In the last assignment, we implemented the Account ADT using a binary search tree (BST). In this assignment, we augment this ADT so that it implements the Set interface. In particular, this requires that we implement iterators, including the ability to remove transactions. We will start with the same data structure as in Homework # 8, but will add “parent” links as explained in the “tree navigation” handout.

1 Concerning the parent field in each Node

In order to enable traversal in any direction in the binary search tree, we add a “parent” link in every node. The root’s parent is null, every other node has a parent link pointing to the node for which the left or right links points to this node.

1.1 The invariant

The invariant currently checks that the tree is well formed using a recursive private helper. It is an easy thing to add a parameter specifying the parent of the node being checked. We have done this for you. Also easy is to use this value to check the parent link itself, which we have left for you. You can use the TextInternals to check (but one of the tests also checks the iterator invariant).

1.2 Adding new nodes

New nodes added to the tree need to have the parent node set correctly. This is easy to do in a number of different ways. Look at the code (from the solution to Homework # 8), think, and then choose the way that is clearest to you.

1.3 Removing nodes

Removing a node requires changes to the parent link of other nodes when they are moved to new places in the tree. When you implement removal, you will need to handle this situation when removing a value in a node with children in the tree.

2 Concerning the Iterator over the Account

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.

Additionally, in order to enable efficient implementation of the Set remove method, we will add a new iterator method that starts the iteration in the “middle” of the tree. The iteration will start with a transaction with the given date, or the transaction soonest next.

2.1 Data Structure

The iterator will keep the “next node” in addition to fields we have seen before (a local copy of the version and a boolean for whether “next()” has been called).
The only substantive aspect of the invariant is that the “next node” must be in the tree (not in some parallel universe). But of course, you need to check the outer invariant and also trivially accept the data structure if the versions don’t match.

2.2 Moving the iterator

We will use parent pointers to handle moving back “up” the tree. For more details, please read the handout on tree navigation on the “Examples” page.

2.3 The remove method

In order to implement the iterator remove() method, you will need to use the “next node” field to locate the node to remove. If the “next node” is null, then the node to remove is the “rightmost” node. This node will always have a null “right” link which makes it easier to remove (why?).

If the “next node” has a non-null “left” link, the node to remove again is the rightmost within the left subtree, again with a null “right” link.

Otherwise, as described in the textbook, the node whose value needs to be removed has at least a non-null “right” link because the “next node” is in that subtree. Indeed the “next node” will be the node to remove in place of that node (and always has a non-null left link). In this case, the value in the “next node” will be moved back to that ancestor.

As mentioned above, when a node is removed, you will need to adjust the parent node of a node being given a new location in the tree.

As well as the technical heart of the operation, you will also need to implement the by-now expected fail-fast semantics as well as the check that “next()” was called.

3 Concerning Efficient Implementations for the Set ADT

As a collection, The Set ADT provides two methods, contains(Object) and remove(Object). The inherited implementations from AbstractCollection work just fine, but are inefficient compared to binary tree access. Thus, once you have passed all the non-efficiency tests, you should override these methods to provide efficient implementations.

These methods both start in the same way: if the argument is not a transaction, they can safely return “false.” Otherwise, they can use the iterator-in-the-middle ability to start an iterator and see if the next item is indeed the transaction to check or remove. If so, we handle it (return true, or remove and return true).

4 What you need to do

You need to

- Update _checkInRange, add (or placeUnder)
- Implement the MyIterator class, first the hasNext and next methods, and once they are working, the remove method. Finally add “fail fast” semantics.
- Implement getEarliest and getNext using iterators.
- Implement contains and remove using iterators.