Assignment 3 README

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 Nov 15 8:00 PM~~ Mon Nov 18 8:00 PM

>>> Checkpoint deadline: ~~Tues Nov 12 8:00pm~~ Wed Nov 13, 10:30pm

>>> Required solution files:
>> CHECKPOINT SUBMISSION: (at least) compress.cpp, uncompress.cpp, HCNode.hpp, HCNode.cpp, HCTree.hpp, HCTree.cpp
>> FINAL SUBMISSION: compress.cpp, uncompress.cpp, HCNode.hpp, HCNode.cpp, HCTree.hpp, HCTree.cpp, BitInputStream.hpp, BitInputStream.cpp, BitOutputStream.hpp, BitOutputStream.cpp

The files that contain your solution to the assignment must be turned in (i.e. tagged with the commit message "FINAL SUBMISSION" and pushed to github) by the deadline. The checkpoint submission MUST contain the files listed above and be tagged "CHECKPOINT SUBMISSION" before the checkpoint deadline. You do not need to push this submission to github (but you are encouraged to do so), but this message much be present and this commit must contain all of the files listed above. More information is available below in the section Checkpoint Deadline.

To turn in your assignment, the procedure is the same as for your previous assignments. You must verify that (a) your checkpoint and final versions of the repository contain all of the required files and (b) that all of these files have been successfully pushed to github after your final commit. **If you are missing files, you will not receive credit for what is missing. Be sure to log on to github to verify what is in your repository!**

A very simple check script (`checkFiles.sh`) has been provided to you to double check if your DIRECTORY contains all of the required files. You will need to verify they are on Github yourself. To run the script type following:

```
cd PA3_DIRECTORYNAME
```

```
./checkFiles.sh PA3
```

This script will either output a list of the missing files, or print that you have everything you need. Please make sure to run it before submitting your final submissions.

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 Drozdek, Chapter 11.1-11.2; this assignment uses concepts introduced there. Lecture notes on Huffman code trees (which are a form of binary trie) will also be useful, as will readings and lecture notes on C++ I/O.

What your code will do: File compression and decompression

First some background: 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).

In a nutshell, "all" you will do in this assignment is to write two C++ programs. The first is a program named `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 (`infile`), 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 (`outfile`). 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.

The second program you will write is 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 anyfile F, after running compress F G uncompress G H it must be the case that the contents of F and H are identical.

How to break the problem down: Thinking about 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.

The challenge of this assignment is to put this high-level algorithm into code. Notice that we have provided you with very little starter code, though the required submission files, and the text below (as well as class and discussion sections!) contain many hints on how to perform your implementation.

Data structures and object-oriented design

One crucial data structure you will need is a binary trie (i.e. code tree or encoding tree) that represents a Huffman code. The `HCTree.hpp` header file is a start on a possible interface for this structure (included in your repo to start). 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. Note that you will also need the files HCNode.hpp (provided, you may modify) and HCNode.cpp in your submitted repo (both at both the checkpoint and the final) commit. **Be sure to add HCTree.cpp and HCNode.cpp (and all other files you create that are required for the submission) into your repo! You will not get credit for them if they are not in your repo!**

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.

Checkpoint Deadline: Working "compression" using ASCII, on small files only

The rest of the writeup below gives you some crucial information about implementing the full solution. However, for your checkpoint deadline you've got all the information you need. *For your checkpoint submission* your compress and uncompress should work as follows:

- `compress infile outfile` should "compress" the infile by building the appropriate Huffman code tree based on the frequencies in infile (which is guarenteed to have only ASCII text and be very small (<1KB)), and then writing out the tree and the coded infile *as plain (ASCII) text* to outfile. That is outfile should be a readable text file that contains your representation of the tree as well as the code that represents the original file. I.e., the 1's and 0's in your code will be written to the "compressed" file using the ASCII characters for 1 and 0. This is why "compress" is in quotes. Your "compressed" outfile will actually be larger than the original! (The point here is to get the algorithm working).
- `uncompress infile outfile` should "uncompress" the infile by reading in the code tree (which is represented at the beginning of the file) and then "uncompressing"(i.e. decoding) the rest of it and writing it back to the outfile, as plain (ASCII) text.

We will grade your checkpoint submission as follows:

- We will use your compress program to encode several files. If they produce a file, +1
- We will look at these files to ensure their contents "look" encoded. (I.e., they can't just contain the same text as the original file!). If the files look encoded +2.
- We will use your uncompress program to uncompress the "compressed" files. If we get back the original files, another +2, which rounds out the +5 for the checkpoint deadline.

See the section below for some hints on testing your program (many of the overall testing hints apply equally to the checkpoint). Also, note that there are suggested methods in the `HCTree.hpp` header file that you might find useful specifically for this checkpoint, but that you should NOT use for the final submission. Finally, because you are not required to implement or use `BitInputStream` and `BitOutputStream` for the checkpoint, you will either need to remove all references to them from the provided files (by commenting them out) and edit the Makefile, or create "dummy" versions of these classes (with both header and implementation files) to get the code to compile.

The rest of the work

In the rest of the assignment, you will lift the restrictions from the checkpoint. This means that your programs must deal with:

- Input files that will be MUCH larger than 1KB (the provided files, discussed below, give you some ideas of the size we mean)
- Input files that are not restricted to text files (may be binary files)
- Writing compressed files that are actually smaller than the input file.

For more information on how to go from the checkpoint to the final submission, read on.

Bitwise I/O

Well, as you learned from your checkpoint, if you encode your files using ASCII representations of 0 and 1, you don't get any compression at all. In fact, quite the opposite! So in this section, now that you've got your Huffman tree working, you'll modify your code so that you can write data to a file one bit "at a time".

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 (as you saw in the checkpoint), 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). Getting the checkpoint submission working is a great middle-step along the way to a full working program. In addition, because the checkpoint writes in plain text, you can actually check the codes produced for small files by hand!

For all deadlines (checkpoint and final), 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 provided in ../public/P3/input_files (and also in your repo) that may be useful as test cases, but you will want to test on more than these.

(More) 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` that you will find in your repo 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.)

Star points!

Many of you are doing quite well and seem ready for more of a challenge. So this PA we're introducing the option to earn what we call "star points" by going above and beyond the assignment specification. What are star points, you ask? They are points that you earn that make you feel good about completing a challenge. They mean you are a star! Seriously, though, they don't factor directly into your grade, but they will push you up at the end of the quarter if you are on the border of two grades.

The star point option for this PA is, in addition to implementing the required compress and uncompress programs, add a flag to your compress and uncompress programs -adaptive. If this flag is present, your coding will occur in *only one pass* as described in section 11.2.1 in your book. The requirements for your 1-pass adaptive compression are the following:

- It must compress and uncompress just as your required programs did (i.e. successfully)
- It must run significantly faster than your two-pass approach, particularly for large files (should be about half the time)
- It must compress the files to "close to" the size of the compression produced by the two-pass approach, where "close to" is defined as much closer to the size of the one-pass compressed file than to the size of the original file

The last two criteria will be easiest to test with large files.

Some caveats and details:

- This extension is NOT EASY. You will not be able to get much (if any) support from the course staff. You're really on your own here, if you choose to accept the challenge.
- Because it's so challenging, you can submit it at any time until the end of the quarter

- Because we want you to take this seriously (not just throw us any old thing in hopes that you'll get a little bonus), you must ensure that you submit something that meets the above requirements. If you submit a star point submission that *obviously does not work* you will earn a **negative star point** which has the opposite of a star point on your final grade.

The bottom line here is, do this for fun and challenge, not to try to help your grade. If you are doing this, you shouldn't need help with your grade anyway.

If you accept the challenge with a working submission, submit the star point submission by committing to your repository with the commit message "STAR POINT SUBMISSION" and then pushing to github. This commit will not affect the date that you submitted PA3. I.e. if you push your repository with the STAR POINT SUBMISSION" message after the deadline, you will not be charged slip days (as long as your FINAL SUBMISSION checkpoint was before the original deadline).

Grading

There are 25 possible points on the assignment. 5 points are awarded based on the checkpoint, as described above. 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.

15 (remaining) points for your compress and uncompress programs that use Huffman coding to correctly compress and decompress files of all sizes (from small, to very large).

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 (and two others we won't tell you about in advance).

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

Comments

You do not have permission to add comments.