Word histograms

This assignment will explore simple, unbalanced, binary search trees of strings. You will create a program, `hist`, which produces a histogram of its command-line arguments; that is, a table showing the number of times each string appeared on the command line, sorted alphabetically. For example:

```
$ ./hist foo bar foo foo quux blah bar
   2    bar
   1    blah
   3    foo
   1    quux
$ ./hist x y z z y
   1    x
   2    y
   2    z
$ ./hist
$ ./hist a a a a a a a a a a a a a a
  14    a
```

To maintain the counts, and keep the strings sorted, you will implement a very simple tree structure according to this header:

tree.h

```
1  struct tree {
2          struct tree *left;
3          struct tree *right;
4          char *value;
5          unsigned count;
6  };

8  struct tree *tree_add(struct tree *tree, char *value);
9  void tree_dump(struct tree *tree);
10 void tree_free(struct tree *tree);
```

A `struct tree *` represents the root of a binary search tree of strings. Its `left` member is its left sub-tree, or `NULL` if there is none; similarly for `right`. The string itself is `value`, and the invariant is that `value` is greater than all of the `value`s down the left sub-tree, and less than all of the `value`s down the right sub-tree. (You will certainly want the `strcmp` function, from `string.h`, to determine which of two strings is greater or if they are equal.) If a `value` is added to the tree multiple times, create only one node for it, but increment `count` to track the number of appearances.

The `tree_add` function takes an existing tree (possibly `NULL`, for an empty tree) and a `value`, and returns the `tree` with the value added to it. For example, to start with an empty tree and add the values `"foo"` and `"bar"` to it:

```
struct tree *tree = NULL;
tree = tree_add(tree, "foo");
tree = tree_add(tree, "bar");
```

The `tree_dump` function prints the histogram table as shown above. Hint: Use in-order traversal of the tree in order to print out each value in sorted order. A good `printf` format string for the `count` and `value` on each line is `"%4u\t%s\n"`.

Because each node will be allocated with `malloc`, you will also write `tree_free`, which recursively calls `free` on every node in a tree.

Put each function in its own file; for example, implement `tree_add` in `tree_add.c`. In a file named `hist.c`, write a `main` function that adds each of its command-line arguments (not including `argv[0]`, the name of the program) to an initially empty tree, dumps it, and then frees it. If you name your files as indicated, the following `Makefile` will enable you to build your program by simply typing `make`.

Makefile

```
CFLAGS=-Wall -g

OBJECTS=\
  hist.o \
  tree_add.o \
  tree_dump.o \
  tree_free.o

hist: $(OBJECTS)
hist.o: hist.c tree.h
tree_add.o: tree_add.c tree.h
tree_dump.o: tree_dump.c tree.h
tree_free.o: tree_free.c tree.h

.PHONY: clean
clean:
        rm -f hist $(OBJECTS)
```

On CMS, submit your source files, `hist.c`, `tree_add.c`, `tree_dump.c`, and `tree_free.c`; when grading, I will provide the `tree.h` header file and the `Makefile` as they appear above.