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

Lab 8 - Linked Lists

Objective

This is the first in a series of labs that focus on learning how to design and implement functions that operate on singly-linked lists.

Online Submission

Submit file singly_linked_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:

- singly_linked_list.c contains four fully-implemented functions: `intnode_construct`, `push`, `length` and `print_list`. This file also contains incomplete definitions of five functions you have to design and implement.
- singly_linked_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 singly_linked_list.h.**
- main.c contains a simple *test harness* that exercises the functions in singly_linked_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.**

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.

- 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.

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

Step 3: You must add main.c and singly_linked_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 singly_linked_list.c.

You don't need to add singly_linked_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 `count`, `max`, `index`, `extend` and `pop` 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 singly_linked_list.c and do Exercises 0 through 5. If you become "stuck" while working on the exercises, consider using C Tutor to help you discover the problems in your solution. Links to C Tutor "templates" for the exercises are posted on cuLearn.

Exercise 0

The cuLearn course page has links to C Tutor examples that will help you visualize and understand the `length` and `print_list` functions (the links are in the *Lecture Materials* section). Execute these examples, step-by-step. Make sure you understand the algorithm for traversing a linked-list's nodes. Make sure you understand why the loop condition in `length` is slightly different from the one in `print_list`.

Exercise 1

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

```
int count(intnode_t *head, int target);
```

Parameter `head` points to the first node in a linked list.

This function counts the number of nodes that contain an integer equal to `target` and returns that number.

This function should return 0 if the list is empty (parameter `head` is `NULL`).

Finish the implementation of `count`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. <u>Review the console output and verify that your `count` function passes all the tests before you start Exercise 2.</u>

Exercise 2

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

```
int max(intnode_t *head);
```

Parameter `head` points to the first node in a linked list.

This function returns the largest number stored in the linked list.

This function should terminate (via `assert`) if the list is empty (parameter `head` is `NULL`).

Finish the implementation of `max`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. <u>Review the console output and verify that your `max` function passes all the tests before you start Exercise 3.</u>

Exercise 3

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

```
int index(intnode_t *head, int target);
```

Parameter `head` points to the first node in a linked list.

This function that returns the index (position) of the first node in the list that contains an integer equal to `target`. The function uses the numbering convention that the first node is at index 0, the second node is at index 1, and so on.

The function should return -1 if the list is empty (parameter `head` is `NULL`) or if `target` is not in the list.

Finish the implementation of `index`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output and verify that your `index` function passes all the tests before you start Exercise 4.

Exercise 4

On the cuLearn course page, find the links to the C Tutor examples that demonstrate the `intnode_construct` and `push` functions. Execute these examples, step-by-step.

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

```
void extend(intnode_t *head, int *other);
```

Parameters `head` and `other` point to the first nodes in two distinct linked lists. (In other words, `head` and `other` don't point to the same linked list.)

The function extends the linked list pointed to by `head` so that it contains *copies* of the values stored in the linked list pointed to by `other`.

The function terminates (via `assert`) if the linked list pointed to by `head` is empty.

Finish the implementation of `extend`.

Note 1: Something along the lines of:

```
last_node->next = other;
```

where `last_node` points to the last node in the list pointed to by `head`, is **not** correct. This simply "glues" the last node of one list to the first node of the other.

Note 2: Your `extend` function may not call the `append` function that was presented in lectures. This would be inefficient, because the list pointed to by `head` would be traversed every time `append` is called. Hint: an efficient solution requires exactly one traversal of each of the two lists.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output and verify that your `extend` function passes all the tests before you start Exercise 5.

Exercise 5

For this exercise, you're going to implement a function that is the inverse of `push`: it retrieves the value stored in a linked-list's head node and removes that node from the linked list.

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

```
intnode_t *pop(intnode_t *head, int *popped_value);
```

Parameter `head` points to the first node in a linked list.

This function copies the value in the list's head node to the location pointed to by parameter `popped_value` and returns a pointer to the first node in the modified list. (Hint: remember that `pop` must deallocate the head node correctly, because all nodes were allocated from the heap.)

The function should terminate via `assert` if the linked list is empty.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. <u>Review the console output, and verify that your `pop` function passes all the tests.</u>

Wrap-up

1. Remember to have a TA review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the attendance/grading 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.

3. **You'll need your linked_list module for Lab 9. That lab assumes your module passes all the tests in the Lab 8 test harness. Remember to complete any unfinished exercises before your next lab period.**

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.

If you didn't use C Tutor to help you implement the solutions to the exercises, use the tool to visualize the execution of your `count`, `max`, `index`, `extend` and `pop` functions. For each function:

1. Click on the link to the corresponding C Tutor template and copy your function definition into the template.

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

3. Use C Tutor to trace the program one statement at a time, stopping just before each `return` statement is executed. Compare your diagrams to the visualization displayed by C Tutor.