Homework # 8
due Monday, November 4, 10:00 PM

In this homework assignment, you will implement a Set ADT using a binary search tree (BST) data structure. A BST permits an efficient lookup mechanism (as opposed to linear search) so there are efficiency tests to ensure your code can handle half a million entries.

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 many web-pages/lecture notes on BST. The wiki page may be a good starting 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 class 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 RasterSet ADT

Similar to previous assignments, the `RasterSet` ADT stores a collection of `Raster` objects, disallowing duplicates. We are not implementing `Collection` and we will not support removal or iteration. This is in order to reduce the complexity of the implementation. Removal and iterators will be added in the following homework assignment.

In order to allow for efficient search we must be able to compare `Raster`s. We will use two comparators, one sorting primarily by x-dimension, and the other primarily by y-dimension. To make things easy for ourselves, we design these comparators so that any two rasters that are not equal will not compare equivalently. Since we don't allow duplicates, we will never have to deal with equivalent elements, and we will also assume that any two rasters that compare equivalently are equal.

To achieve this, our comparators must discriminate between rasters with even a single pixel of difference. So they must first compare the dimensions of the rasters, and then compare the pixels at each location in the raster. We have provided a comparator for `Pixel` objects that you can use. We have also implemented one of the two `Raster` comparators for you.

The efficiency tests check to see that you build and navigate the tree correctly. If you write the code efficiently, these tests shouldn't take more than 15 seconds or so. If you use inefficient (but easy) techniques, the tests will take much longer and you will lose points.

The following public operations are supported:

size() Return the number of rasters in the set.

isEmpty() Return whether the set is empty.

clear() Empty the set.

contains(r) Return whether r is in the set.

add(a) Add a to the set. Return false if it as already in the set.

xFirst() Change the comparator to the x-first comparator, resorting if necessary. From an ADT perspective, this doesn't really change the behavior of the set. This is because a set is an unordered collection. Its purpose is to improve efficiency for `xRange` calls.

yFirst() Change the comparator to the y-first comparator, much like `xFirst()`.

xRange(lo, hi) Return an array containing all the rasters in the set with an x-dimension greater than or equal to `lo` and less than or equal to `hi`. *Note*: When the x-first comparator is being used, this method should ignore some subtrees. But when the y-first comparator is being used, it must check them all!

yRange(lo, hi) As `xRange` but for the y-dimension.

3 Private Helper Methods

The implementation will make use of several helper methods, some of which must be recursive:

checkInRange(n, lo, hi) This is used by `wellFormed` to test the invariant. For the subtree rooted at n, make sure its raster is larger than its largest smaller ancestor, and smaller than its smallest larger ancestor (given by `lo` and `hi`). Then recursively call on left and right children with updated `lo` and `hi`. Return the size of the subtree, using the result of the recursive calls, or `-1` if there is an problem. This method passes bounds information down, and returns count and error information up.

resort() This is used by `xFirst()` and `yFirst()`. Reorganize the tree according to the current (just changed) comparator. This method should recursively traverse the tree and add each element into a new tree using the comparator. *Note:* we expect that in some sets, rasters with higher x-dimensions will also have higher y-dimensions. This means if we add them in sorted order, we might wind up with a very unbalanced tree. (The efficiency tests test this!) To avoid this potential problem, make sure you add the raster in the current node before you add anything in its left and right subtrees. You might have to create another private helper method to implement this method.

trimFromNull(array) This method copies data in the given array into a new array, stopping when it encounters a null. This will help you trim your arrays down for your `xRange` and `yRange` methods. This method is given to you.

copyXRange(array, index, n, lo, hi) This is used by `xRange` to recursively copy data into an array. It should copy the data stored at `n` into the array if it is in the range given by `lo` and `hi`. It should also recursively call on its children, with the appropriately updated index. Again, in some cases, you should avoid calling on certain subtrees. This method passes index and bounds information down, and returns index information up.

copyYRange(array, index, n, lo, hi) As `copyXRange`, but for the y-dimension.

4 What you need to do

Write the `compare()` method for one of the comparators. Implement all of the public operations described above. Implement and use the private helper methods to support them.

We provide internals tests, ADT operation tests, and efficiency tests. Make sure you do well on the efficiency tests, as many of the requirements for the BST only concern efficiency.