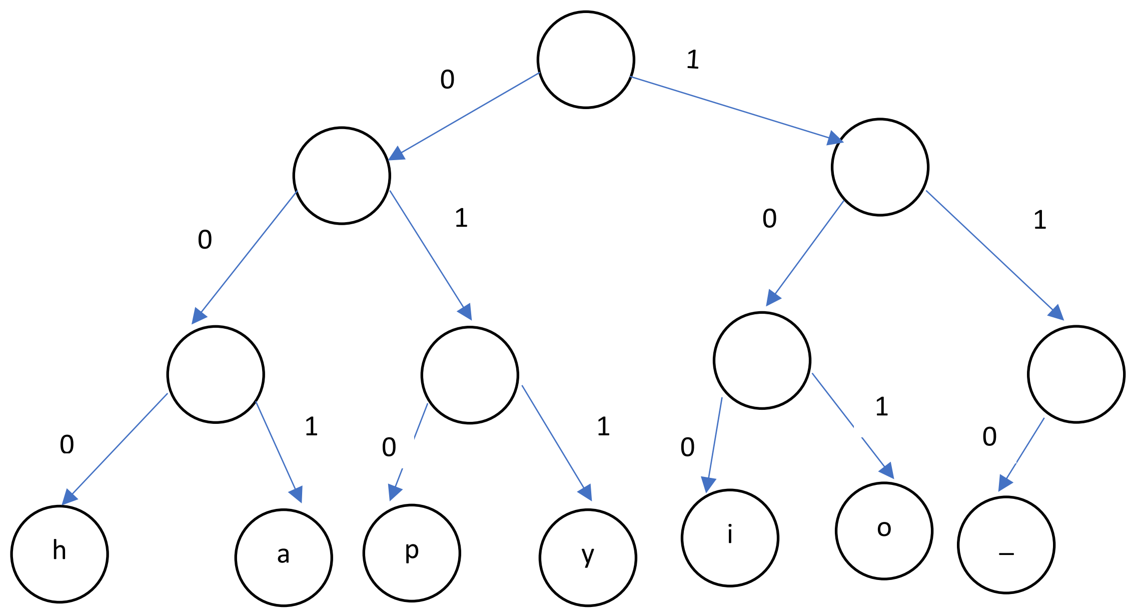
char	bit pattern
h	01
a	000
p	10
y	1111
i	001
o	1110
space	110

Each character has a unique bit pattern encoding, but not all characters use the same number of bits. The
string "happy hip hop" encoded using the above variable-length code table is:
01 000 10 10 1111 110 01 001 10 110 01 1110 10
The encoded phrase requires a total of 34 bits, saving a few more bits from the fixed-length version.
What is tricky about a variable-length code is that we no longer can easily determine the boundaries between characters in the encoded stream of bits when decoding. I boxed every other character's bit pattern above to help you

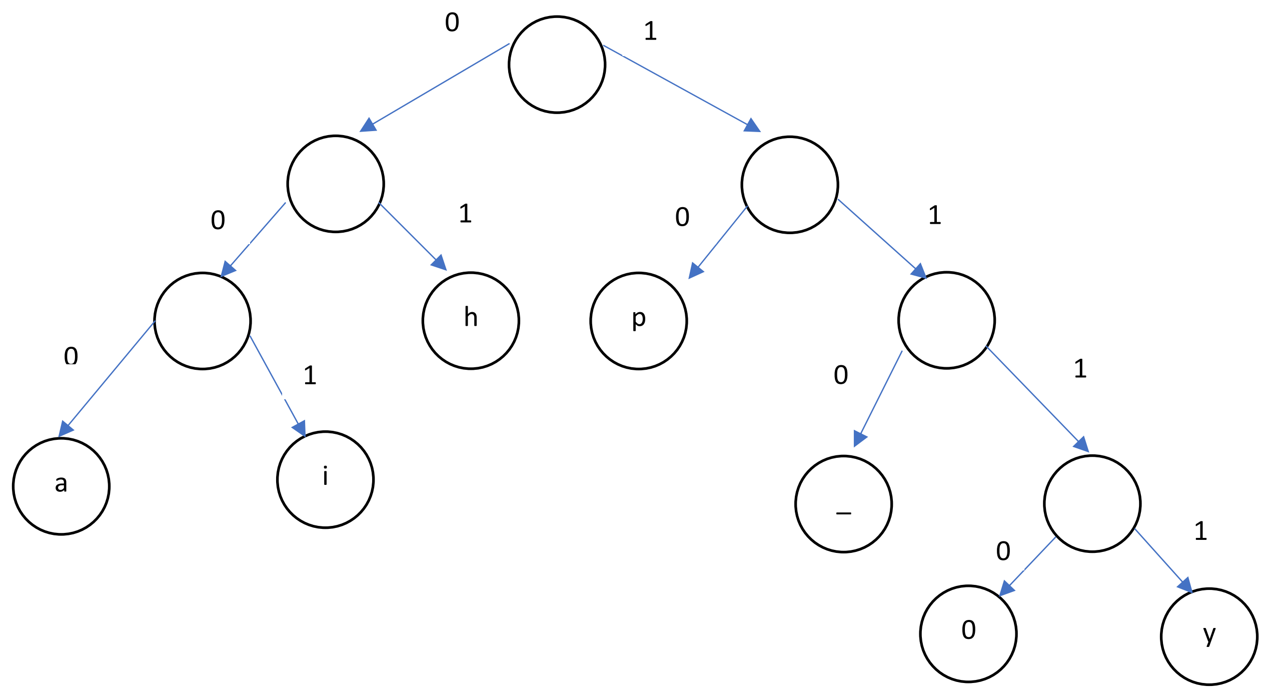
visualize the encoding, but without this aid, you might wonder how you will know
whether the first character is encoded with the two bits 01 or the three bits 010 or perhaps just the first bit 0? If you look at the encoding in the table above, you will see that only one of these options is possible. There is no character that encodes to the single bit 0 and no character that encodes to the sequence
010 or 0100 or 01000 for that matter. There is, however, a character that encodes to 01 and that is 'h'. One of the important features of the table produced by Huffman coding is the prefix property: no character's encoding is a prefix of any other (i.e., if 'h' is encoded with 01 then no other character's encoding will start with 01 and no character is encoded to just 0). With this guarantee, there is no ambiguity in determining where the character boundaries are. We start reading from the beginning, gathering bits in a sequence until we find a match. That indicates the end of a character and we move on to decoding the
next character.
Like the special-purpose fixed-length encoding, a Huffman encoded file will need to provide a header with the information about the table used so we will be able to decode the file. Each file's table will be unique since it is explicitly constructed to be optimal for that file's contents.

Encoding Seen as a Tree

One way to visualize any particular encoding is to diagram it as a binary tree. Each character is stored at a leaf node. Any particular character encoding is obtained by tracing the path from the root to its node.
Each left-going edge represents a 0, each right-going edge a 1. For example, this tree diagrams the compact fixed-length encoding we developed previously:



In the above tree, the encoding for 'y' can be determined by tracing the path from the root to the 'y' node. Going left then right then right again represents a 011 encoding. A similar, much larger tree could be constructed for the entire ASCII set, it would be 8 levels deep and at the bottom would be 256 leaf nodes, one for each character. The node for the character 'a' (97 or 01100001 in binary) would be at the end of the left-right-right-left-left-left-left-right path from the root. We're starting to see why they're called binary trees!
Now, let’s diagram such a tree for the variable-length Huffman encoding we were using:



The path to 'h' is just left right or 01, the path to 'y' is right-right-right-right or 1111. Notice that the prefix property of the Huffman encoding is visually represented by the fact that characters only occupy leaf nodes (i.e., those nodes which are not a prefix of any further nodes).
The tree shown above for "happy hip hop" is, in fact, an optimal tree—there are no other tree encodings by character that use fewer than 34 bits to encode this string. There are other trees that use exactly 34bits; for example, you can simply swap any sibling nodes in the above tree and get a different but equally optimal encoding.
The Huffman tree doesn't appear as balanced as the fixed-length encoding tree. You've heard in our discussion on binary search trees that an unbalanced tree is bad thing. However, when a tree represents a character encoding, that lopsidedness is actually a good thing. The shorter paths represent those frequently occurring characters that are being encoded with fewer bits and the longer paths are used for more rare characters. Our plan is to shrink the total number of bits required by shortening the encoding for some characters at the expense of lengthening others. If all characters occurred with equal frequency, we would have a balanced tree where all paths were roughly equal. In such a situation we can't achieve much
compression since there are no real repetitions or patterns to be exploited.

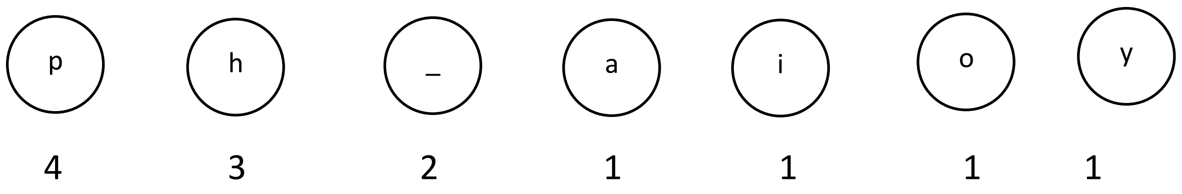
Decoding Using the Tree

A particularly compelling reason to diagram an encoding as a tree is the ease with which it supports decoding. Let's use the fixed-length tree to decode the stream of bits 011101010010011. Start at the beginning of the bits and at the root of the tree. The first bit is 0, so trace one step to the left, the next bit is 1, so follow right from there, the following bit is 1, so take another right. We have now landed at a leaf, which indicates that we have just completed reading the bit pattern for a single character. Looking at the leaf's label, we learn we just read a 'y'. Now we pick up where we left off in the bits and start tracing again from the root. Tracing 101 leads us to 'i'. Continuing through the remaining bits and we decode the string "yippy".
The same path-following strategy works equally well on the Huffman tree. Decoding the stream of bits 111100110101111will first trace four steps down to the left to hit the 'y' leaf, then a left-left-right path to the 'i' leaf and so on. Again, the decoded string is "yippy". Even though the encoded characters don't start and end at evenly spaced boundaries in the Huffman-encoded bits, we have no trouble determining where each character ends because we can easily detect when the path hits a leaf node in the encoding tree.

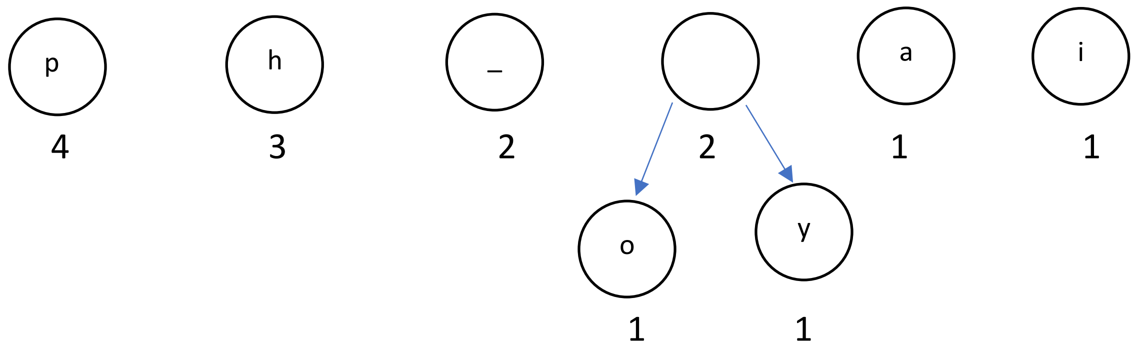
Generating an Optimal Tree

The pertinent question now: how is that special tree constructed? We need an algorithm for constructing the optimal tree giving a minimal per-character encoding for a particular file. The algorithm we will use here was invented by David Huffman in 1952. To begin generating the Huffman tree, each character gets a weight equal to the number of times it occurs in the file. For example, in the "happy hip hop" example, the character 'p' has weight 4, 'h' has weight 3, the space has weight 2, and the other characters have weight 1. Our first task is to calculate these weights, which we can do with a simple pass through the file to get the frequency counts. For each character, we create an unattached tree node containing the character value and its corresponding weight. You can think of each node as a tree with just one entry. The idea is to combine all these separate trees into an optimal tree by wiring them together from the bottom upwards. The general approach is as follows:

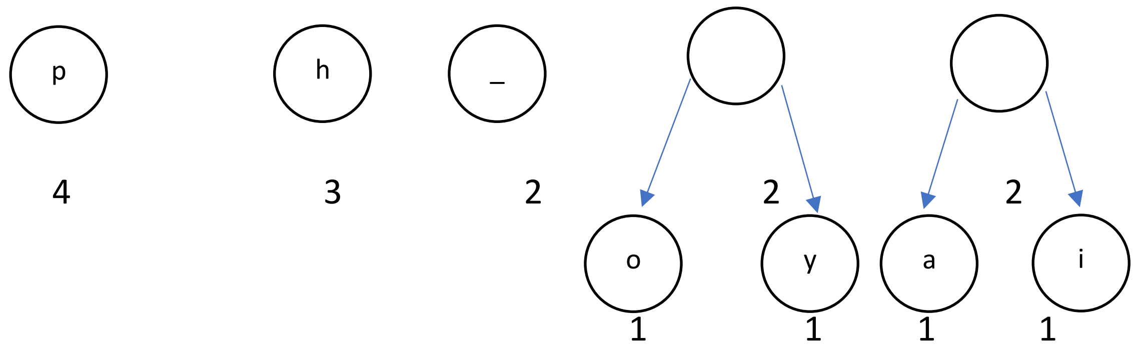
1. Create a collection of singleton trees, one for each character, with weight equal to the character frequency.
   2. From the collection, pick out the two trees with the smallest weights and remove them. Combine them into a new tree whose root has a weight equal to the sum of the weights of the two trees and with the two trees as its left and right subtrees.
   3. Add the new combined tree back into the collection.
   4. Repeat steps 2 and 3 until there is only one tree left.
   5. The remaining node is the root of the optimal encoding tree.
   Sounds simple, doesn't it? Let's walk through building the optimal tree for our example string "happy hip hop". We start with this collection of singletons; the weight of each node is labelled underneath:



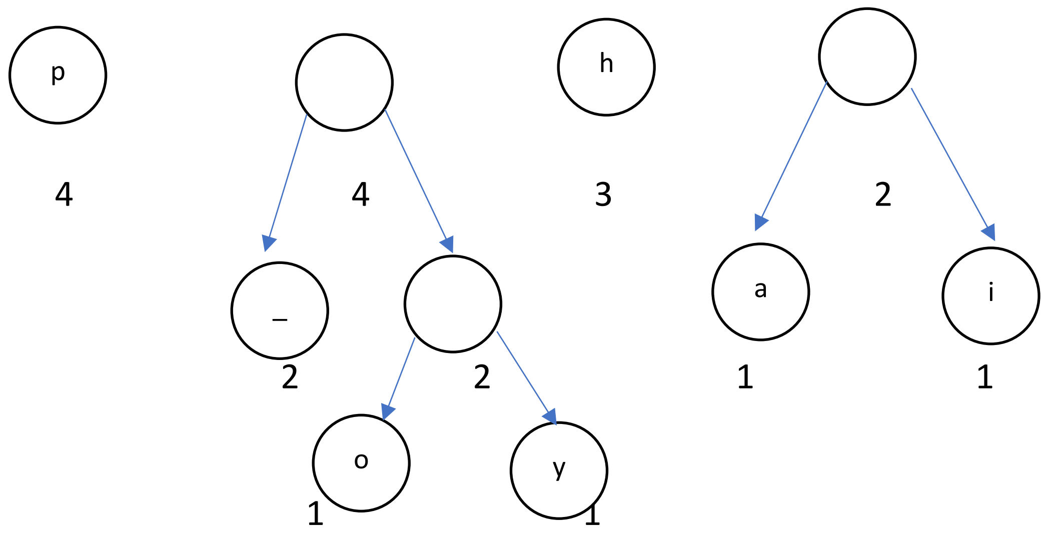
We start by choosing the two smallest nodes. There are four nodes with the minimal weight of one, it doesn't matter which two we pick. We choose 'o' and 'y' and combine them into a new tree whose root is the sum of the weights chosen. We replace those two nodes with the combined tree. The nodes remaining in the collection are shown in the light gray box at each stage.



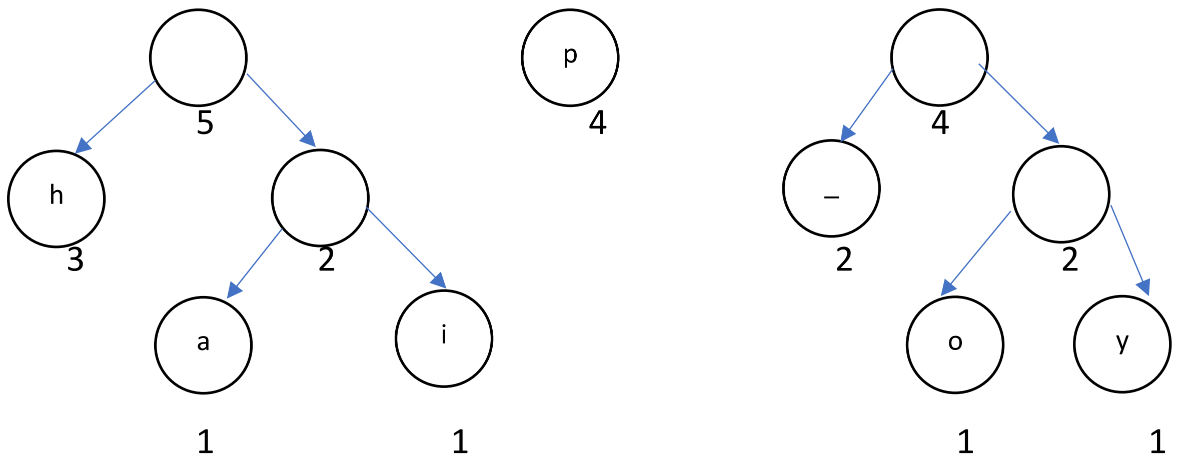
Now we repeat that step, this time there is no choice for the minimal nodes, it must be 'a' and 'i'. We take those out and combine them into a weight 2 tree. Note how the collection of nodes shrinks by one each iteration (we remove two nodes and add a new one back in).



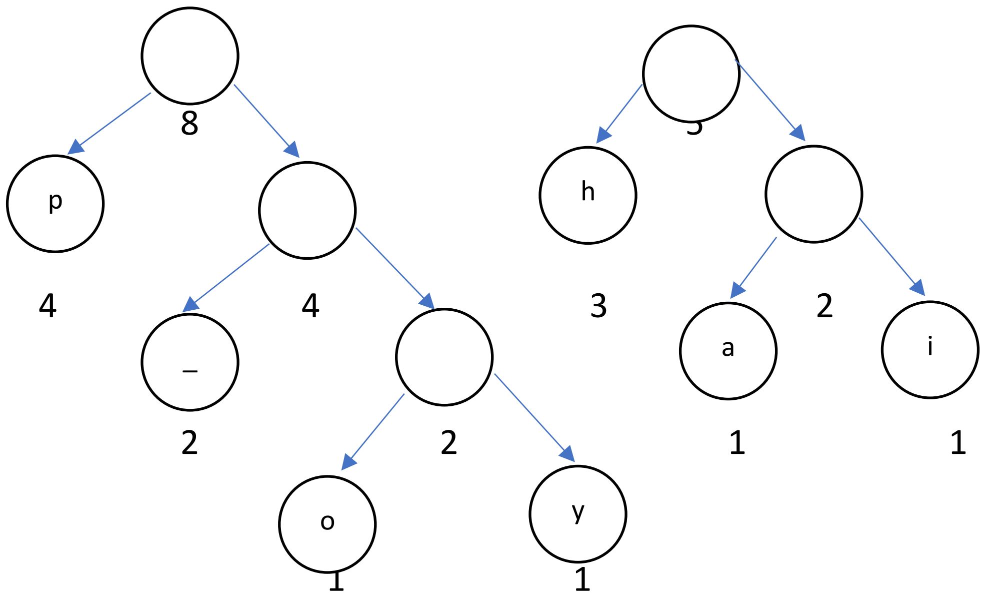
Again, we pull out the two smallest nodes and build a tree of weight 4:



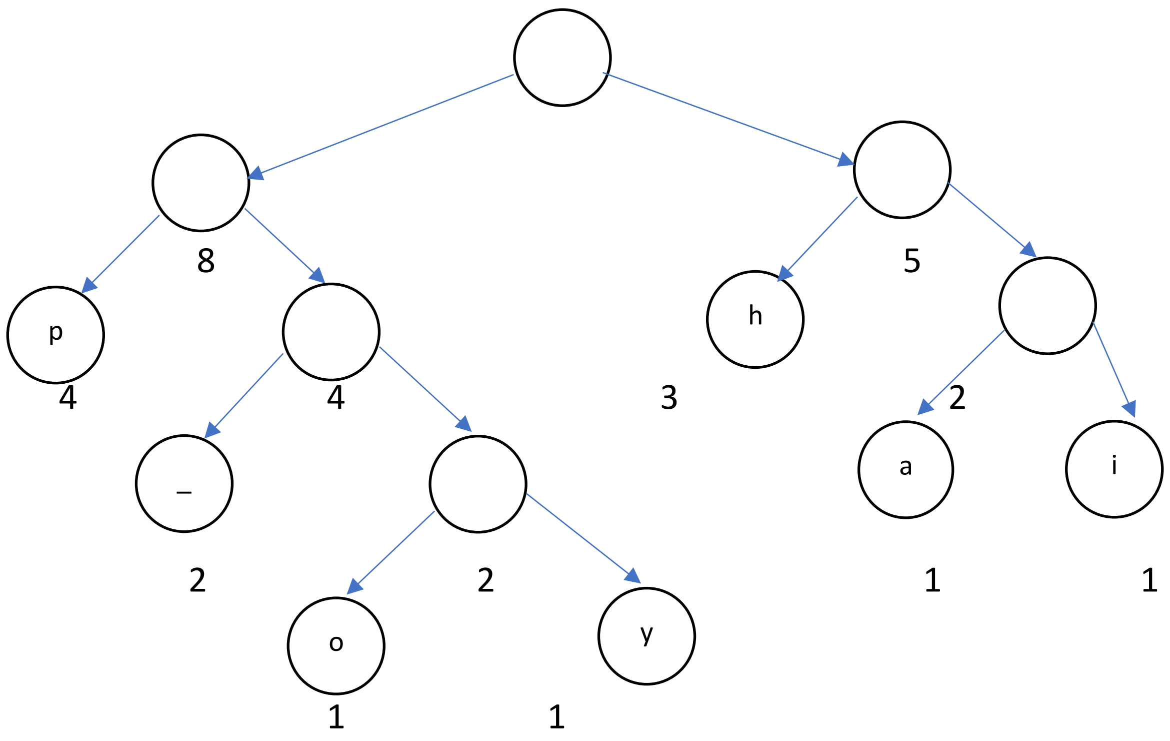
Note when we build a combined node, it doesn't represent a character like the leaf nodes do. These interior nodes are used along the paths that eventually lead to valid encodings, but the prefix itself does not encode a character. One more iteration combines the weight 3 and 2 trees into a combined tree of weight 5:



Combining the two 4s gets a tree of weight 8:



And finally, we combine the last two to get our final tree. The root node of the final tree will always have a weight equal to the number of characters in the input file.

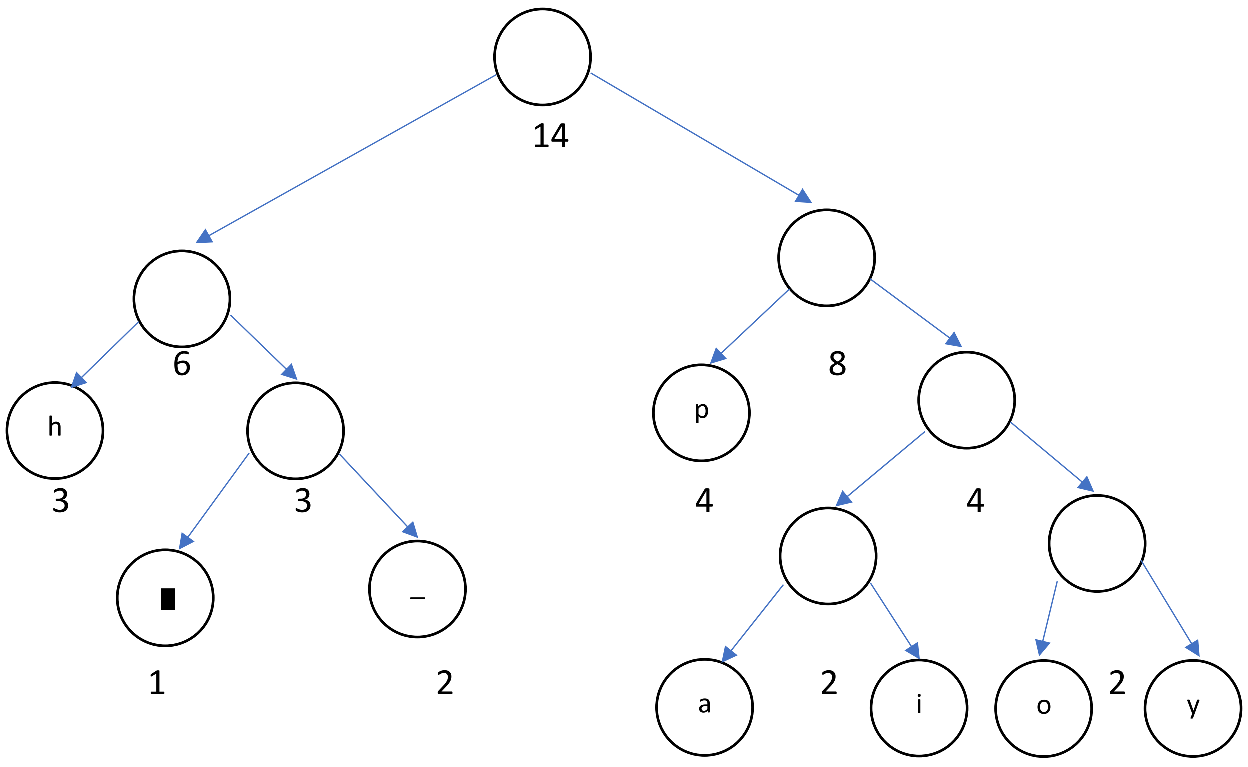


Note that this tree is different from the tree on page 4, and has slightly different bit patterns, but both trees are optimal and the total number of bits required to encode "happy hip hop" is the same for either tree. When we have choices among equally weighted nodes (such as in the first step choosing among the four characters with weight 1) picking a different two will result in a different, but still optimal, encoding. Similarly, when combining two subtrees, it is as equally valid to put one of the trees on the left and the other on the right as it is to reverse them.
Remember that it is essential that you use the same tree to do both encoding and decoding of your files. Since each Huffman tree creates a unique encoding of a particular file, you need to ensure that your decoding algorithm generates the exact same tree, so that you can get back the file.

Practical Considerations: The Pseudo-EOF

The preceding discussion of Huffman coding is correct from a theoretical perspective, but there are a few real-world details we need to address before moving on. One important concern is what happens when we try to store a Huffman-encoded sequence on-disk in a file. Each file on your computer is typically stored as a number of bytes (groups of eight bits); files are usually measured in “megabytes” and “gigabytes” rather than “megabits” or “gigabits.” As a result, if you try to write a Huffman-encoded string of bits into a file, if you don't have exactly a multiple of eight bits in your encoding, the operating system will typically pad the rest of the bits with random bits. For

example, suppose that we want to encode the string “ahoy” using the above Huffman tree. This results in the following sequence of bits:
1101001100111
This is exactly thirteen bits, which means that, when stored on-disk, the sequence would be padded with three extra random bits. Suppose that those bits are 111. In that case, the bit sequence would be written to disk as 1101001100111111.
If we were to then load this back from disk and decode it into a sequence of characters, we would get the string “ahoyi,” which is not the same string that we started with! Even worse, if those random bits end up being 000, then the stored bit sequence would be 11101001100111000.The problem is that as we decode this, we read the first thirteen bits back as “ahoy,” but encounter an error when reading the last three bits because 000 is not a character in our encoding scheme.
To fix this problem, we have to have some way of knowing when we've finished reading back all of the bits that encode our sequence. One way of doing this is to transform our original input string by putting some special marker at the end. This marker won't appear anywhere else in the string and serves purely as an indicator that there is nothing left to read. For example, we might actually represent the string “happy hip hop” as “happy hip hop■”, where ■ marks the end of the input.
When we build up our Huffman encoding tree for this string, we will proceed exactly as before, but would add in an extra node for the ■marker. Here is one possible encoding tree for the characters in this new string:



Now, if we want to encode “happy hip hop■”, we get the following bitstring:
001100101011110110011011001100111010010
This does not come out to a multiple of eight bits (specifically, it's 39 bits long), which means that it will be padded with extra bits when stored on-disk. However, this is of no concern to us – because we have written the ■ marker to the end of the string, as we're decoding, we can tell when to stop reading bits. For example, here is how we might decode the above string:

00	H
1100	A
10	P
10	P
1111	Y
011	
00	H
1101	I
10	P
011	
00	H
1110	O
10	P
010	■
0	Extra bits ignored; we knew to stop when seeing ■.

This ■ character is called a pseudo-end-of-file character or pseudo-EOF character, since it marks where the logical end of the bit stream is, even if the file containing that bit stream contains some extra garbage bits at the end. When you actually implement Huffman encoding in the assignment, you will have to make sure to insert a pseudo-EOF character into your encoding tree and will have to take appropriate steps to ensure that you stop decoding bits when you reach it.

Practical Considerations: The Encoding Table

There is one last issue we have not discussed yet. Suppose that I want to compress a message and send it to you. Using Huffman coding, I can convert the message (plus the pseudo-EOF) into a string of bits and send it to you. However, you cannot decompress the message, because you don't have the encoding tree that I used to send the message. There are many ways to resolve this. We could agree on an encoding tree in advance, but this only works if we

already know the distribution of the letters in advance. This might be true if we were always compressing normal English text, but in general is not possible to do.
A second option, and the option used in the assignment, is to prefix the bit sequence with a header containing enough information to reconstruct the Huffman encoding tree. There are many options you have for reading and writing the encoding table. You could store the table at the head of the file in a long, Huffman-readable string format using the ASCII characters '0' and '1', one entry per line, like this:
h = 01
a = 000
p = 10
y = 1111
Reading this back in would allow you to recreate the tree path by path. You could have a line for every character in the ASCII set; characters that are unused would have an empty bit pattern. Or you could conserve space by only listing those characters that appear in the encoding. In such a case you must record a number that tells how many entries are in the table or put some sort of sentinel or marker at the end so you know when you have processed them all. As an alternative to storing sequences of ASCII '0' and '1' characters for the bit patterns, you could
store just the character frequency counts and rebuild the tree again from those counts in order to decompress. Again, we might include the counts for all characters (including those that are zero) or optimize to only record the non-zero ones. Here is how we might encode the non-zero counts for the "happy hip
hop" string (the 7 at the front says there are 7 entries to follow -6 alphabetic characters, and space):
7 h3 a1 p4 y1 2 i1 o1
Note that in the current version of the assignment, the exact header format is specified, and you should follow the header format from the spec.

Greedy Algorithms

Huffman's algorithm is an example of a greedy algorithm. In general, greedy algorithms use small-grained or local minimal/maximal choices in attempt to result in a global minimum/maximum. At each step, the algorithm makes the near choice that appears to lead toward the goal in the long-term. In Huffman's case, the insight for its strategy is that combining the two smallest nodes makes both of those character encodings one bit longer (because of the added parent node above them) but given these are the rarest characters, it is a better choice to assign them longer bit patterns than the more frequent characters. The Huffman strategy does, in fact, lead to an overall optimal character encoding.

Even when a greedy strategy may not result in the overall best result, it still can be used to approximate when the optimal solution requires an exhaustive or expensive traversal. In a time or space-constrained situation, we might be willing to accept the quick and easy-to-find greedy solution as an approximation.

4. OTHER ALGORITHMS

Other lossless compression algorithms include:

1. Run Length Encoding (RLE)

   Run Length encoding (RLE) is a lossless data compression algorithm, supported by many bitmaps file formats, like BMP, TIFF and PCX.

   Run Length encoding follows a straightforward logic, it just picks the next unique character and appends the character and it's count of subsequent occurrences in the encoded string. Runs here refer to the consecutive occurrences of the same character. The idea is to reduce the physical total size of repeating characters in the data.

2. LZ77
   LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a *length-distance pair*, which is equivalent to the statement "each of the next *length* characters is equal to the characters exactly *distance* characters behind it in the uncompressed stream". (The *distance* is sometimes called the *offset* instead.)

   To spot matches, the encoder must keep track of some amount of the most recent data, such as the last 2 kB, 4 kB, or 32 kB. The structure in which this data is held is called a *sliding window*, which is why LZ77 is sometimes called *sliding-window compression*. The encoder needs to keep this data to look for matches, and the decoder needs to keep this data to interpret the matches the encoder refers to. The larger the sliding window is, the longer back the encoder may search for creating references.

3. Lempel-Ziv-Welch
   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. It is the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format.

5.IMPLEMENTATION

BUILDING HUFFMAN TABLE

1. Get the unique characters from a file and the number of times they appear(frequency).
2. Build the Huffman table that is, a table of characters and frequency

BUILDING HUFFMAN TREE

Implement a minimum heap from the Huffman table. That is, the least occurring character is at the root. The input is an array of unique characters along with their frequency of occurrence and the output is a Huffman tree.

1. Create a leaf node for each unique character and build a minimum heap of all leaf nodes (min heap is used as a priority queue). The value of the frequency field is used to compare the nodes in the heap. Initially the least frequent character is at the root.
2. Extract two nodes with the minimum frequency from the minimum heap. That is the root and the smaller of its two children.
3. Create a new internal node with the frequency equal to the sum of the two extracted nodes frequencies. Make the first extracted node the left child and the other extracted node the its right child. Its value should be set to NULL.
4. Insert this node to the min heap
5. Repeat steps 2 and 3 until the heap contains only one node which is the root node and the tree is complete.

```
Algorithm ExtractMinHeap
Input: minMiheap
Output:  two nodes with the minimum frequencies from the minimum heap
    Create a temporary node and assign it to the root of the minimum heap
    Set the root of the heap to the last node in the heap
    Reduce the size of the heap by 1
    Heapify the heap. reconstruct the heap passing the root
```

```
        Return temporary node
        Get the next node which is now the root and reduce the
        heap by 1 then Heapify
```

The heap size is now reduced by 2.

Algorithm Insert

Input: minheap, Newnode

Inserting the new internal node into the minimum heap. It simply finds a place where the new node will be inserted.

```
        increase the size of the minimum heap by 1. For each new
        internal node size is always reduced by 1

        declare a variable index of last node i

        assign i to the index of the last heap node
        while i and new internal node's frequency is smaller than
        frequency of the parent of the ith node
            copy parent of i to i
            set i to the index of the parent

        create a duplicate of the parent of the ith node then
        compare the new nodes frequency with the next parent of
        the ith node until it finds a place to sit or the position of
        insertion.

        End the loop

    Insert the new internal node at the ith position that met the
    condition.
```

Algorithm BuildHuffmanTree

Builds the Huffman tree using the utility algorithms described above.

```
    Build the heap

    While size not equal 1

        Extract the left node
```

```
            Extract the right node
            Build a new internal node
            Insert the new internal node
        End loop
        Extract the last remaining node
        Insert it into the tree

    You have your Huffman tree
```

Algorithm Getcodes

After building the Huffman tree, you move down the tree until you get to the leaf with the character.

To obtain the codes for each character, each move to the left is a 0 and to the right is 1. Start from the root each time to obtain the code of the leaves in the tree which are basically the unique characters.

Another approach

Huffman tree can also be created without implementing a minimum heap. Here's is another approach;

1. Create the Huffman table containing the character and the frequency
2. Create an array of struct node each containing 5 elements, the frequency, character, left and right children, mark(true or false)

3. Get the two smallest nodes
4. Of the two, select one for replacement (REPLACE) then make a copy of it. Make this copy its left child and the other node the right child
5. Mark it TRUE. The one copied is not marked as true.

6. Sum the frequencies of the two children to obtain the frequency of REPLACE

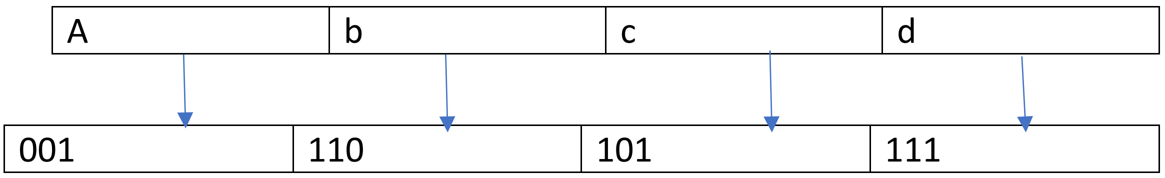
This implementation is inefficient since the encoding algorithm have to search the array to get the node that was not marked true

ENCODING

Encoding is the assigning of Hoffman codes to the unique characters. It involves traversing the tree. if you traverse to the left a 0 is added to the codes array, if you traverse to the right a 1 is added to the codes array. When you get to a leaf then the Huffman code has been found for the character at the leaf.

Convert the codes into string and store in an array. The new codes array holds code at corresponding position as unique character array. To encode a file;

i. the file is read
ii. unique characters and their frequencies are found
iii. build the Huffman tree
iv. generate the Huffman codes as explained above
v. read the file again and this time a character at a time. The character is searched in the character array
vi. the index of the character returned is used to get the code at the corresponding position in the codes array.



The new found code is written and used instead of the characters ASCII code. When the reduced Huffman codes are written to disks instead of the original ASCII codes, the size of the final file is greatly reduced in size. In our case the happy hip hop is originally 13bytes but after writing to disk it becomes 5bytes saving 8bytes.

DECODING

Decoding is as simple as reading the stream of bytes (Huffman bits) from the disk and finding the corresponding characters.

How decoding works

i. Using the key and the characters, reconstruct the Huffman tree. Huffman code is a greedy algorithm and the keys used to encode the file must be used during the decoding. In this case the frequency is not used.
ii. Read bit, if 0 make the left child of the root the root. Get the next bit. If 0, move left otherwise move right.
iii. If the node is a leaf, retrieve the character at the node
iv. Buffer the character to a temporary storage before writing to disk or write to a new file one character at a time.

It all depends on the speed of the buffer I/O over writing each and every time. When the total bits written during encoding is reached, the trailing zeros used for padding are ignored and file closed.

6.OUR IMPLEMENTATION LIMITATION

Challenge with encoding

During the first parse of the file to get the characters and their frequencies, the file is stored in a buffer. It is assumed to fit in memory. If the file is large, this might cause memory problems and program crash.

Proposed Solution

The file character can be read one at a time and their frequency updated each time a character is encountered and store the file in memory. Searching is implemented for check if the character already exists in the array and its corresponding frequency updated.

The encoded file is stored in disk with corresponding keys to help with decoding. If the encoded file bits are not a multiple of 8, add trailing zeros to make it a multiple of 8, this is useful during decoding. It is also good practice to return total bits written minus the trailing zeros.

7.PROBLEM WITH COMPRESSION

Data Compression is not useful in all the cases. Depending on the size of the data, performance requirements and whether the data is encrypted data ,compressions may not be the ideal solution in certain cases. There is a CPU overhead associated with using data compression and compressing the data may adversely impact the performance of the system in certain cases. High volume OLTP databases could be significantly impacted when attempting to implement data compression.

Annex

Huffman story

The story of the invention of Huffman codes is a great story that demonstrates that students can do better than professors. David Huffman (1925-1999) was a student in an electrical engineering course in 1951. His professor, Robert Fano, offered students a choice of taking a final exam or writing a term paper. Huffman did not want to take the final exam so he started working on the term paper.

The topic of the paper was to find the most efficient (optimal) code. What Professor Fano did not tell his students was the fact that it was an open problem and that he was working on the problem himself. Huffman spent a lot of time on the problem and was ready to give up when the solution suddenly came to him. The code he discovered was optimal, that is, it had the lowest possible average message length. The method that Fano had developed for this problem did not always produce an optimal code. Therefore, Huffman did better than his professor. Later Huffman said that likely he would not have even attempted the problem if he had known that his professor was struggling with it.

The project, "Discovery of Huffman Codes," uses excerpts from Fano's work and from Huffman's paper, where they present their encodings. Both Fano and Huffman used greedy strategies to find the codes. However, Fano's greedy algorithm would not always produce an optimal code while Huffman's greedy algorithm would always find an optimal solution. The purpose of the project is for students to learn greedy algorithms, prefix-free codes, Huffman encoding, binary tree representations of codes, and the basics of information theory (unit and amount of information). The project demonstrates that a greedy strategy could be applied in different ways to the same problem, sometimes producing an optimal solution and sometimes not.