UMass Boston CS 444
Project 1
Posted Tuesday, September 30, 2025
Now due Wednesday, October 22 at 11:59 pm
REPOSTED, typo in Grading fixed - it's abchuffman.c, not proj1.c

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 uses the full Huffman algorithm. In this document, let `abc` stand for your initials with length two or more characters. Name your executables `abchuff` and `abchuffman`. At the end of proj1, you will compare the compressed output of your huffman code named `abchuffman.out` to your instructor's compressed output specifically named `jhdhuffman.out`.

2. Executable compiled from `abchuff.c`. This program gets you started, uses an embedded text file of your choice that has a few bytes, only codes the characters for the two least frequently used characters in your file, uses bit manipulation on the output buffer (commands introduced in huffman.pdf slides) and the fwrite command to output the compressed file.

3. Executable compiled from `abchuffman.c`. This program implements the huffman algorithm fully, so the input file is compressed and written outto a .out file using fwrite.

4. Incremental delivery. You will deliver your work in increments - increments of `abchuff` are due in the first week, increments of `abchuffman` are due starting 48 hours before the project is due. An increment consists of your .c file placed in your course directory on the CS servers and compiled and run to give a .out file.

5. How to build `abchuff`.

   - Select your own mini file of ASCII characters and put them in your code. Since Huffman codes are based on counting the ASCII characters in the specific file to be compressed, enter your counts in your code. Since Huffman codes are determined by looking at the sorted non-zero frequencies of the characters in the file, enter your array of characters and frequencies in your code. Since Huffman tree construction builds a forest of leaf nodes including each character and its frequency and then combines them by repeatedly selecting the two lowest frequency subtrees, enter the total number of leaf nodes and total frequency in your code.

- For your embedded text file, select the first two nodes to combine based on the two lowest frequency counts. If there are duplicates, choose at random. Insert a printf statement to output the two characters and two frequencies that you will combine.
- To continue Huffman, with two frequencies removed, you would insert their combined count into the list of frequencies and resort. This would make a second forest of nodes that is smaller in node count but one node has two sub-nodes. Now you are to assign the last digit of the longest codes for the two lowest frequency characters. The lower frequency character gets code '0'. All your characters keep their ASCII code except two. Create an array listing huffman codes for characters with new codes. Put the character and code into the array for the two that you have figured out.
- Now create your output buffer with at least two bytes. Read your data again, character by character, and put the code for each character into the output buffer. When you come across the two least frequency characters, you only put one bit into the buffer. Keep a count of bits in the two-byte, 16 bit buffer. Shift left 8 or 1 bits before adding the next code.
- When the buffer has 8 or more bits, output the rightmost 8 bits as a byte using fwrite. Aim to output your mini input 'file' with this compression — the bytes for these two ASCII characters will be encoded by the binary codes '0' and '1'. Assign '0' to the lowest frequency node. Together, this saves 14 bits each time one of these characters appears! Last step, pad the last byte of the output file with binary zeroes on the right because only two bits will be used. The first x bytes until the first low frequency character is encountered will be ASCII bytes.

6. 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
   - `-j filename` for input file 2 — is `noDupFreq.txt`
   - `-o filename` for output file 1 — is `huffshake.out` which is default output filename
   - `-p filename` for output file 2 — is `huffnodup.out`

   The instructor's executable is `/home/hdeblois/cs444/proj1/jhdhuffman` on the CS Linux servers. In class, you copied it to your course directory. The ASCII files `noDupFreq.txt` and `completeShakespeare.txt` are also in the instructor's `proj1` directory. Make sure your code has both default file names.

   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 (or type ./abchuffman)
./jhdhuffman -i completeShakespeare -o tmp.out (or type ./jhdhuffman -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

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 your name in the code in a comment at the top. 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, include the `getopt.h` library and use it to implement the options. As variables, include a debug switch `-d` to set `int debug = 1` to turn on `printf` statements. You could use multiple debug options as switch letters d, e, f, etc.

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 codes. 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 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. You do not have to 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 duplication gets zero. We will give you practice by running MOSS on your first increment. 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, which may have many similar solutions, and not been creative enough in adding your own illustrative features. That said, please do not pad your code with random lines of code.

3 Grading Rubric

Be sure to "make your code your own," by commenting it and by knowing exactly how it works. You may be required to come in and explain how your code works. Knowing the algorithm before you start coding allows you to code your own way.

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 it 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 script collects *.c files in your directory. Select the prefix `abc` once and use it in all code names. The name `abchuff` is for the starter increment that gets submitted to MOSS as a trial.

The late penalty applies to any code submitted after 11:59pm on the duedate. See syllabus for late penalty. Your `abchuff` is due two weeks 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. 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 the tiny ASCII text data you embedded and the huffman codes zero and one for the two highest frequency characters - give us a getopt switch to display your selected small ASCII text data – we use hexdump -C to see your compressed output.

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.

4. (10 points) code is indented (2 spaces per line is preferred but consistent size of indent is required), you have a starter comment at the top with your name and name of code, etc., 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 manage 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`