Tree deletion

Insertion: we allocated new nodes from top to bottom - $O(\log n)$
Can we do the same for removing a node?
Tree deletion, no mutation
Tree deletion

tree.remove(4)

Easy case: if the value we want to remove is to the left
return new Tree<>(value, left.remove(deadValue), right);
Tree deletion

tree.remove(4)
Easy case: if the value we want to remove is to the right
return new Tree<>(value, left, right.remove(deadValue));
Tree deletion

tree.remove(4)
Okay, we found it. Since it’s empty on one side, just return the other.
if (left.empty())
    return right;
if (right.empty())
    return left;
Tree deletion

tree.remove(5)
But what if both children are non-empty?
Goal: reduce to a smaller problem then continue recursively.
Tree rotation

```java
Tree<T> rotateRight() {
    if (left.empty()) return this;
    return new Tree<T>(
        left.getValue(),
        left.getLeft(),
        new Tree<T>(value, left.getRight(), right));
}
```
Tree deletion

tree.remove(5)
But what if both children are non-empty?

**Rotate and recurse!**

```
return rotateRight().remove(deadValue);
```
Tree deletion

tree.remove(5)
But what if both children are non-empty?

*Rotate and recurse!*

```java
return rotateRight().remove(deadValue);
```

![Tree Visualization](image)
Tree deletion

tree.remove(5)
But what if both children are non-empty?

**Rotate and recurse!**

```java
return rotateRight().remove(deadValue);
```

```plaintext
2
  ↓
  ↓
5 --> 9
  ↓
  ↓
4 --> 7
  ↓
  ↓
3 --> 6 --> 8
```
tree.remove(5)
But what if both children are non-empty?
**Rotate and recurse!**
```java
return rotateRight().remove(deadValue);
```
Tree deletion

tree.remove(5)
But what if both children are non-empty?

*Rotate and recurse!*

```python
return rotateRight().remove(deadValue);
```

diagram:
```
    4
   / \
  2   5
   \   
    9

    7
   / \
  6   8
```

diagram:
```
  3
 / \
2   4
   / \
  5   9
 /     
7       8
```
Tree deletion

tree.remove(5)
But what if both children are non-empty?
**Rotate and recurse!**

```
return rotateRight().remove(deadValue);
```
Tree deletion

tree.remove(5)
But what if both children are non-empty?
**Rotate and recurse!**

\(O(\log n)\) time, \(O(\log n)\) new nodes
And original tree still there if you kept a pointer
Because *no mutation*!
Tree balance

What if we inserted numbers in **sorted** order?

```java
ITree<Integer> tree = Tree.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

We won’t get a very pretty tree: O(n) worst case behavior.
Solution: “balanced” trees

Many, many different algorithms for doing this (AVL, red-black, etc.)

We’re going to learn about treaps [Aragon & Seidel, 1989]
(Then-sophomore Wallach had Seidel as his Comp215-equiv. professor in 1990.)
You’ll implement it for this week’s assignment, so pay attention.

Treap = tree + heap
Every node has a random integer (“priority” or “heapVal”), created when inserted.
  Tree property: nodes to the left are less-than current node; nodes to the right are greater-than (recursively)
  Heap property: current node’s priority is less than left and right’s priority (recursively)

More on heaps / priority queues on Friday. Enough to get you started today.
Tree vs. heap property

Tree value (yellow), heap value / priority (blue)
Treap insertion

First, insert like it’s a normal tree (inserting “6” here)
Note that we need to copy the priorities as we allocate new nodes
Treap insertion

First, insert like it’s a normal tree (inserting “6” here)
Note that we need to copy the priorities as we allocate new nodes
Treap insertion

First, insert like it’s a normal tree (inserting “6” here)
Note that we need to copy the priorities as we allocate new nodes
Treap insertion

First, insert like it’s a normal tree (inserting “6” here)
Note that we need to copy the priorities as we allocate new nodes
Treap insertion

First, insert like it’s a normal tree (inserting “6” here)
Note that we need to copy the priorities as we allocate new nodes
Treap insertion

First, insert like it’s a normal tree (inserting “6” here)
Note that we need to copy the priorities as we allocate new nodes
Treap insertion

We’ve satisfied the “tree” property, but not the “heap” property.
The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.
Treap insertion

We’ve satisfied the “tree” property, but not the “heap” property
The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.
We’ve satisfied the “tree” property, but not the “heap” property.
The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.

Is treap.priority < treap.left.priority and < treap.right.priority?
Treap insertion

We’ve satisfied the “tree” property, but not the “heap” property

The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.

Is treap.priority < treap.left.priority and < treap.right.priority?
No! left priority is smallest.
We’ve satisfied the “tree” property, but not the “heap” property.
The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.

Rotate right to get smallest on top.
We’ve satisfied the “tree” property, but not the “heap” property
The newly inserted node gets a random priority; we must “heapify”. First insert (recursively), then “heapify” on the way out.

Rotate right to get smallest on top.
We’ve satisfied the “tree” property, but not the “heap” property
The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.

Rotate right to get smallest on top.
Treap insertion

We've satisfied the “tree” property, but not the “heap” property
The newly inserted node gets a random priority; we must “heapify”. First insert (recursively), then “heapify” on the way out.

Rotate right (again) to get smallest on top.
We’ve satisfied the “tree” property, but not the “heap” property
The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.

Rotate right (again) to get smallest on top.
Treap insertion

We've satisfied the “tree” property, but not the “heap” property
The newly inserted node gets a random priority; we must “heapify”. First insert (recursively), then “heapify” on the way out.

Rotate right (again) to get smallest on top.
We’ve satisfied the “tree” property, but not the “heap” property
The newly inserted node gets a random priority; we must “heapify”.
First insert (recursively), then “heapify” on the way out.

Heap property satisfied. We're done.
Treap insertion

But that doesn’t look particularly balanced!
Yes, but let’s talk about the odds.
Treap: odds of degenerate case

Degenerate case: start with a “linked list” tree

Measure odds that inserting new value preserves the linked list property

Assume all numbers selected at random. Linked list is highly unlikely.

\[ P \left( h_n > h_{n-1} \right) = 2^{-\left(n-1\right)} \]
Treap: expected # of rotations (in general)

\[ P(1 \text{ rotation}) = 0.5 \]
\[ P(2 \text{ rotations}) = 0.25 \]
\[ P(3 \text{ rotations}) = 0.125 \]
\[ \lim_{n \to \infty} \sum_n P(n \text{ rotations}) = \]
\[ \lim_{n \to \infty} \sum_n n2^{-n} = \]
\[ 2 \text{ rotations} = O(1) \]
You need a source of random numbers (java.util.Random)
Warning: you might get the same priority more than once; don’t panic

When you remove a value, preserve the heap and tree properties
Warning: the remove code from Tree won’t magically work here

Careful if you want to inherit from edu.rice.tree.Tree
Warning: any method in Tree that has “new Tree...” in it won’t work
We advise you against extending Tree.

Careful with the empty treap
What’s the priority of the empty treap?
Treap performance

Prof. Dan's code, JDK 8u51, 2014 MacPro, inserting integers

1M random tree inserts: 0.851 μs per insert
1M random treap inserts: 1.194 μs per insert

10K random tree inserts: 0.199 μs per insert
10K random treap inserts: 0.359 μs per insert

10K sequential tree inserts: 113.925 μs per insert
10K sequential treap inserts: 0.328 μs per insert

Randomized algorithms rock.
Often easier to implement than deterministic. Fast runtime. Stable performance.
Treaps are cool

Attackers can’t pick inputs that cause worst-case behavior
So long as attackers can’t predict the random numbers.
➡️ Useful for intrusion detection systems (e.g., looking for network scanners).

Optional: Replace random priority with a hash of the tree node value
Treap becomes deterministic (same inputs, any order → identical treap structure)
➡️ Recursive hashes over these treaps useful for quick set-equality tests

(Advanced: Read up on Merkle hash trees. Treaps show up in many “authenticated data structures.”)

Once you understand randomized data structures
Useful throughout computer science
(Seidel’s specialty is computational geometry: full of randomized algorithms.)
Live coding (time permitting)

Walkthrough of TreeTest.testTreapPerformance

Walkthrough of different tree traversals
inorder consumer vs. eager list vs. lazy list

Challenge: add together the smallest 1000 entries in a tree with 1 million integers