CS360 -- Lab 3

- **CS360 -- Systems Programming**
- **James S. Plank**
- **This file: http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/labs/Lab-3-Huffman/index.html**
- **Lab Directory: /home/jplank/cs360/labs/Lab-3-Huffman**

What you submit

You will submit the file **huff_dec.c**. This must compile with:

```
gcc -Wall -Wextra -o huff_dec huff_dec.c
```

This means no Libfdr for this lab. I'll give you some help on organizing your data structures below.

---

xxd

Do "man xxd". You may find it a helpful program. In particular, we'll be looking at the output of **xxd -g 1** quite a bit. Please read up on it.

Variable length encoding

Your job this week is to write a decoding program called **huff_dec.c**. (When you compile it, put it into **bin/huff_dec** to make the gradescript work.)

The program takes two command line arguments -- a code defintion file, and an input file. It then uses the code defined in the code definition file to decode the input file. Its output is on standard output.

The code associates strings with sequences of bits. For example, one code associates the following strings with the following sequences of bits:

String	Sequence of bits
" "	10
"\n"	1100
"o"	00
"p"	010
"f"	1101
"t"	111
"the"	011

You'll note that no sequence of bits is a prefix of another sequence of bits. This means that you can decode a stream of bits into unique sequences. For example:

111001001110111000101100

maps to this stream of unique sequence of bits:

111 00 10 011 10 111 00 010 1100 = "to the top\n"

The code definition file defines the associations. Its format is a stream of null-terminated strings that come in pairs: the string and the sequence of bits defined as a stream of zero and one characters. Thus, for example, the above code is stored in data/t-code.txt:

```
UNIX> xxd -g 1 data/t-code.txt
00000000: 6f 00 30 30 00 70 00 30 31 30 00 74 68 65 00 30  o.00.p.010.the.0
00000010: 31 31 00 20 00 31 30 00 0a 00 31 31 30 30 00 66  11. .10...1100.f
00000020: 00 31 31 30 31 00 74 00 31 31 31 00              .1101.t.111.
UNIX>
```

The first string is "o", and it is associated with "00". Next is "p", which is associated with "010", and so on.

The input file is in a specific format. The *last* four bytes represent an unsigned integer (in little endian), which says how many bits should be read from the input file. The preceding bytes contain the stream of bits. If the total number of bits is *t* then the size of the file should be:

ceil(t/8) + 4 bytes.

Within each byte, the stream of bits starts with the least significant bit. Thus, the stream "10000000" is represented by the byte 0x1, and the stream "00000001" is represented by the byte 0x80. (That's backwards from how you usually view bites, but it will actually make your life easier).

Let's look at data/t-encoded.txt:

```
UNIX> xxd -g 1 data/t-encoded.txt
00000000: 27 77 34 18 00 00 00                             'w4....
UNIX>
```

The last four bytes contain the integer 0x18, which equals 24. Thus, the bit stream contains 24 bits, which are stored in three bytes: 0x27, 0x77 and 0x34. Converting them to a bit stream:

```
    0x27 = "11100100"            # Remember, the bits are "backward".
    0x77 = "11101110"
    0x34 = "00101100"
```

Yields the stream "111001001110111000101100" = "111","00","10","011","10","111","00","010","1100" = "to the top\n".

Let's try another example, in data/t-encoded-2.txt:

```
UNIX> xxd -g 1 data/t-encoded-2.txt
00000000: 47 0c 0e 00 00 00                                G.....
UNIX>
```

The last four bytes contain 0xe = 14. Thus, the first two bytes contain a string of 14 bits: 0x47 = "11100010" and 0x0c = "00110000". The string is "11100010001100" = "111","00","010","00","1100" = "topo\n".

Armed with this knowledge, you are now equipped to write **huf_dec**, which should decode the input file using the code defined in the code definition file. Use the C standard I/O library to read both the code definition file and the input file. In other words, do *not* use the Fields library -- it's actually not helpful here. Instead, use **fread()**.

You may assume that the code definition file defines a code in which no sequence of bits is a prefix of another. You may also assume that no string or sequence of bits is longer than 10000 characters.

You must test for the following errors:

- The code definition file is not in the correct format. Do this at the beginning.
- The input file is less than four bytes in size.
- The size of the input file does not match the specified number of bits. Do this after reading in the code definition file.
- There is an unrecognized sequence of bits, or an incomplete sequence at the end of the bit stream. Flag this error after you have decoded as much as you can.

In each case, make sure your program outputs what mine does.

You may assume that the output of **bin/huff_dec** is composed solely of printable characters.

The gradescript can take a little while, because the example files can get pretty big.

Organizing your data structures.

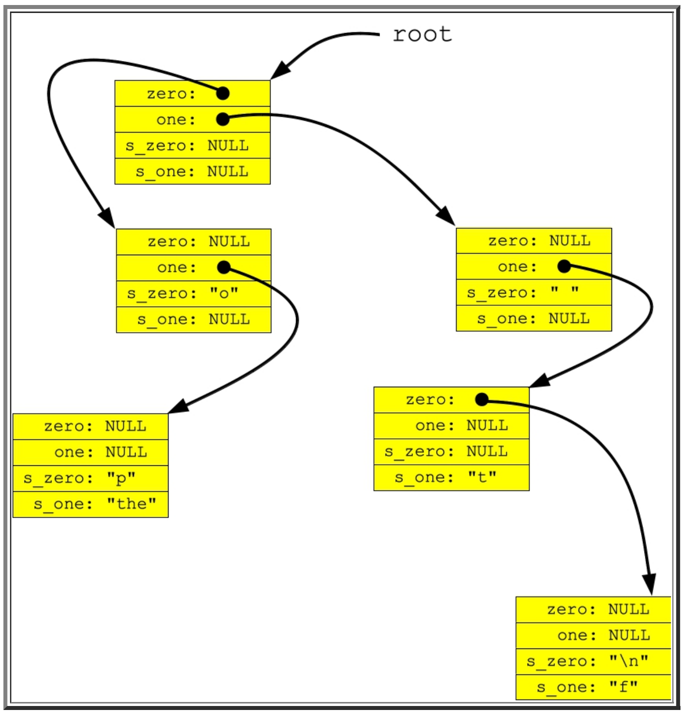
Try something like the following:

```
struct huff_node {
  struct huff_node *zero;
  struct huff_node *one;
  char *s_zero;
  char *s_one;
};
```

Organize nodes in a tree, where each node tells you what to do when you read a bit. If the bit is 0 and **zero** isn't NULL, then you go to the next node for the next bit. If **zero** is NULL, then you emit the string in **s_zero** and start over. Obviously, you handle 1 similarly. If both **zero** and **s_zero** are NULL, then that combination of bits has not been defined.

You can start with a single node at the root with all fields equaling NULL, and then build the tree as you read the code file. This will give you some good experience with pointers and **malloc()**.

Here's a picture of the code above (the one in **data/t-code.txt**):



Fun helper programs

The program **bin/huff_create_code** reads standard input and creates a decent *Huffman* code from it on standard output. It has an optional first argument called *divisor*. The default is 5. If *divisor* is zero, then the code is only composed of characters. Otherwise, the program determines all of the unique words in the file. Suppose that number is *u*. Then it also encodes the *u/divisor* most frequent words in the file. That can lead to better compression. The code is randomized, so if you run this twice on the same input, you'll get a different code.

The program **bin/huff_create_code-2** takes no command line arguments, and produces a pretty crappy code.

If you're bored someday, read about Huffman coding on Wikipedia: https://en.wikipedia.org/wiki/Huffman_coding.

The program **bin/huff_enc** takes two command line arguments -- a code defintion file and the name of an output file. It then encodes standard input to the output file using the code.

For example:

```
UNIX> cat data/celebrate.txt
I just want to celebrate another day of living
I just want to celebrate another day of life
Put my faith in the people but the people let me down
So I turned the other way and I carried on anyhow
I just want to celebrate another day of living
I just want to celebrate another day of life
UNIX> bin/huff_create_code < data/celebrate.txt > data/celebrate-code.txt
UNIX> bin/huff_enc < data/celebrate.txt data/celebrate-code.txt data/celebrate-encoded.txt
UNIX> bin/huff_dec data/celebrate-code.txt data/celebrate-encoded.txt
I just want to celebrate another day of living
I just want to celebrate another day of life
Put my faith in the people but the people let me down
So I turned the other way and I carried on anyhow
I just want to celebrate another day of living
I just want to celebrate another day of life
UNIX> bin/huff_create_code-2 < data/celebrate.txt > data/celebrate-inefficient.txt
UNIX> bin/huff_enc < data/celebrate.txt data/celebrate-inefficient.txt data/celebrate-ineff-encoded.txt
UNIX> bin/huff_dec data/celebrate-inefficient.txt data/celebrate-ineff-encoded.txt
I just want to celebrate another day of living
I just want to celebrate another day of life
Put my faith in the people but the people let me down
So I turned the other way and I carried on anyhow
I just want to celebrate another day of living
I just want to celebrate another day of life
UNIX> ls -l data/cel*
-rw-r--r--. 1 jplank jplank 275 Feb  4 16:24 data/celebrate-code.txt
-rw-r--r--. 1 jplank jplank 109 Feb  4 16:24 data/celebrate-encoded.txt              # This file compresses the input.
-rw-r--r--. 1 jplank jplank 905 Feb  4 16:34 data/celebrate-ineff-encoded.txt        # This code is so bad it expands the file by over a factor of three.
-rw-r--r--. 1 jplank jplank 714 Feb  4 16:34 data/celebrate-inefficient.txt
-rw-r--r--. 1 jplank jplank 288 Feb  4 16:24 data/celebrate.txt
UNIX>
```

You can see the code in **data/celebrate-code.txt** using **xxd**:

```
UNIX> xxd -g 1 data/celebrate-code.txt
00000000: 72 00 30 30 30 30 30 00 77 61 6e 74 00 30 30 30  r.00000.want.000       # 00000 maps to "r" and 00001 maps to "want"
00000010: 30 31 00 67 00 30 30 30 31 30 30 00 6d 00 30 30  01.g.000100.m.00       # 000100 maps to "g"
00000020: 30 31 30 31 00 76 00 30 30 30 31 31 30 00 50 00  0101.v.000110.P.
00000030: 30 30 30 31 31 31 30 00 53 00 30 30 30 31 31 31  0001110.S.000111
00000040: 31 00 69 00 30 30 31 30 00 61 00 30 30 31 31 30  1.i.0010.a.00110
00000050: 00 62 00 30 30 31 31 31 30 30 00 63 00 30 30 31  .b.0011100.c.001
00000060: 31 31 30 31 00 75 00 30 30 31 31 31 31 00 0a 00  1101.u.001111...
00000070: 30 31 30 30 30 00 49 00 30 31 30 30 31 00 68 00  01000.I.01001.h.
00000080: 30 31 30 31 30 00 77 00 30 31 30 31 31 30 00 79  01010.w.010110.y
00000090: 00 30 31 30 31 31 31 00 74 00 30 31 31 30 00 65  .010111.t.0110.e
000000a0: 00 30 31 31 31 00 20 00 31 30 00 66 00 31 31 30  .0111. .10.f.110
000000b0: 30 30 00 6c 00 31 31 30 30 31 00 6f 00 31 31 30  00.l.11001.o.110
000000c0: 31 00 6e 00 31 31 31 30 30 00 61 6e 6f 74 68 65  1.n.11100.anothe
000000d0: 72 00 31 31 31 30 31 30 00 63 65 6c 65 62 72 61  r.111010.celebra
000000e0: 74 65 00 31 31 31 30 31 31 00 64 00 31 31 31 31  te.111011.d.1111
000000f0: 30 30 00 64 61 79 00 31 31 31 31 30 31 00 6a 75  00.day.111101.ju
00000100: 73 74 00 31 31 31 31 31 30 00 70 00 31 31 31 31  st.111110.p.1111
00000110: 31 31 00                                         11.
UNIX>
```