Non-mutating methods

min, max, search, pred, successor
in-order traversal

Just ignore the color!

It's a valid binary search tree

Cost $O(h) = O(\log n)$

In a binary search tree, time complexity to return all elements in range $[b, e]$ is $O(h + k)$ where $h$ is the height of tree and $k$ is the number of elements in the given range.
Insertion

Insert the new element in standard way as a red node.

Black Balanced \{ key properties to NoDoubleReds \} worry about too expensive to fix if violated. Keep it preserved throughout

Take possible NoDoubleReds violation & use recoloring & rotations to move violation up towards root until fixed or reaches root.
Case 1: extraRed’s uncle is red (extraRed could be any of y’s four grandchildren)

possible violation with its parent
(no other violation of noDoubleReds + BlackBalanced is preserved)
Case 2: extraRed’s uncle is black (could be mirror image)

Case 2a: extraRed opposite child as its parent (zig-zag)

Lift the extraRed

extraRed

extraRed

Only Violation of NoDoubleRed

Still have violation (at same level of tree) but now zig-zig
Case 2: extraRed's uncle is black

Case 2b: extraRed same child as its parent (zig-zig)

Lift the extraRed's parent (after which there is no extraRed remaining)

DONE!
Example, insert 0 into below

only nodes are colored

extra red →

extra red

extra Red

extra red

extra Red

extra red

extra Red
Time complexity for insert

binary search tree insertion

$O(h) = O(\log n)$

insert fix up

$\leq h/2$ case 1's (constant time each)

$\leq$ once for case 2a+2b

$O(h)$
Deletion

(details on pages 523-528 G&G)

298-293 CLRS

If node $x$ to delete has 2 children, replace $x$ by its successor $y$ (could use predecessor) whose color is given color of $x$

Then remove $y$. How to do this!

In other cases ($x$ has 0 or 1 child) let $y$ be the node to delete
If $y$ was red
we're done.

If $y$ is black
we make its child "double black"
treat it as 2 in black height
B-Tree Data Structure

First finish overview of deletion in red-black tree
Recall that $y$ is node being deleted (successor of original node to delete had 2 children).

doubleBlack is initialized to reference child of $y$ (or frontier/null if $y$ is leaf)

```java
void deleteFixUp(RBNode doubleBlack){
    while (doubleBlack != root && doubleBlack.isBlack()) {
        // stop if at root or red node
        Use rotations/recoloring to move double Black up towards root
    }
    doubleBlack.setBlack(); // used when loop terminates with a red node as doubleBlack
}
```

red (or root) when loop exits