Assignment 6: Huffman Encoding

Thanks to Owen Astrachan (Duke) and Julie Zelenski.
Updates by Keith Schwarz, Stuart Reges, Marty Stepp, Chris Piech, , Chris Gregg, Marissa Gemma, and Brahm Capoor.

Due: Wednesday, August 9 at 12:00 PM (noon). You may work in pairs for this assignment.

Assignment Overview and Starter Files

For this assignment, you will build a file compression algorithm that uses binary trees and priority queues. Your program will allow the user to compress and decompress files using the standard Huffman algorithm for encoding and decoding. Along the way, you'll also implement your own hash map, which you'll then put to use in implementing the Huffman encoding.

Huffman encoding is an example of a lossless compression algorithm that works particularly well on text but can, in fact, be applied to any type of file. Using Huffman encoding to compress a file can reduce the storage it requires by a third, half, or even more, in some situations. You'll be impressed with the compression algorithm, and you'll be equally impressed that you're outfitted to implement the core of a tool that imitates one you're already very familiar with.

The starter code for this project is available as a ZIP archive. A demo is available as a JAR. Note that the JAR will look for files in the same directory).

Starter Code Demo Jar

You must turn in the following files:

1. `mymap.cpp`: code to implement your hash map
2. `mymap.h`: header file containing declarations for your map
3. `encoding.cpp`: code to perform Huffman encoding and decoding
4. `secretmessage.huf`: a message from you to your section leader, which is compressed by your algorithm.

We provide you with several other support files, but you should not modify them. For example, we provide you with a `huffmanmain.cpp` that contains the program's overall text menu system; you must implement the functions it calls to perform various file compression/ decompression operations. The section on **Implementation details** below explains where in the files to program your solution.

Related reading:
Huffman on Wikipedia
ASCII Table

Huffman Encoding

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing textual data to make a file occupy a smaller number of bytes. Though it is a relatively simple compression algorithm, Huffman is powerful enough that variations of it are still used today in computer networks, fax machines, modems, HDTV, and other areas.

Normally textual data is stored in a standard format of 8 bits per character, using an encoding called *ASCII* that maps each character to a binary integer value from 0-255. The idea of Huffman encoding is to abandon the rigid 8-bits-per-character requirement, and instead to use binary encodings of different lengths for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the very common letter **'e'**, it could be given a shorter encoding (i.e., fewer bits), making the overall 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, on the balance.

The table below compares ASCII values of various characters to possible Huffman encodings for some English text. Frequent characters such as space and 'e' have short encodings, while rarer characters (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 you'll take to do perform a Huffman encoding of a given text source file into a destination compressed file are:

1. **count frequencies**: Examine a source file's contents and count the number of occurrences of each character, and store them in a map using the `MyMap` class you'll write.
2. **build encoding tree**: Build a binary tree with a particular structure, where each node represents a character and its count of occurrences in the file. A **priority** queue is used to help build the tree along the way.
3. **build encoding map**: Traverse the binary tree to discover the binary encodings of each character.
4. **encode data**: Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file.

Your program's output format should exactly match example logs of execution here:

Example Run #1

Example Run #2

Example Run #3

Example Run #4

Example Run #5

Example Run #6

Example Run #7

Example Run #8

Encoding a File Step 1: Counting Frequencies

As an example, 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, including spaces and 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. This frequency table is represented as a map:

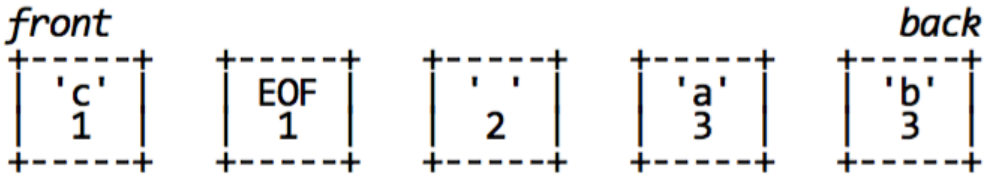
```
{' ':2, 'a':3, 'b':3, 'c':1, EOF:1}
```

Note that **all** characters must be included in the frequency table, including spaces, any punctuation, and the EOF marker.

Below you will find details on implementing the class MyMap, which you must use to store this encoding map as a hash map.

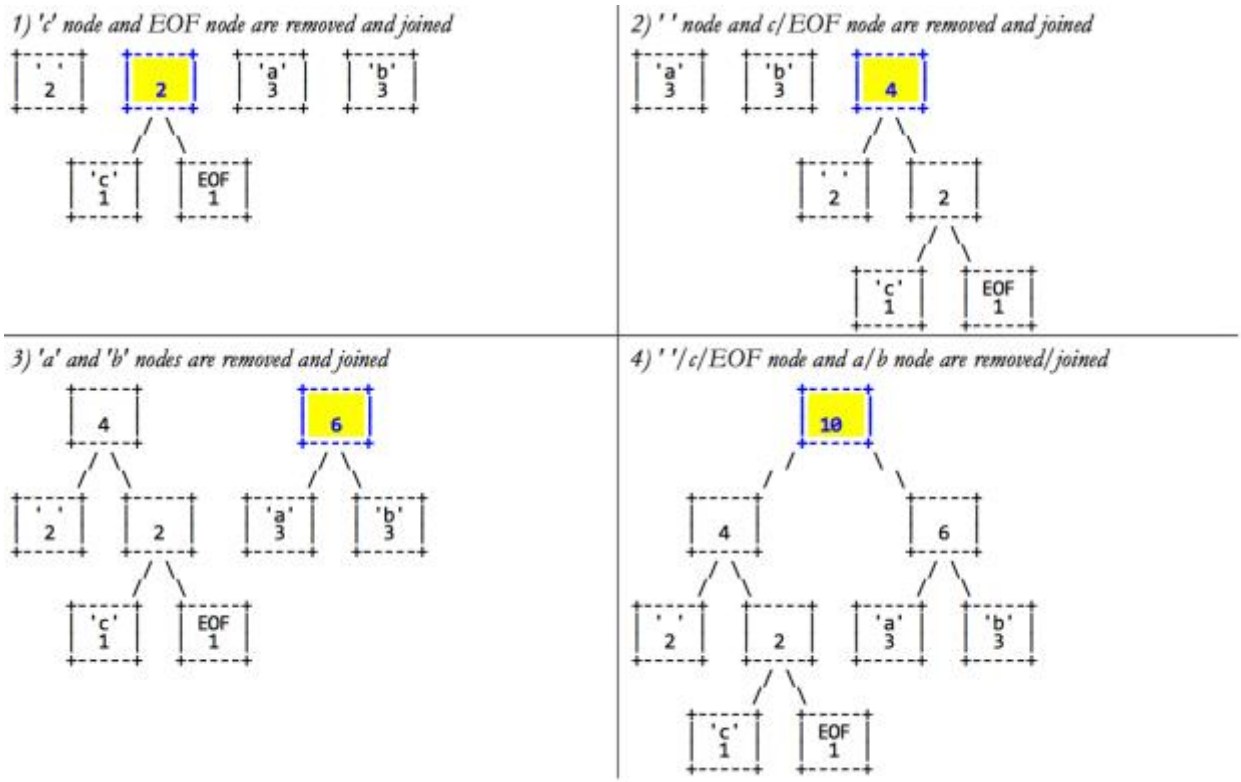
Encoding a File Step 2: Building an Encoding Tree

Step 2 of Huffman's algorithm builds an encoding tree as follows. First, we place our counts into node structs (out of which we will build the binary tree); each node stores a character and a count of its occurrences. Then, we put the nodes into a priority queue, which stores them in prioritized order, where smaller counts have a higher priority. This means that characters with lower counts will come out of the queue sooner, as the figure below shows. (As you will see when you implement it, the priority queue is somewhat arbitrary in how it breaks ties, which is why in this example 'c' can end up before EOF, while 'a' is before 'b'.)



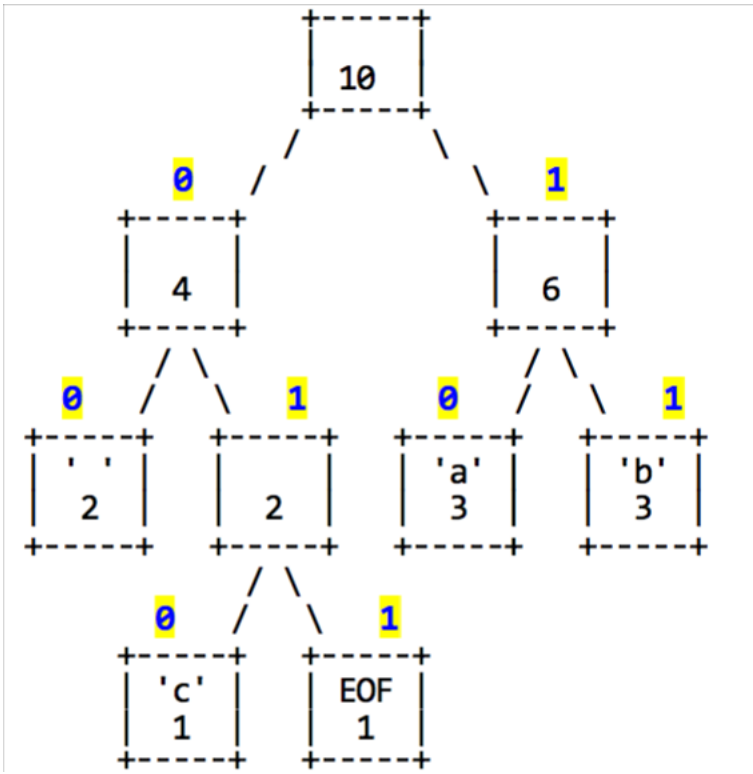
Now, to construct the tree, the algorithm repeatedly removes two nodes from the front of the queue (i.e., the two nodes with the smallest frequencies) and joins them into a new node whose frequency is their sum. The two nodes are positioned as children of the new node; the first node removed becomes the left child, and the second becomes the right. The new node is re-inserted into the queue in sorted order (and we can observe that its priority will now be less urgent, since its frequency is the sum of its children's frequencies). 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 illustrates 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. As we shall see, this structure can be used to create an efficient encoding in the next step.



Encoding a File Step 3: Building an Encoding Map

The Huffman code 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 below:



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. Similarly, 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 have arbitrary lengths, we will store them as strings. For this tree, the encoding map would look like this:

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

Encoding a File Step 4: Encoding the Textual Data

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

```
10110010110001010011011
```

The following table details the char-to-binary mapping in more detail. The overall encoded contents of the file require 22 bits, or a little under 3 bytes, compared to the original file size 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 implementing this; it is part of the underlying file system.

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

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. (If it's not clear to you how this works, trace through the example tree above, or one produced by your own algorithm, to see for yourself.)

Decoding a File

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. 11100100010010100 11
- 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.**1** 1100100010010100 11
- We read a 0 (left), then a 0 (left). We reach ' ' and output a space. 11100100010010100 11
- We read a 1 (right), then a 0 (left). We reach 'a' and output a. 11100100010010100 11
- We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'c' and output c. 11100100010010100 11
- We read a 1 (right), then a 0 (left). We reach 'a' and output a. 11100100010010100 11
- 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,` which 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)
... };
```

The character field is declared as type `int`, but you should think of it as a `char`. (`char` and `int` types are largely interchangeable in C++, but using `int` here allows us to sometimes use the character type 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 ■, marks the end of the encoding and you will need to place it at the end of an encoded stream.
- 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 branch nodes of the Huffman encoding tree that have children, because such nodes do not represent any one individual character.)

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 input/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 default in/output streams. 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. (You can read the documentation for these bit stream classes here: https://stanford.edu/~stepp/cppdoc/bitstream.html.)

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: `MyMap`

Relevant files: **`mymap.cpp, mymap.h`**

In the first part of this assignment, you will be implementing a data structure `MyMap` that behaves identically to a `HashMap` with keys and values of type `int`. In later parts of this assignment, you will be using this data structure to represent a frequency table of the characters in files you wish to compress.

The class exports the following **required** public methods, which you will need to implement:

Name	**Description**	**Runtime**
`MyMap()`	Constructor for your map	O(1)
`~MyMap()`	Destructor for your map. Clears all heap-allocated memory.	O(1)
`void put(int key, int value)`	Associates `key` with `value` in your map, replacing existing values if need be.	O(1)
`int get(int key)`	Returns the value associated with `key` in your map. Throws a string exception if the key is not in the map.	O(1)
`bool containsKey(int key)`	Returns a boolean indicating whether or not `key`is a key in your map.	O(1)
`Vector<int> keys()`	Returns a `Vector` of all the keys in your map.	O(N)

Note that these methods represent the methods required for one possible implementation of Huffman Compression: you are free to implement other public methods for use in your assignment, but the methods detailed above must be implemented. As a general guide, refer to the Stanford `HashMap` documentation for other methods you might want to implement.

In addition, the class exports the `sanityCheck()` method, which has been written for you:

`void sanityCheck()`	Tests your map to ensure that the previously mentioned public methods behave as expected.

Note that the `sanityCheck()` method is not comprehensive - depending on the specifics of your implementation, you are allowed and encouraged to modify the method to better suit you. `sanityCheck()` will not be graded, but is a useful way to check whether your map behaves as intended.

Development strategy and hints

- Your `MyMap` must be implemented using an array of buckets into which key-value pairs are hashed. Each bucket in the array must be the head of a linked list of key-value pairs. You are provided with the following `struct` to help you with this:

```
struct key_val_pair {
      int key;
      int value;
      key_val_pair* next;
}
```

- In order to create an array of buckets, we provide you with the following private method:

`bucketArray createBucketArray(int nBuckets);`	Returns an array of `key_val_pair*` with size `nBuckets.` All elements are set to `nullptr` initially.

Note that the `bucketArray` type is simply a `typedef` (synonym) for the type `key_val_pair**`, that is, pointers to pointers to `key_val_pair`s. While this certainly looks grisly, we simply use `key_val_pair**` to refer to an array of `key_val_pair*`s, just as we use `int*` to refer to an array of type `int`. For the purposes of this assignment, you need not worry about any of this - you can simply work with the `bucketArray` type as below:

```
bucketArray buckets = createBucketArray(8);
key_val_pair* head = buckets[4]; //access element in array
Buckets[3] = new key_val_pair; //set an element in the array
```

- We provide a private method `int hashFunction()`, which you may use when hashing key-value pairs into your map.
- For full credit, ensure that the public methods you implement have the specified runtime.

Implementation Details: `encoding.cpp`

For the second part of this assignment, you will write the functions described below in the file **`encoding.cpp`.** This will allow you to encode and decode data using the Huffman algorithm. Our provided main client program will allow you to test each function one at a time before moving on to the next. The following functions are required, but note that 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.

Name	Description
`MyMap 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(MyMap &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`; that all counts are positive integers; and that it contains at least one key/value pairing. 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 allows each element to be enqueued along with an associated priority.

	The `dequeue` function always returns the element with the minimum priority number.
`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 recursively 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.

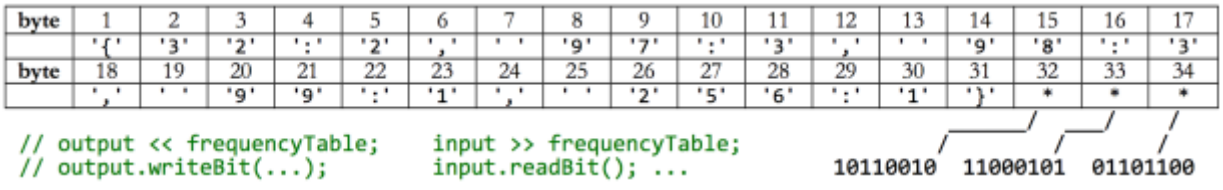
To manually verify that your implementations of `encodeData` and `decodeData` are working correctly, use our provided test code to compress strings of your choice into a sequence of 0s and 1s. The next few paragraphs describe a header that you will add to compressed files, but in `encodeData` and `decodeData`, 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 above 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 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 Maps can easily be read and written to/from streams using `<<` and `>>` operators. We have provided overriden versions of these operators for the `MyMap` class, 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 when you decompress it later. 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:



Looking at this new rendition of the compressed file, you may be thinking, "The file isn't 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:

Name	Description
`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 readable/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 Input File (secretmessage.huf)

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 (appropriate) you choose. Your SL will decompress your message with your program and read it while grading.

Development Strategy and Hints

- 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.
- 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.

Possible Extra Features

Though your solution to this assignment must match all of the specifications mentioned previously, you are allowed and encouraged to add extra features to your program if you'd like to go beyond the basic assignment. Here are some ideas for extra features that you could add to your program.

1. **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!
2. **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.
3. **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.
4. **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.
5. **Implement rehashing for your `MyMap` class.** By default, you are not required to implement rehashing in your map. However, you might choose to do this if the load factor in your map exceeds a particular threshold.
6. **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 extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can easily identify them).

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 feature(s) 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 two versions of your program file: a first one named life.cpp without any extra features added (or with all necessary features disabled or commented out), and a second one named life-extra.cpp with the extra features enabled. Please distinguish them in by explaining which is which in the comment header.

Our submission system saves every submission you make, so if you make more than one we will be able to view all of them; your previously submitted files will not be lost or overwritten.