Problem Set 6: Huff'n Puff

out of 60 points

due by 7:00 P.M. on Friday, 30 November 2007

HACKER EDITION

Goals.

The goals of this problem set are to:

- Huff and puff and blow your house in.
- Build and traverse binary trees using pointers.
- Introduce some new tools.
- Wrap up your exposure to C.

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this class that you have submitted or will submit to another. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

All forms of cheating will be dealt with harshly.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the staff.

Grading Metrics.

Each question is worth the number of points specified parenthetically in line with it.

Your responses to questions requiring exposition will be graded on the basis of their clarity and correctness. Your responses to questions requiring code will be graded along the following axes.

> **Correctness.** To what extent does your code adhere to the problem's specifications?
> **Design.** To what extent is your code written clearly, efficiently, elegantly, and/or logically?
> **Style.** To what extent is your code commented and indented, your variables aptly named, *etc.*?

Rest assured that grades will be normalized across sections at term's end.

HACKER EDITION

Getting Started.

0. SSH to `nice.fas.harvard.edu` and recursively copy `~cs50/pub/ps/distributions/ps6` into your own `~/cs50` directory.

   Navigate your way to `~/cs50/ps6/`. If you list the contents of your current working directory, you should see the below. If you don't, don't hesitate to ask the staff for assistance.

```
dumper.c  forest.h  huffile.c  Makefile  tree.h
forest.c  hth.txt   huffile.h  tree.c
```

The Story of the Three Little Pigs.

1. Once upon a time when pigs spoke rhyme
   And monkeys chewed tobacco,
   And hens took snuff to make them tough,
   And ducks went quack, quack, quack, O!

2. There was an old sow with three little pigs, and as she had not enough to keep them, she sent them out to seek their fortune. The first that went off met a man with a bundle of straw, and said to him:

   "Please, man, give me that straw to build me a house."

   Which the man did, and the little pig built a house with it. Presently came along a wolf, and knocked at the door, and said:

   "Little pig, little pig, let me come in."

   To which the pig answered:

   "No, no, by the hair of my chiny chin chin."

   The wolf then answered to that:

   "Then I'll huff, and I'll puff, and I'll blow your house in."

   So he huffed, and he puffed, and he blew his house in, and ate up the little pig.

   The second little pig met a man with a bundle of furze, and said:

   "Please, man, give me that furze to build a house."

   Which the man did, and the pig built his house. Then along came the wolf, and said:

HACKER EDITION

"Little pig, little pig, let me come in."

"No, no, by the hair of my chiny chin chin."

"Then I'll puff, and I'll huff, and I'll blow your house in."

So he huffed, and he puffed, and he puffed, and he huffed, and at last he blew the house down, and he ate up the little pig.

The third little pig met a man with a load of bricks, and said:

"Please, man, give me those bricks to build a house with."

So the man gave him the bricks, and he built his house with them. So the wolf came, as he did to the other little pigs, and said:

"Little pig, little pig, let me come in."

"No, no, by the hair of my chiny chin chin."

"Then I'll huff, and I'll puff, and I'll blow your house in."

Well, he huffed, and he puffed, and he huffed and he puffed, and he puffed and huffed; but he could *not* get the house down. When he found that he could not, with all his huffing and puffing, blow the house down, he said:

"Little pig, I know where there is a nice field of turnips."

"Where?" said the little pig.

"Oh, in Mr. Smith's Home-field, and if you will be ready tomorrow morning I will call for you, and we will go together, and get some for dinner."

"Very well," said the little pig, "I will be ready. What time do you mean to go?"

"Oh, at six o'clock."

Well, the little pig got up at five, and got the turnips before the wolf came (which he did about six) and who said:

"Little Pig, are you ready?"

The little pig said: "Ready! I have been and come back again, and got a nice potful for dinner."

The wolf felt very angry at this, but thought that he would be up to the little pig somehow or other, so he said:

HACKER EDITION

"Little pig, I know where there is a nice apple-tree."

"Where?" said the pig.

"Down at Merry-garden," replied the wolf, "and if you will not deceive me I will come for you, at five o'clock tomorrow and get some apples."

Well, the little pig bustled up the next morning at four o'clock, and went off for the apples, hoping to get back before the wolf came; but he had further to go, and had to climb the tree, so that just as he was coming down from it, he saw the wolf coming, which, as you may suppose, frightened him very much. When the wolf came up he said:

"Little pig, what! are you here before me? Are they nice apples?"

"Yes, very," said the little pig. "I will throw you down one."

And he threw it so far, that, while the wolf was gone to pick it up, the little pig jumped down and ran home. The next day the wolf came again, and said to the little pig:

"Little pig, there is a fair at Shanklin this afternoon, will you go?"

"Oh yes," said the pig, "I will go; what time shall you be ready?"

"At three," said the wolf. So the little pig went off before the time as usual, and got to the fair, and bought a butter-churn, which he was going home with, when he saw the wolf coming. Then he could not tell what to do. So he got into the churn to hide, and by so doing turned it round, and it rolled down the hill with the pig in it, which frightened the wolf so much, that he ran home without going to the fair. He went to the little pig's house, and told him how frightened he had been by a great round thing which came down the hill past him. Then the little pig said:

"Hah, I frightened you, then. I had been to the fair and bought a butter-churn, and when I saw you, I got into it, and rolled down the hill."

Then the wolf was very angry indeed, and declared he *would* eat up the little pig, and that he would get down the chimney after him. When the little pig saw what he was about, he hung on the pot full of water, and made up a blazing fire, and, just as the wolf was coming down, took off the cover, and in fell the wolf; so the little pig put on the cover again in an instant, boiled him up, and ate him for supper, and lived happy ever afterwards.

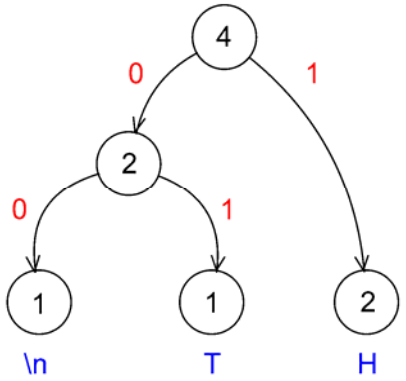
HACKER EDITION

3. Ahem, as soon as you're done reading fairy tales, we have some work for you.

4. The challenge ahead is to implement a program called `huff` that huffs (*i.e.*, Huffman-compresses) files that can then be puffed (*i.e.*, decompressed) with a program that we wrote called `puff`. Let's begin with a story of our own.

5. Once upon a time, there were four little pigs who lived in a four-byte ASCII file. The first little piggy was an `H`. The second little piggy was a `T`. The third little piggy was an `H`. And the fourth little piggy was a newline.

   Presently came along David A. Huffman, and made a tree out of the piggies' frequencies, per the figure below.



   In a file called `ps6/tale.txt`, finish this tale if feeling creative or silly.[1]

   The course's website shall tell the best tales.

6. When represented in ASCII, each of those piggies takes up 8 bits on disk. But, thanks to Huffman, we can generally do better. After all, how many bits does it really take to represent any of three different characters? Just two, of course, as two bits allows us as many as $2^2 = 4$ codes. And so could we represent, per the figure above, a newline with `00`, `T` with `01`, and `H` with `1`. Notice how, even in this tiny example, the least frequent of characters receive, by design, the longest of codes.

   The catch, of course, is that you must be able to reconstruct this tree (or, more generally, recover these codes) if you wish to puff back to ASCII piggies that have been huffed. Perhaps the simplest way to enable a program like `puff` to decompress files that have huffed is to have `huff` include in those files piggies' original frequencies. With those frequencies can `puff` then re-build the same tree that `huff` built. Of course, inclusion of this metadata does cost us some space. But, for large inputs, that cost is more than subsumed by savings in bits.

   Let's get you started on `huff`.

7. Open up `huffile.h` and spend some time looking over the code and comments therein. This file defines a "layer of abstraction" for you in order to facilitate your implementation of `huff` (and our implementation of `puff`). More technically, it defines an API (application programming interface) with which you write (or read) Huffman-compressed files.

   Ultimately, this problem set is as much about learning how to interface with someone else's code (*e.g.*, ours) as it is about building and traversing binary trees. After all, after CS 50, you

[1] Or, better yet, if you'd like to procrastinate.

HACKER EDITION

won't always have someone to walk you through code. But what once looked like Greek should at least now look like C to you.

Notice that, in `huffile.h`, we have defined the following `struct`.

```
typedef struct
{
    int magic;
    int frequencies[SYMBOLS];
    int checksum;
}
Huffeader;
```

As its own name suggests (say it three times fast), this `struct` defines a header for a Huffman-compressed file (much like `BITMAPFILEHEADER` defined a header for BMPs). Before writing out bits (*i.e.*, codes), your implementation of `huff` must first write out this header, so that our version of `puff` can read in the same and reconstruct the tree you used for huffing.

Besides symbols' frequencies, notice that this header includes some magic! Much like JPEGs begin with `0xffd8`, so have we decided (arbitrarily) that `huff`-compressed files must begin with `0x46465548`.[2] A "magic number," then, is a form of signature. We have also decided that huffed files' headers must end with a "checksum," a summation of all frequencies therein.

In other words, if, upon reading some file's first several bytes into a `Huffeader`, `magic` is not `0x46465548` or `checksum` does not equal the sum of all values in `frequencies`, then that file was most certainly not huffed![3]

8. Take a look now at `hth.txt`, but take care not to make any changes. In that file are those four little piggies (even though your text editor might not show you the newline). Let's blow their house down and compress them with our own implementation of `huff`. Run the below to save a compressed version of `hth.txt` in a new file called `hth.bin`.

```
~cs50/pub/ps/solutions/ps6/huff  hth.txt  hth.bin
```

Let's take a look at the huffed file's size. Run the below.

```
ls -l hth.txt hth.bin
```

Ack! Per that command's output, it seems that we have "compressed" 4 bytes to 1034! Such is the cost of that metadata for particularly small files. For larger inputs, though, it won't be so bad.

HACKER EDITION

[2] We say "arbitrarily," but `0x46465548` actually has meaning. What does it spell?

[3] Of course, some non-huffed file's first several bytes might happen to satisfy these conditions as well, in which case it could be mistaken for a huffed file. Probabilistically, that's not too likely to happen. But it's because of that chance that some operating systems (also) rely on files' extensions (*e.g.*, `.bmp`) to distinguish files' types.

Incidentally, if you never quite understood SFTP's distinction between ASCII and binary files, you hopefully will now: `hth.txt` is an ASCII file because it contains ASCII codes, and `hth.bin` is a binary file because it does not! That we've chosen extensions of `.txt` for the former and `.bin` is just for convenience and not by requirement.

Let's take a look at the contents of `hth.bin` in hex with an old friend. Run the below.

```
xxd -g 4 hth.bin
```

Scroll back on up to the start of `xxd`'s output. Take a look at this huffed file's first four bytes! Wait a minute, talk about magic, those bytes are reversed! (And, yes, they do spell `HUFF` if you insist on interpreting those bytes as ASCII, as `xxd` does in its rightmost column. So clever we are.) Recall that a huffed file's first four bytes were supposed to be `0x46465548`, not the reverse. So what's going on?

It turns out that `nice.fas.harvard.edu`, because of its Intel processors, is "little endian," whereby multi-byte primitives (like `int`) are stored with their little end (*i.e.*, least-significant byte) first. Generally speaking, you need not worry about endianness when programming, unless you start manipulating binary files (or network connections). We mention it now so that you understand `xxd`'s output!

Notice, by the way, how many `0`s are in `hth.bin`. Of course, `hth.txt` only had three unique piggies, so most of those `0`s represent the frequencies of ASCII's other (absent) 253 characters. But, if you look closely, scattered among all those `0`s are `01000000`, `02000000`, and `01000000`, which are, of course, little-endian representations of 1, 2, and 1 (in decimal), the frequencies of newlines, `H`, and `T`, respectively, in `hth.txt`! Lower in `xxd`'s output you'll find `04000000`, the sum (*i.e.*, checksum) of those counts. The second-to-last byte in `hth.bin` appears to be `b0` and the very last `06`. Hm, back to those in a bit.

9. Next take a look now at `dumper.c`. That file implements a program with which you can dump `huff`-compressed files in human-readable form. Look over its comments and code to learn how it works.

   Next take a look at `Makefile`, in which we've defined a target for `dumper` but not one for `huff`. (We'll leave that to you.) Notice how `dumper` depends not only on `dumper.c` but also on other `.c` and `.h` files as well. That `dumper.c` itself appears relatively simple is because we have abstracted away important, but potentially distracting, details with APIs.

   Go ahead and build `dumper` with Make. Then run it as follows.

   ```
   dumper hth.bin
   ```

HACKER EDITION

You should see output like the below.

```
! 0       - 0       9 0       E 0       Q 0       ] 0       i 0       u 0
" 0       . 0       : 0       F 0       R 0       ^ 0       j 0       v 0
# 0       / 0       ; 0       G 0       S 0       _ 0       k 0       w 0
$ 0       0 0       < 0       H 2       T 1       ` 0       l 0       x 0
% 0       1 0       = 0       I 0       U 0       a 0       m 0       y 0
& 0       2 0       > 0       J 0       V 0       b 0       n 0       z 0
' 0       3 0       ? 0       K 0       W 0       c 0       o 0       { 0
( 0       4 0       @ 0       L 0       X 0       d 0       p 0       | 0
) 0       5 0       A 0       M 0       Y 0       e 0       q 0       } 0
* 0       6 0       B 0       N 0       Z 0       f 0       r 0       ~ 0
+ 0       7 0       C 0       O 0       [ 0       g 0       s 0
, 0       8 0       D 0       P 0       \ 0       h 0       t 0

101100
```

Atop `dumper`'s output is a table of frequencies, not for all ASCII characters but for those that display nicely in terminal windows. Notice that the frequencies of `H` and `T` are as expected. (Newlines are simply not among the characters shown.)

Below that table, meanwhile, is a sequence of six bits, the compressed version of `hth.txt`! Recall, after all, that our tree told us to represent newline with `00`, `T` with `01`, and `H` with `1`. And, so, the above indeed represents our original text!

Let's take one more look at this file with `xxd`, this time in binary. Try the below.

```
xxd -b hth.bin
```

Take a close look at `hth.bin`'s final two bytes: `10110000` and `00000110`. (You may recall these bytes as `b0` and `06` in hex.) Notice how the former is but `101100` padded with two trailing `0`s. Why those two `0`s? Well, you can write individual bytes to disk but not individual bits. Ergo, even though our implementation of `huff` only called `bwrite` six times in order to write out six bits, our API ultimately has to write out eight bits. To avoid confusion when it's time to read those bits back in, our API employs a trick. We keep track, in a huffed file's very last byte, of just how many bits in the file's second-to-last byte are valid so that `bread` can avoid returning trailing padding lest `puff` mistake extra `0`s for encoded symbols!

Take a look at `huffile.c`. As is the case with most APIs, you don't need to understand how our API works underneath the hood. But, seeing as you're doing the Hacker Edition of this problem set, you probably should. Focus in particular on our manipulation of bits. May that you learn a trick or two by our example!

HACKER EDITION

10. A final stroll through some code, if we may. Recall that, to implement Huffman's algorithm, you can begin with a "forest" of single-node trees, each of which represents a symbol and its frequency within some body of text. Iteratively can you then pick from that forest the two trees with lowest frequencies, join them as siblings with a new parent whose own frequency is the sum of its children's, and plant that new parent in the forest. In time will this forest converge to a lone tree whose branches represent symbols' codes.

    Also recall that the manner in which ties between roots with equal frequencies are broken is important to standardize, lest `huff` and `puff` build different trees. And so we have provided you not only with an API writing (or reading) `Huffiles` but also with an API for forest management.[4] Take a look first at `tree.h`. Notice that we have provided the following definition for trees' nodes.

```
typedef struct tree
{
    char symbol;
    int frequency;
    struct tree *left;
    struct tree *right;
}
Tree;
```

    Rather than store symbols' frequencies as percentages (*i.e.*, floating point values), a node, per this definition, instead stores raw counts.

    As the design of `tree.h` suggests, rather than ever `malloc` a `Tree` yourself, you should instead call `mktree`, which will not only `malloc` a `Tree` for you but also initialize its members to defaults. Similarly should you never call `free` on a `Tree` but, instead, invoke `rmtree`, which will delete that `Tree`'s root for you plus all its descendants.[5]

    Now take a look at `forest.h`. This API happens to implement a `Forest` as a linked list of `Plots`, each of which houses a `Tree`. But you need not worry about such details, as we have abstracted them away for the sake of simplicity (and standardization). Rather than ever `malloc` or `free` a `Forest` yourself, you should instead, much like for `Trees`, call `mkforest` or `rmforest`, respectively. Moreover, rather than ever touch a `Forest`'s linked list, you should instead add `Trees` to a `Forest` with `plant` and remove `Trees` from a `Forest` with `pick`. Note that this API does not build Huffman's tree for you! Rather, it maintains `Trees` that you yourself have planted in sorted order so that you can pick those same `Trees` in order of increasing frequency, with the API (and not you) breaking ties when necessary.

    If curious as to how this all works, take a look at `tree.c` and `forest.c`.

HACKER EDITION

[4] Speaking of forest management, did you know that Harvard owns a forest? Procrastinate at `http://harvardforest.fas.harvard.edu/`.

[5] If familiar with, say, C++, you can think of `mktree` as a sort of constructor and `rmtree` as a sort of destructor.

11. (60 points.) Implement a program called `huff` that compresses files! Allow us to put forth the following requirements.[6]

    i. Your program must accept two and only two command-line arguments: the name of a file to huff followed by the name under which to save the huffed output.
    ii. You must build Huffman's tree using our APIs for `Forests` and `Trees`. That tree must not include nodes for symbols not appearing in the file to be huffed.
    iii. After picking two trees from a forest in order to join them as siblings with a new parent, the first tree picked should become the parent's left child, the second the parent's right.
    iv. Assume that left branches represent `0`s and right branches `1`s.
    v. If huffing a file that contains but one unique symbol, assume the symbol's code is just `0`.
    vi. You must write out a `Huffeader` plus bits using our API for `Huffiles`.
    vii. You must handle all possible errors gracefully (*e.g.*, with error messages); under no circumstances should we be able to crash your code.
    viii. You may not leak any memory.[7] Before quitting, even upon error, your program must free any memory allocated on its heap, either with `free` or, if allocated by our APIs, with `hfclose`, `rmtree`, and/or `rmforest`.
    ix. You must update `Makefile` (however you see fit) with a target for `huff`.[8]

    How to determine if your code is correct? Well, certainly play with the staff's solutions to both `huff` and `puff` in `~cs50/pub/ps/solutions/ps6/`, comparing our output to yours. Also use `ls` with its `-l` switch to compare files' sizes. And, rather than compare outputs visually (*e.g.*, with Nano, Vim, Emacs, `xxd`, `cat`, `more`, or `less`), you can use a popular tool called `diff`. For instance, suppose that you've already run the below.

    ```
    huff  hth.txt  hth.bin
    ```

    And now you'd like to try puffing `hth.bin` with the staff's implemenation of `puff`, and so you run a command like the below.

    ```
    ~cs50/pub/ps/solutions/ps6/puff  hth.bin  mine.txt
    ```

    You can now compare `hth.txt` and `mine.txt` for differences by executing the below.

    ```
    diff  hth.xt  mine.txt
    ```

HACKER EDITION

[6] If, as an aspiring hacker, you would rather not use our APIs or you would like to modify them, you may, provided you inform your teaching fellow to that effect. If you format huffed files in a manner inconsistent with the staff's implementation of `puff`, you must also implement your own version of `puff` so that we can puff your compressed files.
[7] Note that your implementation of `speller` for Problem Set 5 likely leaked memory, because we did not provide means for freeing memory allocated by your dictionary, as via a call to some function called, say, `unload`, at the bottom of `main`. Consider that my fault and my bug! We expect better of you. `:-)`
[8] Recall that a target's second line must begin with a tab. When you hit Tab in Nano, though, you do not get `\t` but instead four spaces. (Spaces tend to print better than tabs.) To insert a true tab using Nano, hit Esc followed by v or Alt-v to provide "verbatim input." Then can you hit Tab to insert `\t`. Incidentally, to "see" whitespace with Nano, hit Esc followed by p or Alt-p. Spaces will suddenly appear as dots (·), and tabs as arrows (»). Hit that same sequence again to hide those same symbols. Similarly does Vim treat Tab as four spaces. To insert a true tab using Vim (while in insert mode), hit Ctrl-v followed by Tab.

If those the files are identical, then `diff` will say nothing! Otherwise it will report lines with differences.

Of course, best to test `huff` with more than just `hth.txt`. Odds are, you have a whole bunch of text files lying around from Problem Set 5 that you can huff with your `huff` and puff with our `puff`! In theory, you can huff binary files as well, even though (conceptually, at least) Huffman's algorithm is meant for ASCII files.

And how can you chase down memory leaks? Well, you know your code best, so certainly think about where your own code might leak. Focus, in particular, on any blocks of code in which your code might return prematurely (as in the case of some error); it's not likely sufficient to free up your heap only, say, at the very end of `main`.

But also take advantage of a tool called Valgrind, whose job is to report memory-related mistakes and, in particular, leaks. Run it with a command like the below.

```
valgrind huff hth.txt hth.bin
```

Valgrind's output is a bit cryptic, but keep an eye out for `ERROR SUMMARY` and, possibly, `LEAK SUMMARY`. For additional hints, run it with some optional switches, per the below.

```
valgrind -v --leak-check=full huff hth.txt hth.bin
```

If uncertain how to interpret its output, simply contact the staff!

Don't forget to check in your code often with RCS and to debug it with GDB!

Alright, off you go. HTH!

Submitting Your Work.

12. Ensure that your work is in `~/cs50/ps6/`. Submit your work by executing the command below.

```
cs50submit ps6 hacker
```

Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.

HACKER EDITION