CS112, Data Structures, Summer, 2015

Assignment 3: SizeBSTs

Read the Course Policies page, **Programming Assignments** section for important information about lateness, program not compiling, etc.

You will do this program individually. Read DCS Academic Integrity Policy for Programming Assignments - you are responsible for abiding by the policy.

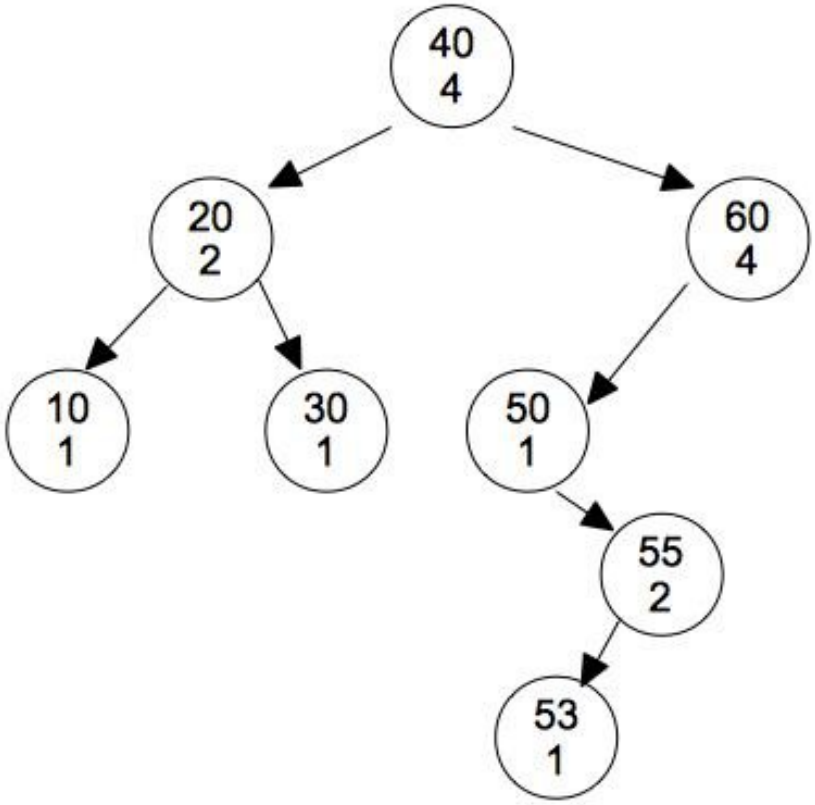
File to download (attached to this assignment): assig3.zip

Do not unzip this file. Import it as a project into Eclipse by following the directions in Sakai > Eclipse > Importing a zipped project into Eclipse.

The Task:

The program you will write will implement what I will call a “Size Binary Search Tree” (SBST). An SBST is like a regular Binary Search Tree, but it is also capable of finding out, given a number j, how many nodes in the tree hold numbers less than or equal to j, and it is capable of doing so in time proportional to the height of the tree. It does so by using an additional field in each node, called the “size” of the node, which keeps a count of the number of nodes in the left subtree of the current node, plus one for the current node.

For example, we might have the following tree (diagram below)



The upper number in a node is the data at the node and the lower number is the “size”. For instance, the node with number 40 has size 4 since there are 4 nodes counting the left subtree (10, 20, and 30) plus the node with the 40 itself. The same is true of the node with the number 60. Note that nodes with empty left subtrees (10, 30, 50, and 53) all have size 1.

The sizes are updated while inserting data into the tree (and while deleting, but deleting is not part of this assignment). The sizes are used while computing the number of nodes less than or equal to a given number.

Inserting

To insert a number into the tree we first have to search to see if it is already there in the tree. There will be a general search method getNode, which either finds the node containing a given number if it is there or else finds the node that would point to the node containing the number if it were in the tree. For instance, getNode called on the root of the tree above with a target of 10 would return the leftmost leaf. If called on the root with target 9 or 11 it would also return the same node.

So, the first thing insert does in call getNode. If the resulting node contains the number we want to enter, that number is already in the tree and there is no need to insert it again, so we stop without changing the tree. If the result of getNode does not contain the new number, we add a new node with the new number either as the left child or the right child of the node getNode returned. E.g., if we try to insert 55 in the tree above, getNode returns the node whose data is 55 and we are done. If we try to insert 58, getNode returns the same node, and we add a right child with data 58 and size 1 (since the subtrees of a node are always null when we add the node).

We have to update the sizes of all the ancestors of the new node. Actually you may want to do this **before** actually adding the new node. To update the sizes we go through the tree as if we were searching for the new node, and each time we are about to go to a node's left child we add 1 to the size of the node we are about to leave. For instance in the tree above if we added a node for 52 we would not change the size of node 40 (we

leave it by going right), we would add 1 to the size of node 60 (we leave it to the left), we would not change the size of node 50, but we would add one to the size of nodes 55 and 53.

Computing number less than or equal to

To compute the number of entries less than or equal to a number, say j, we again go through the tree as if searching for j. This time we sum the sizes of the nodes that we leave by going right. Suppose we were looking for number of entries less than or equal to 55 in the tree above (before doing any insertions), we would add 4 (since we leave node 40 to the right), 1 for node 50, and finally 2 because that is the size of node 55 itself. (We include the node that has the data j, even though we do not leave it in either direction.) This gives 7 and, indeed, there are 7 numbers less than or equal to 55 in the tree above..

nodeString

You also need to fill in the nodeString method of SizeBSTN.java. The format is as follows: [LSubTree data, size Rsubtree]. E.g., a tree that contained only the top 3 nodes of the tree drawn above would result in the string "[[null 20,1 null] 40, 2 [null 60, 1 null]]". (Note that sizes have been updated for the smaller tree.)

Notes

- Methods that you fill in in SizeBSTN are static to make recursive implementations easier.
- You are not required to use recursion but some methods will be hard to write non-recursively.

Rules

- You may not add any classes.
- You may not add any import statements.
- You may not change any code in the existing classes except that:
  - you **must** replace each line that says "// fill in here" or "// replace this line" with one or more lines of code. Both classes (SizeBST and SizeBSTN) have lines like these.
  - you **may** add methods as long as they are **private**.

Handing your work in

Note that the due date is 11:00 **in the morning** of Wednesday, July 8.

A project that is late will receive zero points. Note that Sakai's clock may not agree with your clock, but when Sakai thinks time is up, it is up. Also note that it may take a few minutes to interact with Sakai – if time is up while you are waiting for Sakai to respond, your project is late. So don't hand in at the last minute. Also note that you can hand in as many times as you want. We will grade the latest on-time version you submit. So hand in early and often. And please be sure to click the Submit button after you upload your files.

Attach your files SizeBST.java and SizeBSTN.java as **two separate attachments** to this assignment.

Grading

- A project that does not compile will receive zero points.
- **Each** method must work as specified above and in its comments. We will test each method that you are to fill in.