Homework # 8
due Wednesday, November 1, 10:00 PM

In this homework assignment, you will implement a Realty ADT using a binary search tree (BST) data structure. A BST permits a more efficient lookup mechanism (comparing to a linked list) and so there is an efficiency test to make sure that your code can handle half a million entries.

`https://classroom.github.com/a/i9lDNJUs`

1 The Binary Search Tree Data Structure

Please read section 9.5 in the textbook for a description of the binary search tree data structure. Alternatively, there are tons of webpages/lecture notes on BST, the wiki page maybe a good start point (`https://en.wikipedia.org/wiki/Binary_search_tree`). In the textbook (as well as some online sources), a separate `BTNode` class is used; we will *not* do that. Use a nested node classes as before.

Linked lists and arrays support insertion, removal or finding an entry in time proportional to the number of entries in the container. Trees, on the other hand, offer a significant efficiency advantage in that they generally support these operations in time proportional to the *log* of the number of entries (assuming the tree is kept balanced).

In order to achieve the potential time efficiency of binary search trees, one needs to keep the tree roughly balanced. In CompSci 535, you will learn some of the techniques used to do this (as it happens, the tree-based containers in Java's libraries use "red-black" trees). But in this course, we will ignore the problems of unbalanced trees. Do not make any attempt to re-balance your tree. The efficiency tests we run will make sure to construct a balanced tree.

2 Concerning the Realty.Listing ADT

A realty listing has a price and an address. The price can't be negative or be "too high" (two billion dollars). The address can be any (non-null string). Two realty listings are the same only if they have the same price and exactly the same "address." We override the hash code computation as well; hash codes will be discussed later in the course. A realty listing can be converted into a line of text and back again. Realty listings are implemented with a public static class `Listing` nested inside of `Realty` (described below).

3 Concerning the Realty ADT

Realty listings are often sorted by price because many people are interested in buying a house in a given range (in a particular area of town). You don't want to go too high because you can't afford it, but neither do you want to buy something with too low a price because that indicates something is wrong.

The Realty ADT is a sorted collection of Realty.Listing objects. It supports the following public operations:

size() Return number of listings in the collection.

getMin() Return the cheapest listing in the collection, or null if none.

getNext(int) Return the listing with closest price greater than the argument, or null if no such listing in the collection.

add(Listing) Add a listing, returning whether it was possible (it is not allowed to have two listings with the same price).

addAll(Listing[],int lo,int hi) Recursively add all listings from the array within the index range [lo,hi).

toArray(Listing[]) Copy the listings (in order of the prices) into an array.

We omit "remove" functionality from the ADT for simplicity. The collection is not allowed to have two listings with the same price. (In practice, one would use the address as a secondary key, but that simply complicates the programming without you learning more about binary search trees.)

The `addAll` method takes extra parameters so that it can be its own recursive helper function. The reason why one wishes to use recursion is that if `addAll` were simply to add all the listings in the order they are listed and if the listings are already sorted (which frequently happens if one uses the interactive driver described next), one would have a highly unbalanced tree. Thus `addAll` is supposed to add elements starting from the *middle* and then to add two remaining halves recursively.

The efficiency tests check to see that you built the tree correctly. If you wrote the code efficiently, this test takes at most a second or two. But if you did the inefficient, but easy, technique, the test will take a minute or two.

4 Concerning the Realty Main Program

We provide a text-based interactive program for interacting with Realty objects. You may wish to use this program to run your own tests. It also shows the bare-bones functionality a program using Realty would have.

5 What you need to do

You need to implement the private recursive helper function that checks that the tree is well formed. You need to implement all the public methods for the `Realty` class using efficient binary search tree algorithms. You should implement a private helper method `copyInto` that helps `toArray`. You may define other private (recursive) helper methods, as desired. Our solution has two additional helper methods.

6 Files

Your git repository for this homework includes a skeleton of the `Realty` class, a `Main` class, and some test cases:

UnlockTest.java Unlock all the locked tests without running them.

TestRealty.java An JUnit test case for the ADT.

TestInternals.java A test of the helper methods and the invariant.

TestEfficiency.java A JUnit test case that should take at most a few seconds. If it takes much longer, you are doing something wrong.

The JAR file includes random testing, as with many of our assignments.