Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming

Lab 11 - More Linked List Exercises

Objective

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

Online Submission

Submit file sl_list.c through cuLearn before the deadline.

Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.

General Requirements

You have been provided with three 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 several 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.**

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). Pelles C makes it easy to do this - instructions were provided in Labs 1 and 2.

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.

Instructions

Step 1: Launch Pelles C and create a new Pelles C project named linked_list_lab_11.

- If you're using the 64-bit edition of Pelles C, the project type should be Win 64 Console program (EXE). (Although the 64-bit edition of Pelles C can build 32-bit programs, you may run into difficulties if you attempt to use the debugger to debug 32-bit programs.)
- If you're using the 32-bit edition of Pelles C, the project type should be Win32 Console program (EXE).

When you finish this step, Pelles C will create a folder named linked_list_lab_11.

Step 2: Download file main.c, sl_list.c and sl_list.h from cuLearn. Move these files into your linked_list_lab_11 folder.

Step 3: You must add main.c and sl_list.c to your project. To do this:

- select Project > Add files to project... from the menu bar.
- in the dialogue box, select main.c, then click Open. An icon labelled main.c will appear in the Pelles C project window.
- repeat this for sl_list.c.

You don't need to add sl_list.h to the project. Pelles C will do this after you've added main.c.

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

Step 5: Execute the project. The test harness will show that functions `add` and `every_other` do not produce correct results (look at the output printed in the console window and, for each test case, compare the expected and actual results). This is what we'd expect, because you haven't started working on the functions that the test harness tests.

Step 6: Open sl_list.c and do Exercises 1 and 2.

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 `fetch` function you wrote for Lab 9 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.

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

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.

Wrap-up

1. ~~Remember to have a TA review and grade your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the grading/sign-out sheet.~~
2. ~~Remember to back up your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service. All files you've created on the hard disk will be deleted when you log out.~~

1. Remember to submit your file before the deadline.
2. Remember to save your work on your computer.

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. Launch C Tutor (the *Labs* section on cuLearn has a link to the website).
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.