What about remove?

Idea: Take the smallest node value larger then the one to be removed, and replace the removed node with that one.
Case 1: \[
\begin{align*}
&\text{B is null} \\
&\text{y is staying} \\
&\text{x is being removed} \\
&\text{Know: all values in y's right subtree are } > y
\end{align*}
\]

Case 2: \[
\begin{align*}
&\text{A is null} \\
&\text{Put B where x was}
\end{align*}
\]
Case 3: neither $A$ nor $B$ is null

We know: Every value in $A$ is smaller than all values in $B$.

Largest value in $A$ will go in place of $x$.

$\Rightarrow$

(That node has at most one child, on the left $\Rightarrow$ easy to remove from $A$.)
General idea:

1. Find x
2. Change appropriate pointer of x's parent to refer to the promoted node
3. Remove promoted node from A+B and let that node's children be A+B

Setup:

```c
remove(x) {
    root = removeFromSubtree(x, root);
}  
```

will return the subtree we removed x from
removeFromSubtree(x, subtree) {
    if (subtree == null) return null;
    if (x == subtree.value)
        return removeRootOfSubtree(subtree);
    else if (x < subtree.value)
        subtree.left = removeFromSubtree(x, subtree.left)
    else
        subtree.right = removeFromSubtree(x, subtree.right)
    return subtree;
}
```cpp
removeRootOfSubtree(ptr) {
    if (ptr.left == null)
        return ptr.right
    else if ptr.right == null
        return ptr.left
    else  // neither A nor B is null {
        TreeNode z = removeRightMost(ptr.left);
        z.right = ptr.right;
        if ptr.left != z)  // B
            z.left = ptr.left  // A
    }
}
```
removeRightMost (subtree)  // return removed node
  while (subtree.right != null) {
    if (subtree.right.right == null) // is right most
      temp = subtree.right
    subtree.right = subtree.right.left
    return temp
  }
  subtree = subtree.right
}
Map interface

put(key, value)
get(key) \Rightarrow value

JUnit

TestCase

assertEquals ... CreateMap()

MapTest

TreeMapTest
createMap

OrderedListMapTest
createMap

used in testing