Weekly Programming Quiz 8: Huffman Encoding

CS 5002
Released: November 18, 2017
Due: December 1, 2017

Abstract

Huffman encoding is a data compression algorithm. Data compression means we can store the same amount of information (data) in a smaller amount of space (memory).

Encoding is the process of using a set of rules to transform data to a new set of data. The transformed data might be useful for a number of reasons: perhaps it is merely translated into a different language such that someone else can read it, or it could be obfuscated so that no one else can read it. Or, we might encode in such a way that makes the data unreadable, but smaller, so it's easier to pass around or transmit. Once the data has been transmitted, it can be de-coded into a format that someone finds readable.

Huffman encoding takes input text and generates a sequence of 0's and 1's that is smaller than the input text. The encoded text can then can be decoded by following an algorithm.

1 Quiz Objectives

The objectives of this quiz are to:

- Build trees
- Traverse trees
- Write code to a given interface
- Practice bit manipulation
- Identify your own helper methods and writing them
- Choose, design and implement appropriate data structures where needed

2 Background

In this quiz we're concerned with communicating between two computers. We can only use 0's and 1's to communicate between two computers– which ultimately get transmitted as pulses of electricity across a wire. Thus, we have to figure out how to take everything that we want to communicate (as humans), convert it into a series of 0's and 1's, which are converted into electronic pulses, which can be sent to another receiving device, which converts those electric pulses into 0's and 1's, and then reconstruct those 0's and 1's into letters, and words.

To understand Huffman encoding and it's significance, we need to understand how text is generally represented by computers.

2.1 Basic text encoding

Digital text is encoded with ASCII. Each character is represented by an integer in a range (0-127). That character is represented in 8 bits (0's and 1's) by converting the decimal number to a binary number. For reference, http://www.asciitable.com/ is a resource to see the mapping between characters and ASCII values.

> **Side note:** a `char` is usually 16 bits. The first 8 bits are all 0's. Assume the following discussion pads the 8-bit values with 8 0's on the left/at the beginning.

For example, the word "Bike":

Table 1: ASCII Values for 'Bike'

Character	ASCII Val (Decimal)	Binary
B	66	01000010
i	105	01101001
k	107	01101011
e	101	01100101

That means the sequences of bits that represent the word 'Bike' is:

Table 2: Bit sequence for 'Bike'

01000010	01101001	01101011	01100101

Computers represent the word "Bike" by a bunch of bits: "01000010|01101001|01101011|01100101". (The pipe is just showing the letter boundaries).

The length of this bit stream is always 8∗ [the number of letters]: for "Bike", we always need 32 bits to communicate "Bike". Can we improve on this? How much?

2.2 Improving on basic text encoding: Reduce the number of bits

The goal is to reduce the number of bits we use to represent the word 'Bike'. Since we only have 4 letters in the word 'Bike', we could easily do something like:

Table 3: Minimal Encoding for "Bike"

Character	Binary
B	00
i	01
k	10
e	11

Using this mapping the sequence of bits we transmit is "00-01-10-11"– only 8 bits instead of the previous 32.

Table 4: Variable Length Encoding for "Bicycle"

Character	Binary
B	010
i	011
c	10
y	110
l	111
e	00

This example is almost trivial: it's only 4 characters, and none of them are duplicated. Let's look at a possible encoding for "Bicycle", which has multiple 'c's in it (Table 4).

The sequence of bits with this mapping is "010-011-10-110-10-111-00"– again, only 18 bits, instead of the original 48 needed in the traditional approach. Note that in this mapping, the letter 'c' appears twice in the original word, and has the shortest number of bits to represent it. By mapping 'c' to a shorter sequence of bits (in this case, 2 instead of 3), we save 2 bits in our output.

However, the receiver also needs to know the mapping. The person on the other end has no idea how many bits is representing a single character (so they don't know where the character boundaries are), and also which bit sequence maps to which character. Without this, the receiver doesn't know how to decode '010-011-10-...' to 'Bike'. In the standard approach, the receiver knows that each sequence of 8 bits maps to a character, and has an agreed-upon map to translate the bits to a character (that ASCII table).

2.3 Huffman Encoding

Huffman encoding takes a similar approach to what is described above:

1. Figure out which characters are in the original string/word/document (and their frequencies, but we'll get to that later)
2. Determine a sequence of bits to represent each character in the input
3. Encode the string/word/document using this mapping
4. Transmit the sequence of bits to the receiver
5. Transmit/share the mapping with the receiver
6. Decode the sequence of bits into a message using the mapping

The pieces we're concerned with as a programmer (that is, for this quiz):

- Calculating characters & frequencies
- Creating the "mapping" — the Huffman tree
- Encoding a message with the mapping
- Decoding a message with the mapping

2.3.1 Calculating characters & frequencies

This part of the process should be straightforward. We're just looking to create a frequency table from the given input.

Table 5: Frequency Table for "Bike"

Character	Frequency
B	1
i	1
k	1
e	1

For the input 'Bike', the table is trivial (see Table 5).

For this quiz, the frequency table struct is as follows:

```
1 struct freqTable{
2   int charCount[NUM_ASCII_CHARS];
3 }
```

The array `charCount` represents the counts of each character. The index corresponds to the ASCII value. For example, the ASCII value of 'a' is 97. `charCount[97]` represents the count of `a`'s in the input string.

The function that populates the frequency table is:

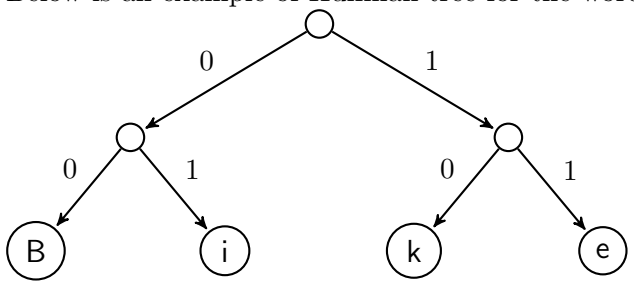
```
1 FreqTable *createFreqTable(char *document);
```

- `document` is the string or document to count frequencies from. Assume the document consists of characters with ASCII values between 32 and 126 (inclusive).

2.3.2 Creating the mapping: The Huffman Tree

With Huffman encoding, we create a tree to encode and decode a characters. The bit sequence to represent a character is simply the path one takes through the tree.

Below is an example of Huffman tree for the word "Bike":



Using the above tree, you can see how we got our earlier mapping of "00" → 'B', '01' → 'i', and so on.

Further, we can use this tree to go from "00011011" to the string "Bike" by tracing each bit through the tree. The first bit is 0, so we start at the root and take the left branch, labeled 0. There is no character at that point, so we take the next bit, and go left again. There is a character, so we now have the first character, 'B'. We go back to the root, and repeat the process starting with the next character, which is also a '0'.

The tree is created thusly:

- Create a node for each character (including the count), and put it in an appropriate data structure

- While there is more than 1 node in the list:
  - Select two nodes with the lowest counts
  - Create a new node that is the parent node of those two nodes
  - Set the count of that new node to be the sum of the counts of the two children nodes
  - Take the two children nodes out of the list
  - Put the new parent node in the list
- The last node in the list is the root of the tree.

The data structure to use for your tree is defined in `quiz8.h`:

```
1 struct HTree{
2   char c;
3   int freq;
4   HTree *p0;
5   HTree *p1;
6 };
```

- `char c` is the character represented by this node. If this node is an internal node (at least one of `p0` or `p1` is not `NULL`), `c` is 0.
- `int freq` is the frequency or count of characters reachable by this node. If this node is a leaf (thereby representing a single character) it's merely the frequency of that character in the input document. If this node is an internal node, it's the sum of the frequencies of the children of this node.
- `HTree *p0`: The left or `0` child of this node.
- `HTree *p1`: The right or `1` child of this node.

The function that will create the tree is as follows:

```
1 HTree* createEncodingTree(FreqTable *);
```

`createEncodingTree` should malloc and build the tree with the given frequencies.

2.3.3 Encode a message

Once we have a Huffman tree, the encoding is straightforward. Each character maps to a sequence of bits that represent a traversal through the tree.

The most efficient way to do the encoding is to generate a table such as Table 4. You can do this by traversing the tree one time. This approach is known as creating a "lookup table". Rather than performing a costly calculation every time we do something, we do the costly calculation one time, cache or store the results, and then use the table to look up the answer.

Thus, the process is:

1. Create the lookup table.
   What data structure might you want to use? You will want to use a traversal. Which traversal will be best for this? You will need to generate all paths in the tree.
2. For each character in the string:
   (a) Look up the sequence of bits in the lookup table
   (b) Append the bits to the output sequence

3. Return the final bit sequence

```
/**
 *  Take in a message, a pointer to a BitSeq (sequence of bits).
 *  Create a HTree to encode/decode the message, and return a pointer
 *.     to the root of the tree.
 *  Ensure the output is in the provided BitSeq.
 */
HTree *encode(char *s, BitSeq *out);
```

2.3.4 Decode a message

To decode, we start with a sequence of bits and a Huffman tree.

Go through each bit in the sequence, and let that be your traversal through the tree. If you hit a character, emit that character and start a new traversal with the next bit in the sequence.

1. For each bit in the sequence:
   (a) Start at the root of the tree.
   (b) If the bit == 0, go to the left; otherwise go to the right.
2. If the node has a character, emit the character. Otherwise, keep traversing.
3. Start at the root of the tree again, with the next bit in the sequence.

```
/**
 *  Take in a BitSeq that is an encoded message and
 *  a Huffman tree, return a pointer to the decode message.
 *  Use the tree to decode the bit sequence.
 */
char *decode(BitSeq *msg, HTree *root);
```

3 The Quiz

3.1 The Quiz

Fill out quiz8.c base on the function prototypes defined in quiz8.h.

You will likely find it helpful to define helper functions. Add those prototypes to quiz8.h, and define them in quiz8.c. Be sure to check in both files in for full credit.

4 Challenge

There are no challenge problems at this point: this is a challenging quiz. If you would like a challenge, post a note to the instructors on Piazza and we'll work faster to provide you with one.

5 Hopefully Helpful Hints

Read Marlin's writeup in the projects section of the C Tutorial Flashcards. The description of Huffman encoding is very similar, but said slightly differently. It sometimes helps to have a different description of an idea. There is one difference, and that's using a BitStream rather than a BitSequence as described here.

- Consider doing the warm-up exercises at the end of this write-up. They should help get you started.
- Feel free to use the `string.h` library.
- Use the defined constants
- Write your own tests! Because this quiz is less structured, we aren't able to write tests for every piece, because your implementation can vary. You know what functions you're writing, and what you expect them to do. Feel free to add tests that prove each function does what you expect it to do.
- If you are struggling with breaking your project down into smaller pieces, talk to an instructor/TA. Spend some time thinking about it, but feel free to ask for guidance on this part.
- Write "print" helper functions. Again, we can't write them all for you, because your implementation might vary. We've provided print functions where we could. Use those as guidance, and ask for help.
- Use `unsigned short` consistently. Sometimes it matters– an `unsigned` number could be considered `unsigned long` (4 bytes instead of 2— a 32 bit number, rather than 16).
- Specifically for creating the `HTree`, consider other data structures we've seen, and feel free to create/implement one to help with the process of creating the `HTree`.
- Design and implement your own structs that help you solve the problem.
- Start early. Take small steps.

It might help to approach the quiz in this order:

1. Write the create/destroy functions for all structs.
2. Implement `BitSeq`:
   (a) Print a `BitSeq`
   (b) Add bits to a `BitSeq`
   (c) Get a single bit from a `BitSeq`
3. Implement `FreqTable`:
   (a) print a `FreqTable`
   (b) populate the `FreqTable`
4. Implement `HTree`
5. Implement `decode()`
6. Implement `encode()`

6 Warm-Up Exercises

This section contains some exercises to get you started and familiar with bit manipulation and tree traversal.

These are not part of the actual quiz, and do not contribute to your final grade, but could be helpful for getting comfortable with bit manipulation and tree traversal if you are not yet comfortable.

6.1 Bit Manipulation

When starting to work with bit manipulation, it's helpful to have a way to print out the bits of a value. The following code (available in `warmup.c`) prints out the bits of an `unsigned` value in C.

> **Side note:** `unsigned` means the bits represent an positive integer between 0 and 2^n, where n is the number of bits. Typically, a 16-bit integer is stored in 2's-complement form, which gives it a range that is positive and negative, centered around 0.

Make sure the following code makes sense to you.

```
1  void displayBits(unsigned value){
2    unsigned c;
3    // Take 1, shift it all the way to the left.
4    unsigned displayMask = 1 << 15;
5
6    printf("%3c%7u␣=␣", value, value);
7
8    for(c=1; c<=16; c++){
9      // putchar prints a character on the screen
10     // 'value & displayMask' gets the relevant bit
11     putchar(value & displayMask ? '1' : '0');
12
13     // Shift value over one
14     value <<= 1;
15
16     // Put a space in for easy reading
17     if (c % 8 == 0){
18       putchar('␣');
19     }
20   }
21   putchar('\n');
22 }
```

To print out the bit values, we're defining a bit mask (`displayMask`) that determines if the bit in the first spot on the left is 1 or not. When it's AND'd with the original value, the output will be 1 if the left most bit is 1, or 0 otherwise.

After a bit is printed, the value is bit-shifted left, meaning all bits are shifted over one, and the right-most bit is filled in with a 0.

The table below shows the mask that never changes throughout the call to the function, and how the value changes throughout the loop, when the original value is 'B' (or 173) (as shown in the first line).

Again, note that the a char is usually 16 bits, but we're only showing the last (right) 8 bits.

Try implementing at least `packCharacters()`.

1. **`packCharacters`:** The left shift operator (<<) can be used to pack two character values into a 2-byte unsigned integer variable. Write a function `packCharacters` that takes two characters as a parameter.

   A Hint: To pack two characters into an unsigned integer variable, assign the first character to the to the unsigned variable, shift the unsigned variable left by 8 bit positions, and combine the unsigned variable with the second character using the bitwise inclusive OR operator. The function should print the characters in their bit format before and after they are packed into the unsigned integer to prove that the characters are in fact packed correctly in the unsigned variable.

Input: 'B'	0	1	0	0	0	0	1	0
mask	1	0	0	0	0	0	0	0
'B'<<0: value_1	0	1	0	0	0	0	1	0
'B'<<1: value_2	1	0	0	0	0	1	0	0
'B'<<2: value_3	0	0	0	0	1	0	0	0
'B'<<3: value_4	0	0	0	1	0	0	0	0
'B'<<4: value_5	0	0	1	0	0	0	0	0
'B'<<5: value_6	0	1	0	0	0	0	0	0
'B'<<6: value_7	1	0	0	0	0	0	0	0
'B'<<7: value_8	0	0	0	0	0	0	0	0

Example: Printing the output of `packCharacters('B', 'i')` should result in:

```
01000010 01101001
```

2. **`unpackCharacters:`** Using the right shift operator, the bitwise `AND` operator, and a mask, write function `unpackCharacters` that takes the `unsigned integer` from above and unpacks it into two characters. As a note: 65280 $\rightarrow$ 11111111 00000000. This might be helpful as a mask for this function. Print the input integer and output characters.
3. **`power2:`** Left shifting an `unsigned integer` by 1 bit is equivalent to multiplying the value by 2. Write function `power2` that takes two integer arguments, `number` and `pow` and calculates ($number * 2^{pow}$). Use the shift operator to calculate the result. The program should print the values as integers and as bits.

6.2 Tree Traversal

1. Given the code in `warmup.c`, fill in the function that prints out each node of the tree in a depth-first, post-order manner. For the given tree below, write down what you think the correct depth-first, post-order traversal of the tree is. Then, write your code and compare the results.