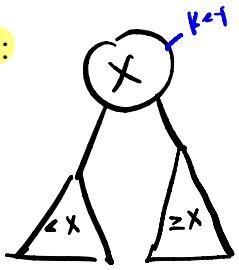
// Balancing Binary Search Trees

Binary Search Trees

- Represented by a linked data structure of nodes.
- root(T) points to the root of the tree T.
- Each node contains fields:
  - Key
  - $left$ which is the pointer to left child node (NIL if no left child exists)
  - $right$ which is the pointer to right child node (NIL if no right child exists)
  - p which is the pointer to parent ($p[root[T]] = NIL$)
  - Also whatever satellite data we're storing in the tree

Stored keys must satisfy the **binary search tree property**:



- $\forall y$ in left subtree of x, then $key[y] < key[x]$
- $\forall y$ in right subtree of x, then $key[y] \geq key[x]$

Find the node in our tree that contains key k using the following algorithm.

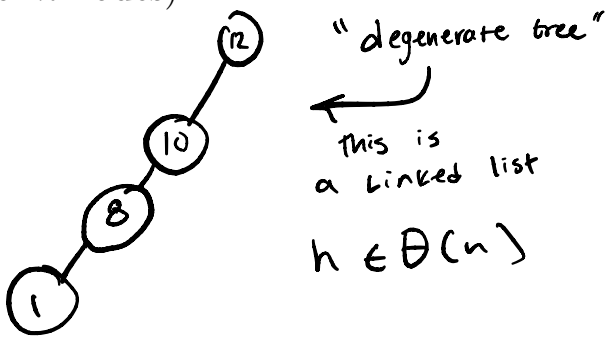
Algorithm 1 node TreeSearch(node x, int k)

```
1: if x == NIL or k == key[x] then
2:     return x;
3: end if
4: if k < key[x] then
5:     return TreeSearch(left[x], k);
6: else
7:     return TreeSearch(right[x], k);
8: end if
```

The runtime of the above algorithm depends on the height, h, of the tree T. Specifically, it runs in time $O(h)$.

If the tree is balanced $h \in \Theta(\log n)$
(the distance from the root to any leaf is roughly equal)

In the worst case $h \in \Theta(n)$
(the tree looks like a linked list of n nodes)



// different than hash tables, as we dont care about # of keys. Our tree will be built w/ any # of keys

dynamic size data structure

We can ensure the tree is balanced using **red-black trees**. This data structure uses an extra one-bit color field. // does not take up much space

worst case

The cost of inserting, deleting, and searching for keys will then take $O(\log n)$ time. // insert / delete are slightly different from normal B.S.T

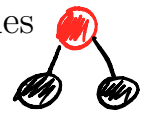
Red-black tree properties:

(1) Every node is red or black

(2) The root is a black node

(3) The leaves (NILs) are black nodes

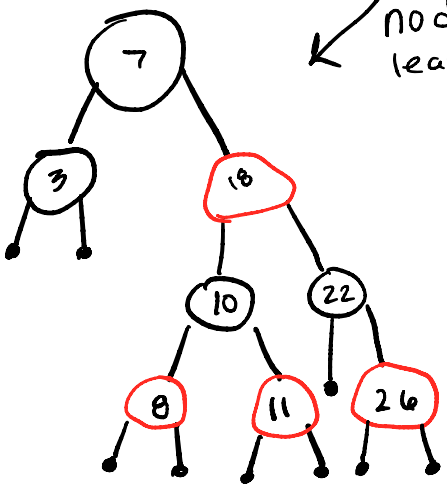
(4) If a node is red, then both of its children are black nodes



(5) All simple paths from the root down to a leaf of the tree contain the same number of black nodes.

Example of a red-black tree:

All have 3 black nodes along path to leaves

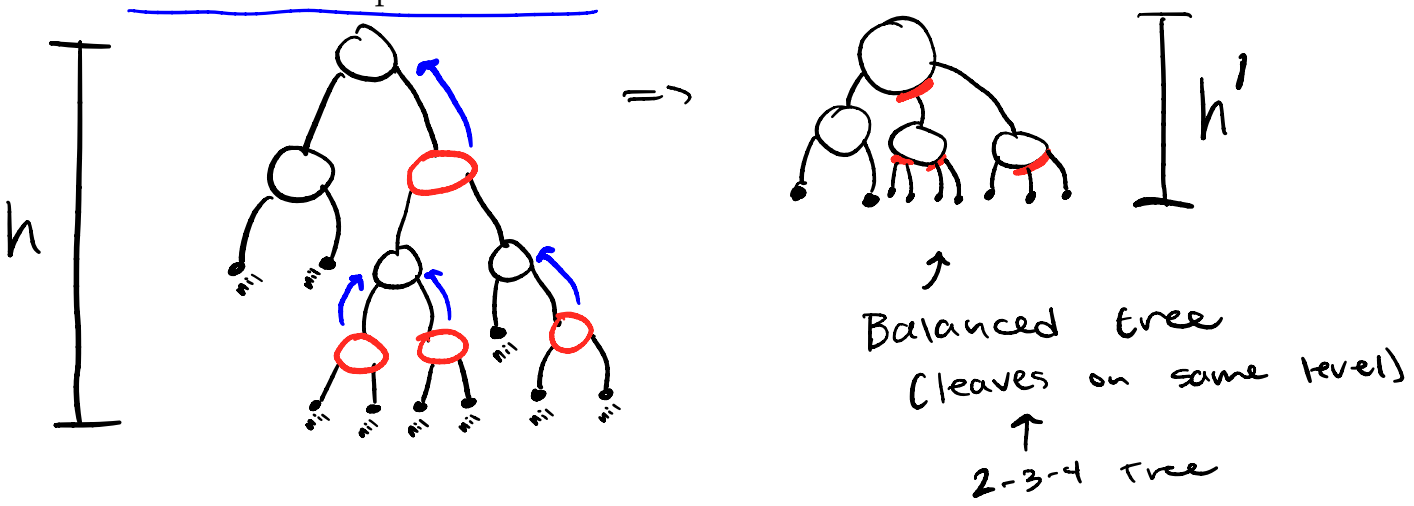


=> Tree does not appear balanced, but because it is a red-black tree - it is.

Theorem.
A red-black tree with n keys has height $h \leq 2\log(n+1)$.

$\therefore h \in \Theta(\log n)$

Proof: Let's imagine collapsing the red nodes of a red-black tree into their black parent nodes



$\therefore h/2 \leq h'$

- Collapsing the red nodes yields a tree where every node has 2, 3, or 4 children (i.e a 2-3-4 tree).
- Let h denote the height of the original red-black tree.
- Let h' denote the height of the 2-3-4 tree.
- The leaves of the 2-3-4 tree are all on the same level (from property 5 of red-black trees).
- $h' \geq h/2$ (from property 4 of red-black trees) → worst case: every other node is red
- The number of NILs in both trees is $n+1$ (can prove this with an inductive argument). ↓ n is # of keys

=> $n = 8$, # nils $= 9$

=> $n+1 \geq 2^{h'}$

↳ # nils if every node in 2-3-4 tree had 2 children

=> $\log_2(n+1) \geq h' \geq h/2$

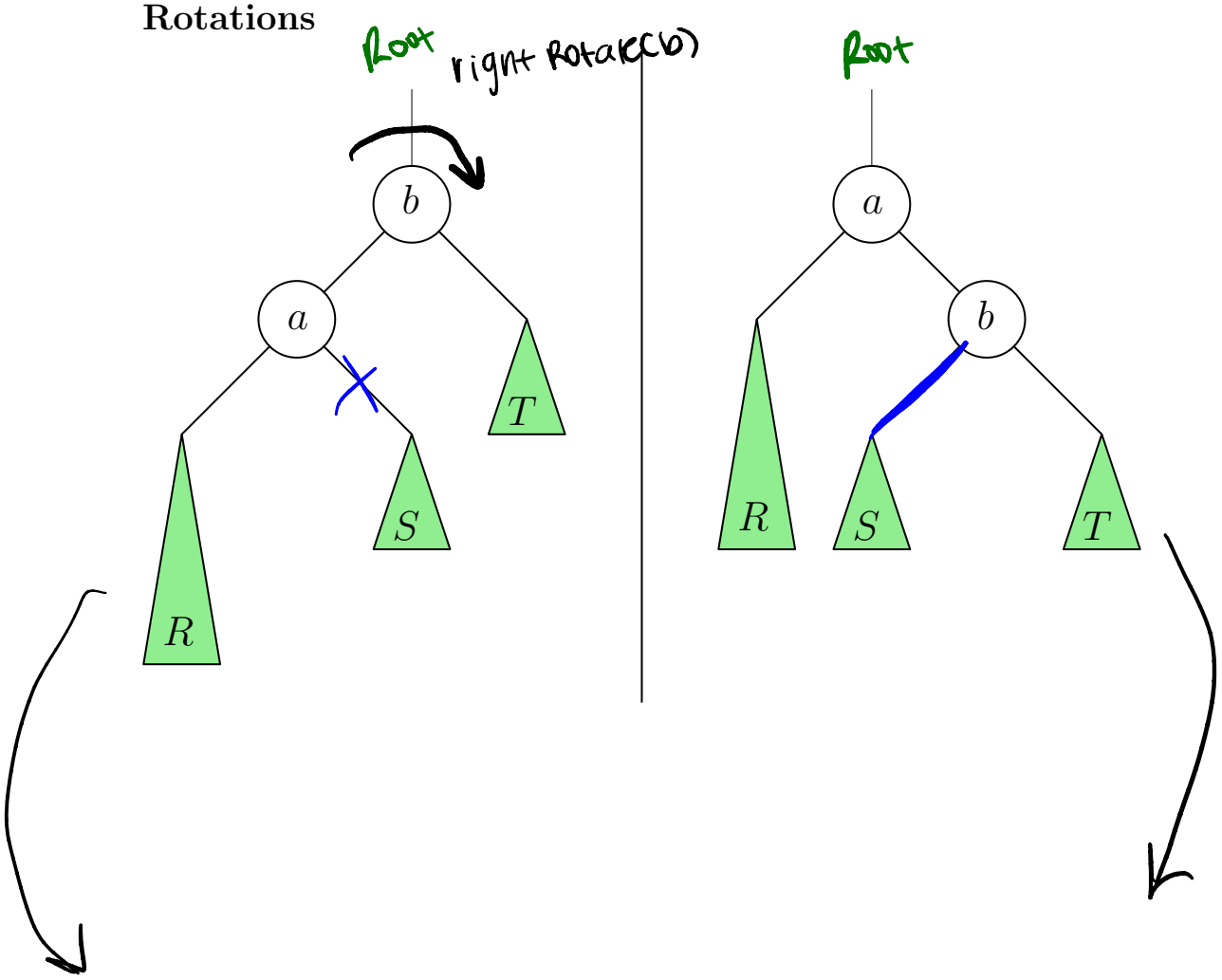
=> $2\log_2(n+1) \geq h$ □ // upper bound of h

The operations Insert and Delete cause modifications to the red-black tree:

- The operation itself.
- Nodes may change colors.
- May require restructuring the links of the tree using rotations.

// R, S, T are large sub trees

Rotations



Rotations maintain the binary search tree property:

$\forall r \in R, s \in S, t \in T: r < a \leq s < b \leq t$

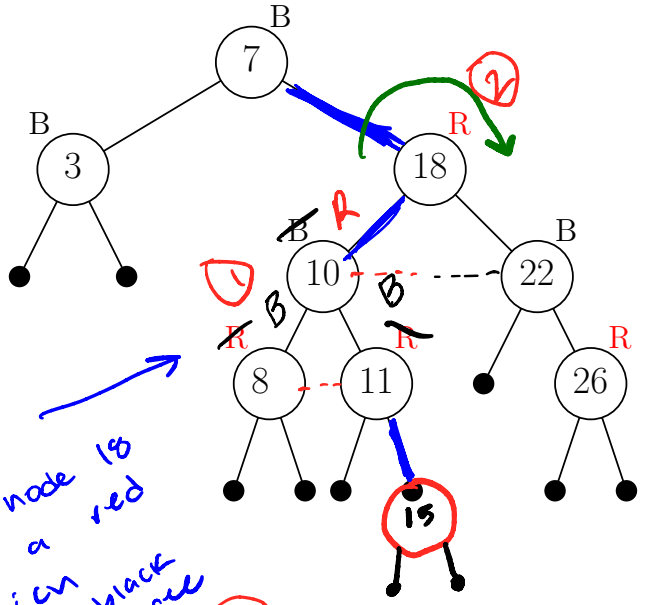
=> change only 3 pointers to rotate nodes

Rotations can be performed in time $\Theta(1)$

Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example.



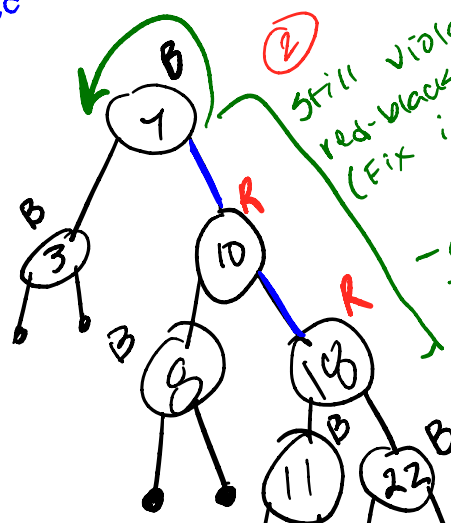
insert (15) (1)
∴ 15 placed in that location as it satisfies B.S.T
// This violates red-black tree properties ... must fix

(case 1) (1) Push red color up the tree
∴ notice node is now has a red child which violates red-black tree (2)
↳ case 1 can be applied here

∴ case 2 (2)

=> right rotate (18)
// Not changing colors, instead moving red nodes around

=>

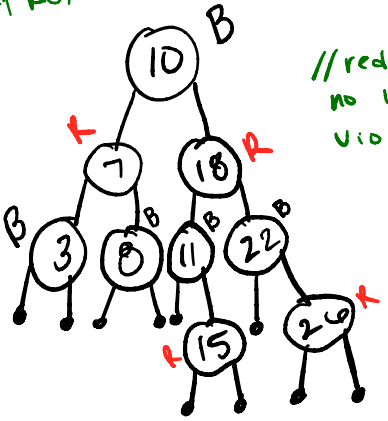


(2) still violates red-black tree (Fix in 1 more step)

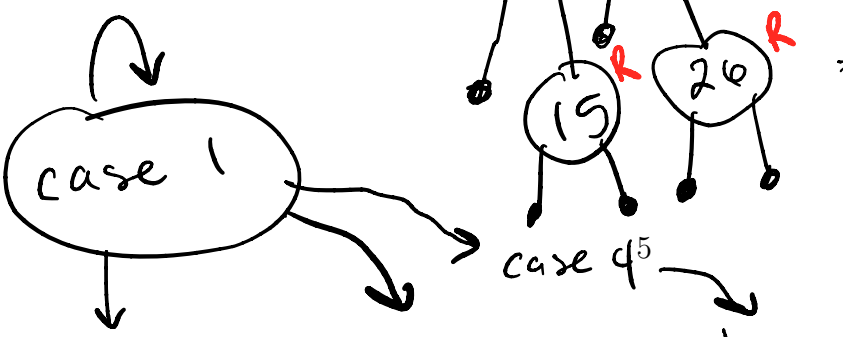
-case 3

=> Fix by doing left rotate on root & a little bit of re-coloring => Moving redness to other side (red is moves red to other side)
=> leftRotate(7)

=>



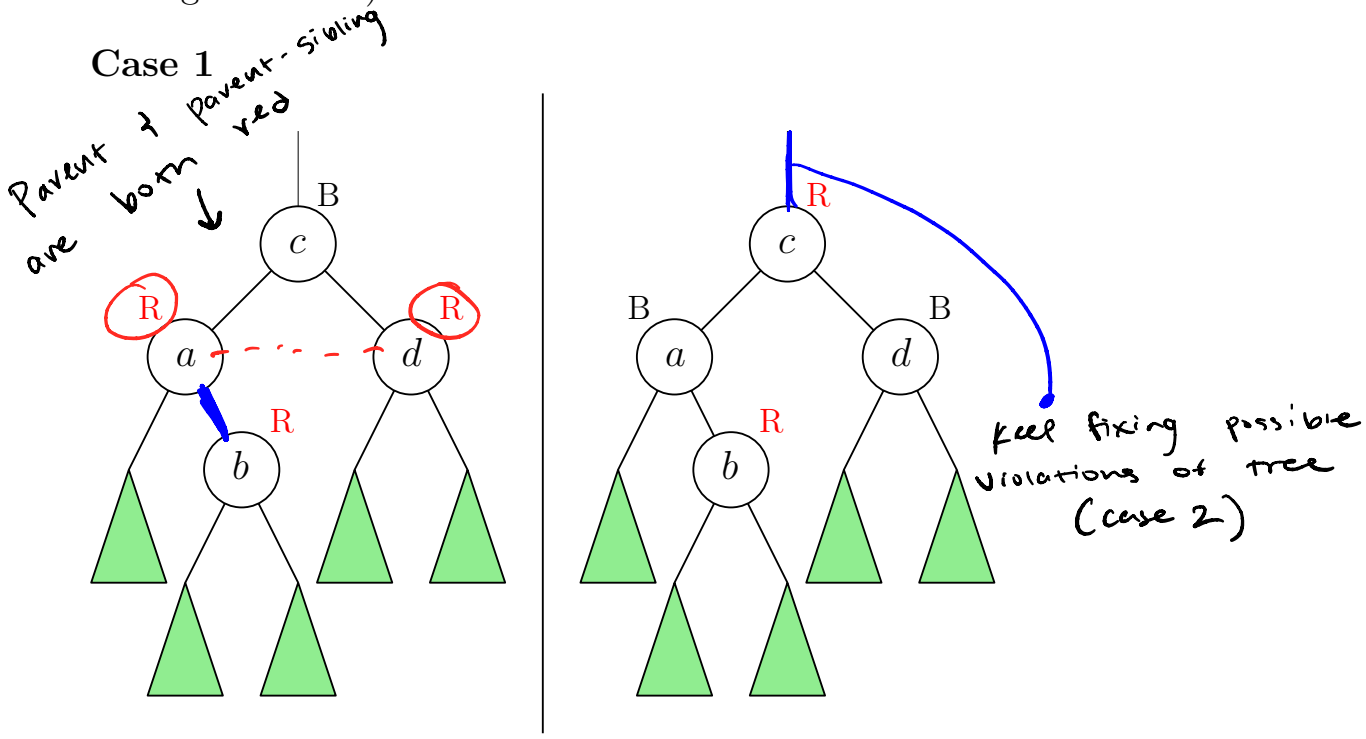
// red-black tree no longer has violations



=> case 2 → case 3 → done

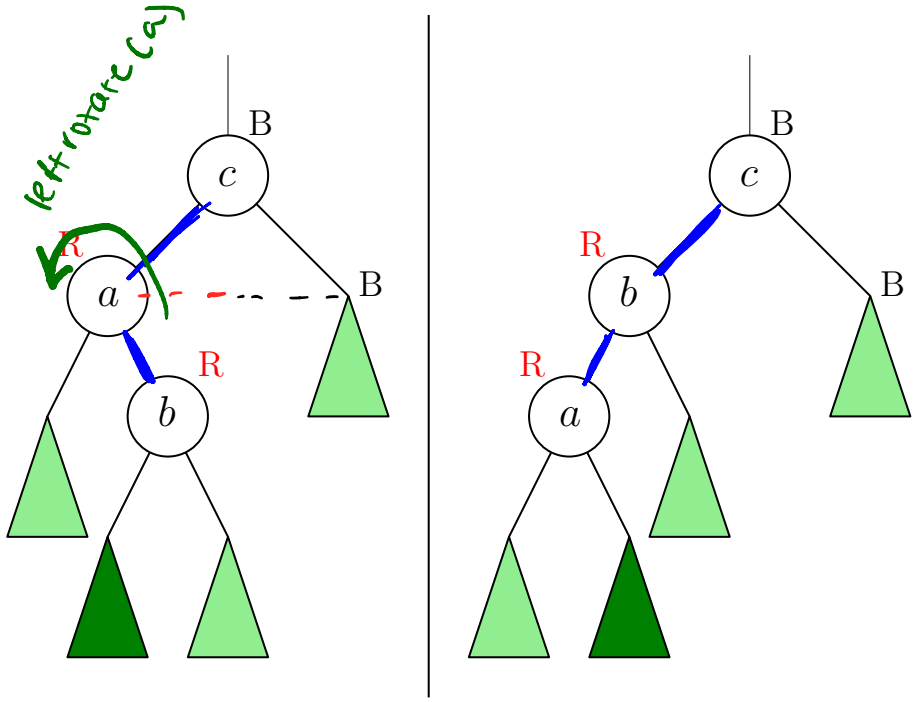
The insertion algorithm considers the following 3 cases (and their left-right mirrors):

Case 1



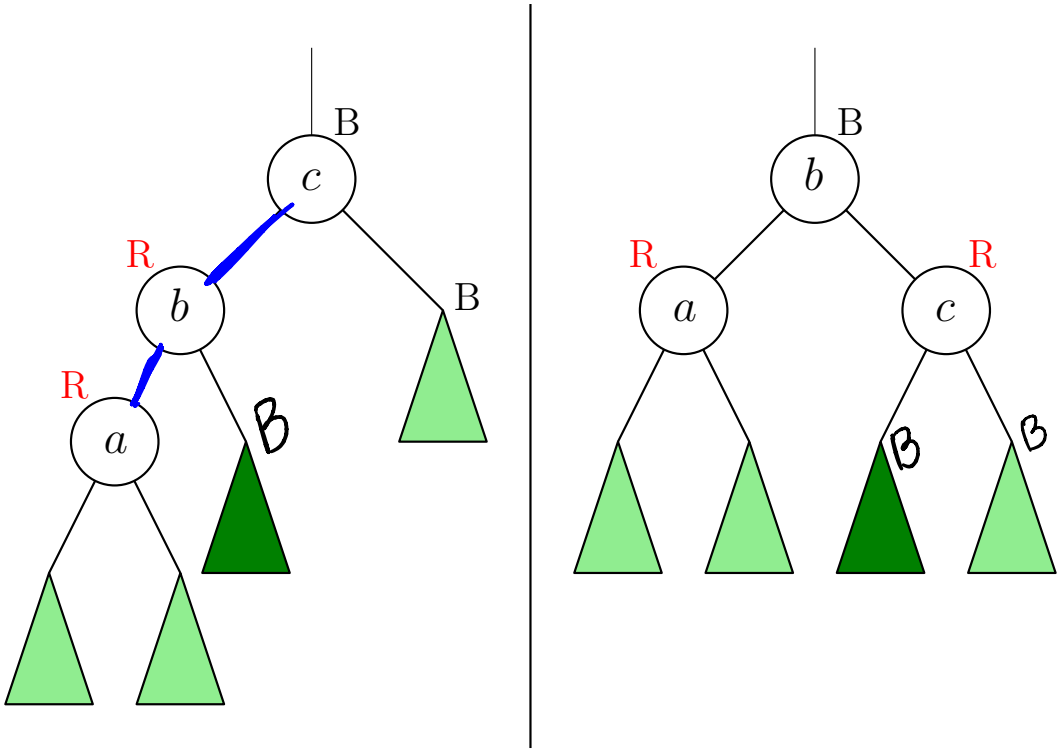
- (continue fixing violations up the root)

Case 2



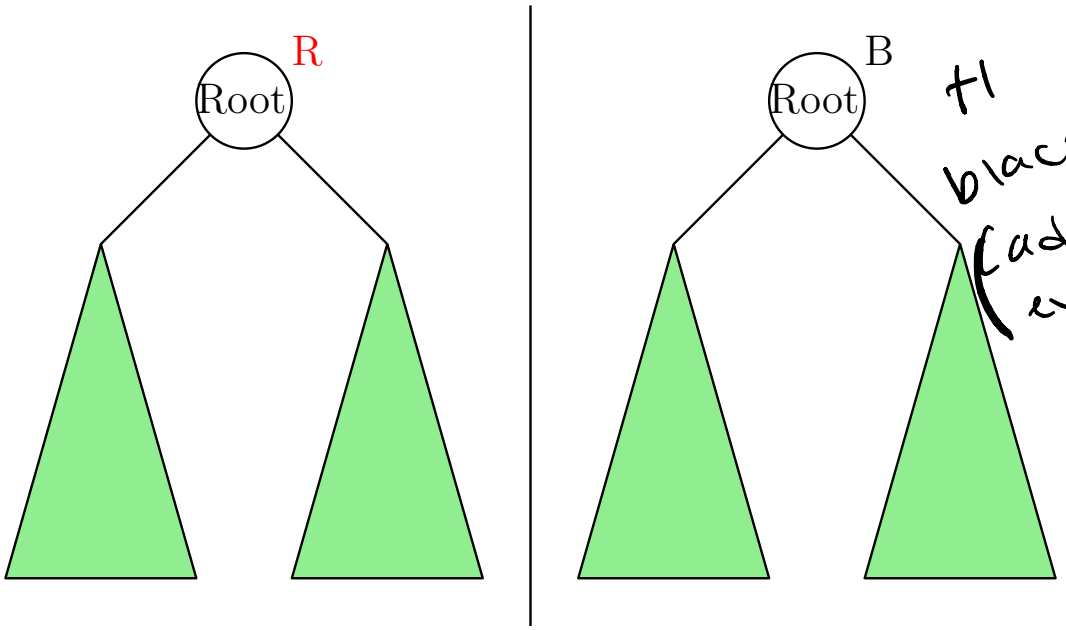
(case 2 always leads directly into case 3)

Case 3



- (no more work is required at this point)

Case 4



(no more work is required at this point)

=> If root ever red, just change to black

// insert(x) $\in \Theta(\log n)$

Analysis

Case 2 and case 3 require only 1 or 2 rotations to fix the tree.

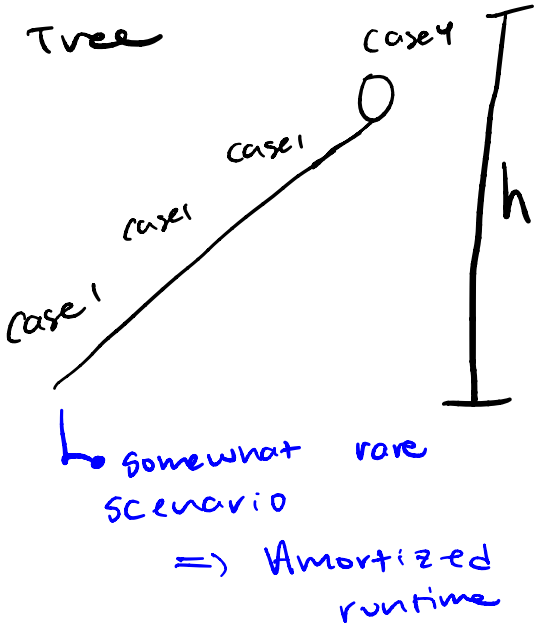
Case 4 only requires only one recolor to fix the tree.

Each case 1 requires three recolor but it doesn't fix the tree and instead pushes it further up the tree.

$\Rightarrow$ Worst case requires $O(\log n)$ operations since we may need to push the problem all the way to the top of the tree.

↳ when you get case 1 all the way up the tree

∴ CLRS for delete(x) if interested.



↳ somewhat rare scenario

⇒ Amortized runtime