Assignment 3

CS 311 Data Structures
Spring Quarter, 2006

DUE: Monday 05/08, 11:59pm

READ THE WHOLE HANDOUT CAREFULLY BEFORE YOU START WRITING CODE.

Huffman compression

In this assignment you will write a compression/uncompression program based on the Huffman algorithm described in class. See `http://www.cs.northwestern.edu/academics/courses/311/notes/huffman.pdf` for a detailed description of the algorithm you are to implement.

The program will be run with the following command arguments:

- A switch indicating the action to be performed. This could be `-c` for compress, `-u` for uncompress, or `-h` for help. See below for details about the various switches.
- The name of a file (only when the switch is `-c`, or `-u` .

The action taken in each case is:

- `huffman -c FILENAME`
  Compress the given file. This will generate a compressed (binary) file with the same name as the input text file and the additional extension `.huff`. For example, if the input file is called `paper.txt`, the compressed version will be called `paper.txt.huff`. If the input file is called `quotes`, the compressed version will be called `quotes.huff`.

  The original file should be deleted (hint: look up the *system()* function)
- `huffman -u FILENAME.huff`
  Uncompress the given file and recreate the original text file.
- `huffman -h`
  This option should print out a "manual" page with information about the use of the huffman program. Type `man gzip` to see an example of a manual page. You are not expected to write a manual page as long as gzip's, but you should make sure to include all information that is needed to successfully use the huffman program.
- Incorrect command line
  If the command line is written incorrectly (e.g. missing arguments), the program should print an appropriate message.

  Type `tar` on the command line, without any switches, to see an example of a typical message for this situation.

In addition to the binary file, you are strongly encouraged to generate a text file containing the encoded output (as characters 0 and 1 instead of bits 0 and 1). This should be called `FILENAME.debug` and it will help you see if there is anything wrong with the way your file is encoded. We will not be looking at this file. However, if your encoder does not generate a correct binary file but it does generate a correct `.debug` text file, then you will receive partial credit.

Data structures

There are three major structures that your program should use:

- Frequency table.
  Store the frequencies of the characters and any other piece of information associated with each character.
- Huffman tree nodes.
  A leaf is associated with an individual character and its frequency. An internal node is associated with the total frequency of the characters stored in the subtree rooted at that node.

  (Hint: You don't really need a separate tree class)
- Priority queue.
  This is used by the Huffman algorithm to create the tree.

Notes on encoding

The compressed file will contain three sections:

- The header
  This is information that must be placed at the beginning of the file to allow it to be uncompressed successfully. Read the next section for more information on the header.
- The compressed bit sequence (i.e. the compressed source file)
- A sequence of bits, called padding bits, that pad the file so that it contains a whole number of bytes.

The header

The header will contain three items:

The first is a magic number (in this assignment this is set to 111) which will be checked during uncompression to verify that its input was compressed by our program.

The second item has to do with the way we write into the compressed file. When writing into a binary file, we cannot write one bit at a time. Instead we must collect a number of bits in a buffer whose size is a multiple of one byte and write a bufferful at a time. For example, the buffer could be one word long. But what if, in the end, we only have 20 more bits to write? If our buffer is one word long it will have an extra 12 bits at the end. But these are not part of the encoding. When we uncompress the file, we'll need to know how many they are so we can ignore them and not try to decode them. These are called *padding bits* and their number (12 in this example) should be saved in the header.

Finally, when uncompressing, we will need to be able to regenerate the original Huffman tree, so we must store the relevant information into the binary file. There are several options here. We may store the frequency of each character, the code of each character or the tree itself. To store the tree perform a pre-order traversal and write each node that is visited. You must find a way to differentiate between internal and leaf nodes (e.g. by storing an extra bit for each node, 0 for internal, 1 for leaf). You are free to choose whichever option you prefer or come up with a new one. Bonus points will be given to the students who write the best compressor.

Notes on binary file I/O

Reading and writing

Data can be read or written only in byte multiples, so you will need to buffer it. The following example shows how to open a file for binary output and write a buffer in it.

```
ofstream outFile;
unsinged int buffer; // 32-bit long buffer
outFile.open("output.bin", ios::out | ios::binary);
outFile.write( (char *)& buffer, sizeof (unsigned int));
```

The second argument to `write` specifies how many bytes to write from `buffer` into `output.bin`.

If `buffer` is declared as an `unsigned int` then the line shown above will write the whole buffer into the file. For example, if `buffer=30`, then the bits written into `output.bin` would be 00000000000000000000000000011110

Shifting and masks

The next thing you need to know is how to make space into your buffer so you can add a bit to it. For example, if your buffer already contains 10011 (I have dropped the leading zeros from this example) and you want to "push" a bit to its right end, you need to shift everything one position to the left to make room for the additional bit. The left-shift operator is <<.

For example, if the buffer is 00000000000000000000000000010011 then

```
buffer = buffer << 1
```

shifts it one to the left and makes it 00000000000000000000000000100110.

In short, this can be written

```
buffer <<= 1
```

When uncompressing, you will need to be able to tell whether a specific bit in the buffer is 0 or 1. To check or modify the value of a bit we use masks. Here's an example:

Suppose your buffer is 011011 and you want to make the last bit a zero. If you had the sequence 111110 and you took the bitwise logical AND between the buffer and the sequence (011011 & 111110) you would get 011010 which is the original buffer with a 0 at the last bit! The sequence 111110 is called a mask.

Now, suppose your buffer is 011011 and you want to see what the first bit is. Again, use a mask. If mask==100000, then, if the first bit of buffer is 1, (buffer & mask) will be the same as mask. If the first bit of buffer is 0, then (buffer & mask) will be 000000

If you know the size of your buffer, then you can easily create the appropriate mask. For example, if the buffer is an unsigned int, then a mask that contains all 1s would be the same as the maximum possible unsigned int (see limits.h).

Moving around

When reading from a file, the `get` pointer is a pointer to the current byte in the file. You can reposition the `get` pointer by using the `seekg()` function. `seekg(x)` places the `get` pointer x bytes from the beginning of the file.

Similarly, the `put` pointer points to the current byte in a file we are writing to. Repositioning the `put` pointer is done with the `seekp()` function.

When you reach the end of a file that you are reading, the EOF flag is set. If you then wish to go back to the beginning of the file and read it again, you must reset the flag. You can reset all flags by using the `clear()` function. See `http://cppreference.com/cppio/index.html` for more examples.

Notes on decoding

The uncompressing function should first read the magic number to verify that the given source file is a file compressed by the same program. If it is not, it should print an error message and abort. iIf the file is valie, it should go on to read the number of padding bits followed by the information needed to generate the Huffman codes. The next step involves reading bits (a bufferful at a time) and decoding them to get back the uncompressed text file. When reading the last bufferful, the padding bits should be ignored.

Partial credit

Students whose encoders generate only the `.debug` text file and whose decoders read from `.debug` instead of `.huff` will receive partial credit.

Extra credit

The best compressions will receive bonus points.

Submitting your files

Submit your files in a tarball. Send your code to `cs311@cs.northwestern.edu`. Make sure that the subject line contains "Programming Assignment" and do not neglect to cc yourself and verify that the files were attached properly and were not corrupted in any way.