Red-Black Trees

Use of RB Tree

- Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time. Not only does this make them valuable in time-sensitive applications such as real-time applications, but it makes them valuable building blocks in other data structures which provide worst-case guarantees; for example, many data structures used in computational geometry can be based on red–black trees, and the Completely Fair Scheduler used in current Linux kernels uses red–black trees.
- Red–black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets which can retain previous versions after mutations. The persistent version of red–black trees requires O(log*n*) space for each insertion or deletion, in addition to time.

A clip from Tarjan's Paper

In this paper we apply the related concepts of *amortized complexity* and *self-adjustment* to binary search trees. We are motivated by the observation that the known kinds of efficient search trees have various drawbacks. Balanced trees, such as height-balanced trees [2, 22], weight-balanced trees [26], and B-trees [6] and their variants [5, 18, 19, 24] have a worst-case time bound of $O(\log n)$ per operation on an n-node tree. However, balanced trees are not as efficient as possible if the access pattern is nonuniform, and they also need extra space for storage of balance information. Optimum search trees [16, 20, 22] guarantee minimum average access time, but only under the assumption of fixed, known access probabilities and no correlation among accesses. Their insertion and deletion costs are also very high. Biased search trees [7, 8, 13] combine the fast average access time of optimum trees with the fast updating of balanced trees but have structural constraints even more complicated and harder to maintain than the constraints of balanced trees. Finger search trees [11, 14, 19, 23, 24] allow fast access in the vicinity of one or more "fingers" but require the storage of extra pointers in each node.

A little history

- 1962: The idea of balancing a search tree is due to Adel'son-Velskii and Landis.
- 1970: Hopcroft introduced 2-3 trees. (B-tree is a generalization of it)
- 1972: Bayer invented Red-black trees.
- 1978: Guibas and Sedgewick introduced the red/black convention.

Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. The root is always black
  3. Every leaf (NULL pointer) is black
     - Note: this means every "real" node has 2 children
  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

Black-Height

- *black-height*: # black nodes on path to leaf
- *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 $h \leq 2 \lg(n + 1)$
  - Proved by induction

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
    - So the subtrees rooted at each child contain at least $2^{bh(x) - 1} - 1$ internal nodes
    - Thus subtree at x contains
      $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1$
      $= 2 \bullet 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$ nodes

Proving Height Bound

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

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

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

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

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

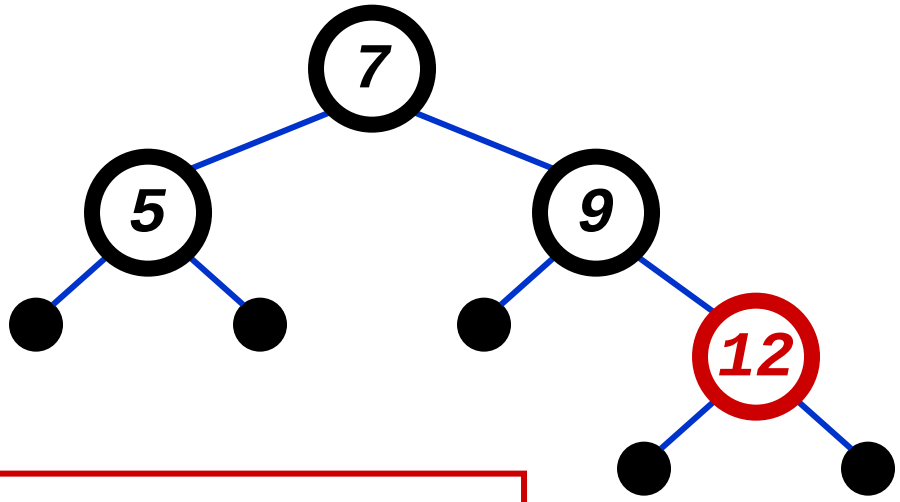
Thus $h = O(\lg n)$

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:*

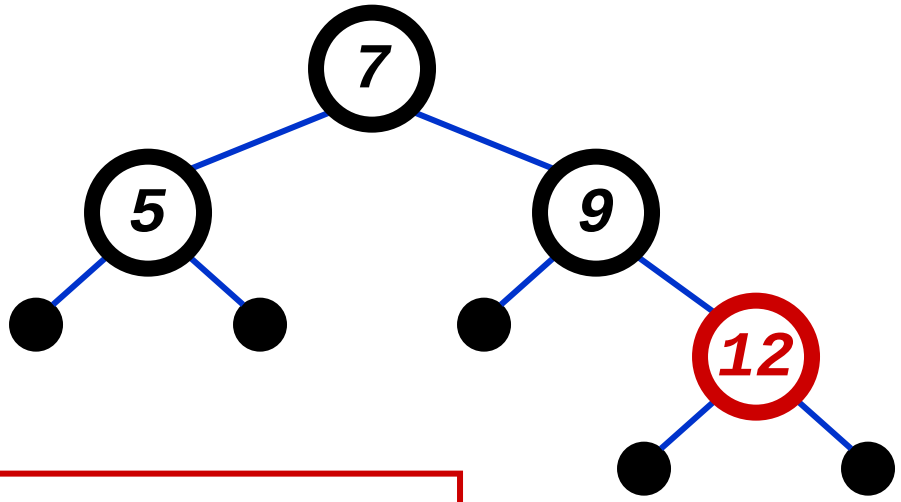


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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?*

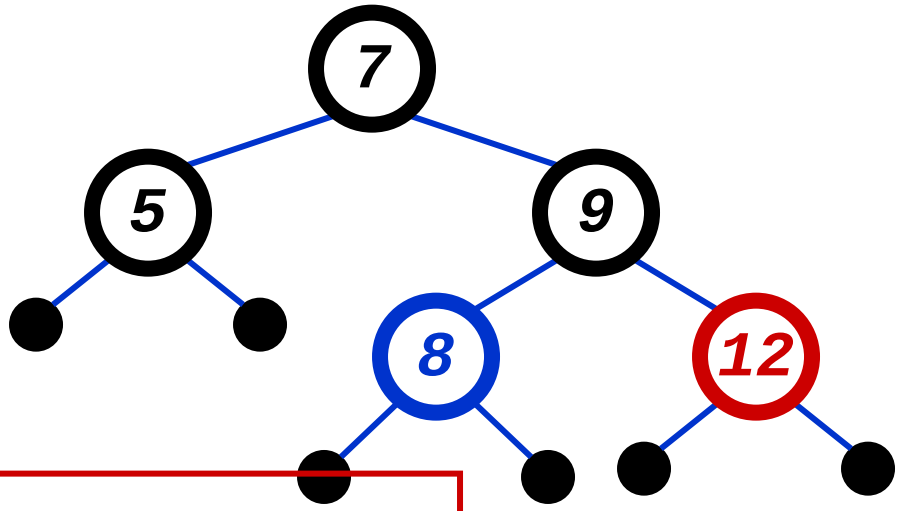


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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?*

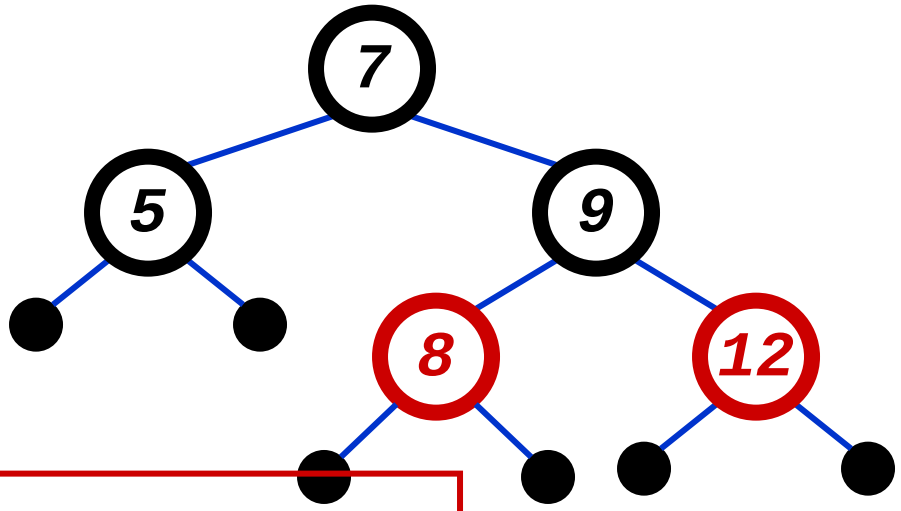


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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?*

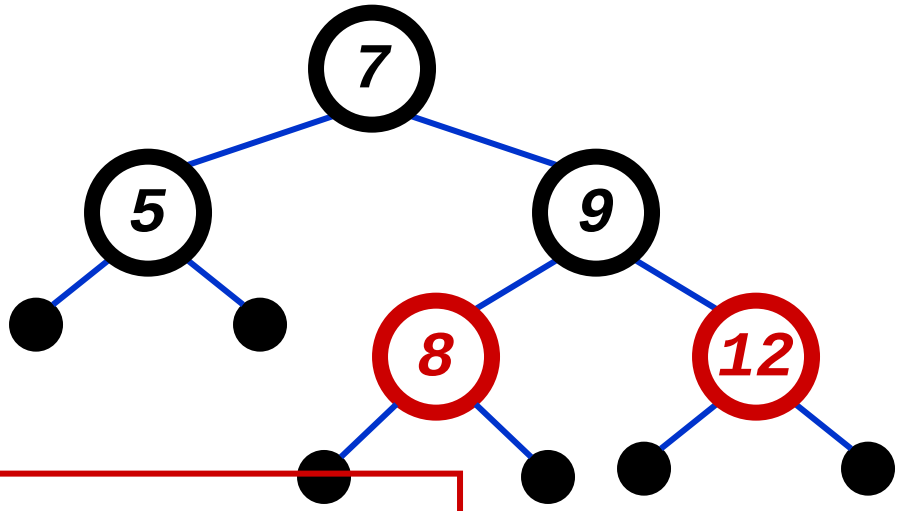


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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?*

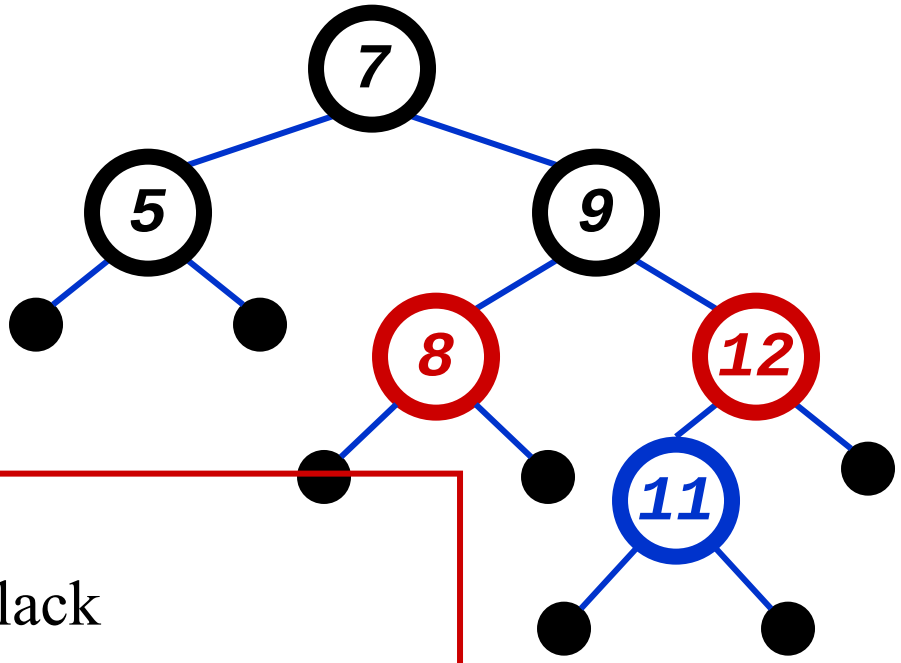


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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?*

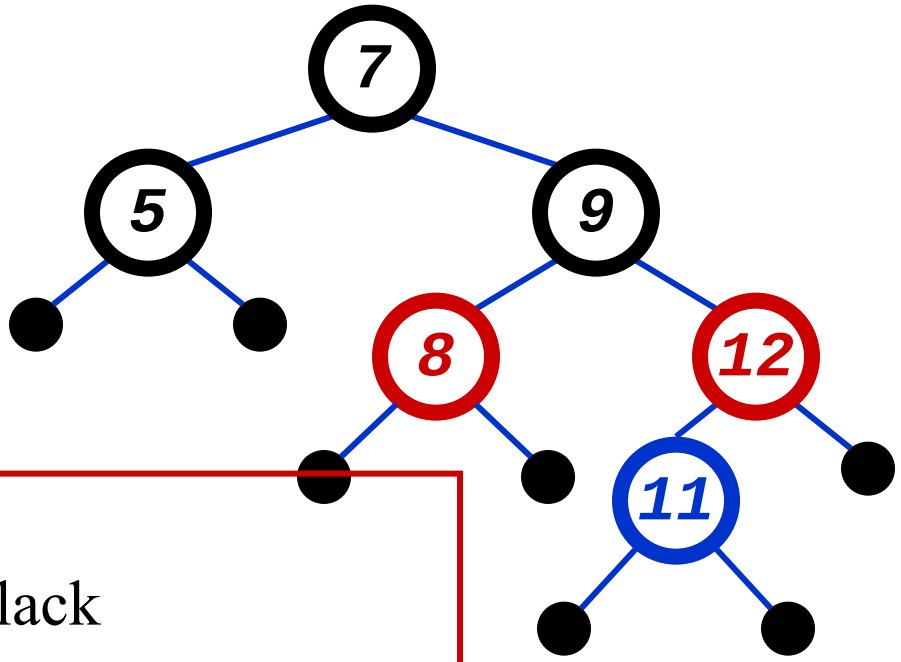


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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)

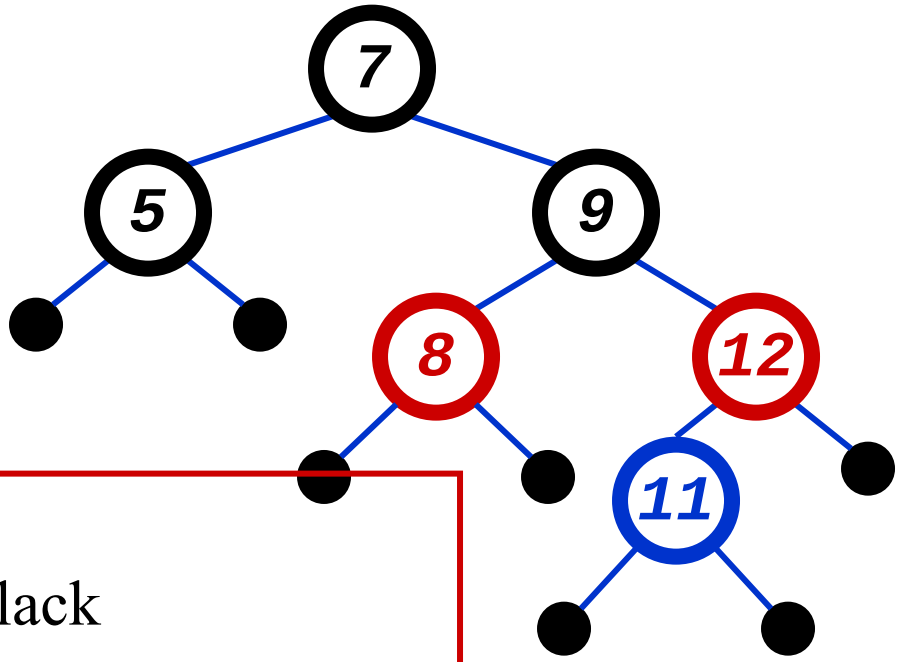


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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)

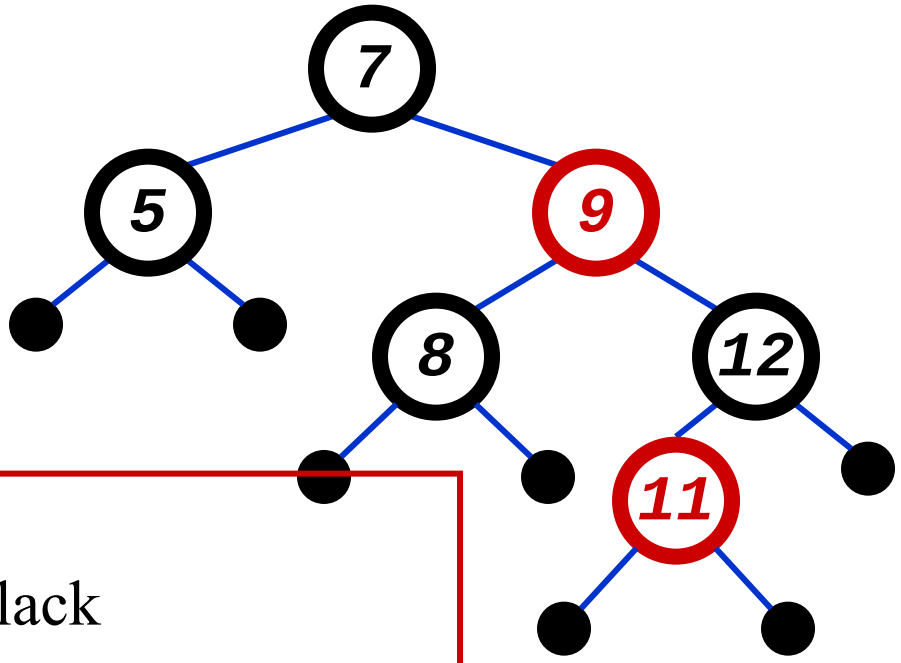


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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

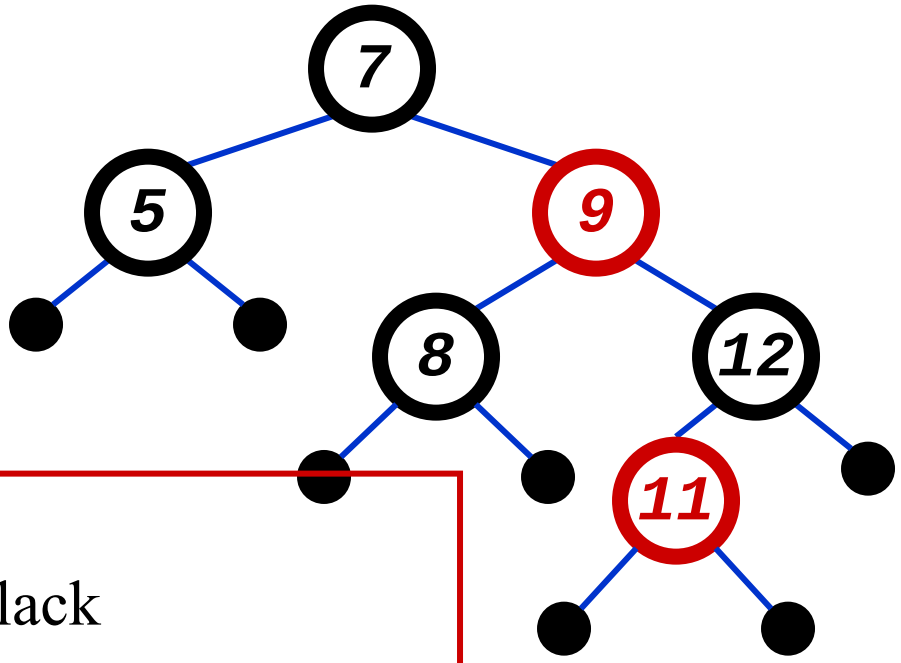


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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?*

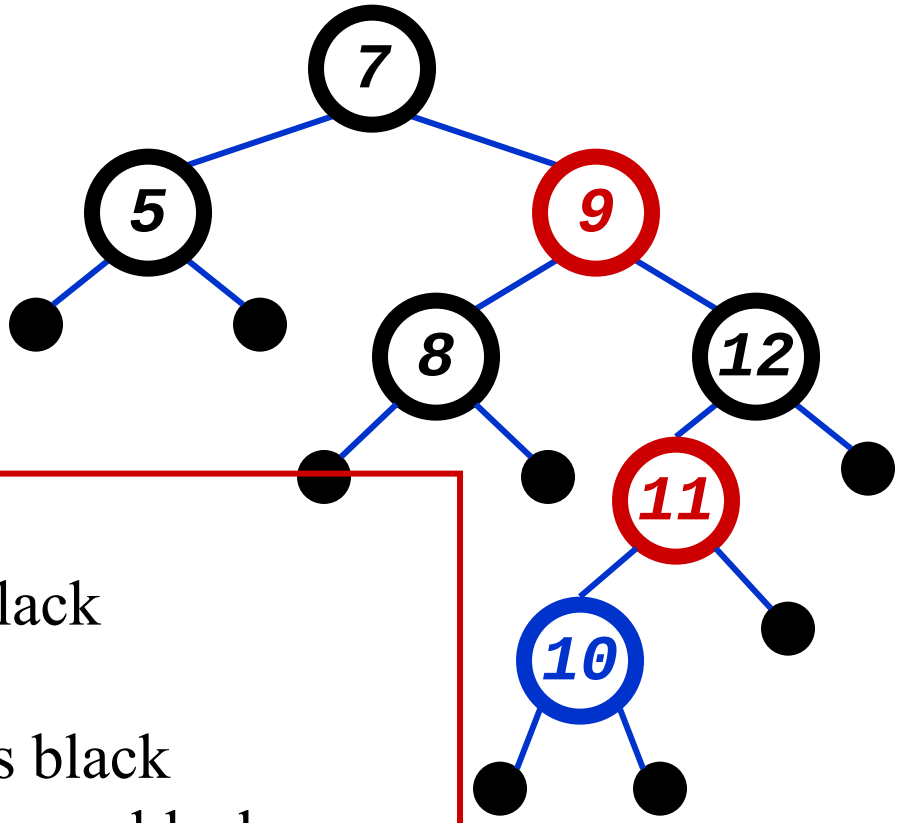


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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?*

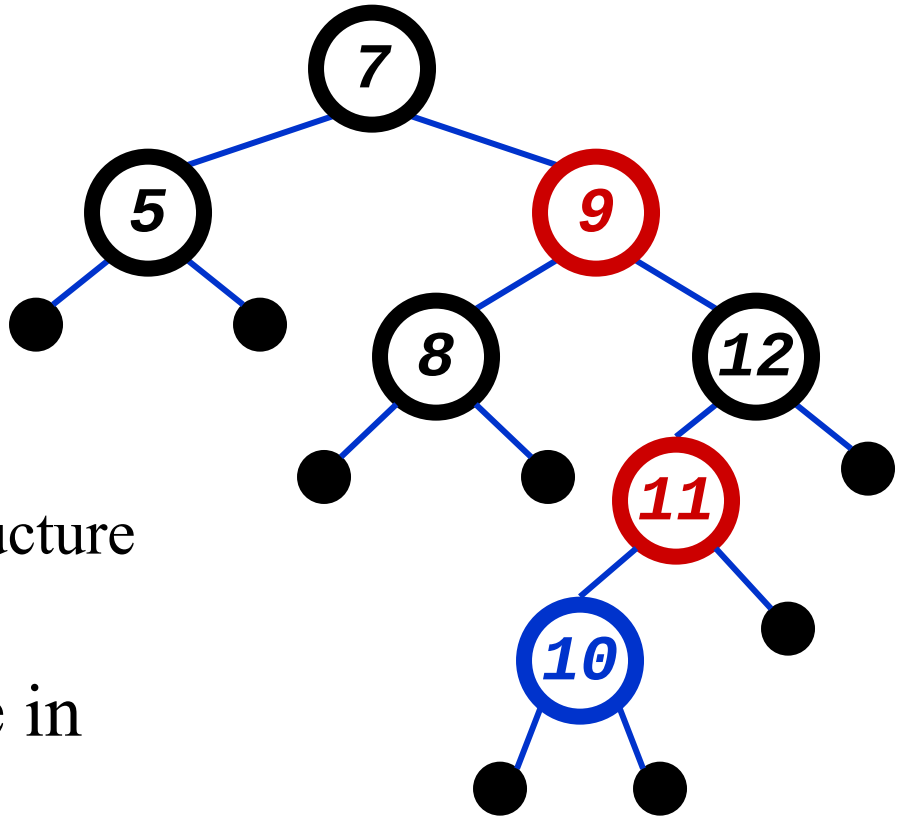


Red-black properties:

1. Every node is either red or black
2. The root is always black
3. Every leaf (NULL pointer) is 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

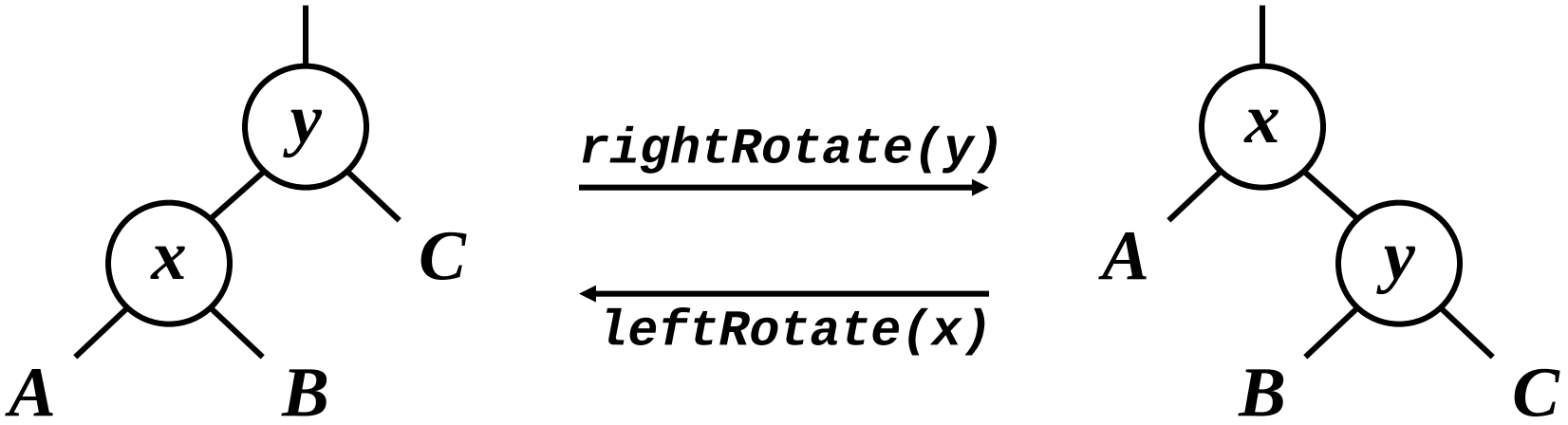
Red-Black Trees: The Problem With Insertion

- 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



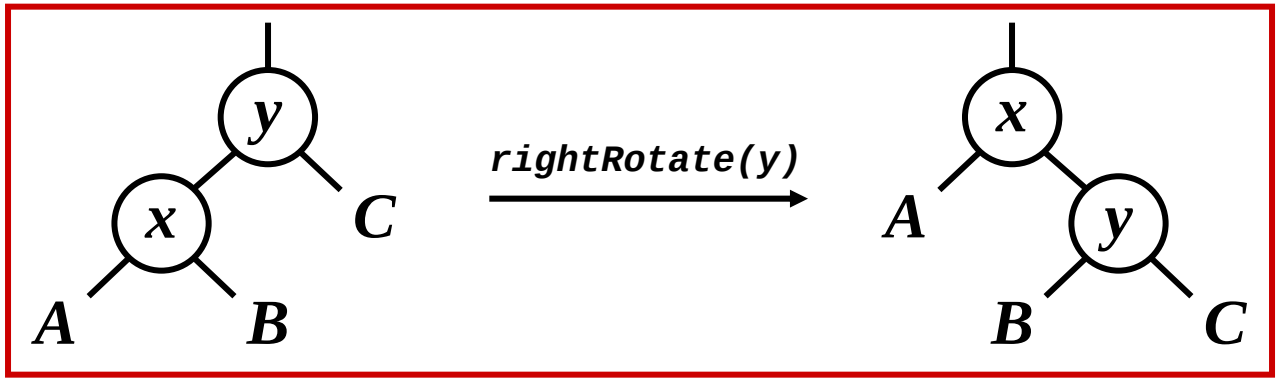
RB Trees: Rotation

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



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

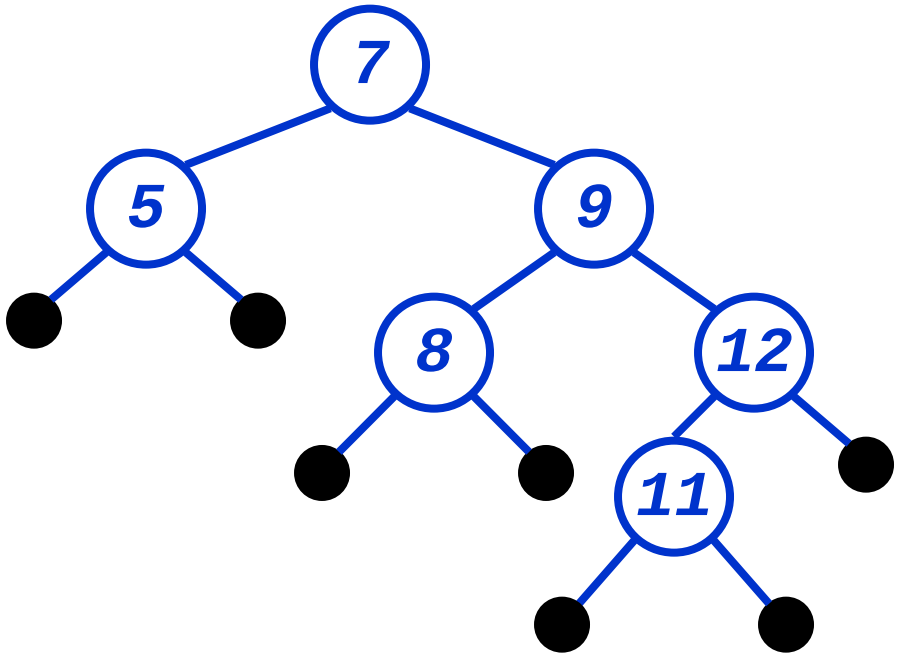
RB Trees: Rotation



- 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?*

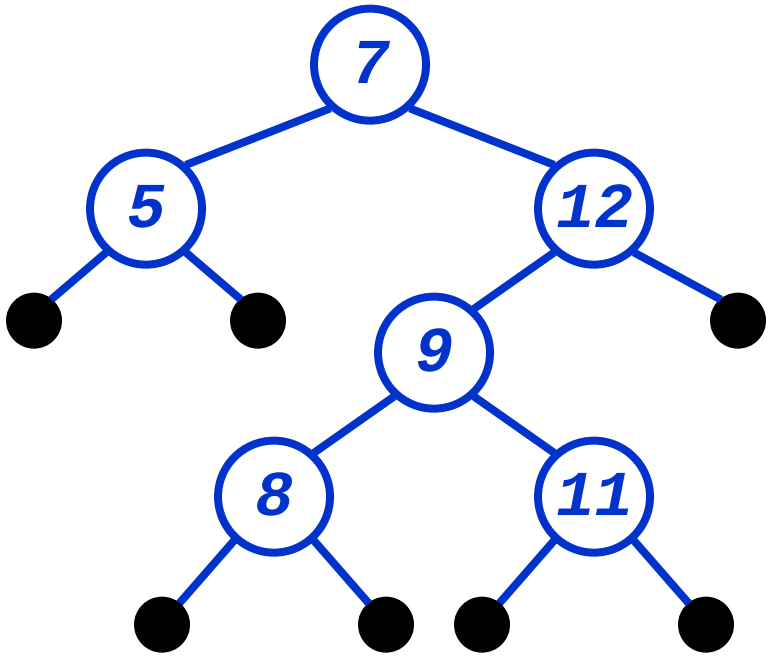
Rotation Example

- Rotate left about 9:



Rotation Example

- Rotate left about 9:



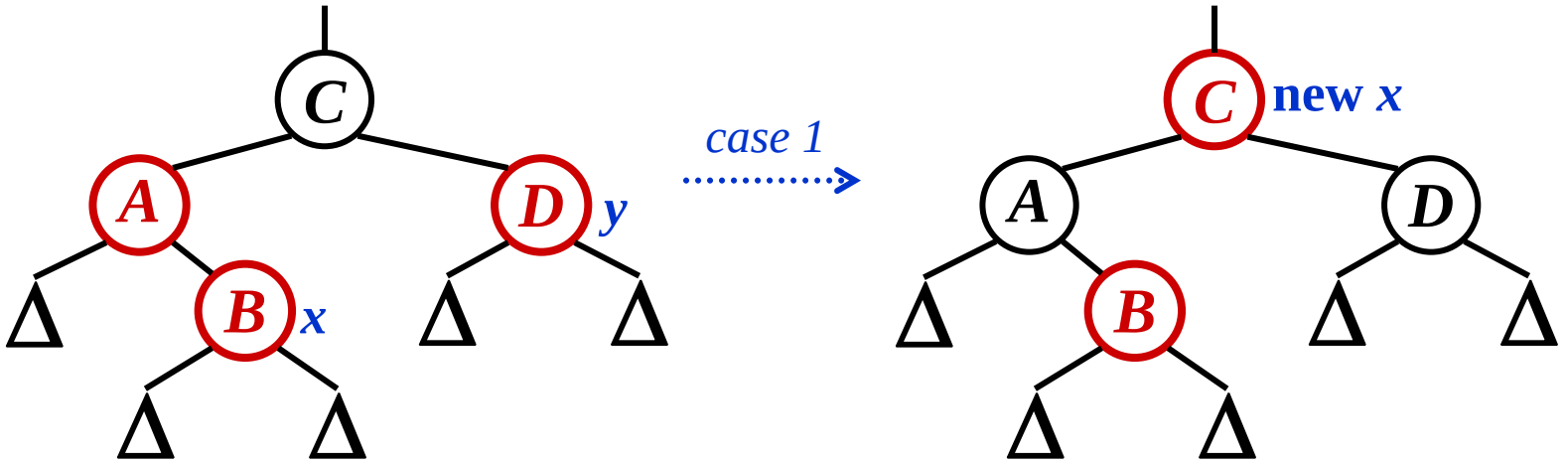
Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert x into tree, color x red
  - Only r-b property 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)

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

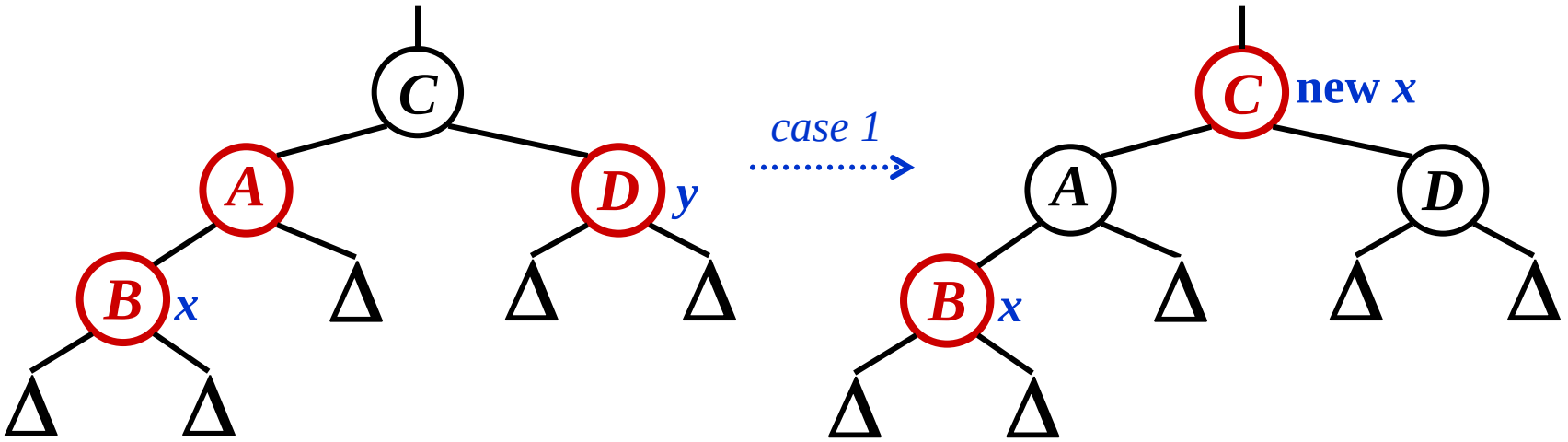


Change colors of some nodes, preserving #5: all downward paths have equal b.h.
The while loop now continues with x's grandparent as the new x

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

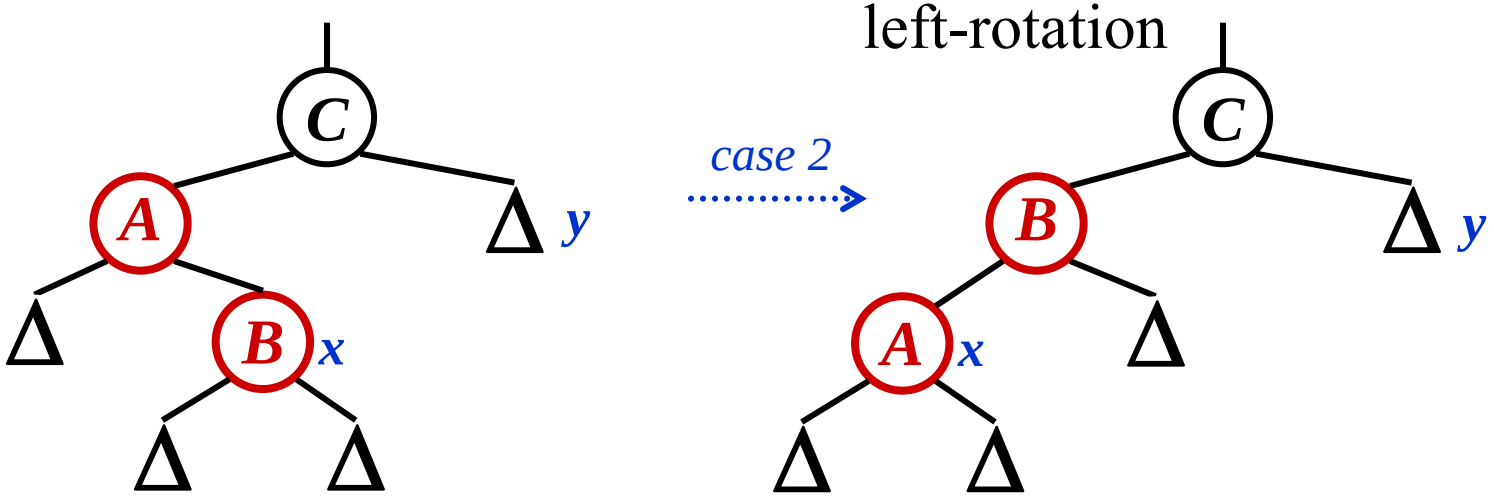


Same action whether x is a left or a right child

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

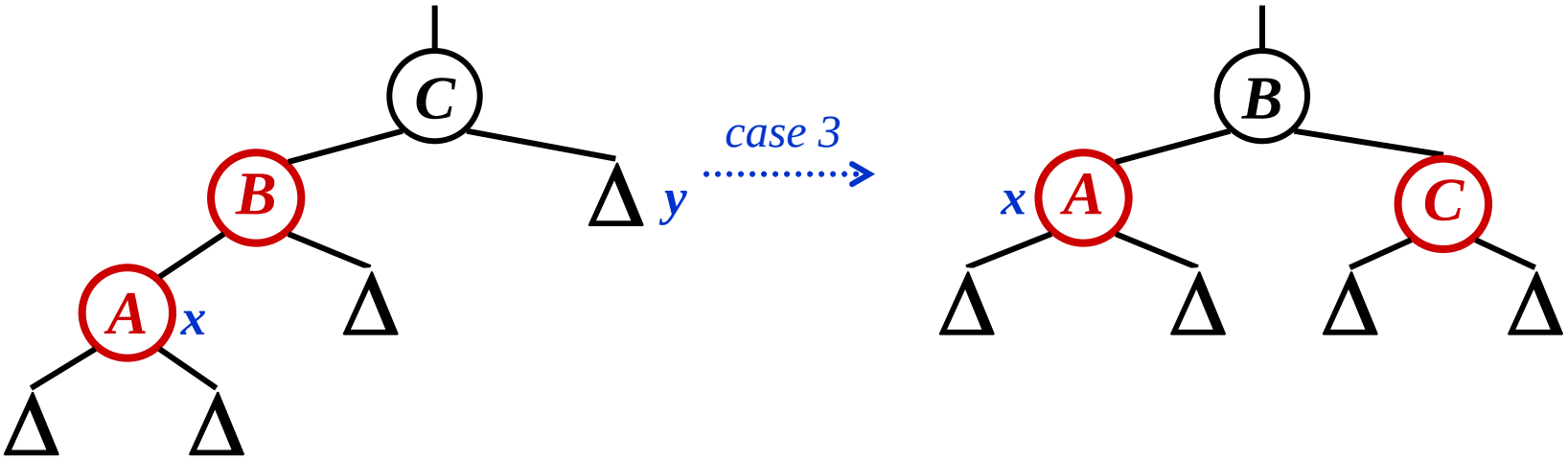


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 5: all downward paths contain same number of black nodes

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

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)

```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #4 up tree, maintaining #5 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;
          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: uncle is RED

Case 2

Case 3

Red-Black Trees: Deletion

- And you thought insertion was tricky…

Red-Black Trees

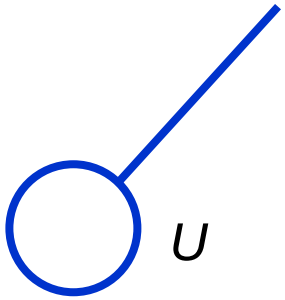
Bottom-Up Deletion

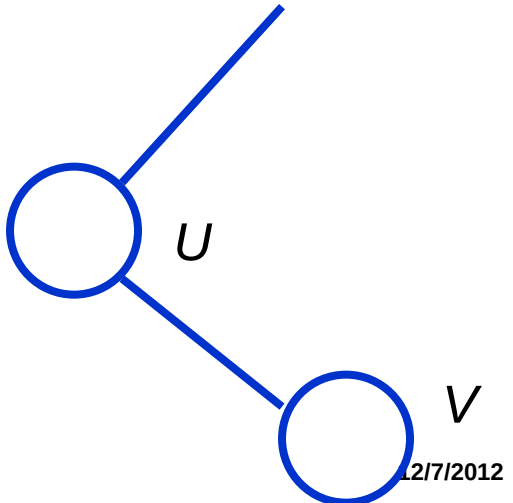
Recall “ordinary” BST Delete

1. If vertex to be deleted is a leaf, just delete it.
2. If vertex to be deleted has just one child, replace it with that child
3. Otherwise, if vertex Z has both a left and a right child. We find Z’s successor U, replace Z’s value by U’s value and then delete U (a recursive step, and Y must be a leaf or has just one child).

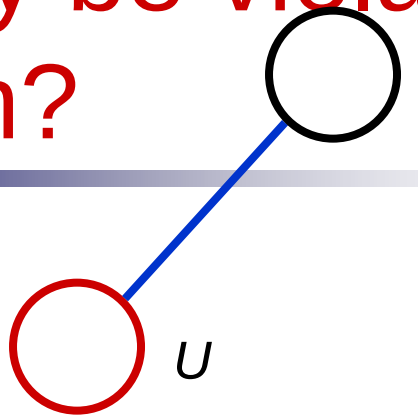
Bottom-Up Deletion

1. Do ordinary BST deletion. Eventually a "case 1" or "case 2" will be done (leaf or just one child). If deleted node, U, is a leaf, think of deletion as replacing with the NULL pointer, V. If U had one child, V, think of deletion as replacing U with V.
2. What can go wrong??





Which RB Property may be violated after deletion?



1. If U is red?

   Not a problem – no RB properties violated

2. If U is black?

   If U is not the root, deleting it will change the black-height along some path

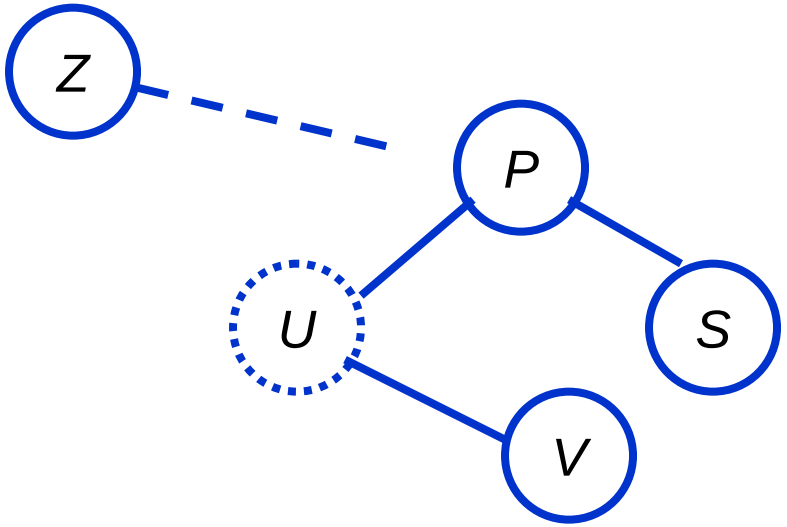
Fixing the problem

- Think of V (NULL pointer or U's only child) as having an "extra" unit of blackness. This extra blackness must be absorbed into the tree (by a red node), or propagated up to the root (without violating the RB properties) and out of the tree.

- If V is red, then we color it black to make it absorb the extra black. Otherwise, V is "double black".
- There are four cases – our examples and "rules" assume that V is a left child. There are symmetric cases for V as a right child

Terminology

- The node just deleted was U (Z' successor!)
- The node that replaces it is V, which has an extra unit of blackness
- The parent of V is P
- The sibling of V is S



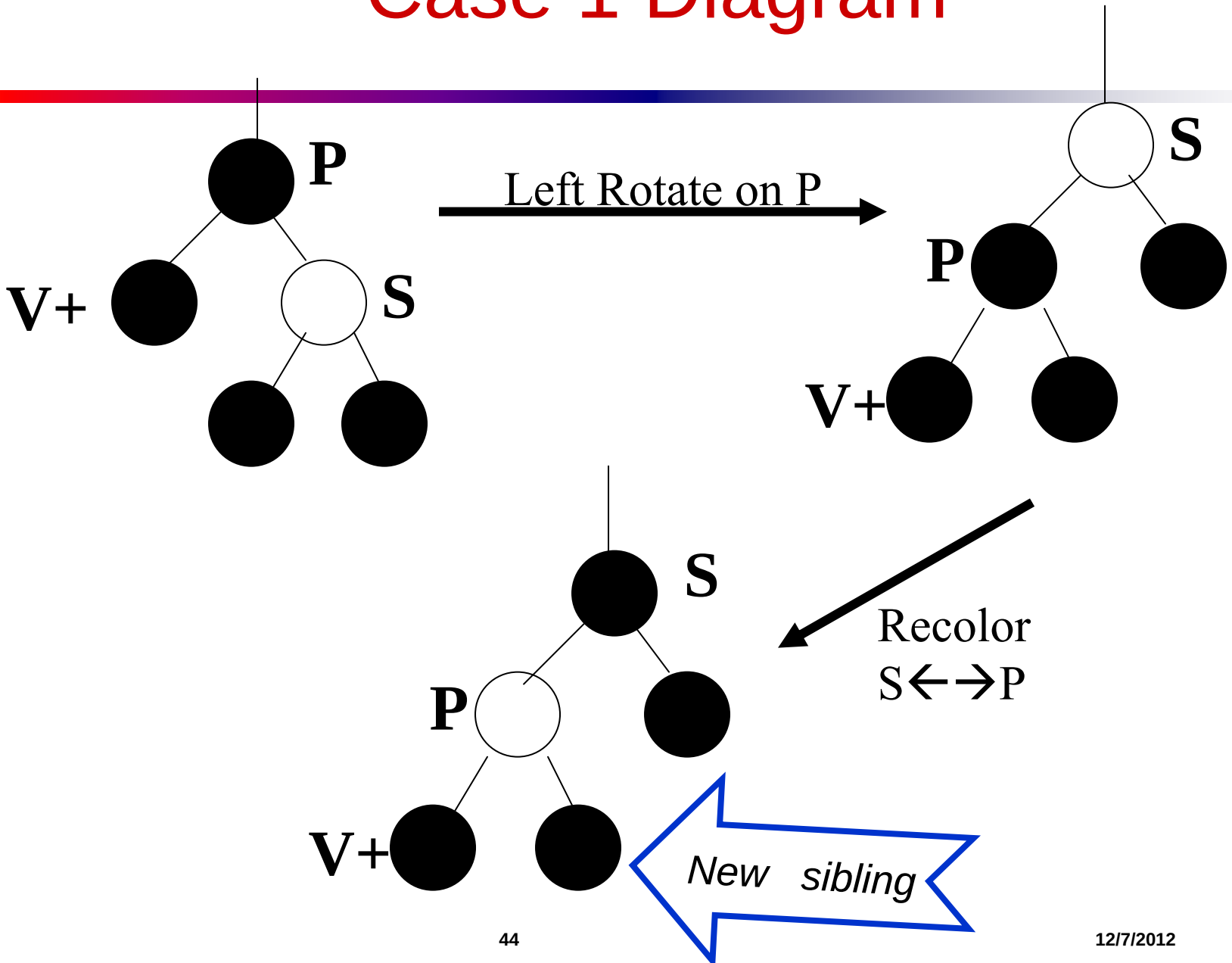
Black Node

Red Node

Red or Black and don't care

- 4 cases:
  - Case 1: V's sibling S is red; → Case 2/3/4
  - Case 2: V's sibling S is black; S's both children are black; →recursive or terminal
  - Case 3: V's sibling S is black; S's left child is red; S's right child is black; → Case 4
  - Case 4: V's sibling S is black; S's left child is red/black; S's right child is red; terminal case

Case 1 Diagram

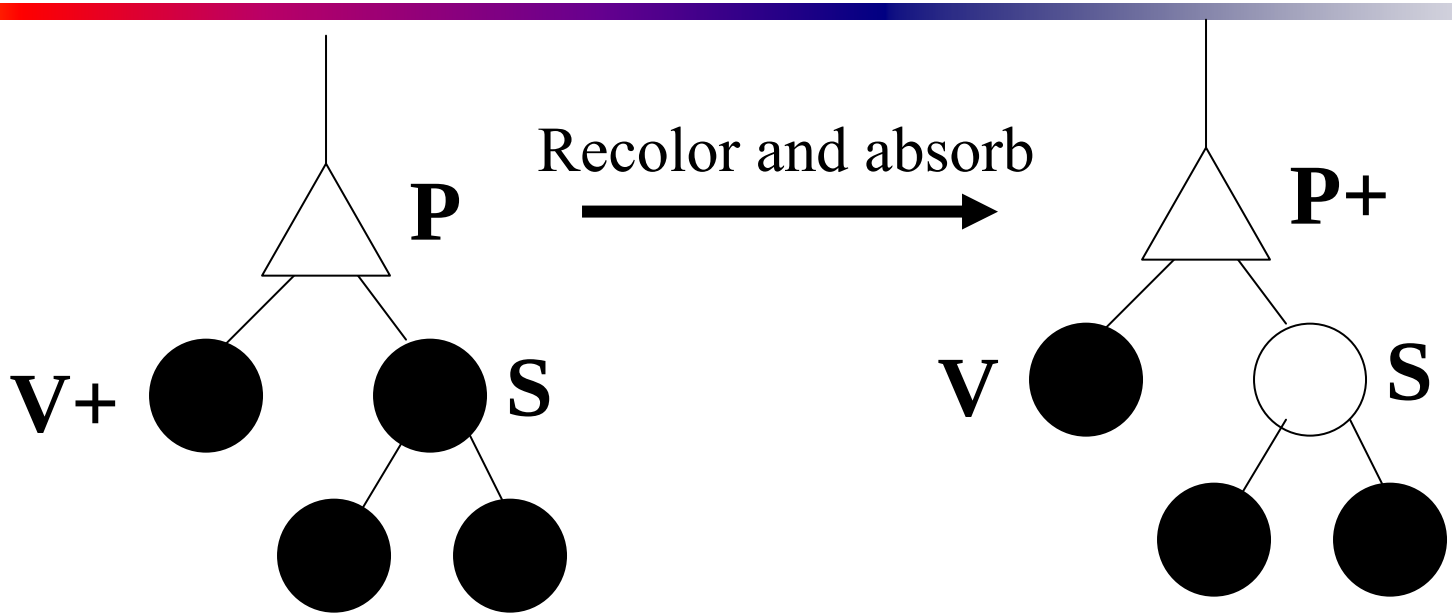


Bottom-Up Deletion
Case 1

- V's sibling, S, is Red
  - Left Rotation on P and recolor S & P
- NOT a terminal case – One of the other cases will now apply
- All other cases apply when S is Black

Back to Case Map

Case 2 diagram



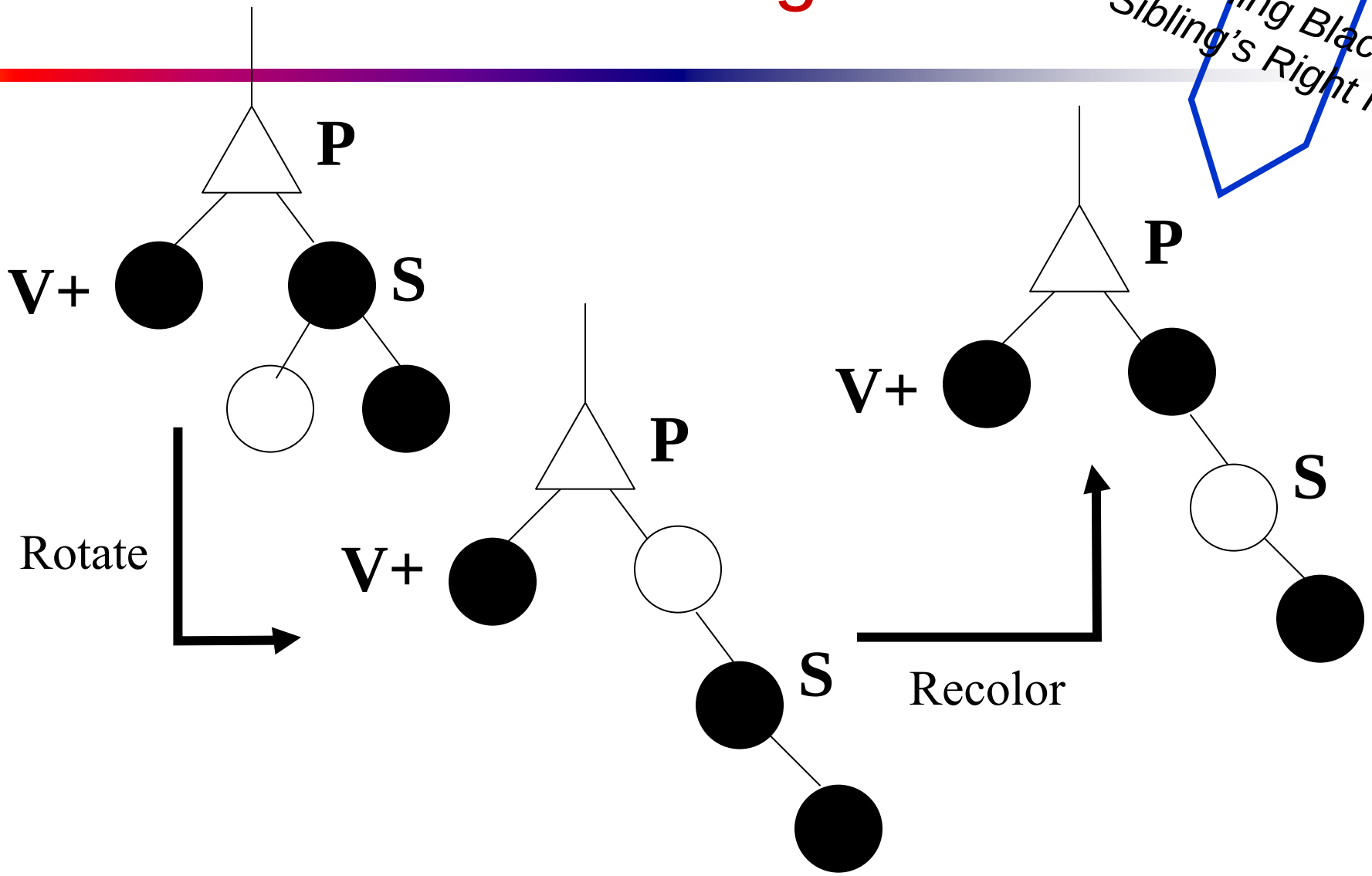
Either extra black absorbed by P (P was Red, now case done) or P now has extra blackness (P was black, now recursive at P+.)

Bottom-Up Deletion
Case 2

- V's sibling, S, is black and has two black children.
  - Recolor S to be Red
  - P absorbs V's extra blackness
    - If P is Red, we're done
    - If P is Black, it now has extra blackness and problem has been propagated up the tree

Back to Case Map

Case 3 Diagrams

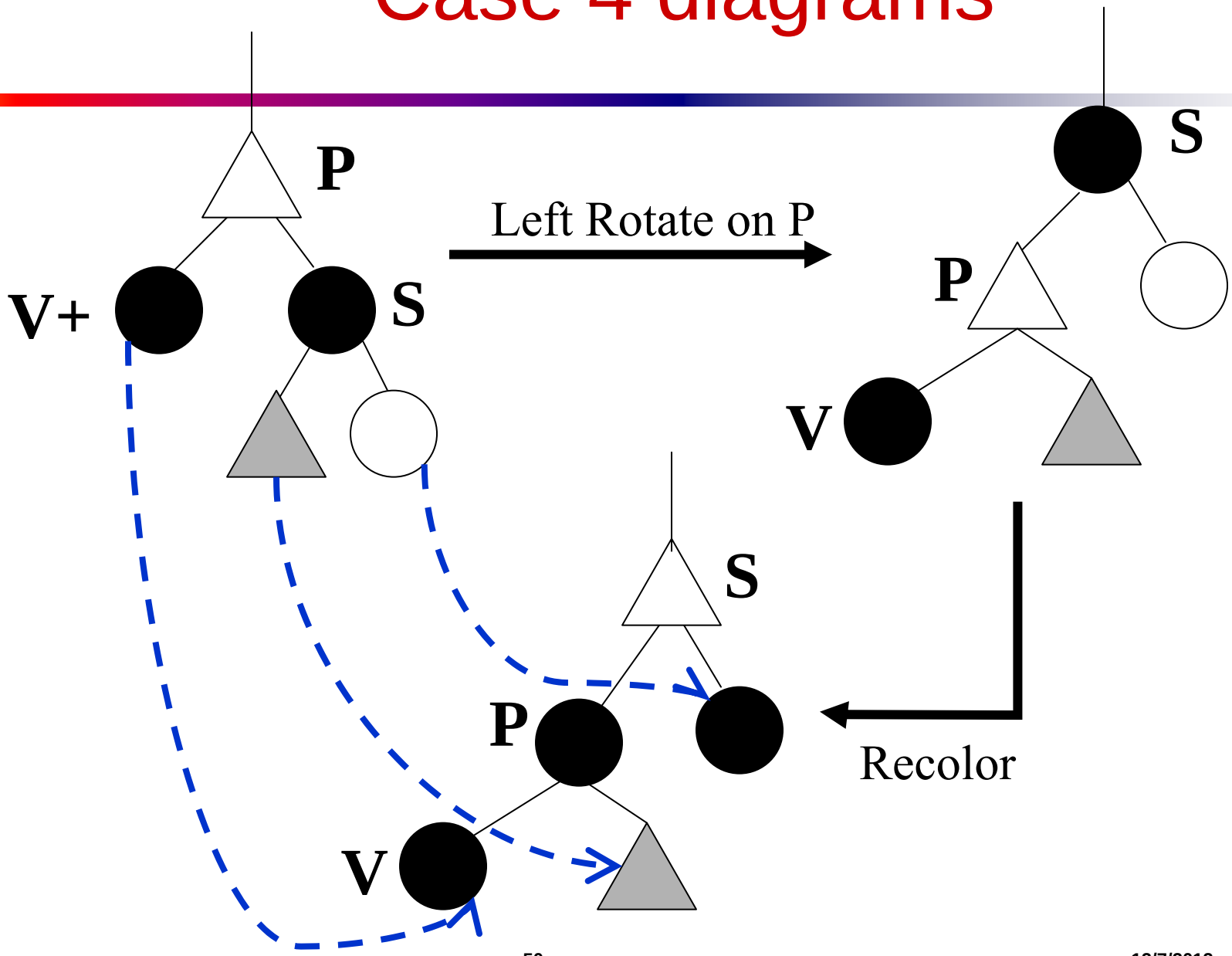


Bottom-Up Deletion
Case 3

- S is Black, S's right child is Black and S's left child is Red
  - Right Rotate on S
  - Swap color of S and S's left child
  - Now in case 4

Back to Case Map

Case 4 diagrams



Bottom-Up Deletion
Case 4

- S is black
- S's RIGHT child is RED (Left child either color)
  - Rotate S around P
  - Swap colors of S and P, and color S's Right child Black
- This is the terminal case – we're done

Back to Case Map

The End