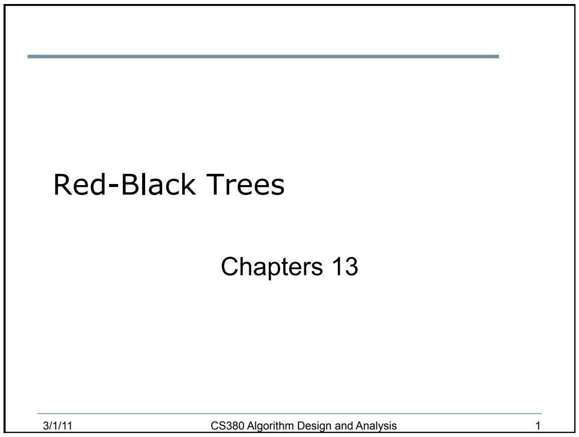
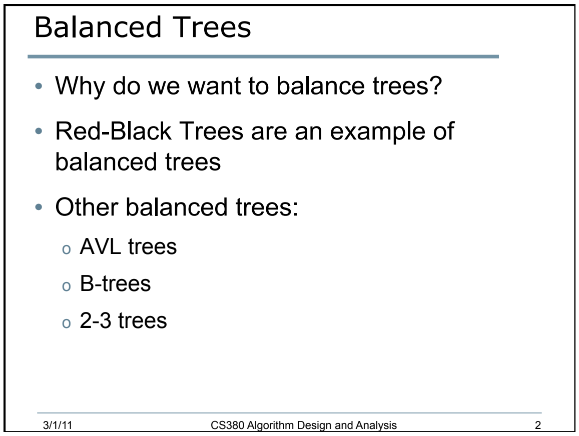


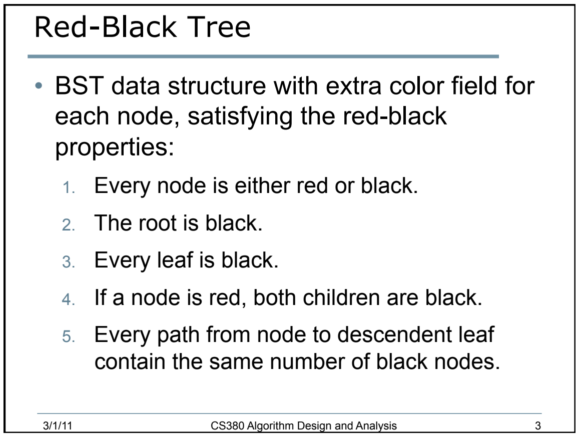
Red-Black Trees

Chapters 13



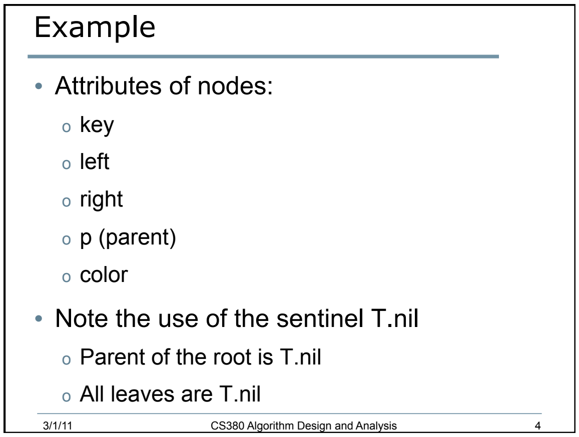
Balanced Trees

- Why do we want to balance trees?
- Red-Black Trees are an example of balanced trees
- Other balanced trees:
  - AVL trees
  - B-trees
  - 2-3 trees



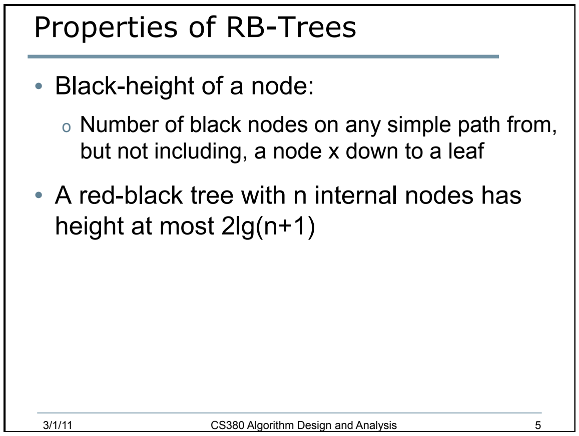
Red-Black Tree

- BST data structure with extra color field for each node, satisfying the red-black properties:
  1. Every node is either red or black.
  2. The root is black.
  3. Every leaf is black.
  4. If a node is red, both children are black.
  5. Every path from node to descendent leaf contain the same number of black nodes.



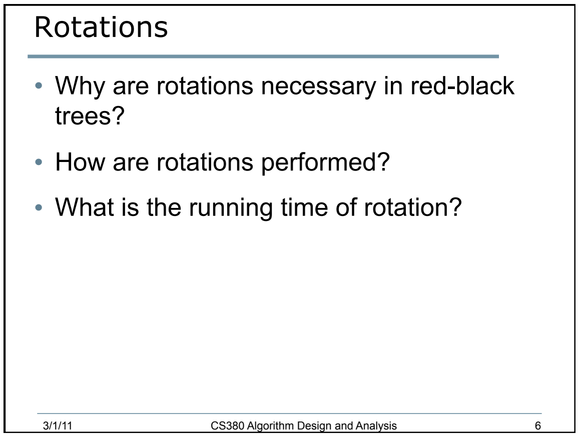
Example

- Attributes of nodes:
  - key
  - left
  - right
  - p (parent)
  - color
- Note the use of the sentinel T.nil
  - Parent of the root is T.nil
  - All leaves are T.nil



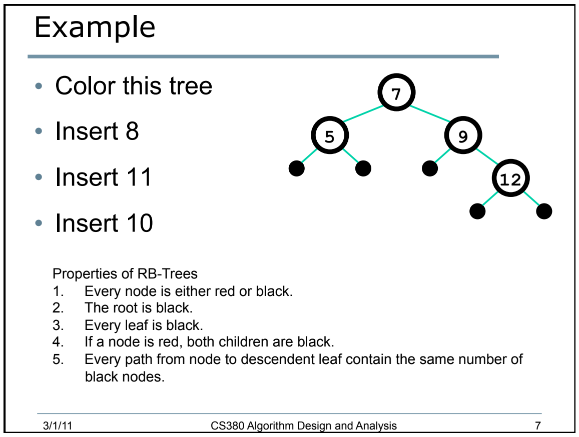
Properties of RB-Trees

- Black-height of a node:
  - Number of black nodes on any simple path from, but not including, a node x down to a leaf
- A red-black tree with n internal nodes has height at most 2lg(n+1)



Rotations

- Why are rotations necessary in red-black trees?
- How are rotations performed?
- What is the running time of rotation?

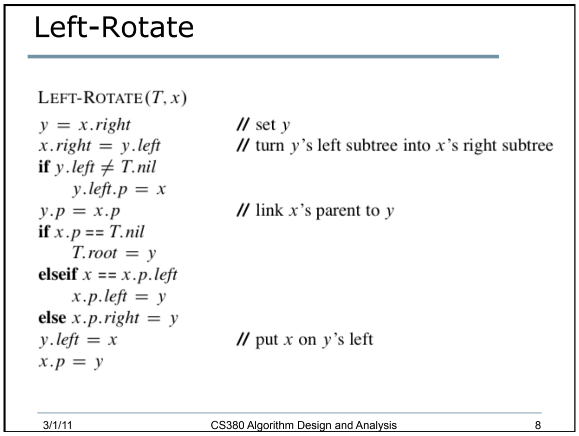


Example

- Color this tree
- Insert 8
- Insert 11
- Insert 10

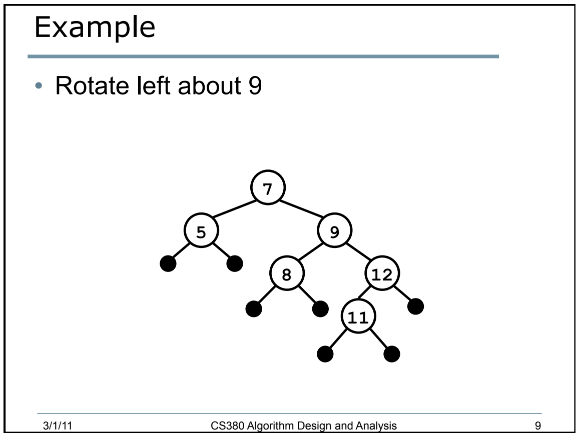
Properties of RB-Trees

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



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 $y.left \neq T.nil$
    $y.left.p = x$
$y.p = x.p$                    // 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
$x.p = y$
```



Example

- Rotate left about 9

Inserting into a RB-Tree

- This is regular binary search tree insertion
- Which RB-Tree property could have been violated?

Properties of RB-Trees

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

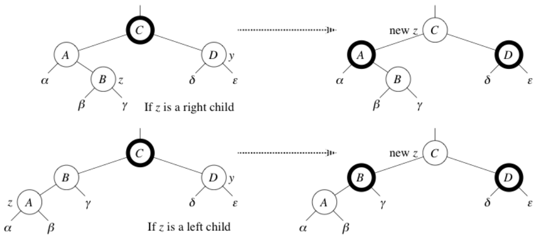
```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

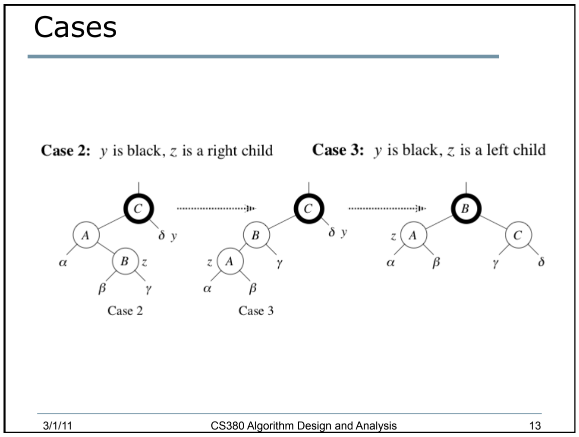
RB-Insert-Fixup

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                  // case 1
            y.color = BLACK                    // case 1
            z.p.p.color = RED                  // case 1
            z = z.p.p                          // case 1
        else if z == z.p.right
                z = z.p                        // case 2
                LEFT-ROTATE(T, z)              // case 2
            z.p.color = BLACK                  // case 3
            z.p.p.color = RED                  // case 3
            RIGHT-ROTATE(T, z.p.p)             // case 3
    else (same as then clause with "right" and "left" exchanged)
T.root.color = BLACK
```

Cases

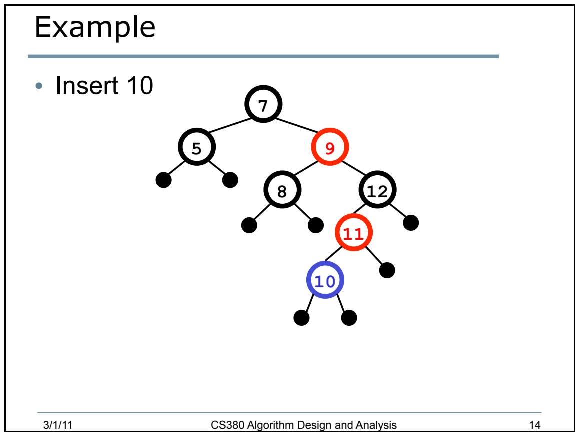
Case 1: *y* is red





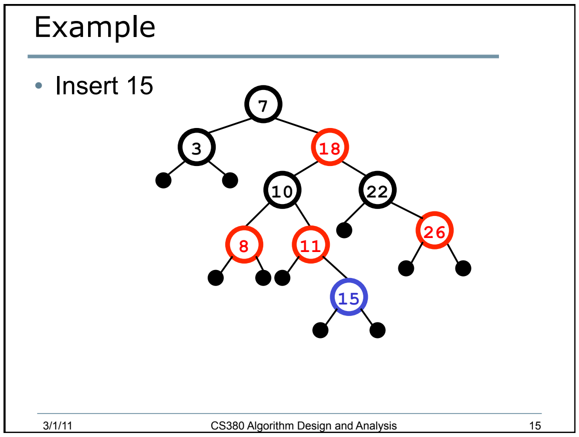
Cases

Case 2: *y* is black, *z* is a right child **Case 3:** *y* is black, *z* is a left child



Example

- Insert 10



Example

- Insert 15