Solutions for Exercises Unit 3

This is a non-graded exercise which should be posted to your learning journal.

In **Subsection 9.4.2,** the textbook says that "if the [binary sort] tree is created by inserting items in a random order, there is a high probability that the tree is approximately balanced." For this exercise, you will do an experiment to test whether that is true.

In the Introduction to this Learning Guide, you read that the **depth** of a node in a binary tree is the length of the path from the root of the tree to that node. That is, the root has depth 0, its children have depth 1, its grandchildren have depth 2, and so on. In a balanced tree, all the leaves in the tree are about the same depth. For example, in a perfectly balanced tree with 1023 nodes, all the leaves are at depth 9. In an approximately balanced tree with 1023 nodes, the average depth of all the leaves should be not too much bigger than 9.

On the other hand, even if the tree is approximately balanced, there might be a few leaves that have much larger depth than the average, so we might also want to look at the maximum depth among all the leaves in a tree.

For this exercise, you should create a random binary sort tree with 1023 nodes. The items in the tree can be real numbers, and you can create the tree by generating 1023 random real numbers and inserting them into the tree, using the usual treeInsert() method for binary sort trees. Once you have the tree, you should compute and output the average depth of all the leaves in the tree and the maximum depth of all the leaves. To do this, you will need three recursive subroutines: one to count the leaves, one to find the sum of the depths of all the leaves, and one to find the maximum depth. The latter two subroutines should have an int-valued parameter, depth, that tells how deep in the tree you've gone. When you call this routine from the main program, the depth parameter is 0; when you call the routine recursively, the parameter increases by 1.

Build your solution on code and principles from section 9.4.2.

Discussion

To create the tree, we copied the *TreeNode* class and the *insertTree()* subroutine from Subsection 9.4.2, and I changed the type of the items in the tree from *String* to double. The main program uses a for loop to add 1023 random real numbers to the tree:

```
for (int i = 0; i < 1023; i++)
treeInsert(Math.random());
```

After that, it's just a matter of writing the routines described in the exercise and calling them to get the desired statistics.

A routine for counting the leaves in the tree is similar to the countNodes() routine from Subsection 9.4.2. That routine, however, counts every node in the tree and now we only want to count the leaves. A leaf is defined to be a node in which both the left and right pointers are null. In the recursion, one of the base cases is when we come to a tree that consists of nothing but a le[illegible]. In that case, the number of leaves is 1. If the node is not a leaf, then we have to count the number of leaves in each of its subtrees and add [illegible]esults:

```
/**
* Return the number of leaves in the tree to which node points.
*/
static int countLeaves(TreeNode node) {
if (node == null)
return 0; // An empty tree has no leaves.
else if (node.left == null && node.right == null)
return 1; // Node is a leaf.
else
return countLeaves(node.left) + countLeaves(node.right);
} // end countNodes()
```

In general structure, the other two routines are similar. That is, there are two base cases: an empty tree and a tree consisting just of a leaf. In the remaining case -- a node that has one or both subtrees non-empty -- the routine has to be applied recursively to the subtrees of the node. Look, for example, at the routine for finding the sum of the depths of all the leaves in the tree:

```
/**
 * When called as sumOfLeafDepths(root,0), this will compute the
```

```
 * sum of the depths of all the leaves in the tree to which root
 * points.  When called recursively, the depth parameter gives
 * the depth of the node and the routine returns the sum of the
 * depths of the leaves in the subtree to which node points.
 * In each recursive call to this routine, depth goes up by one.
 */
static int sumOfLeafDepths( TreeNode node, int depth ) {
   if ( node == null ) {
      // Since the tree is empty and there are no leaves,
      // the sum is zero.
     return 0;
   }
   else if ( node.left == null && node.right == null) {
      // The node is a leaf, and there are no subtrees of node, so
      // the sum of the leaf depths is just the depth of this node.
     return depth;
   }
   else {
      // The node is not a leaf.  Return the sum of the
      // the depths of the leaves in the subtrees.
     return sumOfLeafDepths(node.left, depth + 1)
            + sumOfLeafDepths(node.right, depth + 1);
   }
} // end sumOfLeafDepth()
```

The most interesting aspect of this routine is the way it uses its depth parameter, which is used to keep track of the depth of the node in the complete tree (not just the subtree to which node points). For the root, the depth is 0. Each time the subroutine is called recursively, the node is one level deeper in the tree, and the depth parameter is correspondingly increased by 1. When we get down to a leaf node, where node.left and node.right are null, the value of depth is the depth of that node in the original tree, and the sum of the depths of the leaves in the subtree, which consists of just this one leaf node, is depth. When node is not a leaf, the sums for the two subtrees of node are computed recursively and are added together to give the sum for all the leaves in the whole subtree to which node refers. (If you have trouble believing that this works, remember that recursion works if it works for the base cases and if it correctly breaks down big problems into smaller problems. You don't have to follow the details.)

The routine for computing the maximum depth is similar.

When you run the program several times, you find that the average depth of the leaves in the tree tended to be about 12 -- higher than expected but still only 1/3 more than the average depth in a perfectly balanced tree. The height of the tree tended to be about 20.

The Solution

```
/**
 * This program makes a random binary sort tree containing 1023 random
 * real numbers.  It then computes the height of the tree and the
 * average depth of the leaves of the tree.  Hopefully, the average
 * depth will tend to be close to 9, which is what it would be
 * if the tree were perfectly balanced.  The height of the tree,
 * which is the same as the maximum depth of any leaf can be
 * significantly larger.
 */
public class RandomSortTree {


   static TreeNode root;   // Pointer to the binary sort tree.


   /**
    * An object of type TreeNode represents one node in a binary tree of real numbers.
    */
   static class TreeNode {
      double item;      // The data in this node.
      TreeNode left;    // Pointer to left subtree.
      TreeNode right;   // Pointer to right subtree.
      TreeNode(double x) {
            // Constructor.  Make a node containing x.
```

```
        item = x;
      }
    } // end class TreeNode


    /**
     * Add x to the binary sort tree to which the global variable "root" refers.
     */
    static void treeInsert(double x) {
      if ( root == null ) {
            // The tree is empty.  Set root to point to a new node
            // containing the new item.
        root = new TreeNode( x );
        return;
      }
      TreeNode runner; // Runs down the tree to find a place for newItem.
      runner = root;   // Start at the root.
      while (true) {
        if ( x < runner.item ) {
              // Since the new item is less than the item in runner,
              // it belongs in the left subtree of runner.  If there
              // is an open space at runner.left, add a node there.
              // Otherwise, advance runner down one level to the left.
          if ( runner.left == null ) {
            runner.left = new TreeNode( x );
            return;  // New item has been added to the tree.
          }
          else
            runner = runner.left;
        }
        else {
              // Since the new item is greater than or equal to the
              // item in runner, it belongs in the right subtree of
              // runner.  If there is an open space at runner.right,
              // add a new node there.  Otherwise, advance runner
              // down one level to the right.
          if ( runner.right == null ) {
            runner.right = new TreeNode( x );
            return;  // New item has been added to the tree.
          }
          else
            runner = runner.right;
         }
      } // end while
    }  // end treeInsert()


    /**
     * Return the number of leaves in the tree to which node points.
     */
    static int countLeaves(TreeNode node) {
      if (node == null)
        return 0;
      else if (node.left == null && node.right == null)
        return 1;  // Node is a leaf.
      else
        return countLeaves(node.left) + countLeaves(node.right);
    } // end countNodes()


    /**
     * When called as sumOfLeafDepths(root,0), this will compute the
     * sum of the depths of all the leaves in the tree to which root
     * points.  When called recursively, the depth parameter gives
     * the depth of the node, and the routine returns the sum of the
     * depths of the leaves in the subtree to which node points.
```

```
     * In each recursive call to this routine, depth goes up by one.
     */
    static int sumOfLeafDepths( TreeNode node, int depth ) {
       if ( node == null ) {
            // Since the tree is empty and there are no leaves,
            // the sum is zero.
          return 0;
       }
       else if ( node.left == null && node.right == null) {
            // The node is a leaf, and there are no subtrees of node, so
            // the sum of the leaf depth is just the depths of this node.
          return depth;
       }
       else {
            // The node is not a leaf.  Return the sum of the
            // the depths of the leaves in the subtrees.
          return sumOfLeafDepths(node.left, depth + 1)
                    + sumOfLeafDepths(node.right, depth + 1);
       }
    } // end sumOfLeafDepths()


    /**
     * When called as maximumLeafDepth(root,0), this will compute the
     * max of the depths of all the leaves in the tree to which root
     * points.  When called recursively, the depth parameter gives
     * the depth of the node, and the routine returns the max of the
     * depths of the leaves in the subtree to which node points.
     * In each recursive call to this routine, depth goes up by one.
     */
    static int maximumLeafDepth( TreeNode node, int depth ) {
       if ( node == null ) {
           // The tree is empty.  Return 0.
          return 0;
       }
       else if ( node.left == null && node.right == null) {
            // The node is a leaf, so the maximum depth in this
            // subtree is the depth of this node (the only leaf
            // that it contains).
          return depth;
       }
       else {
            // Get the maximum depths for the two subtrees of this
            // node.  Return the larger of the two values, which
            // represents the maximum in the tree overall.
          int leftMax = maximumLeafDepth(node.left, depth + 1);
          int rightMax =  maximumLeafDepth(node.right, depth + 1);
          if (leftMax > rightMax)
            return leftMax;
          else
            return rightMax;
       }
    } // end maximumLeafDepth()


    /**
     * The main routine makes the random tree and prints the statistics.
     */
    public static void main(String[] args) {

      root = null;  // Start with an empty tree.  Root is a global
                    // variable, defined at the top of the class.

      // Insert 1023 random items.
```

```
        for (int i = 0; i < 1023; i++)
            treeInsert(Math.random());

        // Get the statistics.

        int leafCount = countLeaves(root);
        int depthSum = sumOfLeafDepths(root,0);
        int depthMax = maximumLeafDepth(root,0);
        double averageDepth = ((double)depthSum) / leafCount;

        // Display the results.

        System.out.println("Number of leaves:         " + leafCount);
        System.out.println("Average depth of leaves:  " + averageDepth);
        System.out.println("Maximum depth of leaves:  " + depthMax);

    }  // end main()


} // end class RandomSortTree
```

Last modified: Wednesday, 13 May 2020, 1:16 PM