Assignment 6b: Huffman Encoding

Assignment by Marty Stepp and Victoria Kirst. Based on past work and ideas from Owen Astrachan (Duke); Stuart Reges (Washington); Julie Zelenski, Keith Schwarz.

MARCH 3, 2017

Description Implementation Style FAQ Extras

This problem focuses on Binary Trees. This is an **individual assignment**. Write your own solution and do not work in a pair/group on this program.

Files and Links:


Project Starter ZIP
(open **Huffman.pro**)


Turn in:
encoding.cpp
secretmessage.huf


Demo JAR


Homework Survey


output logs:
log #1
log #2
log #3
log #4
log #5
log #6

Problem Description:

Huffman encoding (Wikipedia) (Wolfram Mathworld) 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 using an encoding called *ASCII* that maps every character to a binary integer value from 0-255. (ASCII encoding table) The idea of Huffman encoding 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 common 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 some English text. 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 four steps involved in Huffman encoding a given text source file into a destination compressed file are:

1. **count character frequencies (`buildFrequencyTable`):** Examine a source file's contents and count the number of occurrences of each character.
2. **build a Huffman encoding tree (`buildEncodingTree`):** 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 a character encoding map (`buildEncodingMap`):** Traverse the binary tree to discover the binary encodings of each character.
4. **encode the file's data (`encodeData`):** Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file.

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 perform the steps listed above, each in a particular required function; you can

add more functions as helpers if you like, particularly to help you implement any recursive algorithms.

The following is one sample partial log of execution of the provided main program using your code. More logs are available above or through the Compare Output feature in your console window.

```
Welcome to CS 106X Shrink-It!
...
1) build character frequency table
2) build encoding tree
3) build encoding map
4) encode data
5) decode data
C) compress file
D) decompress file
F) free tree memory
B) binary file viewer
T) text file viewer
S) side-by-side file comparison
Q) quit

Your choice? c
Input file name: large.txt
Output file name (Enter for large.huf): large.huf
Reading 9768 uncompressed bytes.
Compressing ...
Wrote 5921 compressed bytes.
```

Example log of execution

Here is a supplementary handout on Huffman encoding and file compression if you are interested in more information after reading this spec.

Encoding a File, Step 1: Counting Character Frequencies (`buildFrequencyTable`):

For 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. The 10th is a special "end-of-file" (EOF) byte.

byte	1	2	3	4	5	6	7	8	9
char	**'a'**	**'b'**	**' '**	**'a'**	**'b'**	**' '**	**'c'**	**'a'**	**'b'**
ASCII	**97**	**98**	**32**	**97**	**98**	**32**	**99**	**97**	**98**
binary	**01100001**	**01100010**	**00100000**	**01100001**	**01100010**	**00100000**	**01100011**	**01100001**	**0110**

In Step 1 of Huffman's algorithm, a count of each character is computed. The counts are represented as a map. In this case, the map would contain the following character/count pairs:

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

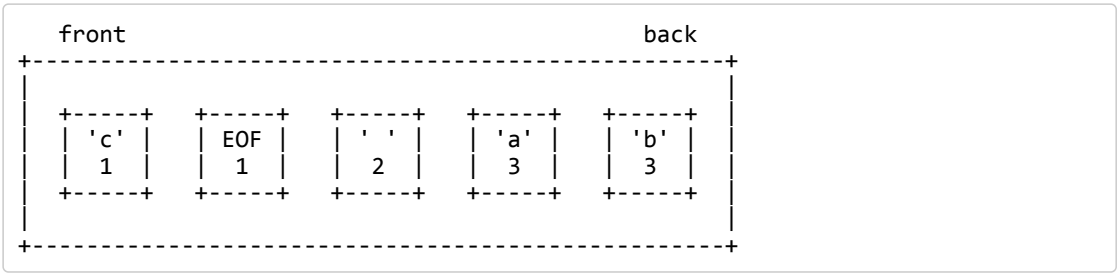
The step of counting character frequencies is represented by the following function that you must write:

```
Map<int, int> buildFrequencyTable(istream& input)
```

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`**.

Encoding a File, Step 2: Building an Encoding Binary Tree (`buildEncodingTree`):

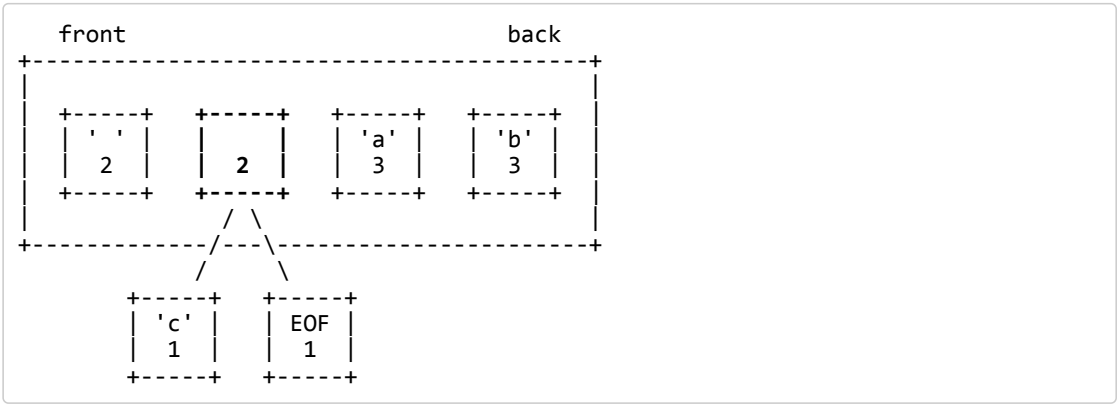
Step 2 of Huffman's algorithm places our counts into binary tree nodes, with each node storing a character and a count of its occurrences. Pointers to the nodes are then put into a priority queue, which keeps them in order with smaller counts having higher priority, so that characters with lower counts 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'`**).



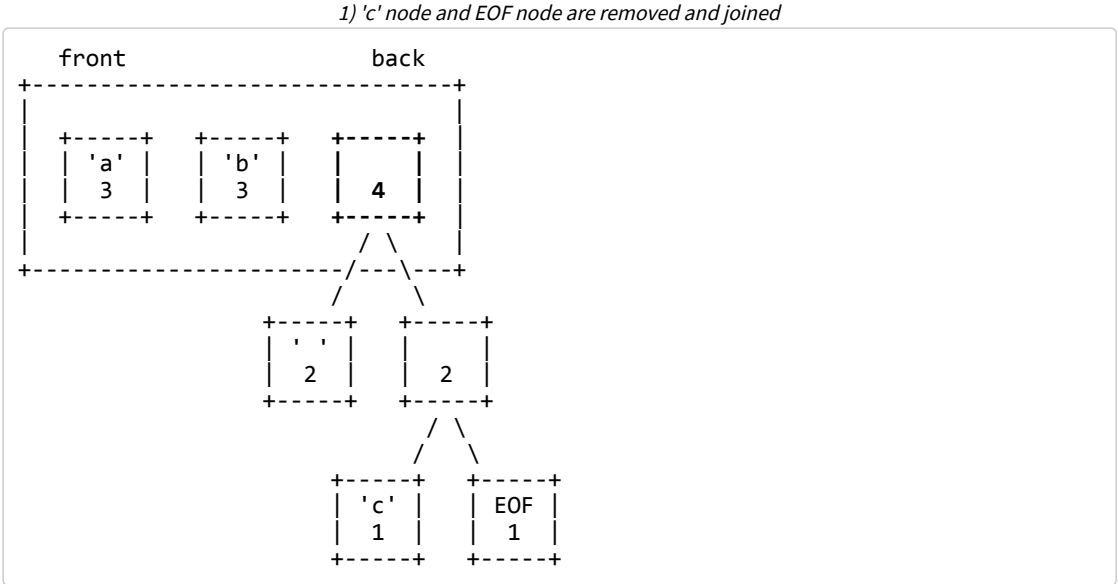
priority queue of character frequencies (size 5)

Now the algorithm repeatedly removes the two node pointers 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.

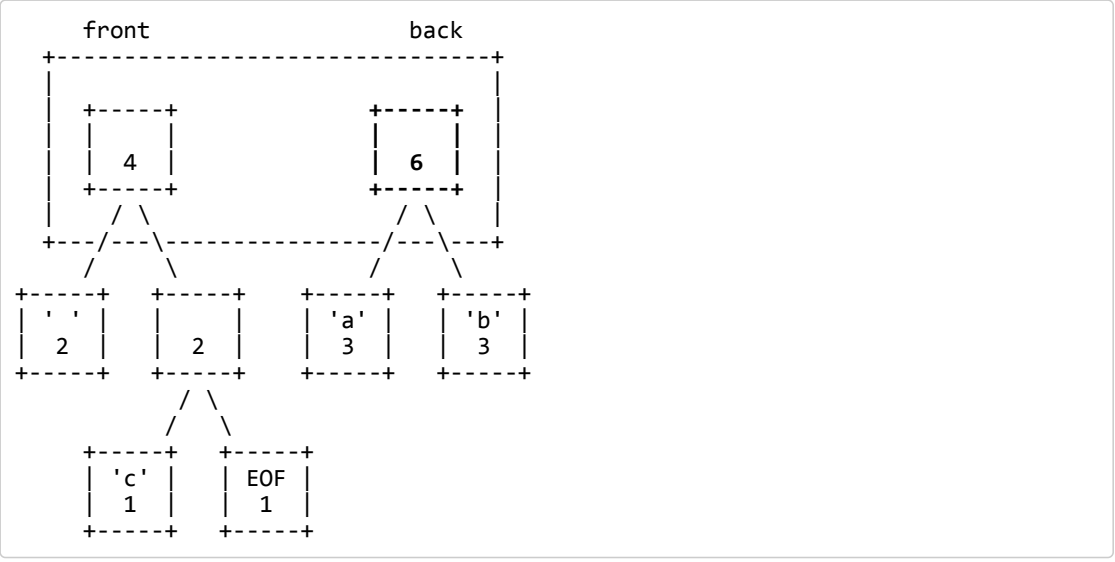
The following diagram shows this process. Notice that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. This structure can be used to create an efficient encoding in the next step.



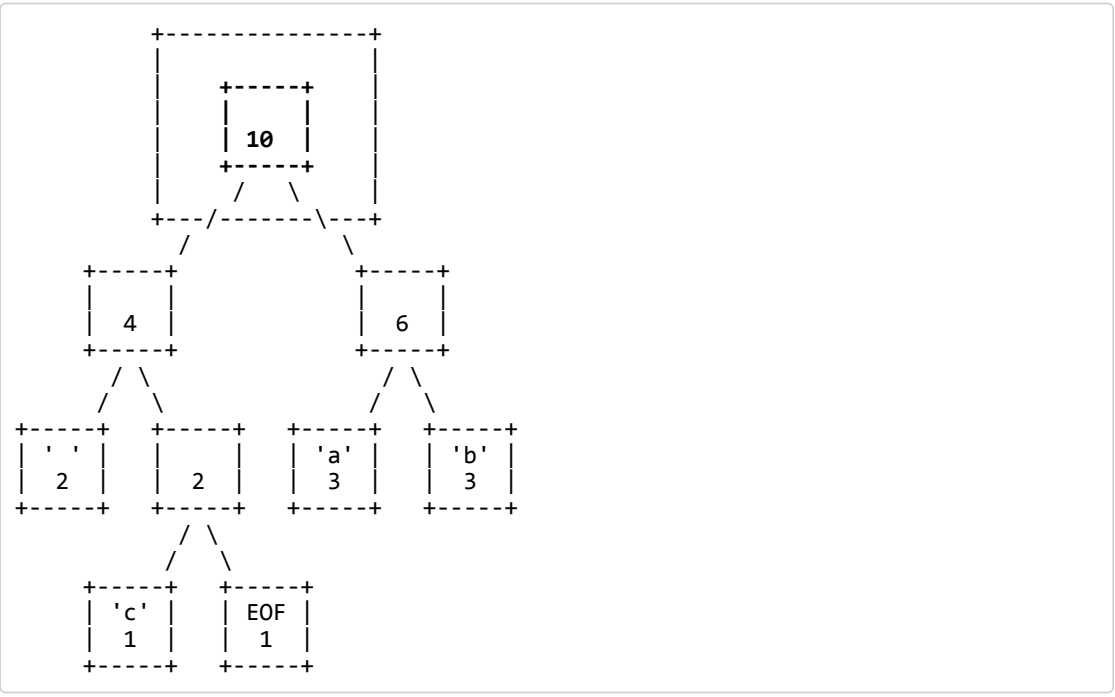
1) 'c' node and EOF node are removed and joined



2) ' ' node and c/EOF node are removed and joined



3) 'a' and 'b' nodes are removed and joined



4) ' '/c/EOF node and a/b node are removed/joined

The step of building the Huffman tree from the character counts is represented by the following function that you must write:

```
HuffmanNode* buildEncodingTree(const Map<int, int>& freqTable)
```

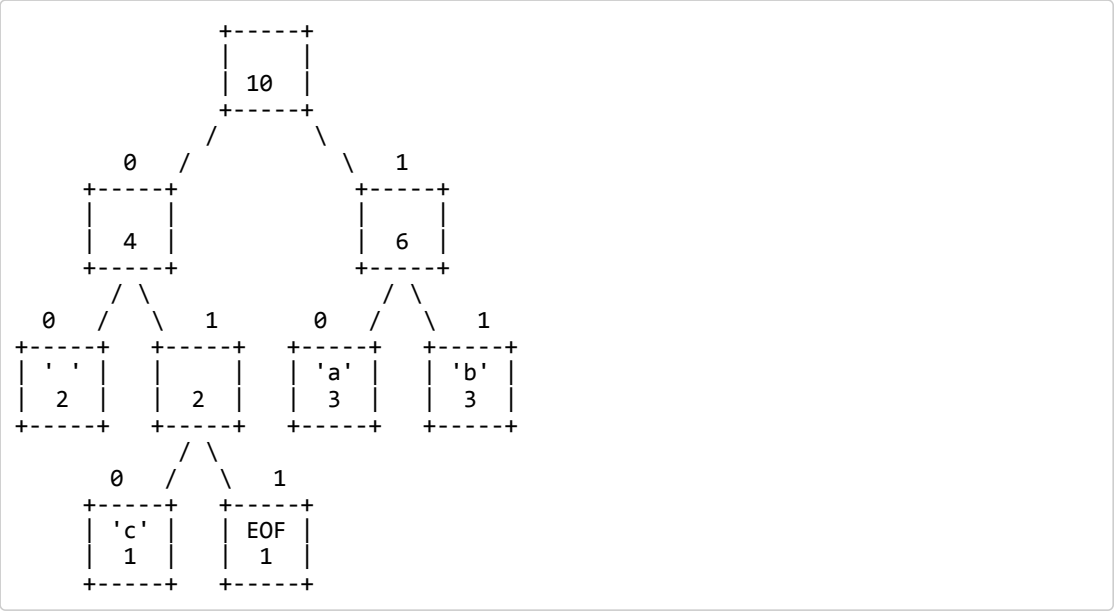
In this function you will accept a frequency table (like the one you built in the last step, **`buildFrequencyTable`**) and use it to create a Huffman encoding tree based on those frequencies. You must return a pointer to the node representing the root of the tree.

When building the encoding tree, use the **`PriorityQueue`** collection provided by the Stanford libraries, defined in library header **`priorityqueue.h`**. This priority queue allows each element to be enqueued along with an associated numeric 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 PriorityQueue documentation on the course website and lecture slides for more information about priority queues.

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; etc.

Encoding a File, Step 3: Building an Encoding Map (`buildEncodingMap`):

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`**. 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"}
```

The step of building the encoding map from the binary tree is represented by the following function that you must write:

```
Map<int, string> buildEncodingMap(HuffmanNode* encodingTree)
```

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", 'b':"11"}`**. If the encoding tree is null, return an empty map.

Encoding a File, Step 4: Encoding the Text 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:

```
1011001011000101011011
```

The following table details 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	**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.

The step of encoding the file's data from the encoding map is represented by the following function that you must write:

```
void encodeData(istream& input, const Map<int, string>& encodingMap, obitstrea
```

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.

Decoding a File (`decodeData`):

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 these bits to traverse the Huffman tree. Starting from the root, 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, and then your algorithm should return to the top of the tree. For example, suppose we are given the same encoding tree above, and we are asked to decode a file containing the following bits:

```
11100100010010100110000
```

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 the leaf `'b'` and output it. Back to the root.
  11100100010010100110000
- We read a 1 (right), then a 0 (left). We reach leaf `'a'` and output it. Back to root.
  1110010001001010011000 00
- We read a 0 (left), then a 1 (right), then a 0 (left). We reach `'c'` and output it.
  111001000100101001100000
- We read a 0 (left), then a 0 (left). We reach `' '` and output a space.
  11100100010010100110000 0
- We read a 1 (right), then a 0 (left). We reach `'a'` and output it.
  1110010001001010011000 00
- We read a 0 (left), then a 1 (right), then a 0 (left). We reach `'c'` and output it.
  11100100010010100110000 0
- We read a 1 (right), then a 0 (left). We reach `'a'` and output it.
  11100100010010100110000 0
- We read a 0, 1, 1. This is our EOF encoding pattern, so we stop.
  11100100010010100110000 0
- The overall decoded text is `"bac aca"`. (Notice that we do not read or decode the final 00000 bits in the last byte because they come after the EOF marker.)

The step of decoding the file's data from the compressed binary bits is represented by the following function that you must write:

```
void decodeData(ibitstream& input, HuffmanNode* encodingTree, ostream& output)
```

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 for file names, nor 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 rest of this document 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 the **`compress`** / **`decompress`** functions described later.

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)
    ...
};
```

The character field is declared as type **int**, but you should think of it as a **char**. (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 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 in/output stream functions:

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

istream (input stream) member	**Description**
int get()	reads a single byte (8-bit character) from the input stream; returns -1 if the stream has reached the end of the file

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; returns -1 if the stream has reached the end of the 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 those, 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.

Compress and Decompress:

The preceding functions implement Huffman's algorithm, but the decoding function requires the encoding tree to be passed as a parameter. Without the encoding tree, you don't know the mappings from bit patterns to characters.

We will work around this by writing the encoding map 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 (**buildFrequencyTable**), 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. 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, with the last three bytes listed at the end:

```
byte  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      {   3   2   :   2   ,       9   7   :   3   ,       9   8   :   3   ,

     20  21  22  23  24  25  26  27  28  29  30  31  32        33        34
      9   9   :   1   ,       2   5   6   :   1   }  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 absolutely 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)
```

In this function you should compress the given input file into the given output file, combining the steps 1-4 described previously. 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. If the file is empty, your code will fail to read the header, so you should stop there and return without decompressing the file. The streams are already opened and ready to be read/written; you do not need to prompt the user for filenames or open/close the files yourself. If your function allocates any dynamic memory on the heap, you must free it and not leak memory.

```
void decompress(ibitstream& input, ostream& output)
```

In this function you should do the opposite of **compress**; you 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 for filenames or open/close files. If your function allocates any dynamic memory on the heap, you must free it and not leak memory.

```
void freeTree(HuffmanNode* node)
```

In this function you 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, **secretmessage.huf**:

Along with your code, submit 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!) that you like. Your section leader will decompress your message with your program and read it while grading. This is worth a small part of your grade.

Development Strategy and Hints:

Do not use type char anywhere in your program. Declare all character variables as type **int**. This is needed for your program to function properly. If you use **char**, non-text files (e.g. images) won't work.

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`** instead. 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 files compressed by your program. You do not need 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_Xxxxxxx** folder. They won't show up in the normal **res/** resource folder of your project.

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1-5 specs, such as the ones about good problem decomposition, parameters, redundancy, using proper C++ idioms, and commenting. The following are additional points of emphasis and style contraints specific to this problem.

Binary tree usage: Part of your grade will come from appropriately utilizing binary trees and recursive algorithms to traverse them. Any functions that traverse a binary tree from top to bottom should implement that traversal **recursively**. If a particular function must traverse a tree multiple times, it is okay to write a loop that initiates each traversal, as long as the traversal itself is recursive. We will check this particular constraint strictly; no exceptions!

Modifying required function headers: Please do not make modifications to the required functions' names, parameter types, or return types. Our client code should be able to call the functions successfully without any modification.

Redundancy: Redundancy is another major grading focus; avoid repeated logic as much as possible. If two of your functions are similar, have one call the other, or utilize a common helper function.

Memory usage: 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. But if you create a Huffman tree yourself to help you implement another function, you must free that entire tree yourself.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

Huffman Encoding FAQ (click to show)

Possible Extra Features:

Here are some ideas for extra features that you could add to your program for a small amount of extra credit:

- **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 checksum. 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 extra features 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 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 with the standard file name without any extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features 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.
