LAB 5, THE HIDDEN DELIGHTS OF LINKED LISTS

Questions are based on the Main and Savitch review questions for chapter 5 in the Exam Preparation section of the webct course page. In case you haven't observed this already, the review questions are sometimes strikingly similar to the exam questions. Since the answers are also posted, some of the programming assignments have been slightly changed; also, I added more questions to amuse those who can cruise through the basics. You should still stick to the style guidelines, even if this lab's code doesn't (in other words, "do as I say, not as I do", which is actually good advice most of the time). Anything that you are expected to type is in red.

Given the node1.h and node1.cxx files in http://www.cs.colorado.edu/~main/chapter5, and a suitable Makefile, all of which you can grab by typing:

```
mkdir csci2270/lab5   ## I assume you have a directory for csci2270 already
cd csci2270/lab5
cp ~main/programs/lab5/* .
```

You can ignore any "permission denied" problems you see; those would be my solution files, which you'll re-create here on your own.

Type

```
make test-node1
```

to get a preliminary build. The test-node1.cxx file contains some very simple test code, with a few function calls to the singly-linked node class (for regular linked lists) and the doubly-linked dnode class (for doubly-linked lists). You'll copy this test code to a new test file (notice the spaces between the cp command and the two filenames:

```
cp node-test1.cxx node-test2.cxx
```

Open the files in a text editor, so you can look them over.

```
emacs node1.h &       ## remember, using & leaves you with a command prompt
emacs node1.cxx &
emacs test-node2.cxx &
```

Some review of linked list ideas:

1. Linked lists as defined here contain data defined as a value_type in the header file. You could change the line in the class definition (in the node1.h file)

```
typedef double value_type;
```

to some other type, like int or char, and the list would still work just fine. In these exercises, we'll keep the double type for simplicity, but other data types are also perfectly possible.

2. Any traversal of a list (sometimes called 'walking' the list) generally involves starting a cursor node at the head of the list. For a doubly linked list, you could also start at the tail and work backwards. List walking can be done with a for loop or a while loop. In either case, we make a node* variable (yes, a pointer) called cursor, or index, or

something of reasonable meaning to most people. Here is an example of code that walks a list and prints the contents.

```
node* cursor;                    // this is a pointer to some element of the list
// start the cursor out at the head of the list, and advance node by node
// keep looping until the cursor is NULL (then we're off the tail, so done)
for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
{
        // print out the contents of a node
        cout << cursor->data() << endl;
}
// just for neatness, print an extra blank line
cout << endl;
```

If this seems confusing, compare the code above to the more familiar example of walking through an array of doubles (the array is called data), instead of a linked list of doubles starting at head_ptr. It's a very similar process. The differences are that we need to know where the last entry of the array is (so assume that we already have an int variable max_length that knows the array length):

```
double *data = new double[max_length];
// assume that some lines of code fill in the array here somehow,
// but don't assume anything besides that the array contains doubles after this
int cursor_subscript;
for (cursor_subscript = 0; cursor_subscript < max_length; cursor_subscript++)
{
    cout << data[cursor_subscript] << endl;
}
// just for neatness, print an extra blank line
cout << endl;
```

3. The end of the list is defined by the last node in the list, this node is commonly called the tail. It's a common mistake to decide that tail must always be NULL, since everyone knows that every linked list ends with a NULL, right? But tail is really the last non-NULL node before the NULL tail (otherwise, we'd never have a reason to store the tail, and you'll know that can't be correct after reading chapter 5). To find the tail, we can always say (using either a for loop or a while loop):

```
node* cursor;                    // this is a pointer to some element of the list
// start the cursor out at the head of the list, and advance node by node
// keep looping until the cursor is NULL (then we're off the tail, so done)
cursor = head_ptr;
while (cursor->link() != NULL)
    cursor = cursor->link());
```

At the end of this loop, cursor is pointing to the list's tail node.

4. Doubly linked lists are just like singly linked lists except that that each node has a back() pointer to the node before it (what's this for the head_ptr?) as well as the forwards pointer you are familiar with from regular single linked lists. Notice that for singly

linked lists, the forward pointer is located by calling link() and for doubly linked lists, it's located by calling fore().

EXERCISES

1. (Review Question 8) Edit the node1.h and node1.cxx files by adding a nonmember function that takes a list of doubles and adds them up to get a sum. What's the order of the operation?

```
// Precondition: head_ptr is the head pointer of a linked list.
// The list might be empty or it might be non-empty.
// Postcondition: The return value is the sum of all the data components
// of all the nodes. NOTE: If the list is empty, the function returns 0.0.
void sum(const node* head_ptr)
{
        double total = 0.0;     // a hint
}
```

2. (Review Question 9) Change the code in the question before this to handle products (hint: you may want to change the starting value of total). You should return 1.0 if the list is empty. What's the order of the operation?

3. (Review Question 10) Implement and test a nonmember function that can add a new node to the list at the end (containing the entry supplied) You can assume that the list uses double data (not int) for this. What's the order of the operation? Is there any way we could do better?

```
// Precondition: head_ptr is the head pointer of a non-empty
// linked list.
// Postcondition: A new node has been added at the tail end
// of the list. The data in the new node is taken from the
// parameter called entry
void list_tail_insert(node* head_ptr, const node::value_type& entry);
{
}
```

4. (Review Question 11) Implement and test a nonmember function that checks if node* p contains the same data as some node on the linked list pointed to by head_ptr.

```
// Precondition: head_ptr is the head pointer of a linked list
// (which might be empty, or might be non-empty). The pointer p
// is a non-NULL pointer to some node on some linked list.
// Postcondition: The return value is true if the data in *p
// appears somewhere in a data field of a node in head_ptr's
// linked list. Otherwise the return value is false.
// None of the nodes on any lists are changed.
bool data_is_on(const node* head_ptr, const node* p)
{
}
```

5. (Review Question 12) Implement and test a nonmember function that checks if node* p is a node on the linked list pointed to by head_ptr. Your code for the last question may serve as a starting point.

```
// Precondition: head_ptr is the head pointer of a linked list
// (which might be empty, or might be non-empty). The pointer p
// is a non-NULL pointer to some node on some linked list.
// Postcondition: The return value is true if p actually points to
// one of the nodes in the head_ptr's linked list. For example,
// p might point to the head node of this list, or the second node,
// or the third node, and so on. Otherwise the return value is
// false. None of the nodes on any lists are changed.
bool is_on(const node* head_ptr, const node* p)
{
}
```

6. (Review Question 13) Implement and test a nonmember routine that adds a new node containing 0.0 to the list after the previous_ptr node, reconnecting the list as needed. What's the order of the operation?

```
// Precondition: previous_ptr is a pointer to a node on a linked list.
// Postcondition: A new node has been added to the list after
// the node that previous_ptr points to. The new node contains 0.
void list_insert_zero(node* previous_ptr)
{
}
```

7. a) How long does it take to add a new element to the front of the list?

b) How long does it take to add a new element to the back if there's a tail pointer? What if there's no tail pointer?

c) How long does it take (ballpark argument) to add a new element somewhere in the middle of the list, assuming no cursor pointer?

d) What can the cursor pointer do to make things like addition/removal faster? What effect does the precursor pointer have, given the cursor?

8. If my test program has the following two lines of code:

```
const node *fred;
fred = head->link();
```

Which of the two link() methods is getting called? How can C++ tell?

9. a) Say you're writing a function that may cause a list you're using to have a new head node after the function completes. What should you say to allow for this in the parameter list?

b) What if the head node says the same but the data it holds changes?

c) What if the head node and the data both stay the same?

10. Suppose I want a method to initialize my linked list to random whole numbers between -15 and 13, for some reason (like –4, 2, 7, -12). Supply me with the missing line or lines in the for loop below to accomplish this.

```
void initialize_list(const node* head_ptr)
{
        node* cursor;
        for (cursor = head_ptr; cursor != NULL; cursor = cursor->link)
                _____________________________________;
}
```

11. (Review Question 26) Suppose we switch to using the dnode class for doubly linked lists, which is defined in dnode.h. Write and test a function to remove a node other than the head or the tail from a doubly linked list, and remember not to leak memory. Notice that you are not given the head of the list (but you can get there, if you need to).

```
// Precondition: p is a pointer to a node on a doubly linked list.
// Postcondition: p has been removed from the list, and links to p
// now point to the appropriate nodes on either side of it. The
// memory for node p has been returned to the heap.
void delete_dnode(dnode* p)
{
}
```

12. Given a change in the last question, modify the code to handle the head and tail cases.

```
// Precondition: p is a pointer to a node on a doubly linked list.
// Postcondition: p has been removed from the list, and links to p
// now point to the appropriate nodes on either side of it (if they exist).
// The memory for node p has been returned to the heap.
void delete_dnode(dnode*& head_ptr, dnode*& tail_ptr, dnode* p)
{
}
```

13. a) Write out, by hand, the steps needed for a routine that adds a new node into a doubly linked list so that the new node becomes the head.

b) Write the steps needed for another routine that adds a new node into a doubly linked list so that the new node becomes the tail.

c) Work out the steps for a third routine that adds a new node into a doubly linked list after the dnode* p, but never has to do this at the head or the tail of the list.

d) Can all of these be combined together into one routine, maybe using default arguments, to add the node anywhere in the list including the head or the tail?

```
// Precondition: head_ptr is the head pointer of a linked list
// (which might be empty, or might be non-empty). tail_ptr is
// the tail pointer of the same linked list (which might be empty,
// or might be non-empty). The pointer p is either a non-NULL
// pointer to some node on the linked list, or NULL.
```

```
// Postcondition: A new node containing the data in &entry has been
// added to the list.  If p is NULL, the new node is now the head.  If p
// points to the tail, then the new node is now the tail.  Else, the new
// node is added after the node *p points at, somewhere in the middle
// part of the list.
void insert_dnode(dnode* p, dnode*& head_ptr, dnode*& tail_ptr, const
node::value_type& entry)
{
}
```

MORE LIST QUESTIONS

These questions do not any have posted answers, at least not yet. If you can answer all of them for the TA, you can leave early.

A. Explain what happens in the following code.

```
node* egg = NULL;              // why set the initial list pointer to NULL?
list_head_insert(3, egg);
list_head_insert(5, egg);      // what's the egg list look like now?
node* cheese = NULL;
list_head_insert(9, cheese);
list_head_insert(13, cheese);  // what's the cheese list look like now?
cheese = egg;                  // what's the cheese list look like now?
list_clear(egg);               // what's the cheese list look like now?
```

B. Notice that one of the list operators is a copy method. Why would we NOT overload the assignment operator = for nodes to copy a list to another list in this case? A hint: what would it do to the linked list code in review question #3, where you say

```
node* cursor;
cursor = head_ptr;
```

if an assignment operator existed? Explain whether this might pose a problem.

C. Write a non-member function that takes 2 linked lists as constant arguments, and returns a new linked list that contains every number the two lists have in common. One or both of the input lists may be empty, or may contain numbers. Don't change either of the two input lists. Assume that each list has at most only one copy of any particular number (so that the list 3, 5, 6, 7, 3, which has 2 copies of 3, is not allowed for this question).

```
node* intersection(const node* head1, const node* head2)
{
}
```

If list1 contains the numbers 3, 2, 4, 9, 8, 1, 0, 5 and list2 contains the numbers 0, 1, 7, 6, 4, 5, 9, then

```
node* intersect_list;
intersect_list = intersection(list1, list2);
```

should return a list of the numbers 0, 1, 4, 5, 9 in any order.

D. Write a non-member function that that takes 2 linked lists as constant arguments, and returns a new linked list that contains every number the two lists do NOT have in common. Use the same assumptions as in part C.

```
node* disjunction(const node* head1, const node* head2)
{
}
```

If list1 contains the numbers 3, 2, 4, 9, 8, 1, 0, 5 and list2 contains the numbers 0, 1, 7, 6, 4, 5, 9, then

```
node* disjunct_list;
disjunct_list = disjunction(list1, list2);
```

should return a list of the numbers 2, 3, 6, 7, 8 in any order.

E. What's the order of the operations described in parts D and E? If the lists were sorted, would this order change? Explain. Rewrite the routines in D and E assuming that list1 and list2 are already sorted smallest to largest.

F. Suppose we have a doubly linked list: 3, 1, 8, 5, 4, 2, 9, 6, 7, 0. Write a routine that swaps any two nodes in the list, resetting their forward and back pointers as needed. You should of course check that nodes a and b are in the list (however you like) and then exchange them in the list if both are present. Notice that the head pointer may change as a result of this operation. And remember to pay attention to the special cases, like:

a and b are separated by one node in the list,
a and b are adjacent to each other in the list,
a or b is the head or the tail of the list.

```
void swap(const node*& head_ptr, node*&a, node *& b)
{
        // up to you
}
```