Advanced DS

Data Structure and Algorithms

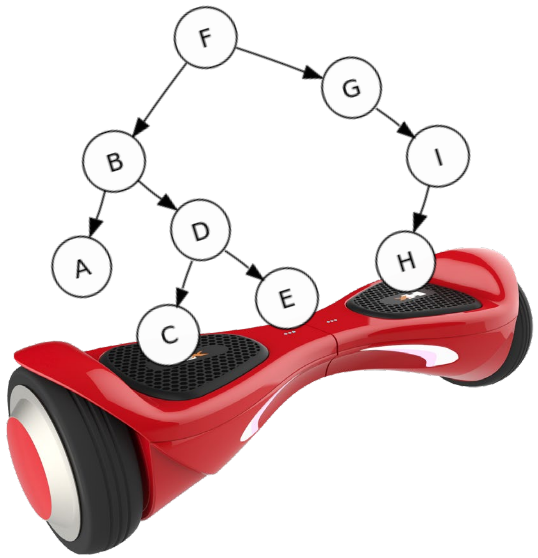
Spring 2024

Red Black Tree

eduko.spikotech.com

University of Engineering and Technology
Lahore Pakistan

Self-Balancing
Binary Search Trees

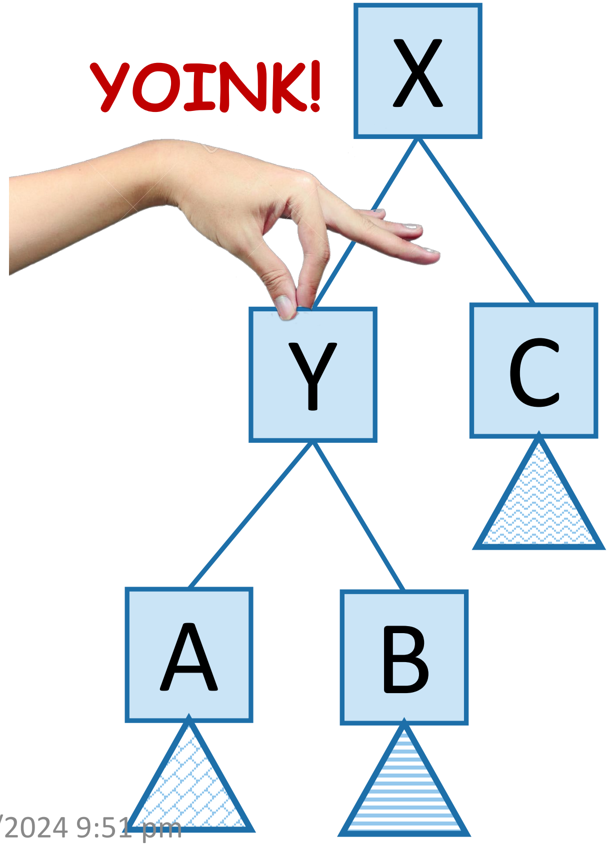


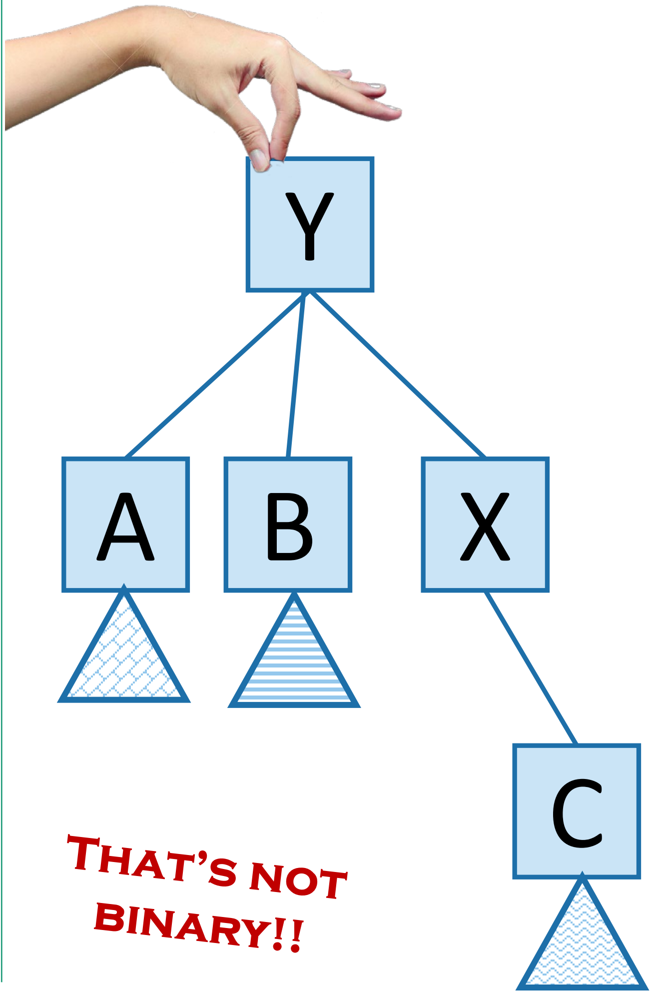
Idea 1: Rotations

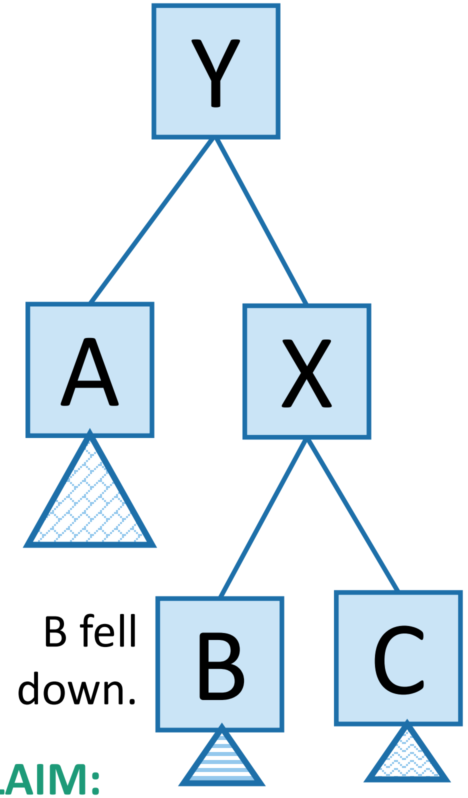
No matter what lives underneath A,B,C,
this takes time O(1). (Why?)

- Maintain Binary Search Tree (BST) property, while moving stuff around.

Note: A, B, C, X, Y are variable names, not the contents of the nodes.

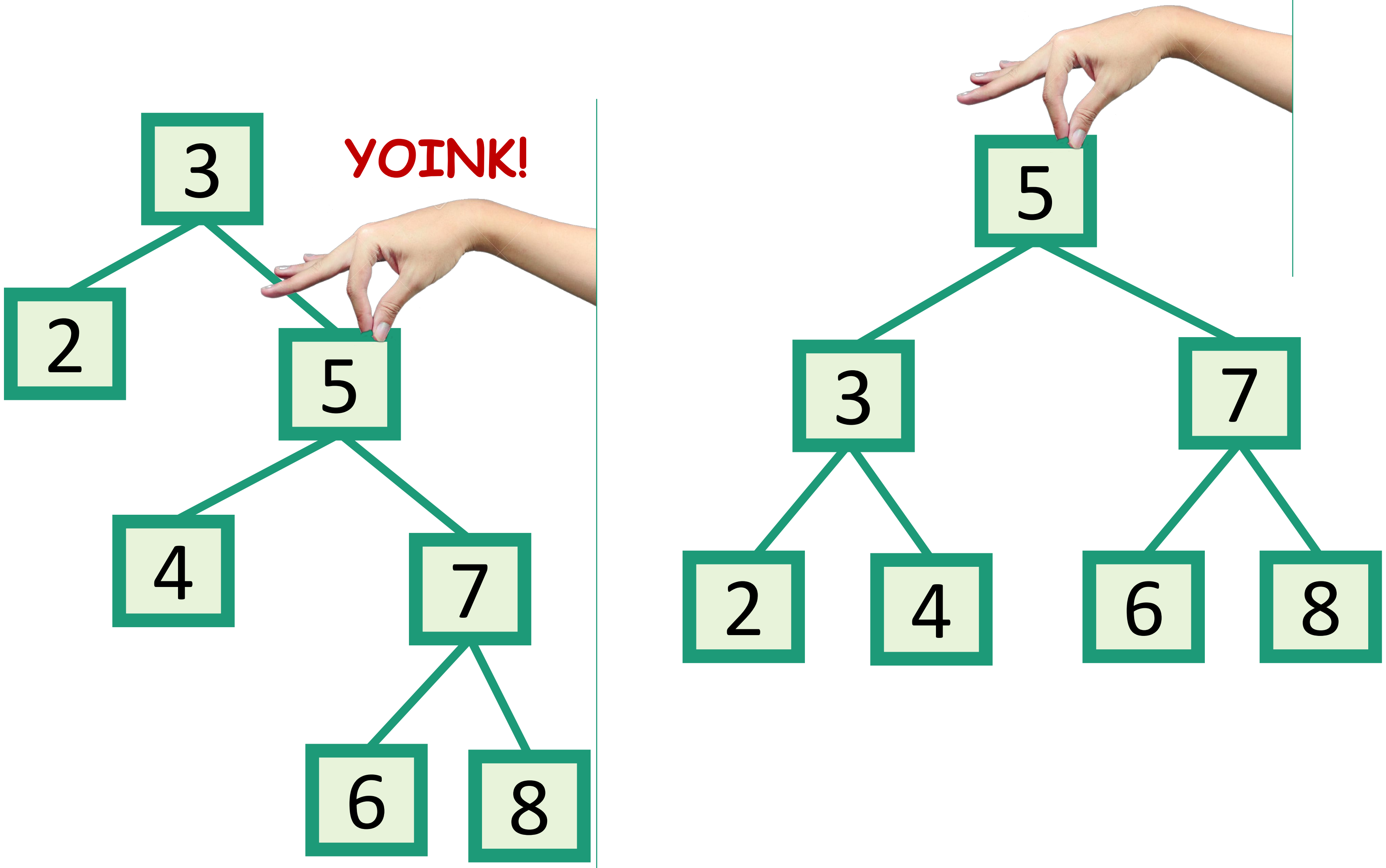






CLAIM:
this still has BST property.

This seems helpful



Strategy?

- Whenever something seems unbalanced, do rotations until it's okay again.

Lucky the Lackadaisical Lemur

Even for Lucky this is pretty vague. What do we mean by "seems unbalanced"? What's "okay"?

Idea 2: have some proxy for balance

- Maintaining perfect balance is too hard.
- Instead, come up with some proxy for balance:
  - If the tree satisfies [SOME PROPERTY], then it's pretty balanced.
  - We can maintain [SOME PROPERTY] using rotations.

There are actually several ways to do this, but today we'll see…

RB Tree Introduction

CLRS Chapter 13 (13.1)

Red-Black Trees

- A Binary Search Tree that balances itself!
- No more time-consuming by-hand balancing!
- Be the envy of your friends and neighbors with the time-saving...

Red-Black tree!

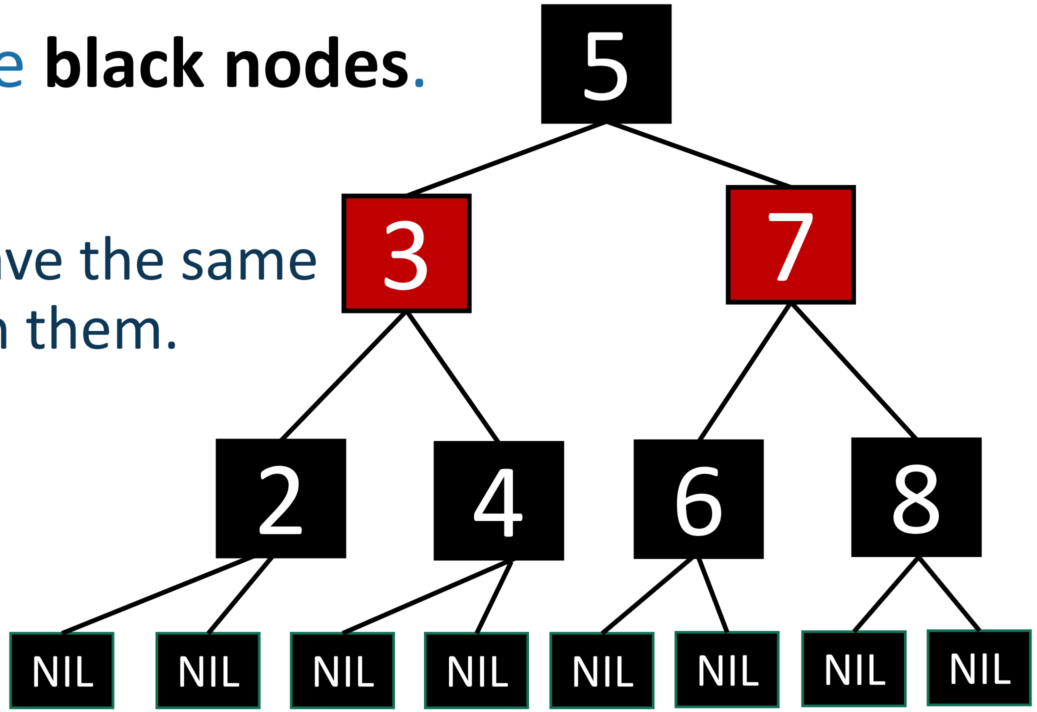
Maintain balance by stipulating that **black nodes** are balanced, and that there *aren't too many* **red nodes**.

It's just good sense!

Red-Black Trees

obey the following rules (which are a proxy for balance)

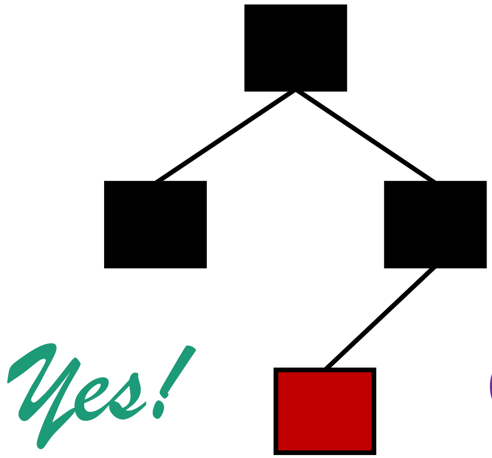
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



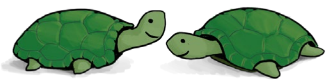
I'm not going to draw the NIL children in the future, but they are treated as black nodes.

Examples(?)

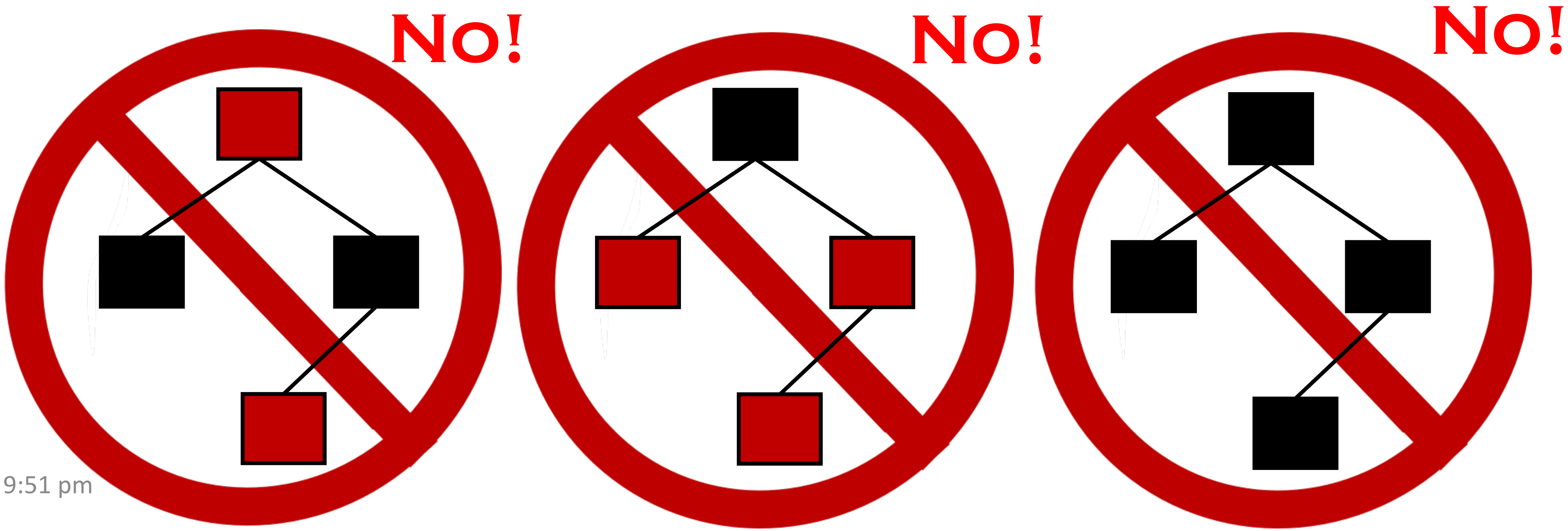
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



Which of these are red-black trees?
(NIL nodes not drawn)

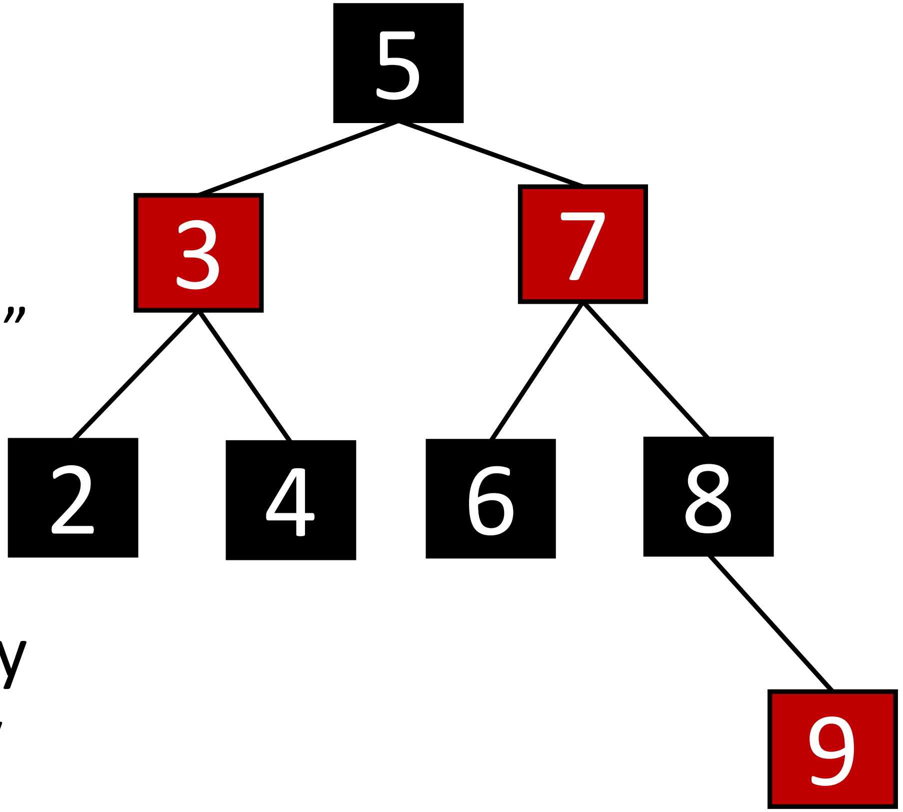


1 minute think
1 minute pair+share



Why **these** rules???????

- This is pretty balanced.
  - The **black nodes** are balanced
  - The **red nodes** are “spread out” so they don’t mess things up too much.



- We can maintain this property as we insert/delete nodes, by using rotations.

This is the really clever idea!

This **Red-Black** structure is a proxy for balance.

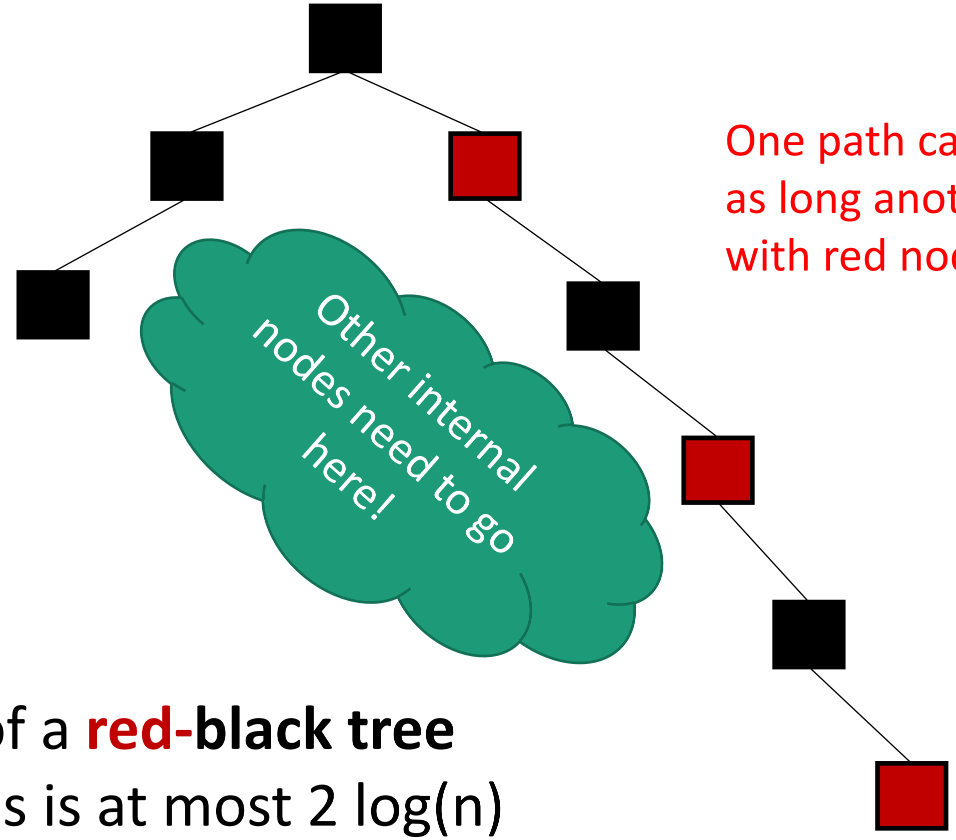
It’s just a smidge weaker than perfect balance, but we can actually maintain it!

Let’s build some intuition!

Lucky the lackadaisical lemur

This is “pretty balanced”

- To see why, intuitively, let’s try to build a Red-Black Tree that’s unbalanced.

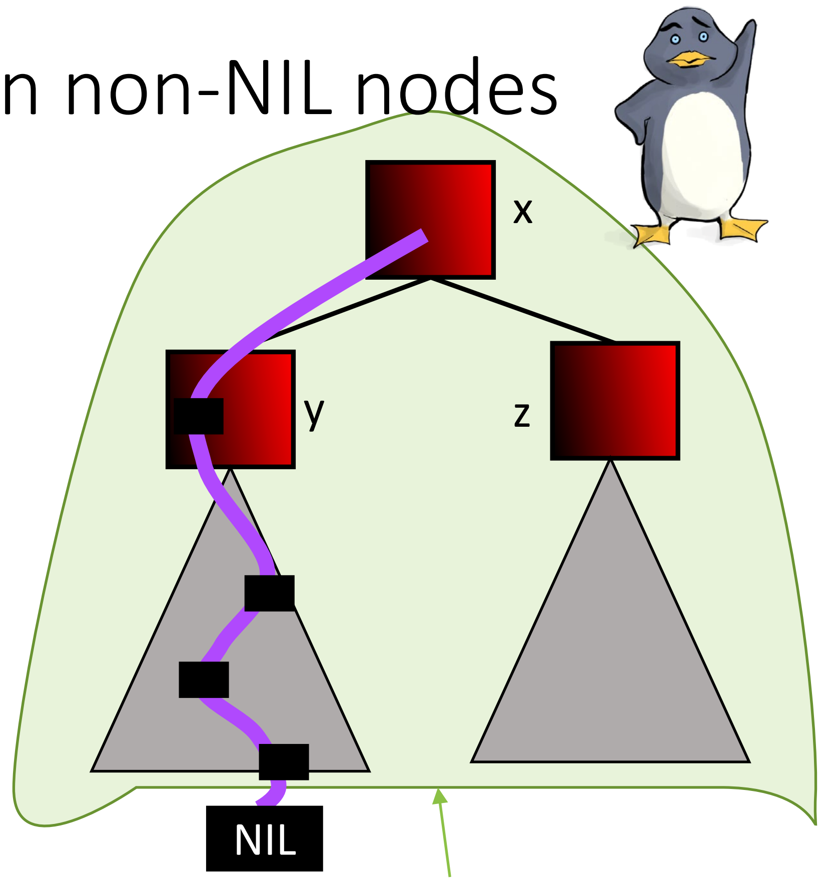


One path can be at most twice as long another if we pad it with red nodes.

Conjecture:
the height of a **red-black tree** with n nodes is at most 2 log(n)

The height of a RB-tree with n non-NIL nodes is at most $2\log(n+1)$

- Define b(x) to be the number of black nodes in any path from x to NIL.
  - (excluding x, including NIL).
- Claim:
  - There are at least $2^{b(x)} - 1$ non-NIL nodes in the subtree underneath x. (Including x).
- [Proof by induction – on board if time]



Claim: at least $2^{b(x)} - 1$ nodes in this WHOLE subtree (of any color).

Then:

$$n \geq 2^{b(root)} - 1 \quad \text{using the Claim}$$

$$\geq 2^{height/2} - 1 \quad \text{b(root) >= height/2 because of RBTree rules.}$$

Rearranging:

$$n + 1 \geq 2^{height/2} \Rightarrow height \leq 2\log(n+1)$$

This is great!

- SEARCH in an RBTree is immediately O(log(n)), since the depth of an RBTree is O(log(n)).

- What about INSERT/DELETE?
  - Turns out, you can INSERT and DELETE items from an RBTree in time O(log(n)), while *maintaining* the RBTree property.
  - That's why this is a good property!

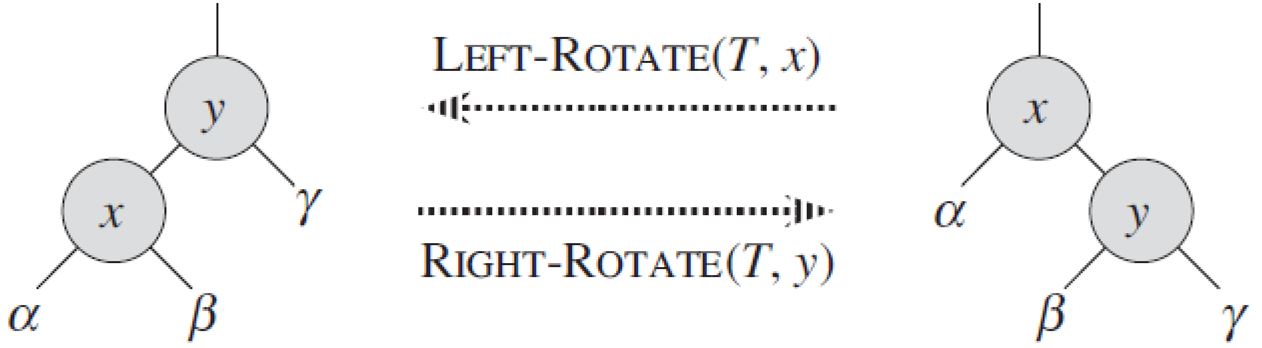
Rotation

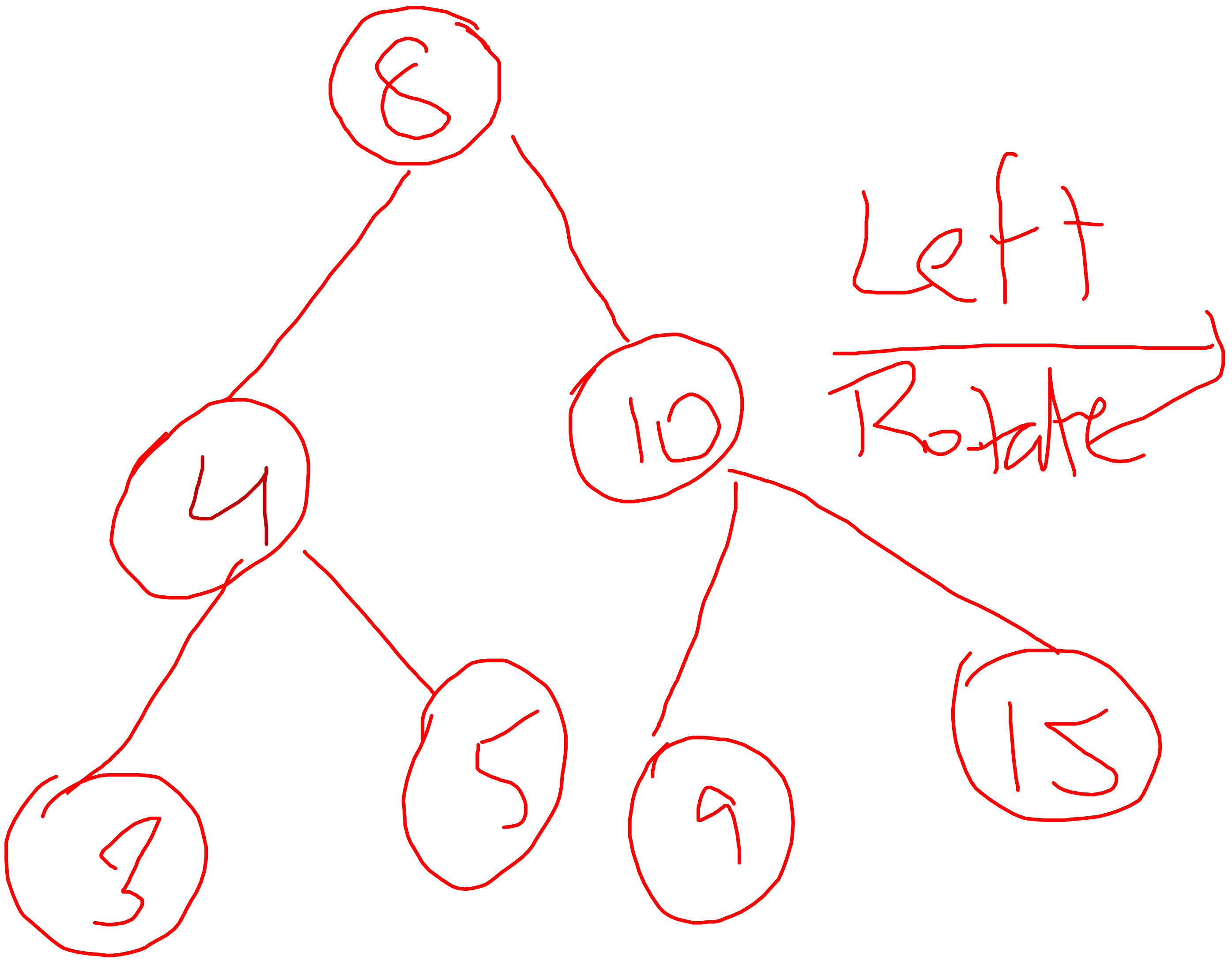
CLRS Chapter 13 (13.2)

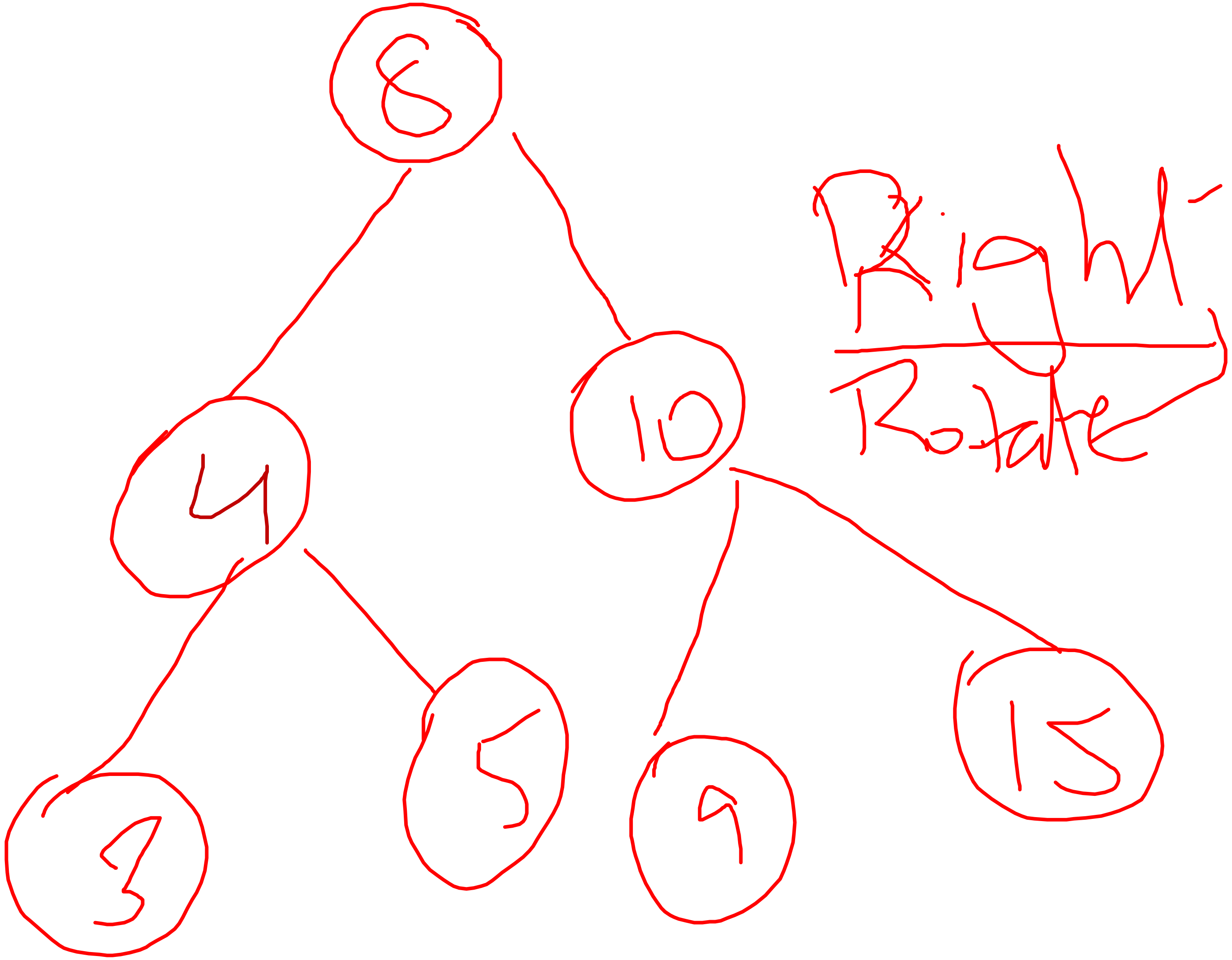
Rotations

LEFT-ROTATE(T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```



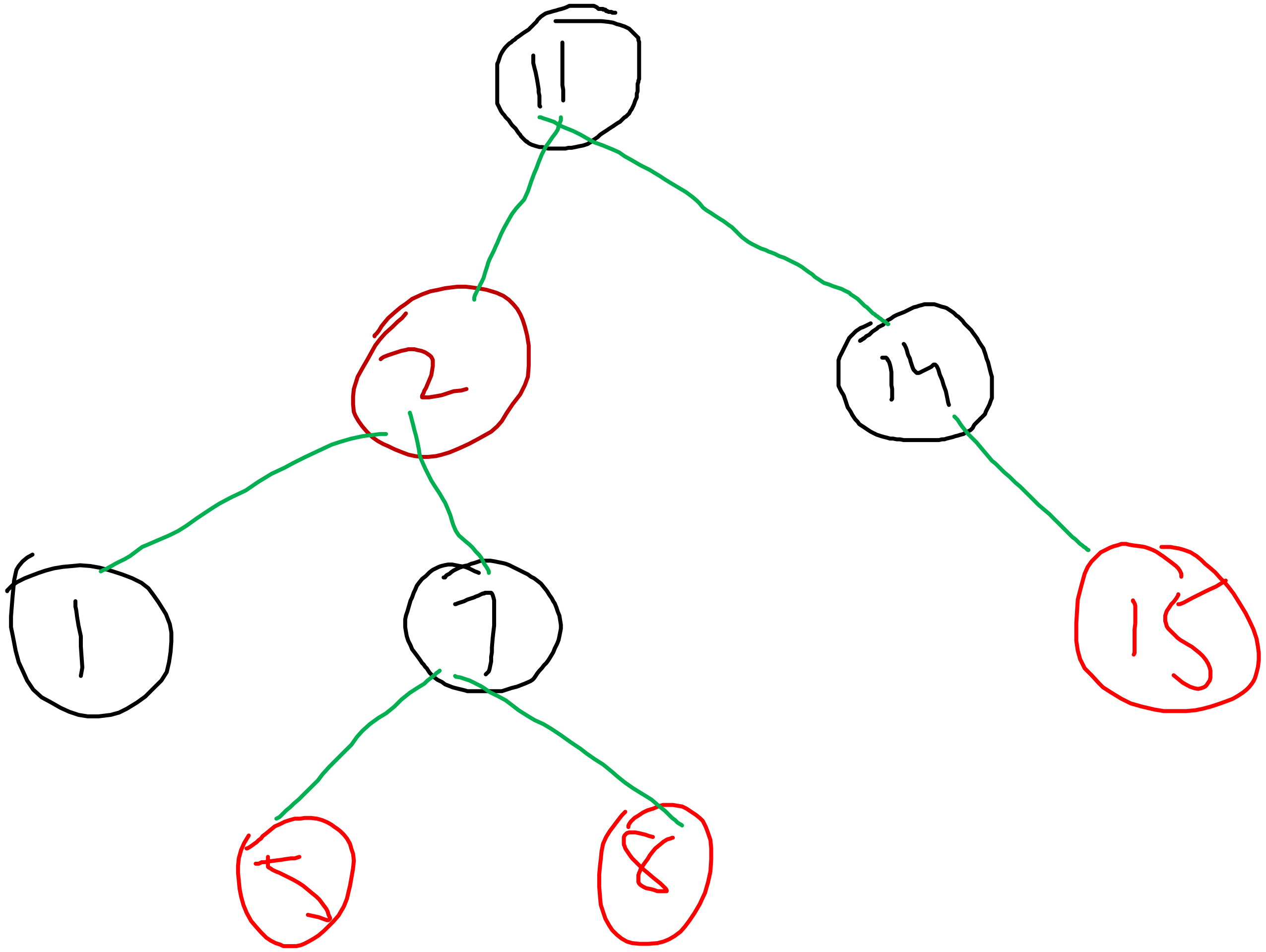

8
4
10
3
5
9
15
Left Rotate


8
4
10
3
5
9
15
Right Rotate

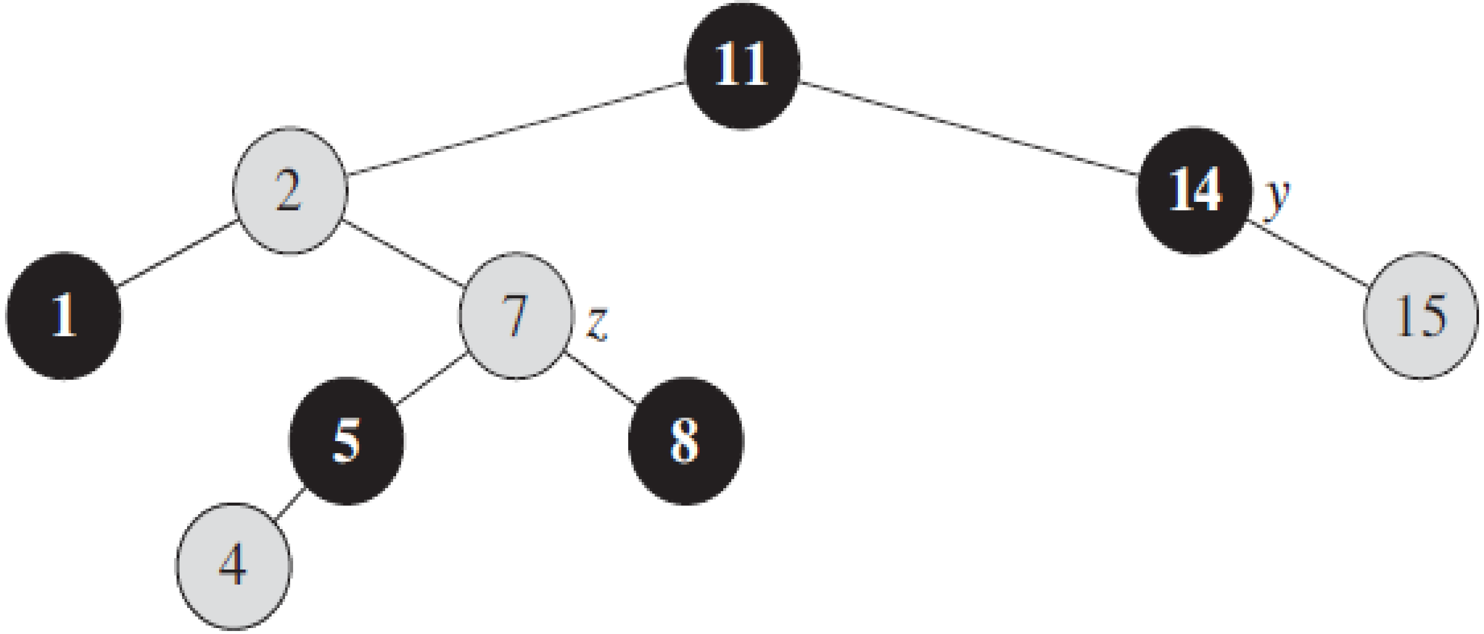
Tree Insertion

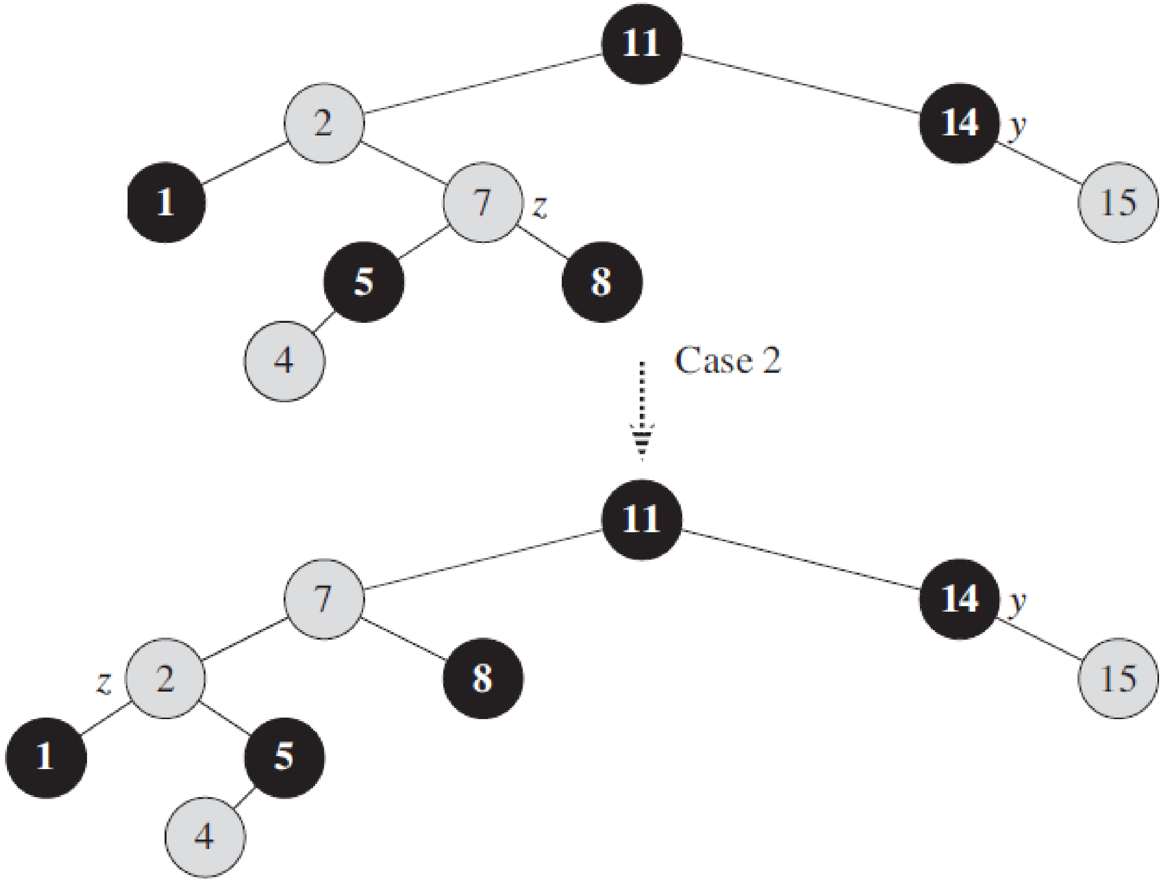
CLRS Chapter 13 (13.3)

Case 1: Z's uncle is red

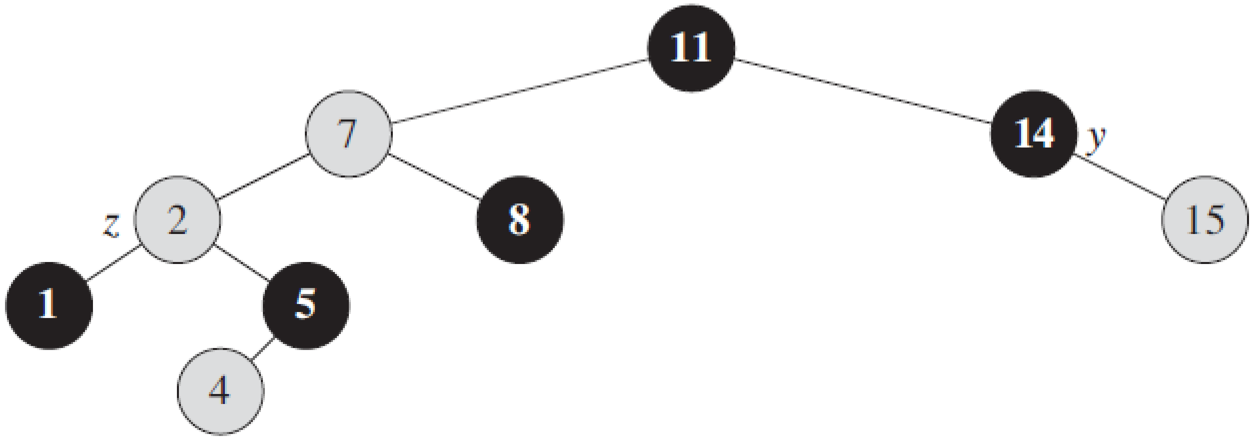


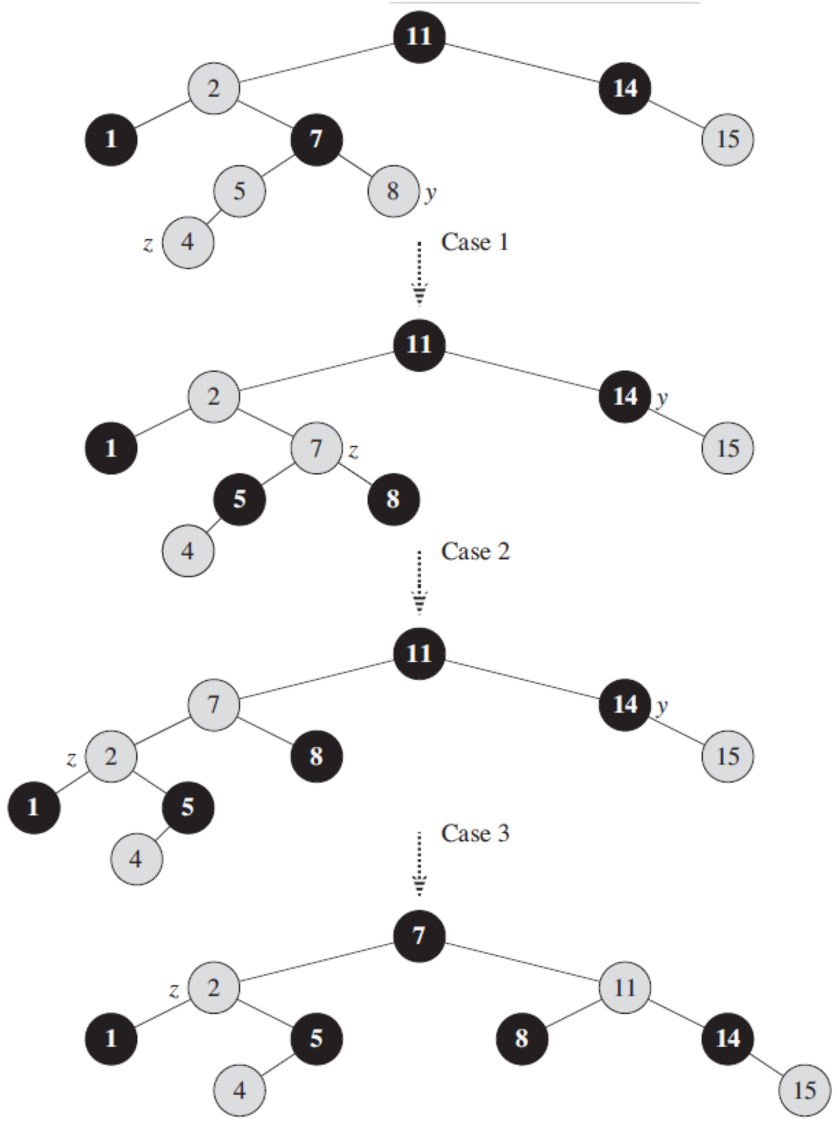
Case 2: Z’s uncle is Black is z is right child




11
2
14
y
1
7
z
15
5
8
4
Case 2
11
7
14
y
z
2
8
15
1
5
4

Case 3: Z's Uncle is Black and Z is left child




11
2
14
1
7
15
5
8
y
z
4
Case 1
11
2
14
y
1
7
z
15
5
8
4
Case 2
11
7
14
y
z
2
8
15
1
5
4
Case 3
7
z
2
11
1
5
8
14
4
15

Tree Insert

RB-INSERT(T, z)

```
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

RB-INSERT-FIXUP(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                  // case 1
            y.color = BLACK                    // case 1
            z.p.p.color = RED                  // case 1
            z = z.p.p                          // case 1
        else if z == z.p.right
                z = z.p                        // case 2
                LEFT-ROTATE(T, z)              // case 2
            z.p.color = BLACK                  // case 3
            z.p.p.color = RED                  // case 3
            RIGHT-ROTATE(T, z.p.p)             // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Deletion

CLRS Chapter 13 (13.4)

Deleting from a Red-Black tree

Fun exercise!

Ollie the over-achieving ostrich

What have we learned?

- Red-Black Trees always have height at most 2log(n+1).
- As with general Binary Search Trees, all operations are O(height)
- So all operations with RBTrees are O(log(n)).

Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	O(log(n))	O(n)	O(log(n))
Delete	O(n)	O(n)	O(log(n))
Insert	O(n)	O(1)	O(log(n))

Thank You