15-122 : Principles of Imperative Computation

Summer 1 2012

Assignment 6: Huffman Coding

(Programming Part)

Due: Monday, June 18, 2011 by 23:59

For the programming portion of this week's homework, you'll write a C_0 implementation of a popular compression technique called Huffman coding You will write two C_0 files corresponding to different tasks:

- `encoding.c0` (described in Section 1.3 and 1.4),
- `decoding.c0` (described in Section 1.5)

In addition, you will be asked to generate a suite of examples using ideas from testing methodologies presented in class.
You should submit these files electronically by the due date. Detailed submission instructions can be found below.

1 Assignment: Huffman Coding(30 points)

Starter code. Download the file `hw6.zip` from the course website. When you unzip it, you will find the following files

`encoding.c0`	Huffman's encoding algorithm	Tasks 1, 2 and 3
`decoding.c0`	Huffman's decoding algorithm	Task 4 an 5
`heaps.c0`	Generic implementation of a priority queue	DO NOT SUBMIT
`heaps.h0`	Client-side implementation of a priority queue	DO NOT SUBMIT
`hash-maps.c0`	Generic implementation of a hash map	DO NOT SUBMIT
`hash-maps.h0`	Client-side implementation of a hash map	DO NOT SUBMIT
`readfile.c0`	Code for reading words from a file	DO NOT SUBMIT
`freqtable.c0`	Code for reading frequencies and characters	DO NOT SUBMIT
`bitstring.c0`	Bitstring representation	DO NOT SUBMIT
`hw6-main.c0`	The driver program for testing	DO NOT SUBMIT

Testing your code: You should test individual functions as you work on the assignment. However, we have also provided the HW6 main in `hw6-main.c0`, which will comprehensively test all of the parts of this assignment. Compile and test with:

```
cc0 -d -x hw6-main.c0
```

This will only work once you've completed all tasks. Please test your individual functions as you work on this programming assignment.

Compiling and running. For this homework, use the `cc0` command as usual to compile your code. Don't forget to test your annotations by compiling with the `-d` switch to enable dynamic checking. **Warning:** *You will lose credit if your code does not compile with the* `-d`*.*

Submitting. Once you have completed some files, you can submit them by running the command

```
handin -a hw6 <file1>.c0 ... <fileN>.c0
```

The `handin` utility accepts a number of other switches you may find useful as well; try `handin -h` for more information.

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. 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.**

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. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

1.1 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 C_0, 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.

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"`.

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'` 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.

1.2 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.

1.3 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.

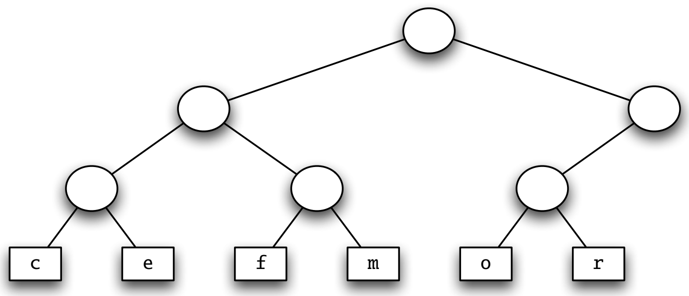


Figure 3: The custom encoding from Figure 1 as a binary tree.

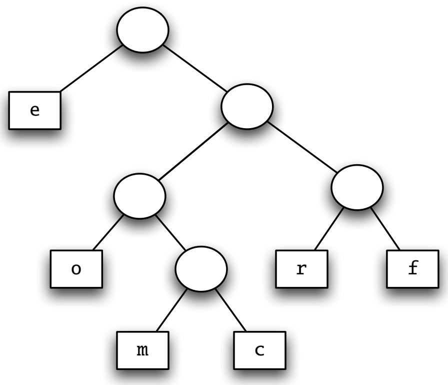


Figure 4: The custom encoding from Figure 2 as a binary tree.

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.

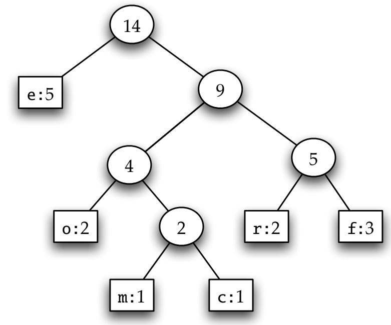


Figure 5: The custom encoding from Figure 2 as a binary tree annotated with frequencies, i.e., a Huffman tree.

We represent both kinds of Huffman tree nodes in C_0 using a `struct htree_node`:

```
typedef struct htree_node * htree;

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

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.

Task 1 (3 pts) *Implement the following functions on a Huffman tree:*

Function:	***Returns true iff...***
`bool is_htree_leaf(htree H)`	*the node is a leaf*
`bool is_htree_node(htree H)`	*the node is an internal node*
`bool is_htree(htree H)`	*the tree is a Huffman tree*

that formalize the tree data structure invariants.

Task 2 (2 pts) *Implement the following utility functions:*

`int htree_size(htree H)`	*return the number all nodes in the tree*
`int htree_count_leaves(htree H)`	*return the number of leaves in the tree*

1.4 Finding Optimal Encodings

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 is 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 3 (6 pts) *Write a function*

```
htree build_htree(char[] chars, int[] freqs, int n)
```

that constructs an optimal encoding for an `n`-character alphabet using Huffman's algorithm. The `chars` array contains the characters of the alphabet and the `freqs` array their frequencies, where `chars[i]` occurs with frequency `freqs[i]`. Use the code in the included `heaps.c0` as your implementation of priority queues.

To test your implementation of `htree build_htree`, you may use the code in `freqtable.c0`, which reads a character frequency table from a file in the following format:

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

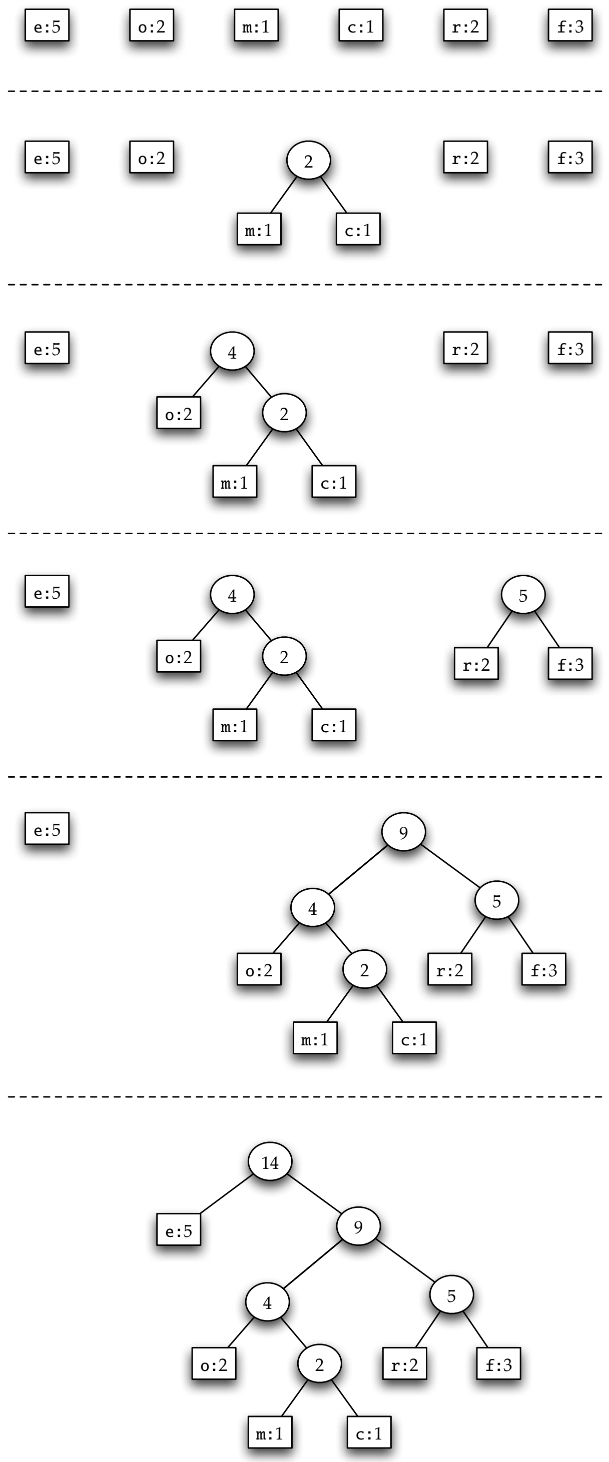
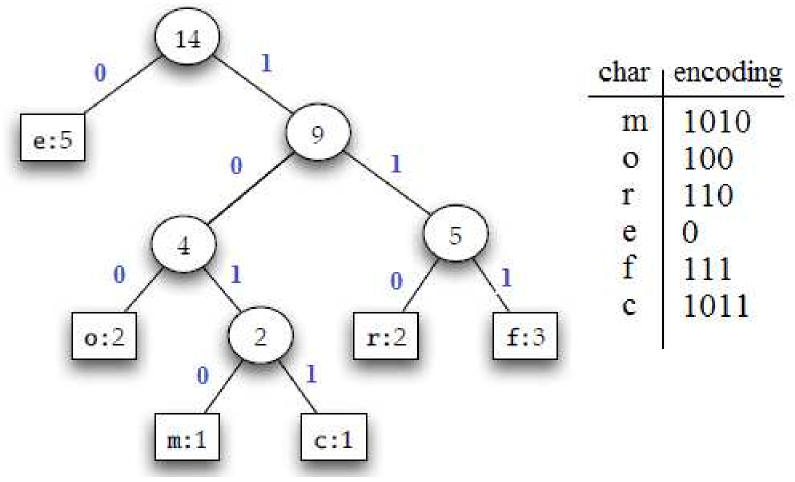


Figure 6: Building an optimal encoding using Huffman's algorithm.

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
```



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.

To encode a string using a Huffman tree, we need to convert each character to its bitstring encoding, returning the resulting string. Given a Huffman tree, we can retrieve the bitstring 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 a hash map to pair characters with their bitstring encodings.

Task 4 (5 pts) *Write a function*

`hmap htree_to_map(htree H)`

mapping characters to their corresponding Huffman-encoded bitstrings. Use the code in the included `hash-maps.c0` as your implementation of hash tables.

Task 5 (3 pts) *Write a function*

`bitstring encode(htree H, string input)`

that ***efficiently*** *encodes the given string `input` using the Huffman tree `H`.*

The function should return the encoded bitstring if the string can be encoded and should signal an error otherwise. Your implementation should have $O(n)$ asymptotic complexity, where n is the length of `input`. *Hint: you may find `htree_to_map` useful.*

1.5 Decoding Bitstrings

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

1. Initialize a pointer to the root of the Huffman tree.
2. Repeatedly read bits from the bitstring 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 bitstring 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 inputs.

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
```

Bitstrings can be represented in C_0 as ordinary strings composed only of characters `'0'` and `'1'`.

Task 6 (6 pts) *Implement a function*

```
string decode(htree H, bitstring bits)
```

that decodes `bits` *based on the Huffman tree* `H`*. The function should return the decoded string if the bitstring can be decoded and should signal an error otherwise.*

1.6 Testing Decode

Task 7 (5 pts) *For this task, write a series of tests (at least 5 tests) that you will use to be confident that your implementation of* `decode` *is correct based on your reading of the spec.*

Recall the notion of *equivalence classes* with regards to testing. Rather than testing *specific* inputs, we can generalize inputs by placing them into particular 'equivalence classes' based on their expected behavior.

Following black box testing methodology, write tests that are representative of different input equivalence classes, and think about the different kinds of results you expect from different inputs. For each test you will submit a test file and potentially some input files. Each individual test should be given a unique number used in the naming conventions below. There should be two primary components of each test:

- **Inputs:** The main portion of each test is a well chosen set of inputs that represents the behavior of the function on a large class of possible inputs. You should either hard-code these inputs into the testing function (see below) or store them in files that you submit along with your tests.

  If you choose to place the inputs in files, put the specification of the Huffman Tree and the bitstring in different files. Name them using the convention:

  `input#-<inputType>.txt`

  where # is the test number and `<inputType>` is `FT` for the frequency table and `BS` for the bitstring (For example, the bitstring input file for your second test should be named `input2-BS.txt`).

- **Main function:** You should write a main function that runs the test on your inputs. The main function should do three things:

  1. *Test Description:* Before doing anything else, you should print a short description of the test to the screen including the class of inputs it is testing and the expected result.
  2. *Build/Import Inputs:* Next you should build or read in the inputs to `decode` for this test. If you import the inputs from files, read in the bitstring with the `read_words` function defined in the `readfile.c0` file and import the frequency table used to build your Huffman Tree using the `read_freqtable` function defined in the file `freqtable.c0`. Call these functions with the bare file name (For example, load the bitstring for your second test by calling `read_words("input2-BS.txt")`).
  3. *Call* `decode`*:* Call `decode` on your inputs.
  4. *Verify Results:* If the test should result in an error, there will be no output to test. Otherwise compare the output to the output you expect for the test and return `0` if the test succeeded and `1` if it did not (recall that 0 is the return value of choice when a function succeeds).

  You should name the file containing the main function for the test using the following format:

  `test#-<expectedOutput>.c0`

  where # is the test number and `<expectedOutput>` is `F` if the function should end in an assertion or annotation failure or `S` if the function should succeed (return `0`). (For example, if your second test should succeed, you would name the file `test2-S.c0`)

Submit all test and input files. We will be looking for a set of tests that check all different possible outputs of the `decode` function without duplicate tests for inputs in the same equivalence class.

A few hints to get you started:

- You should be writing a relatively small number of tests, but at least 5.
- Your examples don't have to be long or even form complete sentences, but they should test the functionality explained in the spec.
- As a start, you might consider using the provided example above.