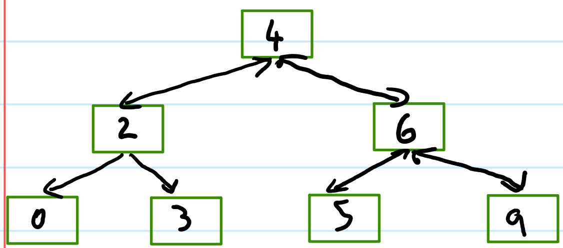
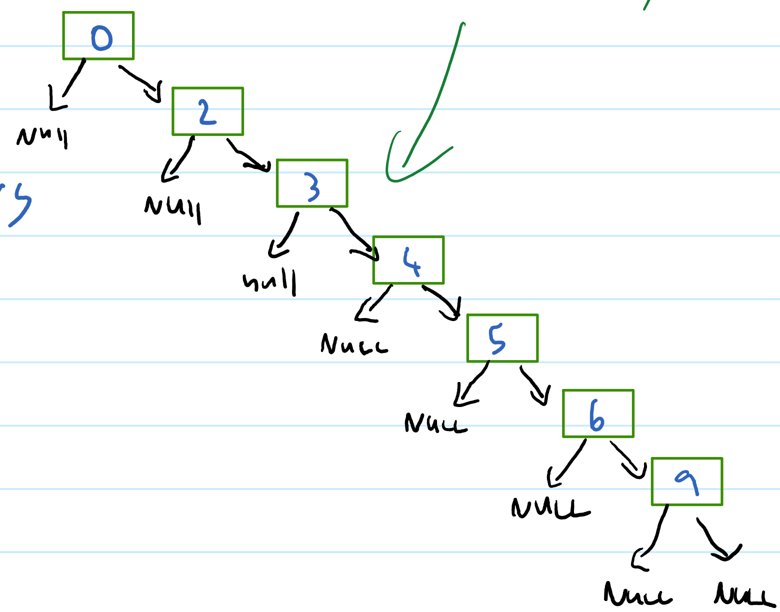
(01) Recall: importance of balance

$O(N)$



vs



$O(\underline{\log N})$

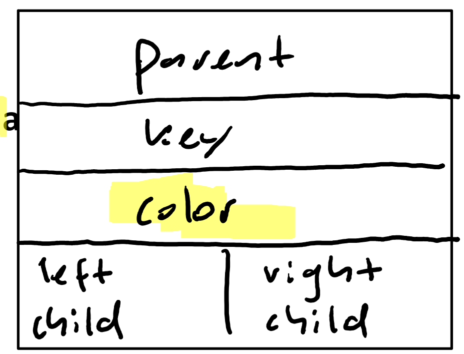
Different Approaches:

- Randomization: maybe data already has a fairly uniform distribution
  - Otherwise maybe you can randomize the data yourself as a pre-processing step
- Amortization: a balancing method that gets called at some predefined time
- Dynamic self-balancing: every time a node is added the tree will *check* whether it violated certain rules. If rules are violated will trigger rebalancing mechanism
  - Examples: AVL, Red-black trees

(02) The red-black tree

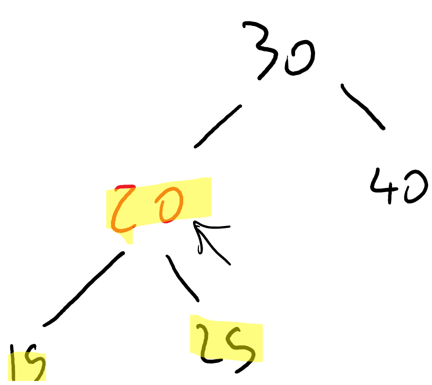
Red-Black Tree:

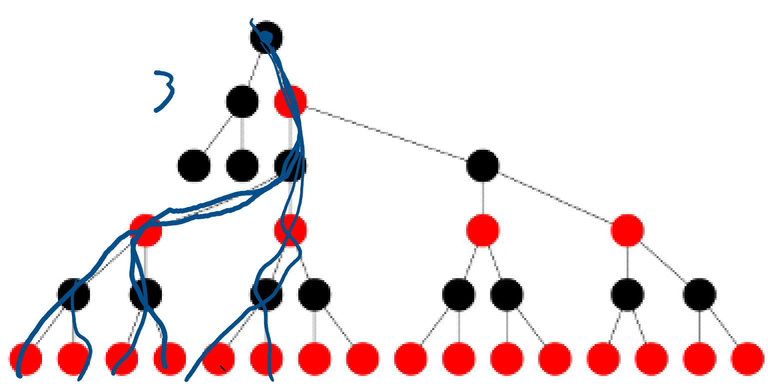
- The node definition for a BST gets updated with a new Boolean data member.
- Now, each node can be described as always having one of two states
- The colors are used to enoforce a strict set of rules on the way the nodes are arranged with respect to each other.
- When a rule is violated, it has to repaired.
- The rules *collectively* limit how unbalanced the tree can ever become.
    - Result: the longest path to any leaf node is at most twice as long as the shortest: 2log(n) -> O(log(n))

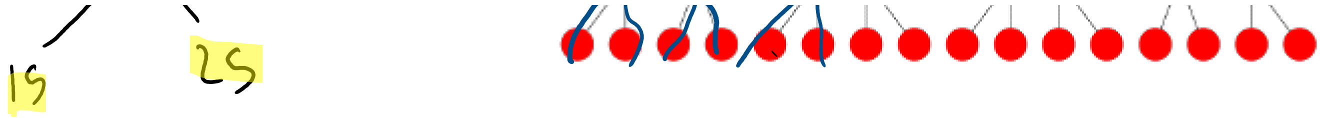


RB Rules:

1. **A node is either red or black. (a node can change colors as a part of rebalancing)**
2. **Root node is black.**
3. **Every leaf node is black, empty, and null**
4. **If node is red, bot its children must be black**
5. **For every node in the tree, all paths to a descendant leaf node must pass through the same number of black nodes.**



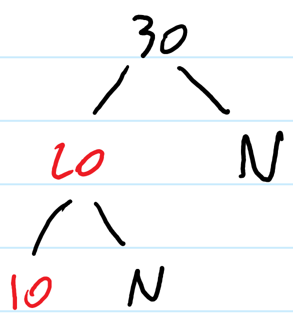



15
25

(03) RB Examples

1. A node is either red or black (color can change as a tree re-balances.)
2. Root node is black.
3. Every leaf node is black, empty, and null.
4. If a node is red, both its children must be black.
5. For every node in the tree, all paths to a descendant leaf node must pass through same number of black nodes.

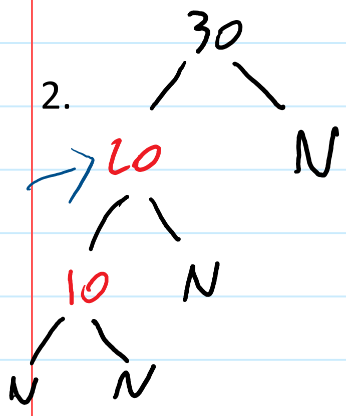
1.



r-b?

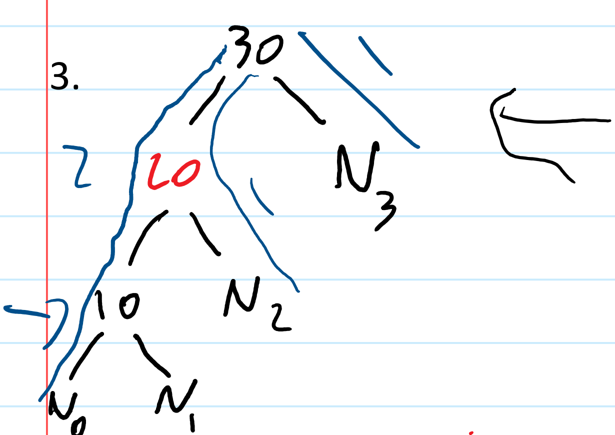
No: rule 3

2.

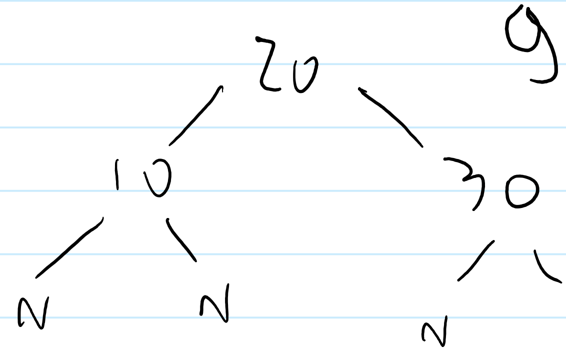


No: rule 4

3.



No: rule 5



good

(04) Rebalancing operations

For red-black trees, we define a set of special operations:

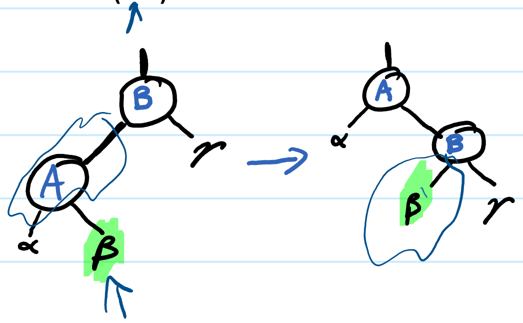
1) recolor a node
2) rotate - changes height of tree
    a. rotate right
    b. rotate left

Sometimes re-coloring will suffice to fix the tree. Other times, need to rotate + re-color.

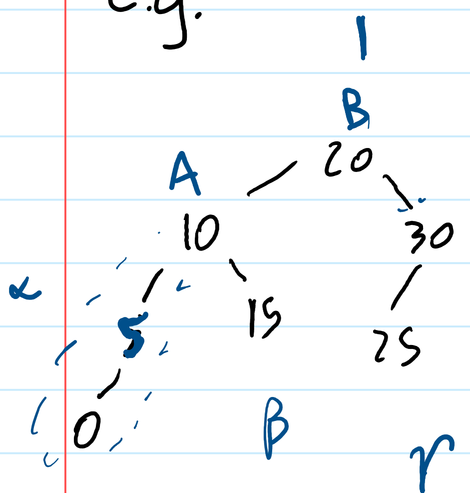
(05) Rotations

· Right rotate

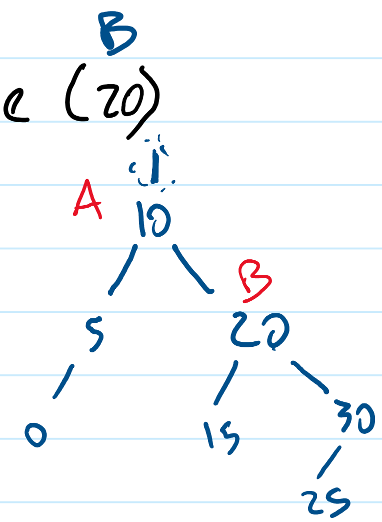
R-rotate(B)



e.g.

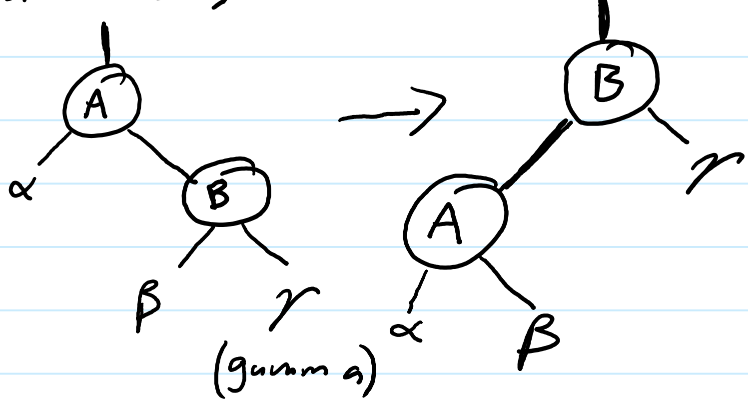


Right Rotate (20)



The analogous left rotate would revert the r-rotate operation.

L-rotate (A)



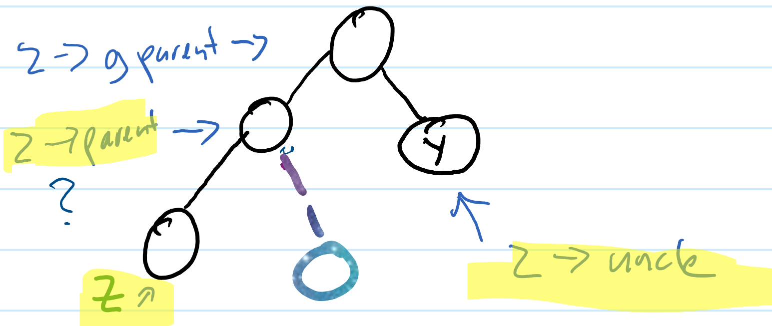
(06) RB Insert 1

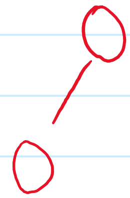
RB Rules:

1. A node is either red or black (color can change as a tree re-balances.)
2. Root node is black.
3. Every leaf node is black, empty, and null.
4. If a node is red, both its children must be black.
5. For every node in the tree, all paths to a descendant leaf node must pass through same number of black nodes.

Insert Steps:

1. Insert node just like you would into a bst.
    - Color the new node red
2. Check if parent node is red,
    if so a repair is needed. One of 6 possible scenarios:





6 Possible scenarios RB tree can encounter after an *insert*

z = new node

A. Parent of z node is LC
   1. Uncle of z node is red
   2. uncle of z node is black *and* z node is a right child
   3. uncle of z node is black *and* z node is a left child

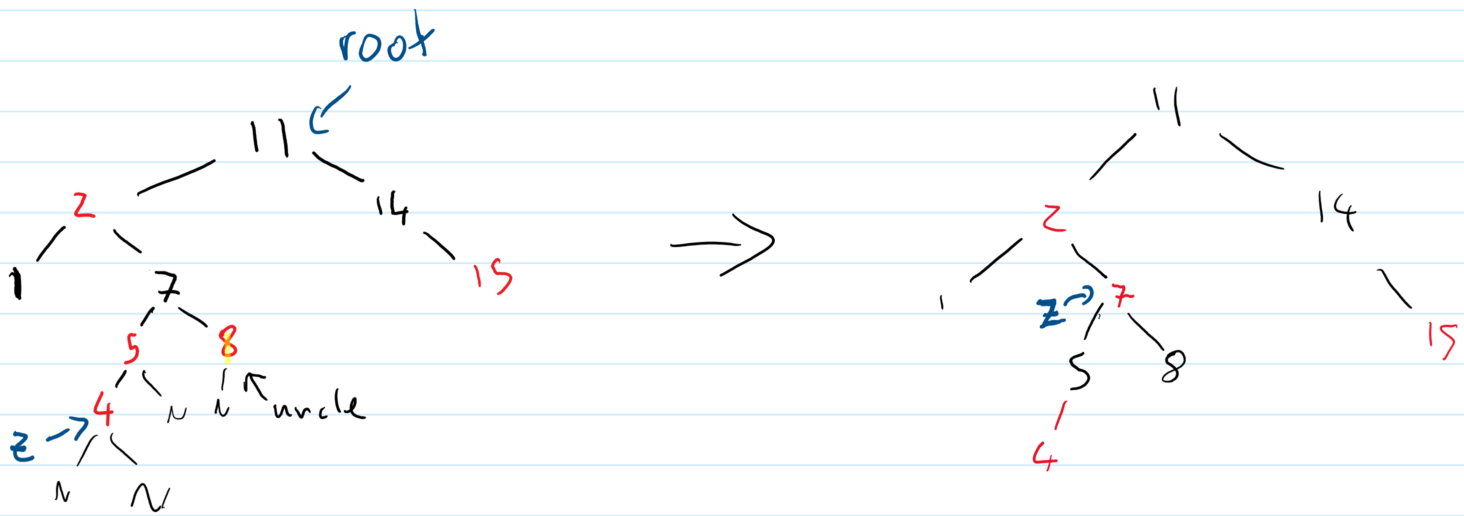
B. Parent of z node is RC
   same 3 scenarios, mirror solutions

(07) RB Insert 2

RB Rules:

1. A node is either red or black (color can change as a tree re-balances.)
2. Root node is black.
3. Every leaf node is black, empty, and null.
4. If a node is red, both its children must be black.
5. For every node in the tree, all paths to a descendant leaf node must pass through same number of black nodes.

Example: given the following tree, `insert(4)` is issued. Let's walk through the checks and rebalanding steps.



how to check uncle node's color in code?

z->parent->parent->rightChild->color == black

case 1: z's uncle node is red

case 1 resolution:

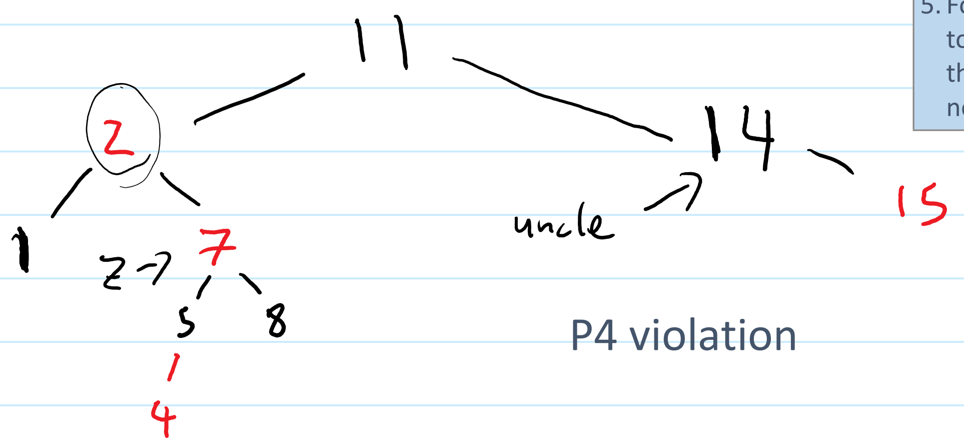
1. color parent node black (z->parent->color = black)
2. color uncle node black (z->parent->parent->rightChild = black)
3. color grand parent red
4. move z pointer up to grand parent

(08) RB Insert 3

RB Rules:

1. A node is either red or black (color can change as a tree re-balances.)
2. Root node is black.
3. Every leaf node is black, empty, and null.
4. If a node is red, both its children must be black.
5. For every node in the tree, all paths to a descendant leaf node must pass through same number of black nodes.

Continued example: **`insert(4)`**



P4 violation

Is z a right child?
z == z->parent->right

case 2: z's uncle node is black and new node is RC

case 2 resolution:

1. **set z to point to its parent**
   z = z->parent
2. **left-rotate on z**

(09) RB Insert 4

RB Rules:

1. A node is either red or black (color can change as a tree re-balances.)
2. Root node is black.
3. Every leaf node is black, empty, and null.
4. If a node is red, both its children must be black.
5. For every node in the tree, all paths to a descendant leaf node must pass through same number of black nodes.

Continued example: **`insert(4)`**

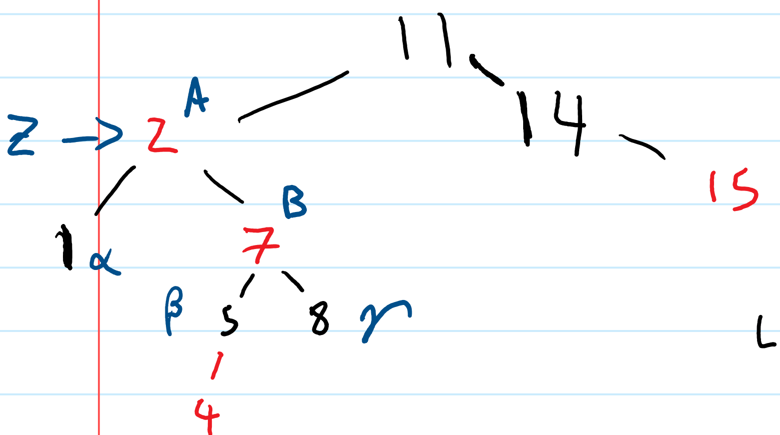
case 2 resolution:

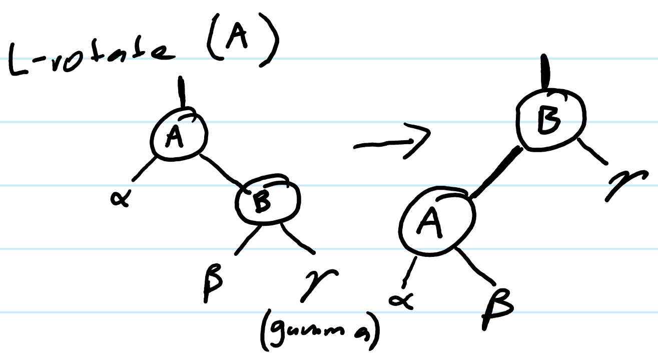
1. set z to point to its parent

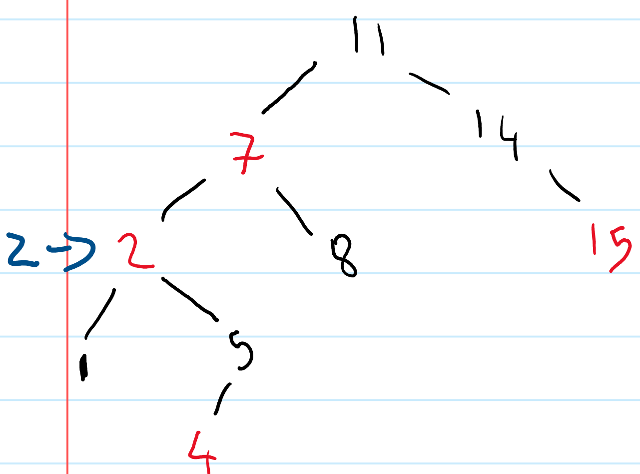
   z=z->parent

2. left rotate on z

   leftRotate(z)







(10) RB Insert 5

Continued example: **`insert(4)`**

case 3: uncle node is black *and* z is a left child

case 3 resolution:

1. color the z parent node black
2. color the z grandparent node red
3. right-rotate on grandparent

RB Rules:

1. A node is either red or black (color can change as a tree re-balances.)
2. Root node is black.
3. Every leaf node is black, empty, and null.
4. If a node is red, both its children must be black.
5. For every node in the tree, all paths to a descendant leaf node must pass through same number of black nodes.