CSE 373 -- Autumn 1999 -- Assignment #3

(*due Wednesday, October 27*)

1. **Draw the trees resulting from the following sequences of operations (show only the final data structure, not all of the intermediate steps). Assume that the < and > operators have been implemented for `mystring` to represent alphabetical order (*e.g.*, "Olive" < "Pickle")**

   (a) *Use lazy deletion (be sure you clearly mark which nodes are deleted)*

   ```
   BinarySearchTree T;
   T.insert("Ed");
   T.insert("Steve");
   T.insert("Gaetano");
   T.insert("Oren");
   T.remove("Steve");
   T.insert("Brad");
   T.insert("AJ");
   T.insert("Sun Liang");
   T.insert("Maria");
   T.remove("Oren");
   T.remove("Ed");
   ```

   (b) *Repeat (a), but actually delete nodes that are removed (for the 2-child case, use the "smallest right descendent" approach).*

   (c) *Repeat (a), but assume that the BinarySearchTree is implemented using an AVL tree (with lazy deletion).*

2. **Given the tree in Figure 4.69, show the order that the nodes will be visited using (a) a preorder, (b) a postorder, and (c) an inorder traversal.**

3. **Given a child/sibling tree implementation:**

   ```
   struct TreeNode {
     int element;
     TreeNode *firstChild;
     TreeNode *nextSibling;
   };
   ```

   write a routine `void PrintTreePost(TreeNode* node)` that prints the tree's nodes in postorder.

4. **Imagine that we've added `height` and `depth` fields to our `TreeNode` class that will store each node's height and depth in the tree:**

```
template <class Comparable>
class BinaryNode {
  Comparable data;
  BinaryNode* left;
  BinaryNode* right;
  int depth;
  int height;
};
```

Write a function `void SetHeightDepth(BinaryNode* node)` that takes the root of a binary tree as input. The routine should set the height and depth fields of every node to their correct values. You may use a helper function or two to achieve this.

5. **Imagine that we have a new search tree implementation designed to make the overall search tree shallower by storing two data items and three children per node:**

```
template <class Comparable>
class NewTreeNode {
  Comparable item1;
  Comparable item2;
  NewTreeNode* childA;
  NewTreeNode* childB;
  NewTreeNode* childC;
};
```

The search tree property for such a tree would be that `item1` is less than `item2`. In addition, all values stored in subtree `childA` are less than `item1`, all values in subtree `childC` are greater than `item2`, and all values in subtree `childB` are between `item1` and `item2`.

(a) *Asymptotically speaking, what is the minimum depth for a tree of N values using this representation? Why?*

(b) *What is the maximum asymptotic depth for a tree of N values? Why?*

(c) *Give a brief explanation of how the traditional BST* `insert()` *operation would have to be modified to work with this approach. This should just be a few sentences indicating some of the differences.*

(d) *Do you think this approach is worthwhile? (Do the benefits outweigh the disadvantages?) Why or why not?*