Programming Assignment 3: Huffman Compress/Decompress and Code Refactoring

Checkpoint deadline: **Wednesday, February 15, 11:59 PM**

Final deadline: **Wednesday, February 22, 11:59 PM**

Assignment Overview

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 using ASCII
- Extend the basic I/O functionality of C++ to include bitwise operations
- Refactor code that you have written to make it more readable or to otherwise improve the design

Set Up

- As done in earlier assignments, follow the instructions provided in this PPT or video to get started with the assignment.

Provided files:
HCNode.h, HCTree.h, Makefile, refcompress, refuncompress, .gitignore.

NOTES (IMPORTANT):

- You should NOT push the input test files (they are large).
- We have provided a .gitignore file that won't allow you to add these files to your repo.
- If there is some specific file you want to add that is not allowed by the .gitignore file, simply open the .gitignore file in a text editor and add a new line at the end with the syntax "!fileName" where fileName

is the name of the file you want to add.

- **Please do not use any other executable names than the ones in your Makefile. Our grading scripts depend on your executables being named "compress" and "uncompress" exactly.**

Checkpoint Instructions

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

1. Implement HCNode and HCTree

The HCNode and HCTree implementations will help you create Huffman tree/code for the input files. For now, they will support encoding and decoding with ASCII '1's and '0's only.
There are suggested methods in the `HCTree.h` header file that you might find useful specifically for this checkpoint, but that you should NOT use for the final submission.

Implementation Checklist:

- Implement the `HCTree.h` methods in a new file `HCTree.cpp`. You can modify both files in any way you want.
- Implement the `HCNode.h` methods (overloaded operator) in a new file `HCNode.cpp`. You can modify both files in any way you want.
- 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.
- Add `HCTree.cpp` and `HCNode.cpp` (and all other files you create that are required for the submission) into your repo

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.

---

2. Implement Compression

Create compress.cpp (from scratch!!) 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:
1. Read the contents of the file named (`infile`).
2. Construct a Huffman code for the contents of that file
3. 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` is guaranteed 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 the checkpoint. See "the file header demystified" and "designing your header" 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. For the checkpoint 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" `outfile` will actually be larger than the original! 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" section in the PA below**

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.

Designing your header: A straightforward, non-optimized method that you MUST USE for the CHECKPOINT

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 integers at the beginning of the compressed file. For your checkpoint 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.

---

3. Implement Uncompress

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:
1. 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
2. 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" section in the PA below**

---

4. Writeup

Verify your compression in Checkpoint.pdf. 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: **checkpoint1.txt and checkpoint2.txt**.
2. Record the encoded output in Checkpoint.pdf
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 Checkpoint.pdf.
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. If you never encountered this situation, mention it in your writeup.
5. We expect the writeup to be 1-3 pages

Final Submission

1. 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:

- Implement Bitwise I/O (See below for details)
  - Implement BitInputStream
  - Implement BitOutputStream
- 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" 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), 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 when run on two particular files, including PA3/input_files/warandpeace.txt (and one others we won't tell you about in advance).
- 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 (Checkpoint.pdf).

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 (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. Refer to the lecture notes for more information.

Efficient header design

The reference solution header is not very efficient. It uses 4-byte ints 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.**

---

2. Code Refactoring

Your final task for this assignment has nothing to do with Huffman coding. You will refactor code that you have written on a previous assignment to make it better.

You will take your DictionaryTrie implementation (in DictionaryTrie.cpp) and try to improve its design by a non-trivial refactor. For example you could refactor predictCompletions or insert or find or any other method that you feel has been designed incorrectly. You could also refactor the whole class. We are not expecting you to improve your code in terms of space or time complexity. Instead we want you to focus on the design of this code. If you feel that your there is no flaw in your design of DictionaryTrie implementation, then you must justify the effectiveness of your design in the writeup. However, even well-written code can usually be improved, so be very wary of thinking that your code needs no improvements.

You will be graded on (1) your final design, (2) how much you improved on the original design, and (3) how well we are able to get this information out of your write up.

Its not about how much you have refactored. Its all about how well you have refactored your code.

Implementation Checklist:

1. Rewrite/redesign your selected code to improve its style. You should include as much as needs to be redesigned from the code you selected (e.g. this could be multiple methods, the classes themselves, etc).
2. Describe your improvements and how they improve your code in a PDF file named Refactor.pdf. We expect this description to be formatted so we can easily understand what you did and why, and to be about 1-2 pages.
3. Place Refactor.pdf and DictionaryTrie.cpp (containing the refactored code) in the folder named refactoredCode. You will have to create this directory manually. Make sure you explicitly add these files to the repo and commit them. You should include your DictionaryTrie.cpp code even if you believe that no changes were necessary.

If you worked with a different partner for PA2 than you are working with for PA3, you need only to refactor ONE of your two implementations from PA2.

Collaboration policy specific to the Refactor portion of PA2

We encourage collaboration with other students (beyond your partner) for this section according to the following guidelines:

1. You MAY look at others' code only in their presence to provide them with advice and/or to get ideas for how to improve your design. You MAY allow others to look at your code only in your presence. You MAY discuss their code or your code verbally. However, during this process you MAY NOT make any written or digital records. This includes:
    1. You MAY NOT copy any portion of anyone else's code, either by hand, with a computer, or with a photo. You may not allow anyone else to copy your code in any way.
    2. You MAY NOT look at anyone else's code unless they are physically present with you. Others MAY NOT look at your code unless you are physically present with them.

3. You MAY NOT take any notes during this conversation. Sorry we have to do this--we'd like to be able to let you take notes, but there's no clear line of what's allowed and what's not, so we'll just have to say no notes at all.
4. You MAY NOT record your meeting or discussion in any way.

Basically, the idea is that we want you to be able to give each other ideas, while ensuring that you (together with your partner) complete the code on your own. If you have ANY questions or you do not understand the restrictions of this collaboration in ANY way, you must come talk to the instructor.

Testing Tips

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! 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.

Edge Cases Checklist:

- Empty File
- Files that contain only one character repeated many times
- Think of more edge cases yourself!

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.

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

You can use the "reference" implementations to verify your results.
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.
Note: 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 machines(NOT in the starter code provided to you in your github repo). The path for these input files is:

```
/home/linux/ieng6/cs100w/public/pa3_input_files
```

To copy these files to your folder, just use

```
cp -r /home/linux/ieng6/cs100w/public/pa3_input_files <path_to_your_pa3_home>
```

Submission Scripts:

we have also added the submission scripts for the checkpoint/final milestones in vocareum.

You can look at them by going into the resource dir in the vocareum terminal as follows.

```
cd ../resource
cd scripts/submit.sh
```

All the supporting files for running the submission scripts are present in

```
cd asnlib/
```

This might help you debug your issues with submissions for checkpoint/final.
A guide on how to use submission scripts is here

Grading Overview

This assignment is out of 30 points. There are 5 points for the Checkpoint, and 25 points for the final submission.

Checkpoint grading (5 points)

- Checkpoint.pdf: 2 points
- Compress and Uncompress.cpp code correctness: 3 points

Final Submission grading (25 points)

- Compress and Uncompress code correctness: 16 points
- Beat our reference solution: 3 points
- Commenting and style (see the minimal style guide): 3 points
- Code refactor: 3 points

If your solution files do not compile and link error-free, you will receive 0 points for correctness portion of this assignment. We will compile your files as in the Makefile distributed with the assignment.

Memory leaks will result in deductions to your points. If your code results in a segfault for a specific test case, you will receive 0 points for that test case. If your solution files do not compile and link error-free, you will receive 0 points for the programming portion of this assignment.

Required solution files for checkpoint submission:

Makefile, compress.cpp, uncompress.cpp, HCNode.h, HCNode.cpp, HCTree.h, HCTree.cpp, Checkpoint.pdf

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, refactored code file DictionaryTrie.cpp along with Refactor.pdf in the folder named refactoredCode

---

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 just like the checkpoint and final submissions. Your code must be heavily commented.

You will be graded on the i) correctness of your implementation ii) Style iii) the quality of your report in terms of pseudo code description (can we understand what you did and why?), discussion of results (discussing unexpected behavior), and the writing (lazy writing will hurt your chances in earning the star point).The better and more thorough and clear starpoint.pdf is , the more likely are you to earn the star point.