`libll` Manual

Nebhrajani A.

2020

Contents

1 Introduction

`libll` is a linked list library (API) for C. It provides creation, node addition/deletion, printing, searching, sorting, appending to, and destruction of a linked list.

2 Installation

2.1 Prerequisites

To use `libll`, you should have the latest version of `gcc` or equivalent C compiler.

2.2 Installation

To install `libll` using `gcc`, the recommended way is to use a shared library. First `cd` into the directory where you want to install. Then:

```
$ gcc -c -fpic /src/libll.c
$ gcc -shared -o libll.so libll.o
```

You may need to add this path to the `$LD_LIBRARY_PATH` variable:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/your/path
```

Add this line to your `/.bashrc` if you want the variable definition to be permanent across sessions.

```
$ echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/your/path/' >> ~/.bashrc
```

> **Note:** Alternatively, you can use the `ldconfig` utility.

2.3 Runtime libraries

If you haven't used `ldconfig`, you'll need to tell the linker explicitly where to look for `libll` at runtime using the `-L` option, like so:

```
$ gcc -L /your/path -o program.c -lll
```

3 Functions

3.1 `lladd`

```
ll *lladd(ll *_list, void *_data)
```

Description Adds an element to a list. If list doesn't exist, creates anew.

Arguments
`ll *_list`: Linked list to add to or create.
`void *_data`: Pointer to your struct.

Returns A pointer to the new list or the pointer to the list added to.

How to use To use `lladd`, you need to do the `malloc` on your own, add data, and pass the pointer to `lladd`.

1. Create a `struct` for the data you want to store.

   > **Note:** You need to include a unique comparable data type in your struct (passed as `_key` to `libll`) to be able to search and delete. This data type acts as a **primary key**.

   Something like this will do:

```c
typedef struct user
{
  int key;
  int y;
  float z;
} usr;
```

Here, `key` is a unique integer.

2. Write your own add function. First `malloc` as much space as you need, and fill up your data. Then pass the pointer to `lladd`, like this:

```c
ll *my_add(ll *pX, int key, int y, float z)
{
  usr *mydata = NULL;

  if ((mydata = (usr *)malloc(sizeof(usr))) == 0) {
    fprintf(stderr, "Unable to allocate memory. Fatal\n");
    exit(1);
  }

  mydata->key = key;
  mydata->y = y;
  mydata->z = z;

  pX = lladd(pX, mydata);

  return pX;
}
```

3. Add elements to your new linked list in `main()`.

```c
  pX = my_add(pX,  1, 123, 123.14);
  pX = my_add(pX,  2, 456,  24.15);
  pX = my_add(pX,  3, 510, 115.43);
  pX = my_add(pX,  4, 910,  15.43);
  pX = my_add(pX,  5, 130, 215.43);
  pX = my_add(pX,  6, 150, 125.43);
  pX = my_add(pX,  7, 105, 165.43);
  pX = my_add(pX,  8, 210, 154.73);
  pX = my_add(pX,  9, 101, 155.43);
  pX = my_add(pX, 10, 410, 145.43);
  pX = my_add(pX, 11, 180, 185.43);
  pX = my_add(pX, 12, 109, 158.43);
```

3.2 `llprint`

`void llprint(ll *_list, char *(* usr_print)(void *_data))`

Description Prints list.

Arguments
`ll *_list`: List to be printed.
`char *(* usr_print)(void *_data)`: Pointer to your print function.

Returns Void.

How to use To print the list, you need to tell `llprint` *how* you want the list printed. Create and pass to `llprint` a pointer to your print function. Your print function needs to accept a pointer to your struct as an argument and return a formatted string. `llprint` will call this function iteratively and print the list.

1. Write your print function, something like the following:

```
char *my_print(usr *this)
{
  static char s[80];
  sprintf(s, "key = %d, y = %d, z = %0.2f\n", this->key, this->y, this->z);
  return s;
}
```

2. In `main()` create a function pointer to your print function and pass it to `llprint`, like this:

```
  char *(*print)(usr*) = my_print;
  llprint(pX, (char *(*)(void *))print);
```

3.3 `llsort`

`ll *llsort(ll *_list, int (* usr_compare)(void *_data1, void *_data2))`

Description Sorts list based on input compare function.

Arguments
`ll *_list`: List to be sorted.
`int (* usr_compare)(void *_data1, void *_data2)`: Pointer to your compare function.

Returns Pointer to sorted list.

How to use Pass `llsort` a pointer to your compare function (see below). It will return a pointer to the sorted list.

1. Write a comparison function. It should take two pointers to two of your structs, compare whichever data you want to, and return values similar to the standard function `strcmp`. One way of doing this is:

```
int my_cmp(usr *mydata1, usr *mydata2)
{
  int a = 0;
  int b = 0;

  a = mydata1->y;
  b = mydata2->y;

  if (a > b)
    return 1;
  else if (a < b)
    return -1;
  else
    return 0;
}
```

> **Note:** To switch between ascending and descending sorts, switch the returns `1` and `-1` in your compare function.

2. In `main()`, create a function pointer to your compare function and pass it to `llsort`, like so:

```
  int (*cmp)(usr *, usr *) = my_cmp;
  pX = llsort(pX, (int (*)(void *, void *))cmp);
```

> **Note:** `llsort` uses an internal function `_lldelete` to only delete and replace the actual nodes in the linked list and not the structs their `void *_data`'s point to: it instead juggles the pointers without ever touching your structures. `llsort` doesn't use `lldelete`, which is meant to delete both the node and the data structure it points to.

3.4 llprint_single

```
void llprint_single(ll *_this, char *(* usr_print)(void *_data))
```

This function behaves the same as `llprint`, but only prints one element: it doesn't loop through the entire linked list. This is meant to be used with `llsearch`. so that only the searched element is printed.

3.5 lldelete

```
ll *lldelete(ll *_list, int _key, int (* usr_compare_key)(void *_data, int key),
             void (* usr_free)(void *_data))
```

Description Deletes element from list.

Arguments
`ll *_list`: Which list to delete from.
`int _key`: Unique identifier for which element to delete.
`int (* usr_compare_key)(void *_data, int key)`: Pointer to your key matching function.
`void (* usr_free)(void *_data)`: Pointer to your free function.

Returns Pointer to list with deleted element.

How to use `lldelete` needs to know which element you want to delete and from which list. It does this using the `_key`. It also needs a way to check if the key has matched: necessitating you to write a key comparison function. Finally, only deleting the node is not enough, the data associated with it must also be freed. To do this, you'll have to write a free function. Each of these must be passed as an argument to `lldelete`.

1. Write a key comparison function. It should accept two pointers to your structures and return `0` if the keys match. For example,

```c
int my_key_cmp(usr *mydata, int key)
{
  int a = 0;
  int b = 0;

  a = mydata->key;
  b = key;

  if (a == b)
    return 0;
  else
    return 1;
}
```

> **Note:** If you're always sorting using the key, you can reuse the compare function you wrote for sort. (`my_cmp`)

2. Write a free function. It accepts a pointer to your struct and frees it, like so:

```c
void my_free(usr *mydata)
{
  if (mydata) {
    free(mydata);
  }
}
```

3. Call `lldelete` in `main()` with the required arguments and by declaring the pointers to the functions you created earlier.

```
int (*key_cmp)(usr *, int) = my_key_cmp;
void (*free)(usr *mydata) = my_free;
pX = lldelete(pX, 3, (int (*)(void *, int))key_cmp, (void (*)(void *))free);
```

3.6 lldestroy

`void lldestroy(ll *_list, void (* usr_free)(void *_data))`

Description Destroys entire list.

Arguments
`ll *_list`: List to be destroyed.
`void (* usr_free)(void *_data)`: Pointer to free function.

Returns Void.

How to use lldestroy is in essence an lldelete which iterates through the entire list. To use it:

```
lldestroy(pX, (void (*)(void *))free);
```

3.7 llsearch

`ll *llsearch(ll *_list, int _key, int (* usr_compare_key)(void *_data, int key))`

Description Searches element in list.

Arguments
`ll *_list`: Which list to search in.
`int _key`: Unique identifier for which element to search for.
`int (* usr_compare_key)(void *_data, int key)`: Pointer to key matching function.

Returns Pointer to the element if found. Otherwise, returns a `NULL` pointer.

How to use The usage is similar to `lldelete`. However, **always store the result of the search in a new variable** of type `ll *`. If you don't, you'll overwrite the pointer to the original linked list. Print this new variable using `llprint_single`, like so:

```
sX = llsearch(pX, 12, (int (*)(void *, int))key_cmp);
llprint_single(sX, (char *(*)(void *))print);
```

3.8 llwhich

`int llwhich(ll *_list, int _key, int (* usr_compare_key)(void *_data, int key))`

Description Whichth node is element at?

Arguments
`ll *_list`: Which list to count in.
`int _key`: Unique identifier for which element to count till.
`int (* usr_compare_key)(void *_data, int key)`: Pointer to key matching function.

Returns Integer.

How to use The use is the same as `llsearch`:

```
i = llwhich(pX, 12, (int (*)(void *, int))key_cmp);
```

3.9 `llappend`

```
ll *llappend(ll *_a, ll *_b)
```

Description Appends to non-null list.

Arguments
`ll *a`: List to append to.
`ll *b`: List to be appended.

Returns Pointer to appended list.

How to use Call `llappend` with the two lists you want to append. Note that you can't append to a `NULL` list, in which case `llappend` will return a `NULL` pointer.

```
1 pX = llappend(pX, qX);
```

> **Note:** To append more than two lists, you can call `llappend` recursively. In fact, this is exactly how `llsort` appends lists.

4 License (MIT)

5 Contact

Feel free to submit any ideas, questions, or problems by reporting an issue. Alternatively, you can reach out to me at <`aditya.v.nebhrajani@gmail.com`>.