CIS 291 – Data Structures in C++ - Spring 2005
Week 11 Lab Exercise
Week 11 Lab Exercise due: by END of LAB on Tuesday, April 5th

Purpose:
Reading and thinking about one implementation of binary trees.

In CIS 291 HW #9, you are getting familiar with using a binary_tree ADT that attempts to "hide" the implementation details --- to provide a concept of a binary tree that can then be implemented in a variety of ways.

As you are seeing in the homework assignment, the fact that pointers are used is hidden via the abstraction of a "current node" --- one node in the tree is the current node, and the user can change what node is the current node via methods such as shift_to_root, shift_left, and shift_right. A user can verify that there is something to shift to via such methods as hash_left_child and has_right_child, and a user can add nodes or subtrees to the current node using methods such as add_left and add_right_subtree. With these methods, one could switch out the linked nodes with, for example, an array of appropriate nodes instead, if one wished, and the user would not really care (as long as performance remained reasonable).

The homework does have you "dig" a little into the particular implementation being used --- a rather classic linked implementation, for which you are implementing a classic recursive in-order traversal. But, for this lab exercise, you will read and consider this binary tree implementation both from the higher ADT level and from the lower implementation level.

Answer the following questions on paper; after coming up with your initial answers, you may discuss them with another student before getting your answers checked, if you wish. When you are ready, put your name on the "Next:" list on the board so that your answers can be checked.

1. What are the specific data fields in the class **binary_tree_node**?

____

2. What are the names of the accessor methods for class **binary_tree_node**? (Note that those that return pointers or references have two versions, a const and a non-const version --- you only need to list their names one time below...!)

____

(Aside: the need for two separate versions of an "accessor"/"observer" method that returns a pointer or a reference is a C++ quirk --- you see, if you return a const version of something, it must not subsequently be used to change something. But, if you only provided a non-const version, const callers would not be able to call the accessor/observer method. The good news is, once you provide both versions, the compiler decides for you which should be called whenever you call that accessor/observer method --- that is, the caller doesn't really know, or care, that there are two versions. To the user, the accessor/observer method simply works when "it" is called.)

3. What are the **names** of the modifier methods for class **binary_tree_node**?

____

4. What are the **names** of the "linked tree node toolkit" non-member functions within **binary_tree_node.h** (and implemented within **binary_tree_node.template**)?

____

5. Consider the following files:

(A) **binary_tree_node.template**: the implementation file for template class **binary_tree_node**
(B) **binary_tree.template**: the implementation file for template class **binary_tree**, in this case implemented as a linked structure of **binary_tree_node** instances
(C) **bst.template**: the implementation file for template class **bst (binary search tree)**, in this case implemented using a **binary_tree** instance as a private data field.
(D) **my_tree_app.cpp**: the implementation file, complete with a main() function, for my program that happens to use a **binary_tree** instance.
(E) **my_bst_app.cpp**: the implementation file, complete with a main() function, for my program that happens to use a **bst** instance.
(F) **my_both_app.cpp**: the implementation file, complete with a main() function, for my program that happens to use both a **binary_tree** instance and a **bst** instance.

(a) In which of the above could you reasonably expect to see calls to methods **get_left** and **get_right** (both of which return pointers to **binary_tree_node** instances); list all that apply:

____

(b) In which of the above could you reasonably expect to see calls to methods **contains** and **add**?

____

(c) In which of the above could you reasonably expect to see calls to methods **add_left_subtree** and **add_right_subtree**?

____

6. Consider: what if you wanted to add another "linked tree node toolkit" non-member function to **binary_tree_node** called **get_tree_depth**. Given a pointer to a node, it returns the depth of the (sub)tree whose root is that node.

This can most easily be done using **recursion** --- you'll remember that a recursive function is a function that calls itself.

Now, sometimes, a function uses one or more **auxiliary**, helper functions to do its work --- and sometimes an auxiliary, helper function happens to be recursive. A common "pattern" using an auxiliary recursive function is as follows: the public function that the user knows about sets things up appropriately, and then calls the private auxiliary, helper function, which calls itself until it comes up with the desired result, which the public function then returns to the user.

(a) Let's consider a recursive function that is not auxiliary. If **get_tree_depth** is included as a "linked tree node toolkit" non-member function, then it can simply be a recursive function.

How? One useful approach, when thinking recursively, is to first figure out the "base" cases --- when do you not need to do any additional work to know the answer? When does the function NOT need to call itself? These cases are called "base" cases because they form the "bases", or stopping points, for the recursion --- they keep the function from calling itself forever!

What is a "base" case for the depth? Well, consider our definitions for depth --- we said that an empty tree has depth **-1**, and that a tree consisting of one node has a depth of **0**. Those are good "base" cases!

So --- given a binary_tree_node pointer, how can you tell that it points to an empty tree? If you can tell that, then you can simply return a depth of **-1** for such a pointer.

And, given a binary_tree_pointer, how can you tell if it points to a leaf? If you can tell that, then you can simply return a depth of **0** for such a pointer.

Ah, but what if it is NOT either of the above? THEN you have a recursive case!

BUT --- it turns out to be pretty easy, also! Because, the very definition of depth is recursive; the depth of a tree that isn't empty or a leaf turns out to be
(1 + max(depth of its left subtree, depth of its right subtree))

And, the library **cstdlib** includes a **max** function!

Write out the **body** for **get_tree_depth** below, as it would appear in **binary_tree_node.template**:

```
// get_tree_depth
//
// preconditions: none
// postconditions: depth of the (sub)tree whose root is <node_ptr>
//    is returned
//
template <typename Item>
int get_tree_depth(const binary_tree_node<Item>* node_ptr)
{








}
```

7. Consider: how could you determine the value in the "leftmost" element within a binary_tree instance?

Write a fragment of C++ code that, assuming that a binary_tree instance **myTree** has ALREADY been declared, will shift to the root and then shift to the leftmost element within myTree, and then attempt to print its value.

8. Within a BST --- what will be true of the leftmost node? what will be true of the rightmost node?

What if we were to add **get_max** and **get_min** methods to **bst?**

Now, I don't know what you called your **binary_tree** data field within your **bst** implementation --- but assume, for this question, that I called it **bst_tree**.

Write the BODY ONLY of **get_max**: