CSE 100 // **PA3**

**Huffman Compress/Decompress**

Checkpoint deadline **Friday, May 18 @ 11:59pm →**  (**No late days** allowed for checkpoint)

Final submission deadline **Friday, May 25 @ 11:59pm**

# **Assignment Overview**

Starter code: [download here](https://drive.google.com/a/eng.ucsd.edu/file/d/1SJfK9cZp_cxqb2xw5Q1J8tF84XA3ADYN/view?usp=sharing)

# In this assignment you will:

* Implement Huffman's algorithm using efficient supporting data structures to support encoding and decoding of files
* Extend the basic I/O functionality of C++ to include bitwise operations

## **Starter Code files**

HCNode.h, HCTree.h, Makefile, refcompress, refuncompress

[What are refcompress & refuncompress?](#_97cztf772dbj)

# **IMPORTANT NOTES**

* **DO NOT HARDCODE ANY TEST CASE. You’ll get a 0 (zero) on this PA if you do.**
* **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. (Except for extra credit)

# 

# **CHECKPOINT** [15 points]

10 out of 15 points are for compress/uncompress correctness and will autograded.

The rest 5 points are given for correct compressed file format and will be manually graded.

To implement Huffman's Algorithm, use 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.
* **How to compile:** 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.

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:

> make compress

Tip: if you do not want to change your Makefile, you need to have the following files in order to compile:

BitInputStream.h HCNode.h compress.cpp

BitInputStream.cpp HCNode.cpp

BitOutputStream.h HCTree.h

BitOutputStream.cpp HCTree.cpp

You may create any of these classes as “dummy” files for now.

> ./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 your input file for reading. (for checkpoint, infile is guaranteed to have only ASCII text and to be very small (<1KB)). You may use any input file you found/created.
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](#_kzu9ssgbxr)" and "[designing your header](#_1irza15b0vqe)" for more details.
6. Open your 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**](#_wd2vhkvlcmw9)**" 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. uncompress.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**

## **Files to submit (checkpoint):**

Makefile compress.cpp HCNode.h HCTree.h

uncompress.cpp HCNode.cpp HCTree.cpp

Also submit any dummy files you created to compile ./compress and ./uncompress (if you did not change Makefile). Make sure that calling “make compress” and “make uncompress” on vocareum terminal does not cause any error (because that’s how we compile and test your files).

# **Final Submission** [85 points]

All 85 points are autograded.

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](#_h7ism1whwdd)" 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**.

## **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. Implement your encoding/decoding methods in BitInputStream.cpp and BitOutputStream.cpp.

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.

## **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.**

## **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](https://docs.google.com/document/d/1A3KJF6LTU3z1cLTcMg_lpAaHqKVHA5D2_3ET5KN3sz8/edit?usp=sharing) 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 and vocareum. If you attempt to run it on a different architecture they will most likely not work.

## **Grading Overview**

This assignment is out of 100 points. There are 15 points for the Checkpoint, and 85 points for the final submission.

**Checkpoint grading (15 points)**

* Compress and Uncompress.cpp code correctness: 10 points (autograded)
* File looks compressed (correct compressed file format): 5 points (manually graded)

**Final Submission grading (85 points)**

* Compress and Uncompress code correctness: 77 points (autograded)
* Beat our reference solution: 8 points (autograded)

**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.

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 this assignment.

## **Files to submit (final submission):**

Makefile compress.cpp HCNode.h BitInputStream.h

uncompress.cpp HCNode.cpp BitInputStream.cpp

HCTree.h BitOutputStream.h

HCTree.cpp BitOutputStream.cpp

Make sure that calling “make compress” and “make uncompress” on vocareum terminal does not cause any error (because that’s how we compile and test your files).

**Extra Credit** [10 points]

Extra credit is NOT autograded.

There are two purely optional components of this assignment that are independently worth 10 points of extra credit. **It is sufficient if you complete ONE (1) of the below two problems.** (Extra credit is capped at 10 points.)

**1. Implement** [**Adaptive Huffman Encoding described here**](https://www2.cs.duke.edu/csed/curious/compression/adaptivehuff.html)

**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.

**Note**: As you can possibly tell, extra credit is getting more challenging as the quarter goes on. Extra credit problems are difficult - and they are designed to be difficult. The goal of having extra credit problems is to encourage self-exploring and researching - it is challenging, but also very fun. Most tutors have not done these extra credit problems before, so there may be limited support in the lab for help with extra credit problems. Good luck...

**We require you to turn in**   
  
1. A writeup named extracredit.pdf that explains:

* What you did?
* Output and Description of test cases (Showing your code works). You must test them on test files that are >500KB.
* 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 edit the Makefile that will generate executables eccompress and ecuncompress and take in input output arguments just like compress and uncompress in the checkpoint and final submissions. **Your code must be heavily commented.**  
You will be graded on the i) correctness of your implementation ii) 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 full extra credits).The better and more thorough and clear extracredit.pdf is, the more likely are you to earn full extra credits.