HW 6: Linked Lists!

Due: Friday, Dec 5, by 2:59PM

Part I: Examine the files `list.h` and `llist.c` in the `src` directory where you found this handout.

You will discover that it is a partially implemented linked list "module".

The lists store numeric values (the type of which can be changed by altering the `typedef` for `ElemType` in `list.h`).

As in previous projects, the `.h` file gives the interface for an ADT while the actual implementation is given in the `.c` file. The members of `list_struct` are also "hidden" in the `.c` file. The ADT defines many natural operations on lists -- some of these have already been implemented and will be used as motivating examples during lecture; others have not been implemented: It is your job to do the implementation! Look for `TODO` labels throughout the files.

A subtle detail: why did I decide to name the header file `list.h` (one `'l'`), but the implementation file `llist.c` (two `'l'`s)???

So… part I is completion of all of the `TODO` items specified.

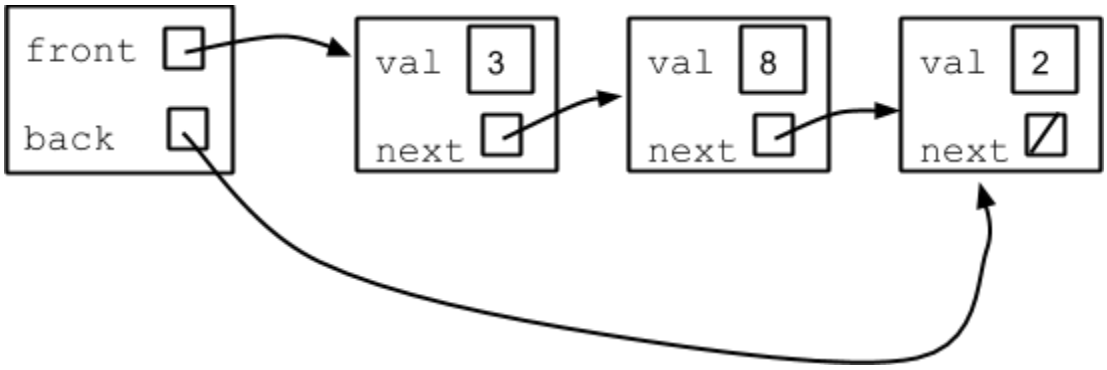
Rules: you cannot change `list.h` (except maybe to experiment with different `ElemType`s). All of your work is in `llist.c` (except testing code).

Discussion: The given linked list structure has two "levels":

> At the "lowest" level are the linked-list nodes themselves specified as:
>
> ```
> typedef struct node {
>     ElemType val;
>     struct node *next;
> } NODE;
> ```
>
> However, the type `NODE` isn't even visible to a client program. Only the type `LIST` is visible to a client (just the type -- not the struct members). Through the header file, `LIST` is equivalent to a `struct list_struct` which is specified as follows:

```
struct list_struct {
    NODE *front;
    NODE *back;
};
```

Here is a diagram of a list with three entries: `<3, 8, 2>`. The `struct` at the left (a `LIST`) gives access to the actual nodes.



List of `TODO` functions:

```
lst_count, lst_pop_back, lst_print_rev, lst_reverse,
lst_sra_bad_case, lst_remove_all_fast, lst_is_sorted,
lst_insert_sorted, lst_merge_sorted
```

Part II: You will write a client program of the list module which will perform a sorting operation. We will call it **Iterative Merge Sort.**

Take a look at the function `lst_merge_sorted`. It takes two lists `a` and `b` which are assumed to be sorted and merges them into a single sorted list stored in `a` (while `b` becomes empty).

Well, if I have a collection of n values that I want to sort, suppose I did this:

> For each of the `n` elements, create a list containing just that element.
>
> Now we have `n` lists of length 1.
>
> Also, they are sorted -- there is only one way to order one element!

Now suppose I call `lst_merge_sorted` on pairs of lists created above. For example, suppose `n=32`; after one "pass" I will have `16` sorted lists, each of length `2`.

If we repeat, we will call `lst_merge_sorted` on `8` pairs of lists producing `8` sorted lists of length `4`.

After the next pass, we will have 4 sorted lists of length 8; then 2 sorted lists of length 16; and after one final pass, a single sorted list of all 32 elements.

That sounds like an algorithm!

Your Program:

You will write a program called `imsort` (source file `imsort.c`) which reads a sequence of integers from `stdin;` sorts those integers by the procedure above and prints the sorted sequence to `stdout`.

Note that you do not know ahead of time how many integers there will be -- sounds like a good application of a dynamic data structure like a linked list!

You should use `scanf` for this program instead of `get_int`. Why? Because `get_int` assumes there is a real person giving input, but you will want to just redirect input from files like this:

```
$ ./imsort < input_file
```

Detailed behavior: the program just reads integers until `scanf` fails. It then sorts and prints out the sorted sequence, one value per line.

That's it!

Submission Details: Stay tuned, but nothing too surprising.