CS106B Handout #23
Spring 2013 May 24, 2013

Assignment 6: Huffman Encoding

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

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 22) 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: Wednesday, June 5th at 2:15PM

Overview of the Program Structure

Your task is to build a program that uses Huffman encoding to compress and decompress files. Like all complex programs, development goes more smoothly if the task is divided into separate pieces that can be developed and tested incrementally. We have already split the program apart into smaller steps for you and provided extensive testing code to try to catch as many mistakes as possible. We provide a lot of starter files in this assignment, but you don't need to fully understand or modify most of them. You should primarily concern yourself with the following files:

- **`HuffmanEncoding.h/cpp`**: These files are where you will do the majority of your coding. We have defined a small set of functions that you will need to implement as part of your solution, and you will do most if not all of your coding here.
- **`HuffmanEncodingTest.cpp`**: This file contains the testing code and **`main`**. When you run the provided code, this is where execution begins. You might want to modify this code if you want to add your own tests.
- **`HuffmanTypes.h`**: A header file defining the node type you will use when building your priority queue, along with a few extra types that will come up in the implementation. You should not make any changes to this file.
- **`bstream.h/.cpp`**: A set of stream classes that you can use to read and write one bit at a time. You do not need to make any changes to these files, though if you're interested in the C++ streams libraries you might find the implementations interesting.

Three other files are provided – **`ReferenceHuffmanEncoding.h`** and **`MemoryDiagnostics.h/.cpp`** – which are used by our testing framework to help ensure that your code is working correctly. You do not need to and should not modify these files.

The structure of each of the new modules is described in more detail in the sections that follow.

The `bstream` Classes — Streams with Single-Bit I/O

Let's first start off with an easy module—the `bstream` module is already written for you and all you need to do is use it. The `bstream` module exports a set of streams classes (similar to the `istream` and `ostream` classes provided by C++) that allow you to read and write data one bit at a time. All of your favorite operations on file streams still work for the `bstream` classes; you can use `getline`, `get`, `<<`, `put`, `>>`, etc. with the `bstream` classes. Additionally, you have access to three new functions:

For `ibstream`:

`int readBit();`	Reads a single bit from the stream.
`void rewind();`	Moves back to beginning of file to read again.
`long size();`	Returns the number of bytes in the open file.

For `obstream`:

`void writeBit(int bit);`	Writes a single bit to the stream
`long size();`	Returns the number of bytes in the open file.

The streams libraries in C++ are part of a class hierarchy. The classes `istream` and `ostream` define generic behavior that is common to all input and output streams, respectively, while the subclasses `ifstream`, `istringstream`, `ofstream`, and `ostringstream` allow you to read and write data to and from files on disk and strings in memory. (If you have never seen this before, you might want to read part of Chapter 4 from the course reader, starting on Page 183.) Similarly, our `ibstream` and `obstream` classes are base classes with four subclasses: `ifbstream`, `istringbstream`, `ofbstream`, and `ostringbstream`, which let you do bit-level I/O to and from files and strings in memory. From your perspective, the functions you will be writing will always work on the more generic `ibstream` and `obstream` classes. This allows your code to be used for compression/decompression of files (for actual use) and of strings (for testing purposes).

Here is some sample code that uses the new classes:

```
ofbstream outfile;
outfile.open(name.c_str());
if (outfile.fail()) error("Can't open output file!");
outfile << 134;
outfile.put('A');
outfile.writebit(0);
outfile.writebit(1);
outfile.close();

ifbstream infile;
infile.open(name.c_str());
if (infile.fail()) error("Can't open input file!");
int num;
infile >> num;
cout << "read " << num << " " << char(infile.get()) << " and "
     << infile.readbit() << infile.readbit() << endl;
infile.close();
```

You will not need to do any work on this module; you're simply a client of its classes.

HuffmanTypes: Fundamental Types for Huffman Encoding

The **HuffmanTypes.h** header file exports two types: **ext_char**, which we'll discuss in a moment, and **Node**, which represents a node in a Huffman encoding tree. You should familiarize yourself with these types before starting your assignment, as the different pieces of the program will make extensive use of them.

The main type to pay attention to in this header file is **ext_char**. This type (short for "extended character") can take one one of three types of values:

- an actual **char** value;
- the constant **PSEUDO_EOF**, which represents the pseudo-EOF value (the symbol, denoted by ■ in the previous 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**, which represents something that isn't actually a character.

Our **Node** type, which represents a node in the encoding tree, stores an **ext_char** describing which character, if any, is represented by that node. Leaf nodes will either hold a real character value, or will store **PSEUDO_EOF** if that node represents the pseudo-EOF value. Internal nodes (nodes that are purely structural and have no actual character content) should have the value **NOT_A_CHAR**.

Throughout the program, you will probably use the **ext_char** type in places where you would normally use a regular **char** so that you can correctly encode either a **char** or the pseudo-EOF value.

HuffmanEncoding: The Actual Assignment

The actual code you will need to write for this assignment is contained within the **HuffmanEncoding.cpp** and **HuffmanEncoding.h** files. You will need to implement a total of seven functions, each of which handles one piece of the compression and decompression.

To begin with, you will need to be able to count up the frequencies of all the characters in a given file (plus the fact that the pseudo-EOF marker appears at the end). As a starting point and to help you get warmed up with our testing infrastructure, implement the function

```
Map<ext_char, int> getFrequencyTable(istream& file);
```

This function takes as input an **istream** connected to the text you need to build a frequency table for (which could be either a file on disk, a string buffer, etc.). It should return a frequency table mapping from **ext_char**s to the number of times that the character (or pseudo-EOF) appears in the file.

To test out your implementation, you can run the manual and automatic tests we've provided as part of the starter code to see what happens when you apply this function to real data. Once you have the ability to count up the frequencies of each letter in the file, it's time to build up the Huffman encoding tree for the data. Your next task is to write a function

```
Node* buildEncodingTree(Map<ext_char, int>& frequencies);
```

This function will take in a frequency table (perhaps the one you just built in the previous step!) and construct a Huffman encoding tree given the specified frequencies. You will use this tree as a key step of the encoding and decoding steps.

When building the encoding tree, you will need to use a priority queue to keep track of which nodes to process next. You have already proven in Assignment 5 that you know how to implement a priority queue, and you don't need to build another one from scratch for this assignment. Instead, look at the **PriorityQueue** type provided as part of the Stanford libraries. You can use it by including the **"pqueue.h"** header file. Unlike the priority queue you wrote for Assignment 5, this priority queue allows each element to be enqueued along with an associated priority. The priority queue then sorts elements by their priority, with the **dequeue** function always returning the element with the *lowest* priority. Consult the documentation on the CS106B course website for more information about how to use this type.

As long as you're writing code to build up a tree, you should also write some code to free the memory from that tree! You should also implement this function below:

```
void freeTree(Node* root);
```

which takes as input a tree, then frees all memory for it.

We have provided you with a fairly elaborate test suite for testing out your implementations of these two functions. The manual testing code will let you build up encoding trees for strings of your choice, which will let you inspect whether your trees are correct. Once you have that working, you can then run the automatic tests, which will subject your functions to a thorough battery of tests.

At this point, you have working code for building up an encoding tree. Now, your task is to implement the following pair of functions:

```
void encodeFile(istream&  infile, Node* encodingTree, obstream& outfile);
void decodeFile(ibstream& infile, Node* encodingTree, ostream&  outfile);
```

The **encodeFile** function will take as parameters an input file that should be encoded, an encoding tree for that file, and an **obstream** to which the compressed bits of that input file should be written. This function should read the input file one character at a time, writing the encoding of that input file to the specified output file. The **decodeFile** function should do just the opposite – it should read the bits from file **infile** one at a time, using the specified decoding tree to write the original contents of that file to the file specified by the **outfile** parameter.

In Handout #22, we mention the necessity of prefixing the compressed file with an encoding table so that you can later decompress it. When implementing **encodeFile** and **decodeFile**, you should *not* write or read this header from the file. Instead, just use the encoding tree you're given as a parameter. You will do the header management in a separate step.

To check that your implementations of **encodeFile** and **decodeFile** are working correctly, you can use our provided test code to compress strings of your choice into a sequence of 0s and 1s. You can use this to manually verify that the encoding is working correctly. Once you've done that, you can run our automatic test suite, which will compress a variety of files using your encoding and decoding routines.

The last step in this process is to glue all of your code together, along with code to read and write the encoding table to the file. Your final task is to implement these two functions:

```
void compress(ibstream& infile, obstream& outfile)
void decompress(ibstream& infile, ostream&  outfile)
```

The **compress** function takes as input an **ibstream** containing the contents of a file to compress, along with an **obstream** indicating where the compressed file should be written to, then writes a compressed version of the original file, plus a header, to the specified file. The **decompress** function does the reverse – it reads the encoding table from the file, then uses that information to decompress the rest of the file into the specified output file.

In the course of writing these functions, you will need to read and write a header to or from one of the files. In the starter code, we have provided you with working code that does just that. You're free to use this code if you would like, but if you'd like to build your own encoding table, feel free to modify it as you see fit!

As a final test of your completed implementation, you can use our provided program to compress or decompress actual files on your system. You can also run our automated test system, which will test your program on a variety of different files.

General Hints and Suggestions

- It will be helpful to include error checking in your functions as an aid for 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 `""`.
- Compressing a file will require reading through the file twice: first to count the characters, and then again when processing each character as part of writing the compressed output. The **`ibstream`** offers a **`rewind`** member function that will be useful here.
- 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 **`bstream`** objects.
- Our supplied testing harness has one extra debugging command, a simple operation to compare two files character-by-character and report the first position at which they differ or whether they match entirely. This will be useful when trying to verify that the decompressed result exactly matches the original. Your program doesn't need the match command (i.e. you do not have to implement this operation), it is just provided to you as a debugging aid in the demo version.
- 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 the role of each module and the various classes/functions each module exports.
- 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.
- Build test cases. Make small test files (two characters, ten characters, one sentence) to practice on before you starting trying to compress War and Peace. 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? Create sample files to test out your theories.
- 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.
- You are responsible for freeing memory. Make sure not to leak any tree nodes, and if you allocate any extra memory, be sure to deallocate it!
- Your program should be able to compress any non-empty file. 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.

- 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.
- Writing/reading bits can be 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.

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.

Good luck!