CIS 291 – Data Structures in C++ - Spring 2005
Homework #9
HW #9 due: THURSDAY, April 7th, BEGINNING of lecture

Purpose: practice with binary trees and binary search trees, practice with inorder tree traversal.

How this will be turned in:
Use **~st10/291submit**, called from the directory on cs-server where the files you wish to submit are stored.

Homework #9

1. On the course web page, you will find the pseudo-UML for a **binary_tree** class.

Because this ADT is trying to hide the underlying implementation, it uses the concept of a "**current node**" to hide what is actually going on underneath.

You can grab the value of the "current node" with the **retrieve** method; you can shift the current node to be the root **(shift_to_root)**, to be down to the left from the current node **(shift_left),** and down to the right from the current node **(shift_right),** and so on.

Nodes are added to the binary_tree instance relative to the "current node"; you can **add_left** to add a left child to the "current node", IF **has_left_child** is false, for example. You can even add subtrees.

You need to carefully read and consider this pseudo-UML; then, see if you can read the big'n'sprawling **test_binary_tree.cpp** example file posted. It is trying to test many of the methods in **binary_tree**.

And, there are files **binary_tree.h** and **binary_tree.template** posted that attempt to implement this UML. Note that it adds a copy constructor, destructor, and assignment operator, as is expected for a linked implementation.

Oh, and if it is linked, you'd expect to need a node class --- and, indeed, you will find **binary_tree_node.h** and **binary_tree_node.template** available, too. As described in class, these nodes have left and right pointers (but not parent pointers, in this particular implementation).

Copy the files **binary_tree_node.h**, **binary_tree_node.template**, **binary_tree.h**, **binary_tree.template**, and **test_binary_tree.cpp** to cs-server. Adapt **test_binary_tree.cpp** to FIRST print a line containing YOUR NAME, and then compile and run **test_binary_tree.cpp**, redirecting its output into **291hw09_1_out** for submission. That will show that you successfully transferred the files and tested them out.

Submit your resulting **291hw09_1_out**. (You'll be making further modifications to **test_binary_tree.cpp**, so do not submit it yet!)

2. For #3, we are going to want to implement an in-order traversal of a binary tree. It turns out that this is most easily accomplished if helper functions/methods are added to **binary_tree_node** and **binary_tree**.

You see, the recursive version that works so well is implemented most simply at the level where you know implementation details. SO --- in **binary_tree_node.h** and **binary_tree_node.template**, add the following NON-MEMBER function **node_print_inorder**, that takes a pointer to a binary_tree_node and prints the results of an in-order traversal, where a node visit results in that node's value being printing on its own line to the screen:

...in **binary_tree_node.h**:

```
// preconditions: node_ptr is a pointer to a node in a
//    binary tree (representing the root of a (sub)tree), or
//    it may be NULL to indicate an empty (sub)tree
// postconditions: prints to the screen the nodes in that
//    (sub)tree in in-order traversal order
//
template <typename Item>
void node_print_inorder(const binary_tree_node<Item>* node_ptr);
```

...in **binary_tree_node.template**:

```
// node_print_inorder
//
// preconditions: node_ptr is a pointer to a node in a
//    binary tree (representing the root of a (sub)tree), or
//    it may be NULL to indicate an empty (sub)tree
// postconditions: prints to the screen the nodes in that
//    (sub)tree in in-order traversal order
//
template <typename Item>
void node_print_inorder(const binary_tree_node<Item>* node_ptr)
{
      you fill in the implementation!
}
```

Now, with that done, **binary_tree.h** and **binary_tree.template** should have added the now-very-simple **tree_print_inorder** METHOD (this one IS part of the binary tree CLASS!) added, which calls **node_print_inorder** appropriately:

... in **binary_tree.h**:

```
    // postconditions: contents of the tree's entries are
    //    written to the screen, one entry per line, in
    //    inorder order.
    //
    void tree_print_inorder();
```

...in **binary_tree.template**:

```
// tree_print_inorder
//
// postconditions: contents of the tree's entries are
//    written to the screen, one entry per line, in
//    inorder order.
//
template <typename Item>
void binary_tree<Item>::tree_print_inorder()
{
      you fill in the implementation!
}
```

Finally, add an appropriate testing call of **tree_print_inorder** to the **test_binary_tree.cpp** that you modified in problem #1; be sure to include a printout of what it SHOULD print out, before you call the function (and let it print what IT prints out...). Redirect the output of your modified **test_binary_tree.cpp** into a file **291hw09_2_out** for submission.

Although you have modified binary_tree.h and binary_tree_node.h also, you are only required to submit your resulting **binary_tree.template**, **binary_tree_node.template**, **test_binary_tree.cpp**, and

291hw09_2_out files.

3. Now that you have your results from problems #1 and #2, you will use them to help to create a very basic version of a binary search tree template class, named **bst**. Note that it does not permit duplicate values.

You'll find the pseudo-UML for **bst** that you are required to implement linked along with this assignment, and you'll find a cursory **test_bst.cpp** that you are expected to use to test your resulting implementation.

We've mentioned how you can use one ADT in implementing another --- here, you will be trying this concept out. Your **bst** class <u>must</u> use a private instance of a **binary_tree** within it. (After all, a bst is really just a binary_tree that meets certain additional criteria --- in bst, we'll ensure that it does by severely limiting what a user can do with a bst, and ensuring that, whenever an element is **add**ed, that it is added in such a way that the bst still meets the binary search tree properties after the addition.)

NOTE the following additional requirements and suggestions

* your bst class must not directly use the **binary_tree_node** class. It must contain a private data field that is a **binary_tree** instance.

* you are required to include and maintain an explicit counter/size data field in class **bst**, to keep track of the number of elements currently within the bst.

* you must explicitly implement a destructor, copy constructor, and assignment operator in your **bst** template class.

* remember that your bst class has NO IDEA **how** binary_tree is implemented --- it only knows the public methods of binary_tree. If you try to call binary_tree_node-related stuff in bst, it will not work. However, within bst's implementation, your binary_tree data field can certainly call any of the binary_tree methods mentioned in the binary_tree pseudo-UML (and also assignment).

* a hint for the **add** method: how can you add an element to a bst, and make sure that the tree is still a bst afterwards? After you have made sure that the element isn't there --- note **add**'s precondition! --- then try to search for the element being added; the point where you can say that it is not there is where it can be safely inserted, you see. Try this "by hand" a few times until you see what I mean.

* Add a print out of **your name** to the beginning of **test_bst.cpp**, and be sure to run it with your resulting implementation. FEEL FREE to add additional tests to this, if you wish (following course testing standards, of course). Redirect its results into **291hw09_3_out** for submission.

You should submit your resulting **bst.h**, **bst.template**, **test_bst.cpp**, and **291hw09_3_out**.

When you are done, submit the following files using **~st10/291submit**:

test_binary_tree.cpp, **291hw09_1_out**
binary_tree.template, **binary_tree_node.template**, **test_binary_tree.cpp**, **291hw09_2_out**
bst.h, **bst.template**, **test_bst.cpp**, **291hw09_3_out**