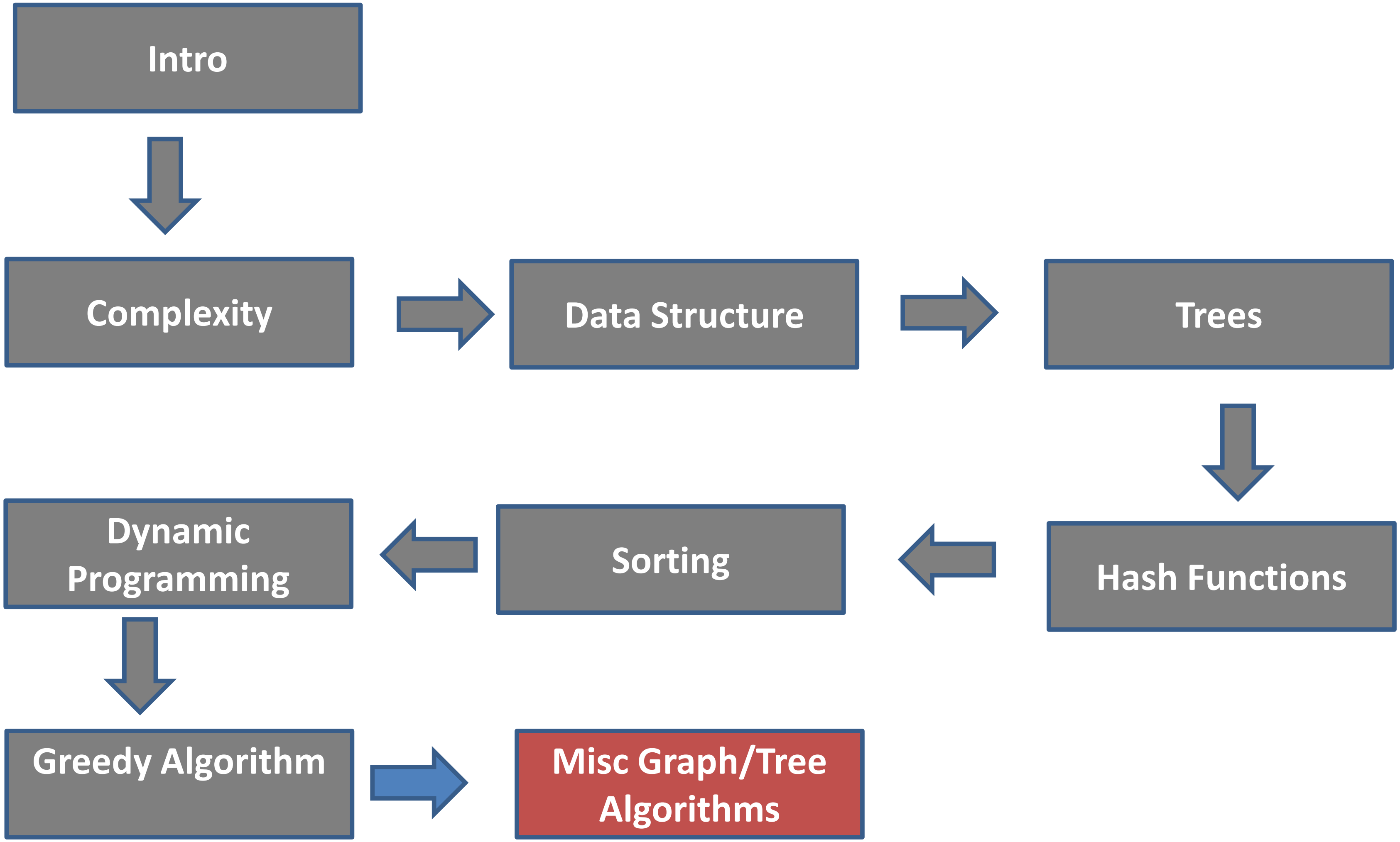
An Introduction to Algorithms

By

Hossein Rahmani

h_rahmani@iust.ac.ir
http://webpages.iust.ac.ir/h_rahmani/


Intro
Complexity
Data Structure
Trees
Hash Functions
Sorting
Dynamic Programming
Greedy Algorithm
Misc Graph/Tree Algorithms

Red-black trees: Overview

- Red-black trees are a variation of binary search trees to ensure that the tree is ***balanced***.
  - Height is $O(\lg n)$, where n is the number of nodes.
- Operations take $O(\lg n)$ time in the worst case.

Red-black Tree

- Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - *key*, *left*, *right*, and *p*.

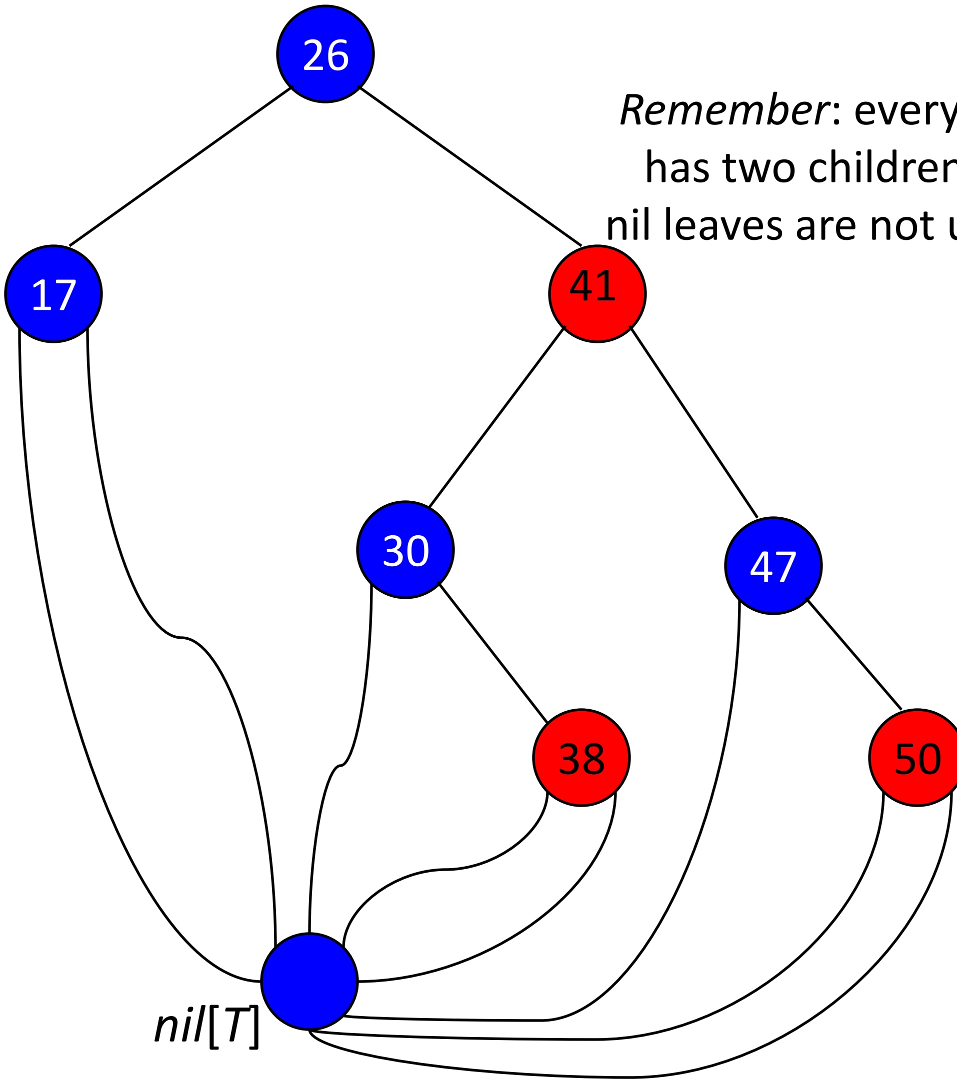
- All empty trees (leaves) are colored black.
  - We use a single sentinel, *nil*, for all the leaves of red-black tree *T*, with *color*[*nil*] = black.
  - The root's parent is also *nil*[*T*].

Red-black Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a <u>node is red</u>, then both its <u>children</u> are <u>black</u>.

5. For each node, <u>all paths</u> from the node to descendant leaves contain the <u>same number</u> of <u>black nodes</u>.

Red-black Tree – Example



Remember: every internal node has two children, even though nil leaves are not usually shown.

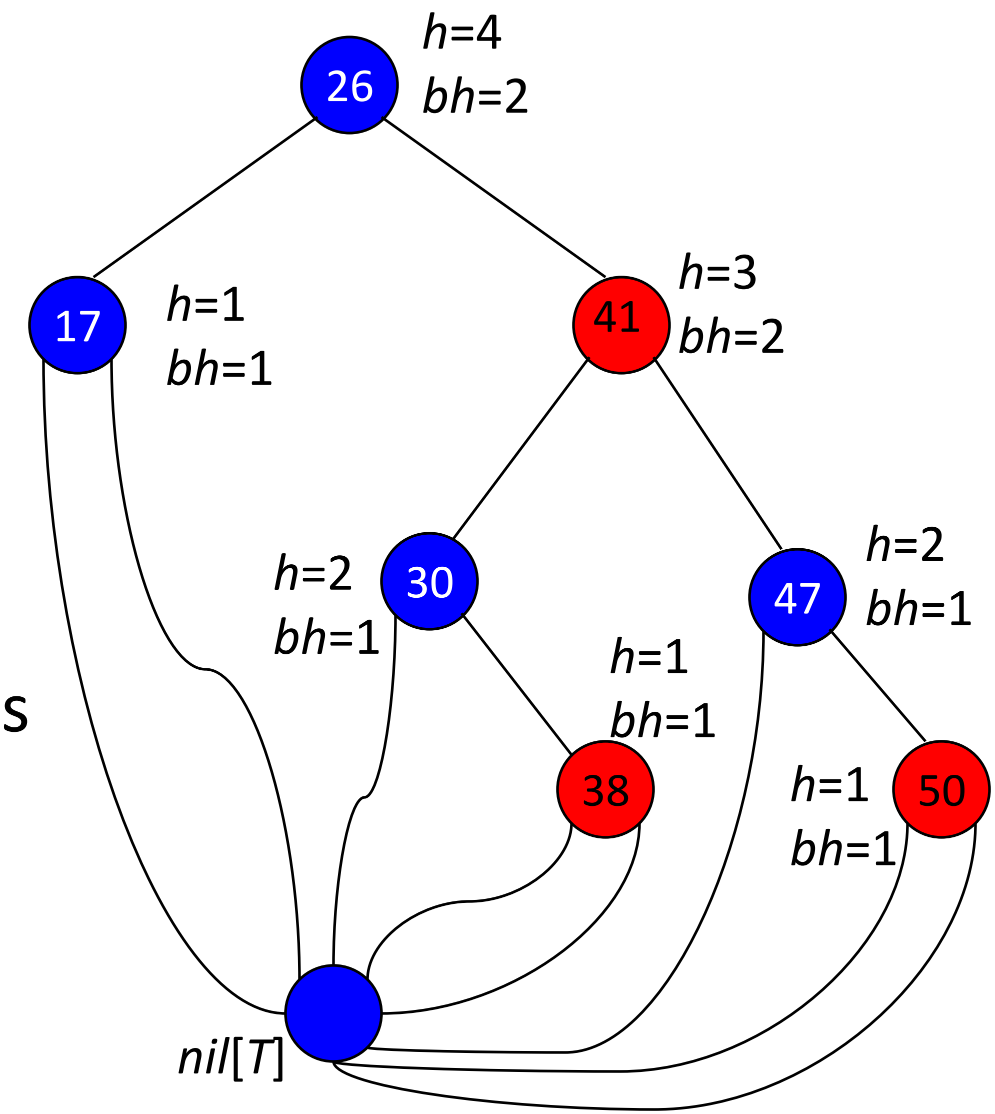
Height of a Red-black Tree

- Height of a node:
  - $h(x)$ = number of <u>edges</u> in a <u>longest path</u> to a leaf.
- Black-height of a node x, $bh(x)$:
  - $bh(x)$ = <u>number of black nodes</u> (including $nil[T]$) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.

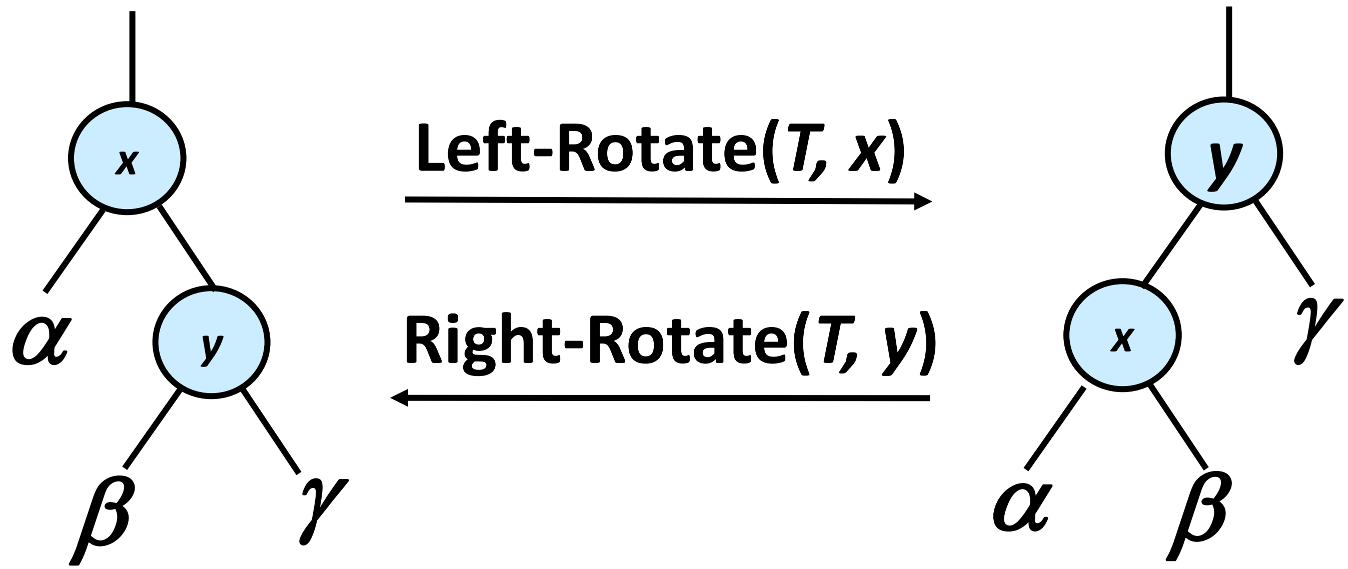
Height of a Red-black Tree

- Example:
- Height of a node:

  $h(x)$ = # of edges in a longest path to a leaf.
- Black-height of a node $bh(x)$ = # of black nodes on path from x to leaf, not counting x.

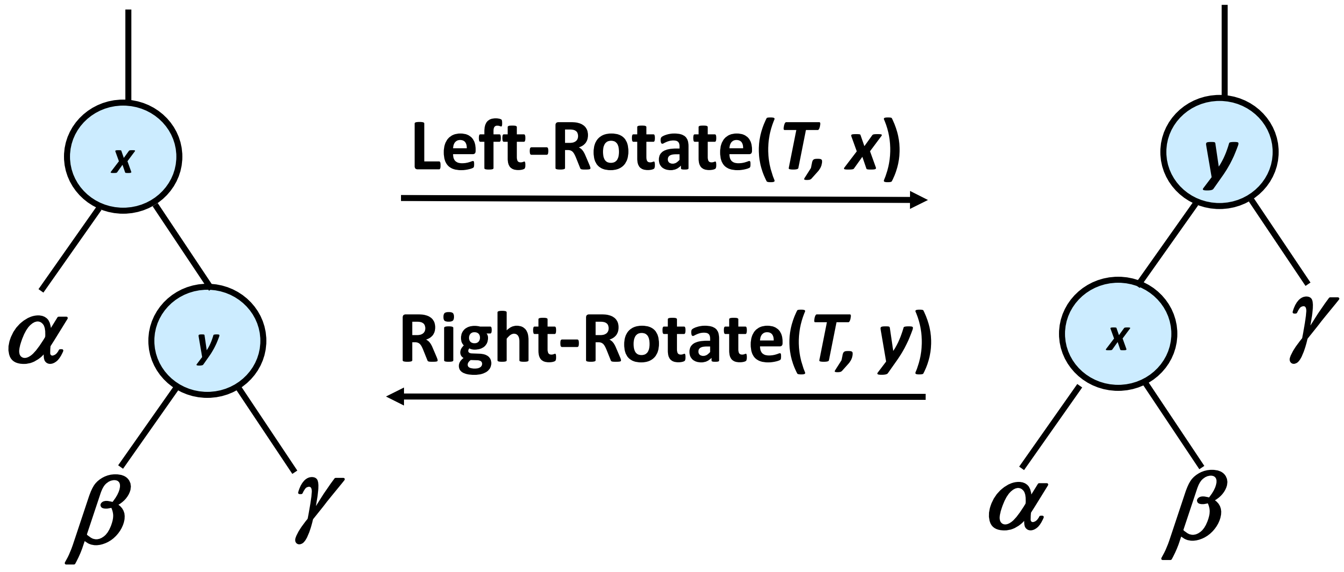


Rotations



Rotations

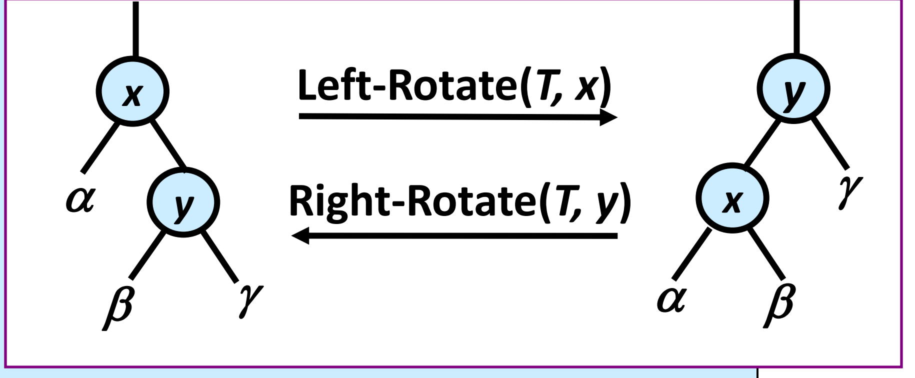
- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and
- Won't violate the binary-search-tree property.
- Left rotation and right rotation are inverses.



Left Rotation – Pseudo-code

Left-Rotate (*T*, *x*)

```
y ← right[x]   // Set y.
right[x] ← left[y]   //Turn y's left subtree into x's right subtree.
if left[y] ≠ nil[T]
    then p[left[y]] ← x
p[y] ← p[x]   // Link x's parent to y.
if p[x] = nil[T]
    then root[T] ← y
    else if x = left[p[x]]
        then left[p[x]] ← y
        else right[p[x]] ← y
left[y] ← x  // Put x on y's left.
p[x] ← y
```



Rotation

- The pseudo-code for Left-Rotate assumes that
  - *right*[*x*] $\neq$ *nil*[*T*], and
  - root's parent is *nil*[*T*].
- Left Rotation on *x*, makes *x* the left child of *y*, and the left subtree of *y* into the right subtree of *x*.
- Pseudocode for Right-Rotate is symmetric: exchange *left* and *right* everywhere.
- ***Time:*** $O(1)$ for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

Insertion in RB Trees

- Insertion must preserve all red-black properties.
- Should an inserted node be colored Red? Black?
- Basic steps:
  - Use Tree-Insert from BST (slightly modified) to insert a node *x* into *T*.
    - Procedure **RB-Insert(*x*)**.
  - Color the node *x* red.
  - Fix the modified tree by re-coloring nodes and performing rotation to preserve RB tree property.
    - Procedure **RB-Insert-Fixup**.

Insertion – Fixup

- <u>Problem</u>: we may have one pair of <u>consecutive reds</u> where we did the insertion.
- <u>Solution</u>: <u>rotate</u> it up the tree and away…

- Three cases have to be handled…

Insertion – Fixup

RB-Insert-Fixup (*T, z*)

while *color*[*p*[*z*]] = RED
    do if *p*[*z*] = *left*[*p*[*p*[*z*]]]
        then *y* ← *right*[*p*[*p*[*z*]]]
            if *color*[*y*] = RED
                then *color*[*p*[*z*]] ← BLACK // Case 1
                    color[*y*] ← BLACK // Case 1
                    color[*p*[*p*[*z*]]] ← RED // Case 1
                    z ← *p*[*p*[*z*]] // Case 1

Insertion – Fixup

RB-Insert-Fixup(*T, z*) (Contd.)

9. **else if** $z = right[p[z]]$ // color$[y] \neq$ RED

10. **then** $z \leftarrow p[z]$ // Case 2

11. LEFT-ROTATE(T, z) // Case 2

12. $color[p[z]] \leftarrow$ BLACK // Case 3

13. $color[p[p[z]]] \leftarrow$ RED // Case 3

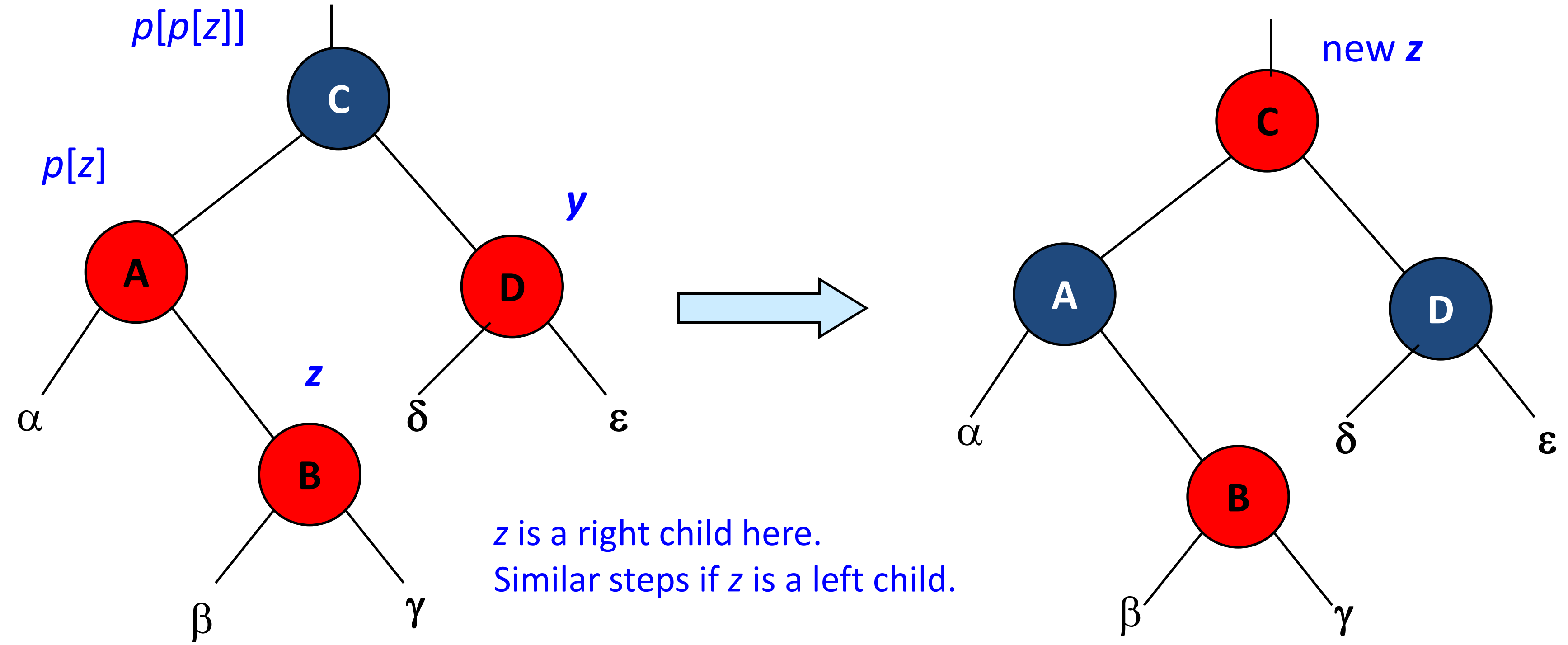
14. RIGHT-ROTATE$(T, p[p[z]])$ // Case 3

15. **else** (if $p[z] = right[p[p[z]]]$)(same as **10-14**

16. with “right” and “left” exchanged)

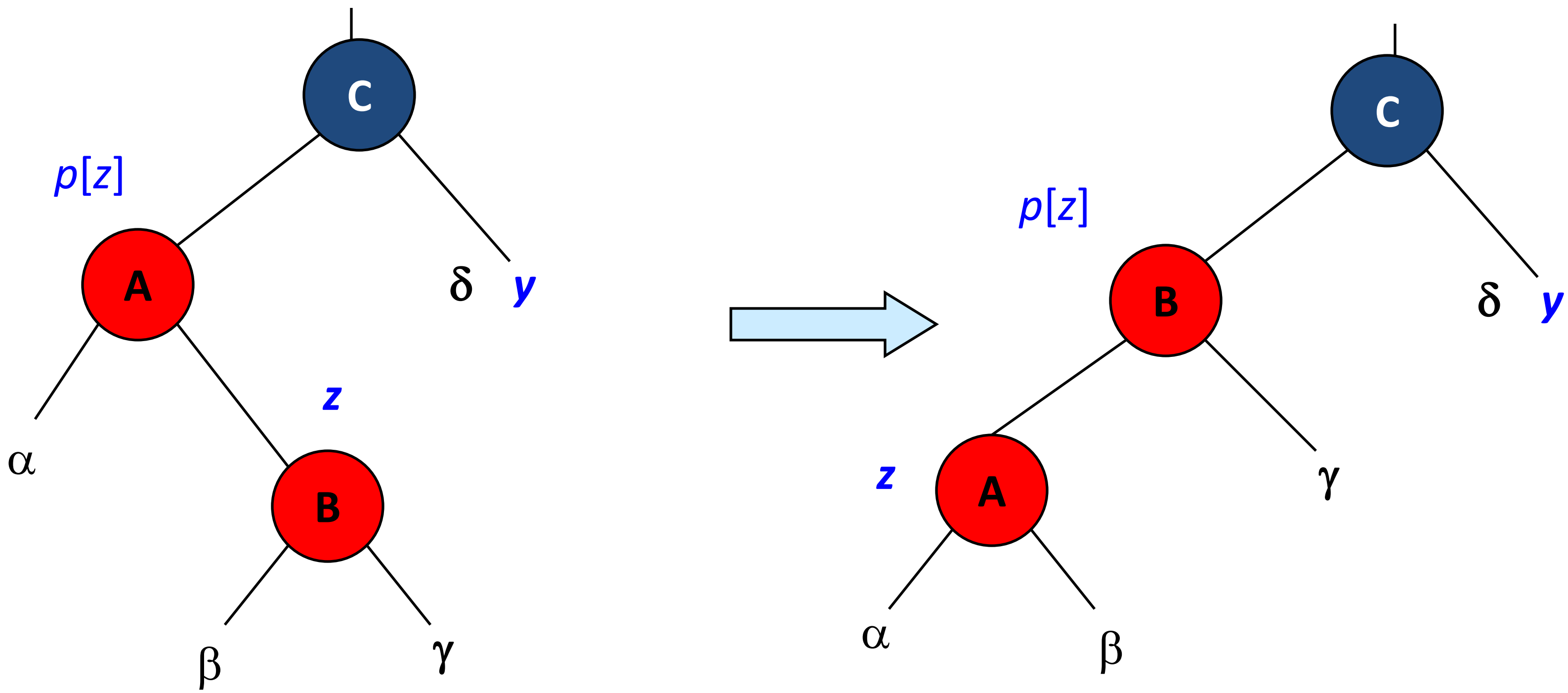
17. $color[root[T]] \leftarrow$ BLACK

Case 1 – uncle *y* is red



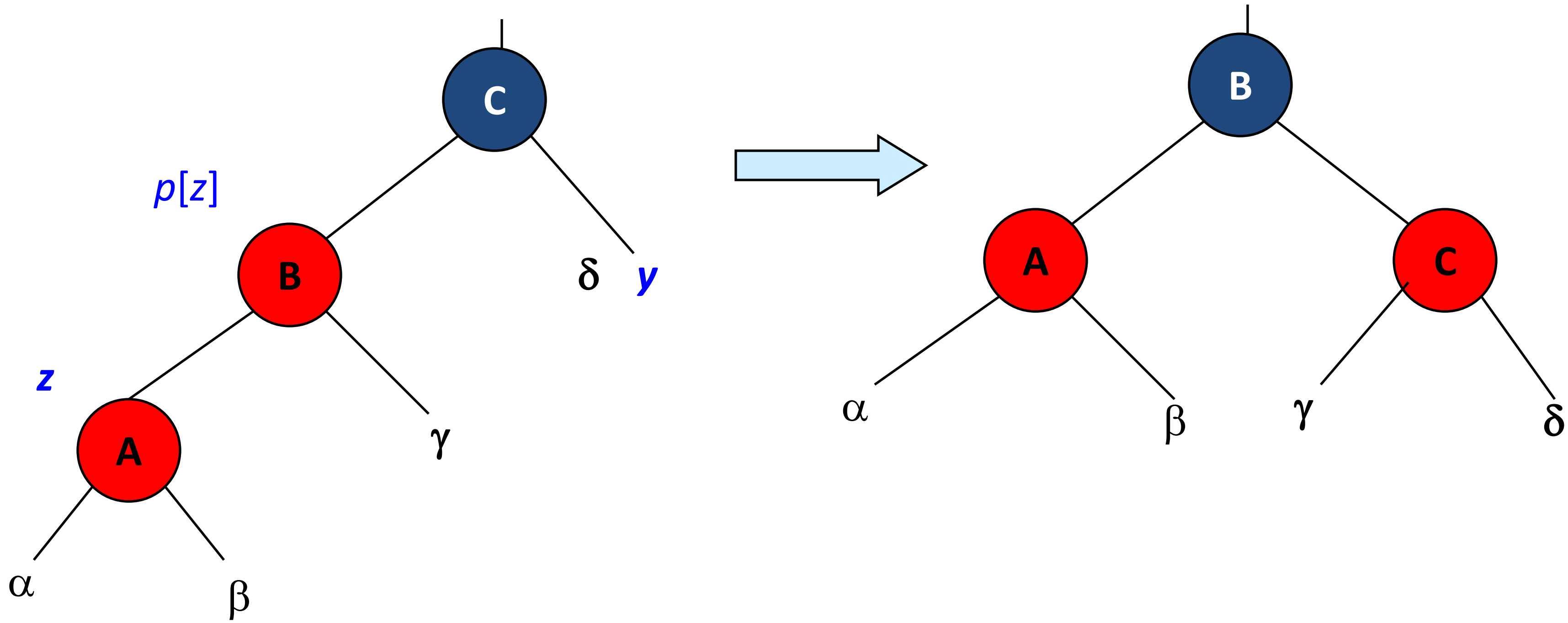
- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black $\Rightarrow$ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Rightarrow$ restores property 5.
- The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).

Case 2 – *y* is black, *z* is a right child



- Left rotate around *p*[*z*], *p*[*z*] and *z* switch roles $\Rightarrow$ now *z* is a left child, and both *z* and *p*[*z*] are red.
- Takes us immediately to case 3.

Case 3 – *y* is black, *z* is a left child



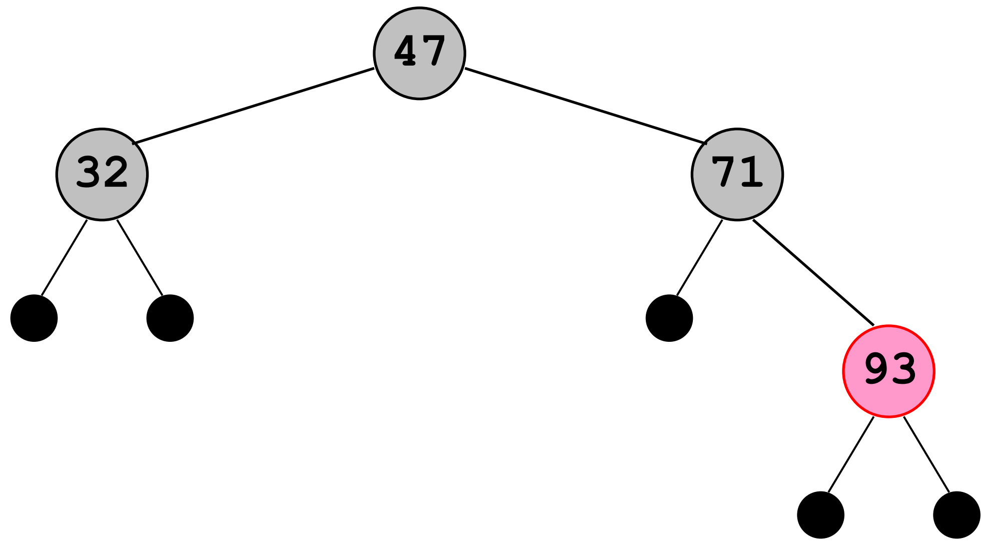
- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

Deletion

- Deletion, like insertion, should preserve all the RB properties.
- The properties that may be violated depends on the color of the deleted node.
  - Red – OK.
  - Black?
- Steps:
  - Do regular BST deletion.
  - Fix any violations of RB properties that may result.

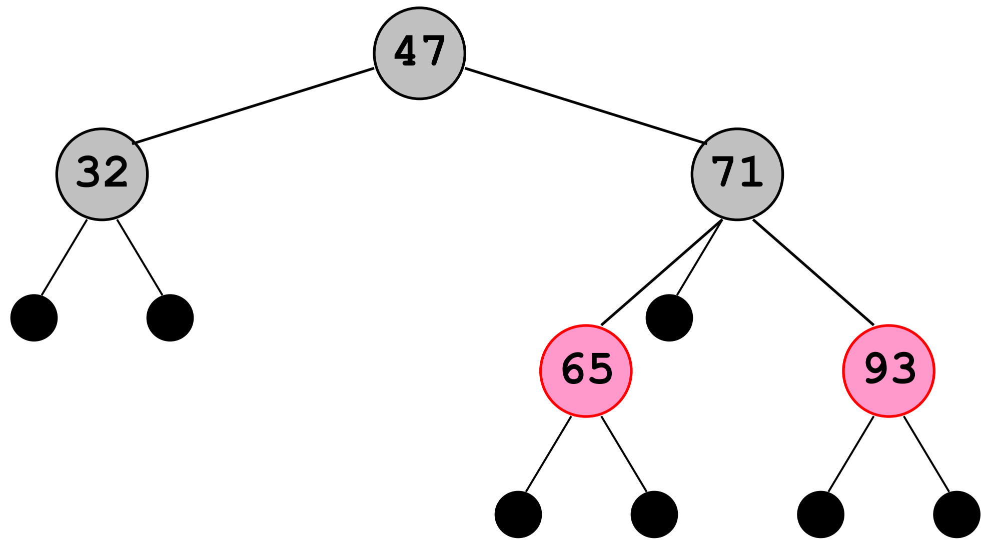
Insertion Example

`Insert 65`



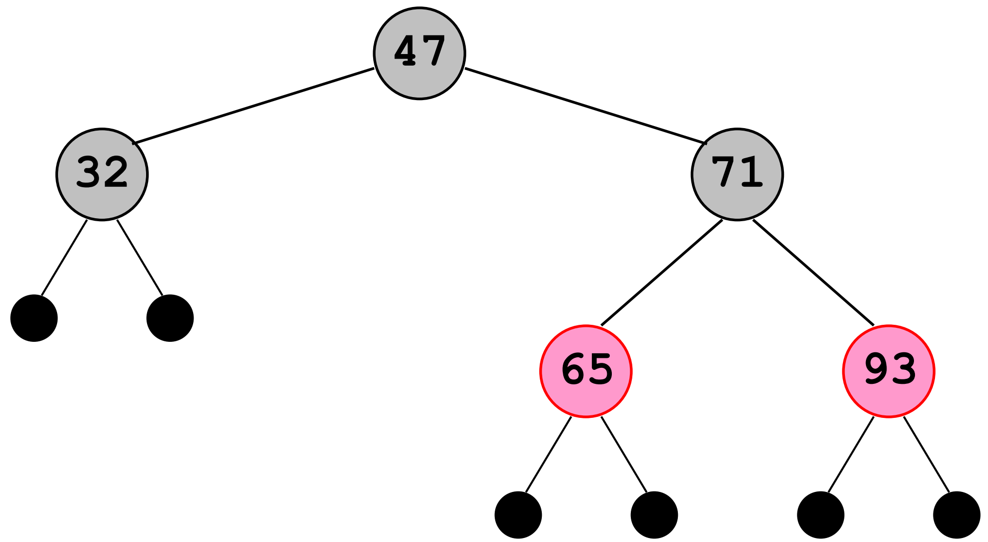
Insertion Example

`Insert 65`



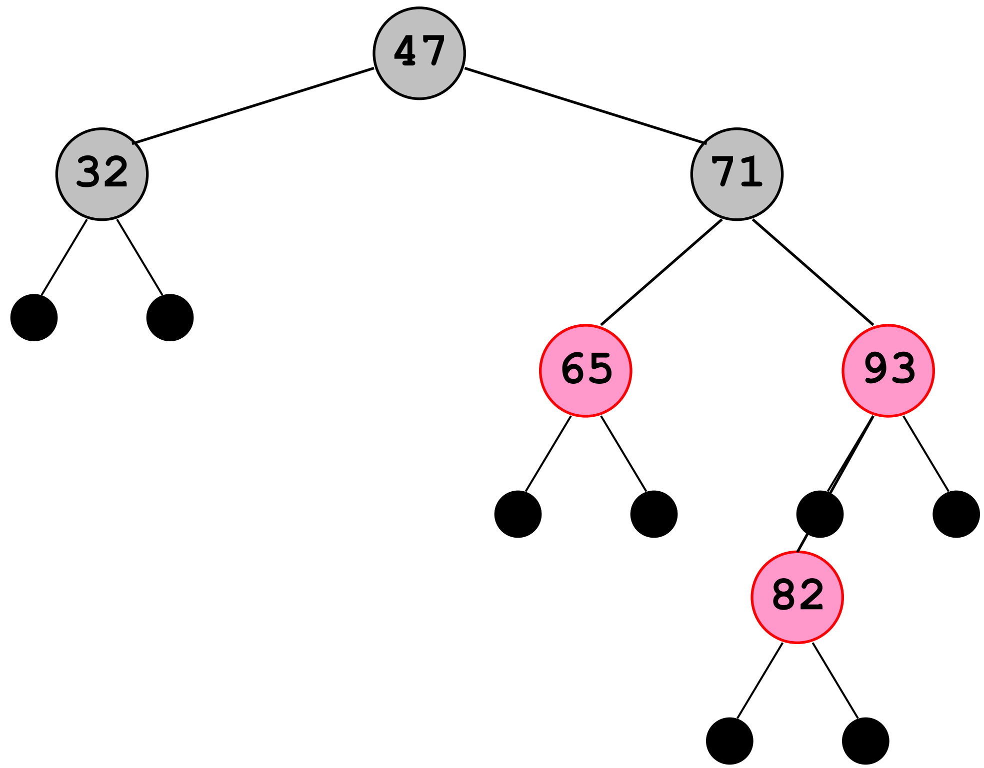
Insertion Example

```
Insert 65
Insert 82
```



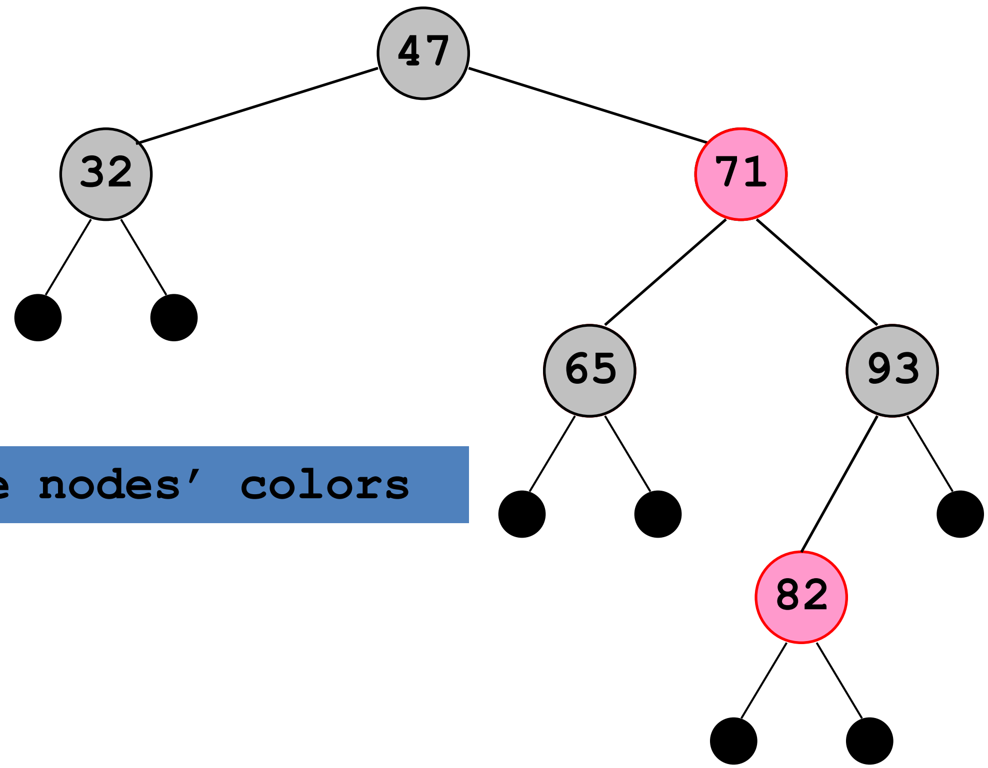
Insertion Example

`Insert 65`
`Insert 82`



Insertion Example

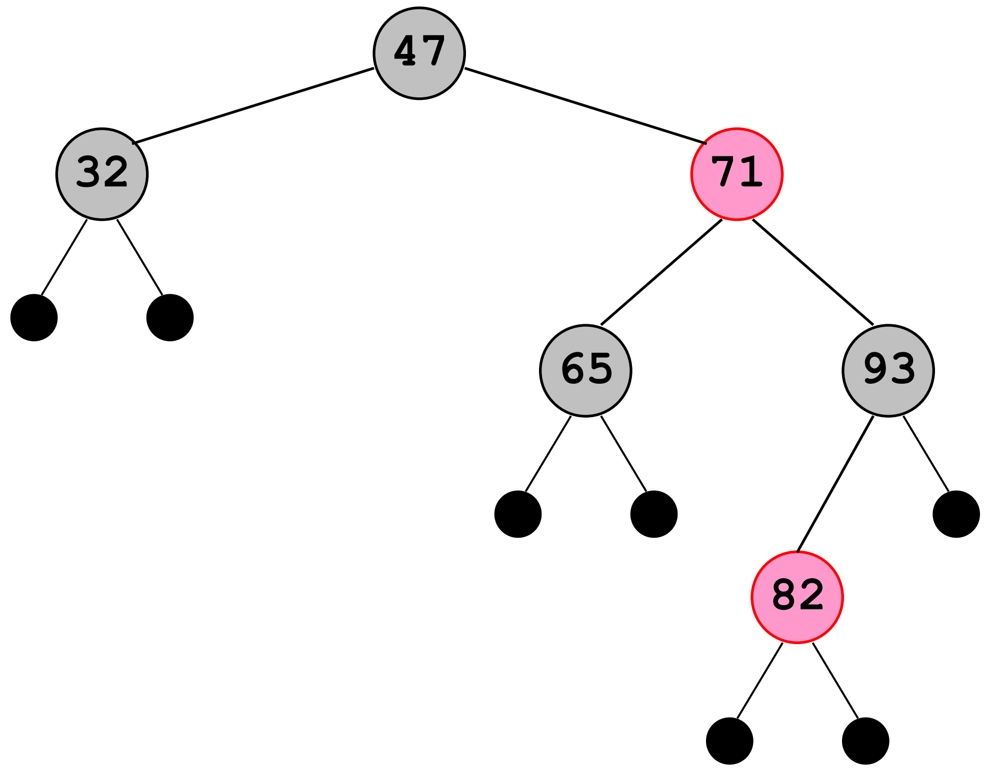
Insert 65
Insert 82



change nodes' colors

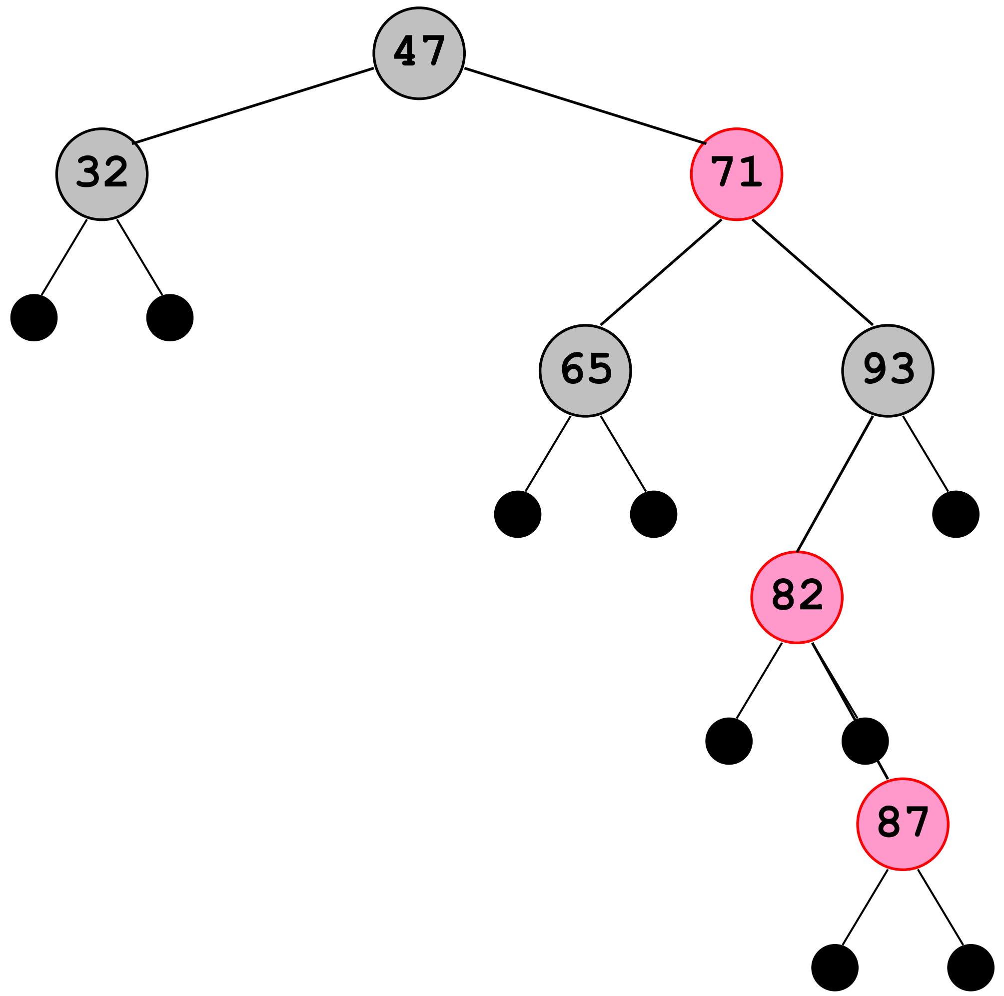
Insertion Example

`Insert 65`
`Insert 82`
`Insert 87`



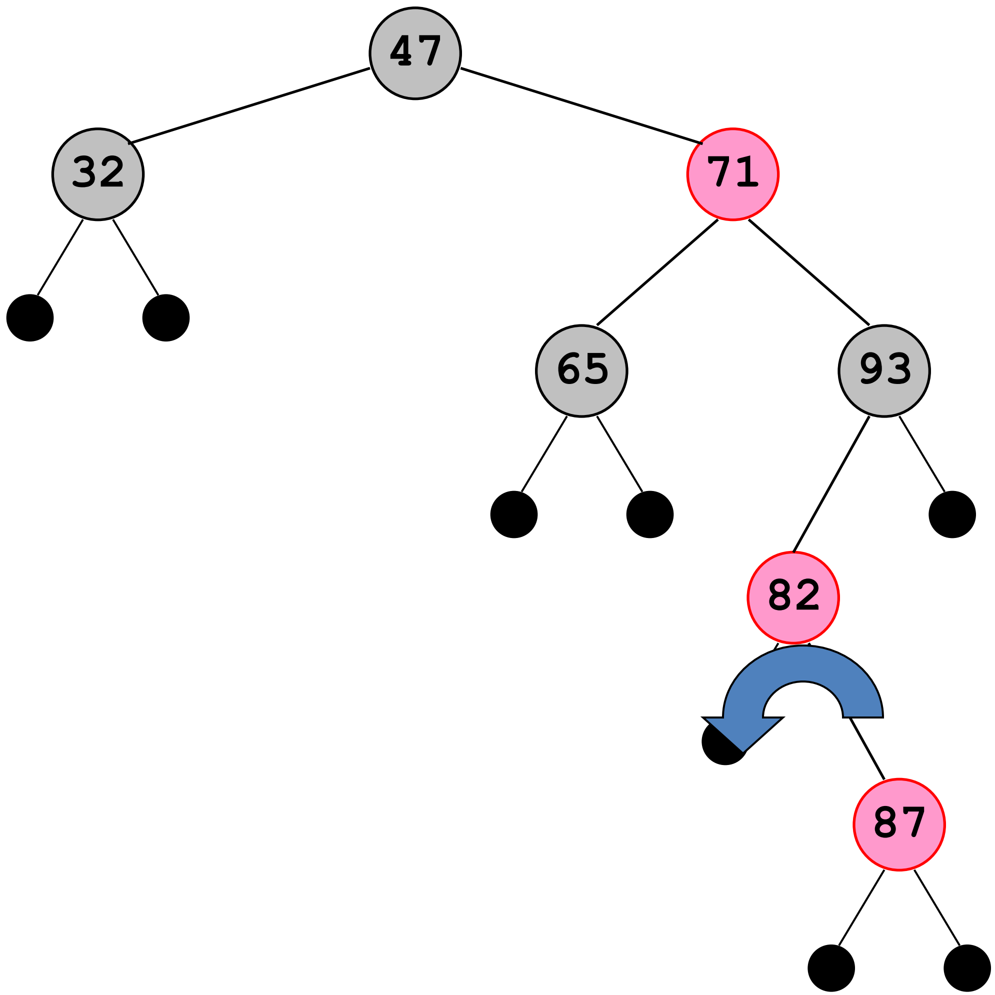
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



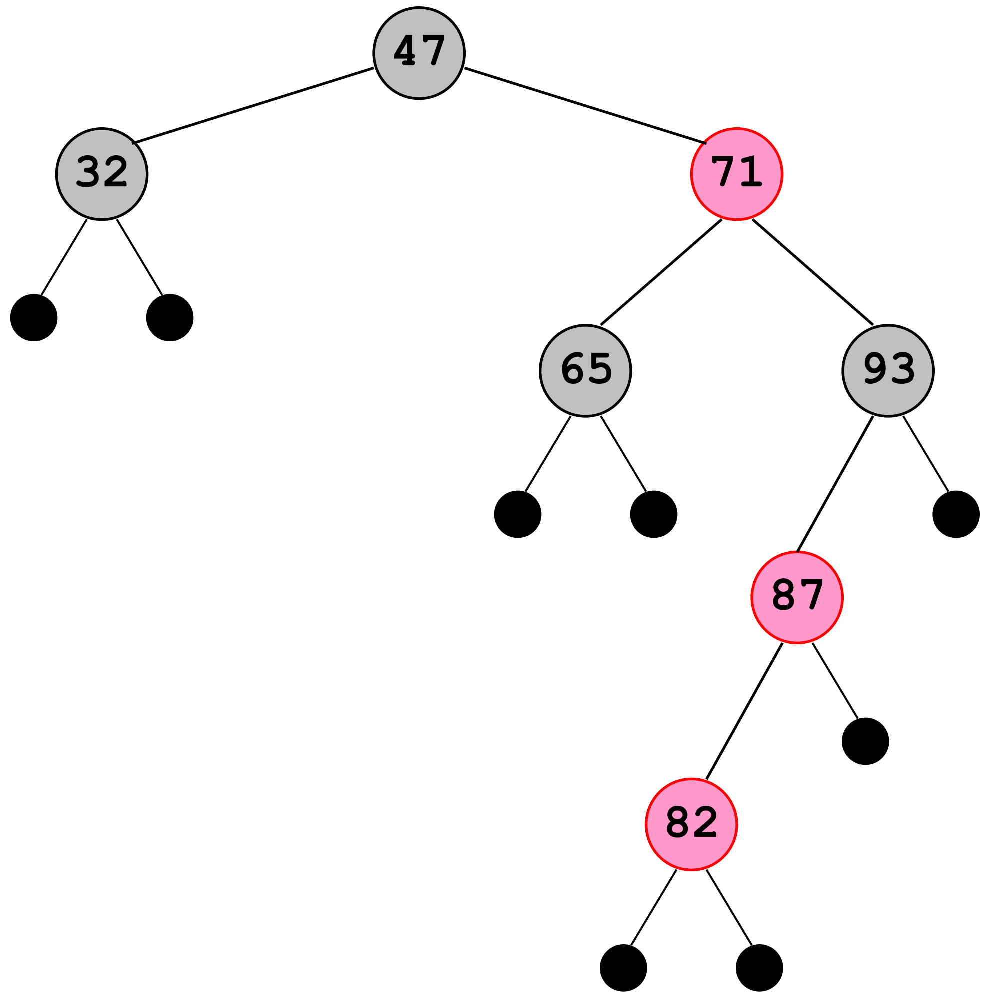
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



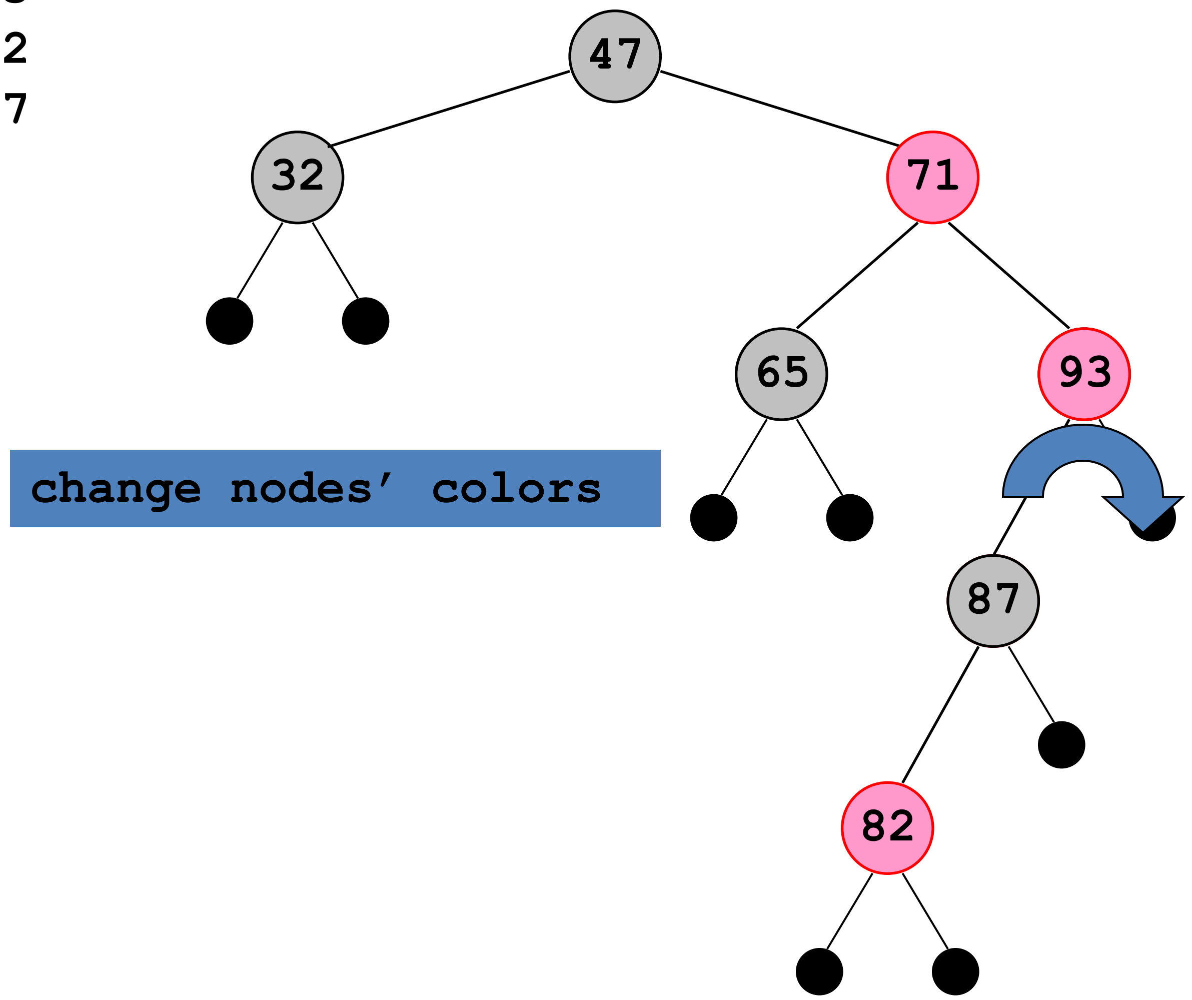
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



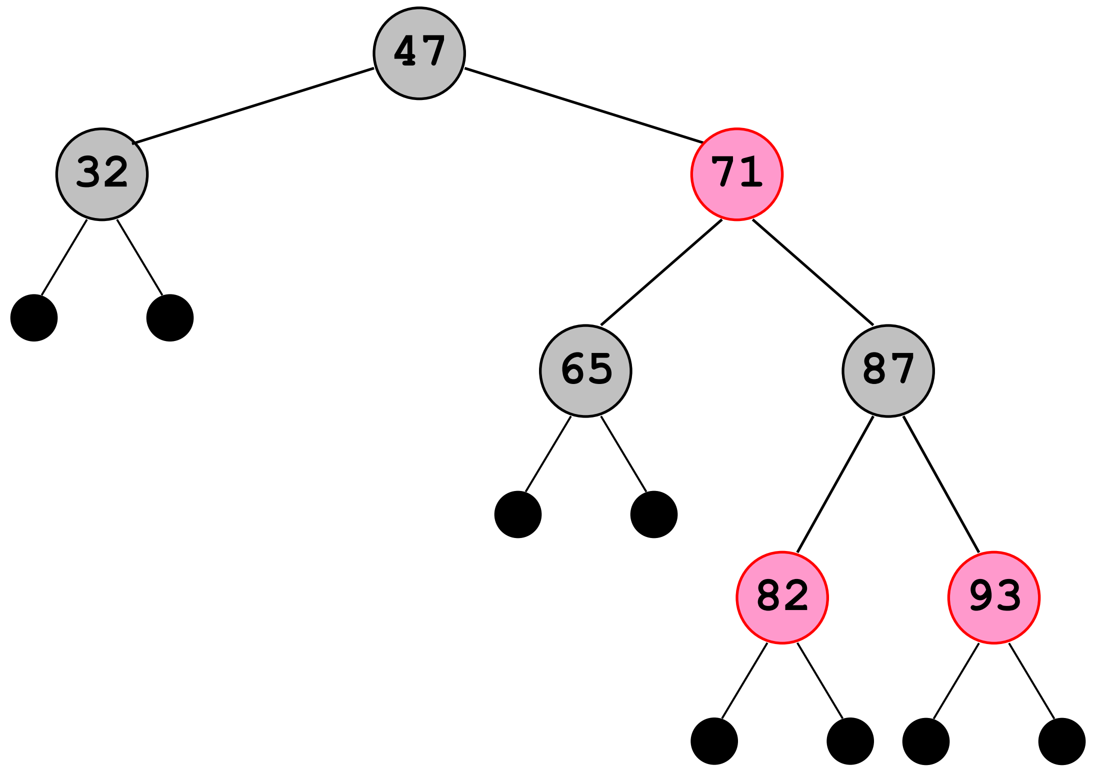
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



Insertion Example

```
Insert 65
Insert 82
Insert 87
```



Any FeedBack!

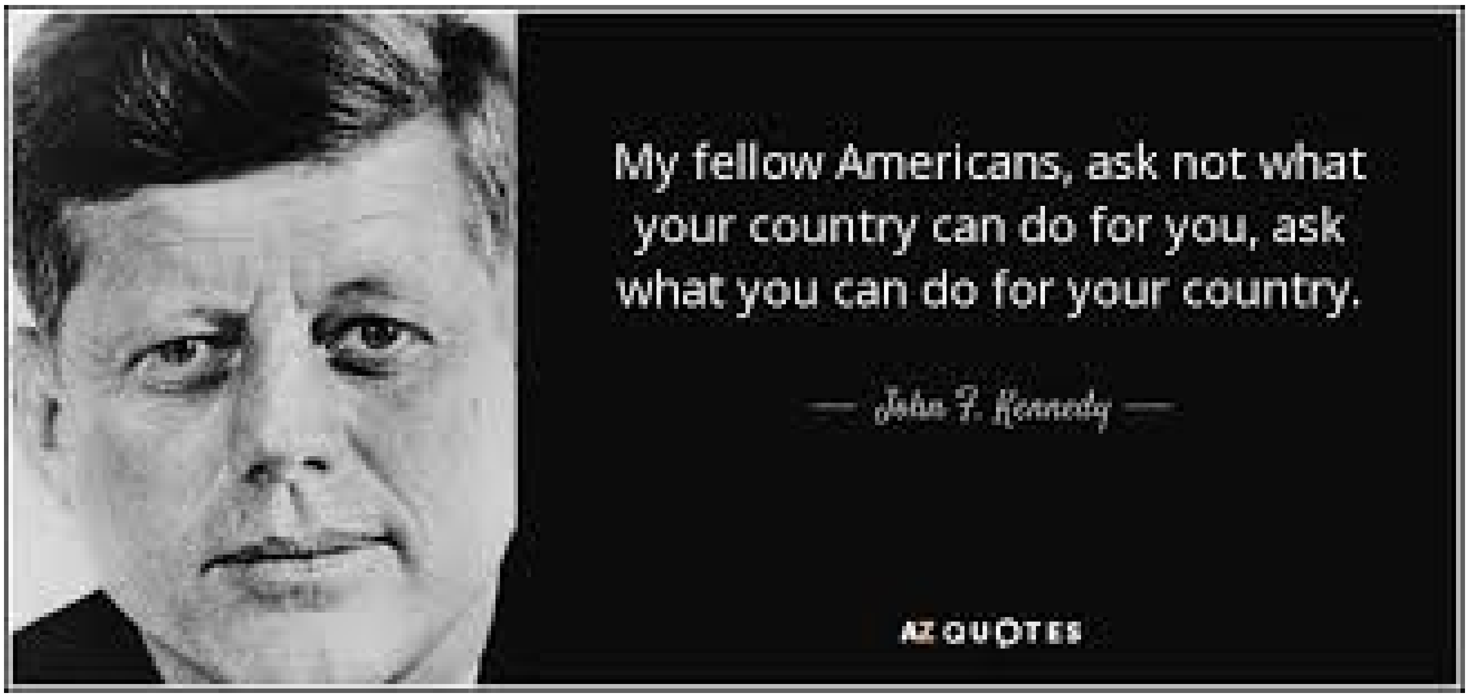
Professor

TA

You/Other students

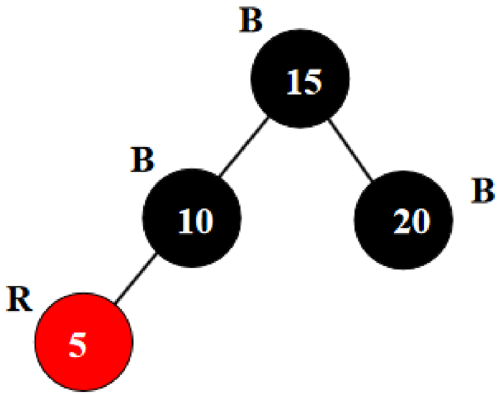
Department/ University

پاسخ به بازخوردهای بچه‌ها (دانشکده)



Quiz 1

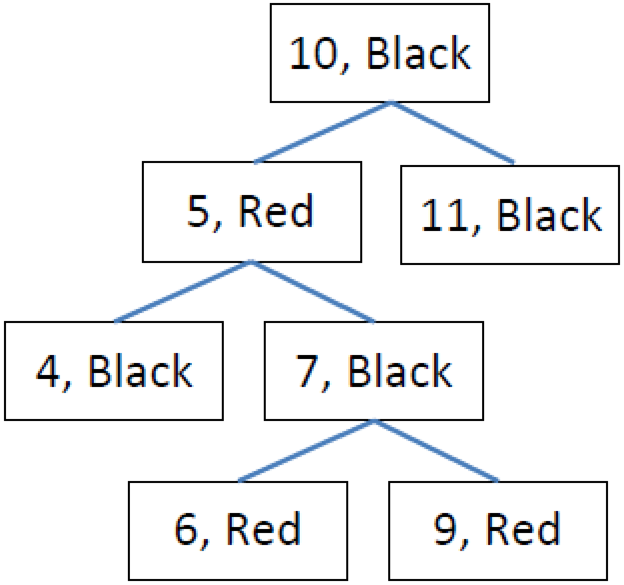
Consider the following valid red-black tree, where "R" indicates a red node, and "B" indicates a black node. Note that the black dummy sentinel leaf nodes are not shown.



Insert Key 3 in the above tree?

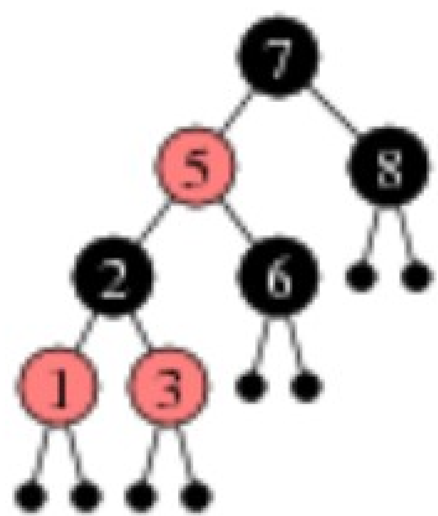
Quiz 2

Given the red -black tree shown below. Insert key 8 and redraw the tree. Remember to write the color of each node



Quiz 3

Suppose we have a red-black tree as diagrammed at right, and then we <u>insert 4</u> using the insertion algorithm discussed in class. Diagram the resulting tree, indicating which nodes are red and which are black.



Quiz 1 Result