Introduction to Algorithms

Balanced Search Trees

Red-Black Trees

- *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 Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every "real" node has 2 children
  3. If a node is red, both children are black
     - Note: can't have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

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 *h* and bh values

Height of Red-Black Trees

- *What is the minimum black-height of a node with height h?*
- A: a height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2\lg(n + 1)$
- *How do you suppose we'll prove this?*

RB Trees: Proving Height Bound

- Prove: n-node RB tree has height $h \leq 2 \lg(n+1)$
- Claim: A subtree rooted at a node x contains at least $2^{\text{bh}(x)} - 1$ internal nodes
  - Proof by induction on height h
  - Base step: x has height 0 (i.e., NULL leaf node)
    - *What is bh(x)?*

RB Trees: Proving Height Bound

- Prove: n-node RB tree has height $h \leq 2 \lg(n+1)$
- Claim: 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

- 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 $bh(x)-1$ *(Why?)*
    - 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
    - Thus subtree at x contains
      $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1$
      $= 2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$ nodes

RB Trees: Proving Height Bound

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

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

$n \geq 2^{h/2} - 1$ *(Why?)*

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

$h \leq 2 \lg(n + 1)$ *(Why?)*

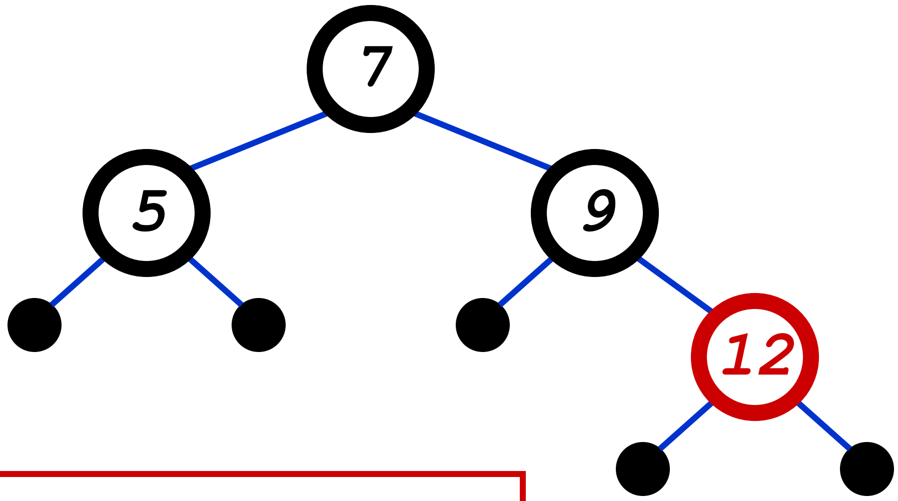
Thus $h = \mathrm{O}(\lg n)$

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

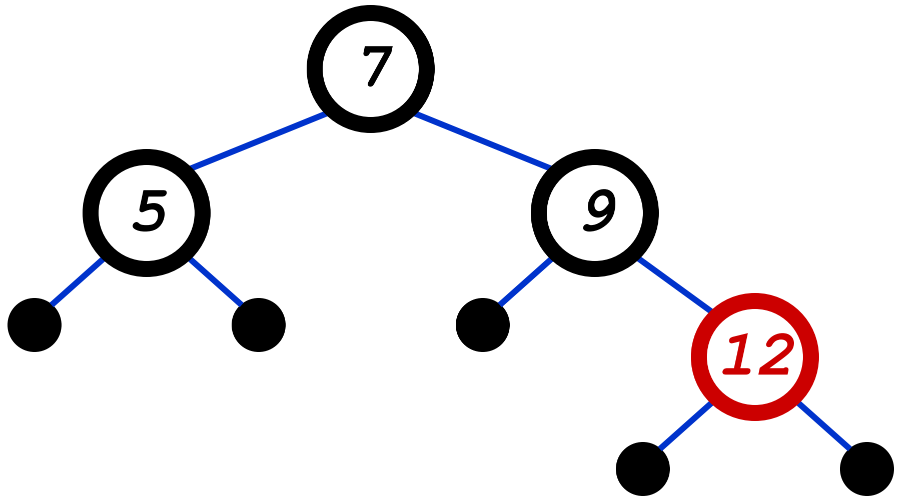
- *Color this tree:*



Red-black 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

Red-Black Trees: The Problem With Insertion

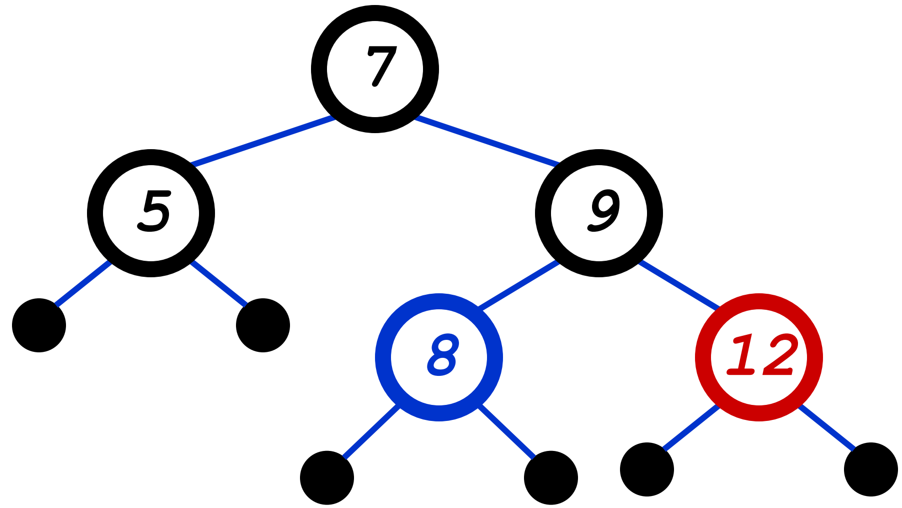
- Insert 8
  - *Where does it go?*



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: The Problem With Insertion

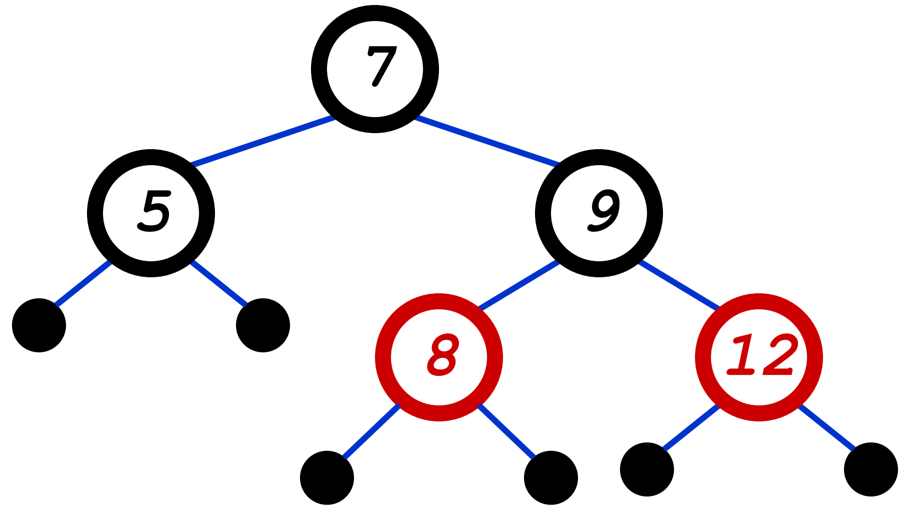
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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: The Problem With Insertion

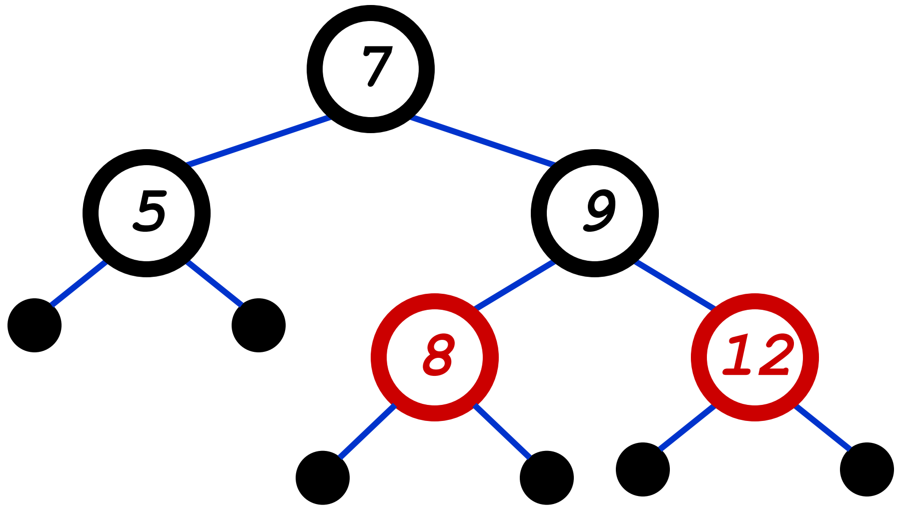
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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: The Problem With Insertion

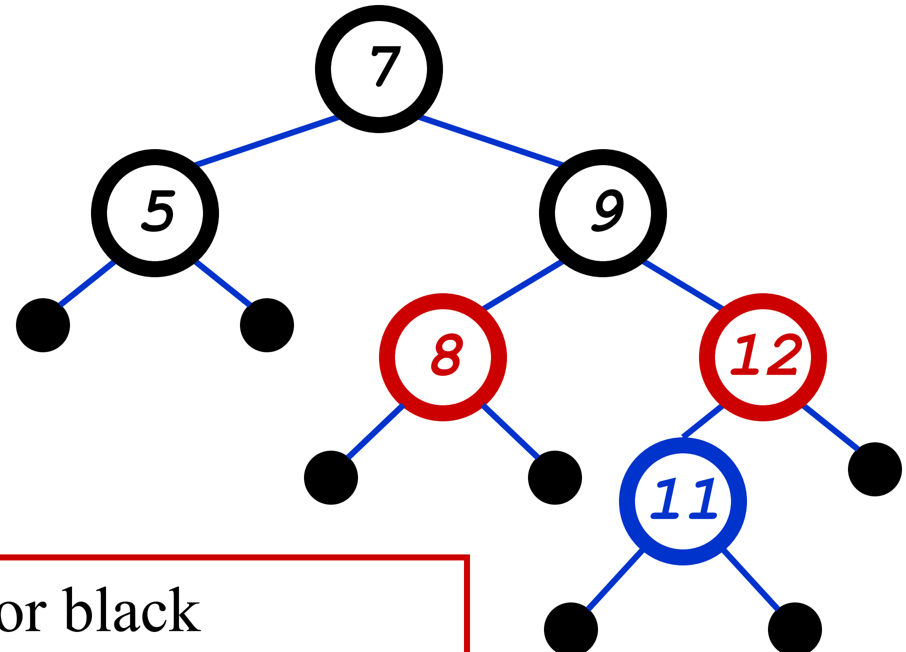
- Insert 11
  - *Where does it go?*



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: The Problem With Insertion

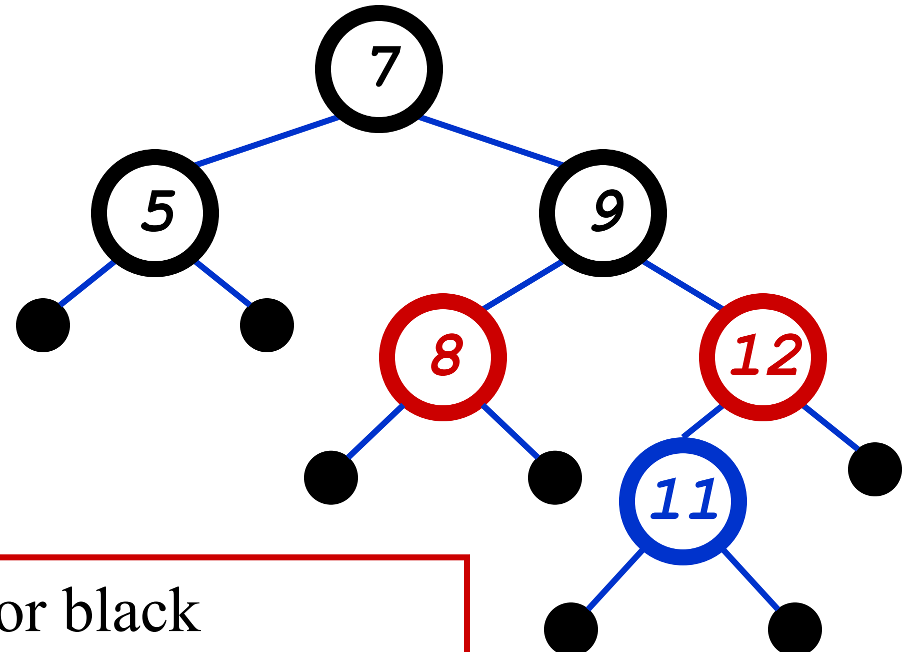
- Insert 11
  - *Where does it go?*
  - *What color?*



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: The Problem With Insertion

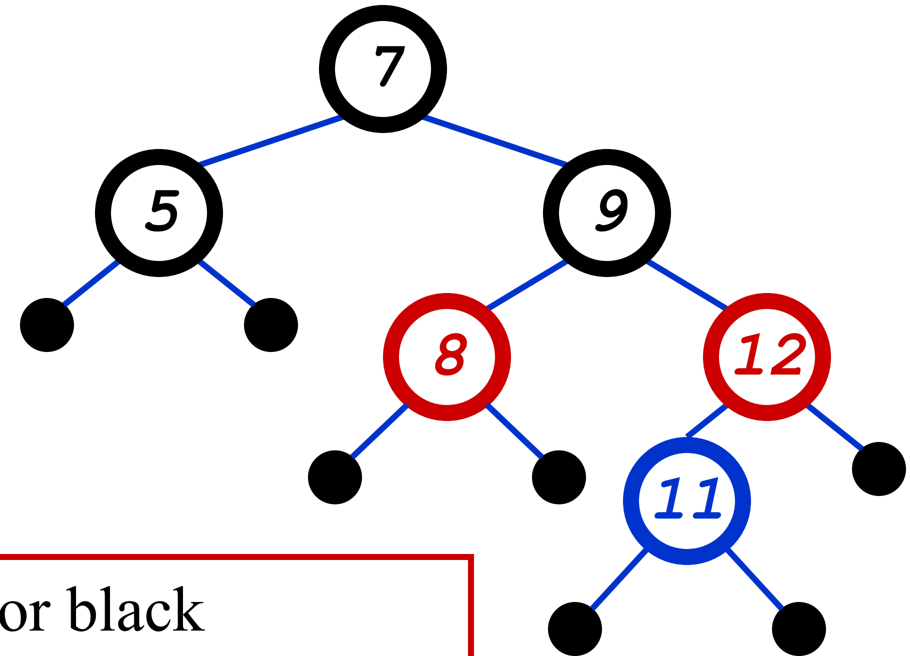
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)



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: The Problem With Insertion

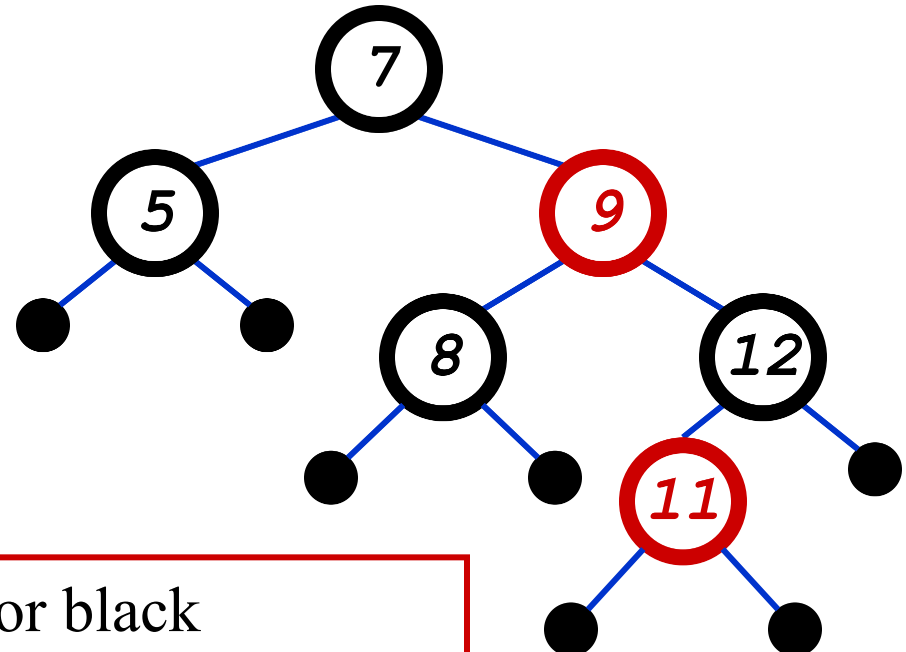
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can’t be red! (#3)
    - Can’t be black! (#4)



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: The Problem With Insertion

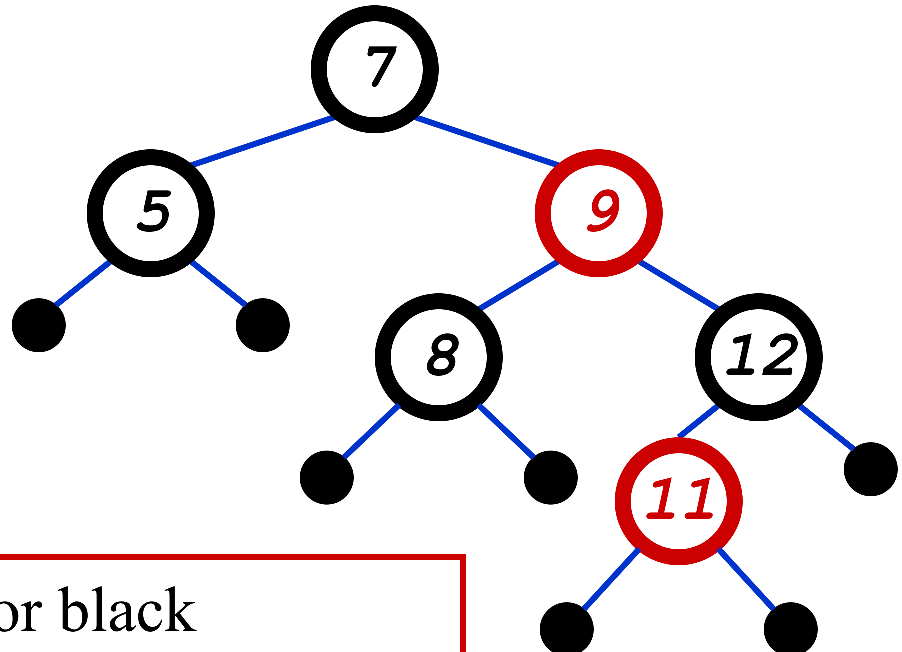
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution: recolor the tree



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: The Problem With Insertion

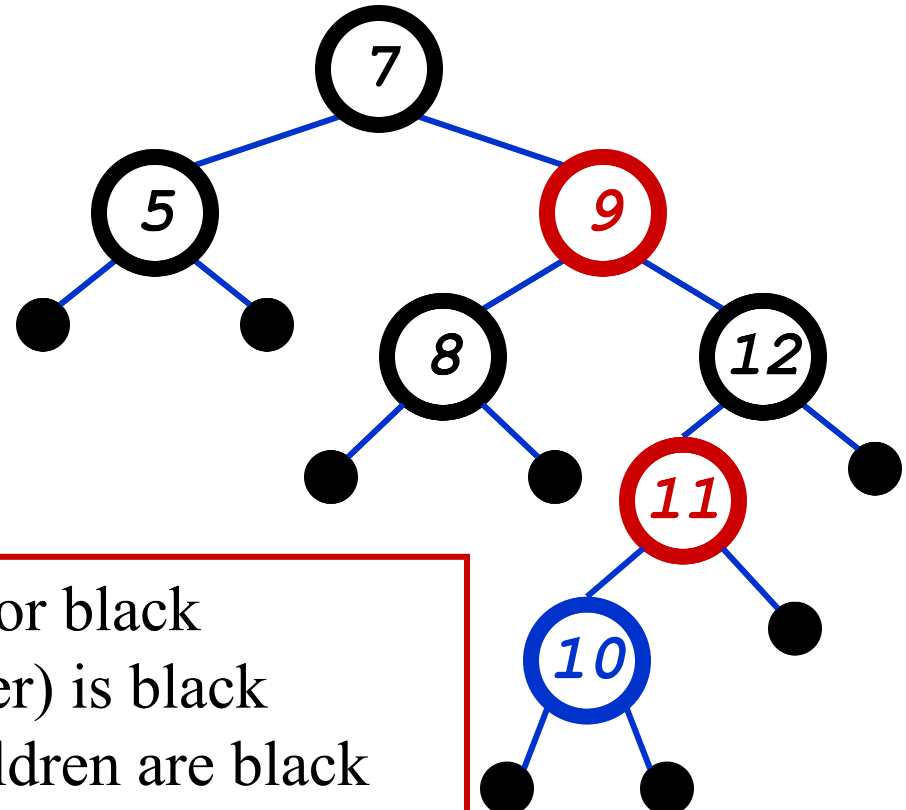
- Insert 10
  - *Where does it go?*



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: The Problem With Insertion

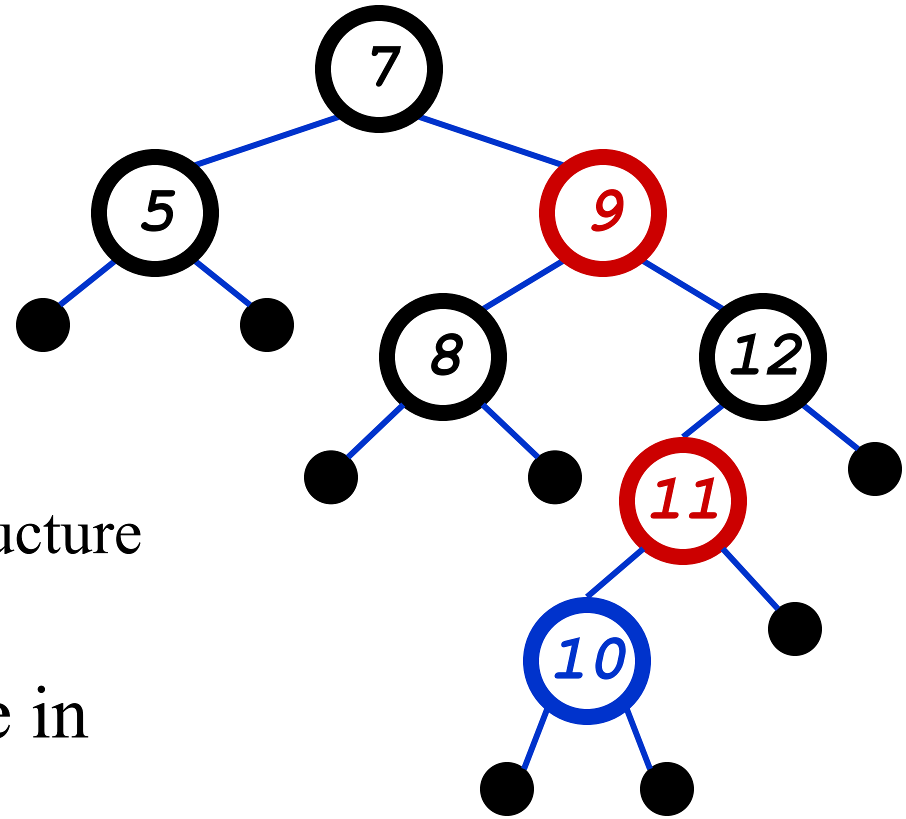
- Insert 10
  - *Where does it go?*
  - *What color?*



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

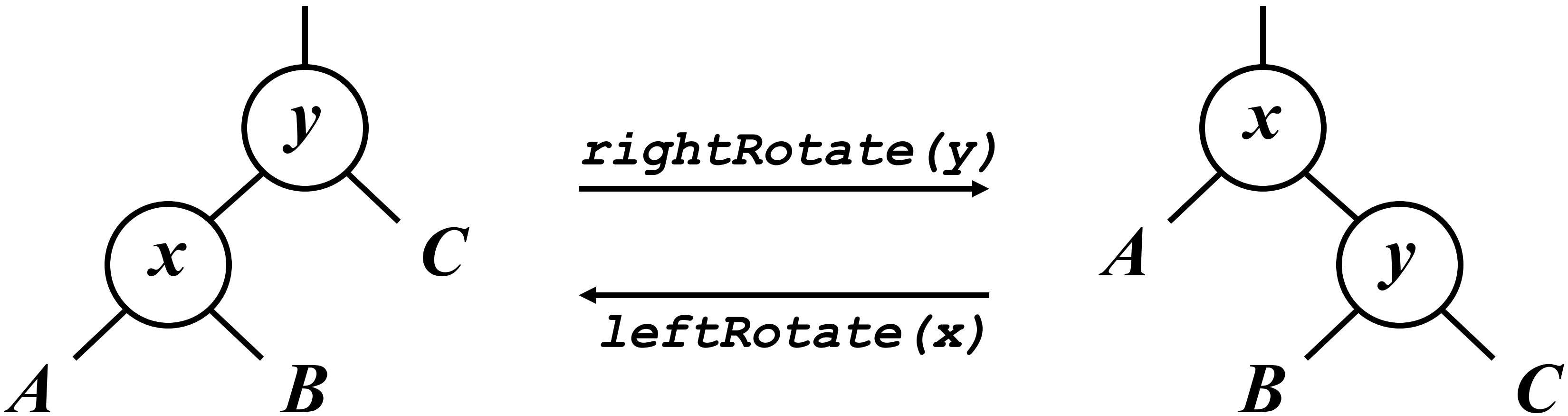
Red-Black Trees: The Problem With Insertion

- 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



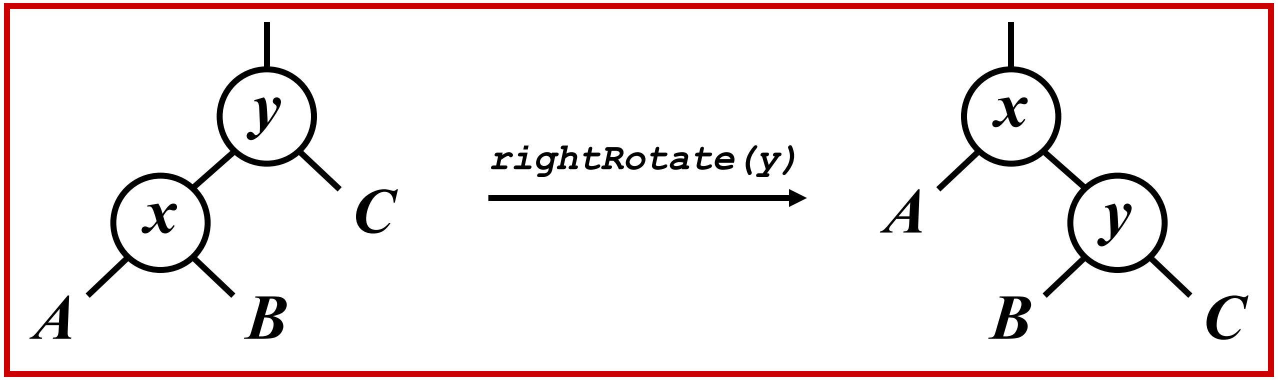
RB Trees: Rotation

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



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

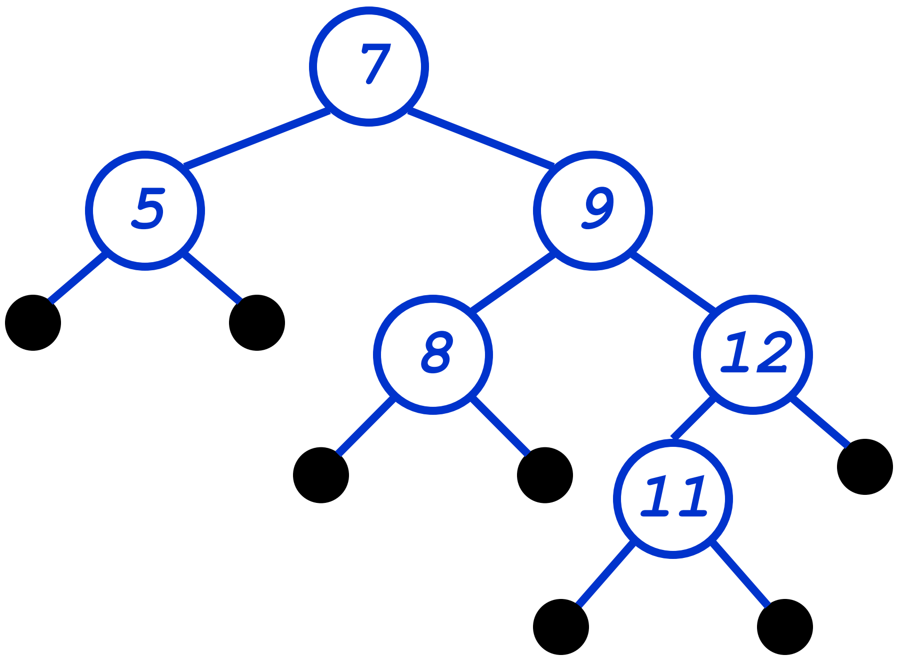
RB Trees: Rotation



- 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?*

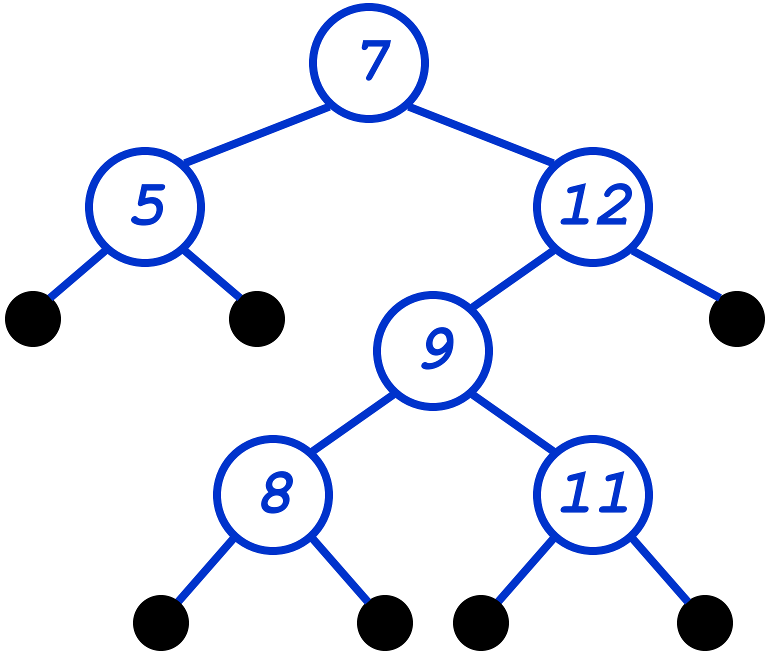
Rotation Example

- 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 p[x] red)
    - If so, move violation up tree until a place is found where it can be fixed
  - Total time will be O(lg n)

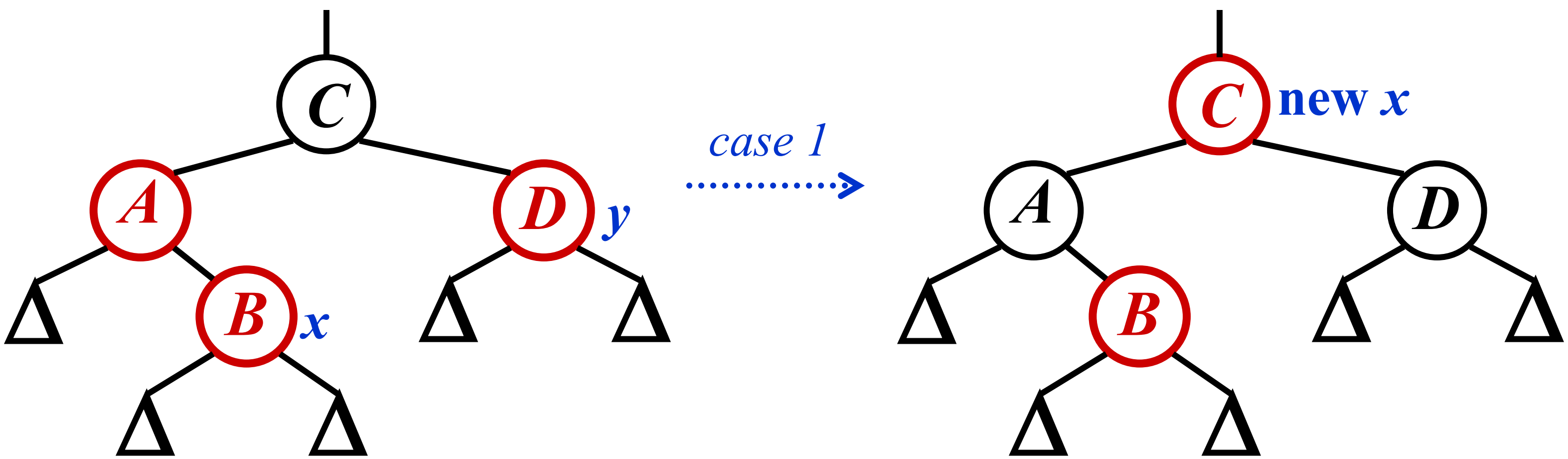
```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #3 up tree, maintaining #4 as invariant:
  while (x!=root && x->p->color == RED)
  if (x->p == x->p->p->left)
      y = x->p->p->right;
      if (y->color == RED)
          x->p->color = BLACK;
          y->color = BLACK;                  Case 1
          x->p->p->color = RED;
          x = x->p->p;
      else    // y->color == BLACK
          if (x == x->p->right)
              x = x->p;                      Case 2
              leftRotate(x);
          x->p->color = BLACK;
          x->p->p->color = RED;              Case 3
          rightRotate(x->p->p);
  else    // x->p == x->p->p->right
      (same as above, but with
       "right" & "left" exchanged)
```

```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #3 up tree, maintaining #4 as invariant:
  while (x!=root && x->p->color == RED)
  if (x->p == x->p->p->left)
      y = x->p->p->right;
      if (y->color == RED)
          x->p->color = BLACK;
          y->color = BLACK;              Case 1: uncle is RED
          x->p->p->color = RED;
          x = x->p->p;
      else    // y->color == BLACK
          if (x == x->p->right)
              x = x->p;                  Case 2
              leftRotate(x);
          x->p->color = BLACK;
          x->p->p->color = RED;          Case 3
          rightRotate(x->p->p);
  else    // x->p == x->p->p->right
      (same as above, but with
       "right" & "left" exchanged)
```

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

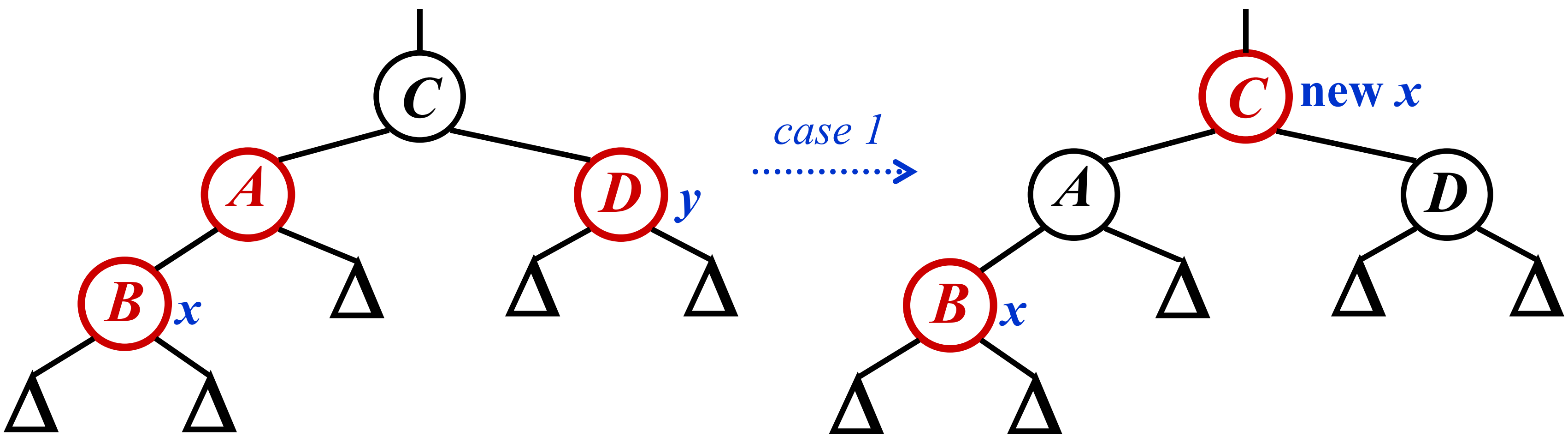


Change colors of some nodes, preserving #4: all downward paths have equal b.h.
The while loop now continues with x's grandparent as the new x

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

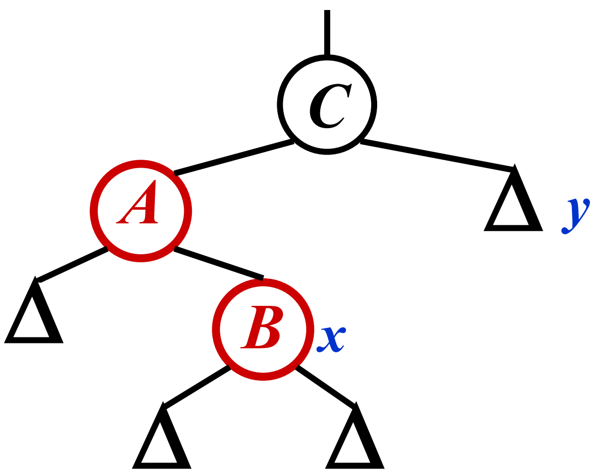


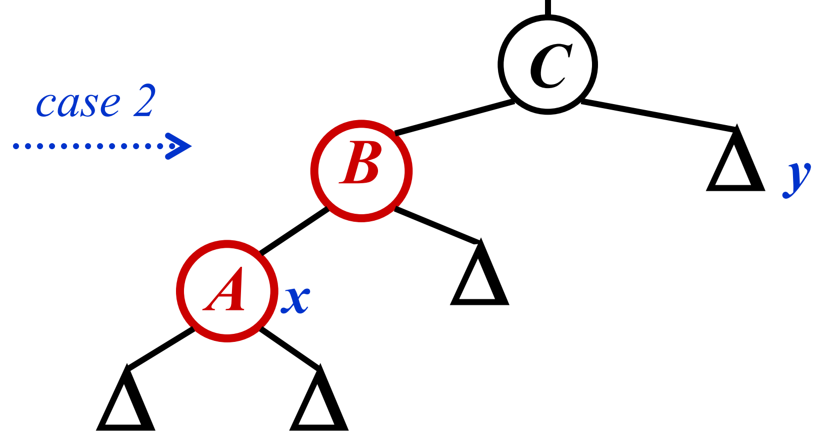
Same action whether x is a left or a right child

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



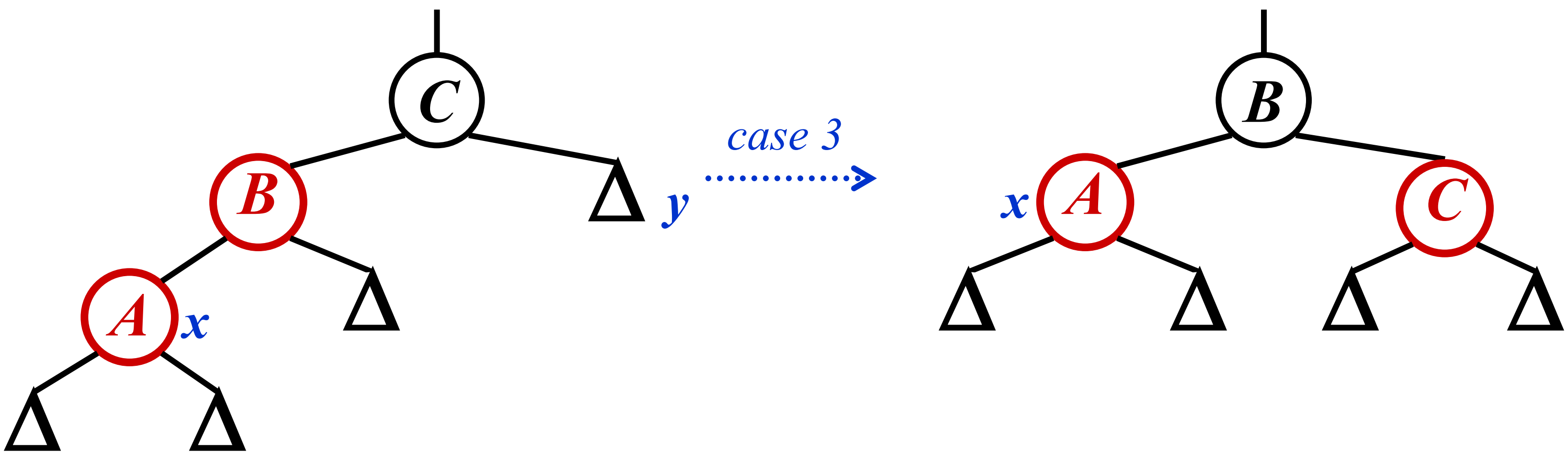


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

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)

Red-Black Trees: Deletion

- And you thought insertion was tricky…
- We will not cover RB delete in class
  - You should read section 14.4 on your own
  - Read for the overall picture, not the details

AVL Tree

- Invented by Georgy **A**delson-**V**elsky and Evgenii **L**andis in 1962
- A balanced binary search tree where the height of the two subtrees (children) of a node differs by at most one. Look-up, insertion, and deletion are O(log n), where n is the number of nodes in the tree.

Definition of Height (reminder)

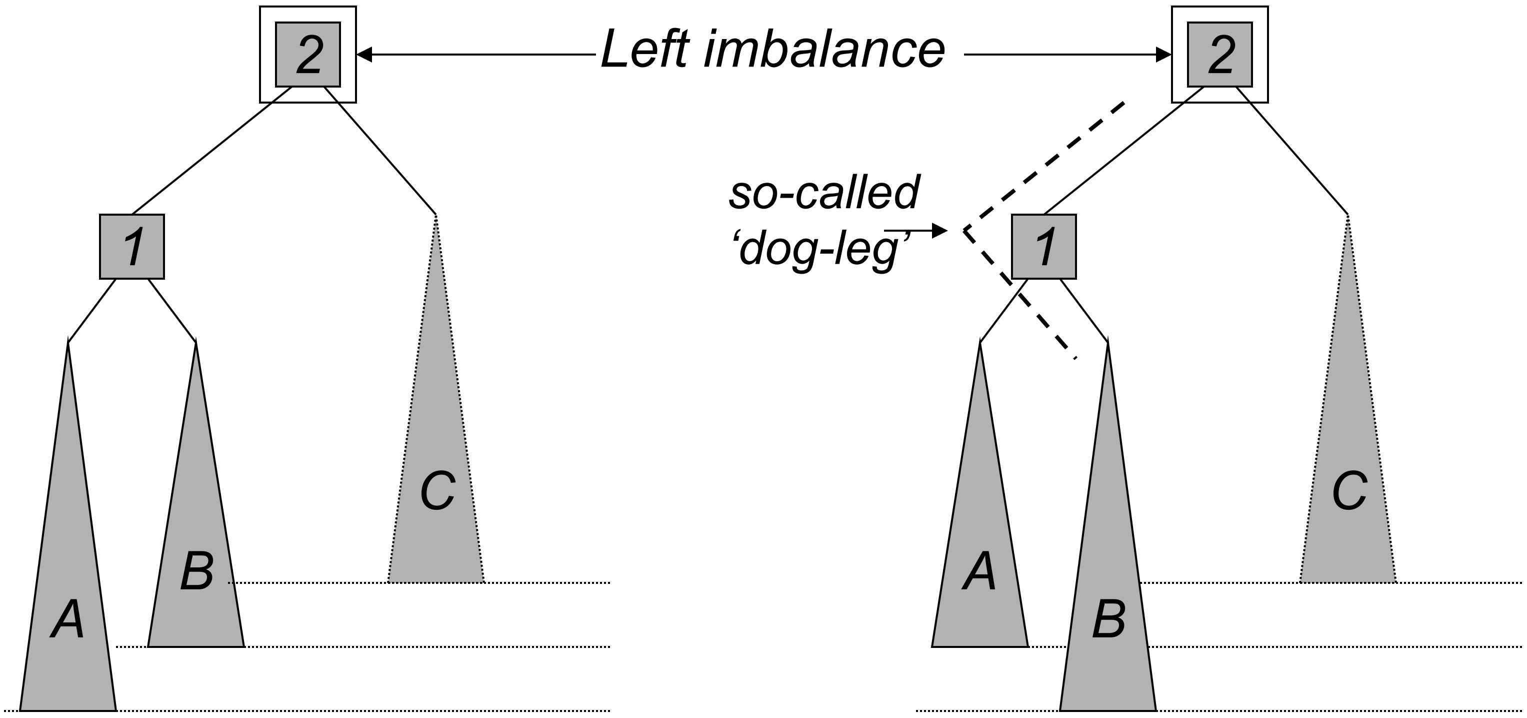
- Height: the length of the longest path from a node to a leaf.
  - All leaves have a height of 0
  - An empty tree has a height of –1

The Insertion Problem

- Unless keys appear in just the right order, imbalance will occur
- It can be shown that there are only two possible types of imbalance:
  - Left-left (or right-right) imbalance
  - Left-right (or right-left) imbalance
  - The right-hand imbalances are the same, by symmetry

The Two Types of Imbalance

- Left-left (right-right)
- Left-right (right-left)



There are no other possibilities for the left (or right) subtree

Localising The Problem

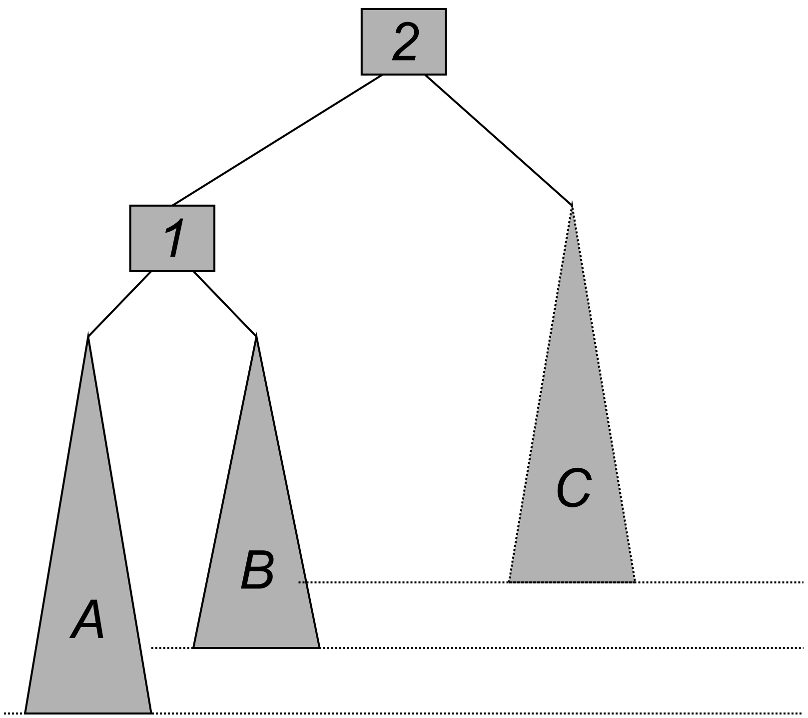
Two principles:

- Imbalance will only occur on the path from the inserted node to the root (only these nodes have had their subtrees altered - local problem)
- Rebalancing should occur at the ***deepest unbalanced node*** (local solution too)

Left(left) imbalance (1)

[and right(right) imbalance, by symmetry]

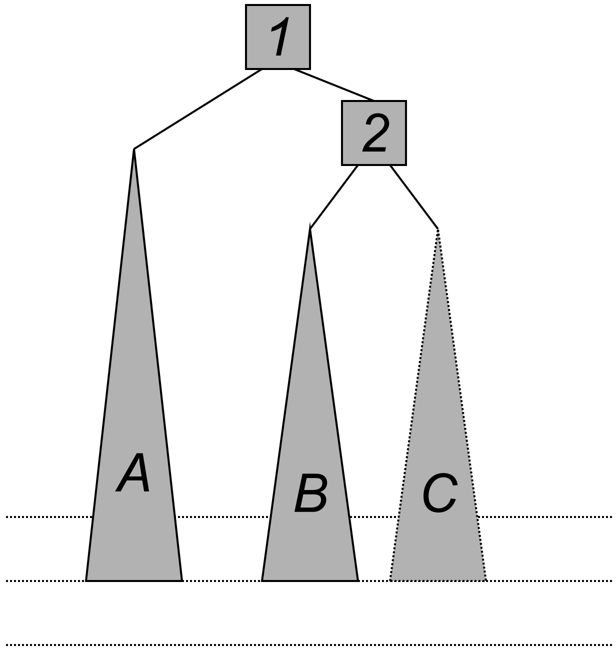
- B and C have the same height
- A is one level higher
- Therefore make 1 the new root, 2 its right child and B and C the subtrees of 2
- Note the levels



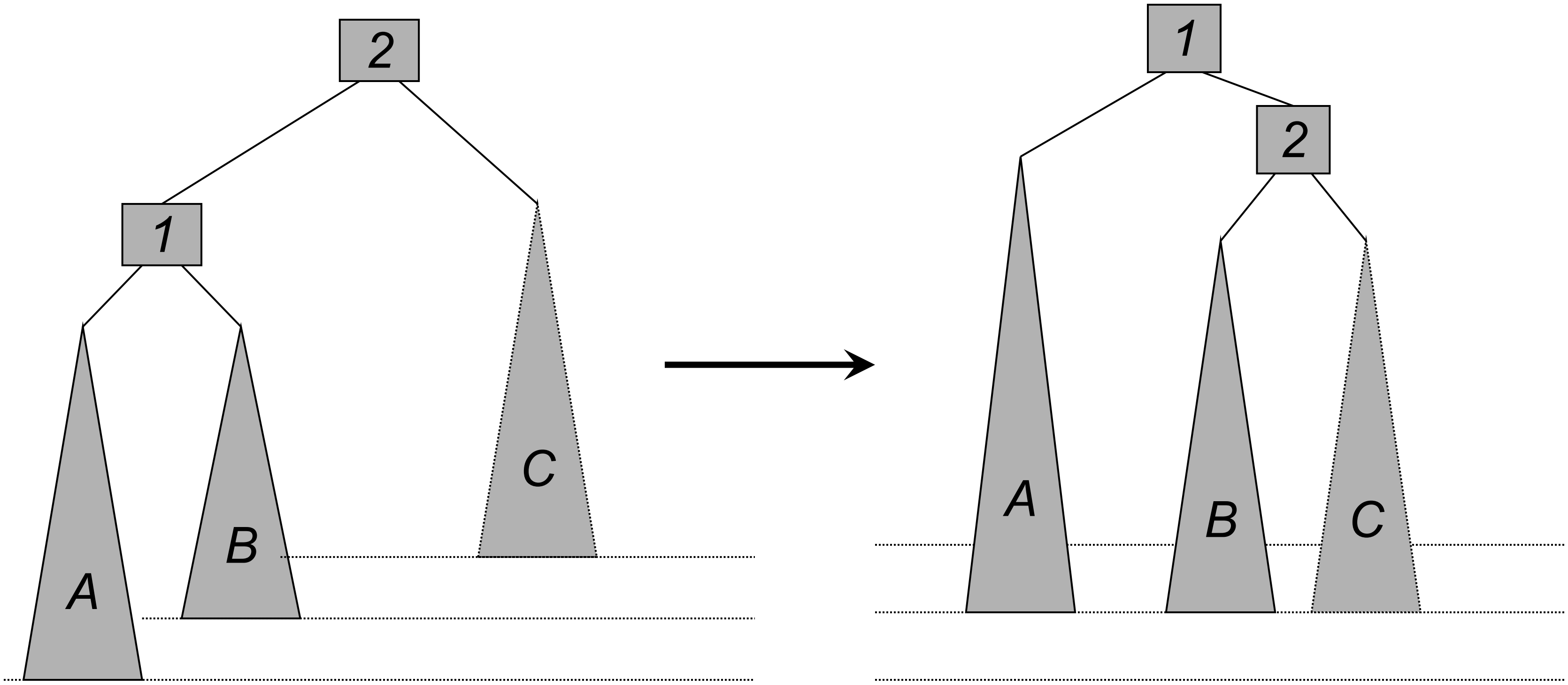
Left(left) imbalance (2)
[and right(right) imbalance, by symmetry]

- B and C have the same height
- A is one level higher
- Therefore make 1 the new root, 2 its right child and B and C the subtrees of 2
- Result: a more balanced and legal AVL tree

- Note the levels

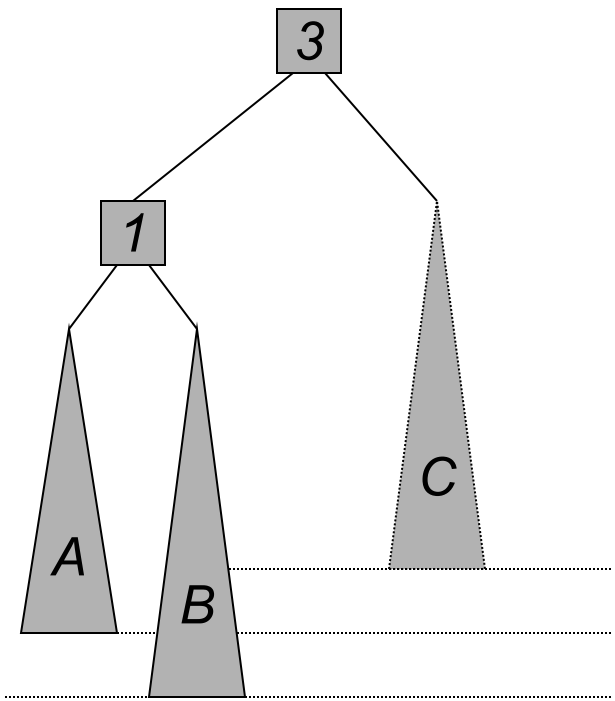


Single rotation



Left(right) imbalance (1)
[and right(left) imbalance by symmetry]

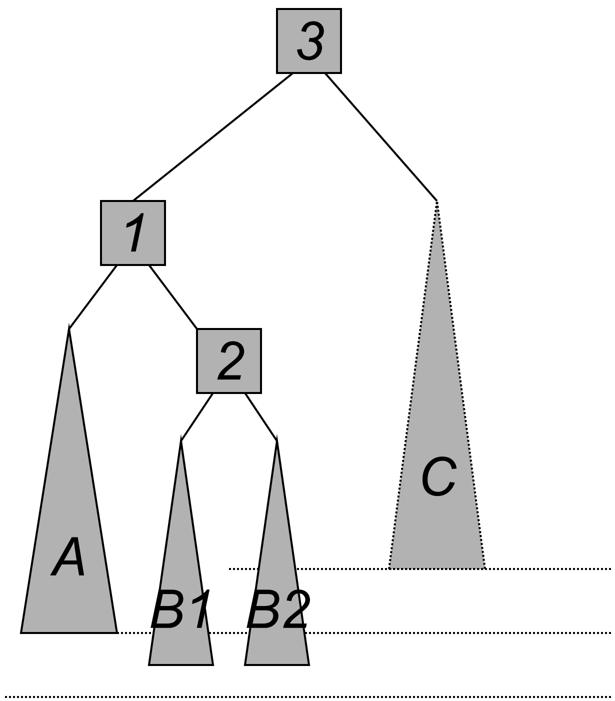
- Can't use the left-left balance trick - because now it's the *middle subtree*, i.e. B, that's too deep.
- Instead consider what's inside B...



Left(right) imbalance (2)

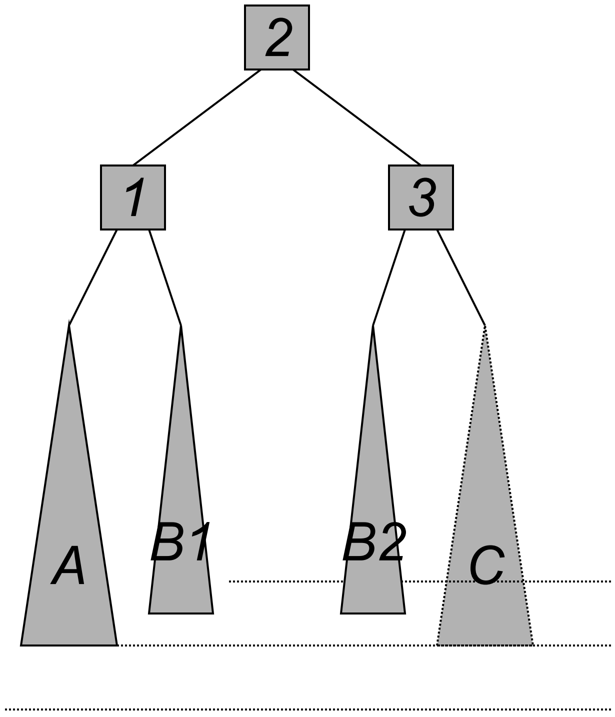
[and right(left) imbalance by symmetry]

- B will have two subtrees containing at least one item (just added
- We do not know which is too deep - set them both to 0.5 levels below subtree A



Left(right) imbalance (3)
[and right(left) imbalance by symmetry]

- Neither 1 nor 3 worked as root node so make 2 the root
- Rearrange the subtrees in the correct order
- No matter how deep B1 or B2 (+/- 0.5 levels) we get a legal AVL tree again



Double Rotation