Red-Black 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

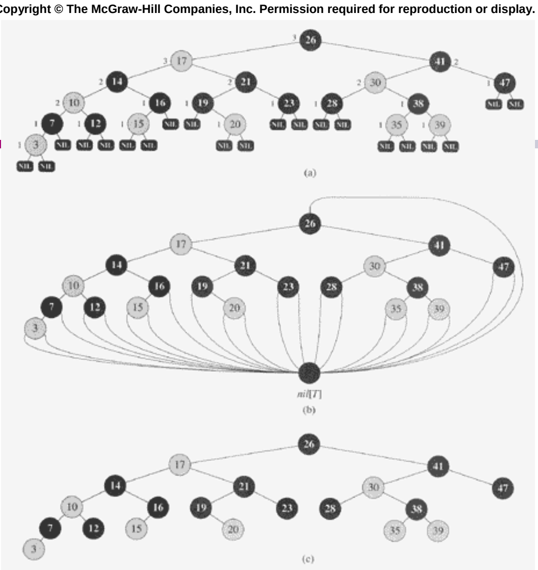


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL’s have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *nil*[*T*], which is always black, and with black-heights omitted. The root’s parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

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

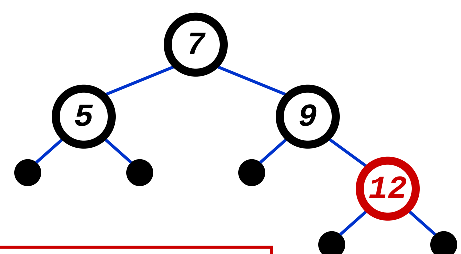
- *What is the minimum black-height of a node with height h?*
- A: a height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$

RB Trees: Worst-Case Time

- We will prove that a red-black tree has $h = O(\lg n)$ 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

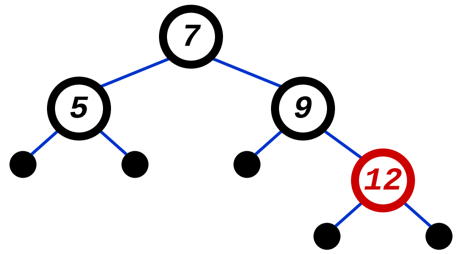
- *Color this tree:*



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

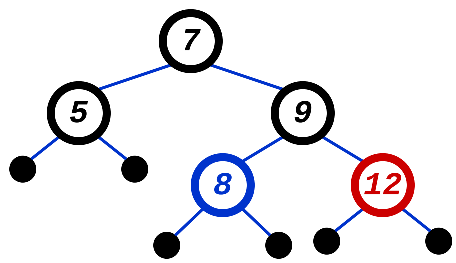
- Insert 8
  - *Where does it go?*



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

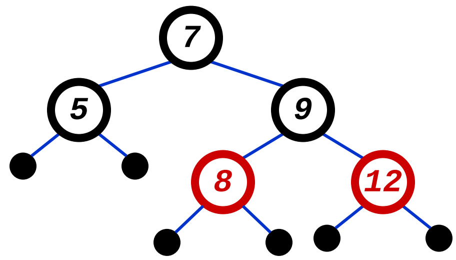
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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

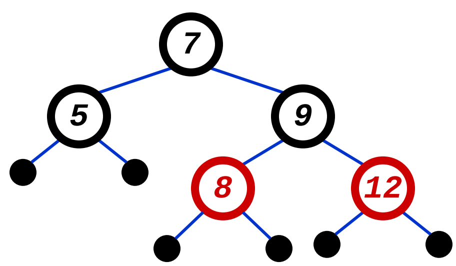
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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

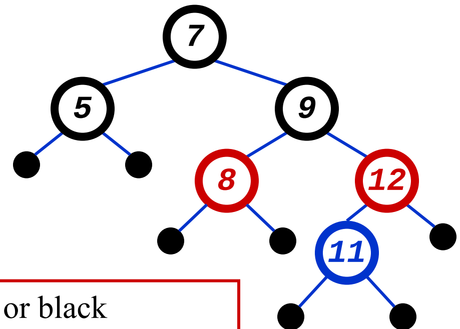
- Insert 11
  - *Where does it go?*



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

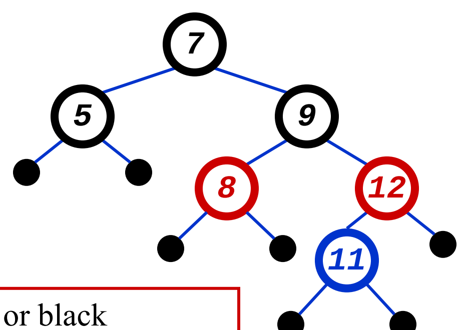
- Insert 11
  - *Where does it go?*
  - *What color?*



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

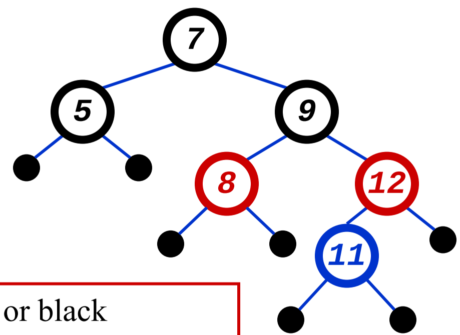
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)



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

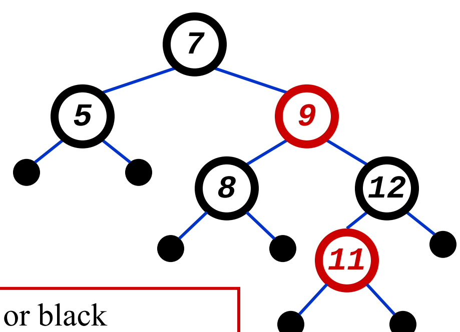
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)
    - Can't be black! (#4)



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

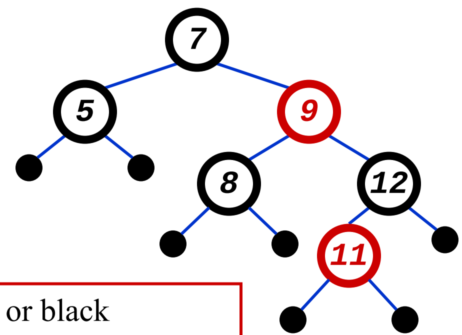
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution: recolor the tree



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

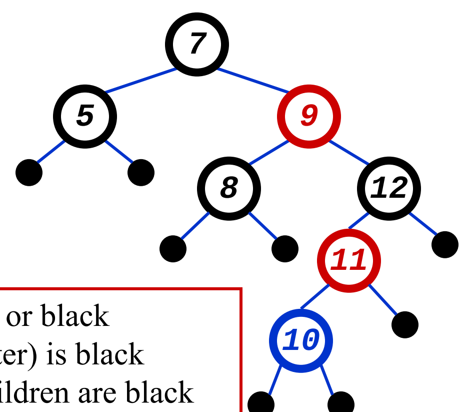
- Insert 10
  - *Where does it go?*



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

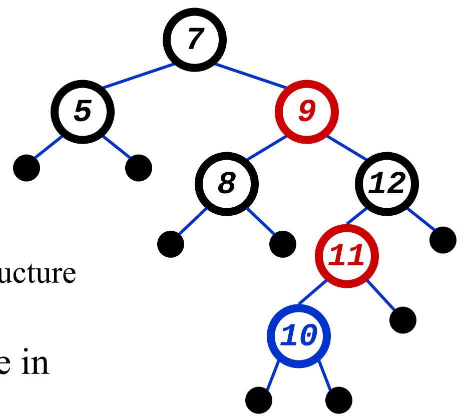
- Insert 10
  - *Where does it go?*
  - *What color?*



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



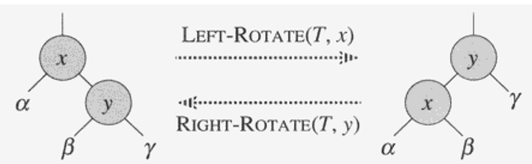
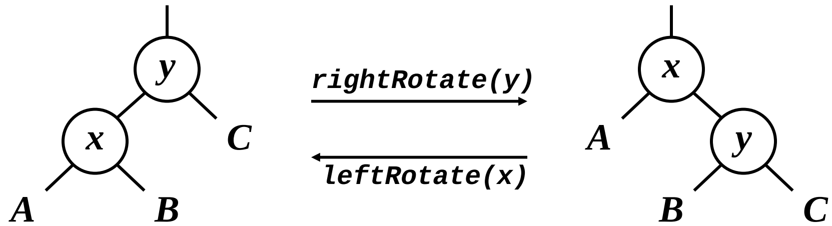


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation RIGHT-ROTATE(T, y). The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $key[x]$, which precedes the keys in β, which precede $key[y]$, which precedes the keys in γ.

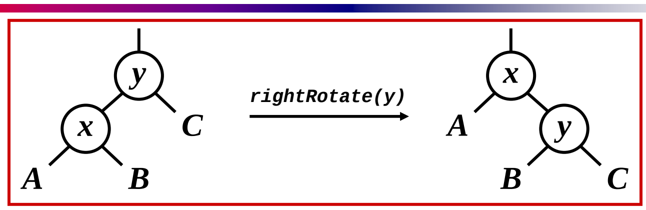
RB Trees: Rotation

- 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?*

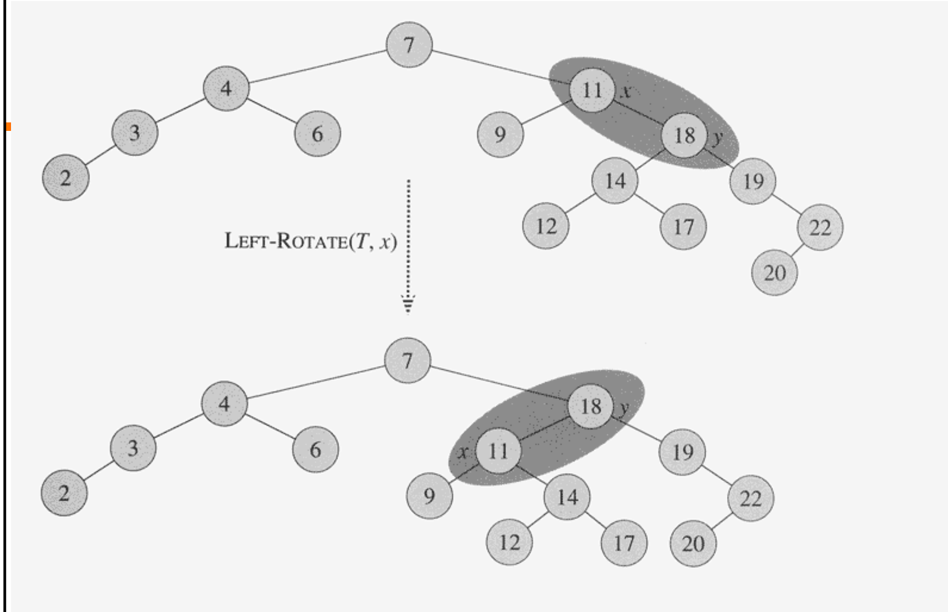
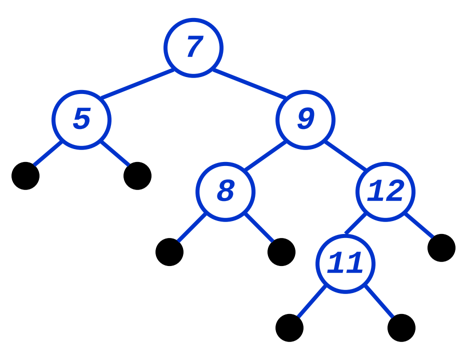


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

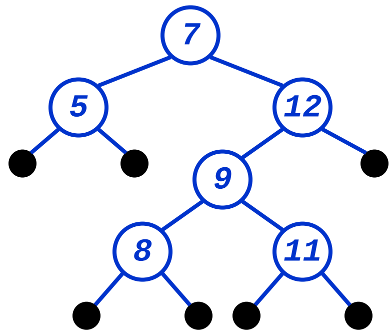
Rotation Example

- 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*)

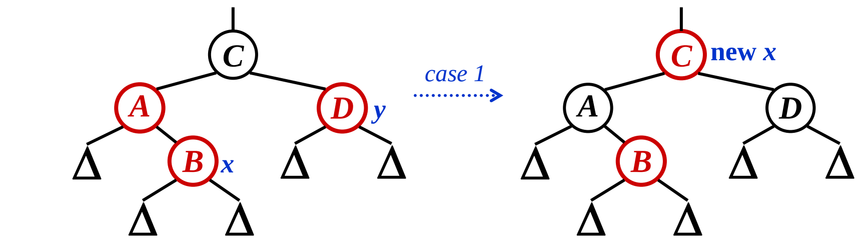
```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #3 up tree, maintaining #4 as invariant:
  while (x!=root && x->p->color == RED)
  if (x->p == x->p->p->left)
      y = x->p->p->right;
      if (y->color == RED)
          x->p->color = BLACK;
          y->color = BLACK;              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

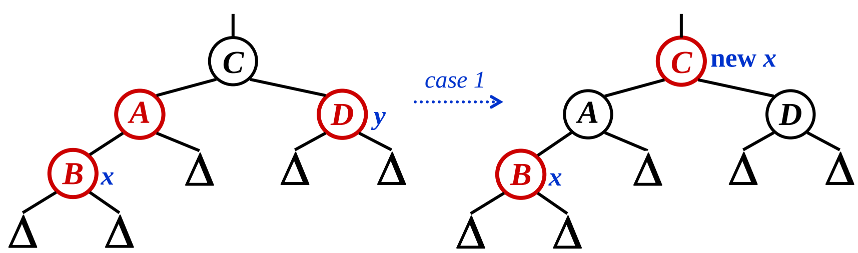


Change colors of some nodes, preserving #4: all downward paths have equal b.h.
The while loop now continues with x’s grandparent as the new x

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

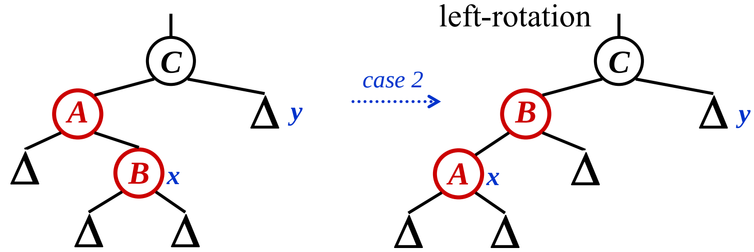


Same action whether x is a left or a right child

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

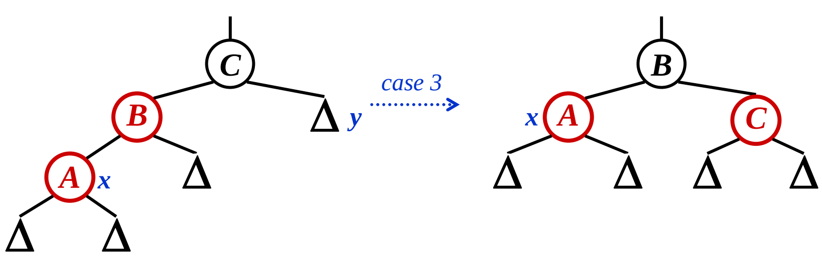


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

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

Red-Black Trees

- Red-black trees do what they do very well
- *What do you think is the worst thing about red-black trees?*
- A: coding them up

Skip Lists

Skip Lists

- A relatively recent data structure
  - "A probabilistic alternative to balanced trees"
  - A randomized algorithm with benefits of r-b trees
    - O(lg *n*) expected time for Search, Insert
    - O(1) time for Min, Max, Succ, Pred
  - *Much* easier to code than r-b trees
  - Fast!

The End