CS 132 - Intro to Computer Science II - Spring 2005
WEEK #11 LAB EXERCISE and Homework #8

Week #11 Lab Exercise due: Wednesday, April 6th, END of lab
HW #8 due: Wednesday, April 13th, 1:00 pm

WEEK #11 LAB EXERCISE

* On the course web page, you will find **binary_tree_node.h** and **binary_tree_node.template**--- this is a node class for a linked implementation of a binary tree. In the tradition of Savitch and Main's **nodeX.h**, it also includes some "binary tree toolkit" functions.

  Note that the .h file includes an explanation from Savitch and/or Main about how it returns *references* to pointers --- thus, the use of & in a number of the functions' return types.

  Note that it also includes both const and non-const versions of the accessors/observers get_data, get_left, and get_right; you may recall a similar issue in node.h. This, you may recall, is because we need one version when we are permitted to change a node, and another version when we are not permitted to change a node, when an accessor/observer returns a pointer to something; see p. 218-221.

* mind you, we want to **hide** this implementation within an ADT. **binary_tree.h** and **binary_tree.template** attempt to do so. A pseudo-UML diagram for this binary_tree ADT has also been posted. You should read this carefully; note the use of a "current node" concept to hide how the tree is actually implemented.

1. Create a file **tree_play.cpp** that contains a main( ) function that uses template class **binary_tree** to:

  * create an empty binary tree **name_tree** able to contain strings;

  * put a name of your choice into name_tree's root;
  * put another name of your choice as name_tree's left child;
  * put another name of your choice as name_tree's right child;

  * call print_tree with a depth of 1 to show name_tree's contents. (notice that it printed the tree "sideways")

  Before getting to the next steps in **tree_play.cpp**... Note, again, that this tree implementation has a concept of a "current node" --- you can retrieve the current node's contents, change them, and more, and you can change the current node via shifting to another node. So, your main function should also:

  * call shift_to_root for name_tree, then print what name should be in the root, then call retrieve for name_tree within a statement printing the value at the current tree node (which should be the root, thanks to shift_to_root).
  * call shift_left for name_tree, then print what name should be in the root's left child, then call retrieve within a statement printing the value at the current tree node (which should be left child of the root, thanks to shift_left from previous current node).
  * call shift_to_root and then shift_right for name_tree, then print what name should be in the root's right child, then call retrieve within a statement printing the value at the current tree node

(which should be the right child of the root, thanks to the two shifts done).

Hopefully, these will give a you a little light familiarity with this binary_tree ADT.

When you are happy with this, put your name on the "Next:" list to get your work checked over. Your work must be completed and checked over before the end of lab.

HOMEWORK #8:

1. What if you happened to fill a binary_tree instance so that it met the binary search tree property? Then, using binary_tree's member functions, you could quite reasonably perform a binary search on that tree to see if a given target was within it.

Create a **template** function **tree_search** that accepts a binary_tree of some type and a target of the same type. It **assumes** that the type of items within the tree have <, >, and == implemented for that type, and it **assumes** that the binary tree passed does indeed satisfy the binary search tree property. It returns true if the target is indeed within the tree, and returns false if it is not. (Notice that passing an empty tree **should** be "legal" with this function --- it should simply return false, then, since the target obviously is not within an empty tree.)

Note: expressing the tree(s) used in the Examples can be tricky here! I'll accept a readably-typed depiction of the trees (each labeled with its name, of course), followed by the usual calls-and-what-they-return. Be sure to test an empty tree, seeking at least one value that is a leaf, seeking at least one value that is a non-root interior node, seeking at least one value that is the root, and seeking at least one value smaller than any in the tree, seeking at least one value greater than any in the tree, and at least one value "within" the range of the tree but not in the tree.

When you are happy with your **tree_search.template** and **test_tree_search.cpp**, run:

```
test_tree_search > 132hw8_1_out
```

...and submit **tree_search.template**, **test_tree_search.cpp**, and **132hw8_1_out**.

2. You might notice that **shift_up** is not an operation in the current binary_tree.

To more easily support this operation, a **parent pointer** within the binary tree node would be useful. That requires modifying **binary_tree_node** accordingly, however.

Modify **binary_tree_node.h**, **binary_tree_node.template**, **binary_tree.h**, and **binary_tree.template** so that each tree node now has a pointer to its parent (which is set to NULL for a root, of course), and add member function **shift_up** to our binary_tree template class. Also change the current implementation of **has_parent** to make direct use of this new pointer. (Be careful --- when does the parent need to be changed? updated?)

(another tip: be careful with **tree_copy**! The statements have to be **reordered** from their current ordering so that you create the new root node BEFORE copying the left and right subtrees; OTHERWISE, new left and right pointers cannot be made to point to the new parent! Remembering those modifier member functions can be helpful here.)

To your **tree_play.cpp** from the lab exercise, add at least two more levels to your tree (although the tree does not have to be full nor complete!), and shift to the lowest level, then shift up repeatedly to the root, each time (after each shift_up call) printing the expected and the actual value of the node that is now the current node.

When you are satisfied, run:

```
tree_play > 132hw8_2_out
```

...and (after problem #3) submit your **binary_tree_node.h**, **binary_tree_node.template**, **binary_tree.h**, **binary_tree.template**, **tree_play.cpp**, and **132hw8_2_out**.

3. Let's make another change to **binary_tree_node**. Let's add a **non-member** "binary tree node toolkit" **template** function named **tree_depth** that computes the **depth** of a (sub)tree, given a pointer to the node that is the root of that (sub)tree.

Your function **tree_depth** *must* be recursive; and, you need at least two base cases in your solution. (Remember, too, that a tree with 1 node is considered to have depth 0, and that an empty tree is considered to have depth -1.) HINT: If your function's body is more than 6 lines long (not counting curly-brace lines), you have gone astray...

Of course, once you have **tree_depth**, it should be trivial to add a **member** to **binary_tree** that returns the depth of the subtree beginning at the "current node". Let's call this new member **get_tree_depth**, and add it to class **binary_tree**.

I'm hoping it will suffice, to at least somewhat test both of these, to add to **tree_play** calls to **get_tree_depth** on at least :

* an empty tree,
* a tree with only 1 node,
* a tree with at least depth 3 whose "current node" is the root,
* a tree with at least depth 3 whose "current node" is a non-root interior node, and
* a tree with at least depth 3 whose "current node" is a leaf.

(I can think of a lot more possible interesting permutations, actually, so feel free to add to these if you'd like a more robust test.)

When you are satisfied, run:

```
tree_play > 132hw8_3_out
```

...and submit your **binary_tree_node.h**, **binary_tree_node.template**, **binary_tree.h**, **binary_tree.template**, **tree_play.cpp**, and **132hw8_3_out**.

And, when you are satisfied with all of the above, submit them using **~st10/132submit** on cs-server.

tree_search.template, test_tree_search.cpp, 132hw8_1_out,
binary_tree_node.h, binary_tree_node.template,
binary_tree.h, binary_tree.template, tree_play.cpp,
132hw8_2_out,
132hw8_3_out