Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2014

Lab 10 - More Linked List Functions

Objective

To develop some functions that operate on linked lists.

Attendance/Demo

To receive credit for this lab, you must make the effort to finish a reasonable number of exercises and demonstrate the code you complete. **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" and must be completed before you work on Lab 11.**

General Requirements

For this lab, you'll need your linked_list project from last week's lab. You should be able to download the zip file you submitted to cuLearn, but remember to extract the project from the compressed folder (don't try to edit and build a project that's stored in a compressed folder).

You have been provided with one file:

- main_Lab10.c contains a simple *test harness* that exercises the functions that you'll define in singly_linked_list.c during this week's lab. Unlike the test harnesses provided in previous labs, this 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.

The exercises in this lab present a greater challenge than the functions you developed for Lab 9. Although the algorithms are more difficult, all of the concepts you need to know have been discussed at length in the lectures.

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. Launch Pelles C and open the linked_list project you worked on during Lab 9. (If you don't have this project, follow the instructions in Lab 9 to create it. You don't need to do the exercises from Lab 9 before starting this lab; none of the functions you'll write this week require the functions from that lab.)

2. You don't need the test harness from Lab 9, so exclude main.c from the project. To do this, go to the Pelles C project window (the window that lists all the files in the linked_list project). In the list of source files, right-click on main.c, then select Exclude from the pop-up menu. A red circle with a slash will appear beside main.c., indicating that this file will not be compiled and linked in when the project is built.

3. You'll need the test harness for this week's lab, so download file main_Lab10.c from cuLearn. Move this file into your linked_list folder.

4. You must add main_Lab10.c to your project. From the menu bar, select Project > Add files to project... In the dialogue box, select main_Lab10.c, then click Open. An icon labelled main_Lab10.c will appear in the Pelles C project window.

5. Open singly_linked_list.c in the editor and add these function definitions to the file:

```
IntNode *insert(IntNode *head, int index, int x)
{
    return intnode_construct(0, NULL);
}

IntNode *delete(IntNode *head, int index)
{
    return intnode_construct(0, NULL);
}

IntNode *delete_target(IntNode *head, int target,
                       _Bool *removed)
{
    return intnode_construct(0, NULL);
}
```

6. Open singly_linked_list.h in the editor and add the following function prototypes to the file:

```
IntNode *insert(IntNode *head, int index, int x);

IntNode *delete(IntNode *head, int index);

IntNode *delete_target(IntNode *head, int target,
                       _Bool *removed);
```

7. Build the project. It should build without any compilation or linking errors.
8. Execute the project. The test harness will run. Read the console output carefully. Read the `main` function, and determine what the tests do.
9. Execute the project. The test harness will report several errors as it runs, which is what we'd expect, because the functions you just added to singly_linked_list.c are incomplete. Read the console output carefully. Read the `main` function, and determine what the tests do.

Exercise 1 (Difficulty Level: Moderate)

In singly_linked_list.c, finish the implementation of the `insert` function, which is passed a pointer to the first node in a linked list, a (non-negative) integer index, and an integer `x`. This function will insert a new node containing `x` at the specified index (position). The function prototype is:

```
IntNode *insert(IntNode *head, int index, int x);
```

Parameter `head` points to the first node in the linked list, or is `NULL` if the linked list is empty. The function **does not** terminate the program (via `assert`) if the linked list is empty.

This 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). If `index` is invalid, the function should terminate via `assert`.

Note that this function does not replace the contents of the node (if any) that is currently at position `index`. Instead, that node will be at position `index + 1` after the new node is inserted ahead of it.

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

There are several cases you should 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.
- The function is passed a pointer to a linked list containing 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.

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

Exercise 2 (Difficulty Level: Challenging)

In singly_linked_list.c, finish the implementation of the `delete` function, which is passed a pointer to the first node in a linked list and a (non-negative) integer index. This function will remove the node at the specified index (position) from the linked list, and deallocate the node via `free`. The function prototype is:

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

Parameter `head` points to the first node in the linked list, or is `NULL` if the linked list is empty. The function should terminate (via `assert`) if the linked list is empty.

This 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*-1 (where *length* is the number of nodes in the linked list). If `index` is invalid, the function should terminate via `assert`.

The function returns a pointer to the first node in the modified linked list (this pointer will be `NULL` if the function deletes the only node in a linked list containing one node, resulting in an empty linked list).

There are several cases you should consider when designing this function.:

- The linked list is empty. The function will terminate via `assert`.
- The function is passed a pointer to a linked list containing exactly one node. There are two subcases:
    - `index` is 0 (the index of the first (and only) node). After the node is freed, the function will return `NULL`, indicating that the linked list is now empty.
    - `index` is invalid.
- The function is passed a pointer to a linked list containing two or more nodes. There are four subcases:
    - `index` is 0.
    - `index` is greater than 0 and less than the index of the last node in the linked list.
    - `index` is the index of the last node.
    - `index` is invalid.

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

Exercise 3 (Difficulty Level: Challenging)

In singly_linked_list.c, finish the implementation of the `delete_target` function, which is passed a pointer to the first node in a linked list. The function removes the first node in a singly-linked list that contains an integer equal to the specified `target` value, and deallocates the node via free. The function prototype is:

```
IntNode *delete_target(IntNode *head, int target,
                       _Bool *removed);
```

Parameter `head` points to the first node in the linked list, or is `NULL` if the linked list is empty. The function **does not** terminate the program (via `assert`) if the linked list is empty.

If `target` is found, the variable pointed to by parameter `removed` should be set to `true`; otherwise it should be set to `false`.

The function returns a pointer to the first node in the list (this pointer will be `NULL` if the function deletes the only node in a linked list containing one node, resulting in an empty linked list).

There are several cases you should consider when designing this function.:

- The linked list is empty. For this case the function will return `NULL` (the target value cannot be in an empty linked list).
- The function is passed a pointer to a linked list containing exactly one node. There are two subcases:
  - `target` is in the first (and only) node. After the node is freed, the function will return `NULL`, indicating that the linked list is now empty.
  - `target` is not in the first (and only) node.
- The function is passed a pointer to a linked list containing two or more nodes. There are four subcases:
  - `target` is in the first node.
  - `target` is only in the last node.
  - `target` is in the linked list, in any node other than the first or last nodes.
  - `target` is not in the linked list.

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 instructions are on the following page.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises before you leave the lab.

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

    - 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 the same 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., lab10.zip, my_project.zip, etc.).

    - 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. Log in to cuLearn and submit linked_list.zip.

    - Click the Submit Lab 10 link. After you click the Add submission button, drag linked_list.zip to the File submissions box. Do not submit another type of file (e.g., a Pelles C .ppj file, RAR file, a .txt file, etc.)

    - After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your file is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button.

    - Once you're sure that you don't want to make any changes, click the Submit assignment button. This will change the submission status to "Submitted for grading". **Note: after you've clicked the Submit assignment button, you cannot resubmit the file.**