University of Virginia Computer Science
CS216: Program and Data Representation, Spring 2006
http://www.cs.virginia.edu/cs216/ps/ps4/
13 March 2006

[PDF version for printing]

Problem Set 4
Huffman Coding

Out: 13 March
Due: Wednesday (beginning of class), 22 March

Collaboration Policy - Read Carefully

For this assignment, you may work on your own or with any one other person of your choice **except for anyone you worked with on PS3**. If you work with a partner, you should turn in one assignment with both of your names on it. **If you would prefer to be assigned a partner, send email to *evans@cs.virginia.edu* before 5pm on Monday, 13 March** (include any constraints or preferences you have on your assigned partner). If a suitable match requests a partner, you will receive a partner assignment.

You may consult any outside resources including books, papers, web sites and people you wish, except you may not copy code from other Huffman encoding implementations. There are many implementations of Huffman encoding and decoding available on the web, and it would certainly defeat the purpose of this assignment if you copied one of them instead of thinking on your own. You are also encouraged to discuss these problems with students in the class. You must acknowledge any people and outside resources you work with on your assignment. If you discuss the assignment with people other than your partner, you may not take any written materials out of your discussion. It is fine to bounce ideas off other people, but the answers you turn in must be your own.

You are **strongly encouraged** to take advantage of the staffed lab hours posted on the CS216 web site.

Purpose

- Gain some experience with low-level programming.
- Learn to use pointers in C.
- Understand explicit memory management.
- Develop an understanding of how numbers are represented in memory and bit-manipulation techniques.
- Implement a Huffman encoder/decoder and understand how it works and why it is optimal.

> **Reading:** Read Chapters 10 and 5 in the textbook. (We recommend reading chapter 10 first.)

You may also find one or more of these C references useful:

- *Learning C from Java*
- Steve Summit's C Programming Notes
- University of Leicester's *Introduction to C Programming*

If you don't find these useful, there are many books on C programming available. You should be able to complete this assignment, however, without needing to become an expert C programmer.

Defined or Undefined

For each of the following 3 questions, you are given a short fragment of C code. If the behavior of the fragment is *undefined*, explain why. Otherwise, describe the output of the fragment.

1.

```
{
   char *s = (char *) malloc (sizeof (*s));
   s[0] = 'a';
   printf ("%c", *(s + 1));
}
```

2.

```
{
   char *s = (char *) malloc (sizeof (*s) * 6);
   char *t;
   strcpy (s, "cs216");
   t = s;
   free (t);
   printf ("%s", s);
}
```

3.

```
char *select (int v, char **s)
{
   if (v) return *s;
   return 1[s];
}

int main (int argc, char **argv)
{
   char **s = (char **) malloc (sizeof (*s) * 2);
   char *t1 = (char *) malloc (sizeof (*t1) * 6);
   char *t2 = (char *) malloc (sizeof (*t2) * 6);
   char *p;

   s[0] = t1;
   s[1] = t2;
   s[0][0] = 'b';
   p = select (0, s);
   p[0] = 'a';

   printf ("%c", **s);

}
```

Huffman Encoding

Download: ps4.zip.

> Huffman worked on the problem for months, developing a number of approaches, but none that he could prove to be the most efficient. Finally, he despaired of ever reaching a solution and decided to start studying for the final. Just as he was throwing his notes in the garbage, the solution came to him. "It was the most singular moment of my life," Huffman says. "There was the absolute lightning of sudden realization."
> From Scientific American's profile of David Huffman

For the rest of this assignment, you will understand and complete an implementation of Huffman encoding and decoding. Huffman encoding was developed in a term paper David Huffman wrote instead of taking the final exam in an information theory course, and is now used in many applications including MPEG and MP3.

The provided code and directions assume you are using Visual Studio, as installed in the ITC labs. You are free to use any C compiler you want, but if you run into problems with some other compiler the course staff may not be able to help you.

Getting Started with Visual Studio

Download ps4.zip and unzip it in your home directory. The download file ps4.zip contains a Visual Studio Solutions file and source code for part of the Huffman encoding implementation.

Click on the file `PS4.sln`. Visual Studio should open, and you will see a view showing the `huffman.c` source file. The right side of the window shows all the source files in the project. There are three files:

- `huffman.c` — defines `main (int, char **)` (the function that is called at the start of execution), and `readInputFile (FILE *)` for reading all the characters in a file into a string.
- `htree.c` — defined procedures for implementing Huffman encoding and decoding
- `htree.h` — a header file that defined the `htree` datatype we use to implement a Huffman tree

The provided code does not yet do anything useful, but try building at by selecting `Build | Build Huffman` from the top menu. You should see the build complete. The resulting executable is `Debug\huffman.exe`.

Try running it from the Windows shell:

```
K:\cs216\ps4\Debug>huffman
Usage: huffman [-d] [-b] <input> [<output>]
```

The `-d` option is used for decoding. The `-b` option is used to select bit-encoding (instead of printing `0` and `1` as characters in the output file). We will use the character encoding until question 9.

Representing Huffman Trees

To represent a Huffman encoding tree, we need a datatype similar to a binary tree. The type is defined in `htree.h`:

```
typedef struct _htree {
  struct _htree *left;
  struct _htree *right;
  struct _htree *parent;
  int count;
  char letter;
} *htree;
```

Because C is designed to be compilable by a one-pass compiler and all types must be declared before they are used, declaring a recursive datatype in C is a bit awkward. Here, we use `struct` to create a structure datatype consisting of the five fields showing. The `typedef` defines the `htree` datatype as a pointer to the `struct _htree` structure. The `left`, `right`, and `parent` fields of `struct _htree` are themselves `htree` objects (but we have to use `struct _htree *` instead, since `htree` is not yet defined). Each tree node maintains a `count` (integer that represents the weight of that node, $C(n)$ in the book), and `letter` (which is meaningful only for leaves).

> **4.** The procedure `htree_unparse` in `htree.c` takes as input a `htree` object and returns a string representation of that encoding tree. The goal here is to produce a machine-readable string, not a human-readable string, so the string is not divided into lines. The `htree_unparse` defined in `htree.c` produces the correct string, but it leaks memory. Add the necessary calls to `free` in `htree_unparse` to plug the memory leak.

Building the Huffman Tree

Section 5.4 of the text explains how to build a Huffman encoding tree. The provided implementation of `htree_buildTree` produces a (very) non-optimal encoding tree. It satisfies the necessary properties for encoding and decoding to work correctly, but places letters in the tree sequentially (without considering their frequency in the input text). We have provided this implementation so you can make progress on the other questions even if you are not able to implement the optimal Huffman encoding tree. You may find parts of the provided code a useful starting point, but you are not required to use it. The provided `htree_print (htree)` routine may be useful to examine the encoding tree your code produces.

> **5.** Implement the `htree_buildTree` procedure that takes as input a string and returns the optimal Huffman encoding tree for that string.

Examine the encoding trees your `htree_buildTree` produces for different input files. You can use the `htree_print` procedure provided in `htree.c` to print out the resulting encoding tree in a (more or less) human readable way.

> **6.** Construct input files that produce Huffman encoding trees with the properties described in each sub-question. For each part, include both your input file and the output Huffman encoding tree your produced in your answer. (Note, you do not need to use the full alphabet for any of these questions. Your input file should use as few symbols as possible to satisfy the property.)
>
> **a.** A tree where the letter `A` is encoded using one bit.
>
> **b.** A tree where each letter is encoded using exactly three bits.
>
> **c.** A tree where a letter requires more than 6 bits to encode.

Encoding and Decoding

We have provided the `htree_encodeChars` routine that takes a null-terminated string to be encoded and an output file, and writes a Huffman-encoded string equivalent to the input string to the output file.

It uses `htree_encodeChar` to obtain a string representation of the Huffman encoding for each character in the input string, and writes it to the output file. Note that we are using a string like "01001" to encode a five-deep character. Since we are using strings, it will be easy to read the output file, but not very useful as a compressor! Each character's encoding expands to 8 * *tree depth* since we are using a full byte to represent each bit in the encoding. (In Questions 9 and 10 you will modify the encoding to only use one bit per encoding bit, instead of a full byte.)

7. What is the asymptotic running time of our `htree_encodeChars` procedure? You may assume the input string is long enough that the time taken to produce the Huffman encoding tree does not matter (so you do not have to consider the running time of `htree_buildTree` and `htree_unparse` in your answer).

8. The provided `htree_encodeChars` procedure is very inefficient. Explain how it could be implmented with running time in $O(n)$ where n is the number of characters in the input string s. (You don't need to modify the code, just explain the basic idea.)

We have provided a partial implementation of `htree_decodeChars`. It takes care of reading in the Huffman encoding tree, but is missing the code needed to decode the encoded file.

9. Complete the implementation of `htree_decodeChars`. We have provided some code that you may find useful in `htree.c`, but you can change the implementation however you want. (If you are stuck on this question, you may find it useful to examine the provided `htree_decodeBits` routine.)

When your implementation is correct, you should be able to encode and decode any file and get the original result:

```
> huffman test.txt test.hcode
> huffman -d test.hcode
```

should ouput the original contents of `test.txt`.

Complete the implementation of `htree_decodeChars`. We have provided some code that you may find useful in `htree.c`, but you can change the implementation however you want. (If you are stuck on this question, you may find it useful to examine the provided `htree_decodeBits` routine.)

Bit Manipulation

As pointed out earlier, our Huffman encoder is not very useful: instead of compressing the original file into a smaller file, it creates a file that is several times the size of the input file. This is because instead of using one bit in the file to represent each bit in the character encoding, we are using a full byte so we can output the readable character `0` or `1`.

For these questions you will implement an encoder that uses one bit to represent each encoding bit instead. We have provided an almost complete implementation of the `htree_decodeBits` routine that will decode a bit-Huffman-encoded file.

C provides bitwise operators for manipulating bytes at the bit-level. For example, the `&` operator performan a bitwise **and**. The i^{th} bit of `a & b` is `1` if and only if the i^{th} bit of `a` and the i^{th} bit of `b` are both `1`.

However, C does not provide a direct bit datatype or a way to write a single bit to a file. Instead, we use the `unsigned char` datatype, which is a byte (8 bits), and must write to the file one byte at a time. The causes some complication at the end of the file, if we are not on an even byte boundary. The solution assumed by our `htree_decodeBits` is that the very last byte of the file represents a number which gives the number of bits in the next-to-last byte that are valid. For example, if the file ends with a 3-bit partial code, we will output a full byte with the first 3 bits being the code and the remaining 5 bits of that byte undetermined, and a final byte with value 3 that indicates that only the first 3 bits of the next-to-last byte are valid.

10. Complete the implementation of `htree_decodeBits` by finishing the assignment to `bit` (marked with `/* Question 10 ...`). The value of `bit` should be zero if the `i`th bit of `c` is zero, and non-zero if the `i`th bit of `c` is one.

You can check if your implementation is correct by trying the test binary-encoded files `alphabet.bh` and `declaration.bh` included in `ps4.zip`:

```
> huffman -bd alphabet.bh
abcdefghijklmnopqrstuvwxyz
> huffman -bd declaration.bh
displays the Declaration of Independence
```

Note that our implementation now is an effective compressor. The original `declaration.txt` is 8586 bytes, but the Huffman-encoded file is 5123 bytes.

11. Implement the `htree_encodeBits` routine. If your implementation is correct, you should be able to decode an encoded file to produce the original result. (Note: it is not necessary to complete question 10 to reach the "green" star level on this assignment, if you answer questions 1-9 well.)