Linked lists

Let us implement a linked-list data structure. A linked list is one of the two main ways of storing a list of items, the other being an array; linked lists have the advantage of permitting insertion anywhere, even the middle, whereas arrays have the advantage of speed of access and simplicity. A linked list is a collection of nodes, at disjoint locations in memory. One such node is the "head", which contains the first `value` in the list and a `next` pointer to the rest of the list, or "tail". That next node itself has the next value and its own `next` pointer, and so forth, with the final node containing the final value and a `NULL` `next` pointer.

A node will be represented as a `struct list`. Three functions will manipulate lists, as shown in the header:

`list.h`

```c
struct list {
        struct list *next;
        char *value;
};

struct list *list_add(struct list *list, char *value);
void list_dump(struct list *list);
void list_free(struct list *list);
```

The `list_add` function takes a list and adds a new `value` to it, maintaining the list in sorted order and ignoring duplicates, and returning the new head of the list. An empty list is represented by a `NULL` pointer.

`list_add.c`

```c
#include <stdlib.h>
#include <string.h>
#include "list.h"

struct list *list_add(struct list *list, char *value)
{
        struct list *tmp;

        if (!list || strcmp(list->value, value) > 0) {
                tmp = malloc(sizeof *tmp);
                tmp->next = list;
                tmp->value = value;
                return tmp;
        }

        if (strcmp(list->value, value) < 0)
                list->next = list_add(list->next, value);

        return list;
}
```

For every `malloc` there must be exactly one `free`; if you `malloc` some memory and do not `free` it, there is a leak and you can run out; if you `free` it twice, your program behavior is “undefined”. To deallocate an entire list at once, call `list_free`.

list_free.c

```c
#include <stdlib.h>
#include "list.h"

void list_free(struct list *list)
{
        if (list->next)
                list_free(list->next);
        free(list);
}
```

In order to provide a way of looking through a list without the debugger, let us also provide a `list_dump` function to print it out.

list_dump.c

```c
#include <stdio.h>
#include "list.h"

void list_dump(struct list *list)
{
        while (list) {
                printf("%s\n", list->value);
                list = list->next;
        }
}
```

The `main` function exercises the various list operations.

list.c

```c
#include <stdlib.h>
#include "list.h"

int main(int argc, char **argv)
{
        struct list *list = NULL;

        list = list_add(list, "hello");
        list = list_add(list, "hello");
        list = list_add(list, "world");
        list = list_add(list, "lion");

        list_dump(list);
        list_free(list);

        return 0;
}
```

Although "hello" is added twice, the duplicate shall be ignored, and although the strings are added out of order, the list will keep them in order.

```
$ ./list
hello
lion
world
```