Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Summer 2019

Lab 9 - Linked Lists

Attendance/Demo

After you finish the exercises, a TA will review your solutions, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will grade the work you've completed, starting about 30 minutes before the end of the lab period. **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:

- linked_list.c contains three fully-implemented functions: `push`, `length` and `print_list`. This file also contains incomplete definitions of five functions you have to design and implement.
- linked_list.h contains the declaration for the nodes in a singly-linked list (see the `typedef` for `node_t`) and prototypes for functions that operate on this linked list. **Do not modify linked_list.h.**
- main.c contains a simple *test harness* that exercises the functions in 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). Instructions for selecting the formatting style and formatting blocks of code are in the Lab 1 handout.

Finish each exercise (i.e., write the function and verify that it passes all 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. (Instructions for creating projects are in the handout for Lab 1.) If you're using the 64-bit edition of Pelles C, select Win 64 Console program (EXE) as the project type. If you're using the 32-bit edition of Pelles C, select Win32 Console program (EXE). **Don't click the icons for Console**

application wizard, Win32 Program (EXE) or Win64 Program (EXE) - these are not correct types for this project.

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

Step 3: Add main.c and linked_list.c to your project. (Instructions for doing this are in the handout for Lab 1.)

You don't need to add 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`, `fetch`, `index` and `extend` 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 linked_list.c and do Exercises 1 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 1

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

```
int count(node_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. Use the console output to help you identify and correct any flaws. Verify that `count` passes all the tests before you start Exercise 2.

Exercise 2

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

```
int max(node_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. Use the console output to help you identify and correct any flaws. Verify that `max` passes all the tests before you start Exercise 3.

Exercise 3

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

```
int fetch(node_t *head, int index);
```

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

This function will return the integer stored in the node at the specified index (position). 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 terminate via `assert`:

- if the list is empty;
- if parameter `index` is invalid. Hint: consider a linked list that has *n* nodes. What is the index of the first node? What is the index of the last node? What is the range of valid index values?

Finish the implementation of `fetch`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `fetch` passes all the tests before you start Exercise 4.

Exercise 4

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

```
int index(node_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. Use the console output to help you identify and correct any flaws. Verify that `index` passes all the tests before you start Exercise 5.

Exercise 5

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

```
void extend(node_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: A solution that looks something like:

```
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 must call the `push` function (which was presented in lectures) to allocate and initialize new nodes.

Note 3: 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. Use the console output to help you identify and correct any flaws. Verify that `extend` passes all the tests.

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 grading/signout sheet.

2. Remember to backup 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.

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`, `fetch`, `index` and `extend` 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 and the heap just before the `return` statements in the function is 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.