Assignment 3

cpe 357 Fall 2020

```
"I'd crawl over an acre of 'Visual This++' and 'Integrated Development
That' to get to gcc, Emacs, and gdb.  Thank you."
(By Vance Petree, Virginia Power)
```

— /usr/games/fortune

Due by 11:59:59pm, Monday, October 23rd.
This assignment is to be done individually.

Programs: `hencode` and `hdecode`

This assignment is to build a file compression tool, `hencode`, that will use Huffman codes to compress a given file, and a file decompression tool, `hdecode`, that will uncompress a file compressed with `hencode`.

Usage:
hencode infile [outfile]
hdecode [(infile | -) [outfile]]

`hencode` must:

- take a command line argument that is the name of the file to be compressed; and
- take a second, optional, command line argument that is the name of the outfile. (If this argument is missing, the output should go to stdout.)
- properly encode its input into a binary Huffman-coded file according to the algorithm below.

`hdecode` must:

- Take an optional command line argument that is the name of the file to be uncompressed. If this argument is missing, or if the input file name is "-", `hdecode` will take its input from standard input.
- Take a second, optional, command line argument that is the name of the output file. (If this argument is missing, the output should go to stdout.)
- Properly decode its input and restore the original data.

Both must:

- use only UNIX unbuffered IO (`read(2)` and `write(2)`) for reading and writing the files[1]; and
- share as much of the code base as is reasonable between the two programs; and
- demonstrate robust programming techniques including proper handling of errors; and
- adhere to a "reasonableness" standard with respect to performance.

See also the "Pesky Details" section below.

[1] That means go ahead and use stdio for error messages.

Huffman Codes

In 1952, David A. Huffman proposed a method for compressing files by generating a variable length encoding of characters based on the relative frequency with which each character occurs in a file[2]. This encoding is what is known as *prefixless code* because no letter encoding is a prefix of any other encoding. In practice, this means that the coding can be stored in a binary tree with the characters at the leaves. Each character's encoding can be determined by looking at the sequence of right or left child node transitions and representing those as either zero bits or one bits.

Generating the Tree

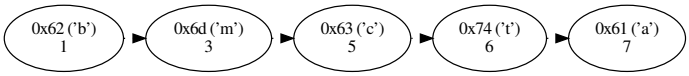
Note: Be sure to follow the tiebreaker conventions outlined in this document. If you do something different, you will still get a valid encoding of your file, but will be different from the reference one. That means that both programs will have to work perfectly all the way through in order to be tested. If you follow the tiebreakers partial credit becomes a possibility.

To build the tree, it is first necessary to generate a histogram of all the characters in the file, remembering that any value from 0 to 255 represents a valid byte. We will demonstrate the algorithm through the following encoding example.

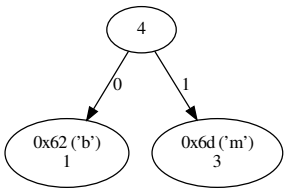
If the file you are encoding consists of the string "`bmmmccccctttttaaaaaaa`" the histogram will be:

Byte	Count
0x61 ('a')	7
0x62 ('b')	1
0x63 ('c')	5
0x6d ('m')	3
0x74 ('t')	6

Take these counts and generate a linked list of them in ascending order of frequency. If there is a tie in frequency, do a secondary sort and place the characters also in ascending order. For the histogram above, this list will look like:

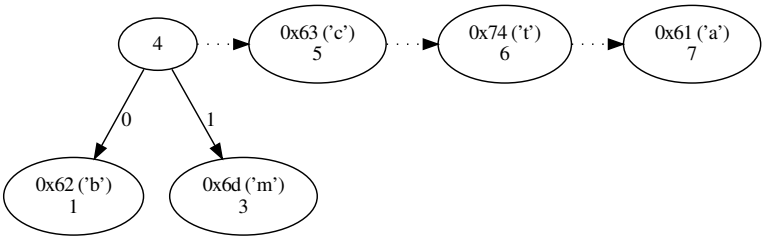


Now, remove the first two nodes on the list and construct a new node whose frequency is the sum of the two removed nodes. Make the first removed node the left child of the new node and the second removed node the right subchild:



[2]David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

Next, reinsert the new node into the remaining list of nodes in order. If there is a frequency tie, insert the new node before the other node in the list. After reinsertion of the new node, our list will look like:



Repeat the process until there is only one node left in the list. The rest of the process for this file is shown in Figure 1.

Now that you have the tree, to encode the file all that is necessary is to do a pass over the tree to extract the encodings into a table, then re-read the input file and translate each input character into the appropriate sequence of bits. If the file ends with a partially filled byte, pad the final byte with zeros.

The codes extracted from the tree above will be:

```
0x61 ('a'):  11
0x62 ('b'):  000
0x63 ('c'):  01
0x6d ('m'):  001
0x74 ('t'):  10
```

File Format

The output of `hencode` varies depending on whether or not it is encoding an empty file.

For empty files, the compressed version is an empty file.

For files with contents, the output format for the encoded file consists of two parts. First, comes a header that contains the frequency information necessary to re-create the tree, then the bits of the encoded file.

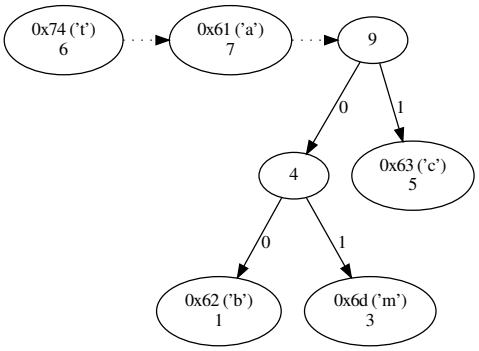
These pieces are illustrated in Figure 2, and explained below.

The first byte of the header consists of an unsigned one-byte integer containing the number of unique characters in the frequency table ("num", in Figure 2) minus 1. (Remember, not all files will contain all 256 possible bytes.)

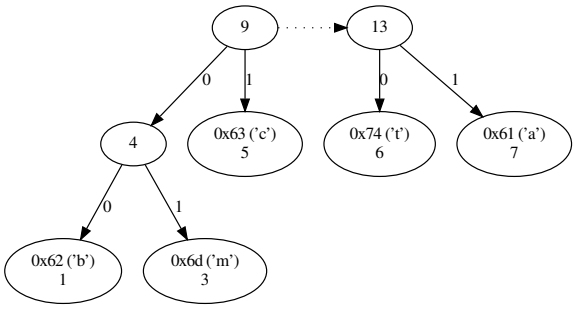
Why num − 1 rather than just num? A file can have anywhere between 0 and 256 unique bytes in it. This is one value more than an unsigned byte can hold. But we've already encoded empty files as empty files, so for non-empty files the real range is $1 \ldots 256$. This can't be fit in a byte, but $0 \ldots 255$ can. By subtracting one, we can get more compression, so this is what we will do.

Following this comes the frequency table, in alphabetical (numerical) order[3]. Each element of the frequency table consists of a single byte that it the byte itself, followed by a four-byte unsigned integer that is the frequency. For portability, multi-byte integers are to be encoded in *network byte order*. (See "Endianness", below for an explanation, but mostly this means using the two functions `htonl(3)` and `ntohl(3)` to convert.)

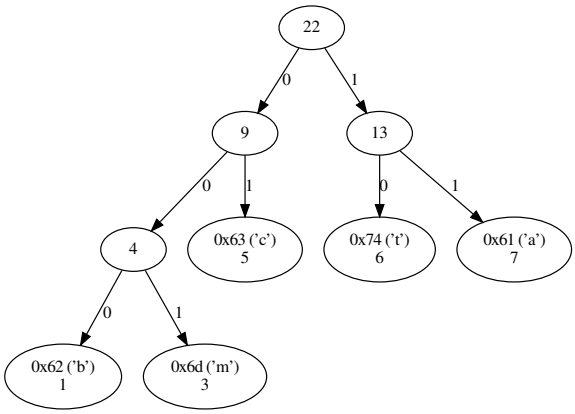
[3]This is not necessary for file compression, but it will make your output match the reference program's output.



(a) After a second pass



(b) After a third pass



(c) The final tree

Figure 1: Finishing tree generation

Num - 1	**c1**	**count of c1**	**c2**	**count of c2**	...	**cn**	**count of cn**
(uint8_t)	(uint8_t)	(uint32_t)	(uint8_t)	(uint32_t)	...	(uint8_t)	(uint32_t)
1 byte	1 byte	4 bytes	1 byte	4 bytes	...	1 byte	4 bytes

Figure 2: The file format used by `hencode` and `hdecode`

Figure 3 shows the resulting encoded file for the input above (for a little-endian machine) as hexadecimal bytes. Boxes are drawn around multi-byte data for clarity. Note that there is no padding in this particular example. Also, there is no separation between the header and the body, nor is there an end of file marker. These are all determined by counting.

	Num-1	**'a'**	**Count**				**'b'**	**Count**								
Header	04	61	00	00	00	07	62	00	00	00	01					
	'c'	**Count**				**'m'**	**Count**				**'t'**	**Count**				
	63	00	00	00	05	6d	00	00	00	03	74	00	00	00	06	
	File Contents															
Body	04	95	56	aa	bf	ff										

Figure 3: "bmmmccccttttttaaaaaaa" as encoded by hencode.

Encoding

To encode the file, build the tree, extract the encodings into a code table, then re-read the input generating the output as you go.

Decoding

To decode, read the header of the encoded file, regenerate the tree, then use the bits of the file to walk the tree to regenerate the file. Start at the root. For each zero bit, go left, for each one bit, go right. Because this is a prefixless code, you will know you have decoded a character when you reach a leaf. Output this character and start again at the root.

You know how many characters are in the output from the counts encoded in the frequency table.

Tricks and Tools

`od(1)` or `xxd(1)`	Useful for reading binary files to see what's what.
`open(2)` `close(2)` `read(2)` `write(2)` `lseek(2)`	The UNIX basic IO functions. Check out the `man` pages.
`ntohl(3)` `htonl(3)`	Convert a 32-bit integer in network byte order into host byte order and vice-versa.

Figure 4: Some potentially useful library functions

Some potentially useful library functions listed in Figure 4. Also, it might be helpful to know that `<stdint.h>` defines a set of fixed-width data types which are more portable than `int`s when you really need to know how big something is. These exist in signed and unsigned versions named `intXX_t` and `uintXX_t`, where XX is the number of bits. For example, `uint32_t size` makes `size` a 32-bit unsigned integer.

Coding Standards and Make

See the pages on coding standards and make on the cpe 357 class web page.

Pesky Details

Endianness

Before any discussion of reading low-level data structures, we must discuss the implications of byte-order. With a single byte, the meaning of an address is clear, but with multi-byte data, such as integers, the question arises, "Which end of the data does the address really point to?" The two obvious possibilities are the most significant byte or the least significant byte. Each is quite valid, but unfortunately, they are incompatable.

Consider the number $0xAABBCCDD$. If represented as a little-endian number at address A, the least significant byte, DD, comes at location A, then the more significant bytes follow at locations $A+1$, $A+2$ and $A+3$. For big-endian, the most significant byte, AA comes at address A, and the less significant bytes follow:

Little Endian

Address	+0	+1	+2	+3
	DD	*CC*	*BB*	*AA*

Big Endian

Address	+0	+1	+2	+3
	AA	*BB*	*CC*	*DD*

One can easily be mapped to the other by reversing the order of the bytes.

The reason you care is that not all machines have the same endianness, and a small number written on a little-endian machine suddently becomes huge if read on a big-endian one. For portability, we must decide on a standard for interchange[4]. That is network byte order.

Intel x86 machines are little-endian. Network byte-order is big-endian.

Bits and their order

- If it is necessary to pad the final byte of the file, pad it with zero bits.
- When filling bits, fill from the high order bits. That is, if the final four bits of an encoded file are 1010, the final byte of the file will be `0xA0`.

What to turn in

Submit via `handin` in the CSL to the `asgn3` directory of the `pn-cs357` account:

- your well-documented source files.

[4]This will also make reading your archives easier in a hex editor since you won't have to be mentally swapping bytes around.

- A makefile (called `Makefile`) that will build your programs when given the command "`make all`".
- A README file that contains:
  - Your name.
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

  The README file should be **plain text,** i.e, **not a Word document**, and should be named "README", all capitals with no extension.

Sample runs

Below are some sample runs of `hencode` and `hdecode`. I have placed a runable versions of `hencode` and `hdecode` in `~pn-cs357/demos` as `hencode`, and `hdecode`.

Here is an example of encoding the file used in the example above:

```
% htable testfile
0x61: 11
0x62: 000
0x63: 01
0x6d: 001
0x74: 10
% hencode testfile testfile.huff
% od -tx1 testfile.huff
0000000 04 61 00 00 00 07 62 00 00 00 01 63 00 00 00 05
0000020 6d 00 00 00 03 74 00 00 00 06 04 95 56 aa bf ff
0000040
% hdecode testfile.huff testfile.out
% diff testfile testfile.out
% ls -l testfile*
-rw-------. 1 pnico pnico 22 Jun 30 11:40 testfile
-rw-------. 1 pnico pnico 32 Jun 30 11:44 testfile.huff
-rw-------. 1 pnico pnico 22 Jun 30 11:44 testfile.out
```

Note that, because `testfile` is so small, this made the "compressed" file bigger. Consider this example with a larger file, the class notes for cpe357:

```
% ls -l notes.tar
-rw-------. 1 pnico pnico 317440 Jun 30 11:47 notes.tar
% hencode notes.tar notes.tar.huff
% ls -l notes.*
-rw-------. 1 pnico pnico 317440 Jun 30 11:47 notes.tar
-rw-------. 1 pnico pnico 201141 Jun 30 11:48 notes.tar.huff
%
```

A reduction of almost 37% isn't so bad.

Appendix

You may be wondering how I got those pretty graphs on the previous pages, and, more likely, how you can see the trees in progress to see if you're building the right tree. Here's how:

`dot(1)` is a program designed to draw directed graphs when given a description of the graph.

My demo versions of `htable`, `hencode`, and `hdecode` have an undocumented option, `-d`, that dumps a dot graph of the tree at each stage, named `stageXXXX.dot` where XXXX is the number of the stage (they also have an undocumented `-v` option that dumps the histogram). You could make yours do this, too. It's a great procrastination opportunity.

The dot language is documented at `https://graphviz.org/documentation/`. (It's simpler than it first looks.)

To compile a dot file into postscript, you do:

```
$ dot -Tps file.dot > file.ps
```

You can then display it with your favorite postscript viewer. It's not the best, but `gs(1)` exists on the CSL machines.

Here's an example of me doing this (I send the output to `/dev/null` because I don't care about it, only the graph):

```
-bash-4.2$ ls
foo
-bash-4.2$ hencode -d foo /dev/null
-bash-4.2$ ls
foo  stage0000.dot  stage0001.dot  stage0002.dot  stage0003.dot
-bash-4.2$ dot -Tps stage0000.dot > stage0000.ps
-bash-4.2$ gs stage0000.ps
```

If you really want to show off your bash programming chops you could compile them all at once thusly:

```
-bash-4.2$ rm stage000*
-bash-4.2$ hencode -d foo /dev/null
-bash-4.2$ for f in stage*dot;  do dot -Tps $f > ‘echo $f | sed s/dot/ps/‘; done
-bash-4.2$ ls
foo            stage0000.ps   stage0001.ps   stage0002.ps   stage0003.ps
stage0000.dot  stage0001.dot  stage0002.dot  stage0003.dot
-bash-4.2$
```