RED-BLACK TREES

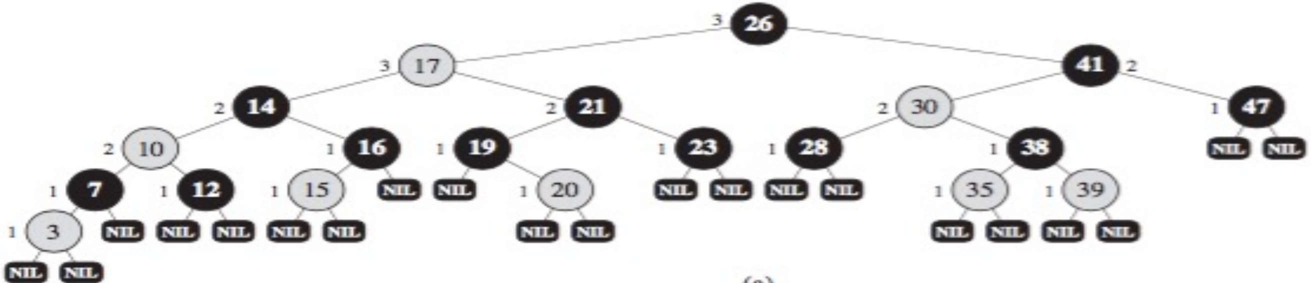
CS340

Red-Black Trees

- Red Black Trees are a kind of Binary Search Tree that are **balanced**
  - guarantee of logarithmic height O(lg n)
  - Non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in O(lg n) time on red-black trees.
  - INSERT and DELETE are tricky! Must maintain red-black tree properties.
- R-B Trees have an extra bit of information at each node
  - the color: Red? or Black?

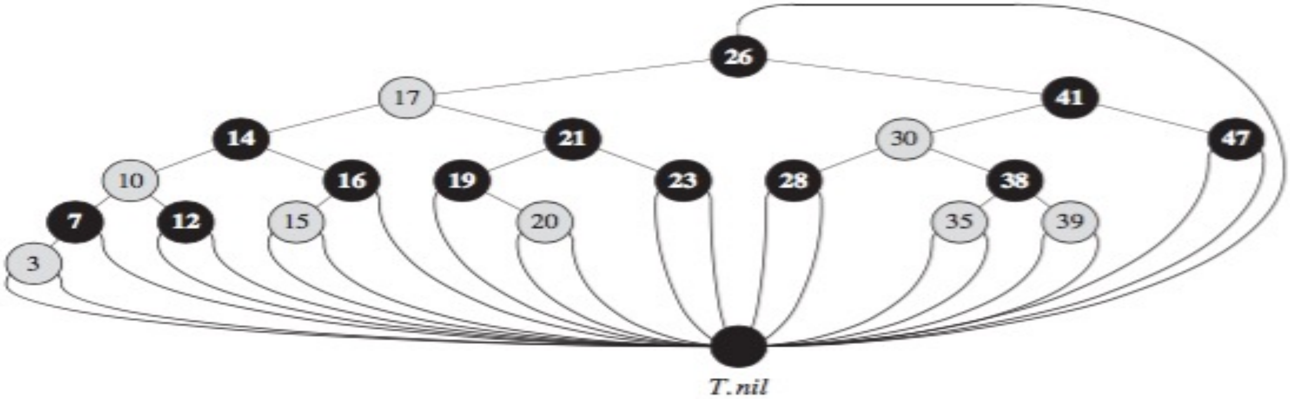
Red-Black Tree Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf is NIL and black.
4. If a node is red then both its children are black.
5. For each node, all simple paths to its descendant leaves have the same number of black nodes.
    - This number defines the black height of a node x, bh(x).
    - By the previous property, $bh(x) \geq h(x)/2$



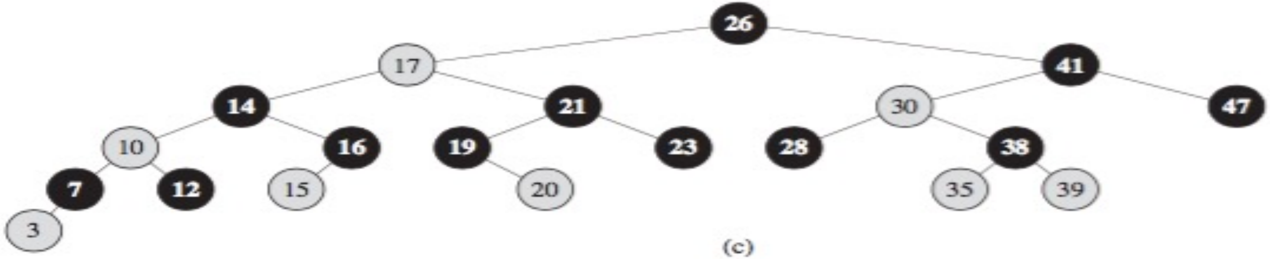
26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
NIL

(a)



26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3
T.nil

(b)

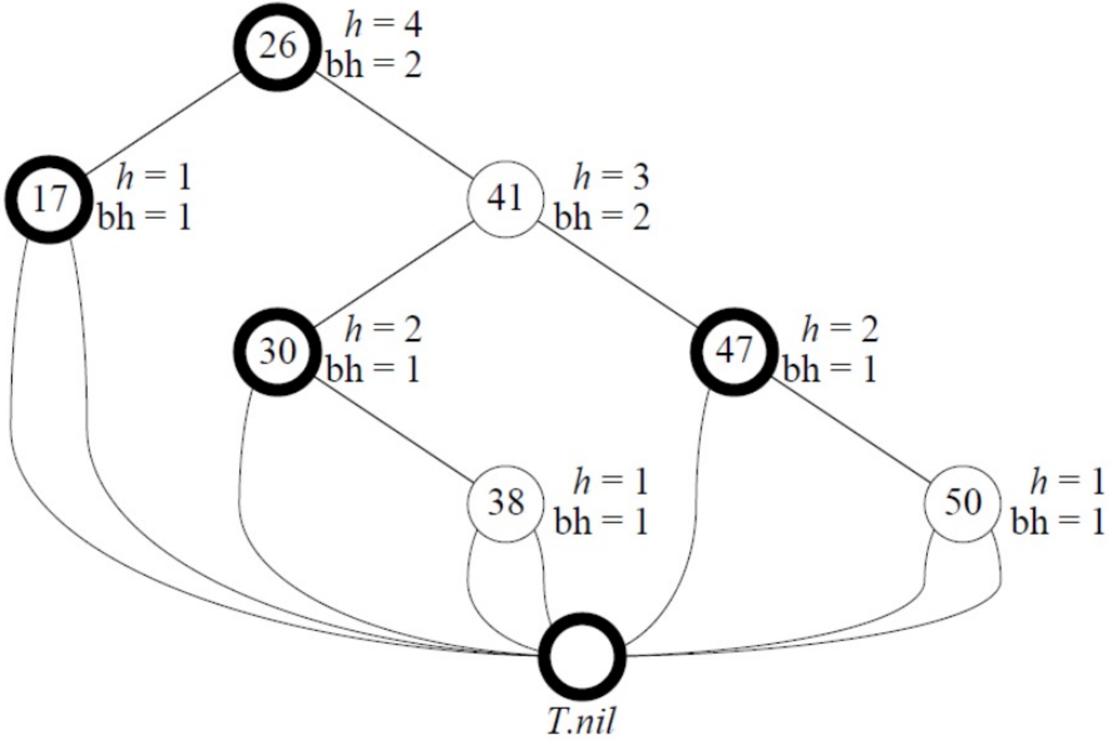


26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3

(c)

Height and Black Height

- Height: number of edges in longest path to a leaf
- Black Height: number of black nodes (including T.nil) on the path from x to leaf, not counting x (see property 5)



Red-black tree properties

Any node with height h has black-height, bh(x) ≥ h/2.

Proof

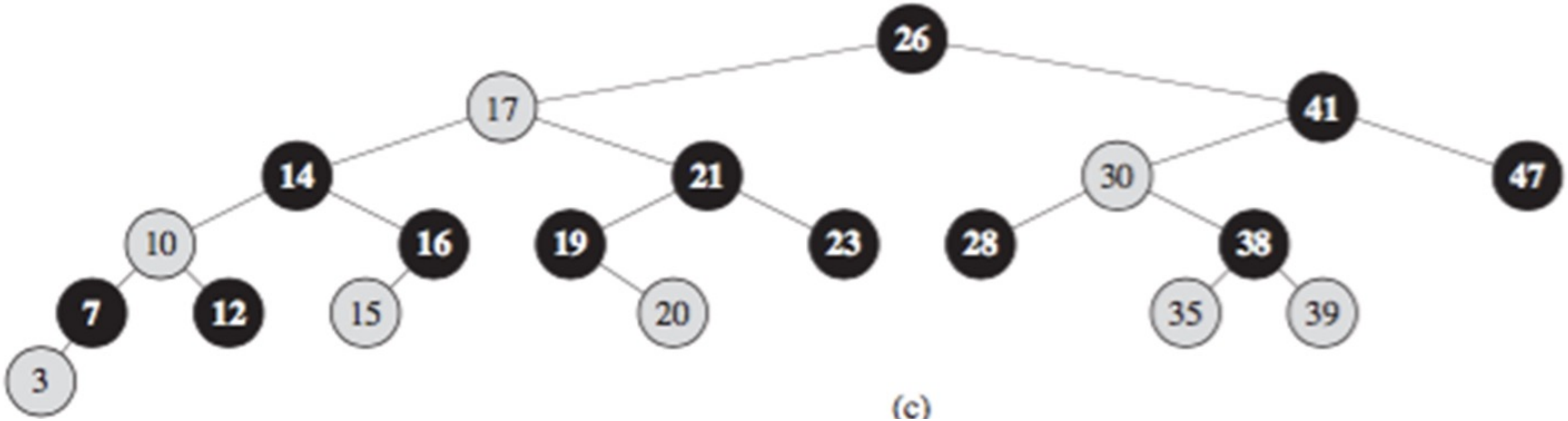
Property 4: If a node is red then both its children are black.

This means that each red node must be followed by a black node. The reverse is not necessarily true.

Therefore, ≤ h/2 nodes on the path from the node to a leaf are red. Hence ≥ h/2 nodes are black.

Interview Questions

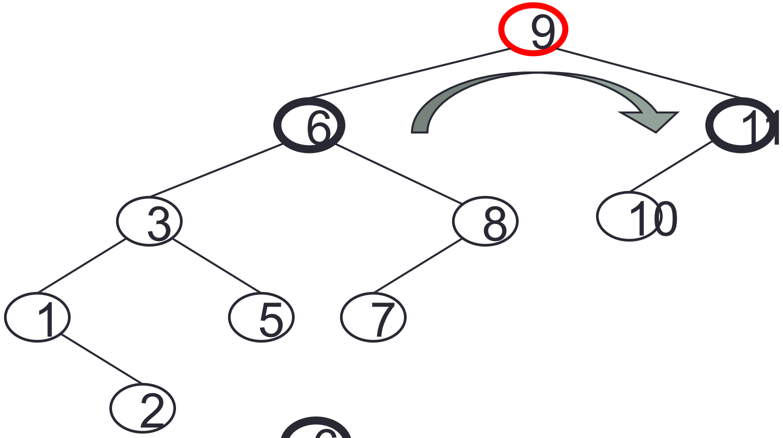
- Draw the red-black tree that results after TREE-INSERT is called on the tree below with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?



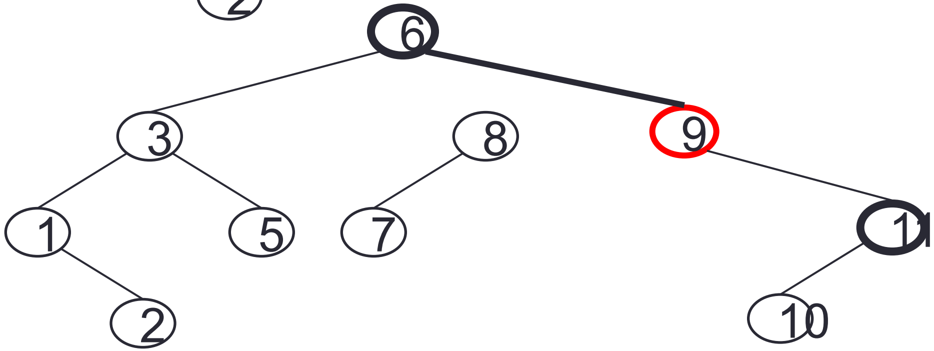
(c)

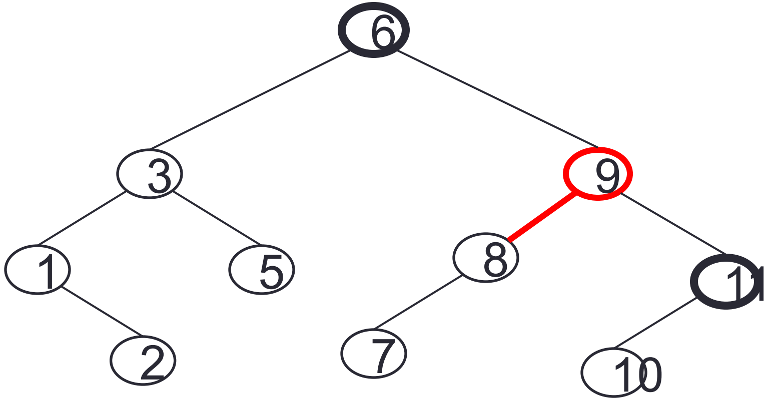
THE BIG PICTURE

Rotations of trees



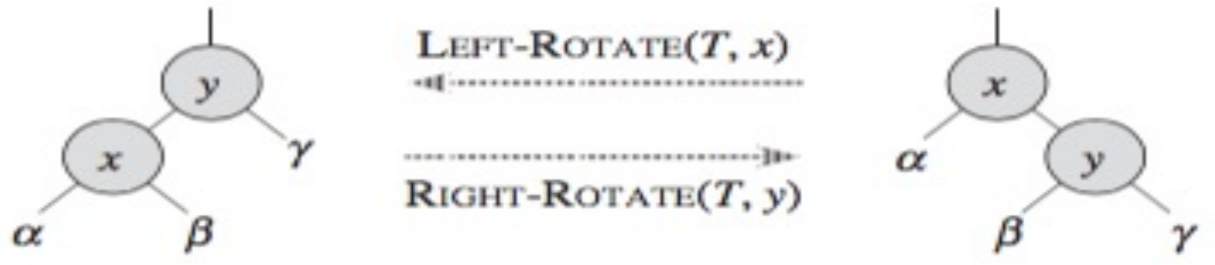
- Rotation to change shape
- One side gets taller, the other shrinks





Rotation of Trees

- Maintaining Red-Black tree properties upon insertion involves **rotations**

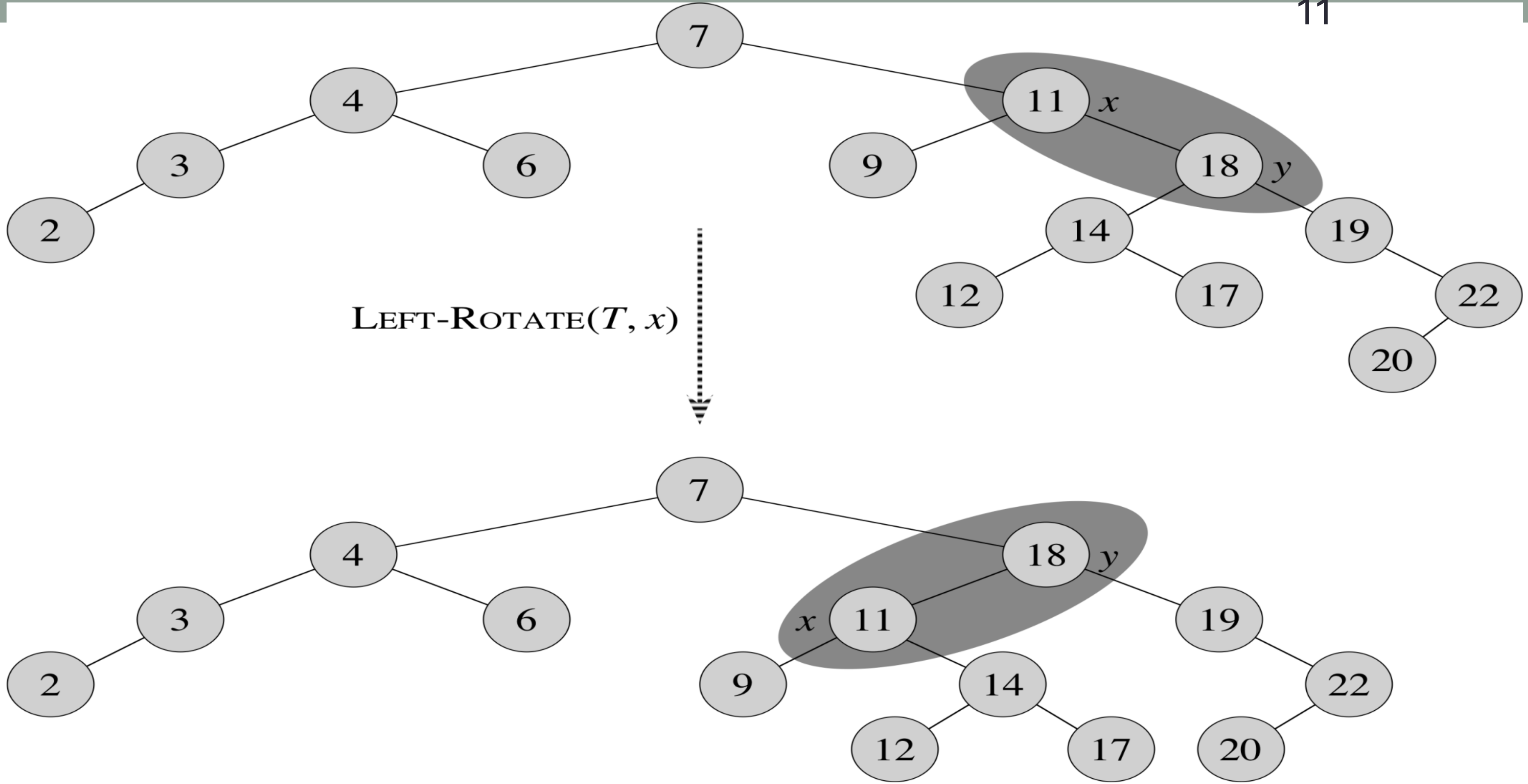


Left-Rotate:

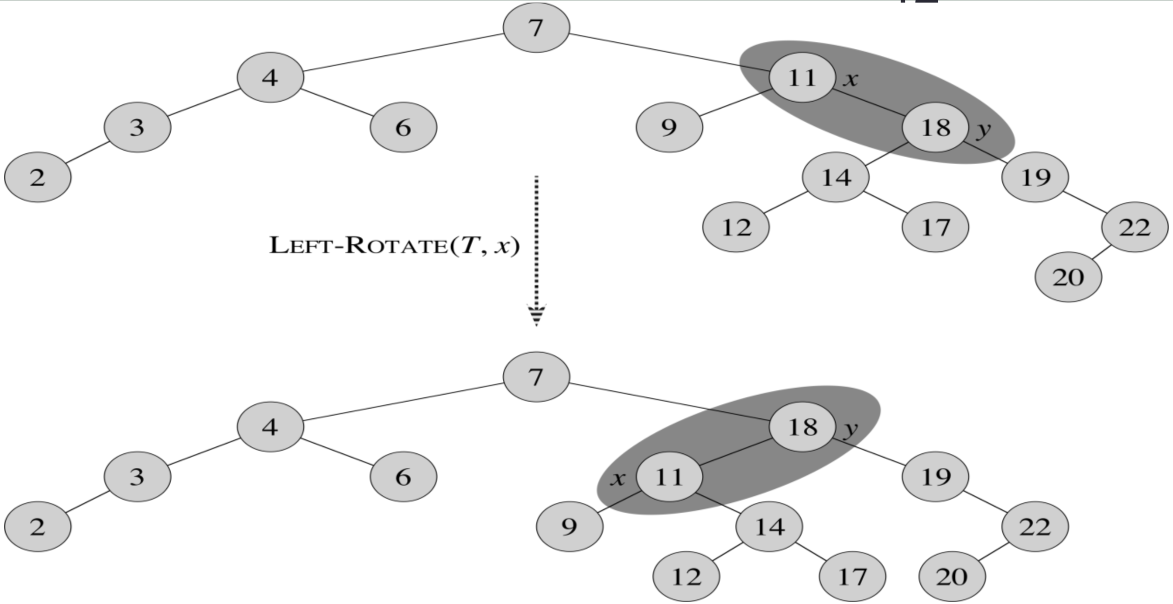
- right child is not T.NIL.
- y is new root.
- x is y's left child.
- y's left child is x's right child.

Rotation maintains inorder ordering of keys.

What happens to the relative subtree heights?


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

Rotations



- Before rotation: keys of x’s left subtree ≤ 11≤ keys of y’s left subtree ≤ 18 ≤ keys of y’s right subtree.
- Rotation makes y’s left subtree into x’s right subtree.
- After rotation: keys of x’s left subtree ≤11 ≤keys of x’s right subtree ≤18≤ keys of y’s right subtree.
- Rotation is done by changing pointers. Time complexity = ??

Left-Rotate

```
LEFT-ROTATE(T, x)
y = x.right                    // set y
x.right = y.left               // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                      // link x's parent to y
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                     // put x on y's left
x.p = y
```

Time complexity?

Insertion and deletion

- SEARCH, MIN, MAX, SUCCESSOR, PREDECESSOR are the same as with binary search tree.

- If we **insert**, what color to make the new node?
  - Red? Might violate property 4. (if node is red, both children are black)
  - Black? Might violate property 5 (all paths to descendants have same number of black nodes)

- If we **delete**, what color was the node that was removed?
  - Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row.
  - Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2 (root is black), if the removed node was the root and its child—which becomes the new root—was red.

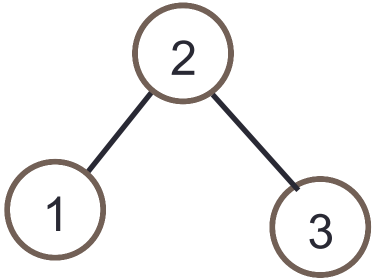
R-B Insertion

- First, insert the new value as a **red** leaf into the R-B tree just like a BST.
  - RB-Insert(T,z)
- This may cause a red-red violation or a red root violation, thus needing to be fixed.
  - RB-Insert-Fixup(T,z)

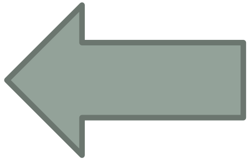
THE LITTLE PICTURE

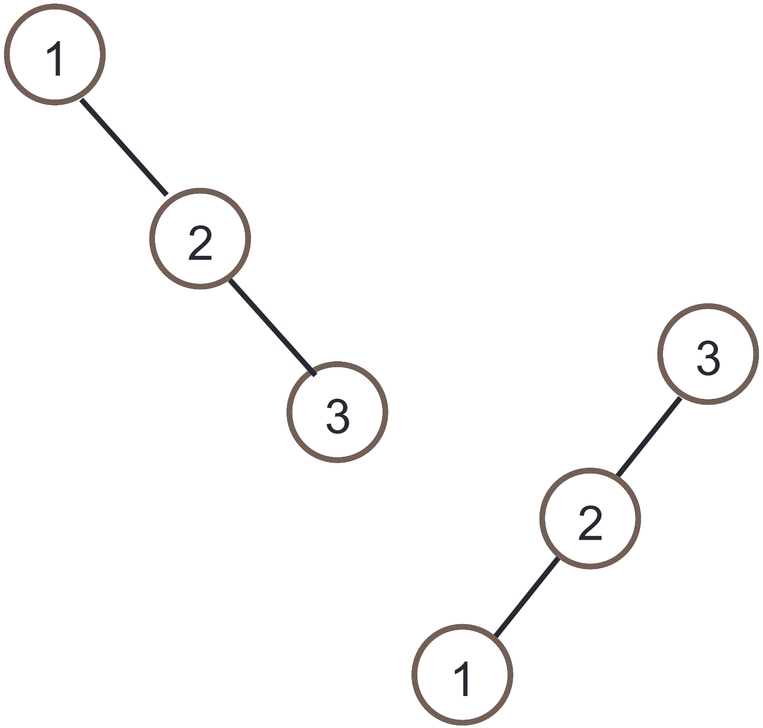
The idea of fixup

- You start with one of these



You want this



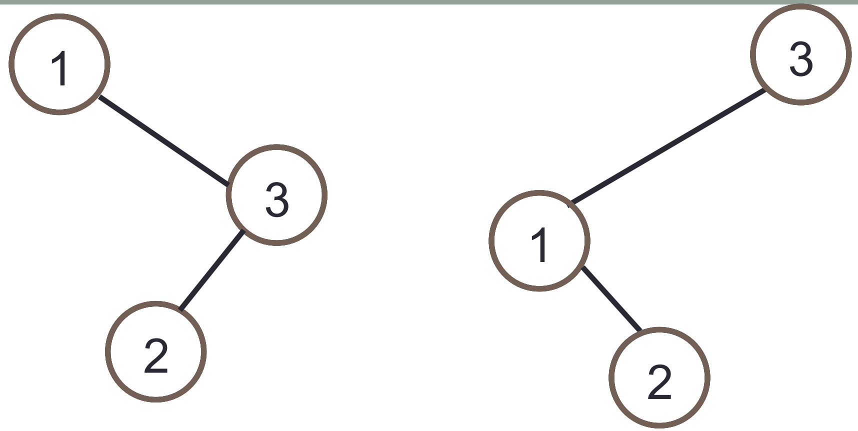


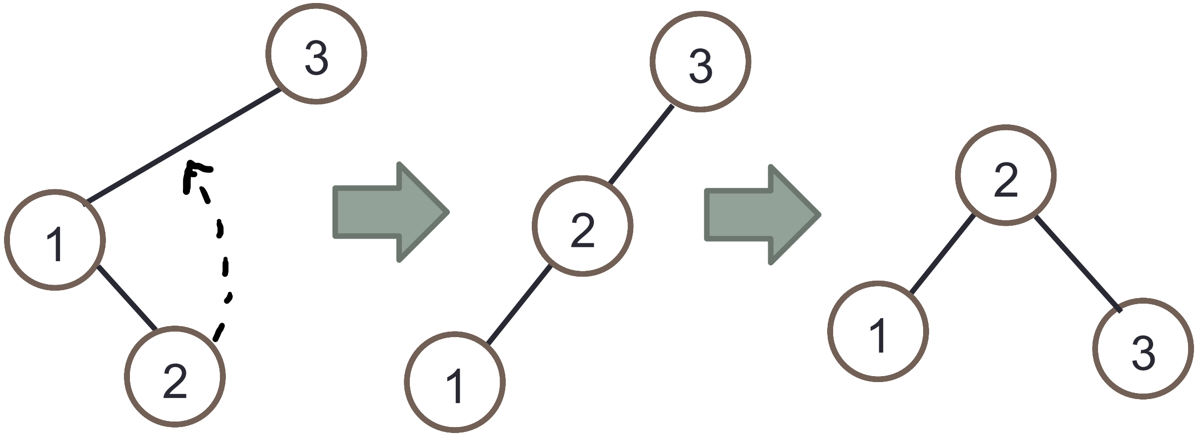
This is CASE 3

Rotate these

The idea of fixup

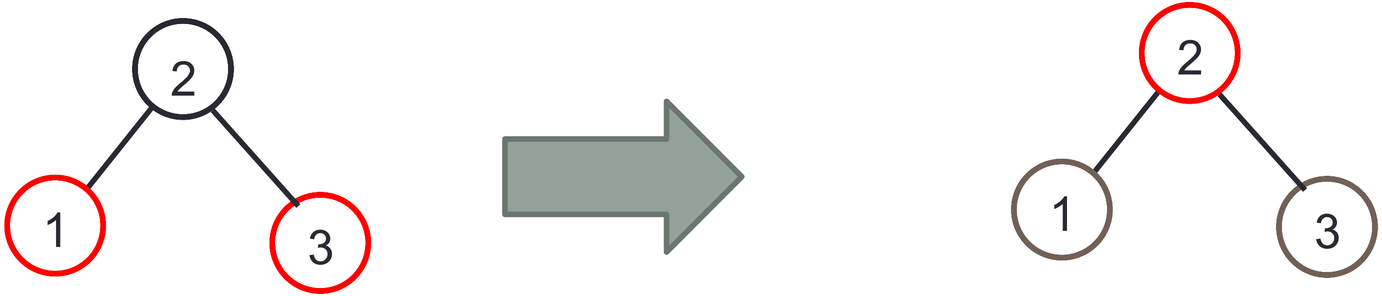
- You start with one of these:
- Remove the Zig-Zag
- Then rotate
- Call this Case 2





The idea of fixup (call this case 1)

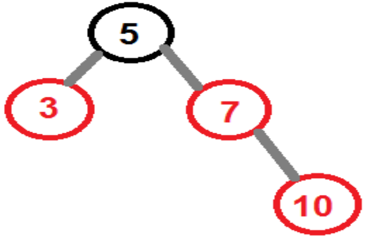
- You start with this



You can change to this:
It does not change the black height.
The next node can be added without further adjustment
You might still have a red-red violation (you have moved the problem up the tree).

Look for the red uncle!

Say 10 is inserted



What case is this?

What is the resulting tree
After the fix-up procedure?

```
RB-INSERT(T, z)
1  y = T.nil
2  x = T.root
3  while x ≠ T.nil
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == T.nil
10     T.root = z
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
14 z.left = T.nil
15 z.right = T.nil
16 z.color = RED
17 RB-INSERT-FIXUP(T, z)
```

Lines 3-7 search for the correct leaf position.

Note that **nil** is not the same as NULL.

Remember that the newly inserted node z is first colored red.

What are 4 differences between Tree-Insert and RB-Insert?

Unfortunately, the Fixup is more complicated!

Note that y is the UNCLE of z.

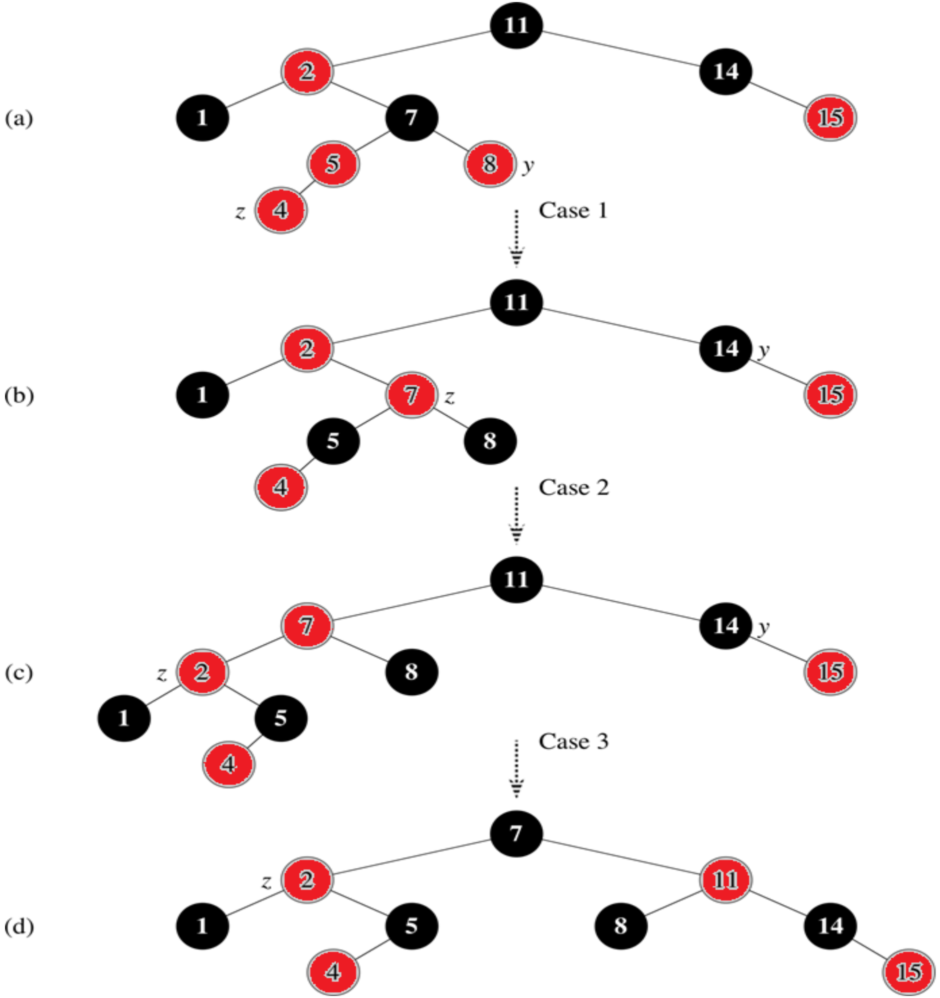
```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED                           if red uncle
            z.p.color = BLACK                       color parent black              // case 1
            y.color = BLACK                         color uncle black               // case 1
            z.p.p.color = RED                       color grandparent red           // case 1
            z = z.p.p                               move problem to gp              // case 1
        else if z == z.p.right                      if right child with black uncle
                z = z.p                             move problem up to parent       // case 2
                LEFT-ROTATE(T, z)                   rotate on parent node           // case 2
            z.p.color = BLACK                       z is a left child with black uncle // case 3
            z.p.p.color = RED                       change parent & gp colors       // case 3
            RIGHT-ROTATE(T, z.p.p)                  rotate right                    // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

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

```
if (uncle.color == red)
{
  # Handle case
}
else
{
    if (z == z.p.right)
    {
         # Handle case 2
    }
    # Handle case 3
}
```

Three Cases, summary

- Case 1: Red uncle. Recolor parent and grandparent. Moves problem to grandparent
- Case 2: Black uncle and zig-zagging. Move problem to parent. Rotate with new parent to remove zig-zag.
- Case 3: Black uncle without zig-zag. Recolor parent and grandparent. Rotate on parent-grandparent.

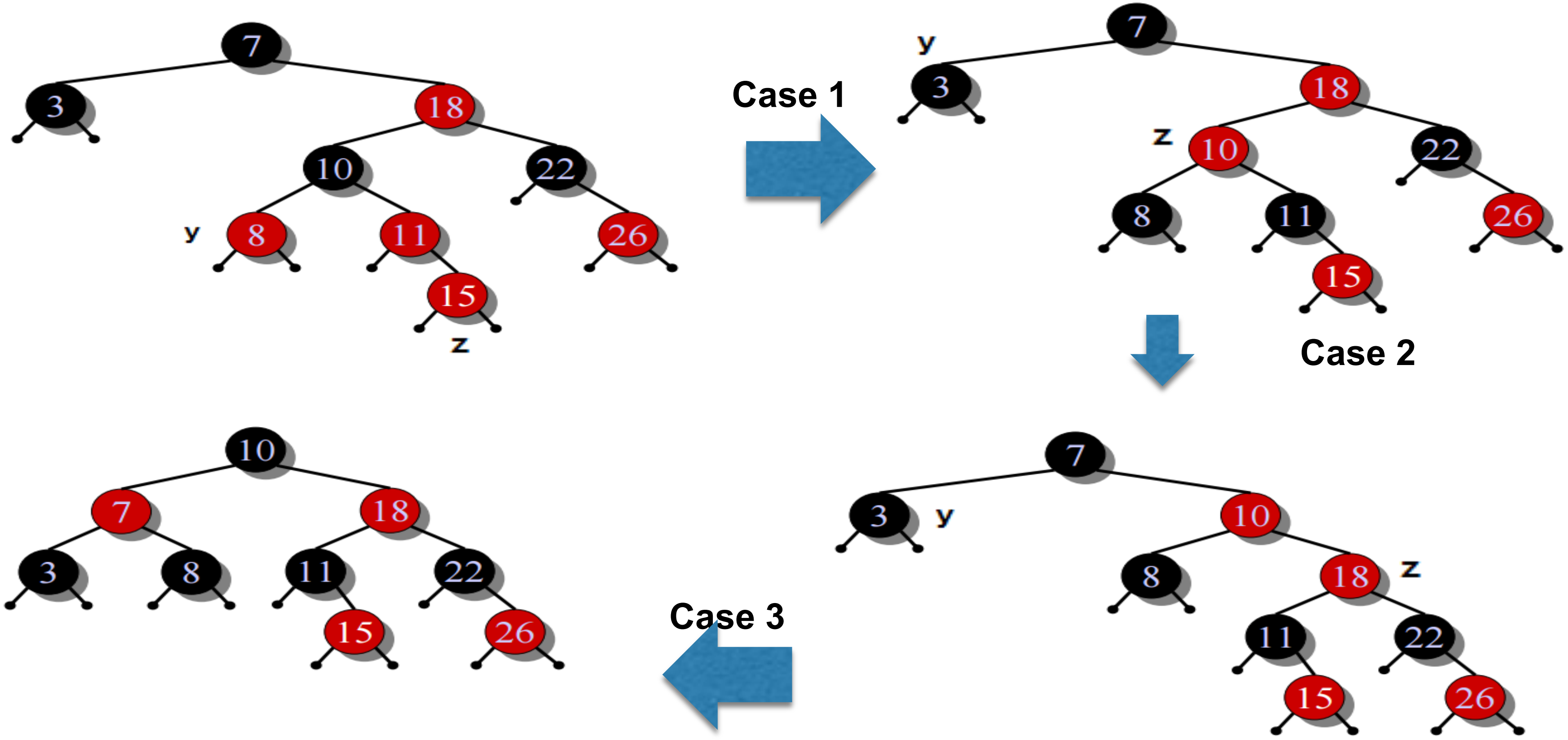


Case 1: If red uncle,
Re-color above generations.
Also, move problem to gp.
Note that y is z's uncle

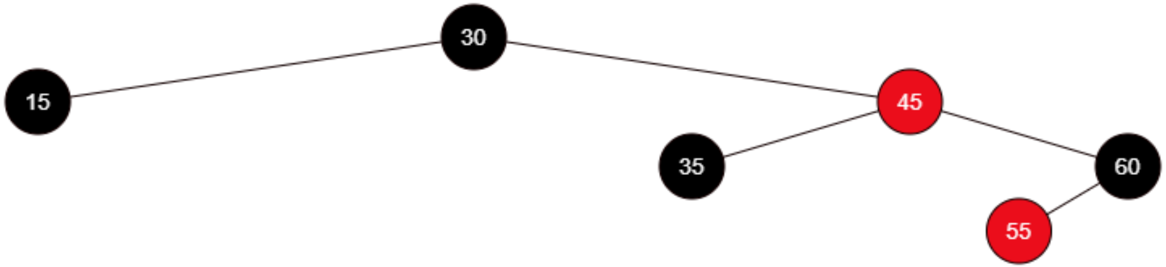
Case 2: Black uncle and
Zig-Zagging
(z is a right child
and z's parent is a left child):
Move problem to parent.
Rotate on parent to remove
zig-zag

Case 3: Black uncle
Without zig-zag
(z is a left child
and z's parent is a left child):
Recolor parent to black and gp to red.
Rotate on parent-gp to balance
Terminal Case!

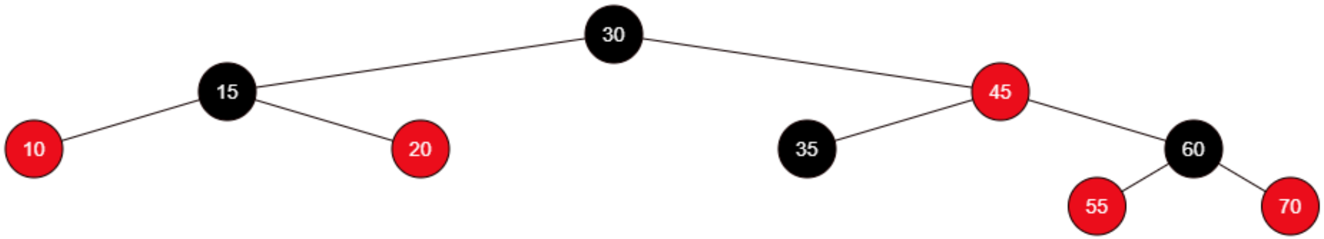
Another Example: Insert 15



Insert 50



Insert 65



Interview Questions

- Why is a new node set to red and not black?
- Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

41, 38, 31, 12, 19, 8