Data Structures (810:052) Lab 9 Name:__________________

Objectives: You will gain experience:

- implementing binary search tree, BST, operations, including some recursive ones

Download the following file to your desktop: http://www.cs.uni.edu/~fienup/cs052s10/labs/lab9.zip
Extract this file to the Desktop by right-clicking on lab6.zip icon and selecting Extract All.

Part A: The `lab9.zip` file you downloaded and extracted contains a `IntBinaryTree` folder with a Visual Studio C++ project file: `IntBinaryTree.sln` inside. Double-click on it to open this project in Visual Studio. Your task is to implement these additional BST operations:

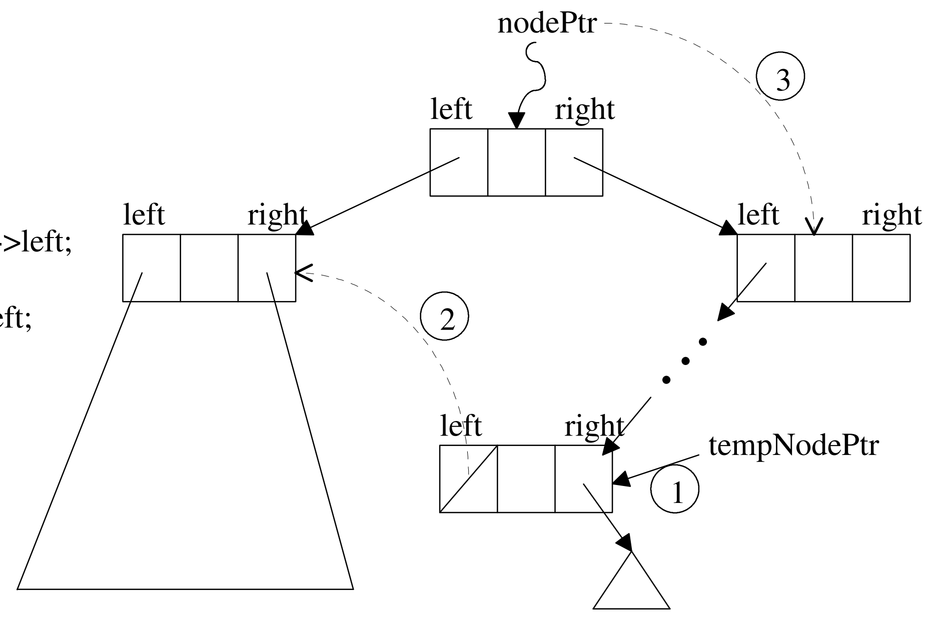
- size - returns the number of nodes in the BST (Hint: I added a new data member, called numberOfNodes, but you'll need to add code to update it on insertNode, remove, and in the constructor)
- clear - empties the tree of all nodes (Hint: call the `destroySubTree` function)
- height - returns the height of the tree (Hint: model this after one of the recursive traversals)

The `main.cpp` file contains an interactive test driver to test your new functions.

After you have implemented and tested your functions, raise your hand and demonstrate your program.

Part B: Recall from yesterday's discussion about the `remove` function, that it uses the recursive `deleteNode` function to "walk" down the tree to find the `num`(ber) to be deleted. Then, `makeDeletion` is called and passing a reference to the TreeNode pointer containing the `num` to be deleted. Before deleting this node **if it has two children**, we must find the *correct home* for it's left-subtree by moving tempNodePtr to the right child and then down the left links until there are no more links.

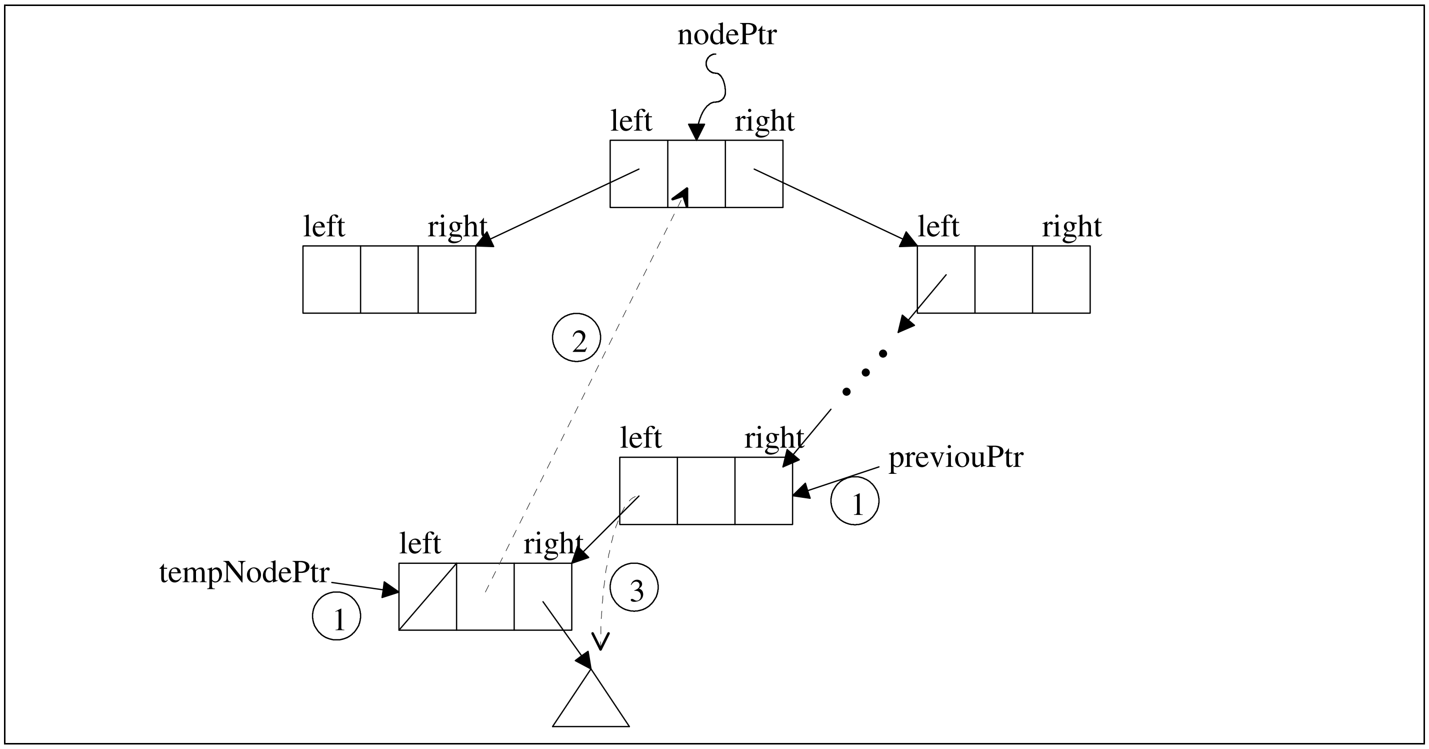
```
// If the node has two children.
else {
    // Move one node the right.
    tempNodePtr = nodePtr->right;
    // Go to the end left node.
    while (tempNodePtr->left)  ①
        tempNodePtr = tempNodePtr->left;
    // Reattach the left subtree.  ②
    tempNodePtr->left = nodePtr->left;
    tempNodePtr = nodePtr;
    // Reattach the right subtree. ③
    nodePtr = nodePtr->right;
    delete tempNodePtr;
} // end if
```



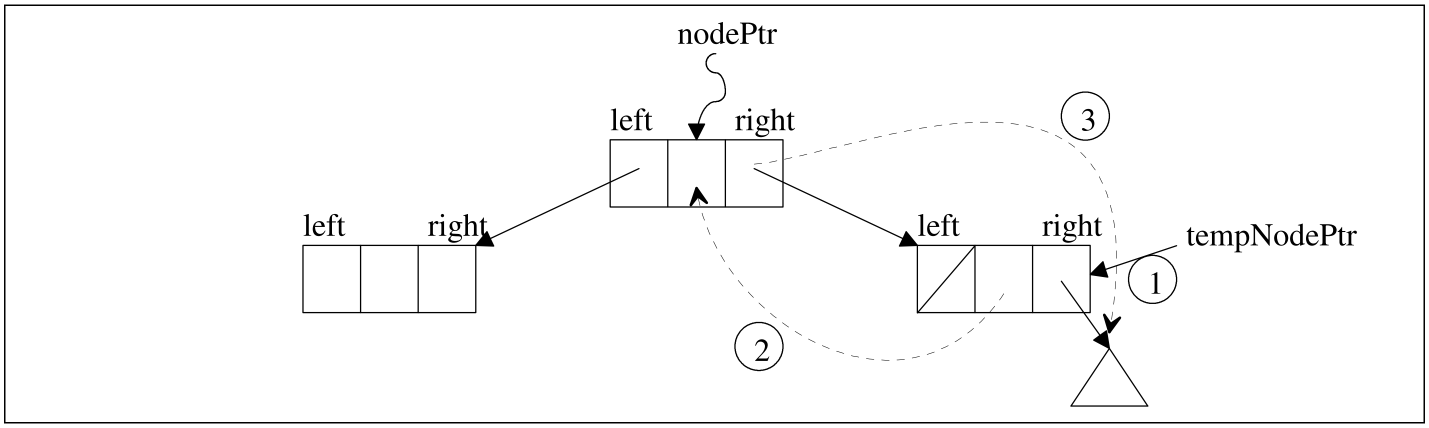
While technically correct (i.e., maintains a BST), the left-subtree of the node being deleted is moved down the tree, so all nodes in that subtree are further down the tree.

Your job is to implement a better alternative for deleting a node with two-children which does NOT move any values down the tree, but instead moves one up the tree. The steps are as follows:

1) move tempNodePtr just like above, but trail it by a previousPtr
2) copy the value of the node pointed at by tempNodePtr into the node pointed at by nodePtr (wiping out the value to be deleted)
3) reroute a pointer in the node pointed at by the previousPtr around the node pointed at my tempNodePtr
4) delete the node pointed at by tempNodePtr



The only tricky part is that step (3) really has two cases. If the right-child of the node to be deleted does not have a left-child, then the while-loop body will not execute and the situation will look like:



In the interactive-tester main program, switch the comment from remove2 to remove, so you can test your `makeDeletion`2 function.

After you have implemented your `makeDeletion2` function and tested it via the `remove2` function, raise your hand and demonstrate your program.

Part C: Uncomment the code at the bottom of `main.cpp` to complete the tables below.

	Initial BST Height	**Max. Height During removes**	**Max. Height During remove2s**
50,000 Elements			

	Time to fill BST in random order	**Initial BST Height**	**Time to perform removes**	**Time to perform remove2s**
200,000 Elements				

After you have completed the above tables, raise your hand and explain your results.