# Lab 11 - More Linked List Exercises

**Objective**

To gain additional experience designing and implementing functions that operate on singly-linked lists.

**General Requirements**

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your functions should produce console output; i.e., contain `printf` statements.

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style).

Finish each exercise (i.e., write the function and verify that it passes all of its tests) before you move on to the next one. Don't leave testing until after you've written all your functions.

## Getting Started

**Step 1:** Depending on your operating system, download (**lab11_win.zip** or **lab11_mac.zip**) the starting code from Brightspace and extract the zipped file.

**Step 2:** Double click on lab11.code-workspace

**Step 3:** lab11_xxx folder contains the following files:

- sl_list.c contains four fully-implemented functions: `intnode_construct`, `push`, `length` and `print_list`. This file also contains incomplete definitions of two functions you have to design and implement.

  **You don't need to copy/paste your solutions to the previous lab's linked-list exercises into sl_list.c.; however, you may be able to reuse code from some of those functions when designing the solutions to this lab's exercises.**

- sl_list.h contains the declaration for the nodes in a singly-linked list (see the `typedef` for `intnode_t`) and prototypes for functions that operate on this linked list. **Do not modify sl_list.h.**

- main.c contains a simple *test harness* that exercises the functions in sl_list.c. Unlike the test harnesses provided in previous labs, this one does not use the sput framework. The harness doesn't compare the actual and expected results of each test and keep track of the number of tests that pass and fail. Instead, the expected and actual results are displayed on the console, and you have to review this output to determine if the functions are correct. **Do not modify `main()` or any of the test functions.**

**Step 4:** Run the project. It should build without any compilation or linking errors.

**Step 5:** Execute the project. For the functions where the test harness is provided (the functions in main.c) will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions.

**Step 6:** Open sl_list.c and do Exercises 1 to 4.

**Exercise 1**

File sl_list.c contains an incomplete definition of a function named `add`. The function prototype is:

```
intnode_t *add(intnode_t *head, int elem, int index);
```

Parameter `head` points to the first node in a linked list, or is `NULL` if the linked list is empty.

This function will insert a new node containing integer `elem` at the specified position (`index`) within the list. The function uses the numbering convention that the first node is at index 0, the second node is at index 1, and so on. Parameter `index` must be in the range 0..*length* (where *length* is the number of nodes in the linked list).

Note that this function does not replace the contents of the node (if any) that is currently at the position specified by `index`. Instead, the new node is inserted so that it has the given index.

The function returns a pointer to the first node in the modified linked list.

There are several cases you need to consider when designing this function:

- The linked list is empty. There are two subcases:
    - `index` is 0. The function will return a pointer to a list containing one node.
    - `index` is invalid, so the function should terminate via `assert`.
- The linked list has one or more nodes. There are four subcases:
    - `index` is 0. The function will insert the node at the front of the linked list.
    - `index` is greater than 0 and less than the number of nodes in the linked list. The function will insert a new node at the specified position.
    - `index` equals the number of nodes in the linked list. The function will append a new node after the last node.
    - `index` is invalid, so the function should terminate via `assert`.

Design and implement `add` (but read the following paragraphs before you do this). You should be able to reuse parts of the algorithms from the `pop` function you wrote for Lab 8 and the `push` and `append` functions that were presented in lectures. (Links to C Tutor implementations of these functions are on cuLearn).

We recommend that you sketch some "before and after" diagrams of the linked list for each of the cases before you write any code. (One diagram will show the linked list before the function is called, the other diagram will show the linked list after the function returns.) Use these diagrams as a guide while you code the function.

We also recommend that you use an iterative, incremental approach, instead of writing the entire function before you start testing. For example, during the first iteration, write just enough code to handle the "linked list is empty, index is 0" case. Run the test harness and fix any flaws. When your function passes the tests for this case, pick another case, for example, "linked list has one or more nodes, index is 0". Write the code for this case and retest your function. Verify it passes all the tests for both cases. Repeat this process until your function handles all the cases except those that use `assert`. You can then add the `assert` statements.

Finally, if you become "stuck" while working on the exercises, consider using C Tutor to help you discover the problems in your solution.

<u>Verify that your `add` function passes all the tests before you start Exercise 2.</u>

**Exercise 2**

File sl_list.c contains an incomplete definition of a function named `every_other`. The function prototype is:

```
void every_other(intnode_t *head);
```

Parameter `head` points to the first node in a linked list, or is `NULL` if the linked list is empty.

The function deletes every other node from the linked list, starting with the second node. In other words, the modified list will contain the first, third, fifth, etc. nodes from the original list. For example, suppose variable `my_list` (of type `intnode_t *`) points to this linked list:

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9
```

and `every_other` is called this way:

```
every_other(my_list);
```

When the function returns, `my_list` points to this linked list: `1 -> 3 -> 5 -> 7 -> 9`

`every_other` should **not** terminate via `assert` or modify the list if it is passed an empty linked list or a list that contains only one node. The function must correctly process linked lists that have an even number of nodes and lists that have an odd number of nodes. The nodes that are removed from the list must be deallocated properly; in other words, your function must not cause memory leaks.

**Step 1:** Reread the specification, and write a list of all the cases that this function must handle.

**Step 2:** Design and implement `every_other`.

Hint: A complete solution requires fewer than 15-20 lines of code. A solution is that is longer than this is likely more complex than it needs to be. This function only needs to make one traversal of the linked list. If your function makes more than one than one traversal, it is more complicated than it needs to be. Also, there is no need to copy values from one node to another.

We recommend that you follow the same approach that suggested for Exercise 1; that is, sketch some "before and after" diagrams of the linked list for each of the cases before you write any code, then use the incremental, iterative technique to code and test the function.

**Exercise 3**

File sl_list.c contains an incomplete definition of a function named `count_in_sll`. The function prototype is:

```
int count_in_sll(node_t *head, int target);
```

This function counts the number of integers in the singly-linked list pointed to by `head` that are equal to `target`, and returns that count. For example, if the linked list pointed to by `head` contains the 11 integers 1, 2, 4, 4, 5, 6, 4, 7, 8, 9 and 12, then `count_in_sll(list, 4)` returns 3 because 4 occurs three times in the list.

Implement `count_in_sll` as a recursive function. Your `count_in_sll` function <u>cannot</u> have any loops. Hint: review the recursive `length` function that was presented in lectures (the lecture slides and the C Tutor links are posted on Brightspace.) We recommend that, before writing any code. you formulate a recursive definition of the solution to the problem, "What is the number of integers in the singly-linked list pointed to by `head` that are equal to `target`?" Then, convert that definition to C code. We also recommend that you use an iterative, incremental approach when implementing the function. For example, during the first iteration, write just enough code to handle the base case(s). Use the console output to help you identify and correct any flaws. When your function passes the tests for this case, write the code recursive case(s).

Function `test_exercise_3` has six test cases for the `count_in_sll` function. It calls `test_count_in_sll` six times, once for each test case. Notice that `test_count_in_sll` has three arguments: the two arguments that will be passed to `count_in_sll`, and the value that a correct implementation of `count_in_sll` will return.

Build and execute the project. Use the console output to help you identify and correct any flaws. Verify that `count_in_sll` passes all of its tests before you start Exercise 4.

**Exercise 4**

File sl_list.c contains an incomplete definition of a function named `last_in_sll`. The function prototype is:

```
int last_in_sll(node_t *head);
```

This function returns the last element in the singly-linked list of integers pointed to by `head`. For example, if the linked list pointed to by `head` contains the 5 integers 1, 2, 4, 4, 6, 5, then `last_in_sll(list)` returns 5 because 5 is the last element in the list.

The function must terminate (via `assert`) if it is passed an empty list.

Implement `last_in_sll` as a recursive function. Your `last_in_sll` function <u>cannot</u> have any loops. We recommend that, before writing any code. you formulate a recursive definition of the solution to the problem, "What is the last element in the singly-linked list pointed to by `head`?" Then, convert that definition to C code. We also recommend that you use an iterative, incremental approach when implementing the function. For example, during the first iteration, write just enough code to handle the base case(s). Use the console output to help you identify and correct any flaws. When your function passes the tests for this case, write the code recursive case(s).

Function `test_exercise_4` has four test cases for the `last_in_sll` function. It calls `test_last_in_sll` four times, once for each test case. Notice that `test_last_in_sll` has two arguments: the argument that will be passed to `last_in_sll`, and the value that a correct implementation of `last_in_sll` will return.

Build and execute the project. Use the console output to help you identify and correct any flaws. Verify that `last_in_sll` passes all of its tests.

## Wrap-up

Submit sl_list.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace* (lab3 file). Remember: submit early, submit often.
- Make sure that sl_list.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you are submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

## Homework Exercise - Visualizing Program Execution

In the final exam, you will be expected to be able to draw diagrams that depict the execution of short C functions that manipulate linked lists, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills when working with linked lists.

1. Click on this link to open C Tutor.
2. Copy the `intnode_t` declaration from `sl_list.h`, `intnode_construct` and your solutions to Exercises 1 and 2, into C Tutor.
3. Write a short `main` function that exercises your list functions. Feel free to borrow code

from this lab's test harness.

4. *Without using C Tutor,* trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before the `return` statements in each of your list functions are executed. Use the same notation as C Tutor.

5. Use C Tutor to trace your program one statement at a time, stopping just before each `return` statement is executed. Compare your diagrams to the visualization displayed by C Tutor.
6. Do not submit your solutions for this exercise.