Red Black Trees

Prof. Zheng Zhang

Harbin Institute of Technology, Shenzhen

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. The root is always black
  4. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path
  5. Every path from node to descendent leaf contains the same number of black nodes

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. The root is always black
  4. If a node is *red*, both children are black
  5. Every path from node to descendent leaf contains the same number of black nodes
- *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 <u>internal nodes</u> 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 $2^{bh(x)} - 1$ internal nodes
  - Proof by induction on height *h*
  - Base step: *x* has height *0* (*i.e.*, NULL *leaf* node)

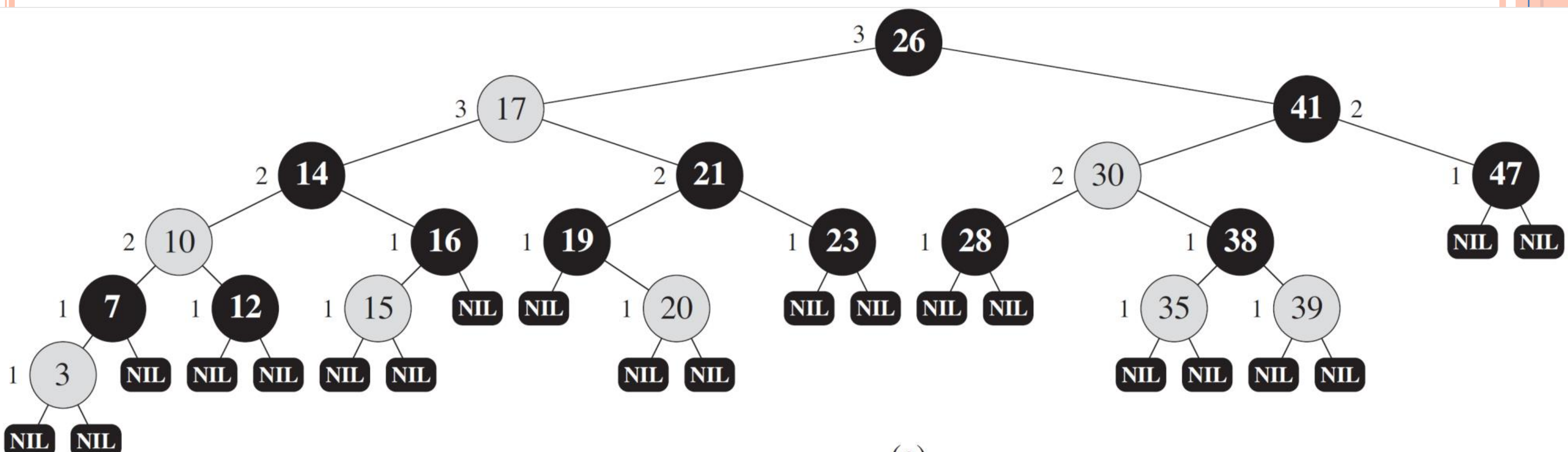
    What is bh(x)?

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



(a)

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 at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$$

$$= 2 \bullet 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1 \text{ nodes}$$

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 $h = O(\lg n)$

Property 4: **If a node is red, both children are black**

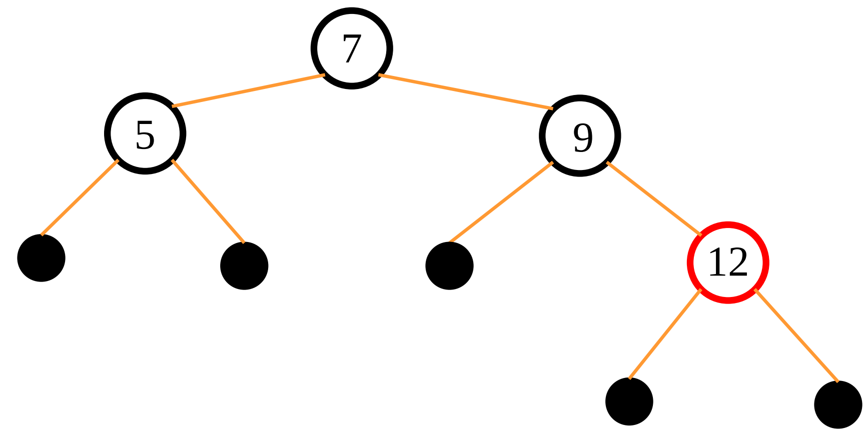
Property 5: Every path from node to descendent leaf contains the same number of black nodes.

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 <u>need special care</u> since they modify tree

Red-Black Trees: An Example

- *Color this tree:*

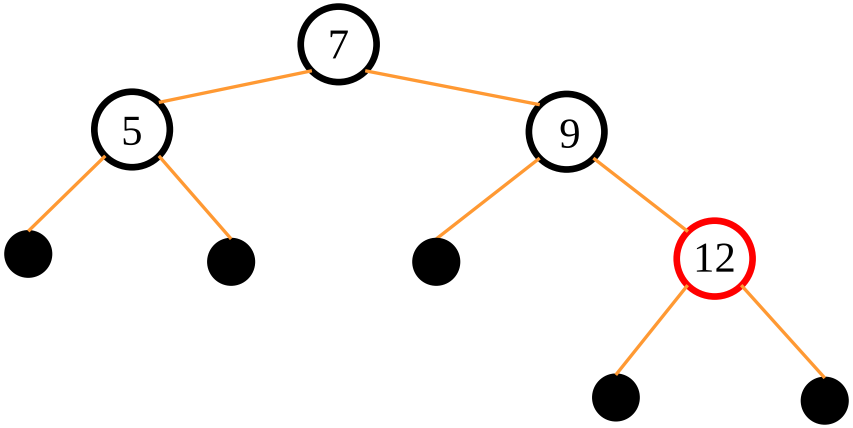


Red-black properties:

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

Red-Black Trees
The Problem With Insertion

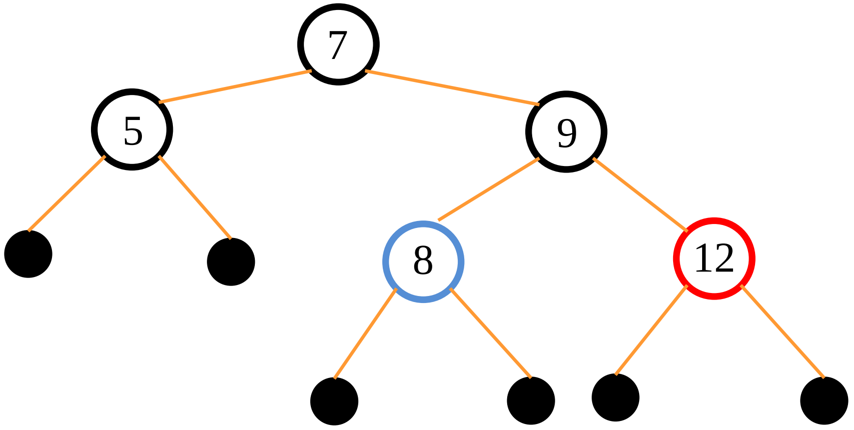
- *Insert 8:*
  - *Where does it go?*



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

Red-Black Trees
The Problem With Insertion

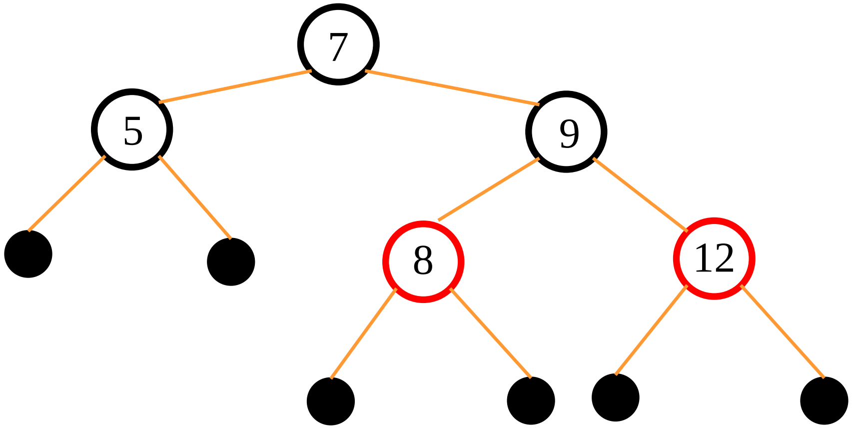
- *Insert 8:*
  - *Where does it go?*
  - *What color should it be?*



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

Red-Black Trees
The Problem With Insertion

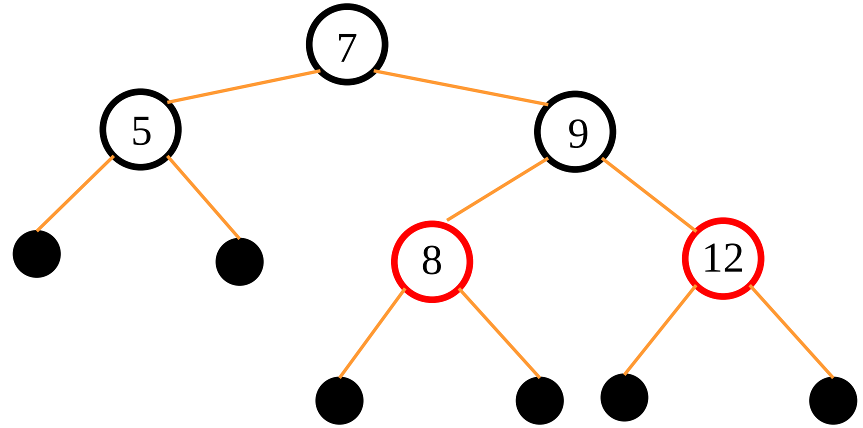
- *Insert 8:*
  - *Where does it go?*
  - *What color should it be?*



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

Red-Black Trees
The Problem With Insertion

- *Insert 11:*
  - *Where does it go?*

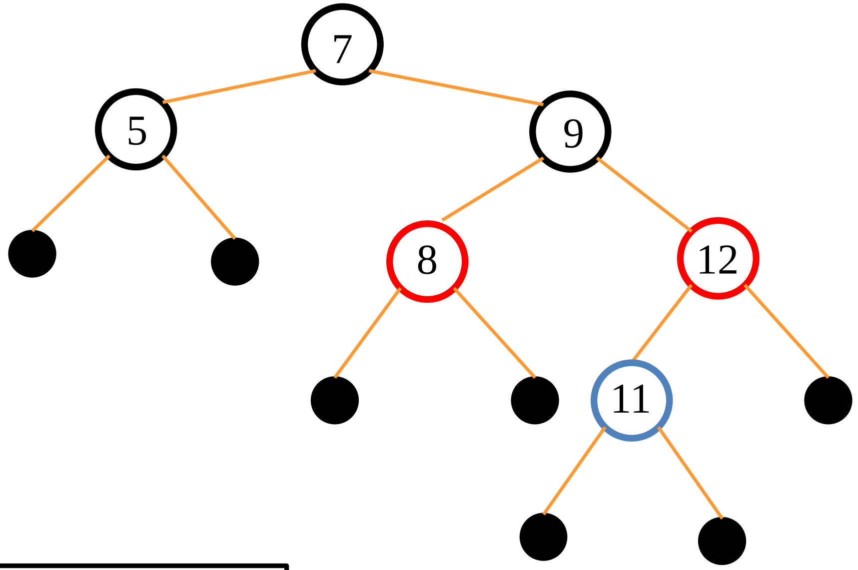


Red-black properties:

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

Red-Black Trees
The Problem With Insertion

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

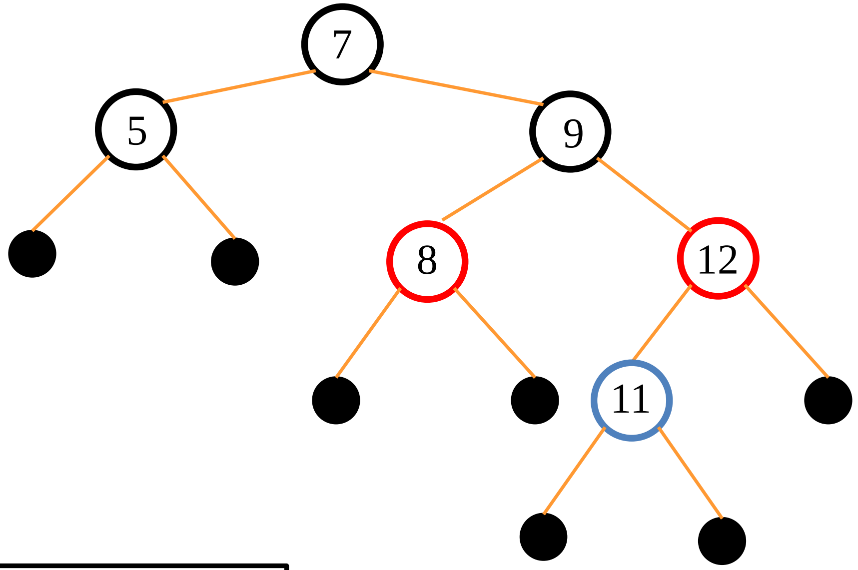


Red-black properties:

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

Red-Black Trees
The Problem With Insertion

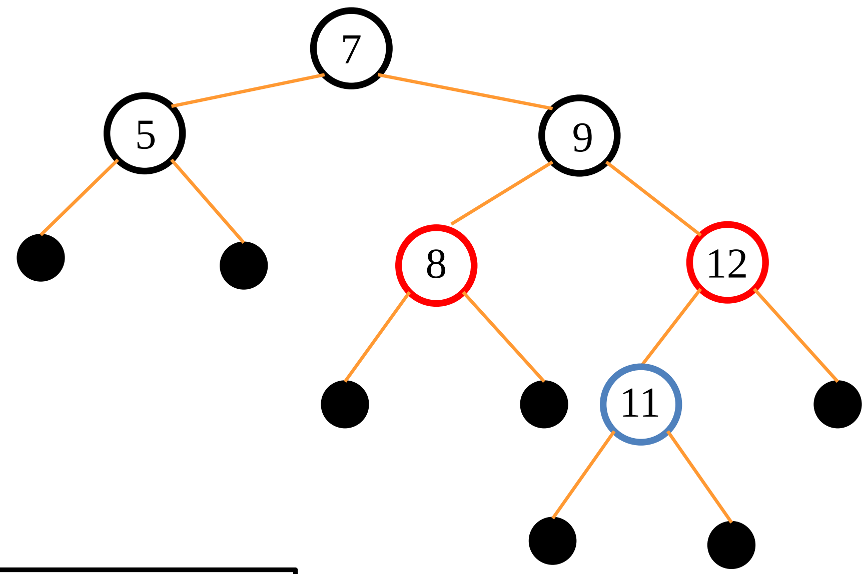
- *Insert 11:*
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#4)



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

RED-BLACK TREES
THE PROBLEM WITH INSERTION

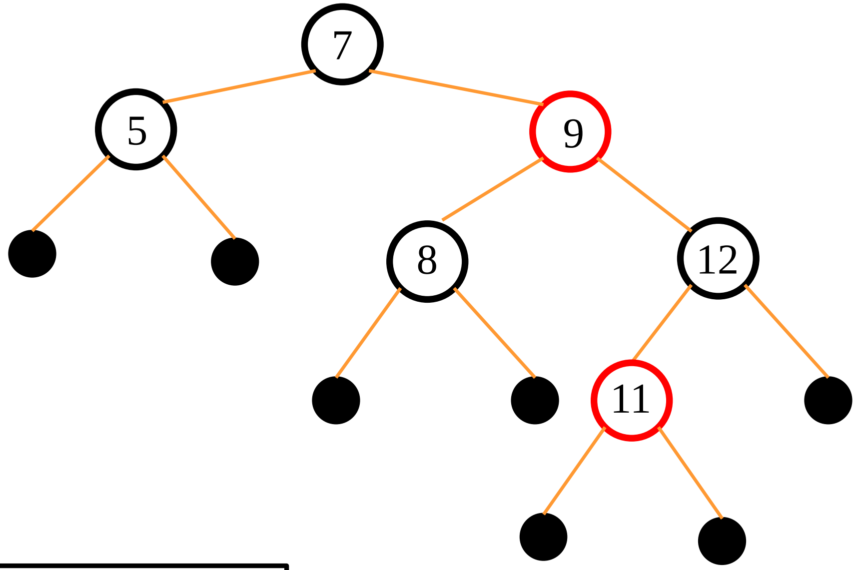
- *Insert 11:*
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#4)
    - Can't be black! (#5)



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

Red-Black Trees
The Problem With Insertion

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

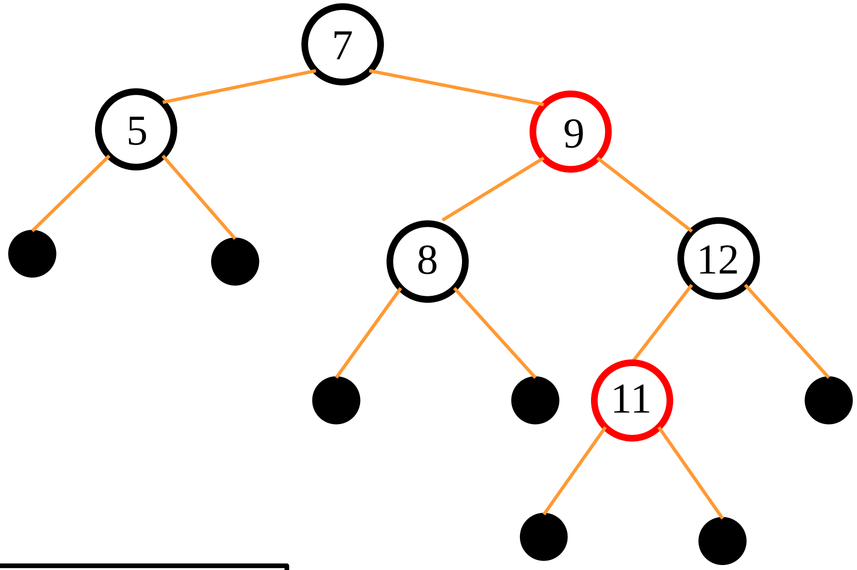


Red-black properties:

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

Red-Black Trees
The Problem With Insertion

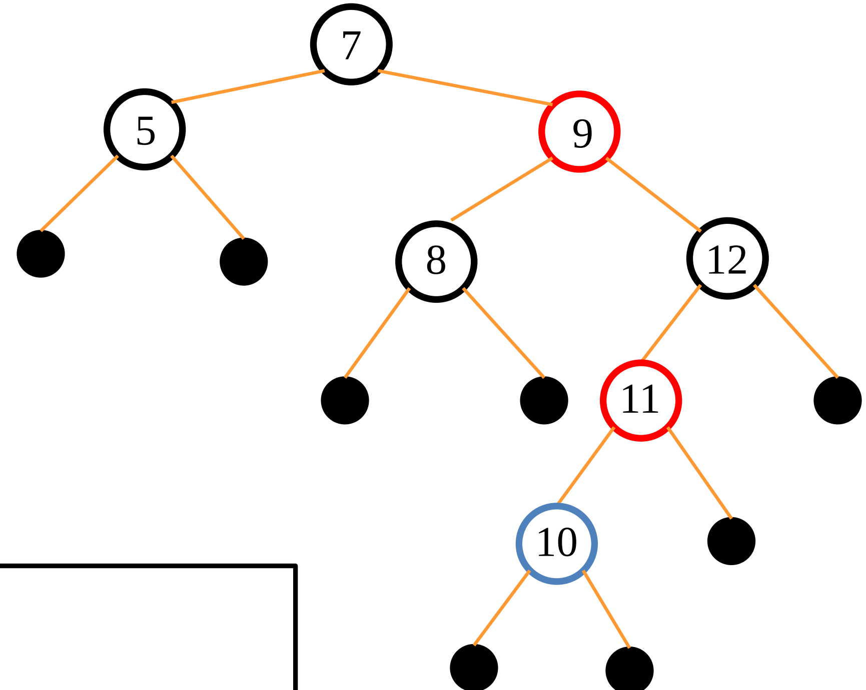
- *Insert 10:*
  - *Where does it go?*



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

Red-Black Trees
The Problem With Insertion

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

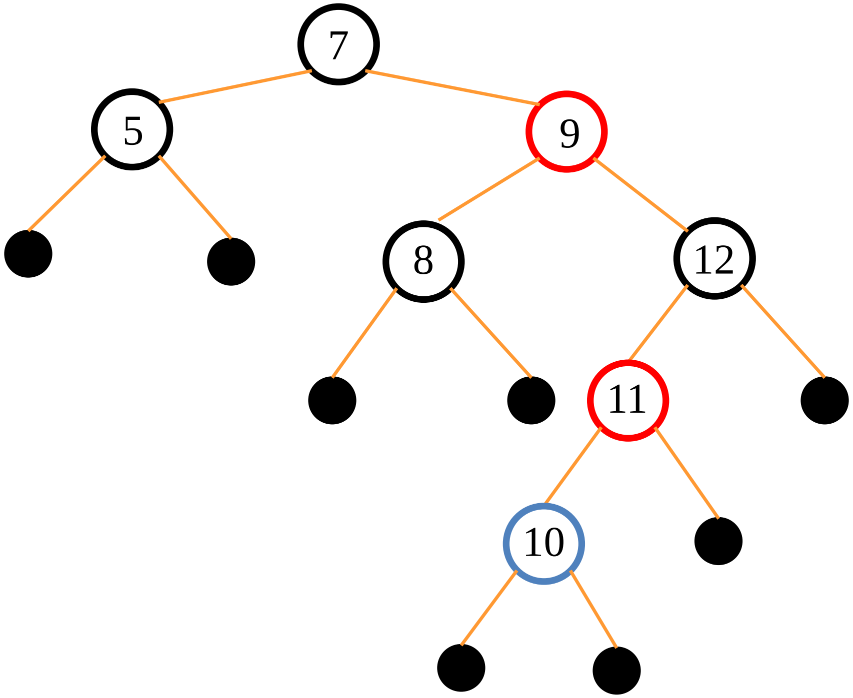


Red-black properties:

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

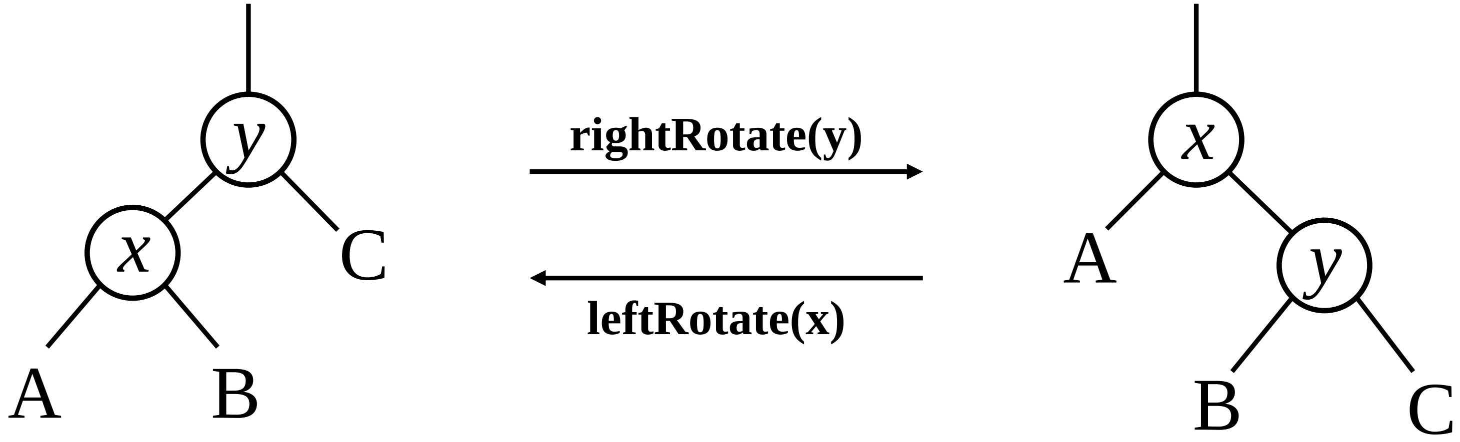
Red-Black Trees
The Problem With Insertion

- *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



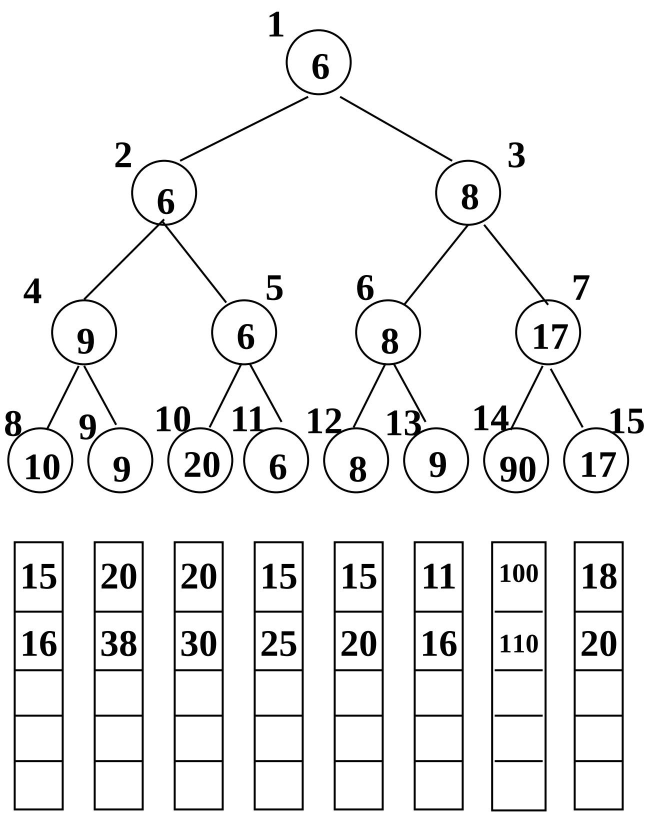
Red-Black Trees: Rotation

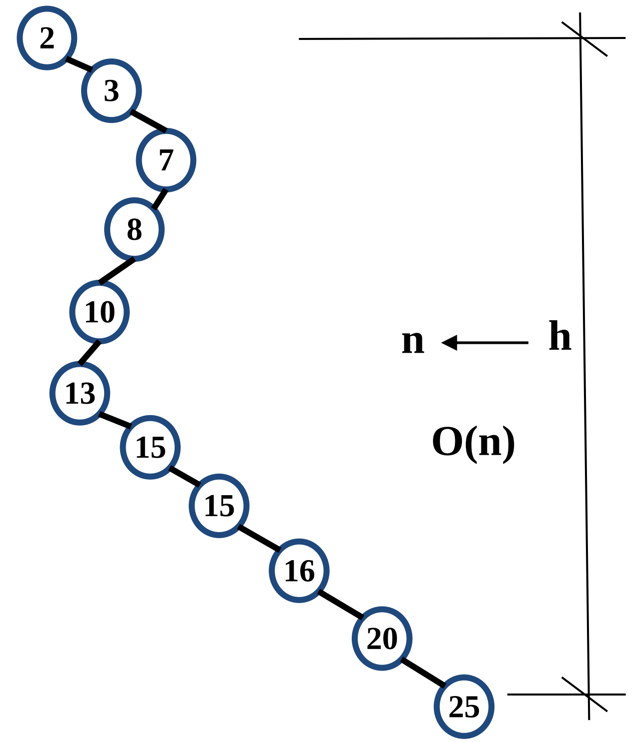
- Our basic operation for changing tree structure is called *rotation*:



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

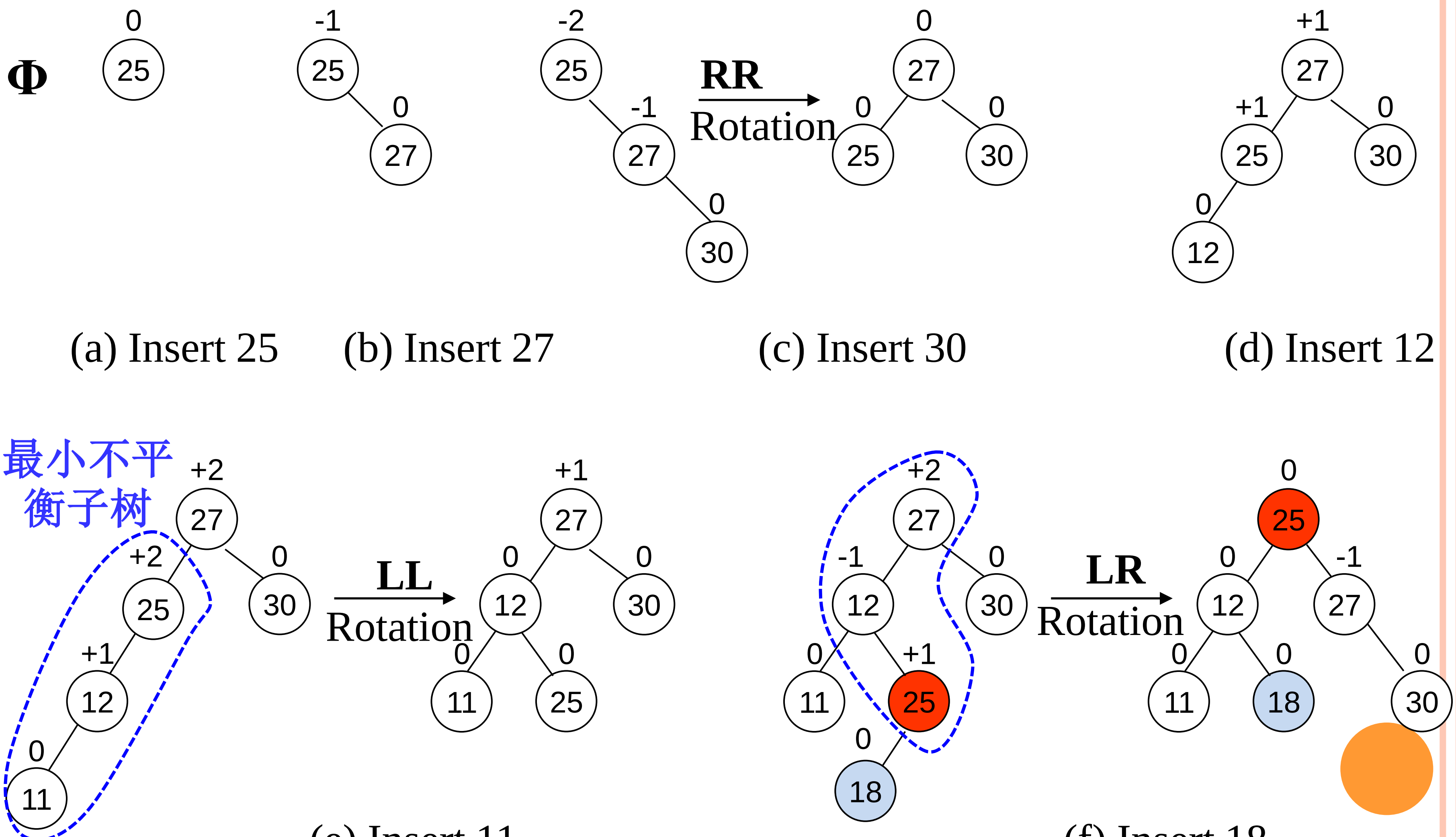
ROTATION (TRIVIAL)





ROTATION (TRIVIAL)

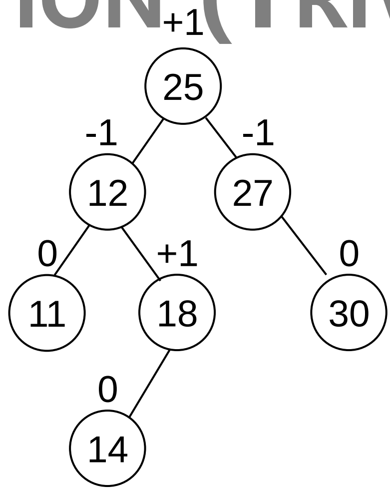
Insert ={25, 27, 30, 12, 11, 18, 14, 20, 15, 22}



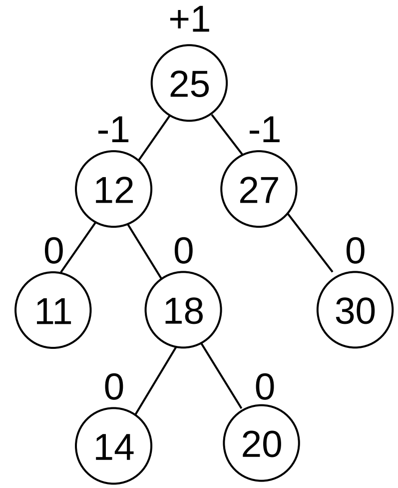
(a) Insert 25 (b) Insert 27 (c) Insert 30 (d) Insert 12

(e) Insert 11 (f) Insert 18

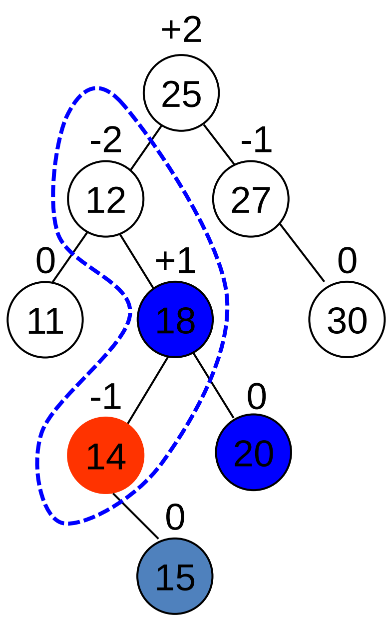
ROTATION (TRIVIAL)

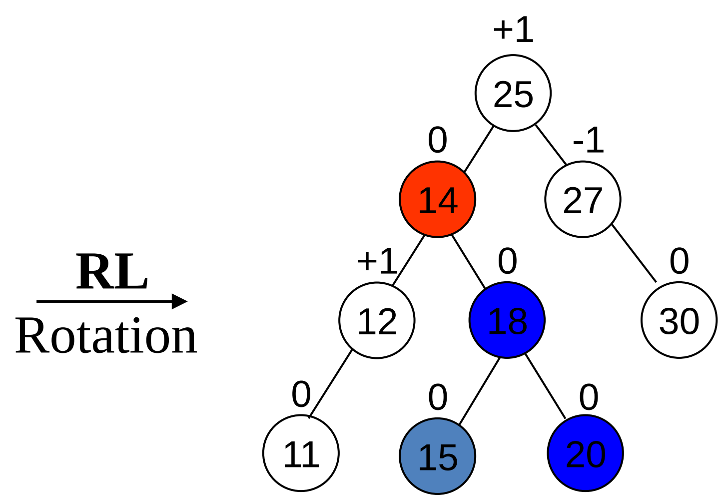


(g) Insert 14



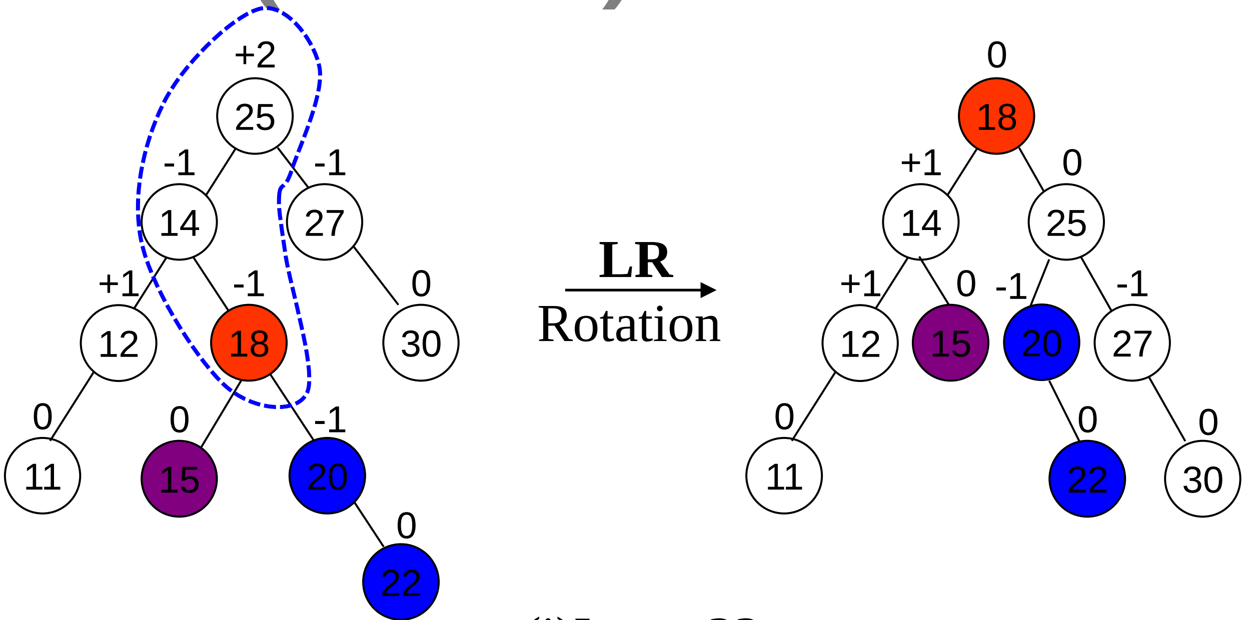
(h) Insert 20





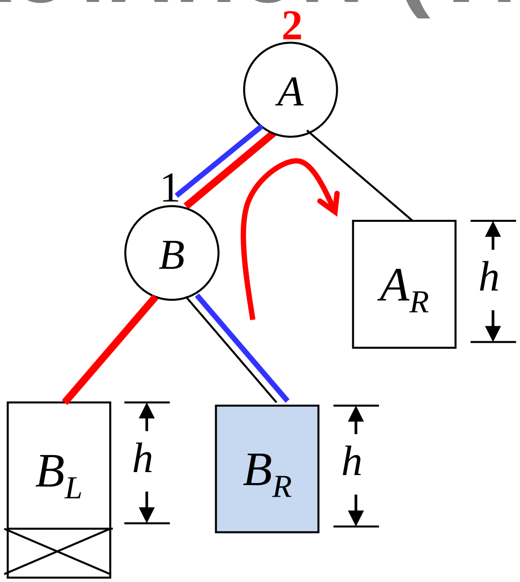
(i) Insert 15

Rotation (Trivial)



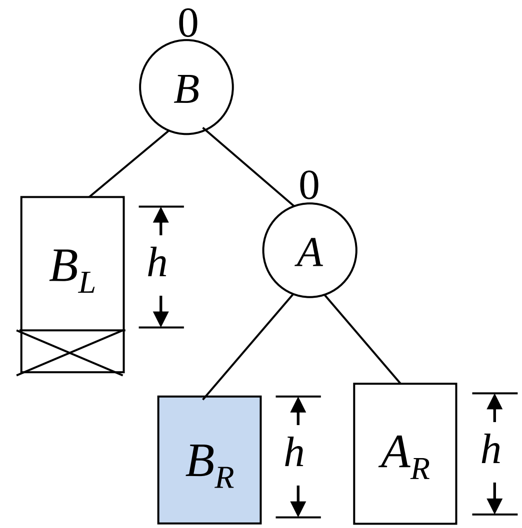
(i)Insert 22

ROTATION (TRIVIAL)



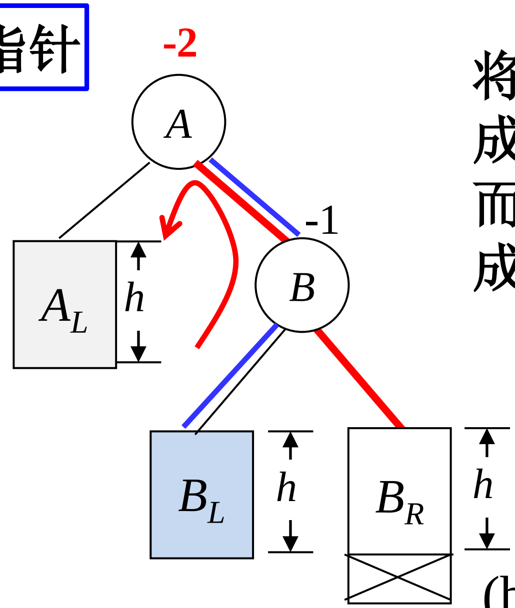
将A顺时针旋转，成为B的右子树，而原来B的右子树成为A的左子树。

LL型（顺）



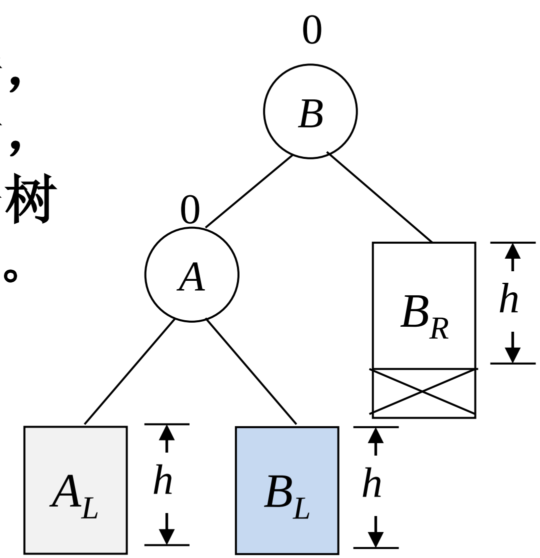
(a) LL型的旋转

修改指针



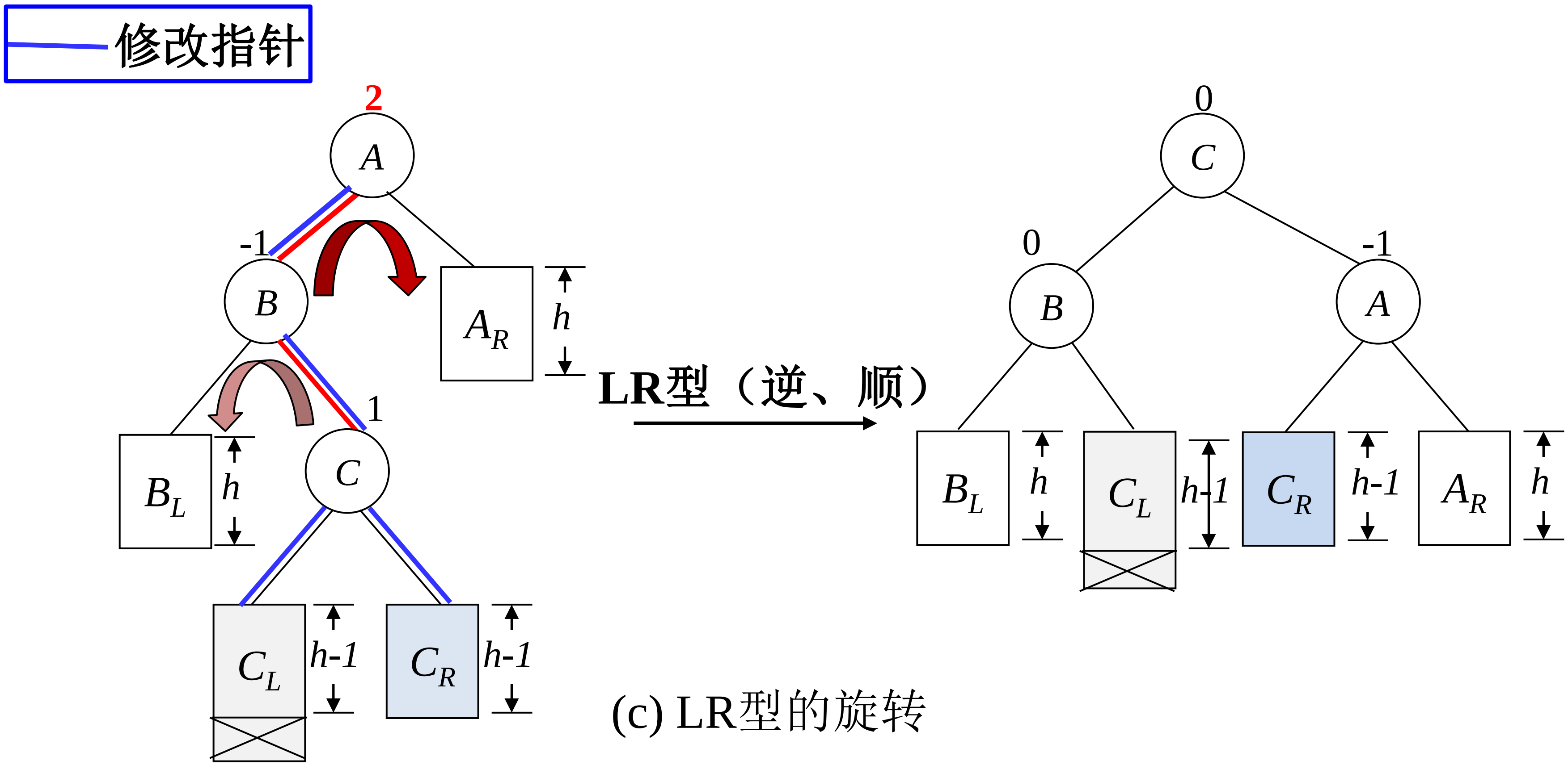
将A逆时针旋转，成为B的左子树，而原来B的左子树成为A的右子树。

RR型（逆）



(b) RR型的旋转

ROTATION (TRIVIAL)

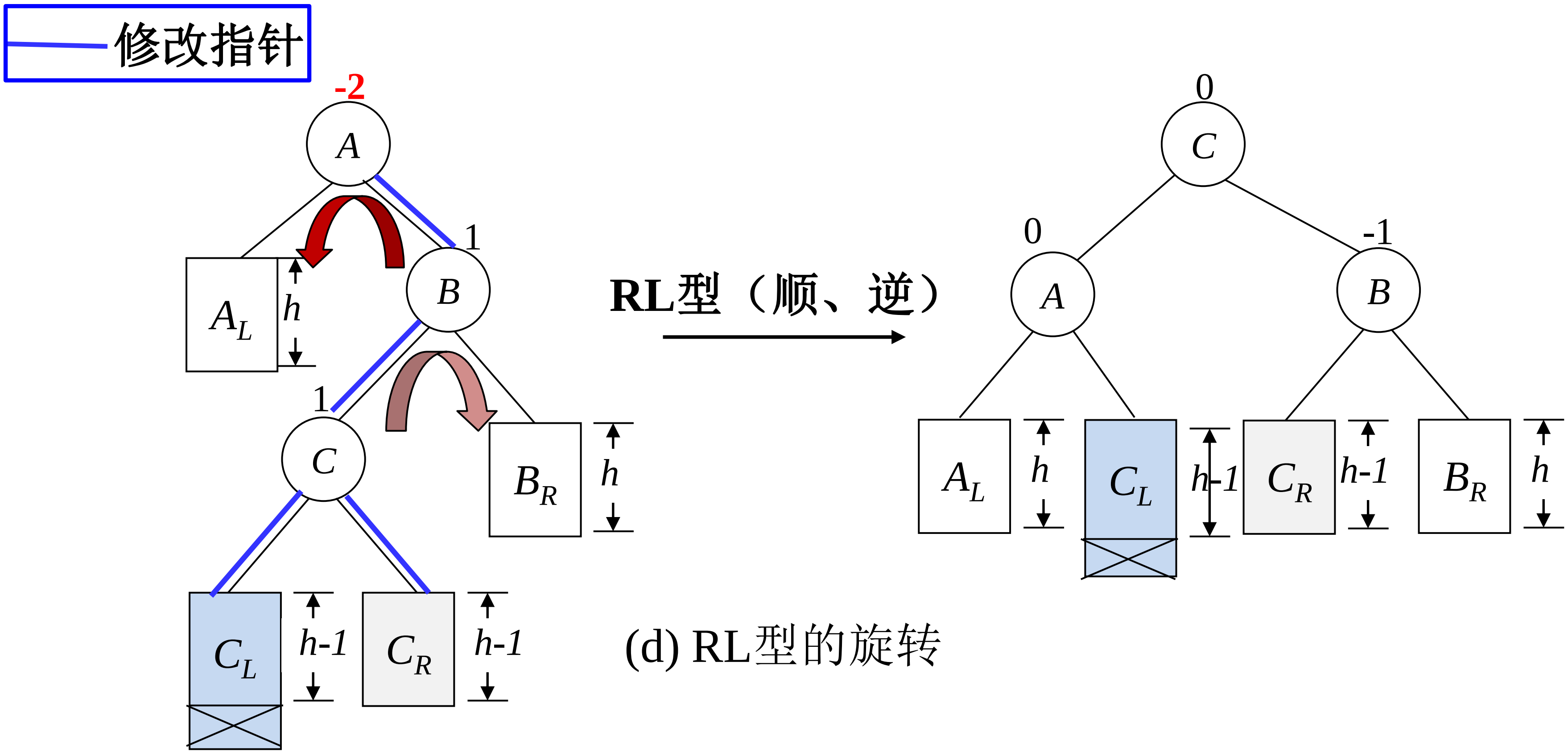


(c) LR型的旋转

（1）绕 ***C***，将 ***B*** 逆时针旋转，***C_L***作为 **B** 的右子树；

（2）绕 ***C***，将 ***A*** 顺时针旋转，***C_R***作为 **A** 的左子树。

ROTATION (TRIVIAL)

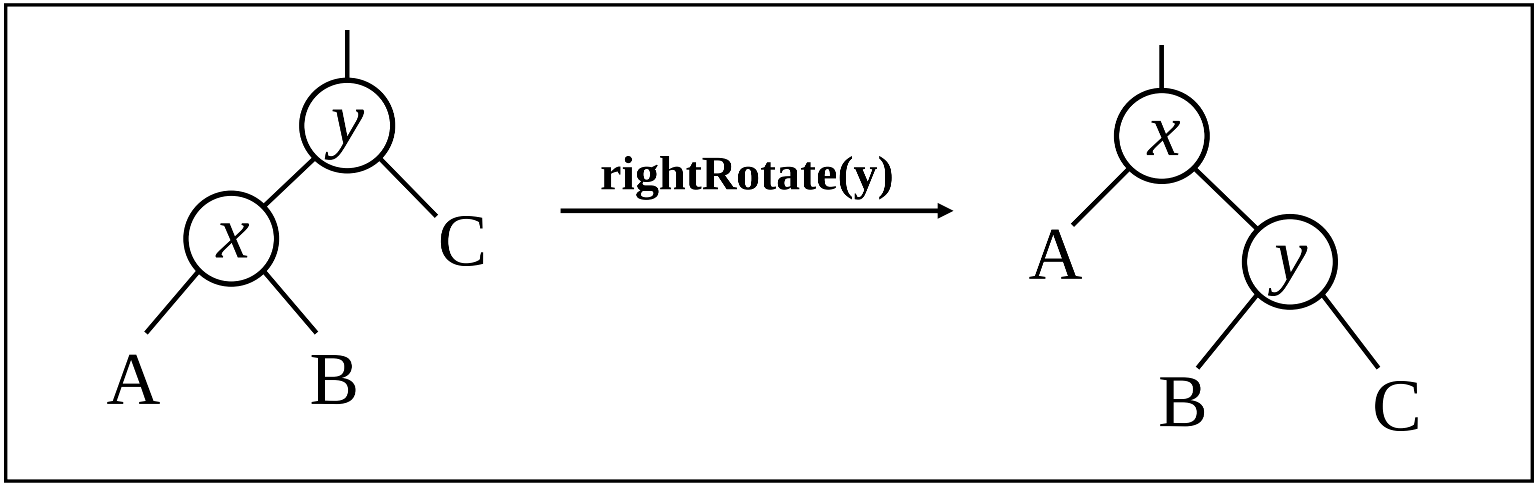


(d) RL型的旋转

（1）绕 *C*，将 *B* 顺时针旋转，C_R作为 B 的左子树；

（2）绕 *C*，将 *A* 逆时针旋转，C_L作为 A 的右子树。

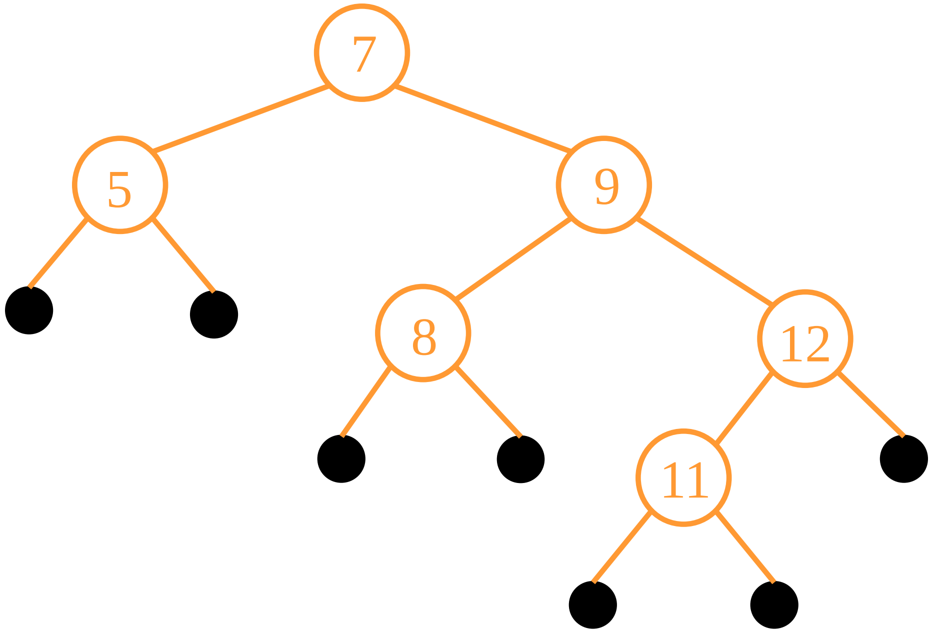
RED-BLACK TREES: ROTATION



- 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?*

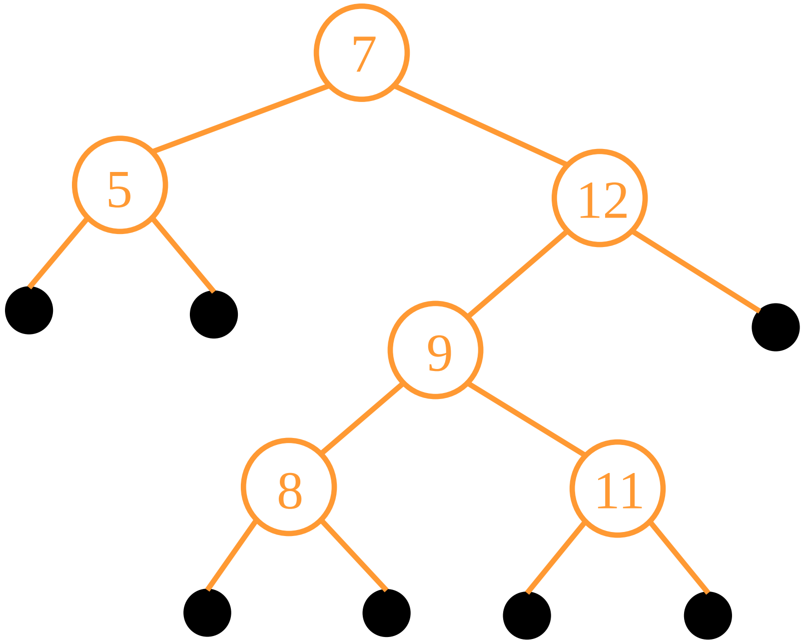
ROTATION EXAMPLE

- Rotate left about 9:



ROTATION EXAMPLE

- Rotate left about 9:



RED-BALCK TREES:INSERTION

- Insertion: the basic idea
  - Insert *x* into tree, color *x* red
  - Only *RB* property 4 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 $O(\lg n)$

Property 4. If a node is red, both children are black

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;
        x->p->p->color = RED;
        x = x->p->p;
    else    // y->color == BLACK
        if (x == x->p->right)
            x = x->p;
            leftRotate(x);
        x->p->color = BLACK;
        x->p->p->color = RED;
        rightRotate(x->p->p);
else    // x->p == x->p->p->right
    (same as above, but with
    "right" & "left" exchanged)
```

Case 1

Case 2

Case 3

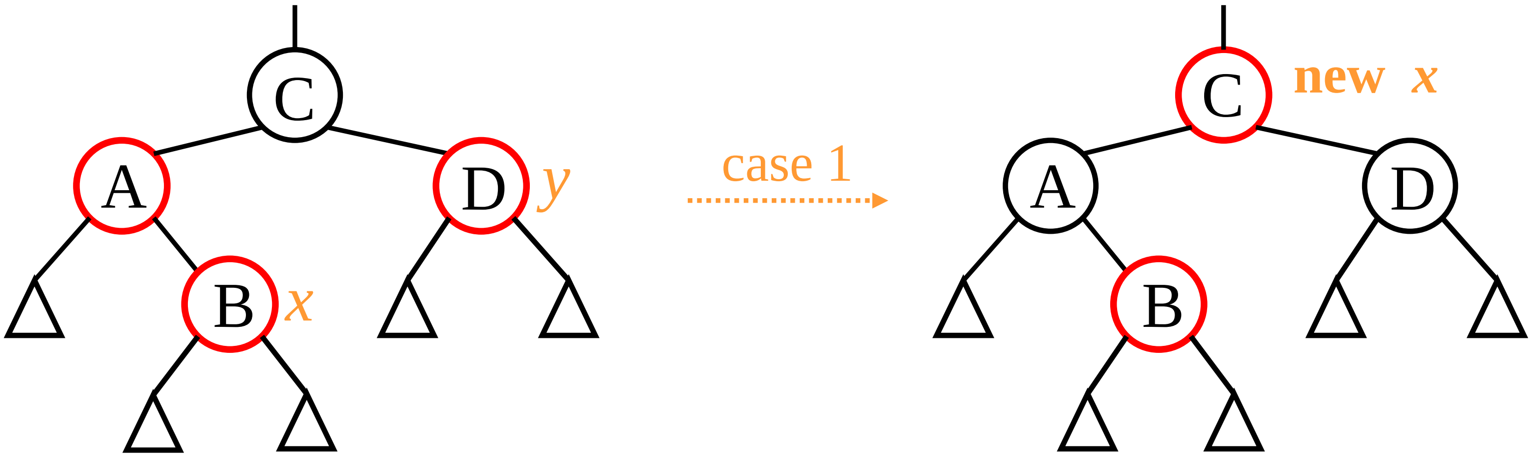
rbInsert(x)

```
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

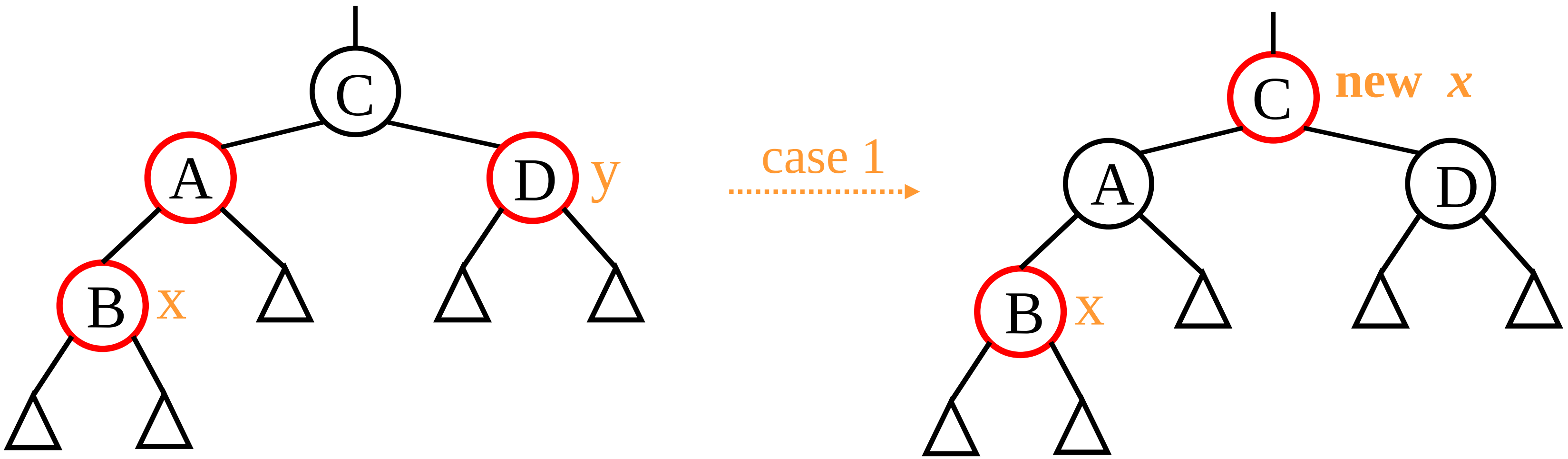


Change colors of some nodes, preserving #5: all downward paths have equal b.h.
The while loop now continues with *x*’s grandparent as the new *x*

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

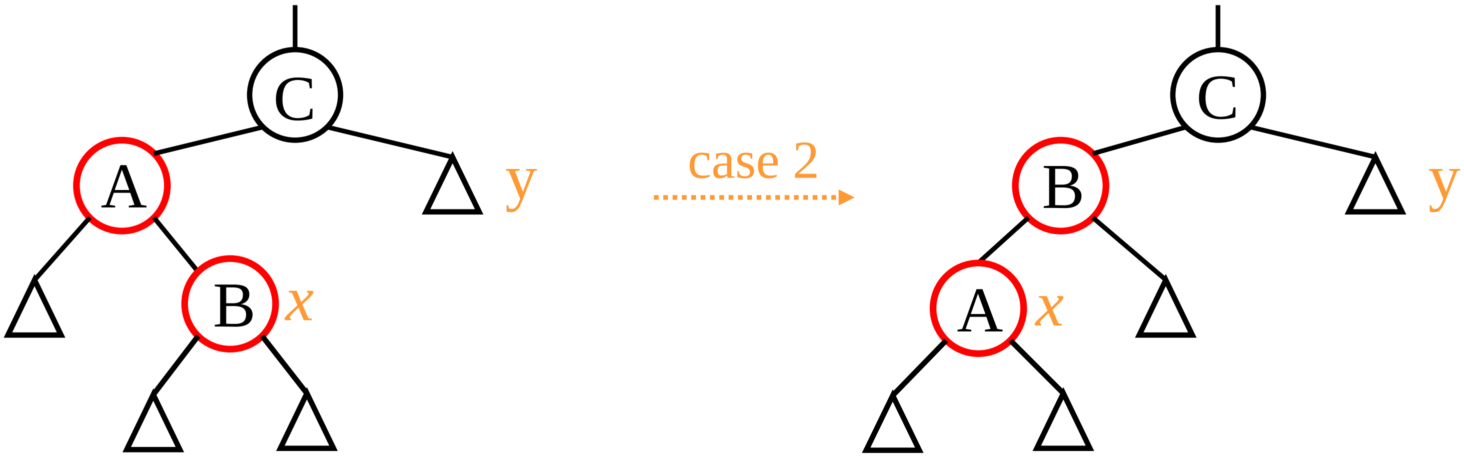


Same action whether *x* is a left or a right child

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

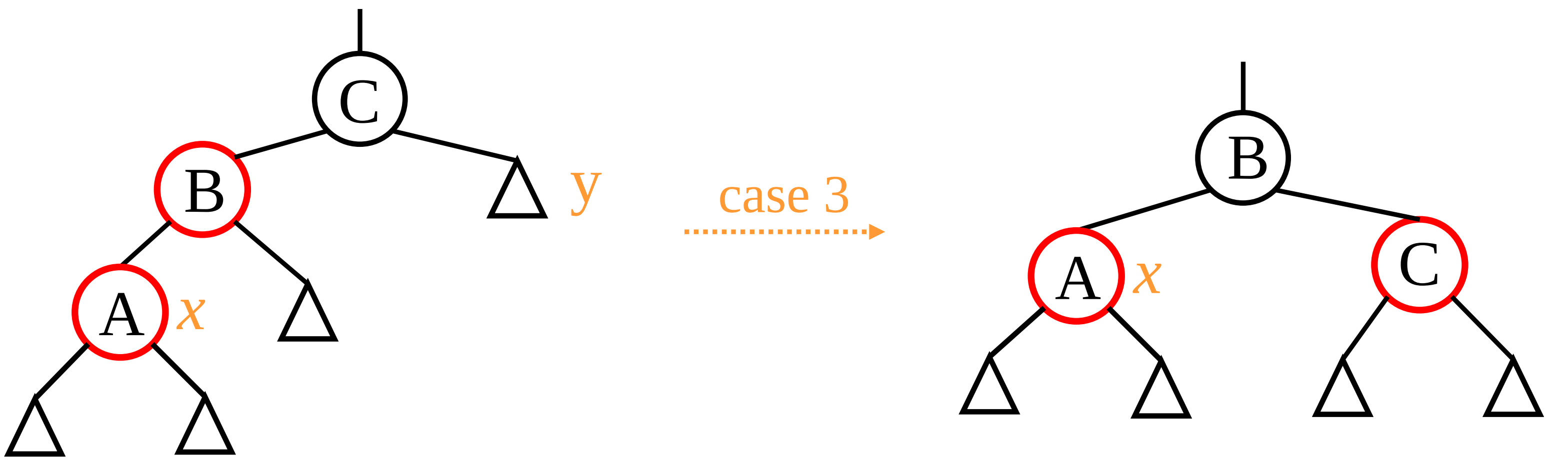


Transform case 2 into case 3 (*x* is left child) with a left rotation
This preserves property 5: all downward paths contain same number of black nodes

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 5: all downward paths contain same number of black nodes