The Linked List in Lab 3 SP23

Contents

Basics

The balls will be stored on either of two linked lists. Your sim structure keeps the head pointers. The sim structure uses void pointers for this because the lab 3 code does not know the internal data type used by the list. *Even though node.h is in the zip, you may not use it in lab 3!* The supplied library implements the list operations for you in lab 3.

The list provides 4 operations we use

- insert (to add a ball to a list)
- deleteSome (to delete zero or more balls from a list)
- iterate (to do something with every ball, one at a time)
- sort

Details on these are in their own sections.

Head Pointer Operations

Your code may do the following to the head pointer that the sim structure holds:

- Set it to NULL (do this once when you initialize the sim structure)
- Compare it to NULL (a NULL value indicates an empty list)
- Pass it to a list function such as iterate that doesn't need to change them (iterate and sort)
- Pass its address to list functions that do need to change them such as insert and deleteSome

Your lab 3 code is **not** allowed to manipulate the head pointer in any other way.

Data on the list

The list will hold any kind of data that you can get the address of. Your code hands a pointer to your data to the list. The list stores that pointer in a void pointer, because the list doesn't know what kind of data you want to keep on the list. You can put strings on the list. You can put balls on the list. You should never put different kinds of data on one list at the same time. Your code should always know what data type is on the list. In lab 3, that will be pointer to ball structures. In your prototypes, I suggest using strings; they are natively pointer based and they are easy to print.

Callback functions

Your code will give the list callback functions. Those callback functions will receive one or more void pointers. Your code **cannot** cast the void pointers, it must assign them to a pointer of the correct type. An example of this is given in the "Data" document section *Interactions Between Structs and the Lists.*

The callbacks below are described in terms of balls. For prototypes, you can have a different set of callbacks that work with strings. Note that you will *never* need to declare any variable to be of type ActionFunction even though you could. You can pass the name of any function fitting that signature to the list as your action function.

Do **not** declare any variable of the typedef'd types:

- **`ActionFunction`**
- **`ComparisonFunction`**
- **`CriteriaFunction`**

The list is NOT re-entrant safe! For example, your code calls iterate. The iterate function calls you action function "foo." The foo function must never call any list function on the *same* list. (In lab 3 balls we delete from the in-play list get inserted to the off-table list and we are OK since those are *different* lists.)

Action Functions

```
typedef void (* ActionFunction)( void *data);
```

Action functions are used to do things to the data you stored on the list. The action function is given a pointer to a ball structure. An example action function would be one that draws a ball. Another use of an action function is to deal with the data held by a node that is about to be deleted. In that case, it should be freed or inserted into a different list.

Comparison Function

```
typedef bool (* ComparisonFunction)(void *data1, void *data2);
```

A common comparison function takes two pointer to ball and returns a Boolean indicating true when the first ball belongs earlier in the list than the second ball. You will write comparison functions and pass their address to parameters of this type. Insert needs one of these to put new balls into the list in the proper order. Sort needs them as well.

Criteria Functions

```
typedef bool (* CriteriaFunction)(void *data);
```

A criteria function is used to give back a Boolean value. The data value will be a pointer to a ball structure. The "always true" criteria function will ignore its parameters. Use "always true" when clearing a list at the end. Other criteria functions will do things like decide if a ball is off the table.

Library functions

The following functions are provided in the linked list library. See the zip fine on piazza. The header file is in the zip as well. Do not implement these in lab 3.

Insert

```
/* int returns FALSE when it fails to do the insert */
bool insert(void *p2head, void *data, ComparisonFunction goesInFrontOf, int text);
```

Because of call-by-value, the first parameter is the address of the head pointer of the list you want to insert the alien on. The data parameter will be a pointer to a ball. When inserting into the in-play list, use a comparison function that uses Y values. The text parameter will always be passed TEXT.

deleteSome

```
int deleteSome(void *p2head, CriteriaFunction mustGo,
                  ActionFunction disposal, int text);
```

This function is used to remove nodes from a linked list. It gets passed the address of a head pointer since it may need to alter that pointer. The criteria function passed yields true if the alien needs to be removed. The text parameter will always be passed TEXT.

One use of deleteSome is to remove a ball that hit the ground. The criteria function decides if the alien needs to leave the list. If so, the action function will be called. Other uses of deleteSome are to clear a list at the end.

deleteSome returns the total number of nodes deleted.

Iterate

```
void iterate(void *head, ActionFunction doThis);
```

Iterate walks through the list, passing each data item on the list in turn to the action function. In lab 3 code, the action function does whatever it does to the ball structure (we only have ball structures on our lists) and then goes on to the next node in the list. (In prototypes, the action function does whatever it does to the data type on the list, typically strings.) Since iterate does not cause any nodes to enter or leave the list, the first parameter is a head pointer *not the address of the head pointer like insert and deleteSome need.*

sort

```
void sort(void *hptr, ComparisonFunction cf);
```

Sort does what you expect using the comparison function that you give. It bubble sorts the list comparing adjacent items using the comparison function. Sort moves data around, but it leaves the links alone. Since sort does not cause any nodes to move, enter, or leave the list, the first parameter is a head pointer *not the address of the head pointer like insert and deleteSome need.*

Other Functions

There are other functions in the linked list library, but we won't use them.

Building with the list library

Your makefile might need something like this:

```
# all of my dependencies must be dot o files
lab3: <a list of all the .o files goes here>
        gcc -g -o $@ $^ -L. -lpb -llinkedlist -lncurses
```

Prototypes and the list

The list really doesn't care what type it holds, as long as your code coughs up a pointer to that data. The easiest way to do prototypes with the list – strongly suggested – is to use strings. They are pointer friendly and print easily and recognizably. Consider this code:

```
void *list = NULL;
rval = insert ( &list, "Go Bucks!", first_letter, 1);
```

Here list is a pointer to void, first_letter is a comparison function that compares the first character of two strings and returns the result of numerically comparing them. Then call

```
iterate(list, print_string);
```

Here print_string is an action function that prints a string (pointer to char) that came in as a pointer to void. It is all of two lines long; a definition of a char pointer and a printf statement.

Don’t bother with ball structures when prototyping the list. Make sure that you can use all four of the list functions. Start with insert and use iterate to test it. Then do a few insertions and delete one thing from the list and use iterate to print what’s left. You might insert stirrings that start with both upper and lower case characters and you delete strings that start with a lower case letter. You could test sort by using a comparison function that compares the strlen of the two strings and then using iterate to print the newly sorted list.