Assignment 6: Huffman Encoding

Assignment by Owen Astrachan of Duke University,
with edits by Julie Zelenski, Keith Schwarz, Daniel Jackoway, and Marty Stepp

Huffman encoding is an example of a lossless compression algorithm that works particularly well on text and, in fact, can be applied to any type of file. It can reduce the storage required by a third or half or even more in some situations. Hopefully, you will be impressed with this awesome algorithm and your ability to implement such a nifty tool!

You are to write a program that allows the user to compress and decompress files using the standard Huffman algorithm for encoding and decoding. Carefully read the Huffman handout (Handout 21) for background information on compression and the specifics of the algorithm. This handout doesn't repeat that material, and instead just describes the structure of the assignment. Even so, this handout is on the long side. It preemptively identifies the tough spots, but we learned during part offerings that several critical details we're easily overlooked by students skimming the handout a bit too quickly. We encourage you to give the handout your full attention!

Due Friday, March 10 at the start of class.

Working in pairs on this assignment is <u>not permitted</u>.
You <u>must</u> complete this assignment individually.

Overview of the Program Structure

Ultimately, your task will be to write a function

```
void compress(istream& input, obitstream& output);
```

that takes as input a stream containing the data to compress and an `obitstream` (described later) containing an output source, then compresses the input file using Huffman coding and writes it to the destination. You'll also write a function

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

that takes as input an `ibitstream` (described later) containing an compressed file, then decompresses the file and writes the result to the given output stream.

Rather than just turning you loose on these functions, we've broken down the assignment into a smaller number of helper functions and tasks. You'll actually build the assignment bottom-up, writing a bunch of helper functions that each do individual odds and ends tasks that build into the final product. (Hey, that's kinda like how Huffman coding works, too!) Our provided starter code is essentially a test driver that will let you try out each individual smaller piece of the assignment along the way, culminating with these two functions.

We've provided you with some helper functions and types that you'll need to use along the way. We'll describe them to you as we go.

Step One: Build the Frequency Table

In order to build the encoding tree that you'll use for encoding and decoding, you'll need to know the frequencies of all the characters in the file that you'll be compressing. Your first task is to write a function

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

that takes as input an `istream` (remember those from Way Back When?) containing the file to compress, then hands back a `Map<int, int>` associating each character in the file with its frequency.

You might be wondering – if we're associating *characters* with frequencies, why are the keys in the `Map` of type `int` and not `char`? If you'll recall, our implementation of Huffman coding builds up an encoding tree for all of the characters of the file, plus a special "pseudo-EOF character" that we use to indicate when all the data has been read out of the file. That pseudo-EOF character has to be different than any possible character. To accomplish this, we've provided you a constant

```
const int PSEUDO_EOF = /* … doesn't really matter … */
```

that you should use to represent the pseudo-EOF character. "But wait," you might ask, "if the keys in the `Map` of are of type `int`, how are we supposed to store characters in them?" The good news is that in C++, you can automatically convert any variable of type `char` to type `int`. Doing so just gives you back an integer that holds the numeric value of the character in question. For example, writing something like

```
int charA = 'A';
```

will give you back an integer holding 65, the numeric code for the letter A. So treat the keys in this map not as integers in the conventional sense, but rather as "characters, plus the weird `PSEUDO_EOF` constant."

This step will require you to be able to read characters from a file one at a time. To do this, you should ***not*** use `getline` or the stream extraction operator >> (the >> operator skips whitespace, which we don't want.) Instead, use the function `istream::get`, which you can invoke like this:

```
char ch;
while (stream.get(ch)) {
    /* … do something with the character ch … */
}
```

This will read all the characters from the file, stopping as soon as there are no more characters left to be read. Each iteration through the loop will let you process the most-recently-read character from the file.

Step Two: Build the Huffman Encoding Tree

Now that you have the frequency map, you can write the core logic to build up the Huffman encoding tree. Your next task is to implement a function

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

that takes as input a frequency map and constructs a Huffman encoding tree using the algorithm we described in class (and elaborated on in Handout #21).

The `HuffmanNode` type here is defined in the `HuffmanNode.h` header file. It represents a tree node, so it has child pointers (conveniently named `zero` and `one`) and has a field to store a frequency count. The `HuffmanNode` type also has a field character that represents which character it represents. This is an `int` (following the convention from Step One) that should either be the numeric value of a character or the constant `PSEUDO_EOF` if the node is a leaf node. Otherwise, it should hold the special value `NOT_A_CHAR`, which is a sentinel that just indicates that a specific value isn't a character.

You will almost certainly want to use the `PriorityQueue` type from the `"priorityqueue.h"` header file in the course of implementing this function. Check the docs on the course website for more info.

Step Three: Clean Up Your Messes

It's important to clean up any messes you make in C++, so in addition to writing the code from Step Two to build the encoding tree, you'll need to write a function

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

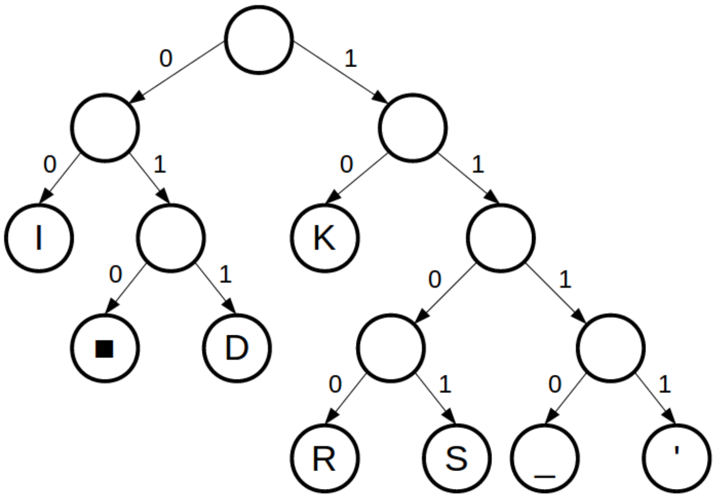
that deallocates all the memory in the specified Huffman tree. Hopefully, this step shouldn't be too tricky given what we've covered in lecture and section.

Step Four: Build the Encoding Map

You've now got yourself a nifty little tree structure that you can use to determine what bit patterns to encode each character with. Nice! But in order for this to be at all useful, you'll need to be able to use that tree to determine what the encodings should be for each character. Your next task is to write a function

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

that takes in as input a Huffman encoding tree and returns a `Map` that associates characters (again, represented as integers) with the string of bits that you'll use to encode that character. For example, suppose you're given this Huffman tree, where the ■ character represents the pseudo-EOF:



Your code should produce a map with these key/value pairings:

```
       'I':  "00",
PSEUDO_EOF:  "010",
       'D':  "011",
       'K':  "10",
       'R':  "1100",
       'S':  "1101",
       ' ':  "1110",
       '\'': "1111"
```

Step Five: Write the Encoder

You now have the text to encode and the encoding map showing which characters get mapped to what. Great! Your next task is to write a function

```
void encodeData(istream& input, const Map<int, string>& encodingMap,
                obitstream& output);
```

that takes as input a stream containing the text to encode and an encoding map associating each character with the bit sequence to use to encode it, then writes the bitwise encoding of the data from the input stream to the specified output `obitstream`.

That last parameter is an `obitstream`, a custom stream type we've written that you can use to write data one bit at a time to a file. To write an individual bit, you can call the `obitstream::writeBit` function like this:

```
output.writeBit(0);                output.writeBit(1);
```

The `obitstream::writeBit` function requires you to pass in an integer that is either 0 or 1 as an argument. Be careful – the integer values 0 and 1 are *not* the same as the *character* values `'0'` and `'1'`. The characters `'0'` and `'1'` have the perplexing values of 48 and 49, so if you see any weird errors about those numeric values, then that might be the culprit.

Similarly, make sure that you do *not* call the function `obitstream::put`. The following code is *incorrect* and *doesn't do what you think it does!*

```
⚠ output.put(0); ⚠                ⚠ output.put(1); ⚠
```

If you write this code, you'll be writing the *character with ASCII value 0* to the file or the *character with ASCII value 1* to the file, which is quite different than writing things one bit at a time. Again, if you get some weirdly confusing errors where the file seems to have the wrong contents, double-check this to make sure that you're calling the proper functions.

As a reminder, *make sure you remember to write the pseudo-EOF character* after you finish writing out the text, since otherwise your decoding logic won't know when to stop reading!

Step Six: Write the Decoder

In Step Five, you've written some code to encoding the contents of a file according to a given encoding map. Your next task is to write a function

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

that takes as input an `ibitstream` containing a stream of bits and a pointer to the root of an encoding tree, then decodes the bits from that stream and writes the result to the specified output stream.

The `ibitstream` type is a custom type that's analogous to the `obitstream` type from Step Five, but for reading from files rather than writing to them. You can read a single bit from the file by using the handy `ibitstream::readBit()` function, as shown here:

```
int bit = input.readBit();
```

Make sure that you *do not* call the `ibitstream::get()` function to read individual bits. The following code is *incorrect* and *will not do what you think it does!*

```
⚠ char bit; input.get(bit); ⚠          ⚠ int bit = input.get(); ⚠
```

This will try reading an entire character from the file, not an individual bit, which isn't what you want to do here.

To write a character to the output stream, use the `ostream::put` function, like this:

```
output.put(character);
```

You should ***not*** use the stream insertion operator (the >> operator) to write to the file. While you can make it work, it's very easy to make hard-to-debug mistakes when doing so, and you're best off without it.

As a hint for this problem, you shouldn't need to know what happens if you call `readBit()` when there are no more bits left in the file. If you set up your encoding and decoding routines properly, you'll be using the `PSEUDO_EOF` to tell you when to stop reading and writing, and you'll never actually hit the end of the file.

Step Seven: Write the End-to-End Compressor and Decompressor

At this point, you've implemented all the pieces that you're going to need to build a complete Huffman encoder and decoder – you just need to assemble them in the right order and take care of some final details.

Your task in this section is to write the pair of compression and decompression functions we talked about at the start of the handout, which are reprinted here:

```
void compress(istream& input, obitstream& output);
void decompress(ibitstream& input, ostream& output);
```

You have all the pieces you need to implement these functions, but there's one last step you'll need to take care of. The decompression code you'll write will need to have some way of knowing what Huffman encoding tree your compressor used, since without it it'll have no way to figure out how to decode the message. Therefore, your compression routine will need to write the frequency table to the output before it starts writing any bits. Fortunately, you can use the stream insertion operator << to do this:

```
Map<int, int> frequencyMap = /* … */;
output << frequencyMap; // Note: don't use endl here!
```

That way, your decompression routine can use the stream insertion operator >> to read the table:

```
Map<int, int> frequencyMap;
input >> frequencyMap;
```

There's one other detail to watch out for. In your compression routine, you'll call `buildFrequencyTable` at some point to construct a frequency table for the file you'll be compressing. That call to `buildFrequencyTable` will read through all the characters of the file, so if you then try to read from the stream again, you'll find that all the data has already been read and that there aren't any characters left! Therefore, before you make a second pass over the file to compress each character, you'll need to *rewind* the stream back to the beginning. You can do this by calling the handy dandy `rewindStream` function:

```
rewindStream(input);
```

This resets the stream back to the start of the file so that you can read all the characters again.

We hope that writing these two functions doesn't take you too much time. You've already built all the pieces (and tested them thoroughly, we hope!) before you've started on this section, so most of the logic here will be coordinating all the pieces so that they run in the right order and do the right thing.

Your decompression routine does *not* need to do any error-handling. You can assume that anything you're decompressing is something that you yourself compressed. That said, when you're first writing this function, it would definitely not be a bad idea to put in a ton of error-handling logic just in case there's something a bit buggy in your compression or decompression routines.

Step Eight: Leave a Message!

You've now put together a compression and decompression routine, so why not leave a nice message for your section leader? Put whatever contents you'd like into the `secretmessage.txt` file in the `res/` directory, compress the file, and save the result to `secretmessage.huf`. That file will be in the build directory that Qt Creator automatically makes for you when you configure a project, not the `res/` directory. Your section leader will then decode the message when grading. You can include anything you'd like in that message, provided of course that you're civil. ☺

Advice, Tips, and Tricks

Here's some general advice and recommendations for working through this assignment:

- ***Include error checking in your functions to simplify debugging***. For example, if a client tries to look up a bit pattern for a character that is out of range or uses an encoding that hasn't been set up yet, it would be more helpful to report that with `error` than to reference outside the array or quietly return `""`.
- ***Make sure you understand each module and the entire program***. Before you try to implement the modules, it would be worthwhile to study this handout thoroughly and make sure you understand what each piece does.
- ***Get each part of the program working before starting on the next one***. You should certainly focus on the individual modules rather than the entire program. Do not try to write all the implementations ahead of time and then see if you can get the program working as a whole. Concentrate on one module in isolation and write, test, and debug it thoroughly before moving on to the next. The provided test code should (hopefully!) make this relatively easy.
- ***Don't leak memory***. Make sure not to leak any tree nodes, and if you allocate any extra memory, be sure to deallocate it!
- ***Test your code thoroughly!*** Your implementation should be robust enough to compress any given 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 the additional overhead of the encoding table) but it should be possible to compress multiple iterations, decompress the same number of iterations, and return to the original file.

  Handling a file containing no characters would require a special case. Do you see why? We will not expect you to do this and will not test against this case.
- ***Understand that bitwise I/O is slow***. The operations that read and write bits are somewhat inefficient and working on a large file (100K and more) will take some time. It is definitely worthwhile to do some tests on large files as part of stress-testing your program, but don't be concerned if the reading/writing phase is takes a bit more time that you might expect.
- ***Only worry about decompressing files you yourself compressed.*** 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. It is not expected that your program will be able to decompress files compressed by other students' programs, since slight variations in the tree-building algorithm can cause the program to encode the same file in many different ways.

Possible Extensions

There are all sorts of fun extensions 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.

Submission Instructions

To submit this assignment, upload your `Encoding.cpp` file, along with `secretmessage.huf`, to Paperless, along with any other files you modified. And that's it! You're done!

Good luck!