CS 201 Discussion 9

BSTS + RECURSION
A binary search tree is:

- A tree (a group of nodes, where one is the root and every other node is some other node’s child)
- Binary (each node has a left and right child. If no child exists, the child is actually null)
- Has the binary search property (if I look at the tree rooted at some node’s left child, all the values in the tree are less than that node’s value)
Tree terms

Root – the node which doesn’t have a parent (here the root is 2)

Leaf – a node which has no children (e.g. 0, 3, and 6 in this example).

Depth – the distance from a node to the root (e.g. 0 has a depth of 2, 5 has a depth of 1)

Height – the length of the longest path from the root to a leaf (e.g. here the tree has height 3)
Adding to a BST

To add to a BST:

Start at the root

Compare the value you’re adding to the root. If it’s less, move left. If it’s greater, move right.

Repeat until there is no child in the direction you’re moving. Then, add a new node there.

An example of adding 4 is shown at right.

The key to this process is that the binary search property is maintained.
Adding practice

Draw the BST produced if:

We start with the value 5 as the root.

We add the values 7, 2, 9, 6 in that order.

(Keep this drawing around – it will be useful later in discussion)

As an aside, when we actually write code to add a node to a BST, it’s very similar to adding a node to a linked list, except rather than just move on to the next node, we have to decide to move left or right.
Recursive data structures

Recall that any linked list (except the empty one) can be thought of as a head node pointing to another linked list. Thus, we have defined a linked list recursively.

Since trees are just generalizations of linked lists, we can apply a similar recursive definition to them.
BSTs and Recursion

If we think about it, a BST is really just a root node with two more BSTs below it – we’ll usually call these the ‘left subtree’ and ‘right subtree.’

This means BSTs lend themselves really nicely to recursion.
BSTs and Recursion

Furthermore, the recursion we do with BSTs for the most part doesn’t at all depend on what the subtrees look like! So, when writing recursive algorithms for BSTs, you’ll want to have a generic picture like the one below in mind.

Note that for BSTs, we always know the left subtree has values smaller than the root, and the right subtrees larger values.
To the right is an outline you can use for almost any BST algorithm. For some methods, you won’t need to check both subtrees, only one.

Note that this is just a less general version of the recursion pseudocode from Discussion 7 – it has the same structure, however.

Also note the similarity to the Linked List recursion outline.

```
BSTmethod(node):
    if at the base case:
        return base case value
    L = BSTmethod(node’s left child)
    R = BSTmethod(node’s right child)
    return some function of L and R
```
BST Recursion Outline

The method outline on the previous slide usually isn’t enough – it only answers the question for the subtree rooted at the input node.

If we want to write a method which solves the problem for the whole tree (i.e. without taking in any arguments), we’ll have to write another method which calls our recursive method, usually on the root.

```python
BSTmethod():
    return BSTmethod(root)

BSTmethod(node):
    if at the base case:
        return base case value
    L = BSTmethod(node’s left child)
    R = BSTmethod(node’s right child)
    return some function of L and R
```
Pseudocode example: Counting nodes

To count the nodes in a tree, we can take advantage of this fact:

The number of nodes in a tree is

1 (the head) +
the nodes in the left subtree +
the nodes in the right subtree.

We write our recursive helper method to answer the question ‘how many nodes are in a tree rooted at N?’.

Then, to answer the question ‘how many nodes are in the whole tree?’ we just call this method on the root.

countNodes():
    return countNodes(root)

countNodes(N):
    if N is empty/null:
        return 0
    L = countNodes(N’s left child)
    R = countNodes(N’s right child)
    return 1+L+R
Practice: BinarySearchTree.java

Download the code for today’s discussion from the snarf folder (Discussion9)

You’ll see a class named BinarySearchTree.java – it has several unfinished methods, which use the structure from the previous slide. For the rest of discussion, work on completing these methods.

Use the provided JUnit class to test your code. Every test depends on the add method working, so make sure you pass testAdd before looking at other tests.

Every method can be completed using the outline we showed you, just remember – you don’t always need to check both subtrees. It may be helpful to work through some examples on paper before – all the JUnit tests use the same tree you drew earlier.