CS 133, Spring 2023
Programming Project #5: Huffman Coding (20 points)
Due Monday, May 22, 2023, 11:59 PM

This program focuses on binary trees, priority queues, and input/output. Turn in files named `HuffmanTree.h`, `HuffmanTree.cpp`, `HuffmanNode.h`, `HuffmanNode.cpp`, `secretmessage.huf`, and `secretmessage.counts` from the Projects section of the web site. You will need support files `huffmanMain.cpp`, `huffmanConstants.cpp`, `BitStream.h`, `BitStream.cpp`, and input files from the course web page.

Huffman Coding:

Huffman coding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing text data to make a file occupy a smaller number of bytes. This relatively simple compression 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, commonly 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 different-length binary encodings for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the letter `'e'`, it could be given a shorter encoding (fewer bits), making the file smaller. The tradeoff is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so the extra cost is worth it.

The table below compares ASCII values of various characters to possible Huffman encodings for the text of Shakespeare's *Hamlet*. Frequent characters such as space and `'e'` have short encodings, while rarer ones like `'z'` have longer ones.

Character	ASCII value	ASCII (binary)	Huffman (binary)
`' '`	32	00100000	10
`'a'`	97	01100001	0001
`'b'`	98	01100010	0111010
`'c'`	99	01100011	001100
`'e'`	101	01100101	1100
`'z'`	122	01111010	00100011010

The steps involved in Huffman coding a given text source file into a destination compressed file are the following:

1. Examine the source file's contents and count the number of occurrences of each character.
2. Place each character and its frequency (count of occurrences) into a sorted "priority" queue.
3. Convert the contents of this priority queue into a binary tree with a particular structure.
4. Traverse the tree to discover the binary encodings of each character.
5. Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file.

Encoding a File:

For example, suppose we have a file named `example.txt` with the following contents:

```
ab ab cab
```

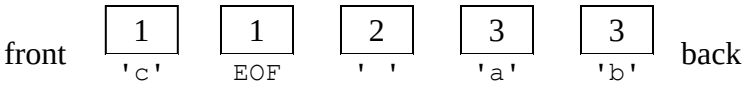
In the original file, this text occupies 10 bytes (80 bits) of data. The 10th is a special "end-of-file" (EOF) byte.

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

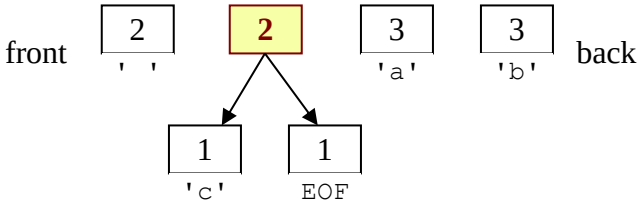
In Step 1 of Huffman's algorithm, a count of each character is computed. (In this assignment, our provided client program does this part for you, so you don't need to do it yourself.) The counts are represented as a map:

```
{' '=2, 'a'=3, 'b'=3, 'c'=1, EOF=1}
```

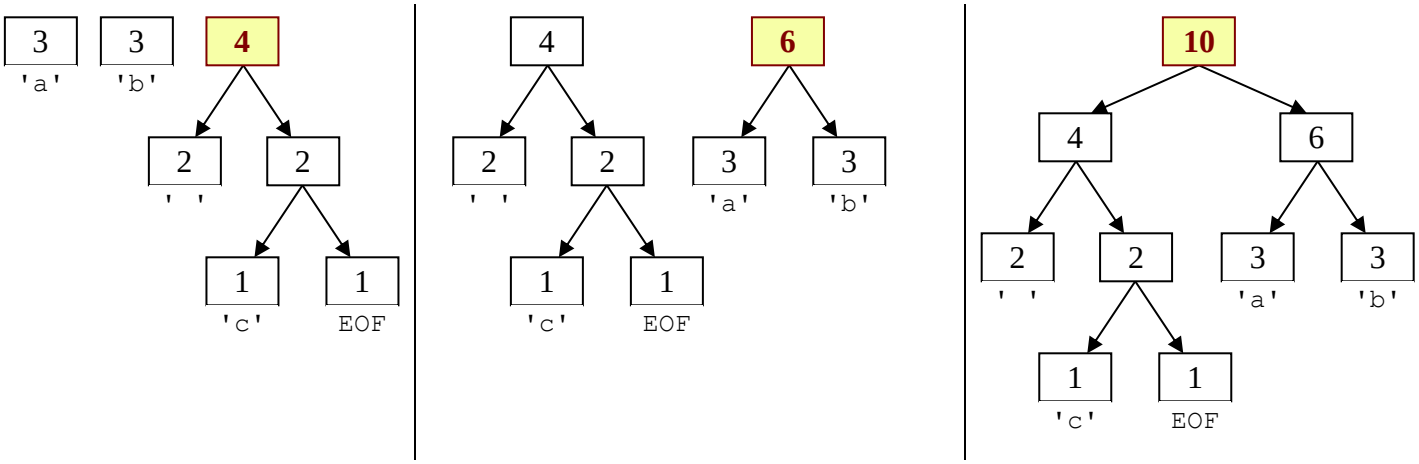
Step 2 of the algorithm places these counts into binary tree nodes, each storing a character and a count of its occurrences. The nodes are put into a priority queue, which keeps them in sorted order with smaller counts at the front of the queue. (Let the priority queue order equal counts as it chooses).



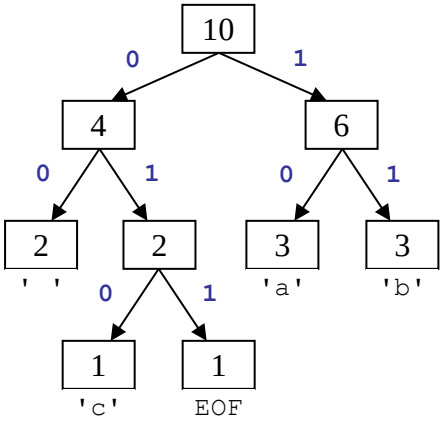
Now the algorithm repeatedly removes the two nodes from the front of the queue (the two with the smallest frequencies) and joins them into a new node whose frequency is their sum. The two nodes are placed as children of the new node; the first removed becomes the left child, and the second the right. 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 following diagram shows this process:



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. The Huffman code is derived from this tree by thinking of each left branch as a bit value of 0 and each right branch as a bit value of 1:



The code for each character can be determined by traversing the tree. To reach `' '` 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. For this tree, it would be:

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

Using this map, we can encode the file into a shorter binary representation. The text `ab ab cab` would be encoded as:

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

The overall encoded contents of the file are `10110010110001010011011`, which is 22 bits, or almost 3 bytes, compared to the original file which was 10 bytes. (Many Huffman-encoded text files compress to about half their original size.)

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

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 byte are filled with 0s. You do not need to worry about this in the assignment; it is part of the underlying file system.

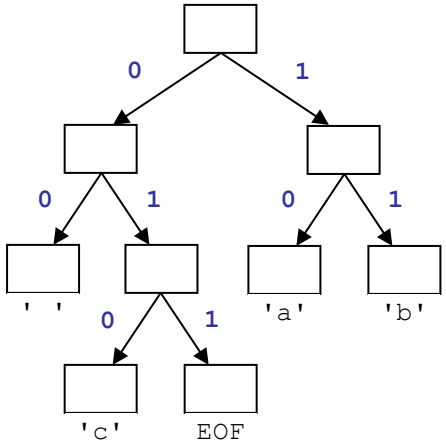
It might worry you that the characters are stored without any delimiters between them, since their encodings can be different lengths and characters can cross byte boundaries, as with `'a'` at the end of the second byte above. But this will not cause problems in decoding the compressed file, because Huffman encodings have a *prefix property* where no character's encoding can ever occur as the start of another's encoding. This is important when you decode the file later.

Decoding a File:

You can use a Huffman tree to decode text that was compressed with its encodings. 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 asked to decode a file containing the following bits:

```
101101000110111011
```

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



- First we read a `1` (right), then a `0` (left). We reach `'a'` and output `a`. Back to the root. `10`110100011011011
- We read a `1` (right), then a `1` (right). We reach `'b'` and output `b`. 10`11`01000110111011
- We read a `0` (left), then a `1` (right), then a `0` (left). We reach `'c'` and output `c`. 1011`010`00110111011
- We read a `0` (left), then a `0` (left). We reach `' '` and output a space. 101101`00`110111011
- We read a `1` (right), then a `1` (right). We reach `'b'` and output `b`. 10110100`11`0111011
- We read a `0` (left), then a `1` (right), then a `0` (left). We reach `'c'` and output `c`. 101101000`110`111011
- We read a `1` (right), then a `1` (right). We reach `'b'` and output `b`. 1011010001101`11`011
- We read a `0` (left), then a `1` (right), then a `1` (right). We reach `EOF` and stop. 10110100011011`011`

So the overall decoded text is `abc bcb`. (Notice that we don't print `EOF`)

Implementation Details:

In this assignment you will create a class `HuffmanTree` to represent the overall tree of character frequencies drawn on the previous page. You will also create a class `HuffmanNode` where each node stores information about one character. The contents of the `HuffmanNode` class are up to you, but it should not perform a large share of the overall algorithm.

Your `HuffmanTree` class must have the following public constructor and member functions:

`HuffmanTree(map<char, int> counts)`

In this constructor you are passed a map from characters (`char`) to the number of occurrences of that character (`int`). You should use this map to build your Huffman tree using a priority queue (`priority_queue`) as previously described.

`HuffmanTree(istream* in)`

In this constructor you are passed an input stream containing the contents of your encodings map (see below function). It will contain pairs of lines the first will contain the ASCII code for a character, the second its encoding as a string of `1`s and `0`s. You must reconstruct a Huffman tree from this so that you can use it to decode a compressed file. To do this, whenever you see a `0`, go left, whenever you see a `1`, go right (add a node if there isn't one there already). Store the character in the last node you create from its code.

`map<char, string> createEncodings()`

In this function you should traverse your Huffman tree and produce a mapping from each character in the tree to its encoded binary representation as a `string`. For the example shown on the previous pages, the map is the following:

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

`void compress(ifstream* input, OBitStream* output)`

In this function you should read the text data from the given input file stream and use your Huffman encodings to write a Huffman-compressed version of this data to the given bit output file stream. You will use an `OBitStream` object to help you write the binary output one bit at a time, as described below.

`void decompress(IBitStream* input, OBitStream* output)`

In this function you should read the compressed binary data from the given bit input file stream and use your Huffman tree to write a decompressed text version of this data to the given output file stream. Stop reading data when you reach the `EOF` character. You may assume that all characters in the input file were represented in the stream passed to your tree's constructor. You will use an `IBitStream` object to help you read the binary input one bit at a time, as described below.

You may have additional functions, so long as they are `private`. Note that the functions might be called in any order.

If a parameter passed to any member function above is `nullptr`, you should throw a `string` exception. If your object is asked to compress/decompress an empty file, the result should also be an empty file. Functions that traverse your tree should be implemented recursively whenever practical.

`OBitStream` and `IBitStream`:

To compress/decompress files, you will want to read and write binary data one bit at a time. C++'s built-in input/output streams have operations to read an entire line, token or character, but don't have one to read or write a single bit. Therefore, we are providing you with `OBitStream` and `IBitStream` classes with `writeBits` and `readBit` member functions to make it easier.

`OBitStream` Member Function	Description
`void` **`write`**`(char letter)`	writes a character to the output
`void` **`writeBits`**`(string bits)`	treats each character of the given string as a bit (`'0'` or `'1'`) and writes each of those bits to the output
`void` **`close`**`()`	stops writing (important to call this to ensure data is saved)

`IBitStream` Member Function	Description
`int` **`readBit`**`()`	reads a single 0 or 1 bit from input; returns -1 at end of file
`void` **`close`**`()`	stops reading

Development Strategy and Hints:

We suggest that you first focus on building your Huffman tree properly from the given map of character counts. Then work on creating the map of `char` → `string` encodings from your tree. Then work on using your encodings to compress files, next work on reconstructing your tree (the constructor that takes a stream) and lastly work on trying to decompress a file that you have previously compressed.

For your nodes to be able to be stored in a priority queue, the queue needs to know how to sort them. Therefore, when you construct your `priority_queue` you must specify the type to store in it, the underlying structure to use (`vector`) and a struct with an `operator()` implemented that takes two `HuffmanNode*`s. This operator should return `true` only when the character stored in the first parameter occurs more often than the character stored in the second parameter.

It can be difficult to tell whether you have compressed/decompressed a file correctly. If you open a Huffman-compressed binary file in a text editor, the appearance will be gibberish (because the text editor will try to interpret the bytes as ASCII encodings, which is not the way the data is stored). While developing your program, it can be helpful to write out each 0 or 1 as an entire character (byte) rather than as a bit. This defeats the purpose of compression, because the "compressed" file is actually larger than the original, but it can help you see whether the 0s and 1s are what you expect.

Our `OBitStream` and `IBitStream` can do this in debug mode. To write out your 0s and 1s as entire bytes instead of as bits, you can simply set `DEBUG` in `huffmanConstants.h` to `true`.

The provided `huffmanMain.cpp` client program can compress any text file. We suggest you start with a very small input file such as the example shown in this document, and work your way up to larger files once that works.

Creative Aspect (`secretmessage.huf` and `secretmessage.counts`):

Along with your program you should turn in files named `secretmessage.huf` and `secretmessage.counts` that represent a "secret" compressed message, and its counts file. The message can be anything you want, as long as it is not offensive. We will decompress your message with your tree and read it while grading.

Style Guidelines and Grading:

Part of your grade will come from appropriately utilizing **binary trees** and **recursion** to implement your Huffman tree as described previously. We will also grade on the elegance of your recursive algorithms; don't create special cases in your recursive code if they are not necessary or repeat cases already handled. Redundancy is another major grading focus; some member functions are similar in behavior or based off of each other's behavior. You should avoid repeated logic as much as possible. Your class may have other member functions besides those specified, but any other member functions you add should be `private`.

You should follow good general style guidelines such as: making member variables `private` and avoiding unnecessary member variables; appropriately using control structures like loops and `if`/`else`; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

Comment your code descriptively in your own words at the top of your class, each member function, and on complex sections of your code. Comments should explain each member function's behavior, parameters, return, pre/post-conditions, and exceptions. Remember that client facing comments (what you would want to know if you wanted to call a function but only had access to the header file) should go in the header file and comments about implementation should go in the cpp file.