UMass Boston CS 444
Project 1
Posted Monday, February 16, 2026
Due Saturday, February 28 at 11:59 pm

J.H. DeBlois

1 Project Description

1. Overview. For proj1, write two C programs that use Huffman file compression. The first one is relatively simple, as explained below. The second one implements the full Huffman algorithm to compress a given input file.

2. Naming. In this document, let `abc` stand for your initials with length two or more characters. Name your code `abchuff.c` and `abchuffman.c`. Then name your executables `abchuff` and `abchuffman`. Name your proj1 description file `abcReadMe.txt`.

3. Goal. At the end of proj1, using the input file, `completeShakespeare.txt`, you will compare the compressed output of your huffman code to the compressed output of your instructor's huffman code. Your output file will be named `abchuffman.out`. The instructor's output file is named `jhdhuffman.out`. Since there is an optimal Huffman code for any file as long as there are no duplicate frequency counts, you will be able to run the diff command on the two compressed .out files to show whether they match.

4. How to build `abchuff`.

   - Create your `abchuff.c` file in your `proj1` subdirectory. Include: 1) a C code header comment with filename and author, 2) the standard input/output library #include line, 3) the basic `int main(){...}` function and 4) a printf statement of your choosing. Compile and run.
   - Select 3 ASCII characters from the ASCII code chart. Create a printf statement in main to display each character in ASCII, hex and binary. Then write a short message (7-10 characters) as a string using only your 3 ASCII characters. Let the frequency of one of the characters be 1. Then vary the numbers of the other two. Write printf statement 2 to display the message. Compile and run.
   - Since Huffman codes are based on counts of the ASCII characters in the specific file to be compressed, you can now add printf statement 3 that gives the count of each of your 3 characters in your message. You do not have to do the counting in your code, just write a statement tellng the frequencies of your three characters in your message string.
   - By hand, draw your Huffman tree and determine the Huffman codes for each character. Handwrite each Huffman code right-justified in an 8-bit byte, shown in binary. Also write the precise number of bits in each Huffman code. Add printf statement 4 in main to print out the 3 ASCII characters with the byte containing its code and the number of digits in the code.

- Create a buffer in your code as a string. For each character in your message (reading left to right), get its code and the count of its number of digits, put that many binary digits into the string and keep going until all the codes are entered. Keep a count of bits added. When you have completed entering all the codes, add zeroes to pad to a full byte. Add printf statement 5 to print out this string of binary digits, the number of bits and the number of bytes.
- Copy the string to a new buffer. Use the bit operations from the Huffman slides to isolate the first (leftmost) byte of the coded string. For instance, shift right until all bits except the last 8 bits roll off the right end. Then use binary fwrite to output the byte to file abchuff.out. Use `hexdump -C abchuff.out` to display the byte as hexadecimal. Make sure it is correct. Then work again using another copy of the complete string. Use bit shift left and right and/or masking and/or anding to isolate the second byte to write out. Continue until all the bytes have been written out. Use `hexdump -C abchuff.out` to check the full output.
- At this point, everyone's code is highly similar, although the data varies. We would expect MOSS numbers above the limit of maximum 45

5. How to Build `abchuffman`. Your code must include getopt.h and command line switches. Use the example in program2.c that allows a switch to accept an argument, so the data (for example, a file name) is available in your program. The executable must accept the following command-line options and default files names:

   - `-i filename` for input file 1 — is `completeShakespeare.txt` which is default input filename
   - `-o filename` for output file 1 — is `huffshake.out` which is default output filename

   The instructor's executable is `/home/hdeblois/cs444/proj1/jhdhuffman` on the CS Linux servers. Go there and use the "cp" command to copy it to your course directory. The ASCII files `noDupFreq.txt` and `completeShakespeare.txt` are also in the instructor's `proj1` directory. Copy those files, too. You will need the exact file name to create a default input file in your code.

   At the end of building code for all the increments, assuming your initials are 'abc', you can test your huffman algorithm via these commands:

   ```
   ./abchuffman -i completeShakespeare.txt -o huffshake.out
   ./jhdhuffman -i completeShakespeare -o tmp.out
   diff huffshake.out tmp.out
   ```

   If your code works correctly, `diff` will find no difference between the two output files. If there is an error, check your codes using -d switch on both.

2 Project Details

Before start this section, please review the Syllabus for the course and the Huffman slides.

In the `cs444` folder under your `courses` directory on the CS server, create a folder called `proj1`, and put all your files in there, including C code, executables and a `readMe.txt`. Spell the names carefully and exactly as given because we will use a script to collect your work.

For your C code names, prefix your initials to your huff.c and your huffman.c.

Write your C code on the server using a text editor, e.g., `nano`, `vi` or `emacs`. Compile and run on the CS server each time you work on it.

Put the name of your file and your name as author in the code in a C code comment at the top. Include tags: `filename: abchuff.c or abchuffman.c`, `author: <your name>`.

If, in addition, you compile your code on your laptop, be aware that the C libraries on the server may not be the same as on your laptop so you must upload your code via scp and recompile via gcc on the server.

For command-line options, you are required to include the `getopt.h` library and use it to implement the options. See program2.c for a sample implementation. Each command line option can be a switch or a switch and data.

As variables, include a debug switch `-d` (without a variable) to signal your code to set `int debug = 1` to turn on `printf` statements that start with the phrase `if debug`. You could use multiple debug options as switch letters d, e, f, etc. if you want to vary the output you see.

We follow the four-step implementation described on slide 27 of the `huffman.pdf` slides but we focus on steps 1 and 4 more at first to write `abchuff.c`:

1. Read the input file and use a 256 element array to count the frequency of each character. Turn debug on to print out frequencies.

2. First, use the frequencies to build a forest of tree nodes for characters that have non-zero counts. Second, sort by frequency and build the huffman tree by combining the two nodes with lowest frequency into a new node with those nodes as children, resorting and repeating. Turn debug on to print out the total combined frequency, which will match the size of the input file.

3. Traverse the tree to collect the code for each leaf by length and bit sequence. Turn debug on to print out each character with its code.

4. Read the file again, code each character, put the bits in a buffer keeping track of how many and output a byte using `fwrite` when the buffer has eight bits, saving the rest for the next byte if necessary.

You can run the instructor-supplied executable with various command line options. Set debug=1 to see the huffman codes of any input file: type ./jhdhuffman -d 1 -i noDupFreq.txt and see the huffman codes created for that input file. To see ASCII of a file in hex, type hexdump -C noDupFreq.txt.

Prepare a `readMe.txt` to list how you developed your code in increments. For each time you work, make an entry in your readMe.txt file with date and what you worked on. Keep incremental copies of your code on the server. For example, at the start of work, copy your code to abchuffman_oct1.c and recompile. Code will be collected from the server often. Be sure to enter your full name at the start of the readMe.txt file and in the main comment at the top of your code.

If you consult and/or use (small features of) code found on the internet or generated in chatgpt or other LLM, you are required to 1) list each source in your readMe.txt and 2) for those lines of code you copy in and use or modify, cite the source in your code before the start line and place an end marker after the last line.

Follow the format in the MIT integrity guide referred to in the syllabus.

For generated code, give the LLM and prompt you used verbatim (although rerunning the prompt may not give the same result).

Do not cite fellow student's code because it is not a published source and you are not supposed to share your code. Cite any code given you by the instructor.

In grading, we run Stanford's Measure of Software Similarity (MOSS) to check that no one duplicates someone else's code. The maximum similarity for your code to another student's code is 45 per cent. Too high a level of similarity on abchuffman gets you zero on proj1. We will give you a second round of practice by running MOSS on your first increment of abchuffman.

Note: If you cite each part of the code you bring in, you will not have an integrity issue, but you could still be too similar to another student's code by virtue of having copied too much from the internet and not being creative enough in adding your own illustrative features.

3 Grading Rubric

Be sure to "make your code your own" by reading it carefully and taking time to comment it. You may be required to come in and explain how your code works.

There are 8 steps in grading: 1) we check whether your submission matches the technical requirements for your files including incremental delivery, 2) we run your submitted executable(s), 3) we copy your submitted code files to another location, recompile and compare it to your submitted executable(s), checking that they match, 4) we check your `readMe.txt` file, 5) we read your code to check citations and formatting requirements, 6) we submit your code to MOSS, 7) we ask you for a code walkthru if needed, and 8) we verify the outputs of your code.

Be sure to notice how the uppercase and lowercase letters are used in naming the files and directories that you are required to create `abchuff`, `abchuffman`, `readMe.txt`). Names must match exactly or your submission will not be collected.

The late penalty applies to any code submitted after 11:59pm on the duedate. See syllabus for late penalty.

Your `abchuff` is due one week before the project duedate so we can see an early MOSS result. Your first increment of `abchuffman` is due 48 hours ahead of the duedate. We will run MOSS again then. Please submit your work incrementally, compile on the server and copy each prior increment to a different name such as `abchuffman_oct1` before creating your next increment `abchuffman`.

1. (10 points) `readMe.txt` contains full name, list of what was worked on in each increment and list of sources consulted or statement that you did not consult outside sources of code. Code cited in your .c files would be a subset of this list.

2. (20 points) `abchuff.c` compiles, runs and outputs initial compressed file for your three characters with various repeats that makes up your message. We will grade you on `abchuff` when you turn it, but you may update it for this final delivery.

3. (20 points) `abchuffman` code includes getopt.h, default values and switches to enable all combinations of command-line arguments specified above, code compiles and each combination runs. See supplied file `program2.c` for a working example of how to use getop, which you may include in your `abchuffman` and modify.

4. (10 points) code is indented (2 spaces per line is preferred but consistent size of indent is required); there is a header comment at the top of each program with filename and author; names used in your code well-chosen; and citing is specific including end markers for each citation.

5. (5 points) you use bit manipulation correctly in your code to add variable number of code bits for each character to your buffer(s) and write them via fwrite to your output buffer.

6. (5 points) `abchuffman.c` has a getopt switch for debug that displays huffman codes for the selected input

7. (10 points) `abchuffman.c` has a getopt switch for displaying an original output that explains something you would like us to see – output says what it is, why you chose it and displays it

8. (5 points) correct output for `noDupFreq.txt`

9. (15 points) correct output for `completeShakespeare.txt`