Programming Assignment 3: Huffman Compress/Decompress

Review submission deadline: Monday, November 6, 11:59 PM

Final submission deadline: Thursday, November 9, 11:59 PM

Assignment Overview

*** Remember to join a PA3 group XXX(a number) with your partner, if any, before* ***November 5, 11:59 PM****. We won't accept any re-group request after that. Join an empty group if working alone.*

In this assignment you will:

- Implement Huffman's algorithm using efficient supporting data structures to support encoding and decoding of files
- Justify why your algorithm runs correctly on test examples with ASCII
- Extend the basic I/O functionality of C++ to include bitwise operations

Provided files:

HCNode.h, HCTree.h, Makefile, refcompress, refuncompress, .gitignore, asnlib/ (with binary.bat, check1.txt, check2.txt), submit.sh

We also provide some test files in ieng6, see "*Testing Guide"* near the end of this document.

Goal: To implement Huffman's Algorithm, using ASCII I/O to write and read the encoded files.

Instructions

Step 1. Implement HCNode and HCTree

The HCNode and HCTree implementations will help you create Huffman tree/code for the input files.

Implement the following:

1) Implement the `HCTree.h` methods in a new file `HCTree.cpp` (from scratch). You can modify both files in any way you want.

2) Implement the `HCNode.h` methods (overloaded operator) in a new file `HCNode.cpp`. You can modify both files in any way you want.

3) Because you do NOT need to implement or use `BitInputStream` and `BitOutputStream` yet, you will either create "dummy" versions of these classes (with both header and implementation files) to get the code to compile, or remove all references to them from the provided files (by commenting them out) and edit the `Makefile`.

4) When you finish this step, you should have added `HCTree.cpp`, and `HCNode.cpp` with all the methods implemented, except for `encode()` and `decode()` using `BitInputStream` and `BitOutputStream`, respectively.

Note: When implementing Huffman's algorithm, you should use multiple data structures (e.g., a priority queue, etc.). You should also use good object-oriented design. For example, since a Huffman code tree will be used by both your `compress` and `uncompress` programs, it makes sense to encapsulate its functionality inside a single class accessible by both programs. 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.

Step 2. Implement `Compression`

Create `compress.cpp` (from scratch too!) to compress small files in plain ASCII. The `compress.cpp` should generate a program named `compress` that can be invoked from the command line as follows

```
> ./compress infile outfile
```

`./compress` will:

- Read the contents of the file named (`infile`).
- Construct a Huffman code for the contents of that file
- Use that code to construct a compressed file named (`outfile`).

The challenge of this assignment is to translate the following high-level algorithm into real code:

Implement The Following Control Flow in `compress.cpp`:

1) Open the input file for reading. (`infile` can be assumed here to have only ASCII text and to be very small (<1KB))

2) Read bytes from the file. Count the number of occurrences of each byte value. Close the file.

3) Use the byte counts to construct a Huffman coding tree. Each unique byte with a non-zero count will be a leaf node in the Huffman 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. You should write the header as plain (ASCII) text for this step. See *"the file header demystified"* right below for more details.

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 (a new line). For here, this will be done with plain ASCII characters '1' and '0'.

(**Note**: you have just written your entire `outfile` in ASCII characters. Thus, your "compressed" file will actually be larger than the original one! The point here is purely to get the algorithm working.)

8) Close the input and output files.

9) Test your solution. For more details, see the "*Testing Guide*" section near the end of this document.

Note: Your `Makefile` must create the executables "`compress`" and "`uncompress`" with those exact names from the "`make all`" command. The `Makefile` given to you already does this, so just make sure you don't change this part of the file.

The "file header" demystified

Both the `compress` and `uncompress` programs need to construct the Huffman Tree before they can successfully encode and decode information, respectively. The `compress` program has access to the original file, so it can build the tree by first deciphering the symbol counts. However, the `uncompress` program only has access to the compressed input file and not the original file, so it has to use some other information to build the tree. The information needed for the `uncompress` program to build the Huffman tree is stored in the header of the compressed file. So the header information should be sufficient in reconstructing the tree. Note that the "file header" is not a .h file but rather the top portion of the compressed file.

Here is the way to design your header at the step: A straightforward, non-optimized method that you MUST USE in order to receive points!

It is to save the frequency counts of the bytes in the original uncompressed file as a sequence of 256 integers at the beginning of the compressed file. You MUST write this frequency-based header as 256 lines where each line contains a single `int` written as plain text. E.g.:

```
0
0
0
23
145
0
0
...
```

Where 0's represent characters with no occurrences in the file (e.g. above the ASCII values 0, 1 and 2 do not occur in the file), and any non-zero number represents the number of times the ASCII value occurs in the file.

<u>Step 3. Implement `Uncompress`</u>

Create `uncompress.cpp` (from scratch!!) that will use your above implementations to support decompress for small files. The uncompres.cpp should generate a program named `uncompress` that can be invoked from the command line as follows

```
> ./uncompress infile outfile
```

`./uncompress` 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
- 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. The uncompressed file must be exactly identical to the original file.

Implement The Following Control Flow in `uncompress.cpp`:

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.

6) Test your solution. For more details, see the "*Testing Guide*" section near the end of this document.

Step 4. Writeup

Verify your compression in **Report.doc** One easy way to verify your compression is to manually compress some simple strings in the following way:

Implementation Checklist:

1) Run your compressor using the provided input files: **check1.txt** and **check2.txt**.

2) Record the encoded output in **Report.doc** (will be converted to PDF at Step 6.) Don’t just copy and paste the output, which would contain many zeros. Specify your header by the line numbers and non-zeros, and ignore the zero frequency lines.

For example,

line 48 12

line 49 10

Encoded string: 111001001

…

3) Use the same input to manually construct a Huffman coding tree. Draw the Huffman coding tree, describe how you build the tree and how you find the code word for each byte in your output.

4) Manually encode the strings and compare with your compressor output. If your output is wrong, explain why your hand-coded text is different from your compressor output and how you fixed it.

5) We expect it to be 1-3 pages at this point. You have reached about 40% of this PA.

Step 5. Extend your functionality to use Bitwise I/O to actually compress the files

You will now implement a full Huffman Compression program along with bitwise I/O.

Implementation Checklist:

1) Implement Bitwise I/O (See "*Bitwise I/O*" below for details about why we are doing this)

2) Implement `BitInputStream`

3) Implement `BitOutputStream`

4) Implement `encode()` and `decode()` using `BitOutputStream` and `BitInputStream` in HCTree

5) Implement `compress` with new designed header

6) Implement `uncompress`

Note:

- You must handle input files that will **be MUCH larger than 1KB** (up to 1GB so a particular byte value may occur up to 1 billion times in the file) See "*Efficient header design*" below for more details.
- You must handle input files that are **not restricted to text files** (binary files, images, videos etc.)
- To receive any credit, your compressed files must be at least as small as the compressed files produced by the reference solution, within **~25%** of the compression obtained by the reference, that is, to have smaller than ~125% of compresson by the reference solution, and your programs must properly compress and uncompress the files exactly.
- Compressing and uncompressing the files should be done within **a timeout of 180 seconds** for files smaller than 30mb.
- For bigger files (100-300mb) your method should be able to compress and uncompress **within twice the time it takes for reference solution**.
- You will gain 2 points for a correct implementation of `compress` that creates a smaller compressed file than the reference implementation (at least 1 byte smaller), running on two particular files, including a test file called `warandpeace.txt` on ieng6 (and one other we won't tell you about in advance). This means that you should use a more efficient method for writing the header and Bitwise I/O to write and read the codes.
- You will gain 1 point if you are able to compress huge files (like 1GB). If you used a different algorithm/method for compressing huge files mention that in your writeup (Report.doc).

Bitwise I/O

If you encode your files using ASCII representations of 0 and 1, you don't get any compression at all because you are using 1 byte to store the '0' or '1'. Once 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, 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. Refer to the lecture notes for more information.

Efficient header design

The reference solution header is not very efficient. It uses 4-byte `int` to store the frequencies of each character, using 4*256 bytes for the entire header no matter what the statistics of the input file are.

In order to earn full credit for your final submission, you must BEAT our reference solution by coming up with a more efficient way to represent this header.

There are several possible solutions, but a good approach is to represent the structure of the tree itself in the header. With some cleverness, it is possible to optimize the header size to about 10*M bits, where M is the number of distinct byte values that actually appear in the input file. However, we strongly encourage you to implement the naive (1024-byte) approach first, and do not attempt to reduce the size of the header until you've gotten your `compress` and `uncompress` to work correctly for the provided inputs.

Step 6. Writeup again

In your **Report.doc**, please add to the end that how you designed a more efficient header. Specifically, what are your algorithm and data structure to make a smaller header than the naive 1024-byte approach.

When you are done, take a deep breath, and **convert your Report.doc to PDF**.

Testing Guide

Getting the first 3 steps working is a great middle-step along the way to a full working program. Remember large programs are hard to debug. So test each function that you write before writing more code. We will only be doing "black box" testing of your program, so you will not receive partial credit for each function that you write. However, to get correct end-to-end behavior you must unit test your program extensively. For some useful testing tools, refer to this doc. Don't try out your compressor on large files (say > 10 MB) until you have it working on the smaller test files (< 1 MB).

Even with compression, larger files 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.

Edge Cases Checklist:

1) Empty File

2) Files that contain only one character repeated many times

3) Think of more edge cases yourself!

The reference solution (`refcompress` / `refuncompress`):

You can use the "reference" implementations to verify your results (file size).
Note:

- Your `compress` is not expected to work with `refuncompress`, and your `uncompress` is not expected to work with `refcompress`. The provided reference executable files are a matched pair.
- The reference binaries were compiled to run on ieng6. If you attempt to run it on a different architecture they will most likely not work.

Data files for test (Input test files):

The data files are available in the public folder on your ieng6 server (NOT in the starter code provided to you). The path for these input files is:

```
/home/linux/ieng6/cs100f/public/pa3_input_files
```

If you want to have them in your home directory for your convenience, we would advise against copying them, as ieng6 accounts have relatively limited storage space. Instead, create a symbolic link to the directory:

```
ln -s /home/linux/ieng6/cs100f/public/pa3_input_files
```

Submission Guide

A submission script is given in the starter code. Inside the folder with all the files, run

```
./submit.sh
```

Required solution files for final submission:

Makefile, compress.cpp, uncompress.cpp, HCNode.h, HCNode.cpp, HCTree.h, HCTree.cpp, BitInputStream.h, BitInputStream.cpp, BitOutputStream.h, BitOutputStream.cpp, Report.pdf

It will run a simple test with all the required files, compilation, check1.txt, binary.dat (53kB binary file with 12 unique symbols most of which have byte values 0-4), valgrind memory leak test.

If all test pass, `PA3.tar.gz` will be produced. Please submit this file to TritonEd in your NEW group for PA3. You need to join a new PA3 group with or without a partner in order to submit.

Grading Overview

This assignment is out of **27 points**.

- `Compress` and `Uncompress` correctness: 16 points
- Beat our reference solution and compress our large file: 3 points
- Report.pdf: 5 points
- Commenting and style (see the minimal style guide): 3 points

Note: If you are stuck, go to Piazza first and look through or search for possible related questions. Post questions if no one has ever asked before. Going to lab hour and ask for tutors' advice is useful but don't go there until the last minute!

Star Point (Optional)

There are three purely optional components of this assignment that are independently worth a "star point":

1. Implement Adaptive Huffman Encoding described here
2. Implement Huffman Encoding "by parts":
    - Huffman coding depends on symbol frequencies within a message, so it is clear that messages with significantly different symbol frequencies might not reach optimal compression if we look at them in their entirety. For example, in the human genome (a string of As, Cs, Gs, and Ts), some regions are rich in AT and others are rich in CG, but zoomed out, we have roughly 25% frequency of each nucleotide.
    - If we were to split the message into chunks, we could better compress each individual chunk. However, each time we split the file, we add an overhead file size cost because of the added header we need to encode.
    - Can you think of an optimal way to split up a given string Message into chunks that result in the optimal (i.e., minimal) file size when Huffman compression is run on each individual chunk of the file?
        - Input: A string Message
        - Output: Huffman compression of the chunks of Message resulting in optimal cuts
    - Note: A correct and efficient solution may not exist, so if you suggest some heuristic, you must provide explanation as to why your heuristic would be good.

It is sufficient if you complete 1 of the above two problems.

We require you to turn in

1. A writeup named starpoint.pdf that explains:

- What you did?
- Output and Description of test cases (Showing your code works). You must test them on all the test files that we provided you for the PA.
- A comparison between your technique and the standard Huffman Compression algorithm in terms of a) Running Time (Empirical) b) Compression
- An explanation about why you see these results.

2. Your code, your test cases that will demonstrate what your code can do and instructions on how we can run your code. We need you to write a `Makefile` that will generate executables `compress` and `uncompress` and take in input output arguments. Your code must be heavily commented.