CSc 127B — Introduction to Computer Science II
Fall 2015 (McCann)

http://www.cs.arizona.edu/classes/cs127b/fall15/

Section 13: BSTs and Iterators

Pair up with anyone who is agreeable to pairing up with you, pick the first driver, and let's get to work!

PART I: Reconstructing a Binary Tree from its Traversals

At the start of section, your SL showed you how to rebuild a binary tree from its preorder and inorder traversals. In this part you'll do the same, but with inorder and postorder traversals, and you'll briefly explore whether or not the same can be done with preorder and postorder traversals.

1. Here are the inorder and postorder traversals of a binary search tree of integers. Using a similar process as your SL just described for preorder and inorder, draw the tree that produces this pair of traversals.

   Inorder: 12 15 16 22 23 30 32 36 41 46 Postorder: 15 12 16 23 36 32 30 22 46 41

2. Take a few minutes to think about the algorithm you followed to reconstruct the tree. Is it recursive?
3. Reconstructing the tree is a naturally recursive algorithm, although you may not have thought of it as such while you were doing it. What are the general case(s) and base case(s) of this algorithm?
4. Now consider the problem of reconstructing a tree from its preorder and postorder traversals. Does your recursive tree reconstruction algorithm work given these traversals? Be ready to explain your answer.
5. It's impossible to reconstruct every binary tree from just its preorder and postorder traversals. Prove this to yourselves: Pick three values. Create two different three-node binary trees, each containing those three values but in different arrangements, such that both trees have the same preorder and the same postorder traversals. What do your trees look like?

✔ **CHECKPOINT 1** Raise your hand. Your SL will come over and inquire about your answers.

PART II: Counting the Leaf Nodes of a Binary Search Tree

We've seen the idea of providing two methods with the same name, one public and one private, so that users can ask for a service (such as inserting into a BST) without needing to know about implementation details (such as tree nodes). In this part, we'll use the same idea to create a method that counts the number of leaf nodes in a BST.

1. Load the program `Section13.java` into DrJava. This is (mostly) the `T15n01.java` example program; we're going to add to it.
2. Because the `Section13.java` file has some inner classes, it's easy to accidently add a new BST instance method to the wrong class. Search the `Section13.java` file for the identifier `countLeaves`. You should find a stub of a public method with that name. But don't write any code yet!

(Continued ...)

3. In a few minutes, you will be writing a recursive private method named `countLeaves()` that accepts a reference to a `BinaryNode` object and returns the count of the number of leaf nodes in the tree.

   Before you do that, take a few minutes to answer our usual sequence of questions for recursive methods:

   - "What's (slightly) simpler than ... ?"
   - Can your answer to (a) be used to solve the original problem? If so, how? (This/These are your general cases.)
   - What is/are the simple situation(s) that can be answered without doing much (if any) work? (This/These are your base cases.)

   Happy with your answers? We hope so, because you'll need to use those answers to help you write the private recursive version of `countLeaves()`. Write it!

4. Complete the stub of the existing public `countLeaves()` method. All it should do is call the private `countLeaves()` method on the root of the tree, and return that method's answer as its own.

5. `Section13.java` has a `main()` method that creates a few binary search tree objects and calls their `countLeaves()` methods. The expected output is in the comment at the top of `Section13.java`. When your output matches that, you're ready for the checkpoint.

   ✔ **CHECKPOINT 2** Raise your hand. Your SL will come over and check that your `countLeaves()` methods are working.

PART III: Adding an Iterator to a Binary Search Tree Class

Your SL introduced the idea of an iterator to you at the start of section today. Our binary search tree class (which we added to in Part II) has the `inOrder()` method for printing the tree's content. But what if the user of our BST class wants to do something other than print the tree's data? Rather than try to guess everything a user would want to do and provide methods for all of it, we can provide an iterator so that a user can get the data in sequence and do with it whatever s/he wishes. In this part, you'll add an iterator to that class.

1. In `Section13.java`'s `main()` method, below the testing code for Part II, is testing code for this part. Take a look at the `for` loops in that testing code. That form of a `for` loop is known as a "for each" loop, because the loop variable (`i` in this program) represents each of the elements in the collection of data items in turn. Because our `BinarySearchTree` class will have an iterator by the time you're done with this part, the "for each" loops will automatically recognize and use it to allow us to iterate over the tree's content.

   All we have to do is tell Java how to iterate over a tree, within the framework of an iterator. That will require some work; as we know, tree traversals – which put a tree's data in a nice linear sequence, perfect for an iterator – are naturally recursive, while iterators, as the name suggests, are naturally iterative. Happily, we'll tell you what to do; you just have to follow along.

2. Search `Section13.java` for the phrase "Code for Part III", which should be just below your `countLeaves()` code from Part II. Below that, you'll see the stub of a method named `iterator()` that returns a reference to an object that implements the `Iterator` interface. Every class that offers an iterator needs a method to return a reference to an `Iterator` object; usually, the method is named `iterator()`, as ours is. We also have a private inner class named `InorderIterator` that creates our BST iterator objects. All that the `iterator()` method needs to do is make a new `InorderIterator` object. and return a reference to it. Complete `iterator()` to do that.

(Continued . . .)

3. The `InorderIterator` class is just below the `iterator()` method. We've provided stubs for the constructor and the three methods required by the `Iterator` interface: `hasNext()`, `next()`, and `remove()`.

   Let's start with the constructor. The two instance variables are already declared at the top of the class; the constructor needs to initialize them. `currentNode` needs to reference the tree's root node, and `nodeStack` needs to reference a new `Stack` object that is a stack of references to `BinaryNode<E>` objects. Complete the constructor.

   (Yes, we're using Java's `Stack` class, the one that Java suggests we don't use. We figured you'd like to get to use it at least once. Outside of this section activity, you really should use a `Deque`-implementing class instead.)

4. At the bottom of the `InorderIterator` class is the stub of the `remove()` method. It's an optional method, meaning that it doesn't have to do anything – but we still need to have the method! And we don't just want it to return when called; that might leave a programmer thinking that it did something useful. Instead, we'll have our `remove()` method throw an `UnsupportedOperationException`, which is appropriate for a method that must exist but doesn't work. Complete the `remove()` method to throw that exception.

5. Just below the constructor is the stub of the `hasNext()` method. The method needs to return true in two situations: If the stack is not empty, or the variable `currentNode` is not null. Otherwise, it returns false. Complete `hasNext()`.

6. The last method is `next()`, which has to find the next tree value in the inorder traversal sequence. We *could* remember the last sequence value returned, recursively generate the inorder sequence, find that last value in the sequence, and return the one after it. That's very crude; we'll use a more sophisticated algorithm that makes use of a stack.

   Implement the `next()` method by turning this pseudocode into legal Java:

```
While the currentNode is not null:
    Push its value onto the stack
    Advance currentNode to the left child

If the stack is not empty,
    Pop the top node reference from the stack
    Store that node reference into the variable nextNode
    Store that same node's right child reference into currentNode
Otherwise,
    Throw a NoSuchElementException

Return a reference to the data held by nextNode's node
```

   (Optional!) If you're curious as to how this algorithm can always find the next value in the inorder traversal sequence, read the rest of this step. Otherwise, feel free to move onto the next numbered step; you won't need to know this explanation.

   Still here? Good! Whenever we do an inorder traversal, the first value in the sequence is on the far left of the tree. Notice how the loop in the pseudocode dives left? That's why. Along the way, we stack the nodes we slide past on the way down; we will need to revisit them, and we don't have recursion to remember them for us, so we'll use the stack.

   That loop runs off the left end of the tree. That's why we immediately pop the stack, to back up to the last node we saw before running off the tree. You can think of the stack as sort of 'The Story of The Aft Reference' as the Little Brother method works through the tree. We aren't actually using the Little Brother method, but it's similar.

(Continued . . .)

The node we back up to holds a reference to the next value in the inorder sequence, so we keep track of the node (by using nextNode to reference it). Before we can return its data, we have to get currentNode set up for the next call to this method. Remember that in doing an inorder traversal, we 'visit' then go right? That's why currentNode is updated to be whatever is to the right of nextNode.

You may be wondering what happens if nextNode's node doesn't have a right subtree. In that case, currentNode will be null. During the next call to `next()`, the loop will be skipped (there's no point in diving left in a subtree that doesn't exist), we'll pop the stack to go back up the tree, and continue on. When currentNode is null and the stack is empty, the traversal is finished. This situation is the logical negation of the condition you put in `hasNext()` a few minutes ago.

That's how the algorithm works. To really understand this algorithm, you need to trace through it on a few trees. Finish this section first; if you have time afterward, you can do that tracing.

7. If you've coded the constructor and the three iterator methods correctly, `Section13.java` should compile and, when run, produce the output show in the program's comments. When you're seeing that output, you're done.

✔ **CHECKPOINT 3** Raise your hand. Your SL will come over and verify that your iterator is correctly iterating.

PART IV: Clean Up!

1. Log out of your computer; pick up your papers, writing implements, cell phones, trash, etc.; push in your chair(s).

✔ **CHECKPOINT 4** Raise your hand. Your SL will come over and look for evidence of moist gum under your desktop.

You're free to go! But, if you have time, we recommend that you use it to work some more on one or both of the current programming assignments (individually, of course), or to walk through that iteration algorithm from Part III.