Red-Black Trees

Antonio Carzaniga

Faculty of Informatics
University of Lugano

October 31, 2008

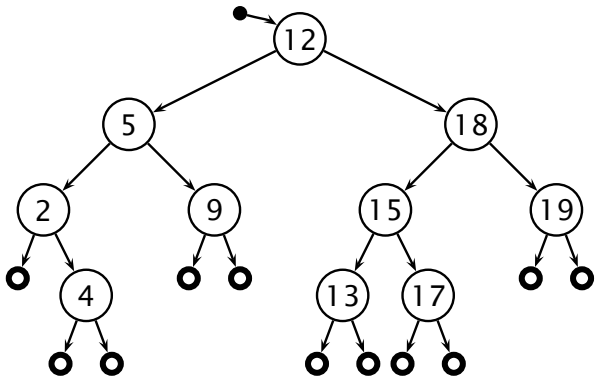
Outline

- Red-black trees

Summary on Binary Search Trees

- Binary search trees
  - data structure embodying the *divide-and-conquer* search strategy
  - Search, Insert, Min, and Max operations are $O(h)$, where h is the *height of the tree*
  - in general, $h(N) = \Omega(\log N)$ and $h(N) = O(N)$
  - *randomization* can be used to make the worst-case scenario $h(N) = N$ highly unlikely
- Problem
  - worst-case scenario is unlikely but still possible
  - simply bad cases are even more probable

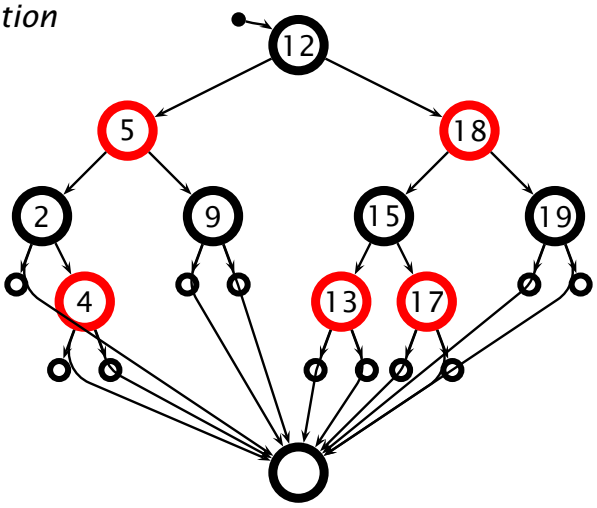
Red-Black Tree



- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every node x, each path from x to its descendant leaves has the same number of **black** nodes $bh(x)$ (the *black-height* of x)

Red-Black Tree (2)

- *Implementation*



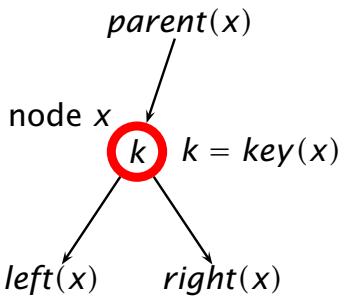
  - we use a common "sentinel" node to represent leaf nodes
  - the sentinel is also the parent of the root node

Red-Black Tree (3)

- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $root(T)$ is the root node of tree T
  - $nil(T)$ is the "sentinel" node of tree T

Nodes

- $parent(x)$ is the parent of node x
- $key(x)$ is the key stored in node x
- $left(x)$ is the left child of node x
- $right(x)$ is the right child of node x
- $color(x) \in \{\text{red}, \text{black}\}$ is the color of node x



Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $N = size(x)$ internal nodes is at most $2\log(N+1)$.

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:

   1.1 *base case:* x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

   1.2 *induction step:* consider y_1, y_2, and x such that $parent(y_1) = parent(y_2) = x$, and assume (induction) that $size(y_1) \geq 2^{bh(y_1)} - 1$ and $size(y_2) \geq 2^{bh(y_2)} - 1$;
   prove that $size(x) \geq 2^{bh(x)} - 1$
   proof:
   $size(x) = size(y_1) + size(y_2) + 1 \geq (2^{bh(y_1)} - 1) + (2^{bh(y_2)} - 1) + 1$
   since

$$bh(x) = \begin{cases} bh(y) & \text{if } color(y) = \text{red} \\ bh(y) + 1 & \text{if } color(y) = \text{black} \end{cases}$$

   $size(x) \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$

Height of a Red-Black Tree (2)

1. $size(x) \geq 2^{bh(x)} - 1$
2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $bh(x) \geq h(x)/2$
3. From steps 1 and 2, $N = size(x) \geq 2^{h(x)/2} - 1$, therefore

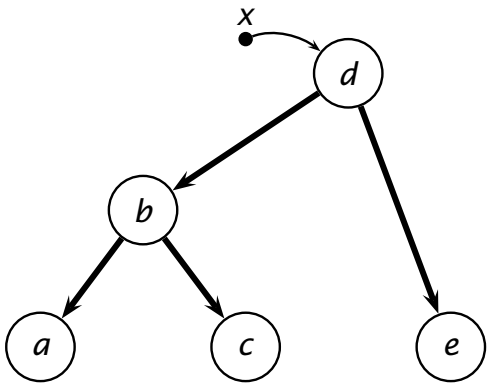
$$h \leq 2\log(N+1)$$

- A red-black tree works as a binary search tree for search, etc.
- Therefore, the complexity of those operations is $T(N) = O(h)$, that is

$$\boxed{T(N) = O(\log N)}$$

  - which is also the *worst-case* complexity

Rotation



- $x \leftarrow$ Right-Rotate(x)
- $x \leftarrow$ Left-Rotate(x)

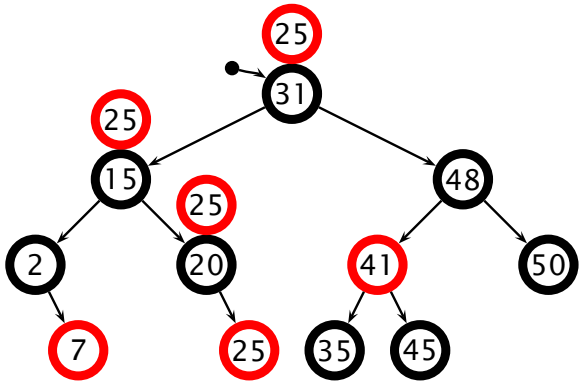
Red-Black Insertion

- RB-Insert(T, z) works as in a binary search tree
- Except that it must preserve the *red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every node x, each path from x to its descendant leaves has the same number of **black** nodes $bh(x)$ (the *black-height* of x)
- *General strategy*
  1. insert z as in a binary search tree
  2. color z **red** so as to preserve property 5
  3. *fix the tree* to correct possible violations of property 4

RB-Insert

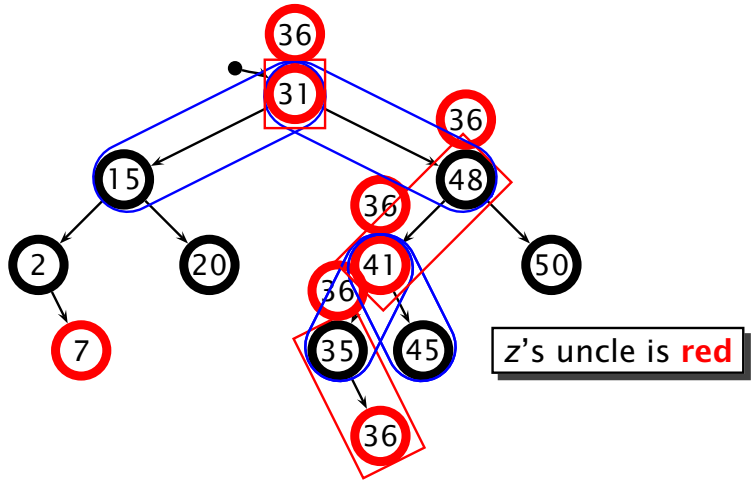
```
RB-Insert(T, z)
 1  y ← nil(T)
 2  x ← root(T)
 3  while x ≠ nil(T)
 4        do y ← x
 5           if key(z) < key(x)
 6              then x ← left(x)
 7              else  x ← right(x)
 8  parent(z) ← y
 9  if y = nil(T)
10     then root(T) ← z
11     else  if key(z) < key(y)
12              then left(y) ← z
13              else  left(y) ← z
14  left(z) ← right(z) ← nil(T)
15  color(z) ← red
16  RB-Insert-Fixup(T, z)
```


Red-Black Insertion (2)



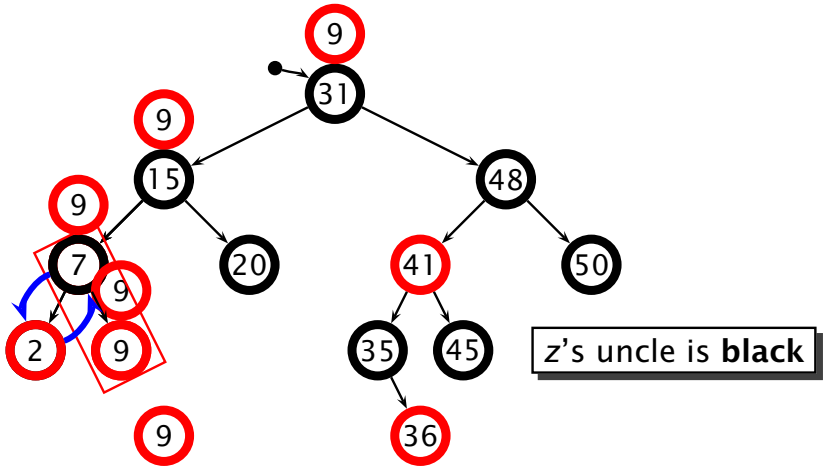
- z's father is **black**, so no fixup needed

Red-Black Insertion (3)



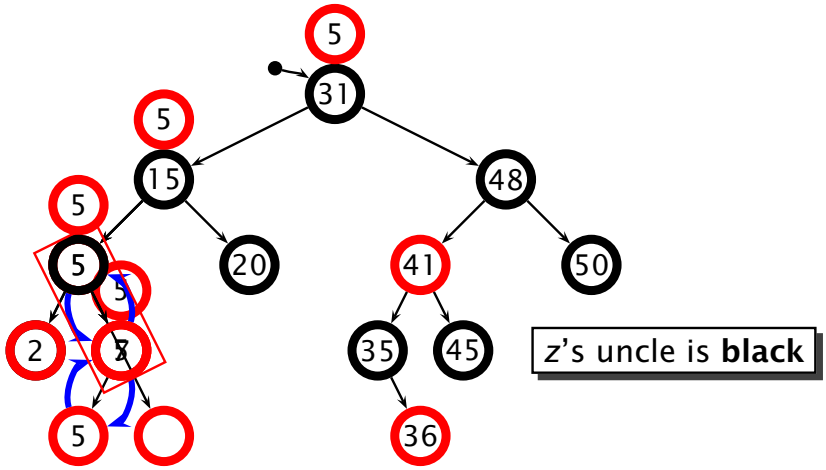
- A **black** node can become **red** and transfer its **black** color to its two children
- This may cause other **red**–**red** conflicts, so we iterate. . .
- The root can change to **black** without causing conflicts

Red-Black Insertion (4)



- An *in-line* ***red***–**red** conflicts can be resolved with a rotation plus a color switch

Red-Black Insertion (5)



- A *zig-zag* **red**–**red** conflicts can be resolved with a rotation to turn it into an *in-line* conflict, and then a rotation plus a color switch