Homework # 13
due Monday, December 6, 10:00 PM

In this assignment you will update our TreeMap (from Homework #10) to implement an efficient `putAll` operation. You will implement mergesort on linked lists for this purpose.

`https://classroom.github.com/a/wwgrh6dc`

1 Motivation

The `putAll` implementation in `AbstractMap` works correctly by putting the elements one by one:

```
public void putAll(Map<? extends K, ? extends V> m) {
    for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
        put(e.getKey(), e.getValue());
    }
}
```

The "`? extends K`" syntax is saying that we don't care what the key type (or value type) is as long as it is compatible with our key type, so that "`put`" will be happy with the arguments.

If the tree stays balanced, then this implementation is efficient: each "`put`" call takes log time, and overall we end up with something like $O(n \log n)$ time. It's hard to do better than this.

Now, with our BST implementation, there's a danger that the tree will get unbalanced. Now, assuming that the tree is reasonably balanced *before* `putAll` runs, then as long as the map we are adding is no bigger than our current map, then we're unlikely to get too far unbalanced.[1] But if the map we're adding is large (and especially if it is also sorted!), then the default implementation can yield a very unbalanced tree in quadratic time (and if we use recursion, we can easily use up our entire allotted stack space).

A better approach is to *sort* the new entries and then merge them into our binary search tree while keeping things balanced. By using mergesort, we can guarantee $O(n \log n)$ time even if the tree isn't balanced. But how do we "merge" on a binary search tree?

2 Technique

It has been noted that binary-search-tree nodes look a lot like doubly-linked-list (DLL) nodes: they have some payload and then two link fields. In this homework, we exploit this commonality by repurposing the BST nodes to build a cyclic DLL and do mergesort on the lists. Specifically, we implement `putAll` with the following steps:

1. We create a cyclic DLL of the entries from the parameter map. There's no guarantee that the entries are in sorted order.

[1]To be perfectly safe, we should only allow $\log n$ additions.

2. We sort the list of new entries using mergesort. We now have a sorted cyclic DLL.
3. We convert our existing BST into a cyclic DLL without allocating any new nodes, just using the existing nodes, and interpreting "left" and "right" fields to refer to previous and next elements in a DLL, respectively. Since we started with a BST, and because we can, the result will be sorted.
4. We then *merge* the list of existing entries with the (sorted list) of the parameter map's entries. The result is one big sorted cyclic DLL.
5. Finally we convert the sorted list of entries into a completely new BST using all the nodes, converting them back into the BST that they were intended to be. We stash the new root and new size into the data structure and then (after checking that the invariant is still OK), continue on our merry way, with the added benefit that our BST now will be as balanced as possible.

This basic picture is complicated by a number of details:

- Because it seemed wasteful to create a raft of new dummy nodes to handle all the lists,[2] we will not be using dummy nodes in our cyclic DLLs. Instead the last element links around to the first one (with its `right` field) and *vice versa* (with the `left` field). The empty cyclic DLL will be represented by `null`.
- With `putAll`, a later "put" overrides an earlier one. So our "merge" operation will be somewhat unusual in the way it handles duplicates: if the same key shows up (or more precisely, if the two lists have keys that the comparator compares as zero), then the entry from the *first* list will be discarded. As a result, the result of a merge can have fewer elements than the sum of the lengths of the lists being merged.
- The need to handle null (empty) lists as special cases means the code for mergesort can quickly get unmanageable. The first two attempts of the instructor to write `merge` on cyclic DLLs had serious flaws. Rather than have huge series of pointer assignments (e.g. `p.left.right = t.right.left;`) in the code, it was decided to use several helper methods, each of which does a well-defined operation on cyclic DLLs.
- Furthermore, the code makes extensive use of assertions. So, for example, `merge` asserts that its two arguments are sorted cyclic DLLs. Now if these assertions were checked always, efficiency would be lost, but using Java's `assert` keyword, we can ensure that these are turned on while we are testing the functionality, but turned *off* during efficiency tests. Several errors in our solution were detected through these assertions.

3 Helper Methods

In this homework assignment, you are implementing a single public method, but many private helper methods. The starter code gives stubs and assertions for all of them. As mentioned

[2]And because a surprising number of students claimed on the mid-term examination that dummy nodes were necessary for cyclic lists.

above, each has a well-defined purpose. In our solution, the body of each is no longer than fifteen lines of code.

We represent cyclic DLLs by the head node (if not empty) or by null (if empty). Since the lists are cyclic, we can easily get to the tail (of a non-empty list) by going "left" from the head.

Here are our ten helper methods:

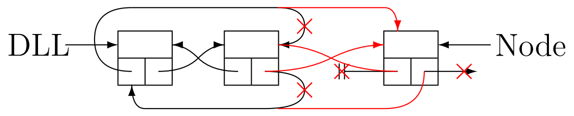
wellFormed(DLL) This method takes an alleged cyclic DLL and, in the same manner as our data structure well-formedness routines, checks that it is a well-formed cyclic DLL and reports any problems it finds. The result is also returned in a boolean.

isSorted(DLL) This method takes an allegedly sorted DLL and checks that it is indeed sorted according to the comparator. Like `wellFormed`, it reports anything that is not sorted, but also returns a boolean indicating whether the DLL is sorted.

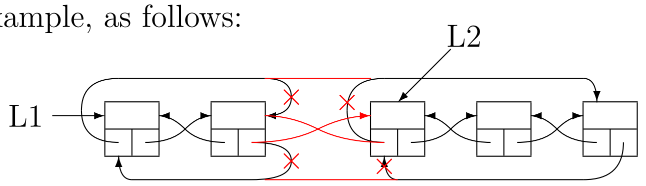
None of the remaining methods directly report problems, but they do assert conditions that should be true.

length(DLL) This method returns the length of the cyclic DLL. It assumes (and asserts) that the list is well-formed. But it does not require the list to be sorted.

add(DLL,Node) This method adds the node to end of the cyclic DLL and returns the list. Unless the list was empty, the returned list is the same (head node) as came in. The original values of the node's "left" and "right" fields are ignored. For example:



append(L1,L2) This method attaches the second DLL to the end of the first. If either of the arguments is null, we simply return the other. Otherwise, it makes changes, for example, as follows:



toList(BST) This helper method converts a binary search tree into a sorted cyclic DLL. It's passed the root of a subtree. If it's not null, it delegates to its two underlings to convert their subtrees into DLLs, and then puts together the results using the previous helper methods.

split(DLL,n) This helper method does the opposite of `append`, it splits apart a DLL after the first "n" nodes, and returns the *second* list. Here "n" will never be zero, nor greater than the length of the list. The special case `split(l,1)` can be used to remove the first element of the non-empty cyclic DLL `l`. This helper method is useful for simplifying the logic of our next helper method:

merge(L1,L2) This helper method is the "brains" of mergesort: it takes two sorted cyclic DLLs and returns a single sorted cyclic DLL that combines the elements of the two inputs. Along the way, duplicates in `L1` are discarded. It uses the comparator to decide which of the two lists to take from first. It handles the special case that either list can be null. This method is easiest to write if you use the previous helper methods.

sort(DLL) This helper method implements mergesort: unless we're at a base case, we split the list into two halves, sort them recursively and then merge the results.

toTree(DLL) This helper method takes a sorted cyclic DLL and converts it into a balanced BST, using recursion. It splits the list into three parts, where the middle part is a single node and the first and third parts are equal in size (or one different in length). The middle node becomes the root of the subtree and the other parts are recursively built into subtrees for "left" and "right."

The `putAll` method should use these helper methods to do its work.

4 Files

The repository for this assignment includes the following files:

src/edu/uwm/cs351/TreeMap.java The solution to Homework #10, together with stubs for all ten helper methods, the `putAll` overriding and 100 internal tests that can be accessed by:

src/TestInternals.java Access point for the internal tests.

src/TestPutAll.java Unit tests for the public method.

src/TestMergesort.java Exhaustive tests for mergesort using `putAll`.

src/TestEfficiency.java Efficiency test for `putAll`.

5 What you need to do

First unlock all tests to ensure you understand your tasks.

Then implement the ten helper methods. Use the `TestInternals` test suite to test each as you finish it (they are ordered for this purpose). Make sure to draw pictures for an example before coding, and if you fail a test case. Don't make random code changes; always make sure your code works on a paper example.

Then implement `putAll` using the helper methods. Test using the `TestPutAll` tests and then the `TestMergesort` tests. We also provide random testing.

Finally ensure that your implementation is efficient using `TestEfficiency`. The test should take at most a couple of seconds.