Assignment 6: Huffman Coding

Thanks to Owen Astrachan (Duke) and Julie Zelenski for creating the assignment.
Edits over time by Keith Schwarz, Stuart Reges (UW), Marty Stepp, and Cynthia Lee.

Due: Friday, May 20th at 6:00pm

Pair programming is permitted on this assignment. See course information sheet and honor code.

We discussed Huffman Coding background and implementation details in class on Monday, May 9. If you missed it, be sure to go back and check the slides and video!

The assignment has four main purposes:

1. To encourage a new view of computer systems, by exploring how computers operate in binary at the lowest levels, including considering the universality of binary and the multitude of encoding/interpretation systems layered on top of binary.
2. To give you experience implementing a node-based tree structure (your previous experience with binary heaps was array-based, not node-based).
3. To give you experience implementing a theoretically interesting and non-trivial algorithm.
4. To give you a behind the scenes look at one approach to the process of compression, a tool used in everyday life (e.g., zip files, mp3 files, video files).

Deliverables (turn in these files):

*   `encoding.cpp:` code to perform Huffman encoding and decoding
*   `secretmessage.huf:` a message from you to your section leader, which is compressed by your algorithm
*   Do not modify the other files in the starter code, except to (optionally) add new top level menu options to support extensions. Do not modify behavior of any existing menu.

This is a pair assignment. If you work as a pair, comment both members' names on top of every submitted code file. Only one of you should submit the assignment; do not turn in two copies.

HUFFMAN CODING

Huffman coding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing text data to make a file smaller (fewer bytes). This relatively simple algorithm is powerful enough that variations of it are still used today in computer networks, fax machines, modems, HDTV, and other areas.

Normally text data is stored in a standard format of 8 bits per character using an encoding called ASCII that maps every character to a binary integer value from 0-255. The idea of Huffman coding is to abandon the rigid 8-bits-per-character requirement and use variable-length binary encodings for different characters. The advantage of doing this is that very frequently occurring

characters (say "e" in English text) could be given very short encodings (fewer bits). Although this means some characters may need more than 8 bits, the tradeoff is often worth it because those characters occur very infrequently. **Refer to the Huffman lecture for introductory examples.**

The steps involved in Huffman encoding are:

1. **Count character frequencies:** Make a quick pass through the file to be encoded, counting how many of each distinct character you find in the text.
2. **Build encoding tree:** Build a binary tree using a specific queue-based algorithm.
3. **Build encoding map:** Traverse the binary tree to record the binary encodings of each character in a map for future quick-access use.
4. **Encode the file:** Do a second pass through the file, looking up each byte (each character) in the map from Step 3, to get the encoding for each byte.

STEP 1: COUNT CHARACTER FREQUENCIES
Suppose we have a file named example.txt, whose contents are: `"ab ab cab"`:

In the original file, this text occupies 10 bytes (80 bits) of data. The 10th is a special "end-of-file" (EOF) byte.

byte	1	2	3	4	5	6	7	8	9	10
char	`'a'`	`'b'`	`' '`	`'a'`	`'b'`	`' '`	`'c'`	`'a'`	`'b'`	EOF
ASCII	97	98	32	97	98	32	99	97	98	256
binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010	N/A

In Step 1 of Huffman's algorithm, a count of each character is computed. The counts are represented as a map: `{' ':2, 'a':3, 'b':3, 'c':1, EOF:1}`
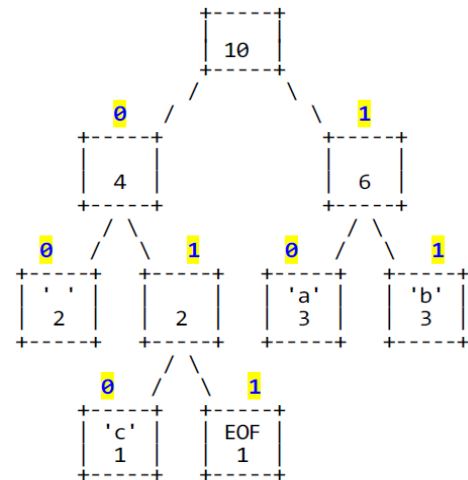
STEP 2: BUILD ENCODING TREE
Step 2 of Huffman's algorithm places our counts into binary tree nodes, with each node storing a character and a count of its occurrences. The nodes are then put into a **priority queue** (yay, Assignment 5 memories!), which keeps them in prioritized order with smaller counts having higher priority, so that characters with lower counts will come out of the queue sooner. (The priority queue is somewhat arbitrary in how it breaks ties, such as `'c'` being before EOF and `'a'` being before `'b'`).

```
front                                    back
+-----+   +-----+   +-----+   +-----+   +-----+
| 'c' |   | EOF |   | ' ' |   | 'a' |   | 'b' |
|  1  |   |  1  |   |  2  |   |  3  |   |  3  |
+-----+   +-----+   +-----+   +-----+   +-----+
```

Now the algorithm repeatedly removes the two nodes from the front of the queue (the two with the smallest frequencies) and **joins** them into a new node whose frequency is their sum. The two

nodes are placed as children of the new node; the first removed becomes the left child, and the second the right. The new node is re-inserted into the queue in sorted order. This process is repeated until the queue contains only one binary tree node with all the others as its children. This will be the root of our finished Huffman tree. The diagram at right shows the final resulting tree. **(See Huffman lecture slides for a step-by-step diagrams of this example!)**

Notice that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. This structure can be used to create an efficient encoding in the next step.

STEP 3: BUILD ENCODING MAP

Step 3 of Huffman's algorithm takes the tree from Step 2 (see diagram above) and performs a traversal of it to harvest all the binary sequences that will be the encodings for each character.

The binary sequence for each character is derived from your binary tree by thinking of each left branch as a bit value of 0 and each right branch as a bit value of 1, as shown in the diagram at above. The code for each character can be determined by traversing the tree. To reach `' '` [space], we go left twice from the root, so the code for `' '` is `00`. The code for `'c'` is `010`, the code for `EOF` is `011`, the code for `'a'` is `10` and the code for `'b'` is `11`. By traversing the tree, we can produce a map from characters to their binary representations. Though the binary representations are integers, since they consist of binary digits and can be arbitrary length, we will store them as strings. For this tree, it would be:

`{' ':"00", 'a':"10", 'b':"11", 'c':"010", EOF:"011"}`

STEP 4: ENCODING THE TEXT DATA

Using the encoding map from Step 3, we can encode the file's contents into a shorter binary representation. Using the preceding encoding map, the text `"ab ab cab"` would be encoded as `1011001011000101011011`.

The following table gives the char-to-binary mapping in more detail. The overall encoded contents of the file require 22 bits, or almost 3 bytes, compared to the original file of 10 bytes.

char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
binary	10	11	00	10	11	00	010	10	11	011

Since the character encodings have different lengths, often the length of a Huffman-encoded file does not come out to an exact multiple of 8 bits. Files are stored as sequences of whole bytes, so

in cases like this the remaining digits of the last bit are filled with 0s. You do not need to worry about this; it is part of the underlying file system.

byte	1	2	3
char	a b a	b c a	b EOF
binary	<u>10</u> <u>11</u> <u>00</u> <u>10</u>	<u>11</u> <u>00</u> <u>010</u> 1	0 <u>11</u> <u>011</u> 00

It might worry you that the characters are stored without any delimiters between them, since their encodings can be different lengths and characters can cross byte boundaries, as with `'a'` at the end of the second byte. But this will not cause problems in decoding the file, because Huffman encodings by definition have a useful prefix property where no character's encoding can ever occur as the start of another's encoding. **(See the end of the lecture video for a discussion of this point.)**

DECODING THE FILE

A compressed file would not be very useful unless we have the ability to decompress it again! You can use a Huffman tree to decode text that was previously encoded with its binary patterns. The decoding algorithm is to read each bit from the file, one at a time, and use this bit to traverse the Huffman tree. If the bit is a 0, you move left in the tree. If the bit is 1, you move right. You do this until you hit a leaf node. Leaf nodes represent characters, so once you reach a leaf, you output that character. For example, suppose we are given the same encoding tree above, and we are asked to decode a file containing the following bits: `1110010001001010011`

Using the Huffman tree, we walk from the root until we find characters, then output them and go back to the root.

- We read a 1 (right), then a 1 (right). We reach `'b'` and output b. Back to the root.
  `1110010001001010011`
- We read a 1 (right), then a 0 (left). We reach `'a'` and output a. Back to root.
  `1110010001001010011`
- We read a 0 (left), then a 1 (right), then a 0 (left). We reach `'c'` and output c.
  `1110010001001010011`
- We read a 0 (left), then a 0 (left). We reach `' '` and output a space.
  `1110010001001010011`
- We read a 1 (right), then a 0 (left). We reach `'a'` and output a.
  `1110010001001010011`
- We read a 0 (left), then a 1 (right), then a 0 (left). We reach `'c'` and output c.
  `1110010001001010011`
- We read a 1 (right), then a 0 (left). We reach `'a'` and output a.
  `1110010001001010011`
- We read a 0, 1, 1. This is our `EOF` encoding pattern, so we stop. The overall decoded text is `"bac aca"`.

PROVIDED CODE

We provide you with a file `HuffmanNode.h` that declares some useful support code including the `HuffmanNode` structure, which represents a node in a Huffman encoding tree.

```
struct HuffmanNode {
    int character;      // character being represented by this node
    int count;          // number of occurrences of that character
    HuffmanNode* zero;  // 0 (left) subtree (NULL if empty)
    HuffmanNode* one;   // 1 (right) subtree (NULL if empty)
    ...
};
```

!! IMPORTANT !! The character field is declared as type `int`, but you should think of it as a `char`. Probably the #1 FAQ for this assignment on Piazza involves not following this not only in the struct, but throughout the code that interacts with the struct (i.e., temporary variables for holding the character). Types `char` and `int` are largely interchangeable in C++, but using `int` here allows us to sometimes use character to store values outside the normal range of `char`, for use as special flags.) The character field can take one of three types of values:

- an actual `char` value;
- the constant `PSEUDO_EOF` (defined in `bitstream.h` in the Stanford library), which represents the pseudo-EOF value (the symbol, denoted by ■ in the supplemental Huffman handout, that marks the end of the encoding) that you will need to place at the end of an encoded stream; or
- the constant `NOT_A_CHAR` (defined in `bitstream.h` in the Stanford library), which represents something that isn't actually a character. **This can be stored in non-leaf nodes of the Huffman encoding tree**, because such nodes do not represent any one individual character. *(In our lecture slides, these simply had an empty where the character would be.)*

BIT INPUT/OUTPUT STREAMS

In parts of this program you will need to read and write bits to files. In the past we have wanted to read input an entire line or word at a time, but in this program it is much better to read one single character (byte) at a time. So you should use the following in/output stream functions:

ostream (output stream) member	Description
void **put**(int byte)	writes a single *byte* (character, 8 bits) to the output stream

istream (input stream) member	Description
int **get**()	reads a single *byte* (character, 8 bits) from input; -1 at EOF

You might also find that you want to read an input stream, then "rewind" it back to the start and read it again. To do this on an input stream variable named input, you can use the `rewindStream` function from `filelib.h`:

```
rewindStream(input);   // tells the stream to seek back to the beginning
```

To read or write a compressed file, even a whole byte is too much; you will want to read and write binary data one single bit at a time, which is not directly supported by the standard in/output streams in C++. Therefore the Stanford C++ library provides `obitstream` and `ibitstream` classes with `writeBit` and `readBit` members to make it easier.

`obitstream` (bit output stream) member	Description
void **writeBit**(int bit)	writes a single *bit* (0 or 1) to the output stream

`ibitstream` (bit input stream) member	Description
int **readBit**()	reads a single *bit* (0 or 1) from input; -1 at end of file

When reading from an bit input stream (`ibitstream`), you can detect the end of the file by either looking for a `readBit` result of -1, or by calling the `fail()` member function on the input stream after trying to read from it, which will return true if the last `readBit` call was unsuccessful due to reaching the end of the file.

Note that the bit in/output streams also provide the same members as the original `ostream` and `istream` classes from the C++ standard library, such as `getline, <<, >>,` etc. But you usually don't want to use them, because they operate on an entire byte (8 bits) at a time, or more; whereas you want to process these streams one bit at a time.

IMPLEMENTATION DETAILS
In this assignment you will write the following functions in the file `encoding.cpp` to encode and decode data using the Huffman algorithm described previously. Our provided main client program will allow you to test each function one at a time before moving on to the next. You must write the following functions; you can add more functions as helpers if you like, particularly to help you implement any recursive algorithms. Any members that traverse a binary tree from top to bottom should implement that traversal recursively whenever practical.

`Map<int, int> buildFrequencyTable(istream& input)`
- This is **Step 1** of the encoding process. In this function you read input from a given `istream` (which could be a file on disk, a string buffer, etc.). You should count and return a mapping from each character (represented as `int` here) to the number of times that character appears in the file. You should also add a single occurrence of the fake character `PSEUDO_EOF` into your map. You may assume that the input file exists and can be read, though the file might be empty. An empty file would cause you to return a map containing only the 1 occurrence of `PSEUDO_EOF`.

`HuffmanNode* buildEncodingTree(Map<int, int> freqTable)`
- This is **Step 2** of the encoding process. In this function you will accept a frequency table (like the one you built in `buildFrequencyTable`) and use it to create a Huffman encoding tree based on those frequencies. Return a pointer to the node representing the root of the tree.
- You may assume that the frequency table is valid: that it does not contain any keys other than `char` values, `PSEUDO_EOF`, and `NOT_A_CHAR`; all counts are positive integers; it contains at least one key/value pairing; etc.
- When building the encoding tree, you will need to use a **priority queue** to keep track of which nodes to process next. Use the `PriorityQueue` collection provided by the Stanford libraries, defined in library header **pqueue.h**. This priority queue allows each element to be enqueued along with an associated priority. The priority queue then sorts elements by their priority, with the `dequeue` function always returning the element with the minimum priority number. Consult the docs on the course website and lecture slides for more information about priority queues.

`Map<int, string> buildEncodingMap(HuffmanNode* encodingTree)`
- This is **Step 3** of the encoding process. In this function will you accept a pointer to the root node of a Huffman tree (like the one you built in `buildEncodingTree`) and use it to create and return a Huffman encoding map based on the tree's structure. Each key in the map is a character, and each value is the binary encoding for that character represented as a string. For example, if the character `'a'` has binary value `10` and `'b'` has `11`, the map should store the key/value pairs `'a':"10"` and `'b':"11"`. If the encoding tree is `NULL`, return an empty map.

`void encodeData(istream& input, const Map<int, string>& encodingMap, obitstream& output)`
- This is **Step 4** of the encoding process. In this function you will read one character at a time from a given input file, and use the provided encoding map to encode each character to binary, then write the character's encoded binary bits to the given bit output bit stream. After writing the file's contents, you should write a single occurrence of the binary encoding for `PSEUDO_EOF` into the output so that you'll be able to identify the end of the data when decompressing the file later. You may assume that the parameters are valid: that the encoding map is valid and contains all needed data, that the input stream is readable, and that the output stream is writable. The streams are already opened and ready to be read/written; you do not need to prompt the user or open/close the files yourself.

`void decodeData(ibitstream& input, HuffmanNode* encodingTree, ostream& output)`
- This is the **"decoding a file"** process described previously. In this function you should do the opposite of encodeData; you read bits from the given input file one at a time, and <u>recursively</u> walk through the specified decoding tree to write the original uncompressed

contents of that file to the given output stream. The streams are already opened and you do not need to prompt the user or open/close the files yourself.

To manually verify that your implementations of `encodeData` and `decodeData` are working correctly, use our provided test code to compress strings of your choice into a sequence of 0s and 1s. The next page describes a header that you will add to compressed files, but in `encodeData` and `decodeData`, you should not write or read this header from the file. Instead, just use the encoding tree you're given. Worry about headers only in `compress/decompress`.
The functions on the previous page implement Huffman's algorithm, but they have one big flaw. The decoding function requires the encoding tree to be passed in as a parameter. Without the encoding tree, you don't know the mappings from bit patterns to characters, so you can't successfully decode the file.

We will work around this by writing the encodings into the compressed file, as a **header**. The idea is that when opening our compressed file later, the first several bytes will store our encoding information, and then those bytes are immediately followed by the compressed binary bits that we compressed earlier. It's actually easier to store the character frequency table, the map from **Step 1** of the encoding process, and we can generate the encoding tree from that. For our `ab ab cab` example, the frequency table stores the following (the keys are shown by their ASCII integer values, such as `32` for `' '` and `97` for `'a'`, because that is the way the map would look if you printed it out): `{32:2, 97:3, 98:3, 99:1, 256:1}`

We don't have to write the encoding header bit-by-bit; just write out normal ASCII characters for our encodings. We could come up with various ways to format the encoding text, but this would require us to carefully write code to write/read the encoding text. There's a simpler way. You already have a `Map` of character frequency counts from Step 1 of encoding. In C++, collections like `Map`s can easily be read and written to/from streams using `<<` and `>>` operators. So all you need to do for your header is write your map into the bit output stream first before you start writing bits into the compressed file, and read that same map back in first later when you decompress it. The overall file is now 34 bytes: 31 for the header and 3 for the binary compressed data. Here's an attempt at a diagram:

byte	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	'{'	'3'	'2'	':'	'2'	','	' '	'9'	'7'	':'	'3'	','	' '	'9'	'8'	':'	'3'
byte	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
	','	' '	'9'	'9'	':'	'1'	','	' '	'2'	'5'	'6'	':'	'1'	'}'	*	*	*

```
// output << frequencyTable;      input >> frequencyTable;
// output.writeBit(...);          input.readBit(); ...          10110010  11000101  01101100
```

Looking at this new rendition of the compressed file, you may be thinking, "The file is not compressed at all; it actually got *larger* than it was before! It went up from 9 bytes (`"ab ab cab"`) to 34!" That's true for this contrived example. But for a larger file, the cost of the header is not so bad relative to the overall file size. There are more compact ways of storing the header, too, but they add too much challenge to this assignment, which is meant to practice trees and data structures and problem solving more than it is meant to produce a truly tight compression.

The last step is to glue all of your code together, along with code to read and write the encoding table to the file:

void compress(istream& input, obitstream& output)

- This is the overall compression function; in this function you should compress the given input file into the given output file. You will take as parameters an input file that should be encoded and an output bit stream to which the compressed bits of that input file should be written. You should read the input file one character at a time, building an encoding of its contents, and write a compressed version of that input file, including a header, to the specified output file. This function should be built on top of the other encoding functions and should call them as needed. You may assume that the streams are both valid and read/writeable, but the input file might be empty. The streams are already opened and ready to be read/written; you do not need to prompt the user or open/close the files yourself.

void decompress(ibitstream& input, ostream& output)

- This function should do the opposite of compress; it should read the bits from the given input file one at a time, including your header packed inside the start of the file, to write the original contents of that file to the file specified by the output parameter. You may assume that the streams are valid and read/writeable, but the input file might be empty. The streams are already open and ready to be used; you do not need to prompt the user or open/close files.

void freeTree(HuffmanNode* node)

- This function should free the memory associated with the tree whose root node is represented by the given pointer. You must free the root node and all nodes in its subtrees. There should be no effect if the tree passed is NULL. If your compress or decompress function creates a Huffman tree, that function should also free the tree.

CREATIVE ASPECT

Along with your program, turn in a file secretmessage.huf that stores a compressed message from you to your section leader. Create the file by compressing a text file with your compress function. The message can be anything non-offensive you want. Your SL will decompress your message with your program and read it while grading.

IMPLEMENTATION STRATEGY

- When writing the bit patterns to the compressed file, note that you do not write the ASCII characters '0' and '1' (that wouldn't do much for compression!), instead the *bits* in the compressed form are written one-by-one using the readBit and writeBit member functions on the bitstream objects. Similarly, when you are trying to read bits from a compressed file, don't use >> or byte-based methods like get or getline; use readBit. The bits that are returned from readBit will be either 0 or 1, but not '0' or '1'.

- Work **step-by-step**. Get each part of the encoding program working before starting on the next one. You can test each function individually using our provided client program, even if others are blank or incomplete.
- Start out with **small test files** (two characters, ten characters, one sentence) to practice on before you start trying to compress large books of text. What sort of files do you expect Huffman to be particularly effective at compressing? On what sort of files will it less effective? Are there files that grow instead of shrink when Huffman encoded? Consider creating sample files to test out your theories.
- Your implementation should be robust enough to compress **any kind of file**: text, binary, image, or even one it has previously compressed. Your program probably won't be able to further squish an already compressed file (and in fact, it can get larger because of header overhead) but it should be possible to compress multiple iterations, decompress the same number of iterations, and return to the original file.
- Your program only has to decompress valid files compressed by your program. You do not need to take special precautions to protect against user error such as trying to decompress a file that isn't in the proper compressed format.
- See the input/output streams section for how to **"rewind" a stream** to the beginning if necessary.
- The operations that read and write bits are somewhat inefficient and working on a large file (100K and more) will take some time. Don't be concerned if the reading/writing phase is **slow** for very large files.
- Note that **Qt Creator** puts the compressed binary files created by your code in your "build" folder. They won't show up in the normal **res** resource folder of your project.

GRADING NOTES

In general, items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignments about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Don't forget to cite any sources you used in your comments. Refer to the course Style Guide for a more thorough discussion of good coding style.

- Part of your grade will come from appropriately utilizing binary trees and recursive algorithms to traverse them. We will check that you use collections in appropriate, efficient ways. Do not pass bulky objects by value since that leads to unnecessary copying. Redundancy is another major grading focus; avoid repeated logic as much as possible.
- Your code should have no memory leaks. Free the memory associated with any new objects you allocate internally. The Huffman nodes you will allocate when building encoding trees are passed back to the caller, so it is that caller's responsibility to call your freeTree function to clean up the memory.
- As for commenting, place a descriptive comment header on each file you submit. Place detailed comment headers next to every function explaining its purpose, parameters, what it returns, any exceptions it throws, assumptions it makes, etc. Also place inline comments as needed on any complex code inside the function bodies.

Please remember to follow the Honor Code on this assignment. Submit your own (pair's) work; do not look at others' solutions. Cite sources. Do not give out your solution; do not place a solution on a public web site or forum.

SUGGESTED EXTENSIONS

There are all sorts of fun extras you can layer on top of this assignment. Here are a few things to consider:

- Make the encoding table more efficient: Our implementation of the encoding table at the start of each file is not at all efficient, and for small files can take up a lot of space. Try to see if you can find a better way of encoding the data. If you're feeling up for a challenge, try looking up succinct data structures and see if you can write out the encoding tree using one bit per node and one byte per character!

- Add support for encryption in addition to encoding: Without knowledge of the encoding table, it's impossible to decode compressed files. Update the encoding table code so that it prompts for a password or uses some other technique to make it hard for Bad People to decompress the data.

- Implement a more advanced compression algorithm: Huffman encoding is a good compression algorithm, but there are much better alternatives in many cases. Try researching and implementing a more advanced algorithm, like LZW, in addition to Huffman coding.

- Gracefully handle bad input files: The normal version of the program doesn't work very well if you feed it bogus input, such as a file that wasn't created by your own algorithm. Make your code more robust by making it able to detect whether a file is valid or invalid and react accordingly. One possible way of doing this would be to insert special bits/bytes near the start of the file that indicate a header flag or check-sum. You can test to see whether these bit patterns are present, and if not, you know the file is bogus.

- Other: If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extras that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your extras cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit your program twice: a first time without any extra features added (or with all necessary extras disabled or commented out), and a second time with the extras enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.