Programming assignment #3: CSE 100 Spring 2013 (Section A00)

In this assignment, you will implement a Huffman code file compressor and decompressor in C++. This will require implementing Huffman's algorithm using efficient supporting data structures, and also will require extending the basic I/O functionality of C++ to include bitwise operations.

>>> Due Fri May 17 8:00 PM

>>> Required solution files: compress.cpp, uncompress.cpp, HCNode.hpp, HCNode.cpp, HCTree.hpp, HCTree.cpp, BitInputStream.hpp, BitInputStream.cpp, BitOutputStream.hpp, BitOutputStream.cpp, SIGNATURE, DECLARATION

The files that contain your solution to the assignment must be turned in by the deadline.

To turn in your assignment, the procedure is the same as for your previous assignments, except the turnin script is named `bundleP3`.

If you are working in a team of 2 students on this assignment, turn in your assignment from *one* of your two accounts only! In your source code files, clearly document the name(s) and login name(s) of the author(s), using the `SIGNATURE` and `DECLARATION` file as in the last assignment (be sure to update the date in the DECLARATION FILE).

It's a good idea to plan to do the assignment well before the deadlines; terminals and tutors can get very busy at the last minute. In doing this assignment, it's your responsibility to understand the course rules for integrity of scholarship.

Getting started

Read Weiss, Chapter 10 section 1; this assignment uses some concepts introduced there. Lecture notes on binary tries and Huffman code trees will also be useful.

File compression and decompression

Even though disk space and bandwidth is cheaper now than ever before in history, there is also more data than ever before. So it is still very useful to be able to compress disk files and network data, thereby allowing a given amount of disk to hold more data, or a given network link to carry more data. It is often possible to make a file significantly smaller without any loss of information whatsoever ("lossless compression"). The trick is to figure out how to do that, and how to reconstruct the original file when needed. Here, we will implement file compression using Huffman coding, a clever technique invented by David Huffman in 1952. Huffman code compression is used today as part of the JPEG image compression and the mp3 audio compression standards, in the Unix file compression command `pack`, and other applications (the popular `zip` and `gzip` compression utilities use compression algorithms somewhat different from Huffman).

You will write a C++ program `compress` that will be invoked with a command line of the form

```
compress infile outfile
```

When run, this program will read the contents of the file named by its first command line argument, construct a Huffman code for the contents of that file, and use that code to construct a compressed version which is written to a file named by the second command line argument. The input file can contain any data (not just ASCII characters) so it should be treated as a binary file. Your compress program must work for input files up to 10 megabytes in size, so a particular byte value may occur up to 10 million times in the file.

You will also write a C++ program `uncompress` that will be invoked with a command line of the form

```
uncompress infile outfile
```

When run, this program will read the contents of the file named by its first command line argument, which should be a file that has been created by the `compress` program. It will use the contents of that file to reconstruct the original, uncompressed version, which is written to a file named by the second command line argument. In particular, for *any* file `F`, after running

```
compress F G
uncompress G H
```

it must be the case that the contents of `F` and `H` are identical.

Control flow

Taking a top-down design approach to decomposing the problem, you can see that your `compress` program basically needs to go through these steps:

1. Open the input file for reading.
2. Read bytes from the file, counting the number of occurrences of each byte value; then close the file.
3. Use these byte counts to construct a Huffman coding tree.
4. Open the output file for writing.
5. Write enough information (a "file header") to the output file to enable the coding tree to be reconstructed when the file is read by your `uncompress` program.
6. Open the input file for reading, again.
7. Using the Huffman coding tree, translate each byte from the input file into its code, and append these codes as a sequence of bits to the output file, after the header.
8. Close the input and output files.

Thinking along similar lines, your `uncompress` program should go through these basic steps:

1. Open the input file for reading.
2. Read the file header at the beginning of the input file, and reconstruct the Huffman coding tree.
3. Open the output file for writing.
4. Using the Huffman coding tree, decode the bits from the input file into the appropriate sequence of bytes, writing them to the output file.
5. Close the input and output files.

Data structures and object-oriented design

One crucial data structure you will need is a binary trie that represents a Huffman code. The `HCTree.hpp` header file is a start on a possible interface for this structure (available in the public/P3 directory). You can modify this in any way you want. In addition you will write a companion `HCTree.cpp` implementation file that implements the interface specified in HCTree.hpp, and then use it in your compress and uncompress programs.

In implementing Huffman's algorithm to construct the trie, you will find it convenient to use other data structures as well (for example, a priority queue is useful, to maintain the forest of trees ranked by count). Any other data structures you find you need, can be implemented in any way you wish; however, you should use good object-oriented design in your solution. For example, since a Huffman code tree will be used by both your `compress` and `uncompress` programs, it makes sense to encapsulate its functionality in its own class, so that they can both use it. With a good design, the main methods in the `compress` and `uncompress` programs will be quite simple; they will create objects of other classes and call their methods to do the necessary work.

One important design detail in this assignment is: how to represent information about the Huffman code in the compressed file, so the file can be correctly uncompressed. Probably the easiest way to do it is to save the frequency counts of the bytes in the original uncompressed file as a sequence of 256 ints. Since this is the information that `compress` uses to create the Huffman code in the first place, it is sufficient. However, it is not very efficient in terms of space: it uses 1024 bytes of disk for the header no matter what the statistics of the input file are. Alternative approaches may use arrays to represent the structure of the tree itself in the header. Other schemes also work. With some cleverness, it is possible to get the header down to about `10*M` bits, where `M` is the number of distinct byte values that actually appear in the input file. *However, don't attempt to reduce the size of the header, until you've gotten your compress and decompressor to work correctly for the provided inputs, and that you can achieve a compression factor at least as good as the reference's, for the provided input file* `input_files/warandpeace.txt`.

Bitwise I/O

All disk I/O operations (and all memory read and write operations, for that matter) deal with a byte as the smallest unit of storage. But in this assignment, it would be convenient to have an API to the filesystem that permits writing and reading one bit at a time. Define classes BitInputStream and BitOutputStream (with separate interface header and implementation files) to provide that interface.

To implement bitwise file I/O, you'll want to make use of the existing C++ IOstream library classes `ifstream` and `ofstream` that 'know how to' read and write files. However, these classes do not support bit-level reading or writing. So, use inheritance or composition to add the desired bitwise functionality.

Testing

Test components of your solution as you develop them, and test your overall running programs as thoroughly as you can to make sure they meet the specifications (we will certainly test them as thoroughly as we can when grading it). Be sure to test on "corner cases": an empty file, files that contain only one character repeated many times, etc. "Working" means, at minimum, that running your `compress` on a file, and then running your `uncompress` on the result, must reproduce the original file. There are some files in `~/../public/P3/input_files` that may be useful as test cases, but you will want to test on more than these.

Notes and hints

Note that there are some differences between the requirements of this assignment and the description of Huffman coding in the textbook; for example, your program is required to work on a file containing any binary data, not just ASCII characters.

It is important to keep in mind that the number of bits in the Huffman-compressed version of a file may not be a multiple of 8; but the compressed file will always contain an integral number of bytes. You need a way

to make sure that any "stray" bits at the end of the last byte in the compressed file are not interpreted as real code bits. One way to do this is to store, in the compressed file, the length in bytes of the uncompressed file, and use that to terminate decoding. (Note that the array of byte counts implicitly contains that information already, since the sum of all of them is equal to the length in bytes of the input file.)

The files `refcompress` and `refuncompress` in that directory are programs that meet the requirements of this assignment, if you want to refer to a reference solution for some reason.

Note: your compress is not expected to work with refuncompress, and your uncompress is not expected to work with refcompress. The provided programs are a matched pair

Please don't try out your compressor on large files (say 10 MB or larger) until you have it working on the smaller test files (< 1 MB). Even with compression, larger files (10MB or more) can take a long time to write to disk, unless your I/O is implemented efficiently.The rule of thumb here is that most of your testing should be done on files that take 15 seconds or less to compress, but never more than about 1 minute. If all of you are writing large files to disk at the same time, you'll experience even larger writing times. Try this only when the system is quiet and you've worked your way through a series of increasing large files so you are confident that the write time will complete in about a minute.

To see the size of a file from the Unix command line, you can use the `wc` command with the `-c` flag:

```
wc -c words.cmp
```

will display the length of `words.cmp` in bytes. Or consider the `ls` command with the `-l` flag:

```
ls -l words.cmp
```

This will display a long listing of information about `words.cmp`, including its length in bytes.

To inspect the contents of a file, especially a binary file, it can be useful to use the 'octal dump' command, `od`. The `-t x1` or `-t u1` flags let you see the values of individual bytes in hexadecimal or unsigned decimal format. See the manual page (do `man od`) for more information.

To quickly check if two files are identical, use the `diff` command. For example,

```
diff foo bar
```

will print nothing if `foo` and `bar` are identical. If they are different, `diff` will print out some information about how they differ. This can be useful to test if your `uncompress` is really undoing what your `compress` does. (A `compress` implementation without a matching `uncompress` implementation, or vice versa, is really useless and will be graded accordingly.)

Grading

There are ~~100~~ 25 possible points on the assignment. (The weighting is the same as the other assignments, we are just changing the way account for points). If your solution files do not compile and link error-free, you will receive 0 points for this assignment. We will compile your files as in the Makefile distributed with the assignment, on ieng6.

To get full credit, style and comments count.

~~80~~ 20 points for your compress and uncompress programs that use Huffman coding to correctly compress and decompress files. *We define acceptable compression to be within 10% of the supplied reference*

*compress program So if the reference compresses a given file to 100KB, you will not lose credit if your **correctly** compressed files are not larger than 110KB You will not be requried to demonstrate compression for files smaller than 10KB, but you must compress them **correctly**. We define correctly in that you can recover the original file from the compressed version.*

~~12~~ 3 points for a correct implementation of compress that creates a smaller compressed file than the reference implementation when run on three particular files, including `~/../public/P3/input_files/warandpeace.txt`. *The other two tests files will not be disclosed, but they will not be shorter than 10KB. (They may or may not be larger, that is left unsaid).*

~~8~~ 2 points for a good object-oriented design, good coding style, and informative, well-written comments.

3 ~~points~~ percent if submitting as a team of two **or** submitting more than 48 hours before the deadline. *Up to 3 points extra credit will be awarded to those who implement a 1 pass compressor as discussed in class. The idea is to read the input in blocks that fit into memory and compress each block separately. You'll need to figure out a good block size, that's part of the experimentation. To get all the extra credit, you need to speed up the write time by a factor of two on files on the order of 20MB or larger.*