CS146: Data Structures and Algorithms
Lecture 11

RED BLACK TREES

INSTRUCTOR: KATERINA POTIKA
CS SJSU

Red-Black Trees (Ch 13)

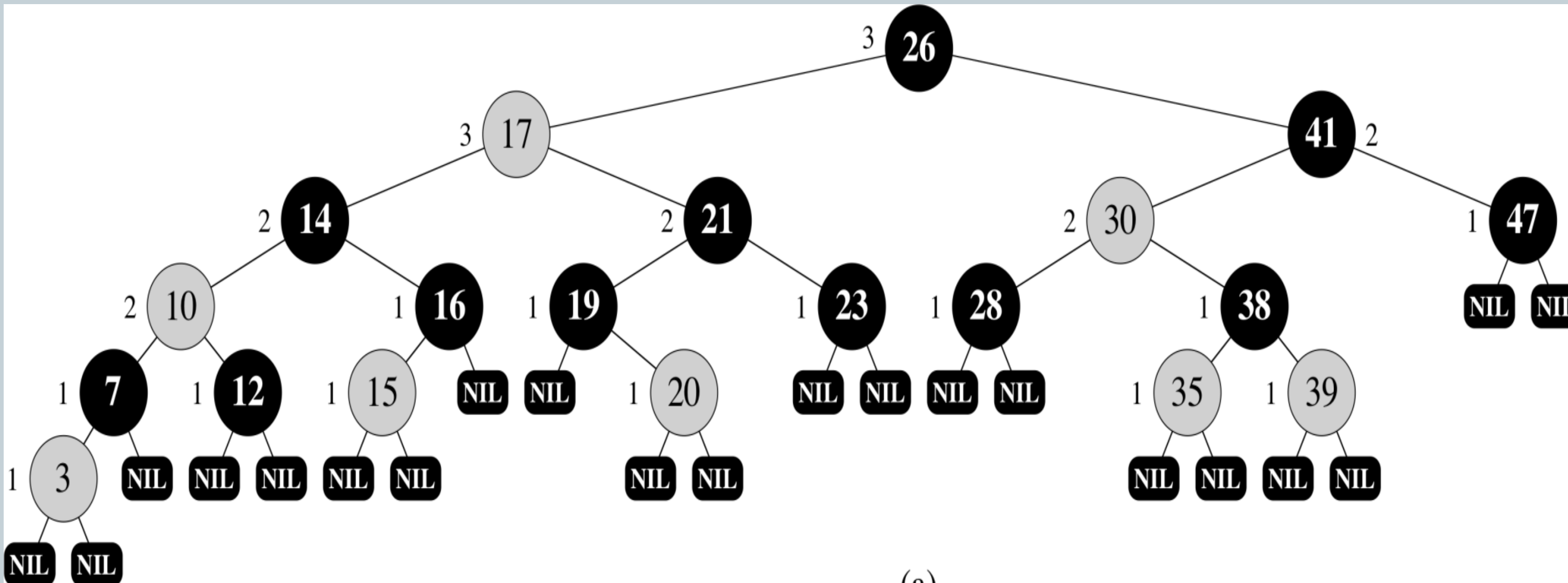
- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height $h = O(\lg n)$

- First: describe the properties of red-black trees
- Then: prove that these guarantee $h = O(\lg n)$
- Finally: describe operations on red-black trees

Red-Black Trees

- Put example on board and verify properties:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black
- *black-height*: # black nodes on path to leaf
  - Label example with bh values

Example



3 26
3 17
41 2
2 14
2 21
2 30
1 47
2 10
1 16
1 19
1 23
1 28
1 38
NIL NIL
1 7
1 12
1 15
NIL
NIL
1 20
NIL NIL NIL NIL
1 35
1 39
1 3
NIL NIL NIL NIL NIL
NIL NIL
NIL NIL NIL NIL
NIL NIL

(a)

Black-Height

- *black-height*: # black nodes on path to leaf (not starting node)
- *What is the minimum black-height of a node with height h?*
- A: a height-h node has black-height $\geq h/2$ (Why?)

- **Theorem**: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$
- Proof: by induction (follows)

RB Trees: Proving Height Bound II

First prove:

- Claim1: A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes
- Proof by induction on height h
  - Base step: x has height 0 (i.e., NULL leaf node)
    - *What is bh(x)?*
    - A: 0
    - So…subtree contains $2^{bh(x)} - 1$
      $= 2^0 - 1$
      $= 0$ internal nodes (TRUE)

RB Trees: Proving Height Bound III

- Inductive proof that subtree at node x contains at least $2^{bh(x)} - 1$ internal nodes
  - Inductive step: x has positive height and 2 children
    - Each child has black-height of $bh(x)$ or $bh(x)-1$
      - Note: the height of a child = (height of x) – 1
    - So the subtrees rooted at each child contain at least $2^{bh(x)-1} - 1$ internal nodes (from hypothesis)
    - Thus subtree at x contains s $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$ $= 2 \bullet 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$ nodes

Review: Proving Height Bound IV

- Thus at the root of the red-black tree:

  $n \geq 2^{\mathrm{bh}(root)} - 1$

  $n \geq 2^{h/2} - 1$ (claim2)

  $\lg(n+1) \geq h/2$

  $h \leq 2 \lg(n + 1)$

  Thus $h = O(\lg n)$

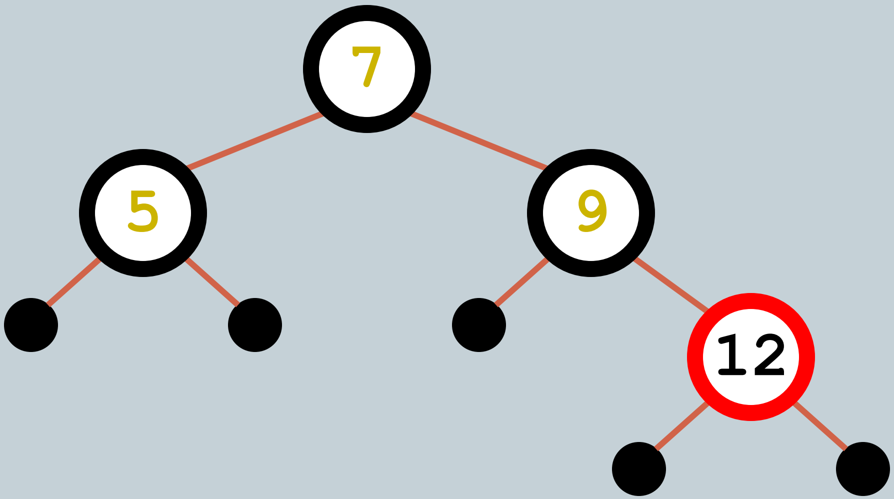
- ***Claim2:*** Any node with height h has black-height $\geq h/2$.
- ***Proof :*** By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black.

RB Trees: Worst-Case Time

- So we've proved that a red-black tree has O(lg n) height
- Corollary: These operations take O(lg n) time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- Insert() and Delete():
  - Will also take O(lg n) time
  - But will need special care since they modify tree

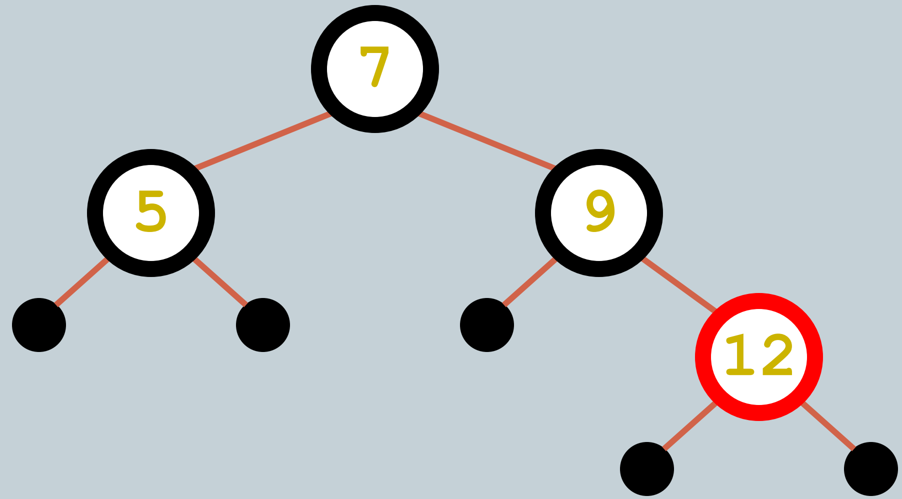
Red-Black Trees: An Example

- *Color this tree:*



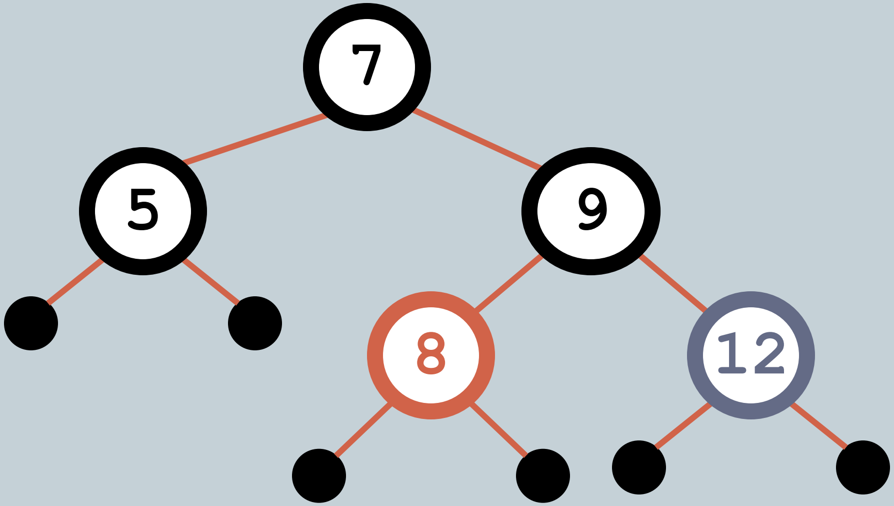
Red-Black Trees: Problem With Insertion

- Insert 8
  - *Where does it go?*



Red-Black Trees: Problem With Insertion

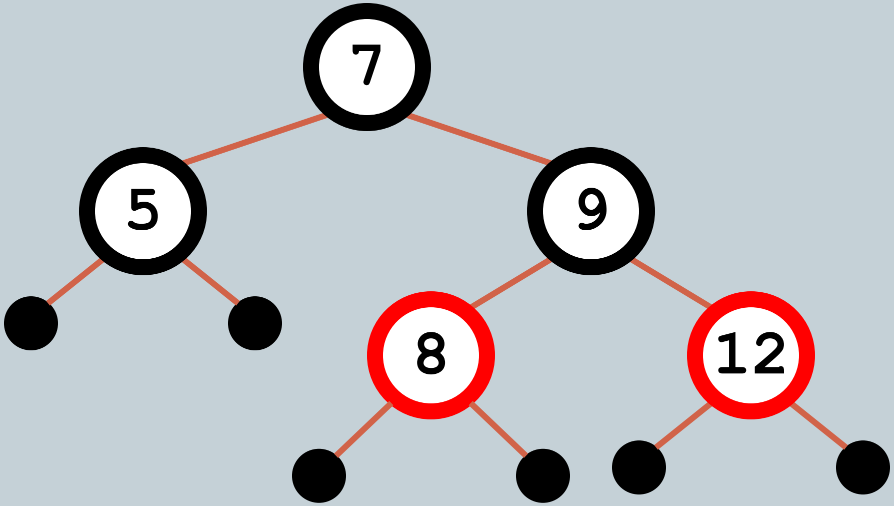
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

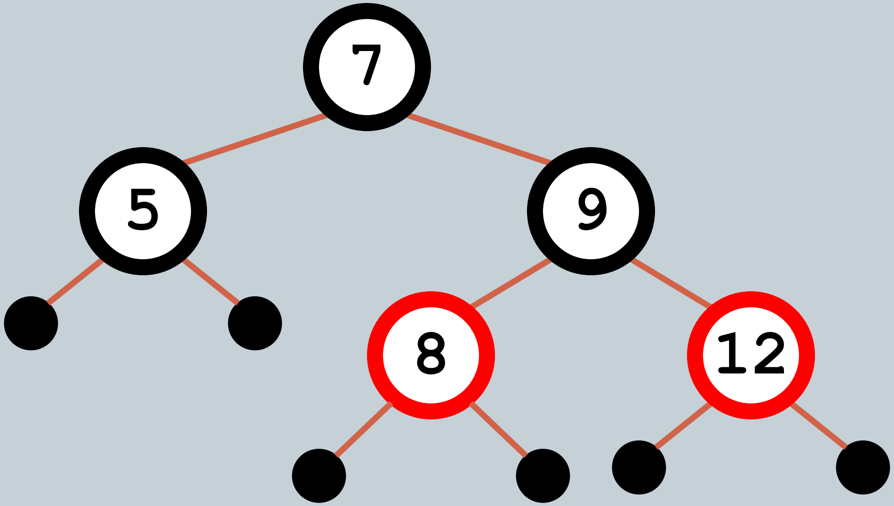
Red-Black Trees: Problem With Insertion

- Insert 8
  - *Where does it go?*
  - *What color should it be?*



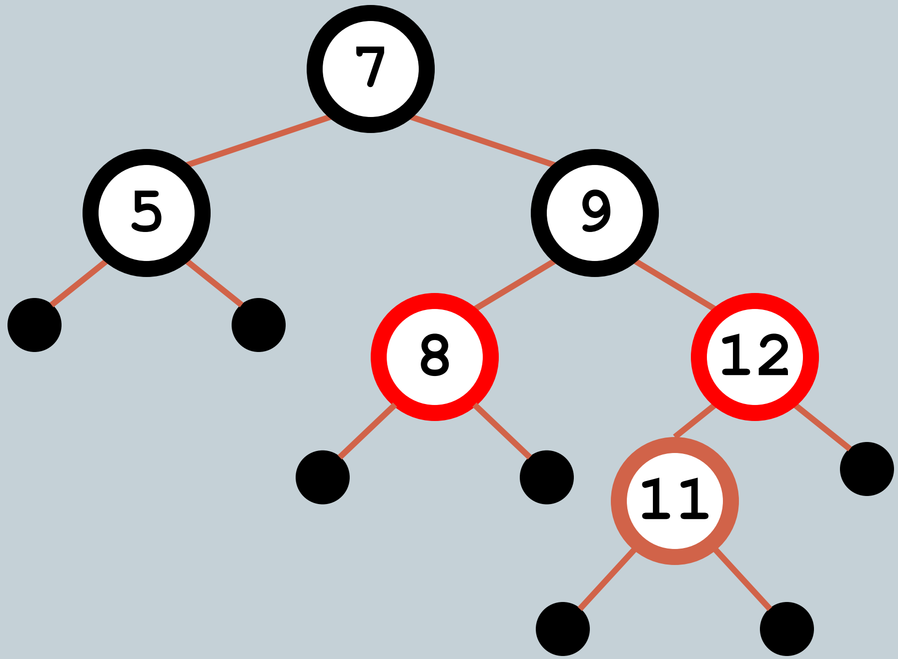
Red-Black Trees: Problem With Insertion

- Insert 11
  - *Where does it go?*



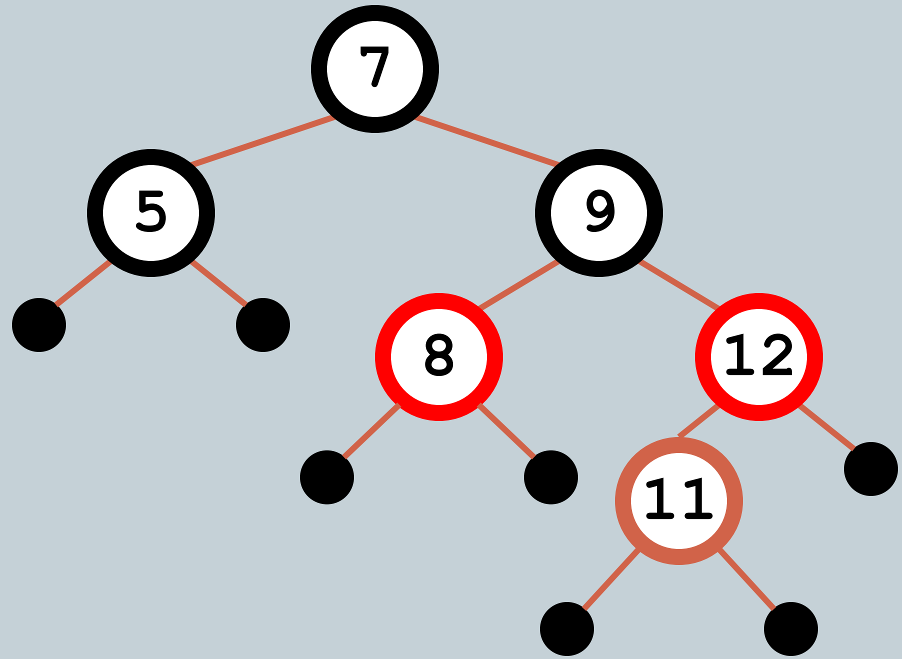
Red-Black Trees: Problem With Insertion

- Insert 11
  - *Where does it go?*
  - *What color?*



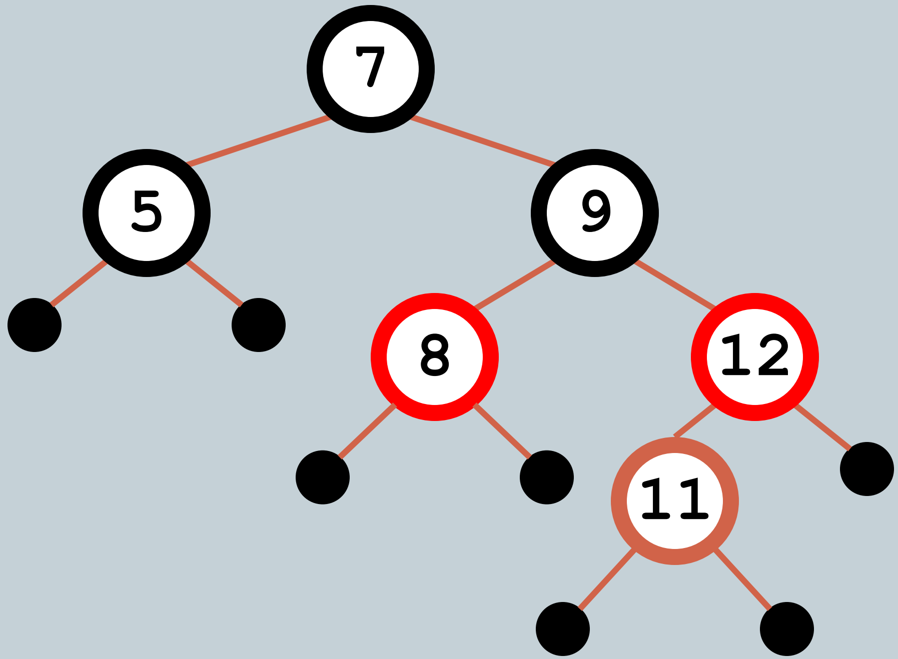
Red-Black Trees: Problem With Insertion

- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)



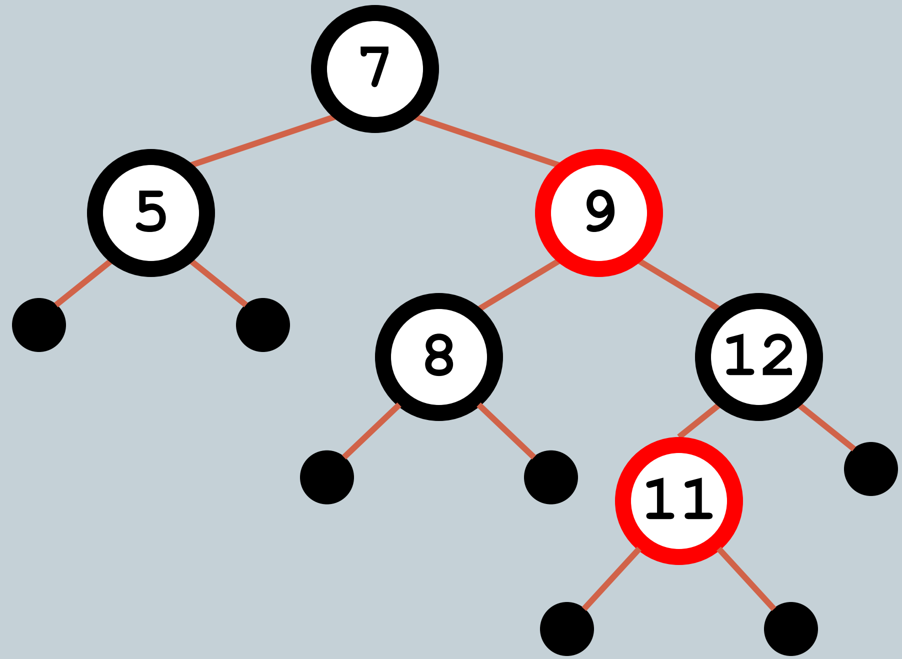
Red-Black Trees: Problem With Insertion

- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)
    - Can't be black! (#4)



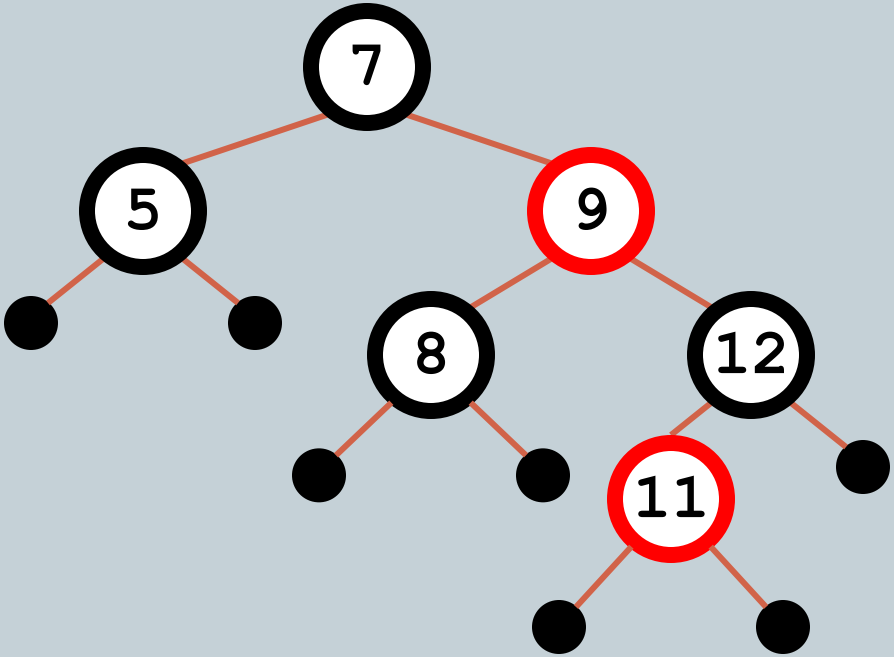
Red-Black Trees: Problem With Insertion

- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution:
      recolor the tree



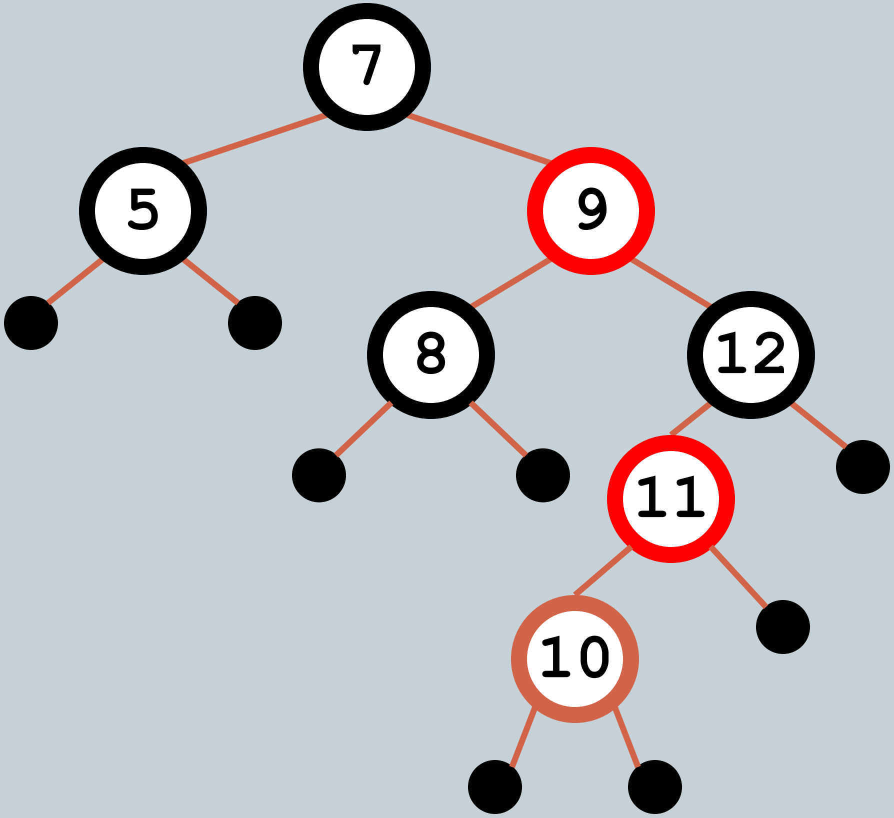
Red-Black Trees: Problem With Insertion

- Insert 10
  - *Where does it go?*



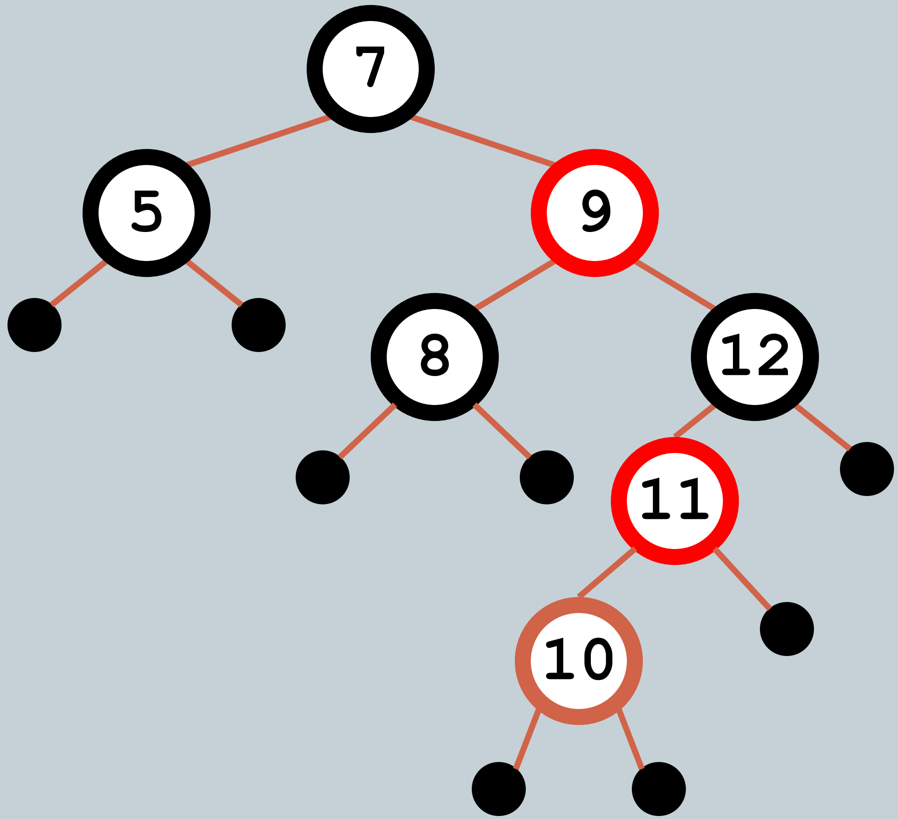
Red-Black Trees: Problem With Insertion

- Insert 10
  - *Where does it go?*
  - *What color?*



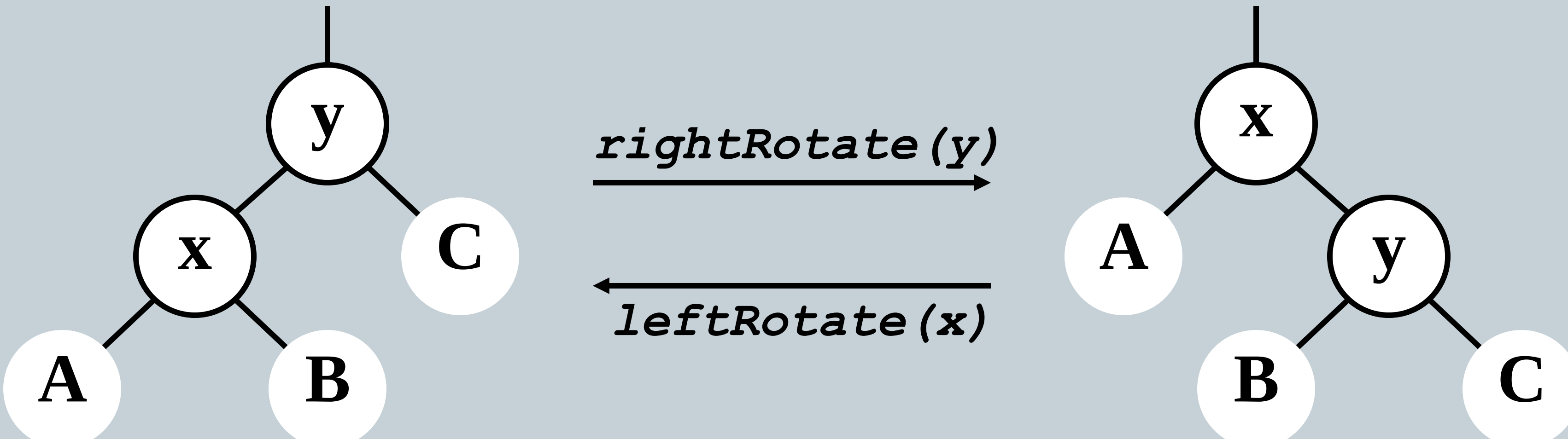
Red-Black Trees: Problem With Insertion

- Insert 10
  - *Where does it go?*
  - *What color?*
    - A: no color! Tree is too imbalanced
    - Must change tree structure to allow recoloring
  - Goal: restructure tree in O(lg *n*) time



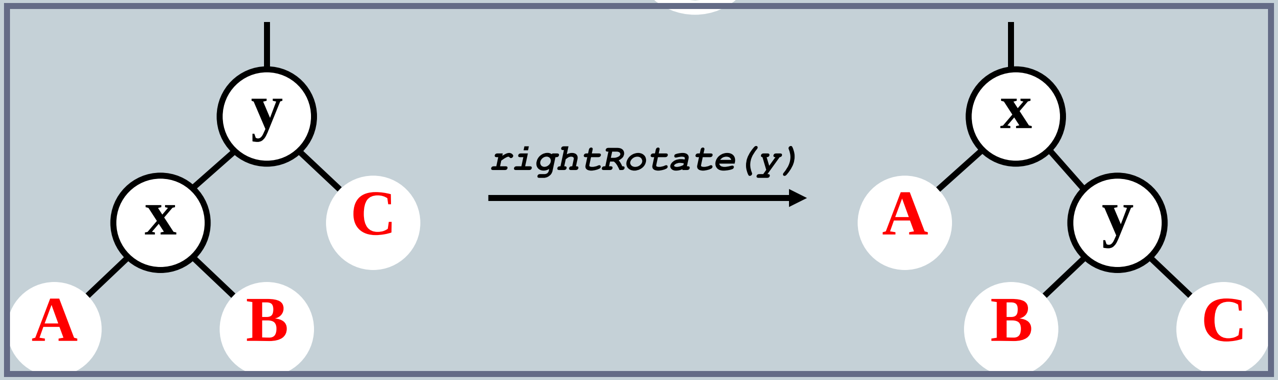
RB Trees: Rotation

- Our basic operation for changing tree structure is called *rotation*:



- *Does rotation preserve inorder key ordering?*
- *What would the code for* **`rightRotate()`** *actually do?*

RB Trees: Rotation



- Answer: A lot of pointer manipulation
  - x keeps its left child
  - y keeps its right child
  - x's right child becomes y's left child
  - x's and y's parents change
- *What is the running time?*

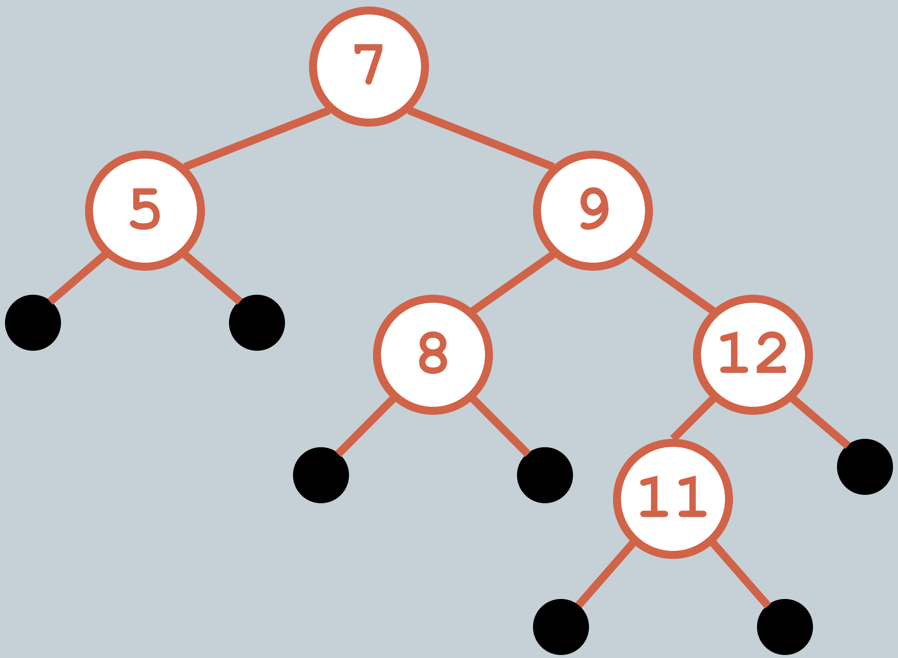
Left Rotate

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
```

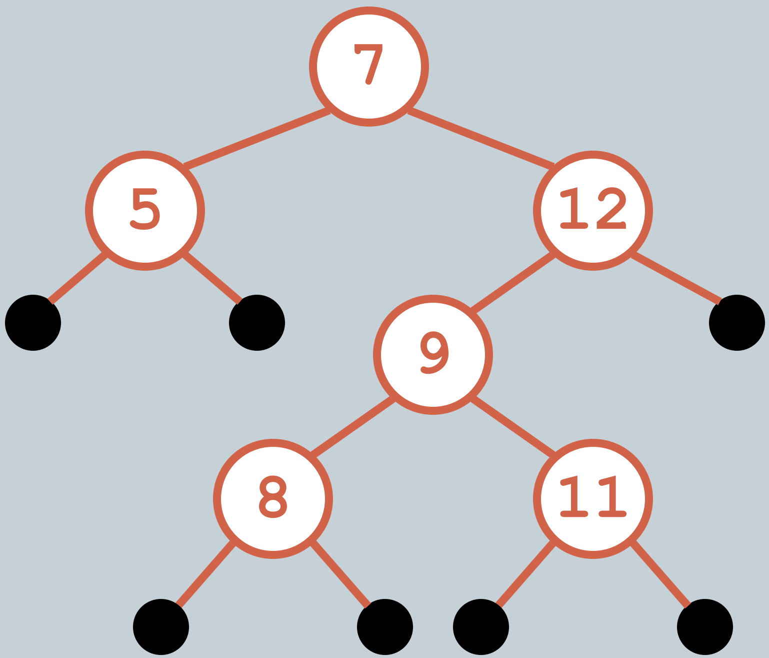
Rotation Example

- Rotate left about 9:



Rotation Example

- Rotate left about 9:



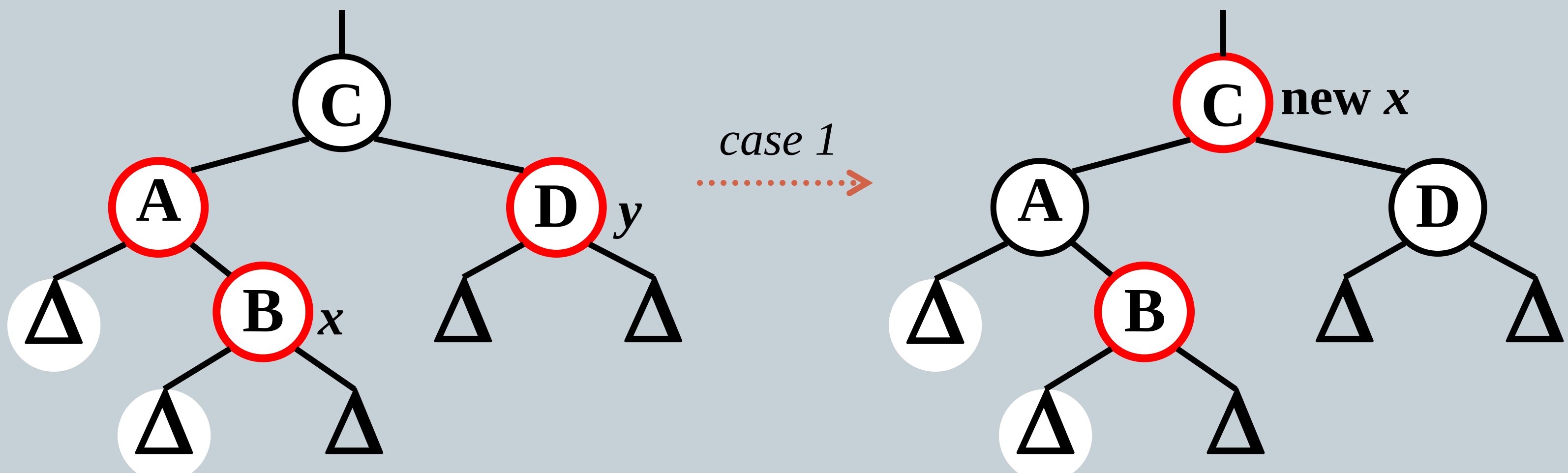
Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert *x* into tree, color *x* red
  - Only r-b property #3 might be violated (if x.p red)
    - If so, move violation up tree until a place is found where it can be fixed
  - Total time will be O(lg *n*)

RB Insert: Case 1


```
if (y.color == RED)
    x.p.color = BLACK;
    y.color = BLACK;
    x.p.p.color = RED;
    x = x.p.p;
```

- Case 1: “uncle” is red
- In figures below, all Δ’s are equal-black-height subtrees

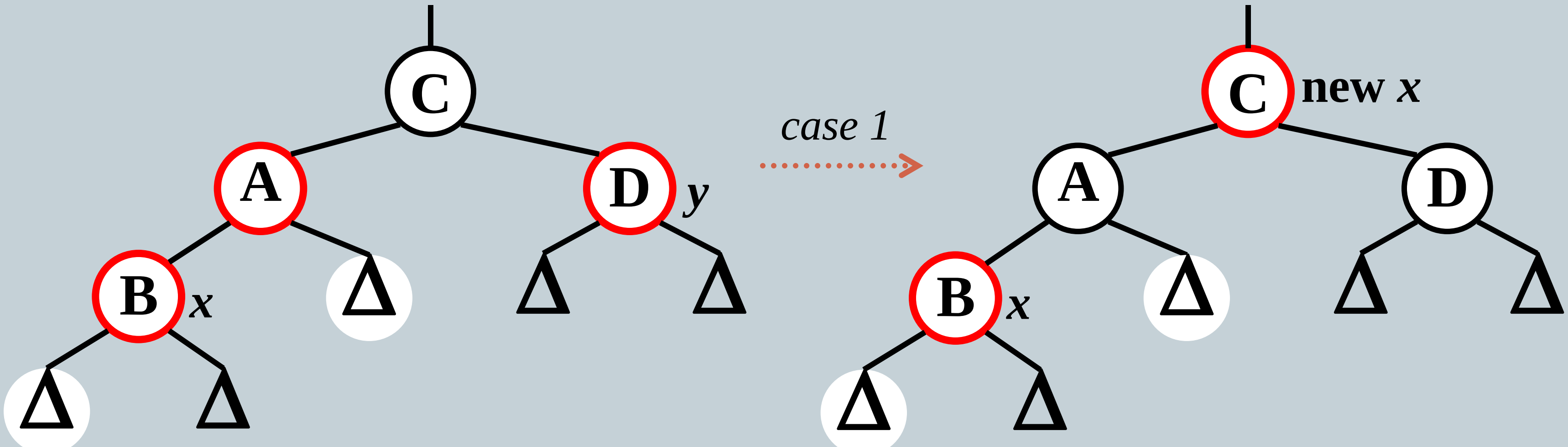


Change colors of some nodes, preserving #4: all downward paths have equal ***bh****.*
The while loop now continues with x’s grandparent as the new x

RB Insert: Case 1's symmetrical


```
if (y.color == RED)
    x.p.color = BLACK;
    y.color = BLACK;
    x.p.p.color = RED;
    x = x.p.p;
```

- Case 1: "uncle" is red
- In figures below, all Δ's are equal-black-height subtrees

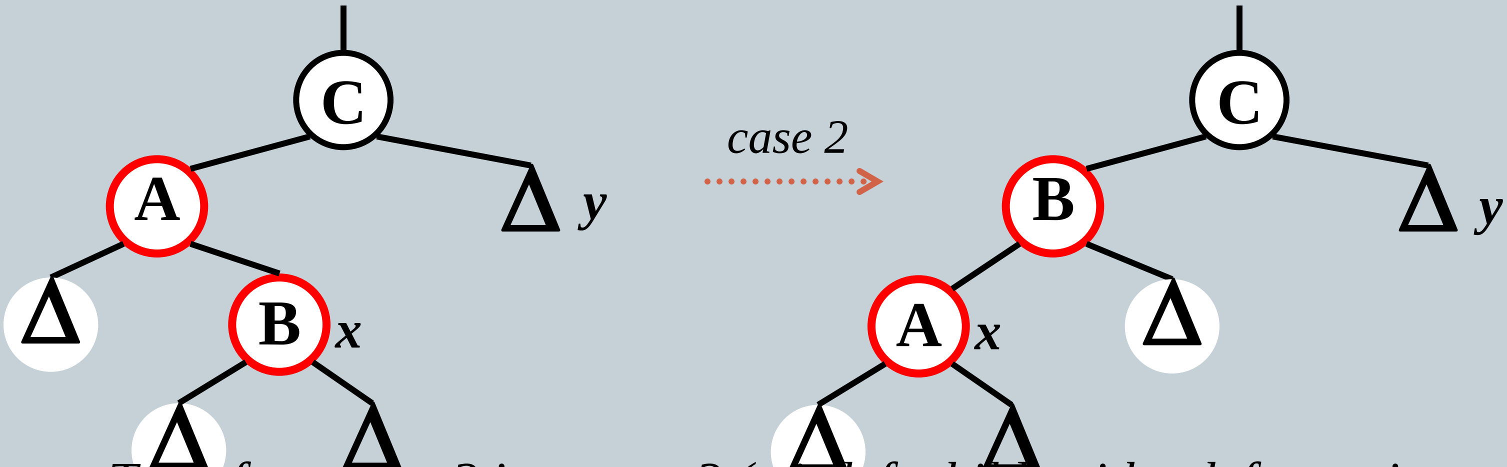


Same action whether x is a left or a right child

RB Insert: Case 2


```
if (x == x.p.right)
    x = x.p;
    leftRotate(x);
// continue with case 3 code
```

- Case 2:
  - “Uncle” is black
  - Node *x* is a right child
- Transform to case 3 via a left-rotation

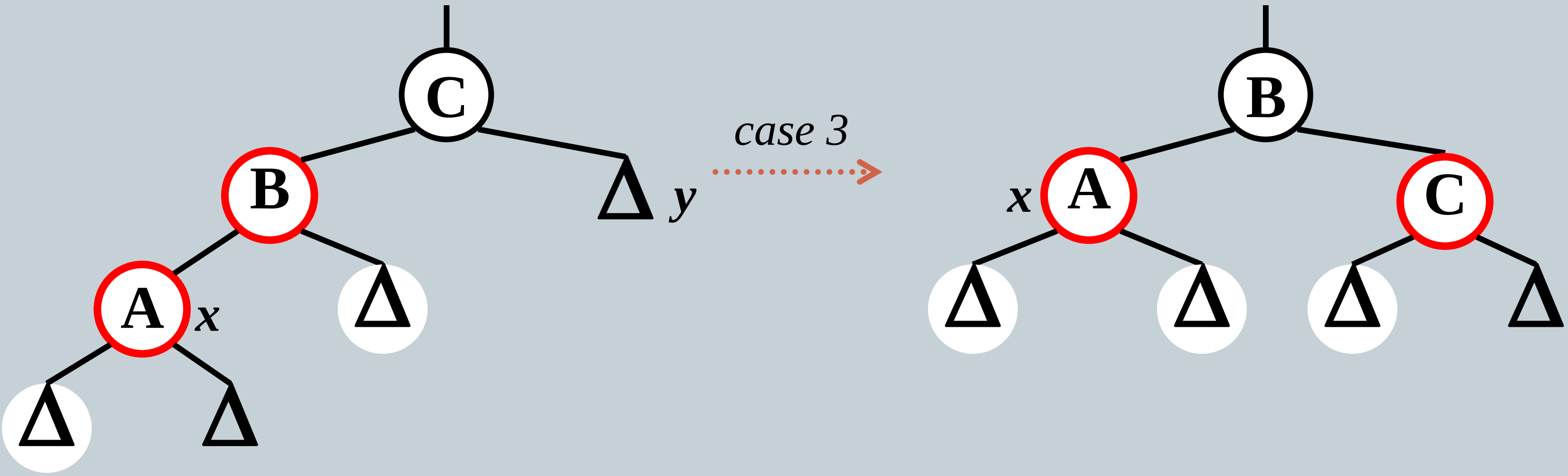


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property# 4: all downward paths contain same number of black nodes

RB Insert: Case 3


```
x.p.color = BLACK;
x.p.p.color = RED;
rightRotate(x.p.p);
```

- Case 3:
  - "Uncle" is black
  - Node *x* is a left child
- Change colors; rotate right



Perform some color changes and do a right rotation

Again, preserves property #4: all downward paths contain same number of black nodes

RB Insert: Cases 4-6

- Cases 1-3 hold if x's parent is a left child
- If x's parent is a right child, cases 4-6 are symmetric (swap left for right)

Practice Red Black Trees

Show the red-black trees that result after successively inserting the keys

a) 41, 38, 31, 12, 19, 42 and 45

b) 15, 10, 18, 8, 20, 22 and 5,

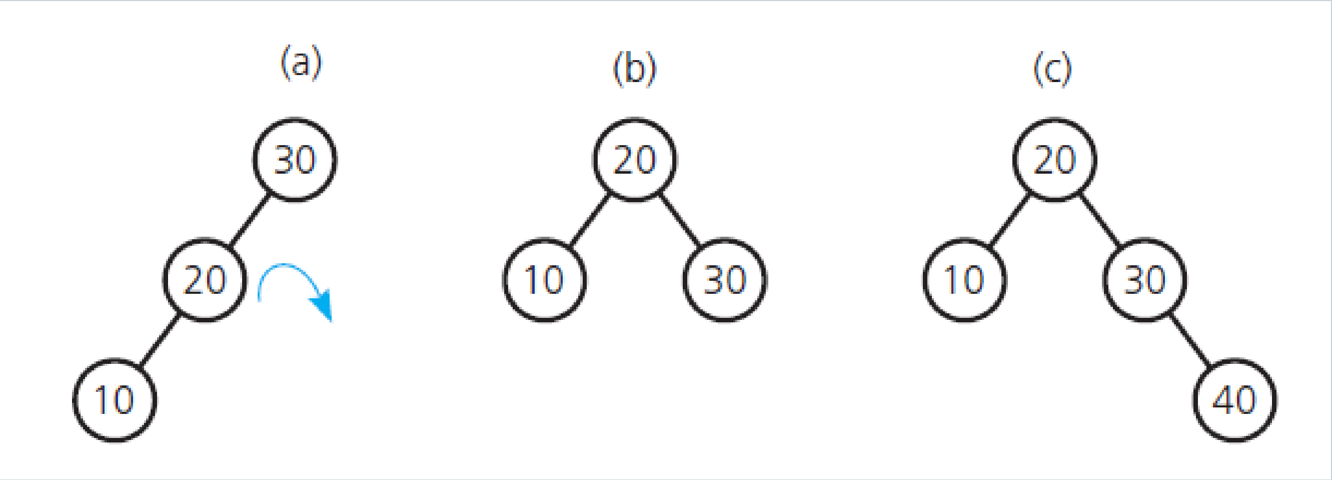
into an initially empty red-black tree

AVL Trees

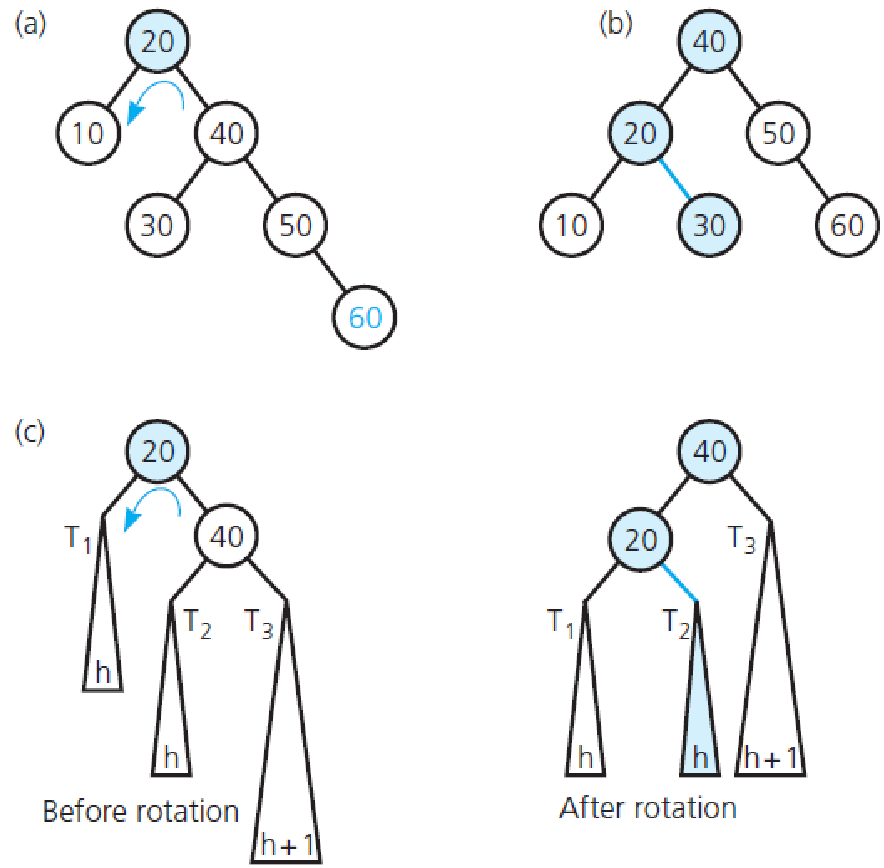
- Named for inventors, Adel'son-Vel'skii and Landis
- A balanced binary search tree
  - Maintains height close to the minimum
  - After insertion or deletion, check the tree is still AVL tree – determine whether any node in tree has left and right subtrees whose heights differ by more than 1
- Can search AVL tree almost as efficiently as minimum-height binary search tree.

AVL Trees

- (a) An unbalanced binary search tree;
  (b) a balanced tree after rotation;
  (c) a balanced tree after insertion

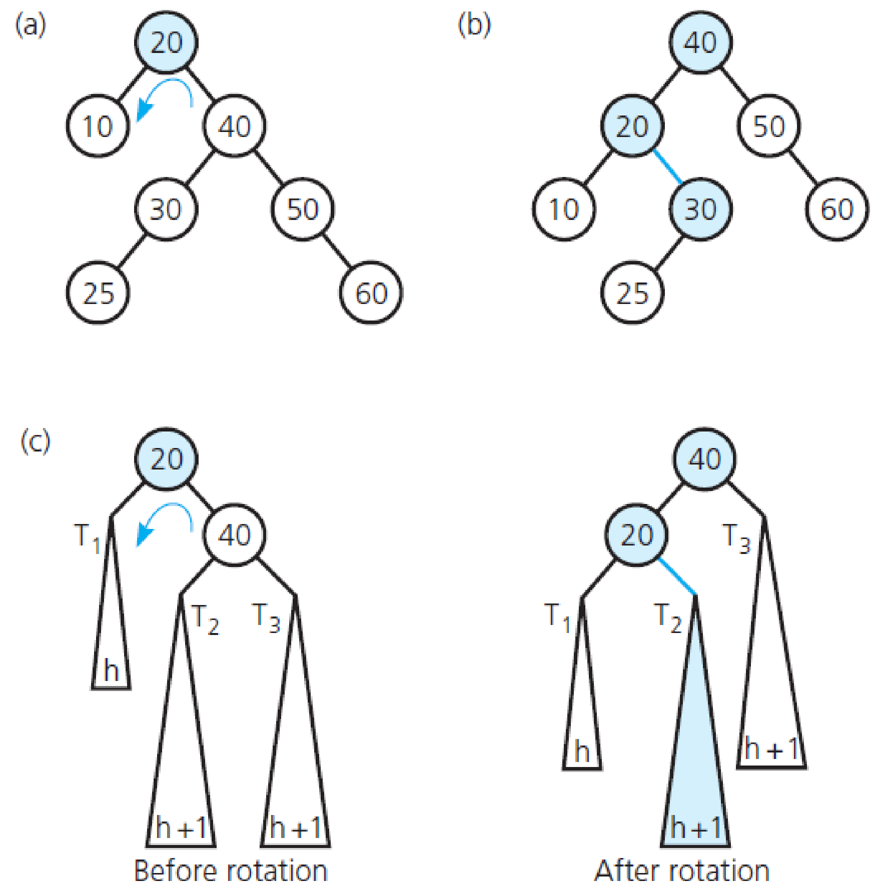


AVL Trees



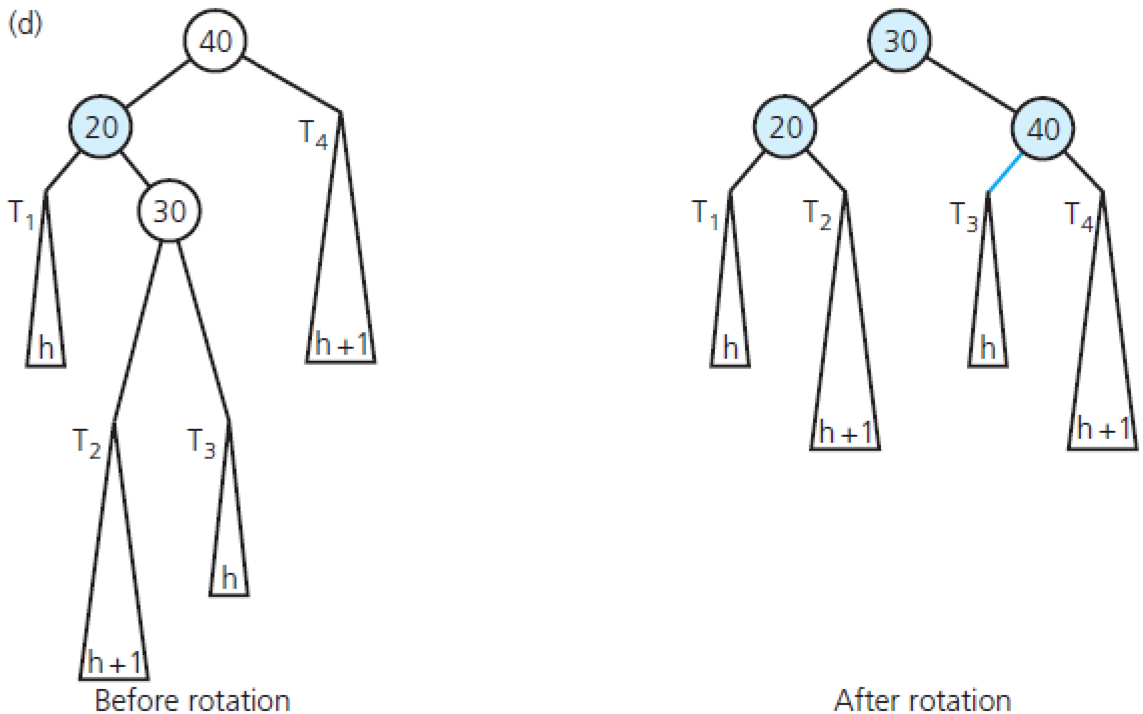
- (a) Before; (b) and after a single left rotation that decreases the tree's height; (c) the rotation in general

AVL Trees



- (a) Before; (b) and after a single left rotation that does not affect the tree's height; (c) the rotation in general

AVL Trees



- (d) the double rotation in general