Testing Report (User 48)

Overview

Methods that Failed:

- toString()
- insert()
- remove()
- split()
- join()
- iterator()
- Custom edge cases

Testing Methodology

insert()

I tested this method by first inserting random integer keys and values into your TreapMap. I stored the values I inserted in a hash map. After inserting 1000 elements, I made sure that every element was found in the HashMap was found in the TreapMap and vice versa. I also made sure that the treap satisfied the binary tree property and the heap property.

remove()

I tested this method similarly to how I tested *insert()*, except I gave the treap a 33% chance of removing an element. I made sure that when an element was removed, when trying to look it up again, null would be returned. I did the same checks as I did in *insert()*.

split() and **join()**

I tested this method by inserting elements randomly into the TreapMap with sizes of powers of 2 until 1024. I split the Treap, made sure that the left Treap had elements strictly less than the key, and made sure that the right Treap had elements greater than or equal to the key. I also made sure both Treaps had the same keys as the original Treap. I then joined the subtreaps and made sure it had the same elements as the original treap. For the subtreaps and the joined treap, I made sure they satisfied the binary tree property and the heap property.

iterator()

I tested this method by constructing a Treap similarly to the Treap constructed in remove() and made sure that each element as it was iterating was greater than its previous element. I made sure that each element that was iterated through was present in the TreapMap. I also made sure that ConcurrentModificationException was thrown when the TreapMap was altered. I made sure that operations that did not affect the TreapMap, such as removing a node that wasn't found or passing in nulls for all operations, did not affect the iterator.

toString()

This was indirectly tested through my testing of the heap property and the bst property. I built a parser to build a tree from string output.

Specific Edge Cases

The edge cases I tested were as follows. I made sure that on an empty Treap, insert and join returned null while split led to an array of empty strings when calling *Arrays.toString()* and join with an empty Treap led to another

empty Treap. I also made sure that when you pass in null to any of these methods, the methods return null and does not alter the Treap. I also made sure that when all entries were removed, the treap was properly reset to an empty Treap.

Notes

- Sorry, I was unable to run your code thoroughly through my harness since it relies on the *toString()* and *iterator()* methods
- I had trouble constructing trees of a large size. This may be due to a bug in rotations or your internal find() (or whatever you call it) method
- *join()* could not join two empty treaps
- I got null pointer exceptions from the following methods. I could not test *join()* since my cases rely on *split()* to be functional (and I couldn't construct two trees manually to see if it worked):
    - *toString()*
        - Occurred for all treaps including empty ones
    - *split()*
        - Occurred for both when the split key was in the treap and when it wasn't
    - *insert()*
        - Occurred for large treaps
    - *remove()*
        - Occurred when trying to remove a node that didn't exist in the treap. Possibly could have occurred in more places
        - Also occurred when trying to delete the root
    - *iterator()*
        - Occurred for all treaps
- I generally could not insert more than one node into the treap.
- Suggestions: as NullPointerException signifies, you probably are not handling your null references properly. You may be forgetting to assign a reference a value, etc. There's two ways you can handle this issue. Either make a SENTINEL node (a node that represents null) so that every time you get a null pointer exception, you know that wasn't intentional or just manually fixing the bad nulls.