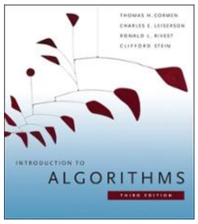
CS146 Data Structures and Algorithms



Chapter 13: Red-Black Trees

Slides made by Dr. Mike Wu of SJSU and some adopted from other universities such as Stanford and U of NC at Chapel Hills.

Balanced search Trees

- Many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take O(lg *n*) time in the worse case.

e.g. AVL trees
Red-black trees

Red-Black Trees

- Red-black trees are a variation of binary search trees to ensure that the tree is ***balanced***.
  - Height is *O*(lg *n*), where *n* is the number of nodes.
- To guarantee that operations take *O*(lg *n*) time in the worst case.

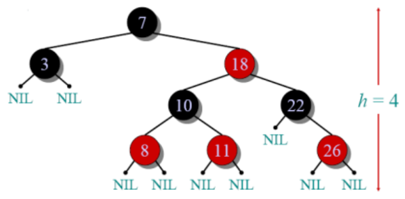
Red-Black Tree

- Binary search tree with + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - *key*, *left*, *right*, and *p*. (These are required for your PA II)
- All empty trees (leaves) are colored **black**.
  - We use a single sentinel, *nil,* for all the leaves of red-black tree *T*, with *nil.color* = black.
  - The root's parent is also *T.nil*.

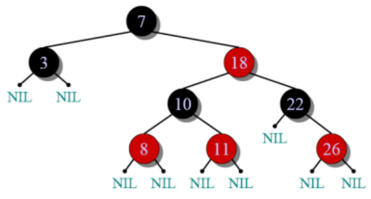
Red-black Properties

1. Every node is either red or **black**.
2. The root is **black**.
3. Every leaf (*nil*) is **black**.
4. If a node is red, then both its children are **black**.
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes. (i.e. black-height(*x*)).

Example of a Red-black Tree

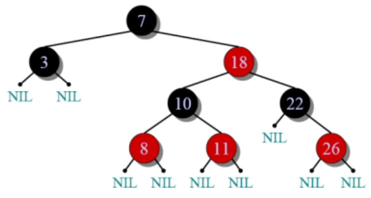


Example of a Red-black Tree



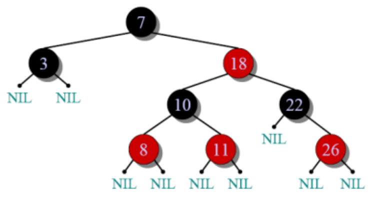
1. Every node is either red or **black**.

Example of a Red-black Tree



2. 3. The root and leaves (NIL's) are **black**.

Example of a Red-black Tree

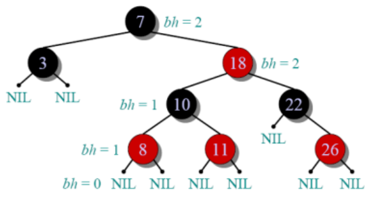


4. If a node is red, then its children are **black**.

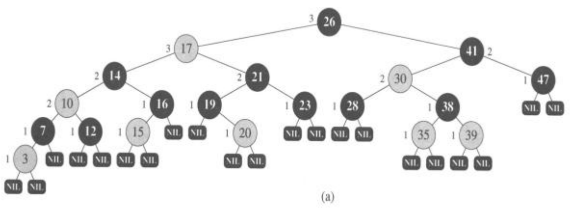
?? Means

If a node is red, then its parent is **black**.

Example of a Red-black Tree



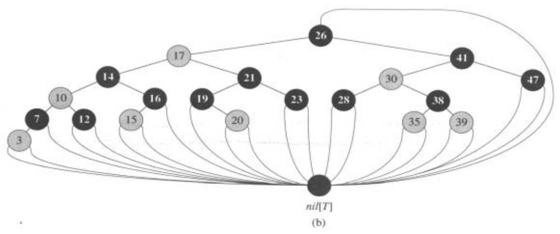
5. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).



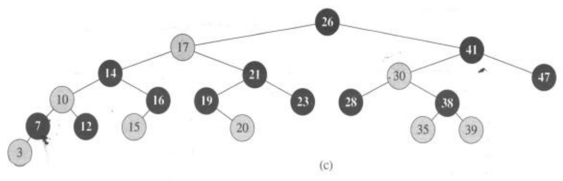
black-height of the node: bh(x)

# of black nodes on any path from, but not including, a node x down to a leaf

black-height of a RB tree = black-height of its root



Leaves and the root's parent omitted entirely



We omit the leaves when we draw BR trees, because we generally confine our interest to the internal nodes, since they hold the key.

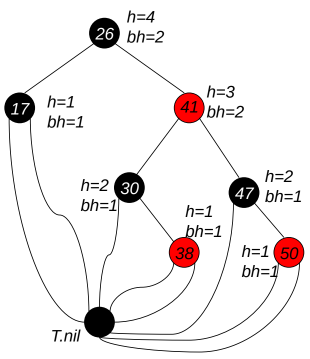
Height of a Red-black Tree

- Height of a node:
  - $h(x)$ = number of edges in a longest path to a leaf.
- Black-height of a node x, $bh(x)$:
  - $bh(x)$ = number of black nodes (including *T.nil*) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.
  - By Property 5, black height is well defined.

Height of a Red-black Tree

- Example:
- Height of a node:

  $h(x)$ = # of edges in a longest path to a leaf.
- Black-height of a node $bh(x)$ = # of black nodes on path from x to leaf, not counting x.
- How are they related?
  - **$bh(x) \le h(x) \le 2\ bh(x)$**



Lemma "RB Height"

Consider a node x in an RB tree: The longest descending path from x to a leaf has length $h(x)$, which is at most twice the length of the shortest descending path from x to a leaf.

Proof:

# black nodes on any path from $x = bh(x)$ (prop 5)

$\le$ # nodes on shortest path from x, $s(x)$. (prop 1)

But, there are no consecutive red (prop 4),

and we end with black (prop 3), so $h(x) \le 2\ bh(x)$.

Thus, $h(x) \le 2\ s(x)$. QED

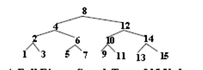
Bound on RB Tree Height

- **Lemma:** The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.
- **Proof:** By induction on height of x.
  - **Base Case:** Height $h(x) = 0 \Rightarrow x$ is a leaf $\Rightarrow$ bh(x) = 0. Subtree has $2^0-1 = 0$ nodes. √
  - **Induction Step:** Height $h(x) = h > 0$ and $bh(x) = b$.
    - Each child of x has height $h - 1$ and black-height either b (child is red) or $b - 1$ (child is **black**).
    - By ind. hyp., each child has $\geq 2^{bh(x)-1}-1$ internal nodes.
    - Subtree rooted at x has $\geq 2\,(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. (The + 1 is for x itself.)

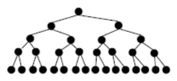
Bound on RB Tree Height

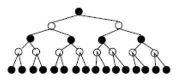
- Lemma: The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.
- **Lemma 13.1:** A red-black tree with n internal nodes has height at most $2\lg(n+1)$.
- **Proof:**
  - By the above lemma, $n \geq 2^{bh} - 1$,
  - and since $bh \geq h/2$, we have $n \geq 2^{h/2} - 1$.
  - $\Rightarrow h \leq 2\lg(n + 1)$.

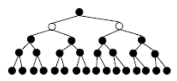
Examples of RB Trees based on full BST of 15 nodes

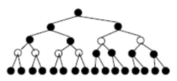


A Full Binary Search Tree of 15 Nodes


NULLs Shown as Nodes, all nodes black
Black height of tree = 4


NULLs Shown as Nodes, black height = 2


NULLs Shown as Nodes, black height = 3

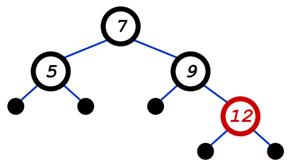

NULLs Shown as Nodes, black height = 3

Operations on RB Trees

- All operations can be performed in ***O*(lg *n*)** time.
- The query operations, which don't modify the tree, are performed in exactly the same way as they are in BSTs.
- These operations take O(lg n) time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- Insert() and Delete():
  - Will also take O(lg n) time
- Insertion and Deletion are not straightforward. Why?

Red-Black Trees: An Example

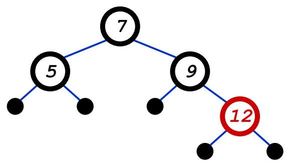
- *Color this tree:*



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

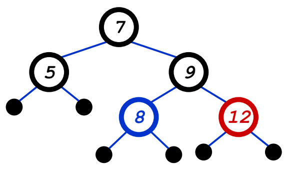
- Insert 8
  - *Where does it go?*



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

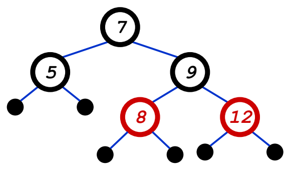
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

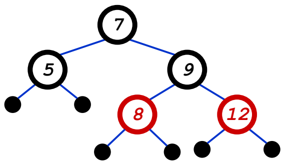
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

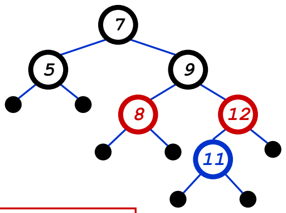
- Insert 11
  - *Where does it go?*



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

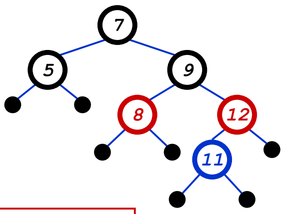
- Insert 11
  - *Where does it go?*
  - *What color?*



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

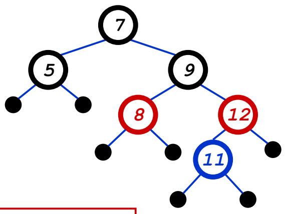
- Insert 11
  - *Where does it go?*
  - *What color?*
    - o Can't be red! (#4)



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

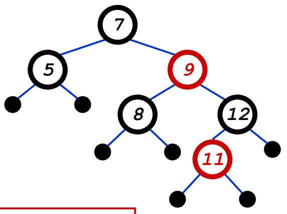
- Insert 11
  - *Where does it go?*
  - *What color?*
    - o Can't be red! (#4)
    - o Can't be black! (#5)



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

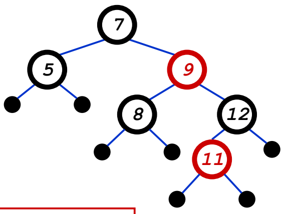
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution: recolor the tree



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

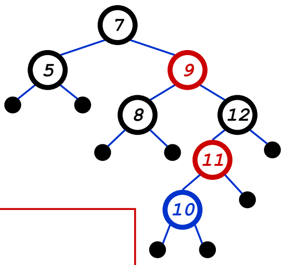
- Insert 10
  - *Where does it go?*



Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL 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

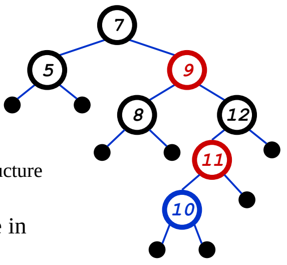
- Insert 10
  - *Where does it go?*
  - *What color?*



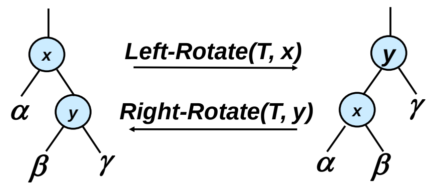
Red-black properties:
1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL pointer) is black
4. If a node is red, both children are black
5. Every path from node to descendent leaf contains the same number of black nodes

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

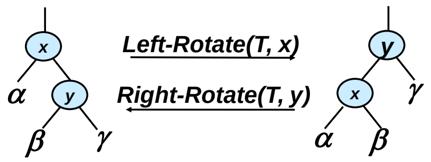


RB Trees: Rotation



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

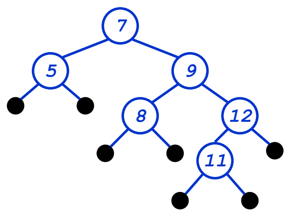
RB Trees: Rotation



- 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?*

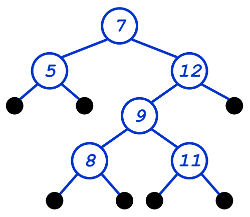
Rotation Example

- Rotate left about 9:



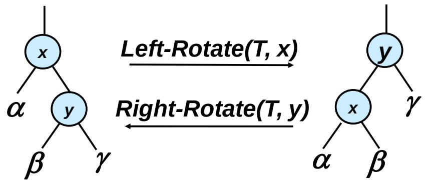
Rotation Example

- Rotate left about 9:



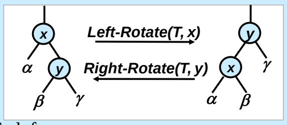
RB Trees: Rotation

- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and
- Won't violate the binary-search-tree property.
- Left rotation and right rotation are inverses.



Left Rotation – Pseudo-code

```
Left-Rotate (T, x)
1.  y = x.right        // Set y.
2.  x.right = y.left   //Turn y's left subtree into x's right subtree.
3.  if y.left ≠ T.nil
4.      then y.left.p = x
5.  y.p = x.p          // Link x's parent to y.
6.  if x.p == T.nil
7.      then T.root = y
8.  else if x == x.p.left
9.          then x.p.left = y
10. else x.p.right = y
11. y.left = x         // Put x on y's left.
12. x.p = y
```



Rotation

- The pseudo-code for Left-Rotate assumes that
  - *X.right* ≠ *T.nil*, and
  - root's parent is *T.nil*.
- Left Rotation on *x*, makes *x* the left child of *y*, and the left subtree of *y* into the right subtree of *x*.
- Pseudocode for Right-Rotate is symmetric: exchange *left* and *right* everywhere.
- ***Time:*** *O*(1) for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.