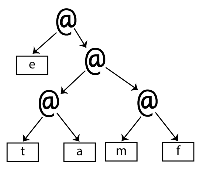
15-122: Principles of Imperative Computation, Spring 2013

Homework 6 Programming: Huffmanlab



Due: Thursday, April 4, 2013 by 23:59

For the programming portion of this week's homework, you'll write a C0 implementation of a popular compression technique called Huffman coding. In fact, you will write parts of it twice, once on C0 and once in C.

- `huffman.c0` (described in Sections 1, 2, 3, and 4)
- `huffman.c` (described in Section 5)

You should submit these files electronically by the due date. Detailed submission instructions can be found below.

Important Note: For this lab you must restrict yourself to **25 or fewer Autolab submissions**. As always, you should compile and run some test cases locally before submitting to Autolab.

Assignment: Huffman Coding (20 points in total)

Starter code. Download the file `hw6-handout.tgz` from the course website. When you untar it, you will find several C0 and C files and a `lib/` directory with some provided libraries.

You should not modify or submit the library code in the `lib/` directory, nor should you rely on the internals of these implementations. When we are testing your code, we will use our own implementations of these libraries with the same interface, but possibly different internals.

Compiling and running. You will compile and run your code using the standard C0 tools and the `gcc` compiler. You should compile and run the C0 version with

```
% cc0 -d -o huff0 huffman.c0 huffman-main.c0
% ./huff0
```

You should compile and run the C version with

```
% gcc -DDEBUG -g -Wall -Wextra -Werror -std=c99 -pedantic -o huff \
  lib/xalloc.c lib/heaps.c freqtable.c huffman.c huffman-main.c
% ./huff
```

Submitting. Once you've completed some files, you can submit them to Autolab. There are two ways to do this:

From the terminal on Andrew Linux (via cluster or ssh) type:

```
% handin hw6 huffman.c0 huffman.c
```

Your score will then be available on the Autolab website.

Your files can also be submitted to the web interface of Autolab. To do so, please `tar` them, for example:

```
% tar -czvf sol.tgz huffman.c0 huffman.c
```

Then, go to `https://autolab.cs.cmu.edu/15122-s13` and submit them as your solution to homework 6 (huffmanlab).

You may submit your assignment up to 25 times. When we grade your assignment, we will consider the most recent version submitted before the due date.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should provide loop invariants and any assertions that you use to check your reasoning. If you write any auxiliary functions, include precise and appropriate pre- and postconditions. Also include corresponding `REQUIRES`, `ENSURES`, and `ASSERT` macros in your C programs, especially where they help explain program properties that are important but not obvious.

You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Unit testing. You should write unit tests for your code. For this assignment, unit tests will not be graded, but they will help you check the correctness of your code, pinpoint the location of bugs, and save you hours of frustration.

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. If you find yourself writing the same code over and over, you should write a separate function to handle that computation and call it whenever you need it. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on Piazza if you're unsure of what constitutes good style.

Task 0 *As with the previous assignment, we expect you to be writing with good style; we have given you feedback on style for multiple assignments and we have shown you many examples of code with reasonable style. This assignment will initially have all 20 points assigned through the autograder, but up to 5 points may be deducted for style issues. We will likely focus our attention on your C code.*

Data Compression: Overview

Whenever we represent data in a computer, we have to choose some sort of *encoding* with which to represent it. When representing strings in C0, for instance, we use ASCII codes to represent the individual characters. Other encodings are possible as well. The UNICODE standard (as another encoding example) defines a variety of character encodings with a variety of different properties. The simplest, UTF-32, uses 32 bits per character.

Under the ASCII encoding, each character is represented using 7 bits, so a string of length n requires $7n$ bits of storage, which we usually round up to $8n$ bits or n bytes. For example, consider the string `"more free coffee"`; ignoring spaces, it can be represented in ASCII as follows with $14 \times 7 = 98$ bits:

$$\begin{gathered}1101101 \cdot 1101111 \cdot 1110010 \cdot 1100101 \cdot 1100110 \cdot \\ 1110010 \cdot 1100101 \cdot 1100101 \cdot 1100011 \cdot 1101111 \cdot \\ 1100110 \cdot 1100110 \cdot 1100101 \cdot 1100101\end{gathered}$$

This encoding of the string is rather wasteful, though. In fact, since there are only 6 distinct characters in the string, we should be able to represent it using a custom encoding that uses only $\lceil \log 6 \rceil = 3$ bits to encode each character. If we were to use the custom encoding shown in Figure 1,

Character	**Code**
`‘c’`	000
`‘e’`	001
`‘f’`	010
`‘m’`	011
`‘o’`	100
`‘r’`	101

Figure 1: A custom fixed-length encoding for the non-whitespace characters in the string `"more free coffee"`.

the string would be represented with only $14 \times 3 = 42$ bits:

$$\begin{gathered}011 \cdot 100 \cdot 101 \cdot 001 \cdot 010 \cdot \\ 101 \cdot 001 \cdot 001 \cdot 000 \cdot 100 \cdot \\ 010 \cdot 010 \cdot 001 \cdot 001\end{gathered}$$

In both cases, we may of course omit the separator “·” between codes; they are included only for readability.

If we confine ourselves to representing each character using the same number of bits, i.e., a *fixed-length encoding*, then this is the best we can do. But if we allow ourselves a *variable-length encoding*, then we can take advantage of special properties of the data: for instance, in the sample string, the character `’e’` occurs very frequently while the characters `’c’` and `’m’` occur very infrequently, so it would be worthwhile to use a smaller bit pattern to encode the character `’e’` even at the expense of having to use longer bit patterns to encode `’c’`

Character	**Code**
`‘e’`	0
`‘o’`	100
`‘m’`	1010
`‘c’`	1011
`‘r’`	110
`‘f’`	111

Figure 2: A custom variable-length encoding for the non-whitespace characters in the string `"more free coffee"`.

and `’m’`. The encoding shown in Figure 2 employs such a strategy, and using it, the sample string can be represented with only 34 bits:

$$1010 \cdot 100 \cdot 110 \cdot 0 \cdot 111 \cdot$$
$$110 \cdot 0 \cdot 0 \cdot 1011 \cdot 100 \cdot$$
$$111 \cdot 111 \cdot 0 \cdot 0$$

Since this encoding is *prefix-free*—no code word is a prefix of any other code word—the “·” separators are redundant here, too.

It can be proven that this encoding is optimal for this particular string: no other encoding can represent the string using fewer than 34 bits. Moreover, the encoding is optimal for *any* string that has the same distribution of characters as the sample string. In this assignment, you will implement a method for constructing such optimal encodings developed by David Huffman.

Huffman Coding: A Brief History

Huffman coding is an algorithm for constructing optimal prefix-free encodings given a frequency distribution over characters. It was developed in 1951 by David Huffman when he was a Ph.D student at MIT taking a course on information theory taught by Robert Fano. It was towards the end of the semester, and Fano had given his students a choice: they could either take a final exam to demonstrate mastery of the material, or they could write a term paper on something pertinent to information theory. Fano suggested a number of possible topics, one of which was efficient binary encodings: while Fano himself had worked on the subject with his colleague Claude Shannon, it was not known at the time how to efficiently construct optimal encodings.

Huffman struggled for some time to make headway on the problem and was about to give up and start studying for the final when he hit upon a key insight and invented the algorithm that bears his name, thus outdoing his professor, making history, and attaining an “A” for the course. Today, Huffman coding enjoys a variety of applications: it is used as part of the DEFLATE algorithm for producing ZIP files and as part of several multimedia codecs like JPEG and MP3.

1 Huffman Trees

Recall that an encoding is *prefix-free* if no code word is a prefix of any other code word. Prefix-free encodings can be represented as binary *full* trees with characters stored at the leaves: a branch to the left represents a 0 bit, a branch to the right represents a 1 bit, and the path from the root to a leaf gives the code word for the character stored at that leaf. For example, the encodings from Figures 1 and 2 are represented by the binary trees in Figures 3 and 4, respectively.

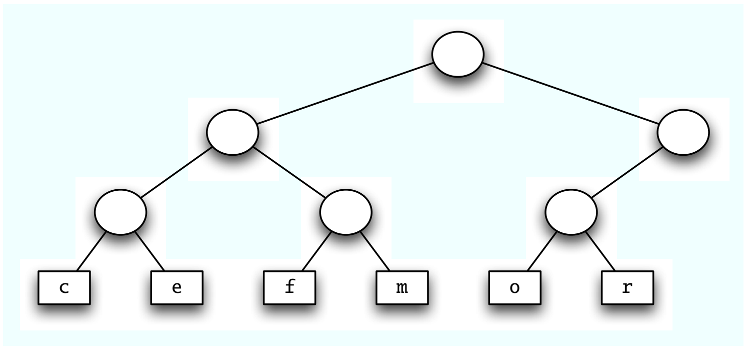


Figure 3: The custom encoding from Figure 1 as a binary tree.

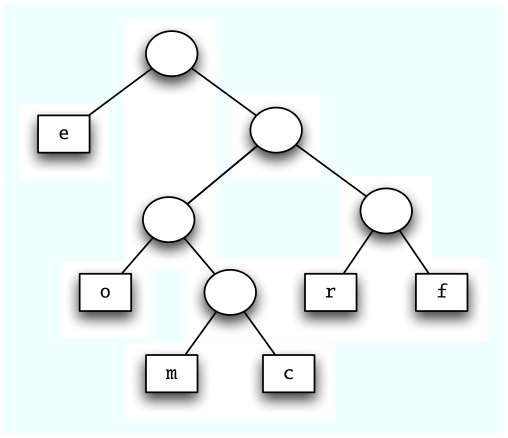


Figure 4: The custom encoding from Figure 2 as a binary tree.

The tree representation reflects the optimality in the following way: frequently-occurring characters have shorter paths to the root. We can see this property clearly if we label each subtree with the total frequency of the characters occurring at its leaves, as shown in Figure 5. A frequency-annotated tree is called a *Huffman tree*.

Huffman trees have a recursive structure: a Huffman tree is either a leaf containing a character and its frequency, or an interior node containing the combined frequency of two child Huffman trees. Since only the leaves contain character data, we draw them as rectangles to distinguish them from the interior nodes, which we draw as circles.

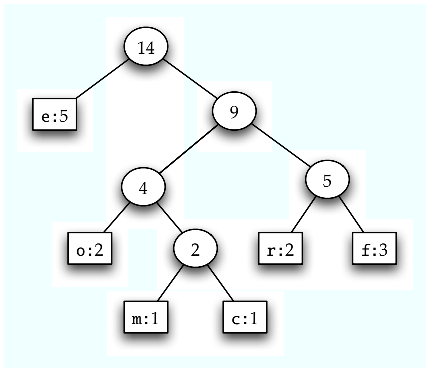


Figure 5: The custom encoding from Figure 2 as a binary tree annotated with frequencies, i.e., a Huffman tree.

We represent both kinds of Huffman tree nodes in C0 using a `struct htree_node`:

```
struct htree_node {
  char value;                          /* '\0' except at leaves */
  int frequency;
  struct htree_node* left;
  struct htree_node* right;
};
typedef struct htree_node htree;
```

The `value` field of an `htree` should consist of a character `'\0'` everywhere except at the leaves of the tree, and every interior node should have exactly two children. These criteria give rise to the following recursive definitions:

> An `htree` is a *valid htree* if it is non-`NULL`, its `frequency` is strictly positive, and it is either a *valid htree leaf* or a *valid htree interior node.*
>
> An `htree` is a *valid htree leaf* if its `value` is *not* `'\0'` and its `left` and `right` children are `NULL`.
>
> An `htree` is a *valid htree interior node* if its `value` is `'\0'`, its `left` and `right` children are *valid htrees*, and its `frequency` is the sum of the `frequency` of its children.

Unlike many other data structures we discussed, Huffman trees have no header struct.

Task 1 (2 pts) *Implement the following functions on a Huffman tree:*

Function:	***Returns true iff...***
`bool is_htree0(htree* H);`	*the node is a leaf*
`bool is_htree2(htree* H);`	*the node is an interior node*
`bool is_htree(htree* H);`	*the tree is a Huffman tree*

that formalize the Hufmann tree data structure invariants.

2 Constructing Huffman Trees

Huffman's key insight was to use the frequencies of characters to build an optimal encoding tree from the bottom up. Given a set of characters and their associated frequencies, we can build an optimal Huffman tree as follows:

1. Construct leaf Huffman trees for each character/frequency pair.
2. Repeatedly choose two minimum-frequency Huffman trees and join them together into a new Huffman tree whose frequency is the sum of their frequencies.
3. When only one Huffman tree remains, it represents an optimal encoding.

This is an example of a *greedy algorithm* since it makes locally optimal choices that nevertheless yield a globally optimal result at the end of the day. Selection of a minimum-frequency tree in step 2 can be accomplished using a *priority queue* based on a heap. A sample run of the algorithm is shown in Figure 6.

Task 2 (7 pts) *Write a function*

```
htree* build_htree(freqtable table)
//@requires is_freqtable(table);
//@ensures is_htree2(\result);
  ;
```

that constructs an optimal encoding for an n-character alphabet using Huffman's algorithm. The frequency table is an integer array of length 128, containing a positive frequency for every ASCII character in the alphabet and 0 *otherwise. See file* `freqtable.c0`*.*

Use the code in the included `lib/heaps.c0` *as your implementation of priority queues.*

Two brief notes: Huffman trees for a frequency table with just one character are not particularly useful, because the optimal encoding of a single character would be the empty bit string. But then source strings, all of which must just repeat the same character, would be mapped to the same empty string. Therefore, `build_htree` must signal an error (by calling the `error` function) if there are fewer than two characters with non-zero frequency, and otherwise return an interior node as a result.

Huffman trees are not unique due to symmetries, additionally complicated by the fact that multiple characters or subtrees might have identical priorites. All possible valid Huffman

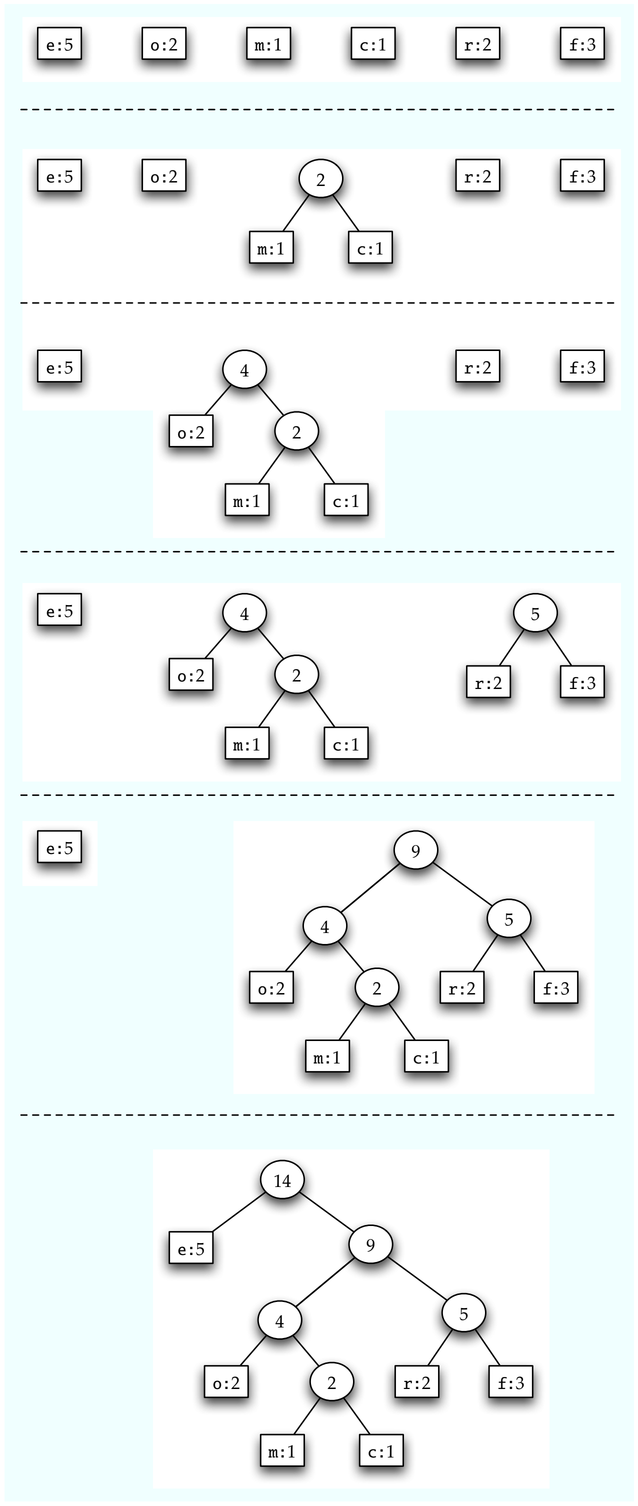


Figure 6: Building an optimal encoding using Huffman's algorithm.

codes of a given string will have the same length (due to its optimality guarantee) but may otherwise be different. So the particular codes produced in your implementation may look different than the ones shown in this handout.

You should test your implementation using some frequency tables. These have a textual representation format

```
<character 1>:<frequency 1>
<character 2>:<frequency 2>
...
```

where the character is separated from the frequency by a colon `':'`.

3 Encoding Strings

Once you successfully constructed a Huffman tree, you move to the next task of encoding a given string. To find a Huffman encoding for a character you traverse the tree to the character you want, outputting a 0 every time you take a lefthand branch, and an 1 every time you take a righthand branch. As an example, we encode `"roomforcreme"` using the encoding from Figure 7:

```
110 100 100 1010 111 100 110 1011 110 0 1010 0
r   o   o   m    f   o   r   c    r   e m    e
```

The spaces here are just for illustration and should not occur in the actual output, which should just be

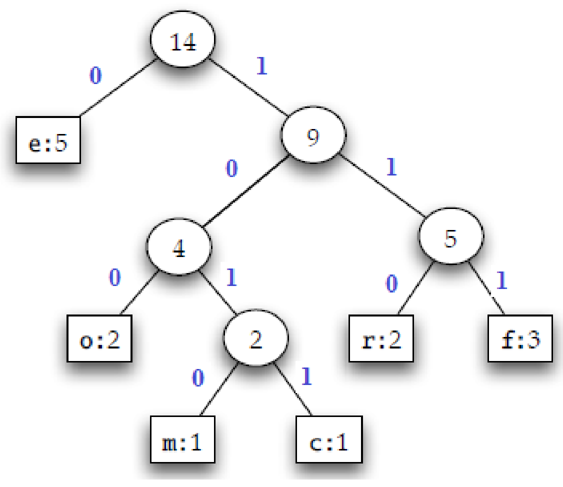
```
"11010010010101111001101011110010100"
```

For simplicity, we represent the output as a string, using the character `'0'` for the bit 0, and the character `'1'` for the bit 1. This helps us to interpret the bit string, but it makes them appear artificially long. In a realistic application, 32 bits would be combined into one machine word.

To encode a string using a Huffman tree, we need to convert each character to its bit string encoding, returning the resulting string. Given a Huffman tree, we can retrieve the bit string encoding corresponding to a particular character by traversing the tree. Encoding a string via this method, however, is inefficient, because we have to traverse the tree once for each character we encode. We can optimize this process by using an array with 128 entries (one for each possible character), mapping ASCII characters to their bit string representations. This array is called a *code table.* A code table maps characters not in the alphabet of the particular Huffman tree to the empty string.

Task 3 (3 pts) *Write a function*

```
string[] htree_to_codetable(htree* H)
//@requires is_htree(H);
//@ensures is_codetable(\result);
  ;
```



char	encoding
m	1010
o	100
r	110
e	0
f	111
c	1011

Figure 7: A Huffman tree annotated with 0's and 1's.

mapping Huffman trees to their code tables.

Finally, we are ready to write the encoding function.

Task 4 (3 pts) *Write a function*

```
string encode(htree* H, string input)
//@requires is_htree2(H);
//@ensures is_bitstring(\result);
  ;
```

that efficiently encodes a given string input using a Huffman tree with at least two characters via its code table.

The function should return the encoded bit string if the string can be encoded and should signal an error otherwise, using the `error` function.

4 Decoding Bit Strings

Huffman trees are a data structure well-suited to the task of decoding encoded data. Given an encoded bit string and the Huffman tree that was used to encode it, we decode the bit string as follows:

1. Initialize a pointer to the root of the Huffman tree.
2. Repeatedly read bits from the bit string and update the pointer: when you read a 0 bit, follow the left branch, and when you read a 1 bit, follow the right branch.
3. Whenever you reach a leaf, output the character at that leaf and reset the pointer to the root of the Huffman tree.

If the bit string was properly encoded, then when all of the bits are exhausted, the pointer should once again point to the root of the Huffman tree. If this is not the case, then the decoding fails for the given input.

As an example, we can use the encoding from Figure 7 to decode the following message:

```
1 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 0
    r     o     o       m     f     o     r       c     r e       m e
```

The implementation of a decode function

```
string decode(htree* H, string bits)
//@requires is_htree2(H);
  ;
```

is already provided for you. The function takes in a bit string and the Huffman tree to decode it with. As mentioned before, Huffman trees are not unique due to symmetries, so different optimal trees can be generated at different times. Moreover, we may not want to use the optimal Huffman tree for every encoding. An example of this would be to have a generic Huffman encoding for English texts based on the widespread frequency of each character. Although the generic tree would not be optimal for all texts, it would eliminate the need to generate a Huffman tree for each text while still providing better compression versus fixed-length encoding.

5 Mapping C0 to C

One of the goals of this homework assignment is to get some basic familiarity with C. In this last task you are asked to translate the code from Tasks 1 (defining Huffman tree invariants) and 2 (building Huffman trees from frequency tables) from C0 into C. We will provide the necessary materials in lecture and recitation.

Task 5 (5 pts) *We have provided starter code for this in the file* `huffman.c` *and header file* `huffman.h`*. Fill in the rest of* `huffman.c` *so it can be compiled with the* `huffman-main.c` *file in order to decode strings with a given Huffman tree. Your implementation does not need to know how to encode a string.*

We will compile and test your file with

```
% gcc -DDEBUG -g -Wall -Wextra -Werror -std=c99 -pedantic -o huff \
  lib/xalloc.c lib/heaps.c freqtable.c huffman.c huffman-main.c
```

where we supply our own version of `huffman-main.c`*. We will also execute your binary with* `valgrind` *in order to make sure that you have not introduced any memory leaks.*

We do not ask you to translate the code tables or the Huffman encoding functions. You should test your C code by decoding bit strings that result from the encoding of a string with your C0 implementation, using the same frequency table. There will be two key things to keep in mind when you translate your code. First, your C code will need to work with the C representation of strings, which is rather different than the C0 representation. Check the Recitation 18 notes or the tutorial on translating from C0 to C at `http://c0.typesafety.net/tutorial/From-C0-to-C:-Basics.html#wiki-strings` for hints here. Second, you will need to carefully free all of the memory that you allocate in order to pass all of valgrind's tests.

It is important that you do not change the essence of how you build Huffman trees from the frequency table, since the exact Huffman tree will depend on choices made in your implementation. Since it also depends on the implementation of priority queues, we have preserved its functionality fully when porting it from C0 to C. A string encoded with one Huffman tree can only be decoded with the same Huffman tree.

Task 6 (Bonus) *For extra credit, complete the implementation so that we can encode as well as decode strings with the C implementation. Make sure that your extension does not introduce any memory leaks.*