

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 -----
1 struct list {
2     struct list *next;
3     char *value;
4 };
6 struct list *list_add(struct list *list, char *value);
7 void list_dump(struct list *list);
8 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 -----
1 #include <stdlib.h>
2 #include <string.h>
3 #include "list.h"
5 struct list *list_add(struct list *list, char *value)
6 {
7     struct list *tmp;
9     if (!list || strcmp(list->value, value) > 0) {
10         tmp = malloc(sizeof *tmp);
11         tmp->next = list;
12         tmp->value = value;
13         return tmp;
14     }
16     if (strcmp(list->value, value) < 0)
17         list->next = list_add(list->next, value);
19     return list;
20 }
-----
```

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 -----
1 #include <stdlib.h>
2 #include "list.h"
4 void list_free(struct list *list)
5 {
6     if (list->next)
7         list_free(list->next);
8     free(list);
9 }
```

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 -----
1 #include <stdio.h>
2 #include "list.h"
4 void list_dump(struct list *list)
5 {
6     while (list) {
7         printf("%s\n", list->value);
8         list = list->next;
9     }
10 }
```

The main function exercises the various list operations.

```
----- list.c -----
1 #include <stdlib.h>
2 #include "list.h"
4 int main(int argc, char **argv)
5 {
6     struct list *list = NULL;
8     list = list_add(list, "hello");
9     list = list_add(list, "hello");
10    list = list_add(list, "world");
11    list = list_add(list, "lion");
13    list_dump(list);
14    list_free(list);
16    return 0;
17 }
```

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
```
