CS 132 - Intro to Computer Science II - Spring 2005
Week #7 Lab Exercise and Homework #5

Week #7 Lab Exercise due: Wednesday, March 2nd, END of lab
HW #5 due: Wednesday, March 9th, beginning of lab

Purpose: review/work with linked data structures

Week #7 Lab Exercise

On the course web page, you will find versions of **node1.h** and **node1.cpp** . These are somewhat based on the versions found in Savitch and Main, although they are *not* identical. They do compile on cs-server, although they have not been thoroughly tested yet. Note that **node1.cpp** also contains the "linked list toolkit" functions mentioned in the course text.

How do I know that these slightly-adapted versions work at least somewhat? I ran a small testing main program to create a node and start a small linked list. And. for this problem, you will do so, also. Create a main function in file **nodePlay.cpp** that uses **node1.h** and **node1.cpp** and meets at least the following criteria: (note that **value_type** for node is currently **double**...)

* the following opening comment block will suffice for **nodePlay.cpp**:

```
//-------------------------------------------------------------
// File: nodePlay.cpp
// Name:
// last modified:
//
// Purpose: play with the node class and linked list toolkit a little
//
//-------------------------------------------------------------
```

* declare two node variables **node1** and **node2**; declare **node1** using the default constructor, and declare **node2** using the constructor that lets the caller specify a value for the data_field (choose any non-zero double value that you wish). Go ahead and declare three pointers to nodes, also: **head**, **tail**, and **cursor**.

* try out the node modifier member functions to set node1's data_field to **13.333** and to set node1's next_field to point to node2.

* print out a message noting that 1 = passed, 0 = failed, and then print the results of comparing the expected to the actual values for the data_field values for node1 and node2 at this point.

  (Hint: to compare real numbers, compare to see if the absolute value of the difference between actual and expected is less than some small fraction. For example, for a value **realVal** that you expect to be precise to two fractional places, to see if it is **25.23** as you expect, you might have the test:

  ```
  cout << ( abs(realVal - 25.23) < 0.001 ) << endl;
  ```

  (note that function **abs** is available from the **algorithm** library, although **iostream** includes **algorithm** as well...)

  This way, small precision differences that you might not consider to be significant in floating point numbers doesn't prevent your tests from "working".)

  At this point, you've just exercised one of **node1**'s accessor member functions.

* to try the other accessor member function for **node1**, print the results of comparing the expected to the actual values for an expression that determines the data_field of the node pointed to by node1's next_field; also verify that node2's link field contains NULL by printing the results of comparing its value to NULL.

* SO FAR, SO GOOD --- we've played a bit with the basic node members. Now for the linked list toolkit functions.

Note that SOME of the linked list toolkit functions are assuming dynamically-allocated memory; node1 and node2 are NOT dynamically allocated! SO,

* now, make head point to a **new** node with the same data_field value as node1,
make tail point to a **new** node with the same data_field value as node2,
and make the next_field of the node pointed to by head point to the node pointed to by tail.

  This is a quick-n-ugly linked list, then.

* print out the results of comparing the expected and actual results of calling **list_length** for this little linked list pointed to by **head**.

* use function **list_head_insert** to insert **14.88** at the head of this linked list. Then print out the results of comparing the expected and actual results of calling **list_length** now, as well as comparing the expected and actual results for what is in the data field in the node pointed to by **head**.

* we discussed in class the importance of being able to "walk" through a linked list; print what the contents of the data fields of list-so-far should be at this point, then write a for-loop that will use a pointer named **my_cursor** to help it to walk through the linked list and print those data fields.

(No, this is not anywhere near a full test of node and the toolkit. But it gets our feet wet.)

Compile, test, run, debug, repeat as necessary --- when you are satisfied with your code, put your name on the Next: list on the board, so that your program can be checked.

HOMEWORK #5:

Now, let's add a few additional functions to the linked list toolkit. For the tests required below, note that it needs to be obvious to the viewer whether they have passed or not (by "print 1's if passed, 0's if failed" when possible, by printing expected and actual values when that works more reasonably.)

* It is quite common for a linked list toolkit to have a function **is_empty** that returns true if the pointer-to-a-node head pointer passed to it points to an empty list, and false otherwise. (It is simply a convenience, and might hide implementation details from the user.)

  Add this function to **node1.h** and **node1.cpp**, giving it an opening comment block similar to those given for the other "linked list toolkit" functions.

  Be sure that your examples include at least one calling this for an empty list, and at least one calling it for a non-empty list.

  Make sure these examples are then run in your modified nodePlay.cpp, PRECEDING their results with a statement saying that **is_empty** is being tested.

* Let's also add a function **get_first** that returns the first **data element** in the linked list whose pointer-to-a-node head pointer is passed to it. It has a precondition --- it must only be called for a non-empty list (and you should write this to fail if an empty list is passed to it).

  Add this function to **node1.h** and **node1.cpp**, giving it an opening comment block similar to those given for the other "linked list toolkit" functions.

  Be sure that your example(s) include at least one calling this for a non-empty list (remember the course standards with regard to examples/tests that violate preconditions...).

  Make sure these example(s) are then run in your modified nodePlay.cpp, PRECEDING their results with a statement saying that **get_first** is being tested.

* And, why not add a function **get_last**, also? It returns the **data element** that is at the end of the list whose pointer-to-a-node **head** pointer is passed to it. (Why not pass the tail? Because I want you to walk through the list, of course... 8-) Part of the purpose here is to make you practice with linked-list traversal, which is an important skill --- so, doing so is **required** for this function.) It, too, has the precondition that it must only be called for a non-empty list.

  Add this function to **node1.h** and **node1.cpp**, giving it an opening comment block similar to those given for the other "linked list toolkit" functions.

  Be sure that your examples include at least one calling this for a one-element list, and at least one calling it for a more-than-one-element list.

  Make sure these example(s) are then run in your modified nodePlay.cpp, PRECEDING their results with a statement saying that **get_last** is being tested.

* You might want to get an element within the list by its position, rather than by searching for a given target value as is currently provided. **get_at** expects a pointer-to-a-node head pointer and a size_t value representing the desired position in the list --- 1 is the position of the first element in the list, 2 is the position of the second element in the list, etc. It then returns the **data element** that is at that position. It too should "insist" that its preconditions be met: it must not be called with an position *not* in the list, and it must not be called with an empty list.

  Add this function to **node1.h** and **node1.cpp**, giving it an opening comment block similar to those given for the other "linked list toolkit" functions.

  You need at least 3 examples: one where the desired position is 1 for a non-empty list, one where the desired position is for the last element in a non-empty list, and one where the desired position is "somewhere in the middle" of a non-empty list.

  Make sure these example(s) are then run in your modified nodePlay.cpp, PRECEDING their results with a statement saying that **get_at** is being tested.

* (This is the trickiest of the lot...) Finally, you might want to remove an element within the list by its position, rather than by removing it from the head or given the pointer directly to the node to remove. **remove_at** expects a pointer-to-a-node head pointer and a size_t value representing the desired position in the list --- again, 1 is the position of the first element in the list, 2 is the position of the second element in the list, etc. It then **removes** the node at that position , and returns the value that was in the node that has now been removed. It, too, should "insist" that its preconditions be met: it must not be called with an position *not* in the list, and it must not be called with an empty list.

  Add this function to **node1.h** and **node1.cpp**, giving it an opening comment block similar to those given for the other "linked list toolkit" functions.

  You need at least 3 non-failing examples: one where the desired position is 1 for a non-empty list, one where the desired position is for the last element in a non-empty list, and one where the desired position is "somewhere in the middle" of a non-empty list.

  Make sure these example(s) are then run in your modified nodePlay.cpp, PRECEDING their results with a statement saying that **remove_at** is being tested.

When you are done with all of these, create an example output file with:

```
nodePlay > hw5_out
```

Make sure, when you are done, that you have submitted to me the following files (using **~st10/132submit** on cs-server):

node1.h, node1.cpp, nodePlay.cpp, hw5_out