CS106X Handout 30
Autumn 2012 November 5th, 2012

Assignment 5: Huffman Encoding

Assignment was pulled together by Owen Astrachan (of Duke University)
and polished by Julie Zelenski.

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. You'll be impressed by the compression algorithm, and you'll be equally impressed that you're outfitted to implement a tool that imitates one you're already very familiar with.

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 31) 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 past offerings that those skimming the handout too quickly overlook several critical details. We recommend a careful, thoughtful reading, and we even marked a few sections with a large star™ to make extra special certain you digest some essential facts.

Due: Wednesday, November 14th at 5:00 p.m.

Overview of the program structure

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 divided the task up into four modules:

- **`huffman`**—This module contains the **`main`** function, and is handles user requests to compress and decompress files. It uses an **`Encoding`** object to build the encoding and **`bstream`** objects to read and write encoded bit patterns to and from disk.
- **`pqueue`**—This is a generalization of your priority queue from Assignment 4. **`Encoding`** objects use priority queues when building an encoding tree. We're including a template version of the **`PQueue`** with the starter files so you don't have to templatize any priority queue data structures yourself. The interface is slightly different than the one you dealt with in Assignment 4, so read the **`pqueue.h`** file. (In particular, the **`enqueue`** method takes a priority value.)
- **`encoding`**—The module defines and implements a class for managing a Huffman encoding. It should have operations to build an encoding tree and map from character to bit pattern and back. It also is responsible for reading and writing the file header containing the encoding table.

- **bstream**—This class is already written for you. It defines new stream classes with specialized I/O operations to read and write single bits.

The structure of each module is described in more detail right here. Read on.

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. It provides two new classes, **ibstream** and **obstream**. These two classes are basically identical to the standard **ifstream** and **ofstream** except that they add a few operations. The new operations are listed here, and you'll find the full specification in the **bstream.h** interface file.

for **ibstream**:

```
int readbit();          // read a single bit from the stream
void rewind();          // move back to beginning of file to read again
long size();            // number of bytes in the open file
```

for **obstream**:

```
void writebit(int bit); // write a single bit to the stream
long size();            // number of bytes in the open file
```

Use these new classes the same way you use **ifstream**s and **ofstream**s. Our classes are **ifstream** and **ofstream** subclasses, and everything that you can use on the standard classes also works for the new classes (**<<** and **>>**, the member functions **get**, **fail**, **open**, **close**, etc.). In your program you will use **ibstream** in the place of **ifstream** and **obstream** in place of **ofstream**. The new classes do everything the standard ones do, along with extended functionality to read and write single bits.

Here is some sample code that uses the new classes:

```
obstream 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();

ibstream 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 don't need to do anything for this module. You're just a client of its two classes.

The **PQueue** template

While building an optimal encoding tree, you'll use a priority queue to process a collection of nodes and partial encoding trees. At each stage you extract the two minimum trees, combine them into a new tree, and insert the new root node back onto the queue for later processing.

Encoding builds and manages character bit-pattern encoding

The **Encoding** class manages an encoding tree mapping each character to a unique bit pattern encoding. The class should implement the classic Huffman algorithm for building an optimal encoding tree for an input file. The class should also include functionality to manipulate a zipped file's header, which should codify the encoding tree used during compression in such a way that the same encoding tree is easily rehydrated come decompression time.

Here is our suggested starting point for the interface of the **Encoding** class:

```
class Encoding {
   public:
      Encoding();
      ~Encoding();

      void compress(ibstream& infile, obstream& outfile);
      void decompress(ibstream& infile, obstream& outfile);

   private:
      // data members and helper functions
};
```

You do not have to follow our suggested interface religiously, but it should give you an idea of what the **public** operations might look like. The data managed by this class is internally stored in the form of an encoding tree as described in the data compression handout. Building this tree comes from analyzing the input file to count the characters and then constructing the optimal tree via the Huffman algorithm. It's your job to design the data structure and any helper functions required to support the **public** operations.

The **Encoding** class should also maintain a secondary structure for quick lookup, instead of having to hunt through the encoding tree to find the character of interest. An effective strategy is to create a **string** array with entries for each character and assign the bit pattern for each character by tracing out the paths in the encoding tree. Then when you need to look up the encoding for a particular character, you have immediate access to it.

Going from bit pattern to character is a matter of tracing your way down the encoding tree making left and right turns on the 0 and 1's to find the character at the end (or possibly to discover that the bit pattern isn't valid for the given encoding). Given that your encoding should already maintain an encoding tree, you will need no additional structure to support converting a bit pattern to its char.

The class also needs to publish information about the encoding table to the compressed file and later read that information to recreate the encoding tree during decompression. The encoding is unique to each file, so we must store information about the encoding tree in a file header so we know how to re-interpret the compressed bit stream during a decompression operation.

There are many options you have for reading and writing the encoding table. You could store the table at the head of the file in a long, human-readable string format using the ASCII characters **`'0'`** and **`'1'`**, one character entry per each line, like this:

```
h = 01
a = 000
p = 10
y = 1111
...
```

Reading this back in would allow you to recreate the tree path by path. You could have a line for every character in the ASCII set; characters that are unused would have an empty bit pattern. Or you could conserve space by only listing those characters that appear in the encoding. In such a case you must record a number that tells how many entries are in the table or put some sort of sentinel or marker at the end so you know when you have processed them all.

As an alternative to storing sequences of ASCII **`'0'`** and **`'1'`** characters for the bit patterns, you could store just the character frequency counts and rebuild the tree again from those counts in order to decompress. Again we might include the counts for all characters (including those that are zero) or optimize to only record those that are non-zero. Here is how we might encode the non-zero character counts for the **`"happy hip hop"`** string (the 7 at the front says there are 7 entries to follow—6 alphabetic characters, and the space character):

```
7 h3 a1 p4 y1  2 i1 o1
```

If you're feeling really inventive, you can go even further in your quest to save space. For example, it's possible to store the bit patterns by writing a sequence of single bits or you can devise a means to efficiently dehydrate the tree itself and store it.

You can use any combination of I/O routines (**get**/**put**, **>>** and **<<**, **readbit**/**writebit**, etc.) to save and restore the encoding table. The most critical issue is that your reading and writing functions are mutually consistent. Whatever strategy you use to write the table must match the way you read it.

Hints and requirements for the Encoding class:

- This is certainly the most complicated of the four modules, so figure that you will spend the bulk of your time developing and debugging this class.
- Our suggested encoding interface uses **int** instead of **char** to avoid somewhat obscure issues with the details of the conversion between **int**s and **char**s. Take care to respect those types. For similar reasons, it is unwise to use a **char** to directly index into an array, so an **int** should be used instead. Our suggestion for avoiding problems is to not use any variables of type **char** for encoding. Use **int** everywhere instead. It can do everything **char** can and more.
- 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 `""`.
- You will use your priority queue class template from Assignment 4 to build the encoding tree. The new priority queue requires that you specify the priority every time you call **enqueue**.
- Any working format you devise for writing and reading the encoding table into the compressed file is acceptable. If you develop a successful space-saving technique (more compact than the list of character counts, let’s say) we’ll consider that an extension worthy of extra credit. The compression needs to be significant and beyond the scope of what’s easily managed by the majority of CS106X students.
- When testing, keep in mind that a program is only expected to decompress files that were compressed using the same encoding header format (i.e. files compressed with your program will not likely be compatible with our demo version and vice versa).

Huffman module—compression and decompression

The final module is the one that manages the compression and decompression tasks. It repeatedly offers the user the option to compress or decompress until the user is done. The functionality in this module could be organized into a class if desired, but more likely you will organize its functionality into a set of ordinary functions.

To compress a file, you will first create the optimal encoding for the input file, using an **Encoding** object. You write the file header containing the encoding table to the output file. Then you process the input file character by character, writing the compressed form of each character to the output file bit-by-bit using **writebit**. Once you have processed all the characters in the input, voila, a compressed file! The program then reports the size of the original file, the size of the compressed file, and the factor by which it was able to shrink the file.

To uncompress a file, you use an **`Encoding`** object to first read the encoding table from the file header so that it can recreate the Huffman tree that was used to compress the data. Then you read through the compressed file bit-by-bit using **`readbit`** and build up a bit pattern. Once that bit pattern matches a sequence for a particular character, output that translated character to the result file. Repeat until all bits processed and, presto, you have reconstituted the original file!

Hints and requirements for the `Huffman` module:

- As humans are careless creatures, be sure to verify that their responses are valid and re-prompt them where necessary to correct errors.
- 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.
- One way to get the size of a file is to use the **`long size()`** member function we supplied on our **`bstream`** classes which returns the length of the currently opened file.
- Our supplied demo 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.

The pseudo-EOF

Although our **`bstream`**s allow you to read and write single bits, all output is actually done in chunks, typically one full byte (8 bits) at a time. If your program writes exactly 37 single bits, 5 full bytes (40 bits total) are actually written to the file, which means there are 3 extra trailing bits. Because of the potential for the existence of these "extra" bits, decompression cannot simply read bits until there are no more left since it might read extra bits written out due to buffering. This means that when reading a compressed file, you **cannot** use code like this:

```
while ((bit = infile.readbit()) != EOF) {
   // process bit
}
```

To avoid this problem, you should invent a pseudo-**EOF** character and stop when the pseudo-**EOF** character is read (in compressed form) instead of reading to the true **EOF**. The last bits of a compressed file should be the bits that correspond to the pseudo-**EOF** char. While decompressing, read bits until you accumulate a bit pattern that represents a character. When the bit pattern just read matches the encoding for the pseudo-**EOF**, the decompression is completed.

It's probably easiest to just act as though ASCII value 256 (higher than any of the existing ASCII characters) represents the pseudo-**EOF**. When preparing the encoding tree, the pseudo-**EOF** with number of occurrences equal to 1 is explicitly added to the character frequency counts. It is assigned an encoding along with the other characters. When writing a compressed file, the last encoded pattern you write will be the one for the pseudo-EOF character. Then, your decompression process can use those bits to know when to terminate decompression.

General hints and suggestions

- First, 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 module 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. You'll probably want to work on the **Encoding** module, and work on the **Huffman** module last.
- 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.
- Create infrastructure to help debug. Since the encoded binary files are impossible to decipher in a normal text editor (try opening one – they look like garbage), it is next to impossible to figure out what has gone astray by looking at the contents of a malformed file. You'll have to be more inventive about coming up with ways to debug during your development. Building infrastructure (for example, debugging routines to print out the

frequency counts, printing out the encoding tree/table, writing a parallel file using ASCII `'0'` and `'1'` characters instead of bits, etc.) will prove to be useful. As Owen said about his Duke students when they did a similar assignment: "most students build test and debugging functions as part of the program or eventually wish they had."

- You are responsible for freeing memory. All your classes should have properly implemented destructors and the **`Huffman`** program should close all files and free the memory used after finishing each compression/decompression operation.
- Your program should be able to compress any nonempty file. Your implementation should be robust enough to compress any given file: text, binary, image, or even one it has previously compressed. Your program 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. Also, given that your and our versions won't necessarily store the file header containing the encoded table in the same format, it is not expected that your program will be able to decompress files compressed by the demo and vice versa.
- 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 takes a while.