CSE 100 PA 2: Binary Search Trees

This document will guide you through the assignment from high-level concept down instead of from each class function up, as was done in PA 1. You will implement file compression and decompression using Huffman's Algorithm, efficient data structures, and bitwise input/output operations. Your program must accomplish effective space management by having comparable results to our reference solution. Included in this writeup are some design notes in part 6 which you need to follow. As always, start early.

Provided Starter Files: HCNode.hpp, HCTree.hpp, Makefile, refcompress, refuncompress, testInput (folder)

Checkpoint Submission [NO SLIP DAY]

Due: Tuesday, August 9th, 2016, 11:59PM

Required Submission Files: (at least) compress.cpp, uncompress.cpp, HCNode.hpp, HCNode.cpp, HCTree.hpp, HCTree.cpp, Checkpoint.pdf

Final Submission [1 Slip Day Allowed]

Due: Saturday, August 13th, 2016, 11:59PM

Required Submission Files: BitInputStream.hpp, BitInputStream.cpp, BitOutputStream.hpp, BitOutputStream.cpp, compress.cpp, uncompress.cpp, HCNode.hpp, HCNode.cpp, HCTree.hpp, HCTree.cpp

Table of Contents:

Part 1: Getting Started

The starter files are located in `/home/linux/ieng6/cs100v/public/pa2`

Copy these files onto your own folder in your account. The test input files are in a folder in the pa2 folder. Make sure those get copied over as well.

If you are unable to use C++ 4.8.3 a.k.a. -std=c++11, you will need to use the command `prep cs100v`

Part 2: File Compression and Decompression

The assignment is basically writing two C++ programs: compress and uncompress. When compress is executed by the following:

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

This program will read the contents of the file as named by the first command line argument (infile), construct a Huffman code for the contents of that file, and use that code to construct a compressed version of the input file to be saved as named by the second command line argument (outfile).

The second program, uncompress, is executed by the following:

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

This program will read the contents of the file as named by the first command line argument (infile). This file should have been constructed by the compress program as detailed previously. The contents of this file will be used to reconstruct the uncompressed, unchanged original file, which will be saved to a file as named by the second command line argument (outfile).

To be clear, for any files <**original**>, <**compressed**>, and <**uncompressed**>, after running "./compress <**original**> <**compressed**>" and then "./uncompress <**compressed**> <**uncompressed**>", the two files <**original**> and <**uncompressed**> must be identical files with absolutely no differences. A quick way to check if two files are identical are to use the `diff` command. Doing `diff` <**original**> <**uncompressed**> will return nothing if the two files are identical, but will print out where they differ if there are any differences between the two files. This can be used to ensure that your ./uncompress is working properly.

Part 3: Checkpoint Submission

The purpose of the checkpoint is to focus on implementing the Huffman algorithm correctly. The "compressed" file will actually not be smaller than the original file because this portion of the assignment will have you compress using ASCII I/O rather than bit I/O.

Your checkpoint submission's 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 guaranteed to have only ASCII text and to be very small (<1KB)). Then, write out the tree and the coded infile as plain (ASCII) text to outfile. The outfile should be a readable text file that contains your representation of the tree, as well as the code that represents the original file. So, the 1's and 0's in your code will be written to the "compressed" file using the ASCII characters for 1 and 0.
- "`./uncompress infile outfile`" should "uncompress" the infile by reading in the file header, reconstructing the tree, and then "uncompressing" (i.e. decoding) the rest of it and writing it back to the outfile, as plain (ASCII) text.

Checkpoint.pdf

You need to verify your compression. Use checkpoint1.txt and checkpoint2.txt to run through the compress and uncompress programs. Save the output in .txt. You need to document your verification process by doing each step in the writeup as follows.

For each input file (checkpoint1.txt and checkpoint2.txt):

1. Describe how you built the tree.
2. Describe how you find the codeword for each byte.
3. Draw the Huffman Tree by hand and include the result in the writeup.
4. Explain why your hand-coded text is different from your compressor output.
5. Explain how you fixed it.
6. If you had no difference, then ignore Steps 5 and 6, and state you had no differences.
7. Save your writeup in .pdf format. No credit will be given for any other format. There is a maximum of 5 pages (this can be done using various tools like LaTex, word conversion to pdf, or others).

You might find the suggestions in HCTree.hpp helpful. Be aware that those suggestions are only good for the checkpoint submission and will not work well for the final submission.

Since you are not required to submit BitInputStream and BitOutputStream for the checkpoint, you will need to solve compilation issues that result from this. You can (1) comment out all references to them from the provided files or (2) create "dummy" versions of these .hpp and .cpp files to get the code to compile. You will also need to edit the Makefile (commenting would be the safest). When you work on the final submission, restore the references to the Bit Stream classes.

Part 4: Final Submission

This is the part where your program will use bitwise input/output to see results in compressed encoded files. The functionality is the same as the checkpoint functionality with the following exceptions. Your program:

1. Must be able to handle files much larger than 1KB, up to 10MB. This means a particular byte value may occur up to 10 million times in one file.
2. Must be able to handle file formats other than text files, such as binary files, images, videos, and more.
3. Must create compressed files that are at least 1 byte smaller than the input file AND the file produced by our reference solution.
4. Must use bitwise input/output to read and write the Huffman codes.
5. Should finish in about 15 seconds, but the hard requirement is that it must finish in under 180 seconds.

Part 5: Control Flow

Here are the high-level steps your programs need to go through.

Compress:

1. Open the input file for reading.
2. Read the file byte-by-byte. 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.
6. Open the input file for reading again.
7. Using the huffman coding tree, translate each byte from the input file into its respective Huffman code. Append these codes as a sequence of bits to the output file after the header. This is done in ASCII for the checkpoint and in bitwise I/O for the final submission.
8. Close the input and output files.

Uncompress:

1. Open the input file for reading.
2. Read the file header at the beginning of the input file. Reconstruct the same Huffman coding tree that compress created.
3. Open the output file for writing.
4. Using the reconstructed Huffman tree, decode the bits from the input file into the appropriate sequence of bytes, saving them into the output file.
5. Close the input and output files.

Reference solution: refcompress/refuncompress

The programs refcompress and refuncompress are "reference" implementations of this assignment you will use to verify your results. The reference programs are likely to work on

ieng6 only. Your compression must beat the reference compressions by at least one byte for the final submission.

Your compress program is not expected to work with the reference uncompress, and your uncompress is not expected to work with the reference compress.

Part 6: Design

(1) **Implementing the Tree and its Nodes.** 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 provides a possible interface for this structure (included in your repo as skeleton code). **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 submission (both at both the checkpoint and the final).

(2) **Object-Oriented Design.** In implementing Huffman's algorithm, you will find it convenient to use multiple data structures. For example, a priority queue will assist in building the Huffman tree. Feel free to utilize other beneficial data structures. 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 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. Thinking about object-oriented design will help you with the following two PAs.

(3) **The header and its use.** Under the section on "Control Flow" you saw references to a header that should be stored by the compress program and later retrieved by the uncompress program. Both the compress and uncompress programs need to construct the Huffman Tree before they can successfully encode and decode information, respectively (and in the same way). 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.

(4) Designing the 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.

(5) Efficient header design(for the FINAL SUBMISSION)

The method described above is extremely inefficient. Even if we improve this approach to store frequency counts as 4-byte ints, storing the frequency counts in your header is not very efficient in terms of space: it uses 1024 bytes of disk for the header if you write your header as ints (not ASCII text), no matter what the statistics of the input file are. This is the approach we use in our reference solution.

For your final submission, you must BEAT our reference solution by coming up with a more efficient way to represent this header. One way to do this is to take a frequency approach, but to think about how many bytes you really need to store each integer. Answering this question relates to the maximum size of the files that you are required to encode. To help you with your header design, please note the following: 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. This fact should help you determine the minimum number of bytes required to represent each frequency count in you header.

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 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 (256-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.***

(6) Note that there are some differences between the requirements of this assignment and the description of Huffman encoding 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.)

(7) **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.

(8) **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.

(9) **Separating Interface from Implementation.** One difference from the assignment (BST) is the introduction of an .hpp file. Unlike other languages, such as Java, that allow for interfaces to be created, C++ does not. Therefore, the closest option a programmer has for decoupling the interface from the actual implementation is the use of header files. The header files contain just the prototypes for the functions and the class. Usually no actual definitions for functions exist in the header files (although there is some in files given to you). The actual implementation is placed inside the cpp files. In general, this is

the convention used by many C++ programmers. C++ templates are sometimes an exception and this has to do with compiler challenges.

Part 7: 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! 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 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 the input_files/ directory in your repo that may be useful as test cases, but you will want to test on more than these.

For some useful testing tools, refer to this doc.

Part 8: Submitting on Vocareum for Checkpoint and Final

Checkpoint Submission [NO SLIP DAY]

Due: Tuesday, August 9th, 2016, 11:59PM

Required Submission Files: (at least) compress.cpp, uncompress.cpp, HCNode.hpp, HCNode.cpp, HCTree.hpp, HCTree.cpp, Checkpoint.pdf

Final Submission [1 Slip Day Allowed]

Due: Saturday, August 13th, 2016, 11:59PM

Required Submission Files: BitInputStream.hpp, BitInputStream.cpp, BitOutputStream.hpp, BitOutputStream.cpp, compress.cpp, uncompress.cpp, HCNode.hpp, HCNode.cpp, HCTree.hpp, HCTree.cpp

---

Choose the files you want to submit. A script will run. **Make sure** that your submission at least:

1. Has all the correct names
2. Compiles
3. Runs without issues with the tester

Fix any errors that arise from Steps 1-3.

Only one of the programmers in a partnership need to submit, but both should verify that the submission works for both programmers.

You can submit once, ten times, any number of times before the deadline and only your last submission will be graded. If you submit within 24 hours after the deadline, you will be charged a slip day. Slip days cannot stack, so there will be no submissions allowed after 24 hours.

Try to save your slip days for the remaining two PAs…

Part 9: Grading Guidelines

This PA is worth 30 points.

The checkpoint is worth 5 points:

- 3 points for working code
- 2 points for writeup

The final submission is worth 25 points:

1. 20 points for compress and uncompress, using Huffman coding to correctly compress and uncompress files of all sizes. Your program will be tested on 10 files. For each file, one point is given if your compressed file is within 25% of the compression obtained by the reference implementation, and one point for full reconstruction with uncompress.
2. 3 points for correct implementation of compress that creates a **smaller** compressed file than the reference implementation when run on three files. One of the files is warandpeace.txt. We won't tell you what the other two files are. :)
3. 2 points for good object-oriented design, style, and informative, well-written comments.

Memory leaks will result in deductions. Each test case that results in a segfault will receive a 0. Compile and link errors will cause the entire assignment to receive a 0. Valgrind and GDB are highly recommended debugging tools to use.