Homework # 9
due Monday, November 15, 10:00 PM

In the last assignment, we implemented the Realty ADT using a binary search tree (BST). In this assignment, we augment this ADT with some new methods and an iterator so that it fulfills the `Set` interface. You will start with our solution to Homework #8.

`https://classroom.github.com/a/ECi57HnV`

1 Concerning "parent" links

Please see the "Navigating Trees" handout in this module for information on how to use parent links. You will need to add a `parent` field to the `Node` class.

Then you need to check that they are set correctly. The easiest way to do so is to add a parameter to the `checkInRange` helper method that gives the *expected* parent field which then should be checked. You can use the `TestInternals` test suite (`testA` through `testN`) to see if you are checking appropriately.

Then you will need to change "add" (or the `placeUnder` helper method) to update the parent link. You can use "TestRealty" to see if you have restored functionality after adding parent links.

Don't continue the homework until these parts are all working. Ask questions on Piazza if stuck! Make sure to give a specific test you are working on, and don't omit important clues and details.

2 Concerning New Private Helper Methods

To assist with iterators (next section), you will implement two private helper methods:

isInTree(Node) This method is used to check if the parameter node is actually in the tree. It also succeeds if the node is null. Your code shouldn't rely on the node having reasonable links: it could be in a little cyclic "tree" all by itself. The code *can* rely on this tree being well formed, so it knows that if the node is in the tree, it will be in a paricular spot.

getNextNode This method finds the next node in the tree in order. It should use parent pointers to be efficient. It should not start from the root as was done in `getNext`. Instead it should start at the node indicated. The algorithm you need to use is described in the "Navigating Trees" handout and in the lab.

We have internal tests (`testO` through `testS`) that you can use to check your implementations. Make sure these are working before you proceed.

3 Concerning `AbstractSet`

In this assignment, you will be implementing the standard "Set" interface, using the helpful `AbstractSet` class. Sets are collections, and `AbstractSet` functions much like `AbstractCollection`. Thus, the most interesting method that needs to be implemented is the iterator.

You should implement the `iterator` method by creating a new instance of the `MyIterator` nested class. You should also make sure that all public methods that override something from `AbstractSet` are correctly marked override—we will be checking this while grading.

In the course of the assignment, you will find it necessary to override methods from `AbstractSet`, because an override is required by Java, or an override is needed to get the correct result, or an override is needed for efficiency. Make sure to indicate the reason ("required," "implementation" or "efficiency") in a comment on the `@Override` tag. Do *not* override something if an override is not needed.

4 Concerning the Iterator over the Realty

We will use the parent pointers and the new helper methods to implement an iterator. The iterator will keep track of the current node (so we know what to remove) and the next node (so we know what, if anything comes next). For fail-fast semantics, we will also keep a version in the main class and one with the iterator. Recall that the version must be updated whenever the number of elements changes, and that the iterator methods need to check the version to see if they are stale.

The iterator will have an invariant too, as you have seen in some earlier assignments. As before, it will check the outer invariant, and then if the versions don't match, it doesn't check its own fields because staleness might make them invalid, instead it says there is no problem. (Or rather, if there is any problem, it's not with the iterator implementation, but rather with the way the client *used* the iterator!) Both the current and next nodes should be null or in the tree. The current node can be null at any time because it might have been removed (or maybe we haven't started iteration yet). If it's not null, then the next node should be its "next node" (use `getNextNode`).

When an iterator is constructed, you will need to set up the next node so that it will be ready to go. Remember where to find the first in-order node in a binary search tree. Hint: it's rarely the root.

4.1 The `remove` method

In order to implement the iterator `remove()` method, we will keep track of the current node in the iterator and then remove it from the tree. As usual, you should make sure you implement the hard removal process in only one place.

The activity this week explains how there are easier and harder cases for removal. In particular if the node has both left and right children, we have to replace it with either the greatest predecessor or the least successor. Since our `remove` method is within an iterator, we have to use the *greatest predecessor* so as not to disturb the running iterator's "next node." Before we perform this replacing, the greatest predecessor has to be removed from the tree first, but this is easy since it always classifies in the easier cases to remove (why?). Do not forget to update the size field and the versions so we have a fail-fast iterator.

5 What you need to do

You need to

1. Have `Realty` extend an appropriate set class. Add "override" declarations. Add the new required method. Add a version field.
2. Add parent field. Update the `checkInRange` method. Pass internal tests, `testA` through `testN`.
3. Re-implement the `add` method and its helper methods so that you can pass all the existing tests (`TestRealty`).

4. Implement the two new private methods and pass internal tests, `testO` through `testS`.
5. Implement the iterator `wellFormed` method and pass remaining internal tests (`testT` through `testY`).
6. Implement the iterator, including "fail-fast" semantics, except for removal, and pass `TestRealtyNew` up through `test35`.
7. Implement iterator removal, and pass all remaining tests in `TestRealtyNew`.
8. See if you can pass efficiency tests. If inefficient, then fix your algorithm, or override as necessary. Then make sure you didn't break anything: run all previous tests.
9. Check for bugs with random testing. If a test is printed, use it to try to diagnose and fix a software error. When fixed, re-run random testing to get a new test case. Once random testing succeeds, re-run efficiency tests, to make sure efficiency didn't break.

We've set things up so that if you do things in this order, you are more likely to stay on track.

6 Files

The repository for this assignment has both old and new tests, and efficiency tests for the updated Realty, and includes the solution from Homework #8 in the `Realty` class. Random testing is available in the JAR. We have more efficiency tests than usual; each test should up to a second; don't worry if all of them combined take a few seconds. Don't be surprised that you can't run `TestCollection` because that's an abstract testing class; use `TestRealtyNew` instead.