Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2015

Lab 9 - Linked Lists

Objective

To review the linked list code that was presented in recent lectures, and to develop some functions that provide additional operations on linked lists.

Attendance/Demo

To receive credit for this lab, you must demonstrate your work. **Also, you must submit your lab work to cuLearn by the end of the lab period**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code 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 Lab 10.**

General Requirements

You have been provided with three files:

- singly_linked_list.c contains several functions that operate on singly-linked list. The `print_linked_list` function prints a linked list of integers using the format:

  ```
  value1 -> value2 -> value3 -> ...
  ```

  In addition, this file has functions that:

  - search a linked list to determine if it contains a specified value;
  - insert an integer at the front of a linked list;
  - append an integer to the rear of a linked list;
  - remove the first node in a linked list;
  - remove the last node in a linked list.

  The code in these functions was presented in recent lectures.

- singly_linked_list.h contains declarations for a singly-linked list data structure and prototypes for functions that operate on this linked list;

- 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 test code 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.

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.

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.

Instructions

1. Create a new folder named Lab 9.

2. Launch Pelles C and create a new Pelles C project named linked_list inside your Lab 9 folder. The project type must be Win32 Console program (EXE). You should now have a folder named linked_list inside your Lab 9 folder. Check this. If you do not have a project folder named linked_list, close this project and repeat Step 2.

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

4. You must add main.c and singly_linked_list.c to your project. From the menu bar, select Project > Add files to project... 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.

5. Build the project. It should build without any compilation or linking errors.

6. Execute the project. The test harness reveals that several functions 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 several of the functions in singly_linked_list.c are incomplete.

7. Open singly_linked_list.c and review the functions that contain the algorithms that were designed and implemented in recent lectures. Make sure you understand these functions before you start Exercise 1.

Exercise 1

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

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

Parameter `head` points to the first node in the 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. Inspect the console output, and verify that your `count` function passes all the tests before you start Exercise 2.

Exercise 2

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

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

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

This function that returns the index (position) of the first node in a linked list that contains an integer equal to `target`. If `target` is not in the list, the function should return -1.

The function should return -1 if the list is empty (parameter `head` is `NULL`).

The function uses the numbering convention that the first node is at index 0, the second node is at index 1, and so on.

Finish the implementation of `index`.

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

Exercise 3

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

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

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

This function will return the integer stored in the node at the specified index (position)

This function must handle these three cases:

- If the list is empty, the function should terminate via `assert`.
- If parameter `index` is valid (0 <= `index` < the number of nodes in the linked list), the

function should return the value stored in the corresponding node. (The index of the first node is 0, the index of the second node is 1, etc.)

- If parameter `index` is negative or >= the number of nodes in the linked list, the function should terminate via `assert`.

Finish the implementation of `fetch`.

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

Exercise 4

File singly_linked_list.c contains a complete definition of a function named `remove_last`. This function was discussed in a recent lecture. Recall that the traversal moves two pointers through the linked list. One pointer ends up pointing at the last node (which is freed). The other pointer ends up pointing at the second last node (which becomes the last node in the linked list).

During the lecture, I discussed how this function could be rewritten to use only one pointer during the list traversal; that is, instead of using two pointer variables, `p1` and `p2`, a single pointer variable is all that's needed.

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

```
IntNode *remove_last_using_one_pointer(IntNode *head);
```

Parameter `head` points to the first node in the linked list. The function terminates (via `assert`) if the linked list is empty.

This function removes the node at the rear of the linked list and returns a pointer to the first node in the modified list. (This pointer will be `NULL` if the linked list contained one node, and is now empty.)

Finish the implementation of `remove_last_using_one_pointer`. Unlike the `remove_last` function, you must use a single pointer to traverse and modify the linked list. In other words, you can't have one pointer that points to the last node and another pointer that points to the second last node.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your function passes all the tests.

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 demo/sign-out sheet.

2. The next thing you'll do is package the project in a ZIP file (compressed folder) named

linked_list.zip. To do this:

2.1. From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. If you named your Pelles C project linked_list, the zip file will have this name by default; otherwise, you'll have to edit the File name: field and rename the file to linked_list before you save it. **Do not use any other name for your zip file** (e.g., lab9.zip, my_project.zip, etc.).

2.2. Click Save. Pelles C will create a compressed (zipped) folder, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder linked_list).

3. Before you leave the lab, log in to cuLearn and submit linked_list.zip. To do this:

3.1. Click the Submit Lab 9 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag linked_list.zip to the File submissions box. **Do not submit another type of file (e.g., a Pelles C .ppj file, a RAR file, a .txt file, etc.)**

3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 3.3.

3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".