CS 158 Lab 20 April 15, 2010
Timing Sorting

1. Make sure your heap sort is working from the previous lab. Add a main method to the **Heap** class, and run an experiment to calculate the time it to sort different sized arrays. Use the **System.currentTimeMillis** method before and after the sorting takes place. Be sure to start the timer AFTER the array has been filled with random data. Time only the time it takes to build the heap and sort from the heap back into the array. Start with an array of size 10000, record the number of milliseconds it takes, repeat 10 times and compute the average (you will find that your results vary, sometimes significantly, due to other activity on the processor). Then repeatedly double the size of the array and repeat the experiment, recording your results for a report until it takes too long to sort or you get an "out of memory" type of error from the system.

2. Create a new package called **bst**. In this package add a **Node** class that contains, as protected member variables, an integer variable called **data** and two **Node**s called **left** and **right**. Add a constructor that takes an integer argument and sets **data** to that value and sets **left** and **right** to **null**. Add a class called **BST** that contains protected member variables: **root** that is a **Node**, **array** that is an array of integers, and **nextAvailable** that is an integer. Add a constructor that takes an integer argument, instantiates **array** to have that as its size, and sets **root** to **null** and **nextAvailable** to zero. Add a **randomInit** method like the one you used in the **Heap** class, and make sure it works. Add a new method called **buildBSTFromArray** that begins by instantiating **root** as a new **Node** that contains as data the first element of **array**. Then the method goes through the rest of the array calling an **insert** method (yet to be written) that takes, as arguments, the current element of the array (not the position, but the element itself) and the **root** variable. This method will build the binary search tree recursively in the next step.

3. Create the method **insert** that takes an integer argument and a **Node** argument. The idea is to compare the input integer with the data in the input node. That determines whether you go left or go right. Then, if in that direction, there is a null subtree, you can hang a new node with the input data in it on that side. Otherwise, you need to make a recursive call with the same data, but passing the subtree as the node. Add a JUnit test for this method. For the test, create a **BST** object with 7 elements, set them to 5, 4, 3, 2, 1, 0, 6, call the **buildBSTFromArray** method, and then assert that the **root** node has 5 in it as data, that the root's **left** node has 4 in it, and the root's **right** node has 6 in it. You can add other tests if you wish.

4. Add a recursive method called **inorderTraverse** that takes a **Node** argument and does an inorder traversal of the binary search tree, placing the visited node's data in the next available element of **array**. Recall that inorder traversal means traverse left if you can, visit the node, traverse right if you can. After this method executes, the data in **array** will be sorted. So you can create a JUnit test for this method that creates a **BST** object with 1000 elements, puts random data into it, builds the binary search tree, calls the **inorderTraverse** method starting at the root, and then asserts that all the elements of the array are in order.

5. Add a main method to the BST class and repeat step 1 with this algorithm.